Improving the Hybrid model MPI+Threads through Applications, Runtimes and Performance tools

Aurèle Maheo

Thesis submitted to obtain the grade of
Doctor in Philosophy of
Université de Versailles-Saint-Quentin-en-Yvelines

École doctorale de Sciences et Technologie de Versailles
Specialized in **Computer Sciences**

By **Aurèle MAHEO**

Improving the Hybrid model MPI+Threads through Applications, Runtimes and Performance tools

---

Hosted by
Exascale Computing Research, Versailles, France

Defended on 25 of September 2015 in front of the following doctoral committee :

Pr. William JALBY	Professor at the University of Versailles	PhD Advisor
Pr. Raymond NAMYST	Professor at the University of Bordeaux	Referee
Pr. Allen MALONY	Professor at the University of Oregon	Referee
Pr. Gaël THOMAS	Professor at Télécom SudParis	Examiner
Pr. Michael KRAJECKI	Professor at the University of Reims	Examiner
Dr. Patrick CARRIBAULT	Research Engineer, CEA, DAM	Examiner
Dr. Marc PÉRACHE	Research Engineer, CEA, DAM	Examiner

Thèse présentée
pour obtenir le grade de
Docteur de
l'Université de Versailles-Saint-Quentin-en-Yvelines

École doctorale de Sciences et Technologie de Versailles
Spécialité : **Informatique**

Présentée par **Aurèle MAHEO**

Amélioration du modèle hybride MPI+Threads à travers les applications, les supports d'exécution et outils d'analyse de performance

---

Organisme d'accueil:
Exascale Computing Research, Versailles, France

Soutenue le 25 Septembre 2015 devant la commission d'examen composée de :

Pr. William JALBY	Professeur à l'université de Versailles	Directeur de thèse
Pr. Raymond NAMYST	Professeur à l'université de Bordeaux	Rapporteur
Pr. Allen MALONY	Professeur à l'université de l'Oregon	Rapporteur
Pr. Gaël THOMAS	Professeur à Télécom SudParis	Examinateur
Pr. Michael KRAJECKI	Professeur à l'Université de Reims	Examinateur
Dr. Patrick CARRIBAULT	Ingénieur de recherche, CEA,DAM	Examinateur
Dr. Marc PÉRACHE	Ingénieur de recherche, CEA,DAM	Examinateur

Acknowledgements / Remerciements

Au moment de terminer ce manuscrit, il me semblait opportun de jeter un coup d'oeil dans le rétroviseur. Je n'ai pas accompli cette thèse seul et beaucoup de personnes de valeur ont apporté leur concours à sa réalisation.

Je voudrais tout d'abord remercier M. Allen Malony et M. Raymond Namyst, en tant que rapporteurs de ma thèse et d'avoir accepté de juger mes travaux. Merci également à M. Gaël Thomas et M. Michael Krajecki d'avoir bien voulu être mes examinateurs.

Je voudrais ensuite exprimer ma gratitude au Professeur William Jalby pour avoir accepté d'être mon directeur de thèse.

Mes pensées vont ensuite à mes tuteurs Marc Pérache et Patrick Carribault qui m'ont accompagné pendant ces trois (ou quatre !) années. Je voudrais tout spécialement saluer Patrick qui m'a réellement impressionné, tout d'abord en tant qu'enseignant pour ses qualités pédagogiques, puis comme encadrant pour sa rigueur intellectuelle, qui a su m'inculquer ce fameux "esprit recherche". Je remercie chaleureusement Marc pour son encadrement également et ses qualités humaines. Merci à eux deux d'avoir su être à l'écoute quand je me suis senti dans l'impasse. Je m'estime chanceux d'avoir bénéficié de leur encadrement.

I would like to thank Dr Sameer Shende for welcoming me in University or Oregon. I'm very excited to continue this collaboration !

Je tiens ensuite à saluer l'équipe MPC, à commencer tout d'abord par les "anciens". Merci à Jean-Yves pour sa bonne humeur, merci de m'avoir légué l'appartement, perpétuant ainsi une tradition consistant à garder le bien au sein de l'équipe MPC. Je crains malheureusement - et ce n'est pas faute d'avoir essayé ! - qu'il ne restera pas dans l'équipe. Merci à Jean-Baptiste pour ses encouragements, à Sébastien, et à Emmanuelle qui devrait également tenter l'aventure transatlantique, Camille et Antoine, et sans oublier Jérôme, le Bordelais spécialiste des tâches.

Salut à mes camarades du mihps et tout spécialement aux motivés (Nicolas, Zakaria, Caner) qui ont décidé que 5 ans d'étude, c'était pas encore assez. Rappelons tout de même que nous fûmes les patients zéro de ce beau Master, soyons fiers !

Je n'oublie pas les autres exascaliens, Franck, Augustin mon voisin du 2.2, Vincent, Michel et l'équipe MAQAO.

J'adresse un salut particulier à Andres qui n'est certainement pas pour rien dans le choix de cette aventure !

Je salue également les autres collègues ou compagnons d'infortune de thèse (hum) que j'ai pu rencontrer lors de conférences, doctoriales ou autres. Il y eut de belles rencontres, des échanges intéressants et des occasions de sortir un peu de mon quotidien.

Et pour sortir un peu du cadre de la thèse, je n'oublie mes amis et proches qui ont réussi m'arracher les yeux de l'écran (pas facile). Salut aux collègues plongeurs, c'est toujours agréable de pouvoir faire quelques bulles entre deux chapitres de manuscrit ! Je pense tout d'abord à Mathieu et Pierre, merci pour ces cavales à droite et gauche, les balades á vélo (le bonheur est simple). Merci à celles et ceux qui, sans être présents physiquement, n'en sont pas moins précieux pour moi. Clin d'oeil à mon vieux camarade Samir, occupé á parcourir l'Outback australien puis la terre du milieu, mais jamais loin par la pensée (et par Messenger :)) ! Je lui souhaite de tout coeur de trouver son petit coin de paradis. Peut-être aura-t-on l'occasion d'explorer des territoires vierges et de palmer un peu parmi les coraux ? Enfin, *no worries !*

Merci à Anaïs, une londonienne qui compte beaucoup pour moi, et avec qui j'ai toujours grand plaisir á échanger. Merci pour son esprit positif et sa gentillesse.

Je tiens à dire à Timothée qu'il a été un ami très précieux tout au long de ce parcours, pour son soutien sans failles. Je crois que nous avons tous deux parfaitement intégré le concept de "reprise d'études". Je n'oublierai ni sa présence ni son amitié. J'espère le convaincre un jour d'aller chausser des skis dans les Rocheuses !

Mes dernières pensées vont mes proches, spécialement mes neveux et nièces, que j'ai toujours plaisir à retrouver ! Je voudrais terminer par rendre hommage à mes parents pour leur soutien inconditionnel et pour avoir su me donner de par leur exemple l'envie nécessaire de me dépasser et d'aller aussi loin que possible.

Contents

List of Figures

Chapter 1

Résumé en français

Cette partie présente un résumé en français de la thèse, et en reprend les éléments clés. Le lecteur voulant approfondir un sujet particulier est invité à consulter les chapitres correspondants.

1.1 Contexte

1.1.1 Introduction

Le domaine du calcul haute performance s'adresse aux machines massivement parallèles utilisées pour les applications ayant trait à la simulation numérique. Cette discipline et les moyens de calcul associés sont apparus lors de la Seconde Guerre Mondiale, avec notamment le fameux projet Manhattan, dans le but de concevoir la première bombe atomique. De nos jours, de nombreux projets d'envergure sont menés à l'aide de la simulation numérique, dont voici quelques exemples:

- Dans le domaine de l'astrophysique, DEUS (Dark Energy Universe Simulation) vise à estimer l'impact de l'énergie noire dans l'univers.
- Le projet The Human Brain Project est une initiative européenne visant à simuler le cerveau humain, en modélisant ses processus biologiques.

Cependant, ces applications requièrent des capacités de calcul conséquentes. C'est ainsi que les supercalculateurs ont émergé dans le but d'aider les scientifiques à mener à bien leurs simulations. Les premiers supercalculateurs sont apparus dans les années 1960, avec le CDC-6600. Dans les années 1970, Seymour Cray fonde Cray Research et une nouvelle famille d'ordinateurs parallèles, en premier lieu le Cray-1. La puissance d'un superordinateur est estimé par le nombre de calculs en virgule flottante qu'il est capable d'effectuer par seconde (FLOP). Si les premières machines parallèles développaient seulement quelques mégaflops, en 2001, la machine Earth Simulator affichait 40 téraflops. Aujourd'hui, les supercalculateurs sont composés d'armoires reliées en réseau rapide, et sont capables de développer une puissance de calcul de l'ordre du pétaflop (10^{15} opérations par seconde). Cependant, la communauté scientifique a aujourd'hui les yeux rivés sur l'Exascale (10^{18} opérations par seconde).

Les processeurs actuels sont équipés de plusieurs coeurs de calcul, obligeant les programmeurs à rendre leurs codes parallèles. De plus, les processeurs graphiques (GPUs), à l'origine destinés à l'interfaçage entre le système et l'utilisateur, puis dédiés au rendu graphique dans le contexte des jeux vidéos, sont désormais utilisés pour effectuer du calcul. Ces tendances ont influencé l'évolution des supercalculateurs. Ainsi, originellement composés de systèmes à mémoire distribuée, les machines actuelles intègrent de plus en plus d'unités de calcul à l'intérieur d'un noeud physique.

Les architectures parallèles se sont également complexifiées: dans les systèmes dits UMA (Uniform Memory Access), les unités de calcul partagent un même bus pour accéder à la mémoire centrale. Mais

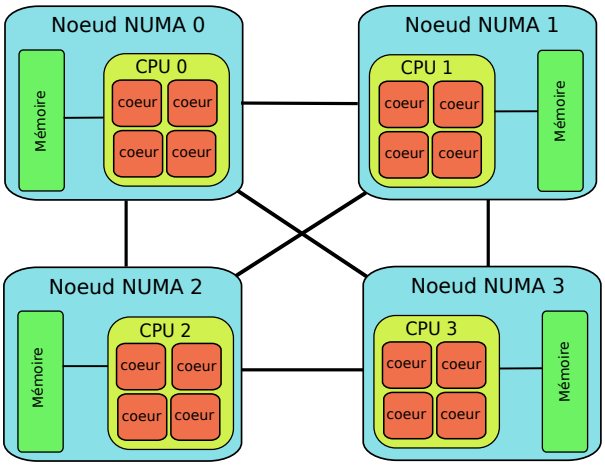


Figure 1.1: Architectures ccNUMA

la multiplication des coeurs a conduit à saturer la bande passante disponible. Ceci a obligé les constructeurs à concevoir des systèmes plus complexes, tels les architectures NUMA (figure 1.1). Dans ce type d'architectures, on définit comme bloc de base l'ensemble constitué d'un processeur relié à un banc mémoire local: on appelle cet ensemble un noeud NUMA. La localité des données constitue une caractéristique importante pour ce type d'architecture, et peut fortement influencer les performances des applications. En effet, la latence d'accès à une donnée peut être très variable selon que cette donnée se situe dans un banc mémoire local ou alors dans un banc mémoire appartenant à un noeud NUMA distant. Ce phénomène est appelé l'effet NUMA.

Pour programmer ces machines et les exploiter au mieux, plusieurs modèles de programmation ont été conçus, et peuvent être répartis entre plusieurs familles:

- Modèles à passage de messages, tel que MPI (Message Passing Interface)
- PGAS (Partitioned Global Address Space)
- Modèles à base de threads: Posix Threads, OpenMP
- Modèles à base de tâches: Cilk

MPI est le modèle principalement utilisé pour paralléliser les applications à travers les clusters de calcul. Cependant, avec l'évolution des supercalculateurs et la diminution de la quantité de mémoire disponible par coeur, la librairie MPI montre de plus en plus de limites concernant le passage à l'échelle des codes de calcul.

C'est pourquoi les scientifiques cherchent désormais à coupler MPI avec un modèle adapté aux machines à mémoire partagée. OpenMP, modèle à base de threads, est un choix privilégié car considéré comme un standard. De plus, il supporte les langages C, C++ et Fortran. Cependant, combiner MPI et OpenMP, modèles munis de paradigmes et de sémantiques différents, est loin d'être une tâche aisée.

1.1.2 Modèle hybride MPI+OpenMP

Nous présentons ainsi une taxonomie décrivant les différentes manières de combiner ces deux modèles, à travers deux principaux aspects: la granularité du code et le placement (figure 1.2).

Au niveau du code, différentes granularités sont possibles s'agissant du mélange de MPI et de OpenMP:

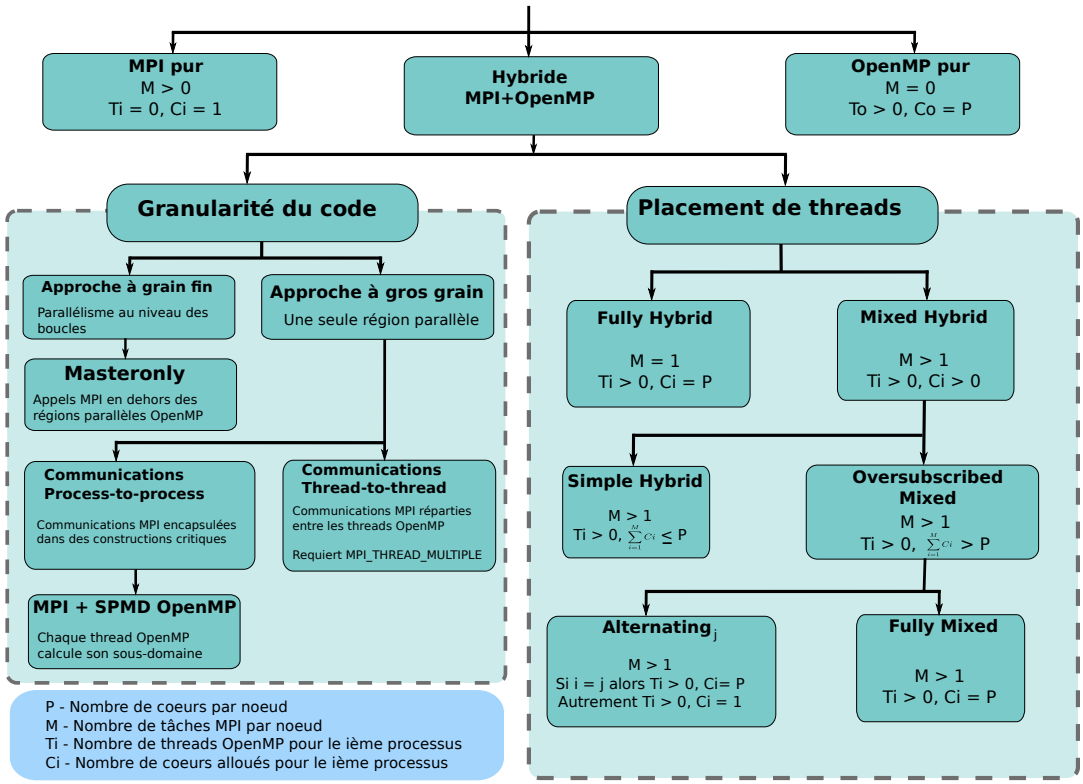


Figure 1.2: Taxonomie hybride MPI+OpenMP

Parallélisme à grain fin. Ce mode consiste, à partir d'un code MPI, à paralléliser des boucles de calcul à l'aide de la construction `#pragma omp parallel for`. On peut décomposer le temps d'exécution d'un code hybride à grain fin à l'aide de l'équation suivante:

$$Ttot = Tseq + Tcomm + \frac{Tcompute}{P} \tag{1.1}$$

où $Ttot$, $Tseq$, $Tcomm$, $Tcompute$ et P sont respectivement le temps total d'exécution, la partie séquentielle, le temps passé dans les communications, le temps passé dans le calcul et le nombre de threads OpenMP.

Plusieurs inconvénients sont à noter concernant cette approche. Tout d'abord, quand des boucles parallèles sont exécutées, deux niveaux de parallélisme sont imbriqués: l'un avec MPI et le second avec OpenMP. Mais en dehors des boucles parallèles, nous perdons un niveau de parallélisme. De plus, les coeurs utilisés par les threads OpenMP deviennent alors inactifs.

Troisièmement, le support OpenMP est susceptible d'être accédé de manière très fréquente dans ce mode, car OpenMP est utilisé uniquement pour paralléliser les boucles, structures potentiellement nombreuses dans le code.

Parallélisme à gros grain. Dans ce mode, on ouvre une ou plusieurs régions parallèles OpenMP. Cette approche est susceptible d'offrir plus de parallélisme avec OpenMP. Cependant, il est possible que ces régions parallèles contiennent des appels à des primitives MPI, ce qui reviendrait alors pour une fonction MPI donnée, à faire un appel par thread OpenMP de chaque tâche MPI. Pour résoudre ce problème, deux variantes du parallélisme à gros grain sont possibles: Process-to-process communications et Thread-to-thread communications. Avec la variante Process-to-process, on s'assure qu'un appel à une fonction MPI donnée est effectuée par tâche MPI à l'aide de constructions critiques telles

que `#pragma omp single` ou `#pragma omp critical`. Dans la variante Thread-to-thread, on parallélise les communications en utilisant plusieurs threads. Et ce faisant, il est alors possible de recouvrir les communications MPI avec du calcul.

Le placement décrit différentes manières de répartir les tâches MPI et les threads OpenMP au sein d'un noeud de calcul. Le premier mode, *Fully hybrid*, utilise MPI pour les communications inter-noeuds et remplit les coeurs du noeud de calcul avec des threads OpenMP. Avec l'approche *Mixed Hybrid*, plusieurs tâches MPI sont lancées par noeud de calcul. Il est possible de surcharger les coeurs avec des tâches MPI ou des threads OpenMP (c'est le mode *Oversubsribed Mixed*).

Suite à l'étude de la taxonomie du modèle hybride MPI+OpenMP, nous présentons différents problèmes freinant le passage à l'échelle des codes MPI+OpenMP:

- Fréquence d'accès à la couche OpenMP dans le mode à grain fin
- Sous-utilisation des coeurs de calcul en dehors des constructions OpenMP, toujours concernant le parallélisme à grain fin
- Complexité des codes MPI+OpenMP dans une approche à gros grain
- Nécessité de pouvoir analyser les performances des codes hybrides pour permettre le passage à l'échelle

1.2 Contributions

1.2.1 Supports exécutifs OpenMP pour les codes MPI+OpenMP

Tout d'abord, considérant le modèle OpenMP comme étant situé en haut de la pile logicielle, il est nécessaire d'avoir de bonnes performances au niveau du support exécutif d'OpenMP. Ainsi, nous nous concentrons sur le surcoût induit par les entrées et sorties dans le moteur d'exécution depuis l'application, à savoir lorsqu'on entre et sort d'une région parallèle, respectivement.

Ceci nous amène à nous intéresser à l'optimisation des support exécutifs OpenMP pour les codes hybrides MPI+OpenMP et en environnement NUMA. Il s'agit d'obtenir un surcoût minimal d'une part dans un contexte où les coeurs de calcul sont partagés entre les tâches MPI et les threads OpenMP, donc l'espace pour chaque équipe OpenMP peut varier. D'autre part, le parallélisme hiérarchique induit par les systèmes NUMA doit être pris en compte, notamment les effets NUMA associés.

Nous pouvons illustrer l'impact des effets NUMA sur le surcoût d'un support exécutif OpenMP en prenant l'exemple de l'activation des threads lors de l'ouverture d'une région parallèle. Dans une implémentation dite centralisée, le thread Master doit réveiller lui-même tous les autres threads. Si le coût pour activer un thread est minime si celui-ci se trouve sur le même socket que le thread Master, il devient important si le thread à réveiller se situe sur un socket distant. Il est donc important d'utiliser des mécanismes adaptés en environnement NUMA.

Cependant, plusieurs contributions ont proposé des optimisations concernant les supports exécutifs OpenMP, suivant plusieurs directions:

- **Implémentation des threads OpenMP**: Pour démarrer leur exécution rapidement ou effectuer un ordonnancement efficace, une implémentation légère des threads peut constituer une piste intéressante. Par exemple, une contribution introduit le concept de microVP, implémenté dans le support MPC. Ces microVPs ordonnancent des threads légers appelés microthreads, utilisant la pile du microVP.
- **Placement hiérarchique des threads**: ForestGOMP propose une représentation hiérarchique des threads, via son outils BubbleSched. Les threads appartenant à une équipe OpenMP sont regroupés au sein d'une même bulle, et la bulle est éclatée suivant la topologie NUMA.

- **Affinité mémoire**: Le placement des threads peut évoluer durant l'exécution d'une application parallèle. Cependant, il est important de s'intéresser aux interactions entre un thread et les données accédées par ce thread (lecture ou écriture). Il peut alors s'avérer pertinent de migrer les données afin de les garder le plus proche possible du thread en question.

Parmi ces différentes pistes, nous nous sommes concentrés sur comment intégrer la topologie NUMA dans la représentation des threads.

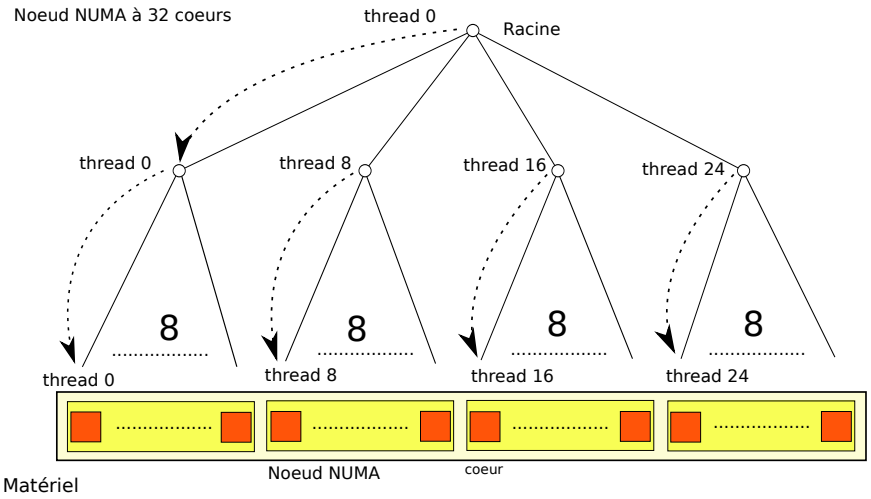


Figure 1.3: Activation hiérarchique des threads

Un travail préliminaire a donc consisté à implémenter une représentation hiérarchique des threads à l'aide d'un arbre équilibré, avec une racine, des noeuds et des feuilles (figure 1.3). Les threads sont répartis sur l'arbre. Cette structure permet au thread Master de déléguer le travail d'activation des threads.

Cependant, pour un nombre réduit de threads OpenMP, l'arbre topologique ne convient peut-être pas, et un arbre plat (avec une racine et une feuille par thread) est sans doute la forme d'arbre la plus adaptée. A contrario, pour un grand nombre de threads, un arbre profond tel un arbre binaire permet de minimiser la contention dûe à l'activation des threads à chaque étage.

Arbre adaptatif

Notre première contribution consiste en un arbre adaptatif, dont la forme change au cours de l'exécution de l'application. Nous construisons un arbre topologique, et l'utilisons en fonction du nombre de threads demandé. Le principe est le suivant: nous utilisons un sous-arbre de l'arbre topologique si le nombre de threads demandés peut être contenu dans ce sous-arbre. Dans ce cas, nous déplaçons la racine initialement placée en haut de l'arbre topologique à la racine du sous-arbre.

Le mécanisme d'arbre adaptatif est transparent pour les implémentations des constructions OpenMP telles que l'activation et la synchronisation des threads. Celles-ci prennent simplement en compte la racine retournée par l'algorithme de court-circuit.

Expérimentations

L'arbre adaptatif a été implémenté au sein de la version 2.5.0 du framework MPC. La forme initiale de l'arbre suit la topologie matérielle, à l'aide de l'outil HWLOC (HardWare LOCality).

Algorithm 1 Bypassing

Require: $tree, numthreads$
1: $node \leftarrow tree.root$
2: **while** $node.typechildren \neq LEAF$ **and** $numthreads \leq node.children[0].maxindex$ **do**
3: $\quad node \leftarrow node.children[0]$
4: **end while**
5: $tree.newroot \leftarrow node$
6: **return** $tree.newroot$

Cette approche a été évaluée sur deux configurations matérielles: 1 noeud composé de 4 processeurs Nehalem EX X7550 à 2 GHz (Tera-100), totalisant 32 coeurs de calcul 1 second noeud Bull Coherency Switch contenant 16 processeurs Intel Xeon E7-4800, avec un total de 128 coeurs de calcul.

Nous avons comparé le surcoût du support OpenMP de MPC utilisant l'approche adaptative, tout d'abord par rapport à d'autres formes d'arbres dans le même support, puis par rapport aux supports OpenMP des compilateurs GCC 4.4.4 et ICC 12.1. Ces comparaisons ont été effectuées sur les deux configurations matérielles présentées, et en se focalisant sur l'activation des threads dans une région parallèle et leur synchronisation dans le cas d'une barrière OpenMP.

Nous nous sommes servis de la suite de microbenchmarks EPCC pour estimer le surcoût du support MPC.

Les expérimentations décrites suivantes ont été menées sur le noeud large contenant 128 coeurs.
Nous montrons tout d'abord que l'arbre adaptatif permet un surcoût inférieur aux autres formes d'arbres (4-32, 4-4-8), concernant l'activation des threads, sur une plage de 1 à 128 threads lancés. La forme d'arbre 4-32, même si elle offre de bonnes performances jusqu'à 8 threads, devient inefficace au-delà car les threads ne peuvent plus être contenus dans un seul socket, et cette forme d'arbre ne respecte pas la topologie. La forme d'arbre 4-4-8 fournit de bonnes performances sur toute la plage de threads car elle suit la topologie matérielle, mais demeure moins compétitive que l'arbre adaptatif jusqu'à 8 threads.

Toujours avec la même configuration matérielle, nous obtenons de bons résultats par rapport à d'autres supports OpenMP. Les performances obtenues avec la librairie libGOMP de GCC 4.4.4 sont largement en deçà de celles de MPC et de ICC 12.1. Sur 128 threads, nous obtenons un surcoût de 8 microsecondes environ avec MPC et l'arbre adaptatif activé, contre 39 microsecondes environ avec ICC 12.1.

1.2.2 Etude des opérations collectives dans un contexte hybride MPI+OpenMP

D'autres goulets d'étranglements que le surcoût des supports exécutifs OpenMP ont été identifiés, freinant le passage à l'échelle des codes MPI+OpenMP, telles que le ratio des parties séquentielle et communication dans le temps total d'exécution, ou la sous-utilisation des coeurs de libre en mode *Masterlonly* (figure 1.4).
Ces aspects nous ont conduit à nous intéresser aux opérations collectives MPI, et notamment les opérations relatives aux réductions. L'optimisation des opérations collectives MPI fait l'objet d'intenses recherches, desquelles nous avons dégagé deux axes: l'utilisation de la mémoire partagée et l'exploitation de la topologie matérielle et réseau sous-jacents.

De nombreuses contributions s'appuient sur la mémoire partagée pour optimiser les opérations collectives. L'une d'entre elles se concentre les opérations de réduction et présente deux algorithmes: *Recursive Halving and Doubling*, et *Binary Blocks*, décomposant une réduction en deux opérations: `Reduce_Scatter` et `Allgather`. Le principe pour ces deux algorithmes est de découper les vecteurs de départ récursivement, et à chaque étape, une moitié est envoyée à la tâche MPI voisine. Une autre

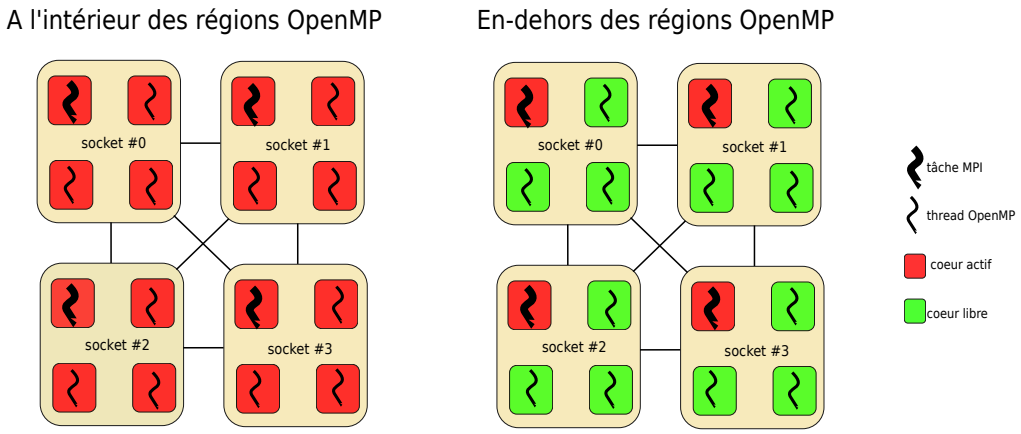


Figure 1.4: Vue matérielle de l'approche Masteronly

contribution propose deux variantes pour paralléliser l'opération `MPI_Allreduce`. Dans la première variante, chaque processus MPI divise son espace mémoire en autant de sous-blocs que de processus, et calcule une réduction locale, stockée dans le premier bloc mémoire. La seconde variante permet d'optimiser le trafic mémoire: les réductions locales sont stockées suivant une approche cyclique.

Exploiter la topologie réseau et matérielle est une autre manière d'optimiser les opérations collectives. Il peut par exemple s'agir de concevoir des algorithmes ayant connaissance de la topologie réseau, en utilisant des sous-communicateurs. Par ailleurs, le framework *HierKNEM* inclut des algorithmes permettant de séparer les communications inter-noeud des communications intra-noeud.

1.2.3 Hybridisation de l'opération collective MPI_Allreduce

Notre seconde contribution se concentre sur l'opération collective Allreduce et consiste à découper les vecteurs de l'opération Allreduce en plusieurs blocs, à l'aide des coeurs de libres. Ainsi, chaque thread OpenMP a en charge d'effectuer l'opération sur un sous-ensemble du vecteur de départ (figure 1.5).

La conséquence principale de l'algorithme consiste à dupliquer les appels au support exécutif de MPI, mais le motif de communication demeure le même pour tous les appels: tous les rangs MPI effectuent une réduction vers un même rang MPI destination, la tâche MPI destination effectue la rédution, puis diffuse le résultat vers toutes les autres tâches. La raison est que les threads OpenMP possèdent une copie du même communicateur d'entrée, avec un ordre des tâches MPI identiques. Ceci peut générer de la contention et du déséquilibre.

Pour pallier à ce défaut, nous proposons une technique complémentaire, appelée Rank Shifting. Celle-ci consiste à donner un rôle différent à chaque thread OpenMP, en effectuant un décalage au niveau du sous-communicateur. Par exemple, le thread de rank 0 aura une copie identique du communicateur de départ. En revanche, le thread de rang 1 fera un décalage de telle sorte que le premier rang MPI dans son sous-communicateur sera celui de rang 1, et ainsi de suite pour les autres threads OpenMP. Dans le cas où il y a plus de threads que de tâches MPI et que nous arrivons au dernier rang MPI, nous effectuons une rotation de sorte à revenir à la première tâche MPI. Cette technique permet donc de rééquilibrer les communications entre tâches MPI.

Les algorithmes présentés ont été implémentés dans un wrapper, qui capture les appels à MPI_Allreduce via l'interface PMPI. Ce wrapper est compilé en tant que librairie partagée et préchargé à l'aide de la variable d'environnement `LD_PRELOAD`, lorsque l'on exécute une application. Grâce à cette approche,

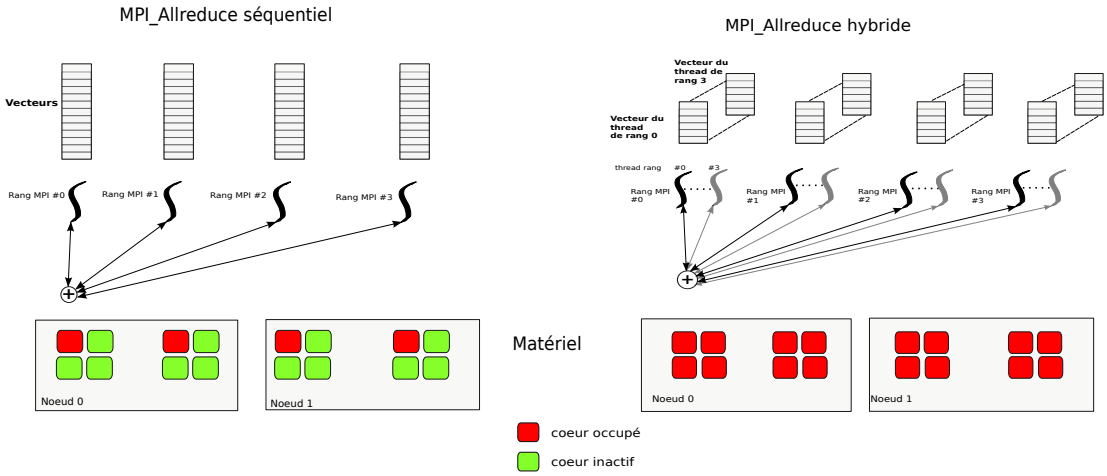


Figure 1.5: MPI_Allreduce hybride

notre implémentation fonctionne avec n'importe quel support exécutif MPI et OpenMP. Cependant, l'implémentation doit fournir le support `MPI_THREAD_MULTIPLE`, du fait que l'on effectue des appels concurrents à la couche MPI.

Expérimentations

Nous avons dans un premier temps évalué l'hybridisation de la collective MPI_Allreduce sur la suite Intel MPI Benchmarks 3.2, sur quatre noeud large contenant 128 coeurs de calculs, avec une tâche MPI par noeud.

Une première expérimentation a consisté à observer le gain de l'hybridisation sur le framework MPC. Nous obtenons la plus grosse accélération en utilisant 8 threads pour l'hybridisation (gain de 2,57), sur une taille de message de 16Mo. La méthode du Rank Shifting sur cette hybridisation apporte une accélération additionnelle de 18,6%. En testant l'hybridisation de la collective avec le support IntelMPI, nous obtenons une accélération maximale de 2,55 avec 90 threads, par rapport aux résultats sans hybridisation avec IntelMPI.

Nous avons dans un second temps évalué notre approche avec une application hybride MPI+OpenMP: MC. Il s'agit d'un code simulatant des déplacements aléatoires de particules, basé sur des méthodes de Monte Carlo. OpenMP est utilisé pour paralléliser les boucles de calcul. Les particules se déplacent sur un domaine unidimensionnel, répliqué sur chaque tâche MPI. L'application contient une opération collective MPI_Allreduce, servant à mettre à jour l'état du maillage à chaque itération de la simulation.

Nous avons tout d'abord mesuré le temps d'exécution de MC avec différents supports exécutifs: MPC, IntelMPI et BullxMPI. Les noeuds de calcul sont intégralement remplis quand les boucles OpenMP sont exécutés. Nous obtenons les meilleures performances avec BullxMPI.

Nous avons ensuite mesuré les temps d'exécution de MC avec les mêmes supports exécutifs, mais en appliquant l'hybridisation de MPI_Allreduce. L'hybridisation n'est pas applicable avec BullxMPI, car celui-ci ne supporte pas le niveau de multithreading `MPI_THREAD_MULTIPLE`. Pour MPC, nous obtenons la meilleure performance avec 8 threads, performance encore améliorée en appliquant la méthode du Rank Shifting (accélération totale de 5.29 par rapport à MPC sans hybridisation). Avec IntelMPI, la

```
int main(int argc, char **argv)
{

 MPI_Init(&argc, &argv);

#pragma omp parallel
 {

 /*  Code to be executed */

#pragma omp barrier

#pragma omp single
  {
    MPI_Barrier();
  }

 /* Code to be executed */

 }

 MPI_Finalize();

}
```

Figure 1.6: Barrière unifiée MPI+OpenMP sans optimisation

meilleure hybridisation est obtenue avec 9 threads.

1.2.4 Opérations collectives unifiées

Nous revenons au parallélisme à gros grain dans le contexte du modèle hybride, et soulignons la complexité de mise en oeuvre de cette approche. OpenMP peut être utilisé dans une optique SPMD (Single Program Multiple Data), ou à mémoire distribuée. A chaque thread OpenMP est assigné un sous-domaine à calculer. Il est possible d'effectuer des opérations collectives telles que diffusion ou réduction en utilisant OpenMP avec ce mode-ci. Si l'on se replace dans un contexte hybride, et couplons MPI avec OpenMP dans un mode SPMD, nous pouvons réfléchir aux moyens de concevoir des opérations collectives impliquant aussi bien des tâches MPI que des threads OpenMP.

Le fait de concevoir des collective mettant en jeu aussi bien des tâches MPI que des threads OpenMP implique une forte coopération entre le support MPI et celui de OpenMP.

Nous motivons notre approche à l'aide d'une preuve de concept, qui est une barrière unifiée, synchronisant toutes les tâches MPI et tous les threads de toutes les équipes OpenMP.

Nous commençons par décrire une implémentation de la barrière unifiée à l'aide de constructions existantes MPI et OpenMP (figure 1.6). Pour synchroniser l'ensemble des tâches MPI et des threads OpenMP, une première barrière OpenMP est nécessaire, puis un appel à MPI_Barrier() et une seconde barrière OpenMP. Cependant, en examinant l'exécution de cette barrière, nous constatons que des optimisations sont possibles. Pour une meilleure compréhension, nous allons décomposer chaque barrière OpenMP en deux semi-barrières: la première semi-barrière se charge de synchroniser les threads, et la seconde semi-barrière libère les threads afin qu'ils continuent leur exécution.

A la fin de la première barrière OpenMP, un seul thread exécute la barrière MPI alors que les autres threads continuent leur exécution et commencent à exécuter la seconde barrière OpenMP. Ils se retrouvent donc à la fin de la semi-barrière à attendre le dernier thread.

Notre contribution concernant cette partie consiste donc à proposer une version optimisée de la barrière unifiée, proposant un appel à une seule fonction, et implémentée au niveau du support exécutif. Dans cette version optimisée, nous effectuons une première semi-barrière OpenMP (figure 1.7). Puis le dernier thread arrivé appelle une barrière MPI (figure 1.8), puis nous effectuons la seconde semi-barrière OpenMP (figure 1.9). Cette conception fonctionne avec toute implémentation de la barrière MPI.

Nous avons implémenté une version optimisée de cette barrière unifiée dans le framework MPC, cette librairie offrant une vision commune entre les tâches MPI et les threads OpenMP. L'idée est

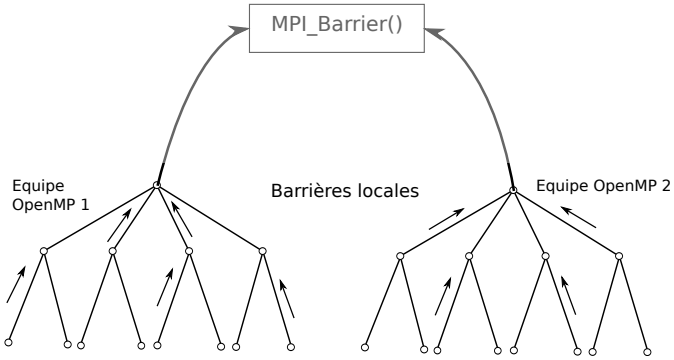


Figure 1.7: Première étape de la barrière unifiée: synchronisation des équipes OpenMP

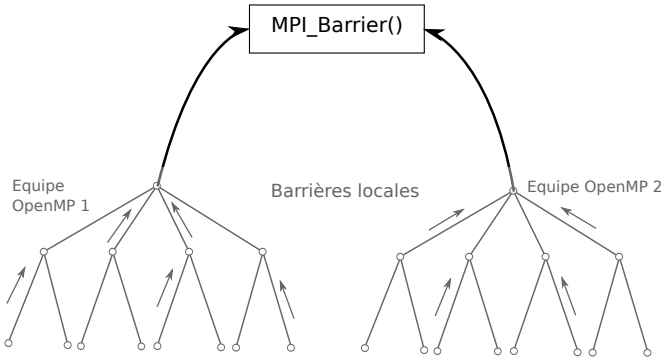


Figure 1.8: Seconde étape de la barrière unifiée: appel de la barrière MPI

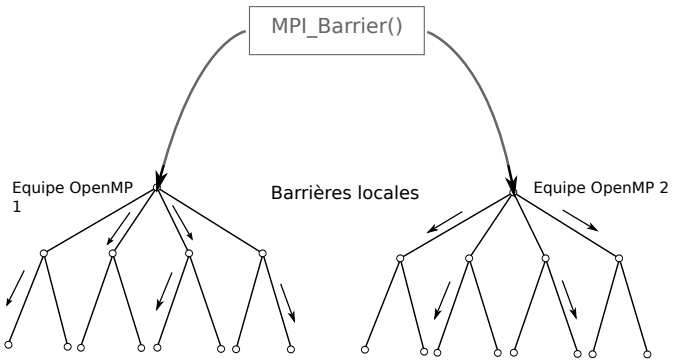


Figure 1.9: Troisième étape de la barrière unifiée: libération des équipes OpenMP

d'effectuer cette opération en un seul appel.

Expérimentations

Pour valider notre approche, nous avons comparé les performances de notre barrière unifiée optimisée avec une implémentation classique d'une barrière unifiée. Pour ce faire, nous avons modifié la suite de microbenchmarks EPCC, et avons effectué des expérimentations sur un noeud large BCS contenant 128 coeurs de calcul, en testant différents ratios de tâches MPI et de threads OpenMP, de telle sorte de remplir le noeud de calcul.

Avec la barrière unifiée optimisée, nous obtenons la meilleure accélération avec une configuration avec 2 tâches MPI et 64 threads OpenMP par tâche. Nous expliquons ce résultat du fait que notre version optimisée économise un appel à une barrière OpenMP entière, ce qui a pour conséquence de gagner un temps significatif avec un grand nombre de threads. En revanche, nous avons une accélération négligeable avec 64 tâches MPI et 2 threads OpenMP par équipe. Nous estimons qu'avec cette configuration, nous passons la majorité du temps dans la barrière MPI, et l'optimisation sur les barrières OpenMP a donc peu d'incidence sur le temps total d'exécution.

1.2.5 Analyse des performances des codes MPI+OpenMP au niveau applicatif et du support d'exécution

Dans la dernière partie, nous nous intéressons aux divers problèmes susceptibles de freiner les performances des codes MPI+OpenMP et empêcher leur passage à l'échelle. Identifier ces goulets d'étranglement est une tâche complexe pour des codes s'exécutant sur des milliers, voire des millions de coeurs de calcul. Cette tâche est complexifiée par le fait que ces problèmes peuvent provenir d'une mauvaise conception de l'application ou bien du support d'exécution, a fortiori lorsque deux supports d'exécution cohabitent, dans le cadre des codes hybrides. Des outils deviennent alors nécessaires afin d'identifier ces problèmes. Nous introduisons dans cette dernière partie ce qui a trait à l'analyse de performances des applications parallèles.

L'analyse de performance des applications parallèles peut être décomposée en trois étapes:

- **Instrumentation**: Une première étape consiste à instrumenter l'application à analyser, à savoir insérer des sondes soit dans le code source, soit dans l'exécutable ou bien encore dans le support d'exécution.
- **Mesure**: La seconde étape permet de mesurer différentes parties de l'application cible à l'aide de l'instrumentation effectuée. Plusieurs techniques sont répandues lors de cette étape, telles que le profilage, permettant par exemple de mesurer le temps passé dans des fonctions, ou bien le traçage, consistant à suivre l'exécution de différentes parties de l'application au fil du temps.
- **Analyse**: La dernière étape permet d'effectuer une analyse des faiblesses de l'application à partir des mesures effectuées, et comprend des fonctionnalités permettant de visualiser les mesures effectuées.

Nous nous focalisons maintenant sur l'étape d'instrumentation et examinons de quoi nous avons besoin pour instrumenter des codes MPI+OpenMP. Des outils d'instrumentation existent pour les codes MPI tels que PMPI.

On trouve diverses contributions proposant des langages pour instrumenter les codes OpenMP: OpenMP Pragma And Region Instrumentor (OPARI), un compilateur source à source permettant de localiser les directives OpenMP dans le code source et d'insérer des fonction à l'aide de l'interface POMP. OpenMP Runtime API (ou Collector API), conçu au sein du compilateur OpenUH, permet d'insérer des événements dans un support d'exécution OpenMP. ORA est notamment supporté par l'outil TAU.

Enfin, nous introduisons OMPT, basé sur les mêmes techniques que Collector API, à savoir l'insertion d'événements dans un support d'exécution. OMPT fournit un plus large panel d'événements que Collector API et est intégré au standard OpenMP.

OMPT, qui doit être implémenté dans un support exécutif OpenMP, fait l'interface entre le support d'exécution et un outil d'analyse de performance.

Les événements fournis par OMPT sont associés aux constructions OpenMP, et sont généralement utilisés par paire: une paire d'événements pour les région parallèles ou encore les boucles OpenMP. Voici quelques exemples d'événements associés à des constructions OpenMP:

- `ompt_event_parallel_begin / ompt_event_parallel_end`
- `ompt_event_loop_begin / ompt_event_parallel_end`

Ces événements, insérés dans un support d'OpenMP, permettent à l'outil de prendre connaissance de l'activité des threads OpenMP, à quel moment ils exécutent des constructions, etc.

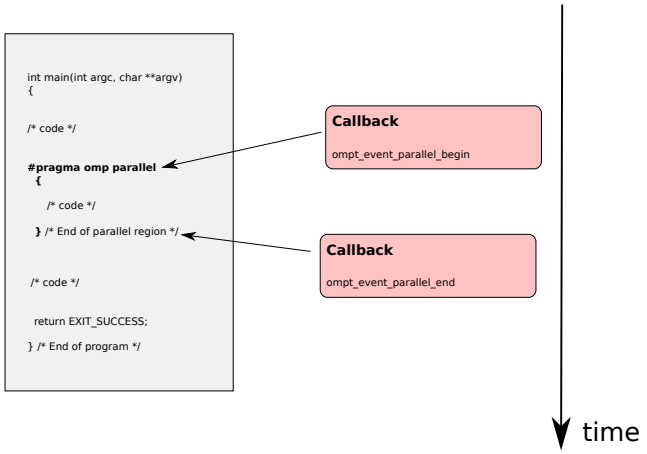


Figure 1.10: Insertion des événements relatifs aux régions parallèles OpenMP à l'intérieur du support d'OpenMP

La figure 1.10 présente un exemple de code contenant une région parallèle OpenMP et montre où doivent être insérés les événements afin d'instrumenter ladite région parallèle.

D'autres événements permettent d'étudier l'efficacité du support exécutif OpenMP, tels que la paire `ompt_event_implicit_task_begin` et `ompt_event_implicit_task_end`, permettant de mesurer le temps d'exécution du code parallèle. On peut alors déduire le surcoût du support exécutif.

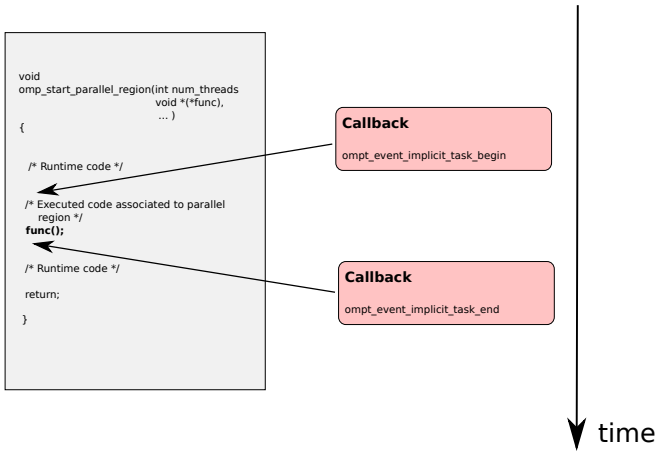


Figure 1.11: Insertion des évévements relatifs aux tâches implicites à l'intérieur du support d'OpenMP

Le code de la figure 1.11 présente le squelette d'une implémentation d'une région parallèle OpenMP. La figure montre où doivent être placés les événements relatifs aux tâches implicites dans le support.

La façon dont OMPT s'interface avec l'outil est la suivante: OMPT est démarré à l'initialisation du support OpenMP et arrêté à la terminaison de ce dernier. Pour initialiser l'outil, la norme propose une fonction `ompt_initialize`, qui doit être implémenté aussi bien du côté du support que celui de l'outil. Une fois OMPT initialisé, l'outil enregistre des callbacks relatifs aux événements susceptibles de l'intéresser, et OMPT doit notifier l'outil lorsque des événements enregistrés par ce dernier sont rencontrés.

Nous présentons ici notre dernière contribution, à savoir l'implémentation de l'outil d'instrumentation OMPT dans le support exécutif MPC, puis son évaluation sur des application MPI+OpenMP.

Les événements OMPT liés aux principales constructions OpenMP ont été implémentés dans la version 2.5.0 du support MPC.

Il est à noter que, puisque la majorité des événements est fournie par paires (une pour l'entrée dans une construction OpenMP et une pour la sortie), des fonctions d'entrée et de sortie sont requises pour insérer ces événements. Cependant, Il n'y a pas de fonction générée pour la construction `#pragma omp loop schedule(static)`, et une seule fonction est générée pour `#pragma omp single`: la fonction retourne 0 ou 1 selon que le thread courant exécute la construction ou non.

Ces problèmes nous ont poussé à modifier l'interface de GCC avec le support d'exécution afin de générer les fonctions requises, permettant l'insertion des événements OMPT liés à ces deux constructions.

Une fois l'outil OMPT implémenté dans le support MPC, nous nous sommes dans un premier temps concentrés sur une étude sur les boucles OpenMP et comment il est possible de les optimiser à l'aide de OMPT. Nous avons donc repris l'application MPI+OpenMP MC, en nous focalisant sur les boucles OpenMP, en étudiant comment il était possible d'optimiser les boucles en réglant la politique d'ordonnancement et la taille des blocs.

Nous avons tout d'abord comparé les politiques d'ordonnancement statique et dynamique pour une boucle d'une fonction donnée, et avons observé un ralentissement très important lorsque l'on passe de la politique statique à la politique dynamique. Plusieurs raisons peuvent expliquer ce ralentissement:

- Dans la norme OpenMP, lorsqu'aucune taille de bloc est spécifiée avec la politique d'ordonnancement statique, la boucle est divisée en autant de blocs que de threads. Ce cas implique un minimum d'intervention du support exécutif, alors que pour la politique d'ordonnancement dynamique, chaque thread choisit un bloc de taille 1 à exécuter, et ceci jusqu'à l'exécution complète de la boucle. Dans ce dernier cas, le support d'exécution est plus sollicité que pour la politique statique.
- La politique d'ordonnancement dynamique autorise un équilibrage de travail entre les threads. Aussi, nombre de supports exécutifs OpenMP implémentent des algorithmes de vol de travail. Il est possible qu'un mauvaise implémentation du vol de travail gêne le travail des threads, ou génère de la contention au niveau des communication.

Par ailleurs, nous nous servons de OMPT afin de déterminer de manière empirique la meilleure combinaison entre la politique d'ordonnancement et la taille de blocs en terme de performances. Nous comparons les performances des boucles avec les supports MPC OpenMP et Intel OpenMP.

Enfin, nous pouvons utiliser OMPT afin d'estimer le surcoût du support OpenMP de MPC, à l'aide de la paire d'événements `ompt_event_implicit_task_begin` / `ompt_event_implicit_task_end`. Le surcoût est déduit en comparant le temps passé dans une région parallèle et celui passé dans une tâche implicite, puisqu'il s'agit du temps mis pour réveiller les threads.

1.3 Conclusion et perspectives

Cette thèse s'est intéressée à différents aspects du modèle hybride MPI+OpenMP, et s'est focalisée sur les limitations venant avec ce modèle, empêchant le passage à l'échelle des codes parallèles. Nous avons présenté des contributions proposant des solutions à ces freins:

- **Arbre adaptatif**: Afin de répondre au besoin d'efficacité des supports exécutifs OpenMP dans un contexte hybride, notre première contribution a consisté proposer un arbre adaptatif permettant d'optimiser l'activation et la synchronisation des threads en environnement NUMA, et sur un large spectre de threads.

- **Hybridisation de l'opération collective MPI_Allreduce**: Nous avons proposé une méthode permettant d'accélérer l'opération collective MPI_Allreduce, en utilisant les threads OpenMP et ainsi réutilisant les coeurs de calcul inactifs. Cette contribution répond une limitation rencontrée avec le parallélisme à grain fin.

- **Opérations collectives unifiées**: Nous avons introduit le concept d'opérations collectives unifiées, ceci afin d'assurer une meilleure coopération entre MPI et OpenMP utilisé dans le mode SPMD. Nous motivons notre approche avec une preuve de concept, la barrière unifiée.

- **Implémentation et évaluation de l'outil d'instrumentation OMPT**: Enfin, nous nous concentrons sur l'analyse de performance des codes MPI+OpenMP et proposons une implémentation de l'outil d'instrumentation OMPT, ainsi que son évaluation sur une application MPI+OpenMP, en ciblant les boucles OpenMP.

Nous terminons en abordant quelques axes de réflexion concernant le modèle hybride.

Tout d'abord, il est à noter que les architectures parallèles devraient être de plus en plus hétérogènes, les noeuds de calcul accueillant des accélérateurs matériels, citons par exemple le processeur Intel Xeon Phi. De par les caractéristiques de ces processeurs spécialisés, les modèles de programmation ont évolué, avec l'apparition de CUDA, du modèle OpenACC puis de l'évolution du standard OpenMP vers la révision 4, fournissant des directives permettant de déporter des calcul vers un GPU par exemple. Cependant, la prochaine étape de notre point de vue consiste à pouvoir exécuter des codes MPI+OpenMP sur ces GPUs. Ainsi, le modèle hybride MPI+OpenMP devrait évoluer suivant deux axes:

- Optimisation des couches MPI et OpenMP pour les architectures manycore

- Evolution de la taxonomie hybride pour les environnements hétérogènes

1.3.1 Travaux futurs à court terme

Optimisation du support exécutif OpenMP en environnment Manycore. Nous avons proposé des techniques afin de réduire le surcoût des supports exécutifs OpenMP pour des processeurs traditionnels. Il est possible d'adapter notre contribution pour les architectures manycore. Nous prenons l'exemple du processeur Intel Xeon Phi et étudions quel arbre serait le mieux adapté afin de minimiser le surcoût du support d'OpenMP. Un arbre respectant la topologie matérielle serait composé de autant de noeuds par coeur, et chaque noeud aurait quatre enfants (un par hyperthread). Cependant, les modèles actuels du Xeon contiennent de 60 à 70 coeurs de calcul. Activer les threads suivant un arbre contenant 60 enfants pourrait conduire à de la contention. Ainsi, se tourner vers des arbres de plus grande profondeur serait peut-être plus pertinent.

Optimisation des opérations collectives MPI à l'aide de threads OpenMP. Concernant l'hybridisation des opérations MPI collectives, nous pensons appliquer le travail effectué sur MPI_Allreduce à destination d'autres opérations collectives, telles MPI_Bcast ou MPI_Gather. Cependant, il faut indiquer que la technique du Rank Shifting ne fonctionne pas sur ces opérations collectives.

Par ailleurs nous aimerions pouvoir effectuer une hybridisation automatique en prédisant quel serait le nombre de threads nécessaires pour garantir la meilleure accélération, en s'appuyant sur un modèle de performance, se basant sur des critères tels que la longueur du vecteur ou bien la topologie matérielle. D'autre part, la technique du Rank Shifting ne prend pas en compte la topologie matérielle. Nous voudrions optimiser cette technique en évaluant différentes distances de décalage.

Opérations collectives unifiées. Concernant les opérations collectives unifiées, nous avons motivé notre approache par une preuve de concept, la barrière unifiée. Nous aimerions appliquer l'approche à d'autres opérations collectives, telles que la réduction.

En dehors des opérations collectives, nous aimerions également concevoir une opération assurant le travail des communications MPI à l'intérieur d'une région parallèle OpenMP, de sorte qu'elle ne soit appelée qu'une fois par tâche MPI.

Analyse de performances de codes hybrides à l'aide de OMPT. Au sujet de notre dernière contribution concernant l'analyse de performance des codes hybrides MPI+OpenMP, nous nous sommes servis de l'implémentation de l'outil OMPT afin de guider l'optimisation des boucles OpenMP. Le ralentissement observé lors d'une étude de cas, en passant d'un ordonnancement statique à un ordonnancement dynamique, nous a poussé à réfléchir sur le rôle joué par le support exécutif dans les performances des boucles. Malheureusement, les événements proposés par OMPT ne nous permettent d'estimer que partiellement l'efficacité du support d'exécution.

1.3.2 Perspectives à long terme

Lors de cette thèse, nous nous sommes concentrés sur certaines problématiques rencontrées lorsque l'on cherche à combiner deux modèles de programmation parallèle.

Empilement de modèles. Nous voudrions maintenant étudier l'imbrication de N modèles et identifier les problématiques associées. Pour cela, nous reprenons notre exemple de barrière unifiée, et décrivons comme elle se déroulerait avec N modèles imbriqués.

Tout d'abord, le modèle M_{n-1} effectuer un semi-barrière, et le modèle appelant effectue à son tour une semi-barrière. Nous continuons ainsi jusqu'à ce que le modèle M_0 effectue une barrière complète. Une fois la barrière complète effectuée, les modèles imbriqués effectuent chacun une semi-barrière dans la direction opposée, jusqu'à atteindre le modèle M_{n-1}.

Ordonnancement des tâches MPI et des threads OpenMP. Tout au long de cette thèse, nous avions proposé des contributions se basant sur le fait que les tâches MPI et les threads OpenMP sont distribués de manière statique sur les coeurs de calcul. Cependant, avec l'introduction des opérations collectives MPI non bloquantes depuis MPI-3, un coeur de calcul utilisé pour effectuer des communications peut être réutilisé pour effectuer des calculs, avant que la communication ne soit terminée.

Ce cas conduit au problème de l'ordonnancement des flux d'exécution sur les coeurs, et conduit à faire des choix entre privilégier les communications et les calculs.

Part I

Context

Chapter 2

Introduction on High Performance Computing

This chapter introduces the field of High Performance Computing coming with the advent of supercomputers, as a support to numerical applications. We will then see how to program these supercomputers with their underlying parallel architectures, what are the challenges coming with these architectures, and how we will take them up.

2.1 High Performance Computing for numerical simulation

In this part, we will talk about Numerical Simulation as a tool allowing to understand phenomena in various domains. This field was first used in research community and was then adopted by the industry as a mean to reduce costs. But Numerical Simulation aims at solving more and more complex mathematical equations and compute infrastructures had evolve to provide the necessary power.

2.1.1 Numerical simulation for physic models

Numerical codes aim at understanding phenomena from the real world, by reproducing them with the help of simulation. This field was introduced with the emergence of computer sciences, during World War II: scientists wanted to control the process of nuclear detonations. This project was as fundamental for the outcome of the war as secret, and was named Manhattan project. Computers were very useful tools to obtain these results. For decades, military needs drove the reasearch in this field, but it also grew for the benefit of other fields, such as physics, biology or medicine.

Here are some examples of such numerical applications:

- In astrophysics, DEUS (Dark Energy Universe Simulation [91]) aims at getting more knowledge about the imprints of dark energy into the universe
- Understanding how human brain works has always been a fascinating subject. The Human Brain Project [80], a European initiative, aims at constructing realistic simulations of the humain brain, by modeling its biological processes.

But designing realistic simulation requires a long process, and various skills.

Figure 2.1 gives a description of the process about numerical simulation. Starting from the real world and a physic phenomenon, physicists try to design a model describing this phenomenon using equations, which often are Partial Derived Equations. Once this model and its equations have been correctly established, these equations need to be analysed and solved. Computers can be used for this step, but the equations have to be expressed in an understandable way. Numerical scientists then design algorithms in order to solve these equations. Moreover, in most numerical codes, instead of calculating an exact solution to the input equation, which is time consuming, we try to find an approximate solution. Several techniques such as Finite Element Method[30] or Monte Carlo methods are available to obtain

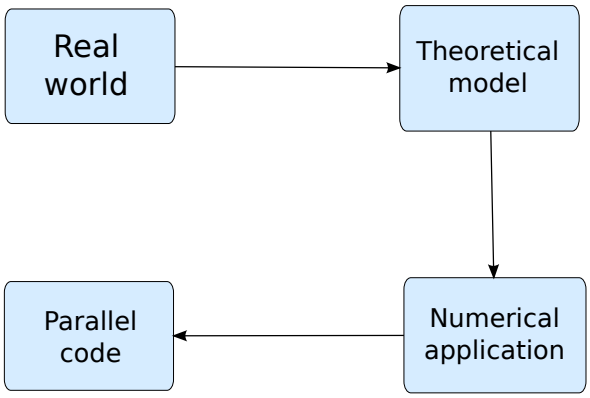


Figure 2.1: Process for numerical simulation

approximate solutions. They are based on iterative computations.

Nowadays, with the help of computer sciences which spread in companies and personal lifes, Numerical Simulation and High Performance Computing are more and more adopted by industry, in fields such as aeronautics, automobile or finance. Indeed, they are seen as tools helping the design of products by reducing both the risks and the duration of development.

But Numerical codes are very demanding of computational cycles and memory resources. The increasing data necessary to perform simulations, and the need to reduce their duration, encourage research community and industries to seek for more compute power and adopt supercomputers.

2.1.2 Parallel machines to support numerical simulation

But performing simulations requires a huge amount of computational capabilities. This is why a special kind of computers emerged in order to help scientists. First supercomputers appeared in the 1960s, with the CDC-6600 computer [102]. In the early 1970s, Seymour Cray founded Cray Research and created a new family of supercomputers, starting with the Cray-1 [95].

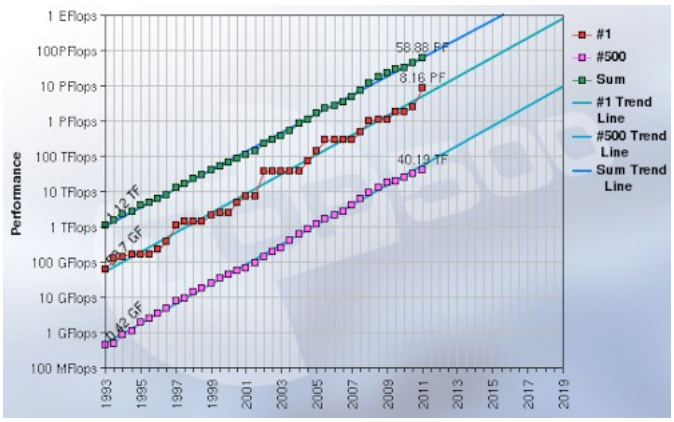


Figure 2.2: Evolution of performances from the first to the last machine in Top500 ranking

The website TOP500 lists the 500 most powerful machines in the world [4]: their performances are estimated by counting the number of floating point operations per second (flops). Performance of machines in this ranking is measured using the benchmark suite Linpack[34], which computes a linear system containing n equations, and leads a solution using Gauss pivoting. Figure 2.2 shows the evolution of the computation capabilities of the most powerful machines, and compares the evolutions between the first and the last computers in the list of Top500. If Cray-1 supercomputer could develop only 138 megaflops, lots of progress have been done since this time. In 2001, the japanese machine Earth Simulator reached 40 teraflops and was designed for studying global climate models [50].

(a) Tianhe-2 computer (b) Curie computer

Figure 2.3: Current supercomputers

Nowadays, current supercomputers are composed of clusters, interconnected by high-speed networks. They reach a compute power of several petaflops (10^{15} floating operations per second), and TOP500 ranking is currently dominated by a chinese machine, Tianhe-2 which reaches more than 30 petaflops and accounts for more than three million cores (Figure 2.3). But as soon as the Petascale barrier was passed, scientists immediately started to seek for the next step.

2.2 Parallel architectures and Memory organization

The last Section introduced supercomputers, from their advent to those encountered nowadays. Starting from the necessity to provide compute power for numerical simulations, efforts in Research&Development have been made to increase capabilities of supercomputers. Thus, in the last Section we focused on the evolution of High Performance machines, from the Cray-1 to Tianhe-2.

In this Section, we get more insight into those machines, describing the different elements composing supercomputers, and how they are organized. We will start from the evolution of microprocessors, its different kinds, and continue with the memory organization. We will see that current supercomputers are more and more parallel.

2.2.1 From sequential to multicore machines

Since their introduction in the late 1960s, microprocessors saw their frequency increase at a constant pace. Following their evolution, Gordon Moore formulated a law, stating that the number of transistors integrated inside a chip would double approximately every second years. Consequently, processor frequency increases (Figure 2.4). This law applied until 2004 when processors frequency reached 3GHz. But clock frequency increased thermal dissipation until reaching a limit: heat could not be sufficiently dissipated. The thermal wall led processor designers to take other directions, like multiplying compute units inside the same socket.

Intel introduced the Core microarchitecture after the Pentium 4, after observing thermal effects would be too high. AMD[1] followed this trend, with its processor Athlon 64 X2.

[1] Advanced Micro Devices

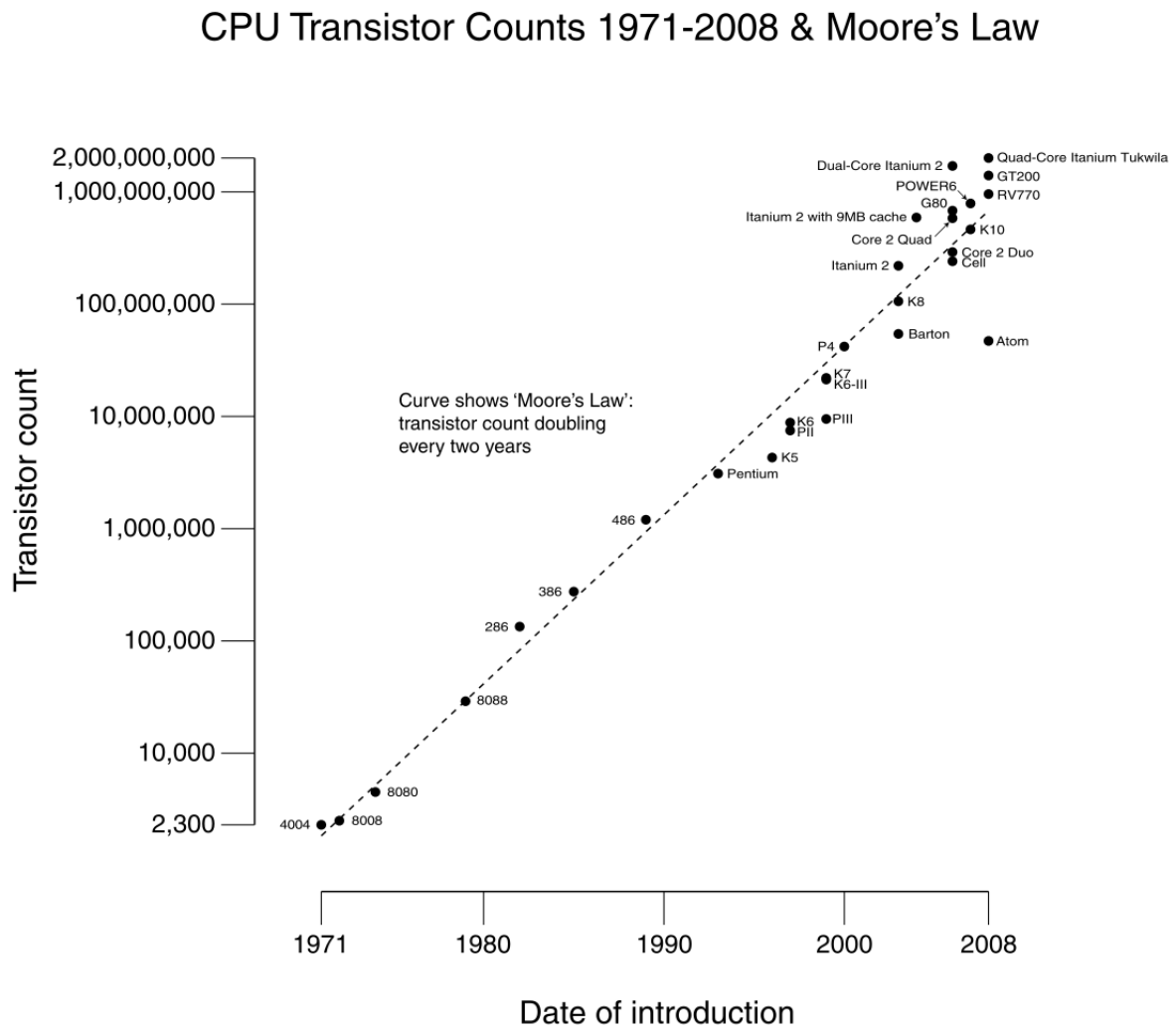


Figure 2.4: Evolution of processors following Moore's law

Meanwhile, processor designers proposed several hardware implementations to provide parallel execution of the compute workflow, inside a single compute core. Instruction Level Parallelism (ILP) allowed accelerating execution of one single instruction, using pipelining: one single instruction is split into steps, and these steps are executed simultanously by different units, which are always kept busy. Superscalar architectures allow execution of several instructions in the same clock tic.

2.2.2 Heterogeneous architectures

In addition to CPUs, another device, the Graphic Processing Unit, provides compute capabilities. GPUs were originally used as drivers to display the interface of the computer on screens. With the advent of video games and tridimensional display, rendering was delegated to the GPU. After being dedicated to graphic purpose, GPUs started to interest scientists for parallel computations, when these processors started to include more and more compute cores, and were able to provide massive parallelism.

They then started to be used in order to accelerate computations, and hardware evolved from being specialized in graphic pipelining to be used for numerical applications, thanks to their compute power. This use is known as General Purpose Graphic Processing Unit (GPGPU), and GPU manufacturers started to propose programming models to exploit them for this purpose. Nowadays, GPUs such as

Geforce brand from Nvidia or Radeon from AMD gather thousands of small compute cores executing simpler instructions than those coming with current CPUs.

From 2010, Intel introduced a new line of processors, called Many Integrated Core Architecture, renamed Xeon Phi in 2012 [58]. This processor follows the development of Larrabee microprocessor [97]. In its first version, it contained 61 compute cores, organized following a ring topology. It also embedded an operating system, run by one of its cores. This architecture is compatible with x86/64 instruction set, and thus provides a portability for applications designed for CPUs. This kind of processors provided capabilities for massive parallelism, and started to be integrated in modern supercomputers.

Sony, Toshiba and IBM introduced the Cell microprocessor in 2005[60]. Its development had started in 2001. The architecture of the Cell combined two kinds of compute cores: the Power Processor Element (PPE) and the Synergistic Processor Elements (SPE). The Cell microprocessor was embedded in the PlayStation 3, developed by Sony, and also equipped the Roadrunner computer [13].

Such architectures, by providing massive parallelism, can be considered as alternatives to CPUs or be combined with them. But for sure, hardware accelerators will be more and more integrated in future supercomputers.

2.2.3 Memory hierarchy

In order to ensure execution of the application, nowadays computers include the central memory, also called RAM (Random Access Memory). The role of this memory system is to store the data needed during the execution of the application, and to feed the CPU as data are required. This memory is reset when the computer is turned off or rebooted. But as CPUs and central memory are separated, the time to retrieve data is expensive and can impact performances. Indeed, if the execution time is mainly used to load data, the CPU will suffer from under utilization.

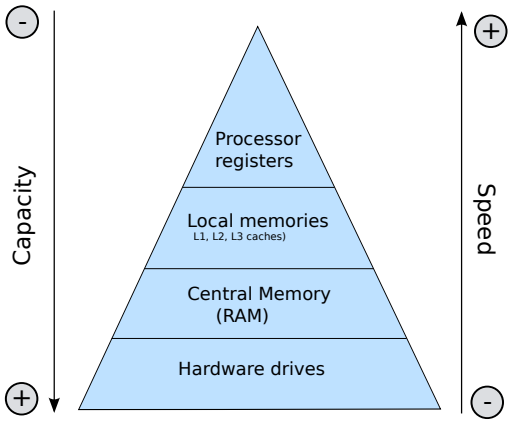


Figure 2.5: Memory hierarchy

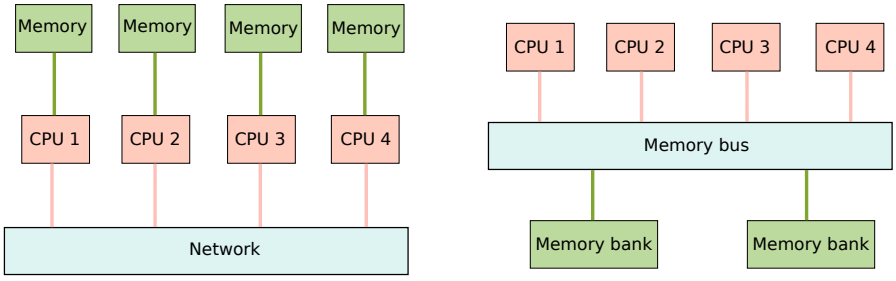
In a more general way, computational resources need data to work properly, and applications have a constant need of data storage. But memory resources come with constraints which are their cost, the latency to access to the memory system and their capacity. These constraints established a memory organization which is depicted on Figure 2.5. We classify memory blocks starting from the peak of the triangle, from the lowest latency to the biggest. All these memories have different functions.
On top of the memory hierarchy, we find the processor registers, located inside a CPU. These registers are faster than any other memory components but also the most expensive to build. When instructions are executed by the processor, data associated to these instructions are temporarily stored in the registers.

To limit accesses from the CPU to the central memory, manufacturers designed smaller memories named caches, and located them near the processors. The latency to access these caches is smaller than the one for central memory, but due to their reduced size, only a subset of data can be contained. This is why only the most frequently used data are placed into caches. Different algorithms are used to decide how to fill caches. Current CPUs contain different levels of caches (L1, L2, L3) which are imbricated. L1 and L2 caches are dedicated to each core, whereas L3 cache is shared between several cores. These local memories have a size increasing following their distance to the cores. They are used in the following manner: when a core needs a data to process, it first requests this data to L1 cache. If this data is contained by this memory, then the core executes it: this is called a cache hit. If this is not the case, then it is considered as a cache miss and data is sought in cache L2. We continue this process until cache L3. At last, if the required data could not be found in any local memory, then the CPU has to retrieve it in the central memory. We see that CPU caches were designed to avoid accesses to the central memory as much as possible. It is also to be noted that different types of cache memory are available: L1, L2, and L3.

At the bottom of memory hierarchy, we find hard drives which are consistent memories. This means data remain in these drives when the computer is turned off.

To summarize, the memory hierarchy has been established to hide latencies from computational resources to memories as much as possible. But this hierarchy implies higher programming efforts for an efficient use of these resources.

2.2.4 Distributed memory systems and shared memory systems



(a) Distributed memory model

(b) Shared Memory model

Figure 2.6: Distributed memory model Vs Shared memory model

In the field of High Performance Computing, memory is organized following various directions. But we can distinguish two main families of parallel systems (Figure 2.6):

- **Distributed memory systems**: In this topology, CPUs are interconnected via a network, and have their own memory system. This kind of system implies, for the programmer, to take care of where the data is located. Indeed, if a required data is located on a remote memory, the programmer has to explicitly take this data. The main bottleneck of distributed memory systems is the network. We find this type of memory organization with Grid Computing [17].
- **Shared memory systems**: Also called Symmetric Multiprocessor (SMP), this model includes several CPUs sharing a same memory bank, on which they access in a concurrent way. Execution flows have then access to the entire memory on a transparent manner. So programming this kind of architecture is simpler than distributed memory systems, as the programmer doesn't have to explicitly copy data from a CPU to another. But this model includes several drawbacks: memory

contention can appear when two many compute units share the same memory bus. Secondly, coherency has to be maintained by the processors: when a data is updated by a compute core, all other units have to be informed about this update. One example of current shared memory system is the Bull Coherency Switch node, containing 128 cores, dispatched in four modules, each module containing 4 8-core CPUs.

2.2.5 From UMA to NUMA architectures

In shared memory systems, different architectures coexist to define access from the processors to the memory.

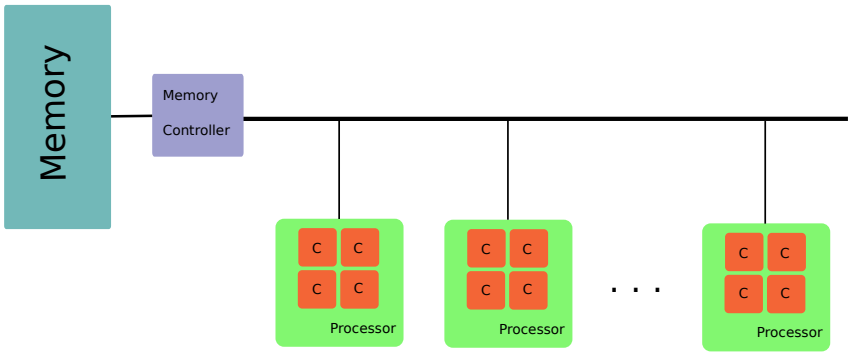


Figure 2.7: Uniform Access to Memory

Uniform Memory Access is one type of shared memory architecture, where all processors access to a same physical memory via a memory bus (Figure 2.7). This architecture provides equal access to the memory to processors, which means the latency of accessing the memory system is constant whatever their location.

But when multiple compute units share the same bus to access resources, memory traffic is increased, and thus the memory bus becomes saturated. Thus, with the multiplication of cores per compute node, this kind of architecture represents a bottleneck for high performance applications.

Non Uniform Memory Access The UMA architectures proved to be insufficient with the multiplication of cores. This is why hardware manufacturers designed more sophisticated shared memory systems. ccNUMA architecture is one of them [66]. It consists of giving each CPU its own memory, in order to solve memory contention.

Figure 2.8 depicts a basic example of ccNUMA architecture, containing four CPUs, each CPU being composed of four compute cores and having a direct access to a dedicated memory, which solves the problem of contention described in UMA systems. Thus, each CPU and its memory are considered as a block called NUMA node. CPUs are interconnected via a bus (QPI [2] for Intel or HyperTransport for AMD). All processors can access to remote memories, and in the user's point of view, NUMA architectures are considered as shared memory systems. And even if memories are separated, cache coherency has to be maintained across the shared memory. But accessing data on distant memories implies additional latencies. Let's take the example of the CPU 2, wanting to access the memory bank located on NUMA node 1. It will have to go through the bus interconnecting sockets before reaching the remote memory. Since this bus offers a lower bandwidth than the one provided between a CPU and its own memory, accessing remote memories becomes more expensive. We all these additional latencies NUMA effects.

[2] QuickPath Interconnect

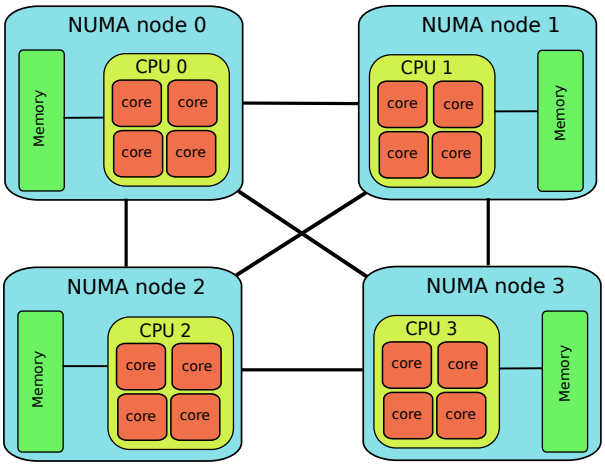


Figure 2.8: ccNUMA architecture

NUMA systems provide a hierarchical parallelism due to their organization, and are complex to program.

2.3 Challenges of Exascale era

As supercomputers provides performance on the order of the Petaflop since 2010, they are expected to reach the Exascale frontier by 2020. But adding more cores will not be enough to reach this step. Thus, HPC scientists already started to describe characteristics of future supercomputers. But these new architectures will come with numerous challenges. These challenges are detailed by several papers [33, 81].

- **Energy**: The cost of electricity becomes a non negligible part of total costs involved in exploiting High Performance machines, and this ratio should continue to increase. Introduced in 2001, the Earth Simulator computer[50] reached 40 teraflops for a power consumption of 3.2 megawatts. Now, Tianhe-2, the most powerful computer since 2014, requires 17 megawatts for a performance peak reaching 33 petaflops. Thus, power constraints influence the design of the next generation of supercomputers. A new ranking, Green500 [40], lists the most energy-efficient computers.

- **Memory wall**: As scientific codes need both computational cycles and memory to execute, both constraints have to be considered. But memory may become the most limiting factor of applications performance, since there is an increasing discrepancy between evolutions of compute resources and memory. One way to compare both resources was to measure the number of bytes per flop. For a while, one byte of memory or more was available per flop. But this ratio started to decrease whith the transition from mono-core to multicore, in 2004. Then this ratio is around 0.1.

- **Reliability**: High performance machines are equipped with thousands of components: compute cores, memory, hard drives, ... The probability of encountering a failure increases with the size of the supercomputer. We define the MTBF[3] [103] as the mean frequency of a hardware fault. It is thus critical for scientific applications to tolerate such faults and make them resilient. Strategies exist to make applications resilient from this constraint such as Checkpoint / Restart [96].

- **Heterogeneity**: Hardware accelerators spread in the field of High Performance Computing and were first used as prototypes. They now start to be integrated in current machines: US computer Stampede includes Xeon Phi co processors [39]. Their compute units are more and more diverse and smarter ways to efficiently program them are required.

[3]Mean Time Between Failure

2.4 Programming models

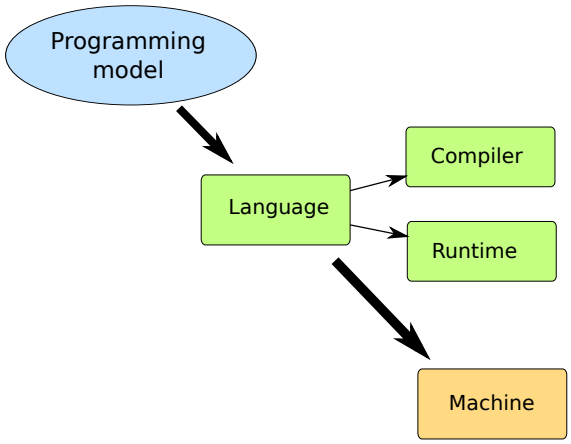


Figure 2.9: From the programming model to the language, from the language to the program

We saw that current supercomputers were massively parallel, with several imbrications of compute and memory resources. These machines are then more and more complex to program, and in order to efficiently exploit them, scientists introduced models to abstract from the compute resources and leverage available parallelism. We define parallel programming models as ways to express parallelism inside applications. These programming models have to be compatible with a language, which comes with a compiler and a runtime. The language is then compiled or interpreted in order to run on the target machine. We illustrate this chain with Figure 2.9.

Various programming models have emerged as parallel machines evolved, and they follow different paradigms. They can be oriented on distributed memory systems, such computers interconnected by a network, and making them work together.

2.4.1 Message Passing models

Early efforts to express parallelism in programs led to the design of processes which are instances of a program, allowing the share of computational resources. A process contains everything needed for executing a program: its own address space, a set of instructions, and a context. Processes were implemented in Unix systems and are the building blocks of Message Passing models. They communicate through pipes, allowing cooperations between the programs over networks.

Message passing models are now able to communicate using simpler techniques.

MPI (Message Passing Interface [35]) is a standard library using message passing paradigm. It was introduced in 1991. It is designed for distributed memory machines, as it allows communications between remote machines. The model involves processes running concurrently and parallelizing application. Communications between processes have to be made explicitly by the user. MPI is now widely adopted for scientific codes, and supports C/C++ and Fortran languages. The standard proposes following features:

- **Point-to-point communications**: MPI includes primitives to enable communications between MPI processes by explicitly giving their ranks.
- **Collective communications**: MPI also allows performing operations such as gathering or reducing data of a set of tasks. For this purpose, it provides structures called communicators to establish the number of processes involved into these operations.

The standard has be revised through its 1.3 version and MPI-2. MPI-3 standard [41] introduced Non Blocking Collectives, allowing asynchronous communications between processes [52]. where a

process can execute other work while a Collective operation is being processed, instead of waiting its termination.

Numerous implementations of MPI are available, such as OpenMPI[47], MPICH2[48], MVAPICH2[71], IntelMPI or MPC (Multi-Processor Computing) [88].

To parallelize computations using MPI model, the Domain decomposition method is often used. Starting from a defined domain containing the problem to solve, this domain is split into parts called subdomains, which are allocated to each MPI task. Each task can then compute its part on a concurrent manner.

2.4.2 Partitioned Global Address Space

When using MPI, it is necessary to specify data transfers. PGAS[4] is a programming models enabling a global address space which is partitioned. This model allows a transparent access to data, whatever they are located on the local node or on a remote one.

Several languages are based on PGAS model:

- **UPC**[5] [105] extends C language and provides a uniform model for both shared and distributed memory systems.
- **Co-Array Fortran** [87] extends Fortran language and adds a syntax to describe parallel operations.
- **Chapel**[25] was developed by Cray since 2003 and provides a higher level way to express parallelism.

2.4.3 Thread-based models

After the advent of processes, lightweight processes appeared, also called threads, which consists in a set of instructions, but do not have their own memory. A thread can be included in a process and can share memory with other threads. Models reying on threads are adapted to shared memory systems, as no data duplication is required for communications between threads. Such models appear to be interesting in the field of High Performance Computing, as they give access to more and more cores per compute node, implying increasing amount of available shared memory. However, they can only be used within a single compute node.

Following are the main examples of Thread-based models.

POSIX Threads [22] also know as pthreads is a standard allowing management of user-level threads. This is part of the POSIX standard (Portable Operating System Interface) which is oriented to Unix systems. Pthreads provide functions for thread creation and termination, and synchronization of threads. This interface allows parallelizing codes in shared-memory environment.

OpenMP 2.5 [27] is the de facto standard programming model for shared memory machines, relying on threads. It has been introduced in 1997 by the Architecture Review Board, and defines directives allowing parallelizing blocks of source codes. The main directives provided by OpenMP consist in opening a parallel region for parts of the code, distributing loops between threads, synchronizing threads, ... OpenMP also offers features for load balancing regarding loops, and allows tuning visibility of variables implied in parallel regions. Its execution model works following a Fork/Join mechanism. Since its 2.5 revision, OpenMP supports C/C++ and Fortran languages. OpenMP is currently supported by compilers and runtimes such as GCC with libGOMP library, Intel OpenMP runtime which was recently open sourced, MPC or OpenUH [69].

[4] Partitioned Global Address Space
[5] Unified Parallel C

2.4.4 Task parallelism

Thread-based models fit to parallelize codes involving structures such as loops, as iteration space is known. But even if some models such as OpenMP include features to correct imbalance, they lack of flexibility and are not adapted for some kinds of data structures. This is the case for linked lists where the size is not known until reaching the end of the list. A last paradigm exist. It can bring flexibility, by declaring tasks associated to a block of instructions or to a function. It is in charge of the runtime to schedule encountered tasks.

In the following paragraphs, we present several Task-based languages:

Cilk [18] appeared in 1994 and was developed at the MIT. It relies on a Fork/Join mechanism and can express threads and tasks with the help of key words. Tasks are linked together by ancestorship. Cilk runtime also includes work stealing scheduler in order to ensure load balancing when executing tasks. It supports C and C++ languages.

Intel TBB [93] is a template library written in C++ and introduced in 2006 by Intel. It allows task parallelism.

OpenMP 3.0 [11] appeared in 2008 and enriched OpenMP standard by offering the possibility for the user to add explicit tasks in OpenMP parallel regions, that can be associated to a block or a function call. It offers more flexibility to parallel codes.

2.5 Limitations of the MPI model

MPI standard has been massively adopted to parallelize numerical codes. Indeed, it is roughly the only message passing model able to work on distributed machines. Classic use of MPI is to bind one MPI task per compute core. But the evolution of supercomputers challenge MPI libraries, as it is more and more complex to reach scalability on millions of cores. We will see in this section the limitations encountered when using MPI on numerical codes.

When studying parallel architectures, we noticed that compute and memory resources tend to evolve at different paces, implying an increasing discrepancy between them. Consequently, amount of memory per compute core should continue to decrease. These trends limit scalability of high performance codes and challenge classic ways to program supercomputers. However, more and more data are needed to perform larger simulations.

Memory footprint, load imbalance and inadequation with the underlying topology are such barriers from performance scalability of MPI codes.

2.5.1 Adequation with underlying topology

With the multiplication of compute units inside nodes and the advent of NUMA architectures, current machines include several levels of parallelism and shared memory. Current nodes contain from 10 to 100 cores per node, and this number should be much higher at exascale horizon, it is critical to efficiently exploit implied shared memory to allow MPI based applications to scale on numerous tasks. Moreover, with the advent of NUMA architectures, compute nodes are organized following a hierarchical way, and this hierarchy is driven by the location of cores relatively to the memory banks. Thus, affinities between compute cores have to be taken into account.

But MPI is a flat model as it has no semantics to take the underlying topology or the characteristics of the architecture into account. Thus, it cannot efficiently exploit hierarchical parallelism, and MPI tasks are distributed on the cores regardless of their organization. At last, since MPI doesn't know if two processes are on the same node or on distant nodes, it cannot make difference between intra-node and inter-node communications.

2.5.2 Memory scalability of domain decomposition method using MPI model

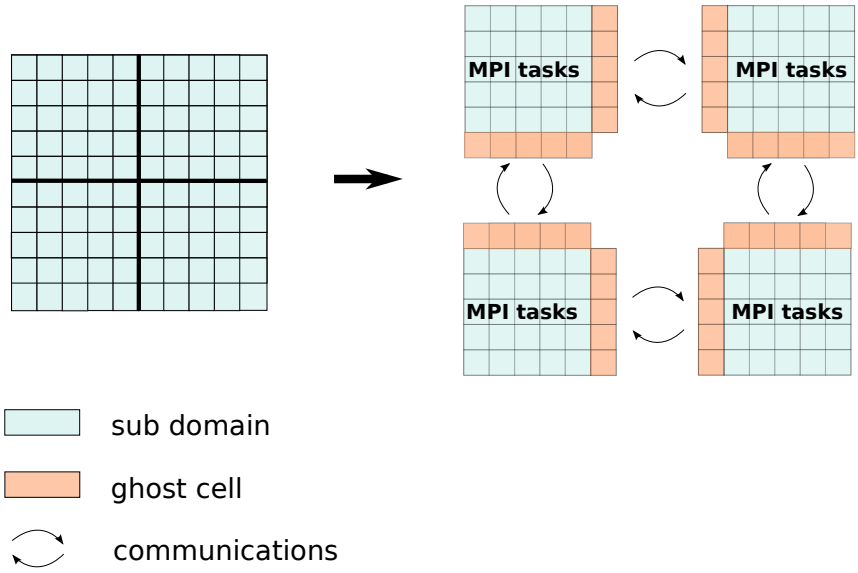


Figure 2.10: Domain decomposition of a 2D mesh with MPI

The domain decomposition method splits the domain into subdomains, and each MPI task is given a subdomain and processes it. But to process subdomains, additional data are required, which are located on the edges of its neighbor subdomains. These are called ghost cells or halo cells, and need to be exchanged between MPI tasks at each time step of the simulation.

Replicating ghost cells in each subdomain implies memory overhead. Since we observed that memory was a limiting factor from reaching high performance, this overhead can prevent from enabling scalability of MPI codes when launching hundreds or thousands of MPI tasks, considering all cores are fully populated.

2.5.3 Problems with load balancing

When writing an MPI application, problem domain is split and equally assigned to MPI tasks. We then assume that all tasks will have the same amount of work to process. This is actually not the case, as all tasks do not run at the same speed. Also, some kinds of applications generate load imbalance. This is the case of ones enabling Adaptive Mesh Refinement algorithm [16]. In this kind of codes, some parts of the mesh are refined during execution, increasing precision on these parts. This algorithm can be used for meteorological codes for example, where local phenomena require more compute power. Thus, tasks do not have equal amount of work, which causes imbalance between them. Several techniques are used to correct this imbalance, like exchanging parts of meshes between processes, even if they are complex to implement. But MPI standard does not provide features to correct such as imbalance.

2.5.4 Optimization of MPI for shared memory

With the multiplication of cores in a single compute node, amount of available shared memory also increases. But there is no difference between intra-node and inter-node communications in MPI standard. For example, in classic implementations of MPI, two memory copies are usually required and two processes located on the same compute node want to communicate data: the sender process performs the first copy on the shared memory, and the receiver process copies back from the shared memory to its internal structures.

But some implementations propose optimizations at several levels for intra-node communications. At first, we can split MPI implementations into two main categories:

- **Process-based MPI**: Each MPI rank is an OS process, with its own private memory. This approach allows easier inter-node communications, and avoid some synchronization between MPI ranks. But it presents limitations when MPI ranks have to communicate inside a node, as two memory copies are generally involved in a local communication: the sender first copies from its private memory the data to a pool on the shared memory, and the receiver copies back to its private memory.

- **Thread-based MPI**: MPI ranks are implemented as OS threads, which share the same address space, excepted the stack and the heap which are private for each rank. But global variables are shared between ranks, which leads to corrupting application state. This problem is solved by privatizing these variables to each thread. MPC is one example of thread-based MPI runtime.

HMPI [42] aims at cumulating advantages of both solutions: optimized for shared memory hardware and providing peak performances for both inter-node and intra-node commmunications. MPI ranks are implemented using OS processes, but the heap segment is shared between them. It is built on top of any MPI library and explores one-copy mechanisms.

Other solutions are investigated to optimize communications between local MPI ranks. KNEM[46] and LiMIC[59] are kernel modules optimizing intra-node communications by implementing a single-copy data transfer between local processes (processes located on the same node). They are respectively implemented in OpenMPI and MVAPICH libraries.

In this section, we exposed several limitations of MPI model. At first, the MPI model is not adequate with the hierarchical parallelism of current architectures as it does not take into account underlying topology or load imbalance in its semantics. Another problem of this model is its memory footprint. While some MPI runtimes propose optimizations to reduce memory footprint in shared memory systems, one copy at least is required for communications inside a compute node. As available shared memory should continue to increase, it is necessary to exploit it efficiently. This trend advocates to mix MPI with a programming model relying on the shared memory paradigm. Several programming models suited for shared memory models have been presented, and do not require data duplications within compute nodes.

2.6 Hybrid programming models

We took into account the hardware trends in the field of High Performance computing, which are:

- the multiplication of cores inside a single compute node, which increases available shared memory.

- the hierarchical parallelism on current supercomputers

- the smaller and smaller availability of memory per compute core

On the one hand, these characteristics challenge numerical codes to scale on high performance machines. On the other hand, we showed that MPI model, due to its semantics, presented limitations and could not meet all these hardware constraints. Another solution is to mix MPI with another model able to efficiently exploit the shared memory.

We define Hybrid programming by combining several programming models with different paradigms: Message passing model mixed threads or mixed with tasks. Resources are then shared between the two models. Since MPI is the most popular and widely used in High Performance codes, it is considered as the first ingredient in Hybrid programming. The advantage of this solution is to overcome limitations of MPI by using a model adapted to shared memory systems. But even if the most intuitive solution is to use Message Passing model for inter-node communications and let the other one perform intra-node computations, we will see that it is not always the best solution. Thus, candidates for the additional are:

OpenMP, POSIX, Cilk, TBB or PGAS languages.

Combining MPI with OpenMP is interesting because both models are standards, and are supported in numerous runtimes. Moreover, OpenMP is the most elected model to tune MPI-based legacy codes. We will therefore focus on this model in this thesis.

2.7 Dissertation Outline

We established that combining MPI and OpenMP models was a promising solution to efficiently exploit current and future supercomputers. In this thesis, we will study hybrid MPI+OpenMP codes at several levels, from the application to the runtime levels, and via tools dedicated to performance analysis:

- In Chapter 2, we will present a taxonomy describing the different kinds of available combinations of MPI and OpenMP. We will then mention the bottlenecks preventing hybrid codes from reaching scalability.
- We will focus in Chapter 3 on the overhead encountered in OpenMP runtimes and then will present our first contribution, allowing to reduce this overhead in the context of hybrid programming, where compute resources are shared between MPI tasks and OpenMP threads.
- Chapter 4 will be dedicated to MPI Collective operations and how to optimize them in an hybrid context. In the second part, we will propose a concept unifying collective operations using both MPI and OpenMP.
- At last, we tackle the aspects of performance analysis on MPI+OpenMP codes in Chapter 5, and introduce OpenMP Tools API as last block to instrument hybrid codes.

Chapter 3

Focus on hybrid model MPI+OpenMP

In Chapter 2, we enumerated the limitations of MPI, and introduced hybrid MPI+X programming, studying several shared memory and task based models as candidates for X. In this chapter, we motivate the advent of MPI+OpenMP model as a solution for code scalability and study encountered problematics by combining MPI with the OpenMP model.

Following sections will describe advantages of this model, show how it is structured through an extended taxonomy, defining various code granularities and thread placements. We will also see that good cooperation between MPI and OpenMP is necessary inside runtimes to take advantage of hybrid programming. Then we will focus on encountered bottlenecks in MPI+OpenMP codes, preventing to obtain code scalability. At last, we will present contributions of this thesis, that aim at tackling enumerated bottlenecks.

3.1 Defining MPI+OpenMP programming model

MPI was traditionally adopted by scientists to parallelize numerical codes. Indeed, as a standard, it was the most obvious programming model to exploit distributed memory machines. But, as shared memory systems started to be disseminated on supercomputers, MPI more and more appeared to show limitations. So OpenMP was adopted to be mixed with it inside parallel codes, in order to leverage such shared memory systems. The principle is to start from MPI based codes and augment them by adding OpenMP constructs at specific points of the code. As a result, compute resources are shared between MPI tasks and OpenMP threads. However, several directions are available to mix both models. At first, choices have to be made about how to augment MPI codes with OpenMP, what we call code granularity: insert OpenMP on loops or parallelize larger portions. Then, we have to decide how resources are shared between MPI tasks and OpenMP threads (e.g placement). Even if reserving OpenMP for compute node and using MPI only for inter-node communications may be the most intuitive choice, it is not always the most best mixing. Do we reserve compute nodes to OpenMP and use MPI only for communications ? Is it interesting to use only MPI in some cases ? Tradeoffs have to be found, and several mixing flavors exist, which will be described below through an extended taxonomy.

3.2 Advantages and issues of hybrid MPI+OpenMP model

In this section we list the benefits we can get from switching pure MPI and Hybrid model, and the shortcomings of this switch.

3.2.1 Adequation between programming model and hardware

Current machines are no longer considered as distributed. They are composed of shared memory systems, and, with the emergence of NUMA architectures, exhibit hierarchical parallelism. MPI can be considered as a flat model: it is unaware of shared memory within compute nodes, and doesn't express

any hierarchical parallelism in its semantics, independently of its implementation. At user's level, it doesn't make difference between intra-node and inter-node communications.

The hybrid model, by adding a level of parallelism, provides a more efficient mapping of the underlying topology. Moreover, OpenMP, as a shared-memory model, allows avoiding intra-node communications.

3.2.2 Reduce memory footprint

Several researches showed that switching from pure MPI to Hybrid provided a memory gain between 80% to 480% with some benchmarks [8]. In [55], authors observed that, for a Real World application, it allowed to reduce memory footprint by 50% with 4 threads, and 60% with 8 threads, on 4,096 cores. By simply allowing to exploit shared memory, the use of the Hybrid model can significantly save memory, at different levels.

Halo cells. As seen in Chapter 1, MPI model, using the domain decomposition method, allows to parallelize computations as each MPI task computes its own sub-domain. But it requires exchanges of neighbor cells to perform computations, increasing memory footprint by a factor corresponding to the number of MPI tasks involved in the code execution. As memory is shared when using the OpenMP model, no neighbor cells need to be exchanged within an OpenMP team since domain decomposition doesn't apply. Figure 3.1 illustrates impacts of hybrid model on domain decomposition inside a compute node, by comparing behaviors of pure MPI code and Hybrid codes when exchanging halos. On the right part of the figure, we only have one MPI task per node, and other cores are fully populated by OpenMP threads. Thus, exchanges of halo cells are done only between compute nodes.

Internal structures. We saw that the MPI model generally needs to perform two memory copies when transmitting messages between tasks inside a node. Even if several MPI implementations proposed optimizations to reduce the number of copies, MPI imposed memory overhead due to its semantics. This problem is solved with OpenMP, where data are directly accessed with read and write operations. Also, some internal buffers and global constants implemented in MPI runtimes, that are duplicated for efficiency purposes, have a non negligible memory footprint as the number of tasks increases. Hence, reducing the number of MPI tasks allows to save memory footprint.

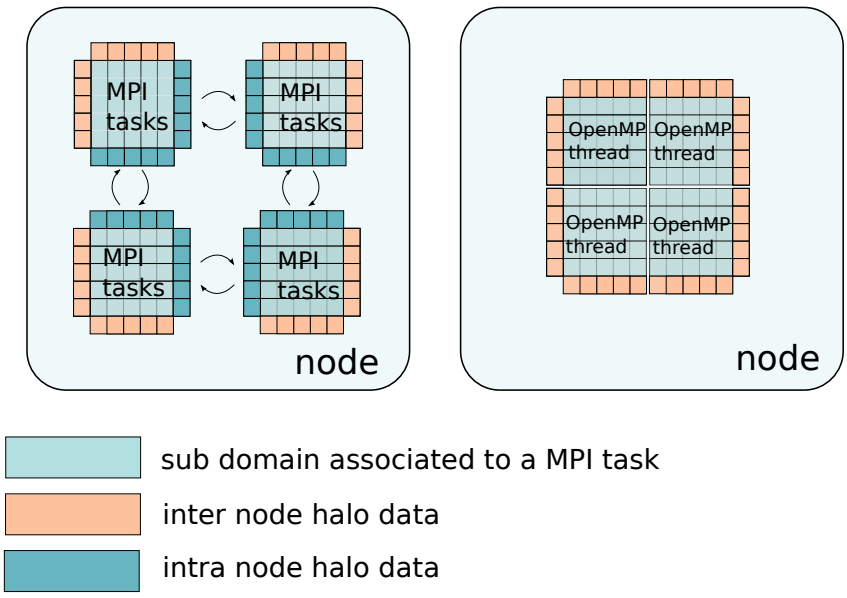


Figure 3.1: Memory representation of a pure MPI code and hybrid MPI+OpenMP code, using domain decomposition

3.2.3 Better load balancing

Load imbalance between MPI tasks can have several causes:

- Bad distribution of work between tasks
- Load imbalance generated by irregular applications, such `AMR` codes (Adaptive Mesh Refinement) [12]

MPI standard doesn't provide any easy interface to correct load imbalance, even if some techniques are possible at user's level. And manually correcting the load imbalance generates some communication overhead. OpenMP proposes an easy interface to correct load imbalance via scheduling policies coming with loop constructs, such as `dynamic` or `guided` [51]. Since OpenMP 3.0 version and the introduction of explicit tasks, load balancing can be done in a more dynamic way. But it is limited to compute nodes with any version of OpenMP.

3.2.4 Drawbacks

But augmenting MPI codes with a different programming model means adding a level of parallelism, and this implies difficulties.

First, as on pure MPI mode, all compute resources were busy during the whole execution time, provided that all the cores were fully populated with MPI tasks. But with hybrid programming, resources are shared between MPI tasks and OpenMP threads, consequently only a fraction of cores is used outside OpenMP parallel regions, depending on the mixing flavor. It happens for example when performing MPI communications, outside OpenMP constructs. This leads to a waste of computational resources.
Secondly, as less MPI tasks are involved in hybrid codes than with pure MPI applications, less internode communications are performed and then, the network bandwidth is under exploited.
Moreover, mixing different models with different paradigms is challenging for both MPI and OpenMP runtimes, and requires them to be aware of each other. It increases code complexity, and it is therefore very easy to introduce bugs. But Hybrid programming proposes different schemes or code granularities: codes can be augmented either by adding OpenMP constructs in an incremental way or using one single OpenMP parallel region, and go through a redesign of the application. Thus complexity of hybrid programming depends on the chosen code granularity.

3.3 Taxonomy of hybrid MPI+OpenMP model

Section 3.2 highlighted advantages and shortcomings of going hybrid for codes. Research related to MPI+OpenMP model introduced taxonomies describing a classification of MPI+OpenMP mixes, following two directions:

- Code granularity
- Thread Placement inside compute node

Figure 3.2, extracted from [24], illustrates an extended taxonomy for MPI+OpenMP applications. We describe in this section this taxonomy, following two main parameters, i.e. different code granularities and placements of MPI tasks and OpenMP threads.

3.3.1 Granularity of OpenMP

Starting from pure MPI codes, different ways exist to hybridize these codes.

Fine-grain parallelism. Fine-grain parallelism is the incremental approach when augmenting MPI codes [23]. It consists in parallelizing existing loops by using `pragma omp parallel for` construct, which combines `omp parallel` and `omp for` constructs. To get speedup with Fine-Grain parallelism, we have to identify hotspots, portions of the code with big execution time. This granularity allows hybridizing application without much increasing the code complexity.

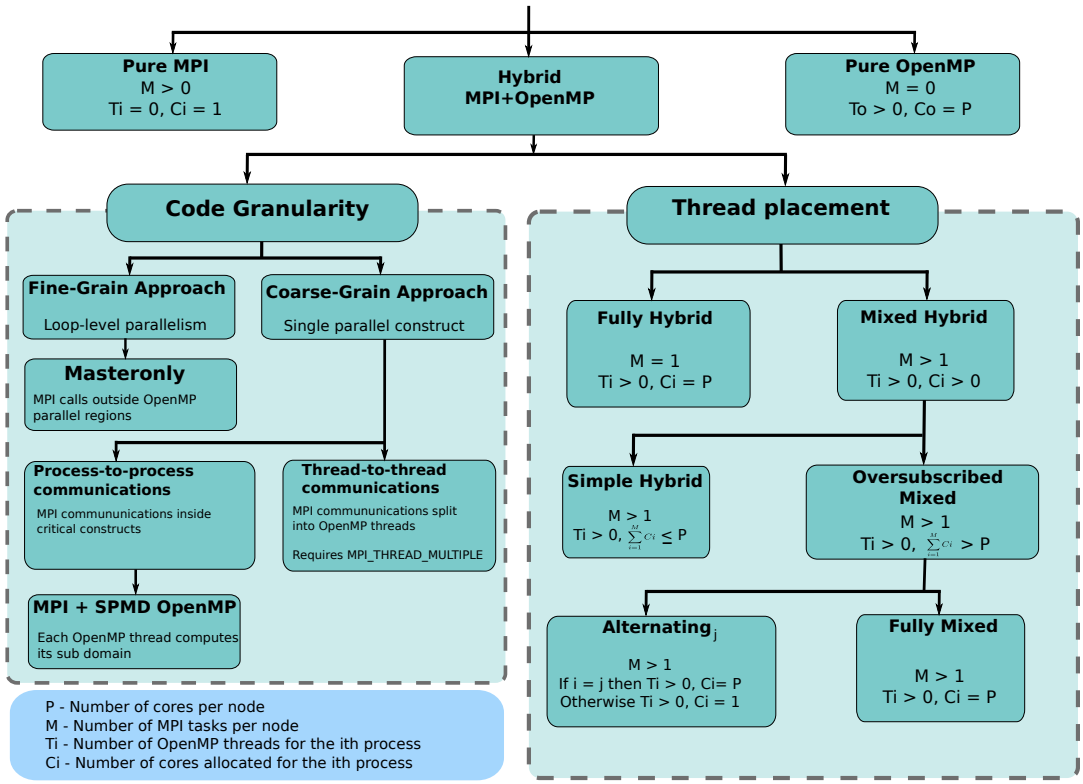


Figure 3.2: Hybrid MPI+OpenMP taxonomy

We can decompose execution time of a hybrid code into:

$$Ttot = Tseq + Tcomm + \frac{Tcompute}{P} \tag{3.1}$$

where $Ttot$, $Tseq$, $Tcomm$ and $Tcompute$ are respectively total execution time, sequential time, time spent in communications and time dedicated to computations. Fine-grain approach presents several limitations. Indeed, since the only component of $Ttot$ that can be accelerated is $Tcompute$, the maximum speedup to be obtained is

$$1 + \frac{tcompute}{tseq + tcomm} \tag{3.2}$$

So scalability of hybrid code with fine grain approach is limited. Secondly, this approach leads to a waste of computational resources, because when performing sequential code, cores dedicated to OpenMP are idle (Figure 3.3).

At last, as a big number of loops can exist in numerical codes, a huge number of OpenMP constructs are then requested. Thus, the high number of enters/exits inside OpenMP runtime may generate overhead due to the cost of activating and synchronizing threads. This overhead is added to sequential time, and at last, global execution time.

We can consider the *Masteronly* approach [51] as an extension of Fine-Grain approach, where MPI calls are done outside OpenMP constructs.

Coarse-grain parallelism. The other approach to hybridize MPI codes, Coarse-Grain, consists in opening one OpenMP parallel region at the beginning of the program, just after launching MPI tasks, and terminating it at the end of the application. MPI functions are then inserted inside this OpenMP parallel region. This approach overcomes some limitations of the Fine-Grain approach. At first, since OpenMP threads are executed during almost all execution time, sequential time is decreased, as it

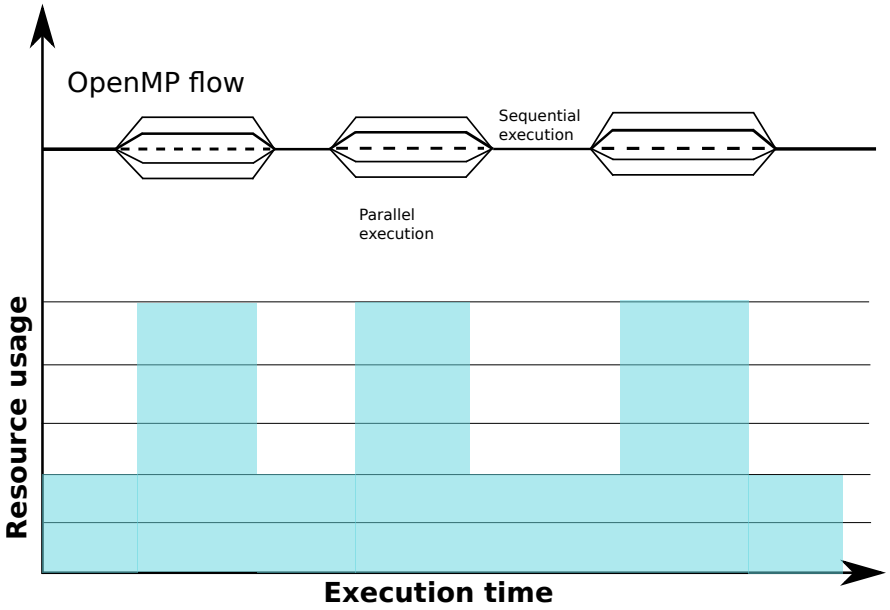


Figure 3.3: Execution timeline with Fine-Grain parallelism

is transferred to compute time. Secondly, with Coarse-Grain style, the code contains larger parallel portions and a better use of resources. At last, it leads to less calls to OpenMP layer and avoids overhead implied by Fork/Join mechanisms.

With the process-to-process configuration, since MPI primitives are called inside OpenMP parallel region, synchronization constructs such `omp single` or `omp master` are required to ensure they are called only once. But, by serializing calls to MPI primitives inside OpenMP parallel regions, we lose a level of parallelism, and the potential benefits of Coarse-grain approach are lowered. These constructs imply additional overhead due to those critical constructs. Moreover, since during these MPI calls one single thread is used, only a fraction of network bandwidth is used [90]. At last, these portions cannot be neglected since communications are performed in these critical sections. So the main limitation is $Tcomm$, referring to communications.

Another drawback of the Coarse-Grain approach is that it increases the code complexity, especially with thread-to-thread mode. It is no longer incremental, unlike the Fine-Grain approach, and implies redesigning the application.

Another issue is the sub-utilization of computational resources when MPI communications are performed.

A solution to reduce the ratio of $Tcomm$, which is proposed with the other variant under Coarse-grain approach, and is called thread-to-thread communications, consists in splitting off calls to MPI primitives using OpenMP threads. Depending on their rank, some OpenMP threads are used to perform MPI communications, and the others are dedicated to computations. An example of thread-to-thread variant is parallelizing exchanges of halo cells with threads. One constraint is that other OpenMP threads performing computations don't need halo data, since communications and computations are done in a parallel way. This variant allows to keep the cores active when performing MPI communications, and overlap communications with computations.

But as it performs concurrent calls to the MPI layer, it requires the highest level of multithreading support. And it implies a much higher complexity of the code as this variant is very intrusive.

MPI + SPMD OpenMP. SPMD[1] is a programming style achieving parallelism in a distributed manner, in that each task processes its own data independently. It can be enabled with OpenMP but requires modifications from a classic OpenMP program. With SPMD, work is distributed at hand between OpenMP threads, and each one computes on its sub-domain. As loops are generally targeted to enable parallelism, they drive the translation into SPMD style. Unlike common OpenMP codes where arrays are shared between threads and accessed in a concurrent way, arrays are privatized with SPMD programming and split into sub-parts, and computations are spread among threads, in order to maximize data locality [72]. Additional structures are inserted when results need to be shared.

OpenMP behaves here as a distributed memory model, like MPI.

Let's take a basic example of a loop to be parallelized.

```
int main(int argc, char **argv)
{

 int iter;

 /* Iteration loop */
 for( iter = 0 ; iter < niter ; iter++)
 {

 /* Code */

   for( i = 0 ; i < n ; i++)
   {
     B[i] = B[i] * a + niter;
   }

 /* Code */

 }

}
```

```
int main(int argc, char **argv)
{

 int iter;

 /* Iteration loop */
 for( iter = 0 ; iter < niter ; iter++)
 {

 /* Code */

#pragma omp parallel for schedule(runtime)
   for( i = 0 ; i < n ; i++)
   {
     B[i] = B[i] * a + niter;
   }

 /* Code */

 }

}
```

```
int main(int argc, char **argv)
{

 int iter;

 int myOMPRank = omp_get_thread_num();
 int nbOMPThreads = omp_get_num_threads();
 int nbnLoc = n / nbOMPThreads;
 int iDeb = 1 + myOMPRank*nbnLoc;
 int iFin = iDeb + nbnLoc - 1;

 if(myOMPRank == nbOMPThreads - 1)
   iFin = n;

 /* Iteration loop */
 for( iter = 0 ; iter < niter ; iter++)
 {

 /* Code */

   for( i = iDeb ; i < iFin ; i++)
   {
     B[i] = B[i] * a + niter;
   }

 /* Code */

 }

}
```

Figure 3.4: Comparison between classic OpenMP and SPMD

[1] Single Program Multiple Data

The second version of the example shows how the loop is split between OpenMP threads using the proper directive. The last version, called SPMD OpenMP, performs a manual distribution of the code to the threads. For this purpose, each OpenMP rank is computed, then lower and upper bounds are deduced following each OpenMP rank. It is to be noted first, that other implementations exist to achieve parallelism in an SPMD way, second, that is it requires to completely redesign code as this mode is very intrusive.

With SPMD style, iterations are manually split between threads. SPMD programming has been studied in literature. In [65], authors present a SPMD version of NAS benchmarks, derived from the MPI version. Experiments show this approach gives better performances.

For example, reduction feature cannot be used any more to reduce a variable inside a loop. This operation has to be done manually.

If we come back to our taxonomy, MPI can be coupled with SPMD OpenMP programming. We consider this combination as a particular case of Coarse-Grain approach with thread-to-thread version. The difference is that OpenMP is used as a distributed model. In this mode, distribution of work can be seen as nested: domain is split between MPI tasks, and sub-domains are again spread among OpenMP threads. Then both models enable a same paradigm consisting in distributed memory style, in a hierarchical level.
While high performance can be reached using this flavor, it is the most complex to implement.

3.3.2 Thread Placement

Beside code granularity, Placement is an important factor in MPI+OpenMP taxonomy. We describe here the different mix between MPI tasks and OpenMP threads inside a compute node, their advantages and limitations regarding code performance and resource exploitation.

As depicted in Figure 3.2 and in order to formalize the different combinations, number of MPI tasks, OpenMP threads are identified with symbols. *M* refers to the number of MPI tasks, *Ti* the number of OpenMP threads, and *Ci* the number of cores allocated for the *ith* process.

The first mode *Fully Hybrid* exploits only one MPI task per node ($M = 1$). The other cores of the node are populated with OpenMP threads. It only exploits MPI for inter-node communications, reserving all node for computations. *Fully Hybrid* mode implies limitations in that network cannot be saturated with one single MPI task per node. With *Mixed Hybrid* mode, several MPI tasks are launched inside compute node ($M > 1$), with one OpenMP team per task. This leads to a better use of network bandwidth. *Simple Mixed* is derived from *Mixed Hybrid* as cores inside compute node are shared between MPI tasks and OpenMP threads, and are fully populated.
It is also possible to oversubscribe resources with *Oversubscribed Mixed* mode. In *Alternating* style, cores are used either by MPI tasks or by OpenMP threads, depending on MPI rank. With *Fully Mixed* combination, all cores of the node are reserved for each MPI task. A single core can be shared by several OpenMP threads.
Figure 3.5 exposes a hardware view of different hybrid combinations.

3.3.3 Overlapping between communications and computations

Keeping all the compute resources active along the code execution is critical for code scalability. But we saw that exploiting all cores at any time was complex due to the different paradigms involving MPI and OpenMP. For example, *Masteronly* approach implies a sub-utilization of resources outside OpenMP parallel regions and a sub-utilization of network bandwidth. Moreover, when MPI communications are performed (e.g with MPI Collective), only a subset of cores is involved, letting other cores idle. This type of MPI operation is called Blocking Collective as code execution is suspended until termination of communications. However, MPI-3 standard revision introduced non blocking Collectives, where the execution can continue before communications have terminated [52].

We talk about overlapping when MPI communications are recovered with computations [51], reducing $Tcomm$ component of the total execution time. Exposed taxonomy showed different ways to overlap communications. First, splitting off communications with OpenMP threads allowed communications and computations to progress in a parallel way. Secondly, oversubscribing resources with OpenMP threads or using the core either for MPI tasks or threads would ensure that no core remains idle.

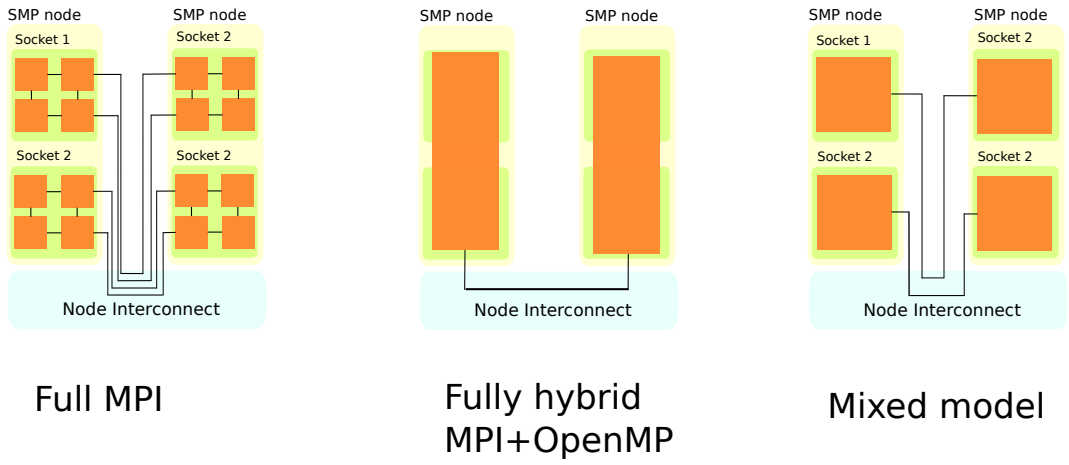


Figure 3.5: Thread placement of hybrid programming model

3.4 Requirements of MPI runtimes

With our extended taxonomy, we introduced various schemes with different code granularities and ways to share resources between MPI and OpenMP. But mixing such models with different paradigms is not only tricky at user's level but also requests features from the runtimes. On the one hand, performing concurrent calls to MPI layer implies the underlying runtime to be thread safe. But several levels of thread safety are available. On the other hand, the placement of MPI tasks and OpenMP threads requests good cooperation between both runtimes.

3.4.1 Thread support

MPI standard defined different levels of multithreading support which can be enabled when initializing MPI. These levels are described as follows, from the most restrictive to the most permissive:

- `MPI_THREAD_SINGLE`: A single thread is allowed per MPI process
- `MPI_THREAD_FUNNELED`: Only the process that initialized can perform MPI calls
- `MPI_THREAD_SERIALIZE`: A process can be multi-threaded, but only one thread at a time can call MPI
- `MPI_THREAD_MULTIPLE`: A process can be multi-threaded, and multiple threads can simultaneously call MPI functions. It is the MPI runtime responsibility to synchronize concurrent calls to MPI functions

`MPI_THREAD_MULTIPLE` is the most constraining level in term of thread safety. This level is required in mixing modes such as thread-to-thread communications in Coarse-Grain approach. Indeed, as MPI primitives are inserted in OpenMP constructs, this scheme implies concurrent calls to MPI layer, whereas process-to-process variant only needs `MPI_THREAD_FUNNELED` level.
But to support this multithreading level, MPI runtimes need to be tuned, and ensuring thread safety at `MPI_THREAD_MULTIPLE` level inside MPI runtime is not free as it requires considering many parts of the runtime [49]. Moreover, this mode adds some overhead [101]. Several MPI implementations support `MPI_THREAD_MULTIPLE` level such IntelMPI, MPC, or MPICH2 [48].

3.4.2 Interoperability between MPI and OpenMP

Exposed taxonomy highlighted different challenges when switching from pure MPI to Hybrid model. Sharing resources between both models covering all mixing combinations described on the one hand, and spawning parallel regions on the other hand, requires a smart cooperation between both models.

A first requirement is to cover all hybrid combinations described in placement part of taxonomy, including oversubscribing (i.e. share same cores between both models). Secondly, placement of MPI tasks and OpenMP threads has to be done following hardware topology. At last, a good cooperation is required in order to ensure fair CPU share between both models.

Topology view for thread placement. To efficiently share compute resources between MPI and OpenMP, a careful distribution of MPI tasks and OpenMP threads is necessary.

This placement requires that both models have a common view of underlying topology. But when MPI and OpenMP runtimes are distinct, each model has its own view of hardware topology, and is not aware of the other model. This case is likely to give a sub-optimal placement.

With MPC, MPI and OpenMP are implemented inside the same runtime, ensuring a good cooperation between them. MPI tasks are distributed in a sparse manner, keeping each task as far as possible from its neighbors, and reserving a set of cores for it. Then, as a task spawned a parallel region, the set of cores is populated with OpenMP threads related to this task, threads being distributed following a compact strategy. If, for example, we execute a code launching 4 MPI tasks on a machine equipped with 4 sockets (i.e. NUMA nodes), each task will be spawned on a socket, and each task will spawn its OpenMP threads inside the socket. This set, gathering the task and its OpenMP team, is called Hybrid instance.

Policies for thread placement can be chosen with each programming model. But the model on the top is not aware of policy chosen by the underlying model.

Cooperation between MPI and OpenMP. Since OpenMP is the top model in runtime stacking, MPI is in charge of activating its related OpenMP team. For a complete integration, MPI has to launch OpenMP threads in an efficient way. So optimizations have been done in MPC framework to improve spawn of OpenMP threads. Thus, a polling method has been implemented, which consists for OpenMP threads to wait for a value to be updated before starting their execution.

3.5 Identify bottlenecks in hybrid programming

The study of the Hybrid model and its various combinations through taxonomy helped us to identify its strengths and weaknesses. We list here the different bottlenecks that can prevent from good scalability of MPI+OpenMP codes.

3.5.1 Overhead of OpenMP runtimes

We recall here how overhead can be encountered in OpenMP runtimes, illustrated by Figure 3.6. OpenMP runtime is called when an OpenMP directive, associated to a parallel region, is encountered. The runtime is exited when the code associated to the parallel region has finished to be executed. But differences have to be made between the execution time of the runtime and the execution time of parallel threads execute their tasks. The reason is that, while executing the parallel region, the Master thread has to create itself other threads and these have to synchronize before they are terminated. These operations, which are know as Fork and Join mechanisms, need time to be executed. This time corresponds to the overhead, and can be added to sequential time.

This overhead is amplified in Fine-Grain approach due to frequent calls to the OpenMP layer.

We illustrate this point in Figure 3.7, showing a code as a simplified example of Fine grained code with loop level parallelism. This example allows us to understand why reducing overhead of OpenMP is critical to reach performances. We can see that, after initializing MPI tasks, we encounter an iteration loop to ensure the progression of simulation. Within the iteration, we have another loop parallelized

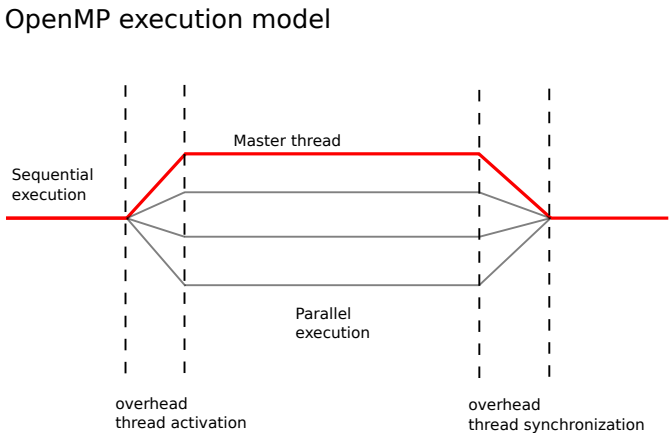


Figure 3.6: Overhead of OpenMP model

```
int main(int argc, char **argv)
{

 MPI_Init(&argc, &argv);

 /* Code */

 /* Iteration loop */
 for( i=0 ; i<niter ; i++)
 {

 /* Code */

#pragma omp parallel for schedule(runtime)
    for( j=0 ; j<n ; j++)
    {
      C[j] = A[j] + B[j];
    }

 /* Code */

 }

 MPI_Finalize();

}
```

Figure 3.7: OpenMP overhead with Fine-Grain code

with OpenMP construct. Consequently, OpenMP runtime will be entered and exited `niter` times, and the generated overhead will be as important as `niter` is high. Since there is only few room to optimize compute time, it is critical to maintain it as minimal as possible. Indeed, this overhead can be considered as additional sequential time. But an overhead can also be observed in Coarse-Grain approach, as OpenMP constructs are required to maintain memory coherency of MPI communications. Frequent calls to OpenMP layer performed in Fine-Grain approach lead us to study the performance of OpenMP runtimes, especially for Fork/Join mechanisms.

Performance penalties due to NUMA environment. NUMA architectures are now available on current supercomputers and exhibit hierarchical topology. Since OpenMP threads need to communicate at some points of the program such as the beginning of parallel region or the synchronization points, NUMA effects can be encountered when threads are not located on a same socket. These effects can introduce additional overhead to OpenMP runtimes, this is why the underlying topology has to be taken into account. As mentioned previously, the need of efficiency of OpenMP, especially when entering and exiting parallel regions, implies to adapt algorithms implementing those operations to hierarchical parallelism exposed by such memory organizations.

Space dedicated to OpenMP vary. With the Hybrid Programming, we introduced the Thread Placement, providing a formalism defining how computational resources inside a compute node can be shared between MPI tasks and OpenMP threads. For example, depending on Placement and on the number of MPI tasks inside compute node, the available cores for each OpenMP team will vary, from entire node to only a socket. So mechanisms for thread management have to be implemented in a flexible way to keep minimal overhead, whatever the number of allocated cores.

3.5.2 Lack of parallelism and resource usage

The study of this taxonomy showed that sequential part of the code (i.e. outside OpenMP constructs) and communications may represent an important part of the execution time. This shortcoming especially appears with the *Masteronly* approach, where the code is sequential outside OpenMP parallel regions, meaning only MPI applies. We saw communications represented a non negligible percentage of this part. This shortcoming was highlighted in Fine-Grain approach, but also with Coarse-Grain scheme, as MPI communications are serialized. A code with an important sequential part present several issues:

- use of only a subset of resources
- limited speedup and scalability of codes

We have to look for optimizations in order to reuse idle cores. We can tackle this bottleneck by optimizing the sequential part as much as possible and focusing on MPI primitives (e.g. Collective operations).

3.5.3 Complexity of Hybrid MPI+OpenMP codes

MPI and OpenMP present different paradigms and different interfaces: MPI provides a set of functions to perform communications whereas OpenMP comes with a set of directives for work distribution or synchronization. Then mixing both models is likely to complexify code and introduce bugs. The cost of having an optimized code, using all resources as much as possible, is heavy. The best example is the thread-to-thread communications mode, where MPI and OpenMP are strongly interleaved. These difficulties are real obstacles to going Hybrid. Yet, OpenMP and MPI propose similar features such as synchronization and reduction, but these are provided with different semantics.

Also, at runtime level, in MPC framework MPI and OpenMP implementations interact for launching OpenMP threads, but no cooperation is done for specific features such as synchronization or reduction.

Mix MPI and SPMD OpenMP. SPMD programming style used with OpenMP has been presented as behaving like a distributed memory model. Then, when coupling SPMD with MPI, programmer had to deal with 2 nested distributed memory models and a hierarchical domain decomposition. But no cooperation was done between MPI and OpenMP for the case when tasks of the application need to be synchronized.

3.5.4 Performance analysis of MPI+OpenMP codes

Ensuring scalability of parallel codes is a difficult task and it requires identifying potential performance issues. This task is made even more complex as these issues can come from the application or the runtime. We also highlighted how mixing two models with different paradigms can introduce bugs and bottlenecks. To reach high performance with parallel codes, tools are often required in order to detect bottlenecks, via different kinds of performance analysis such as profiling or tracing. But those tools rely on interfaces to be plugged to programming models. Some interfaces come with MPI such as PMPI [3]. Starting from MPI codes and hybridizing them can lead to unexpected pitfalls, or performances can be lower than expected. Detecting such pitfalls can be cumbersome, so the performance analysis of hybrid codes is even more necessary. But some blocks miss to ensure good profiling of such codes, especially interfaces dedicated to OpenMP. The performance analysis of hybrid codes can be very complex, and with the increase of code complexity, tools are necessary to profile both models.

MPI standard comes with several APIs such PMPI for performance analysis. But there is a lack of standard tool coming with OpenMP.

3.6 Thesis contributions

Section 3.5 helped identifying bottlenecks related to hybrid model that may prevent from scalability and viability of hybrid codes. We introduce here our contributions, aiming at tackling the previously described difficulties. The goal of this thesis will be to solve bottlenecks of the studied programming styles, chosen by the programmer.

3.6.1 Reduce overhead of OpenMP runtimes in the context of fine grain parallelism in MPI+OpenMP model

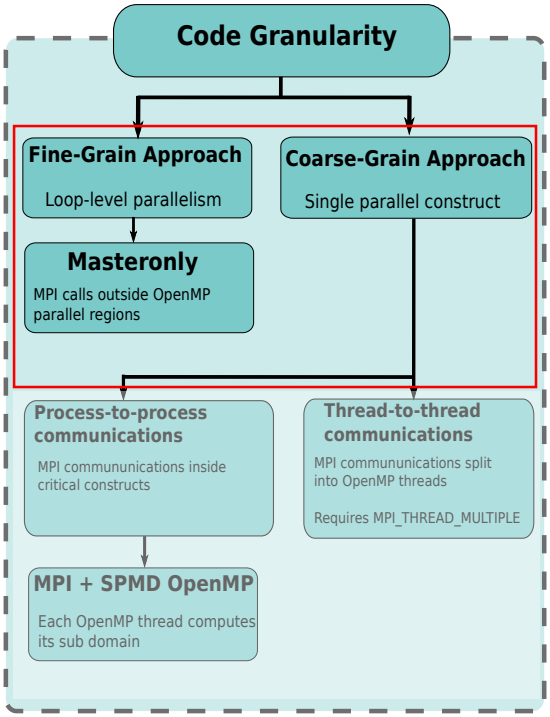


Figure 3.8: Targeted modes for reducing overhead of OpenMP

We saw in Section 3.5 that the OpenMP layer was likely to be frequently accessed in Fine-Grain mode, generating overhead. But as synchronization constructs are also required in Coarse-Grain style, optimization of the OpenMP layer is also desirable in this case. We introduce here our first contribution, tackling one of the issues encountered with Fine-Grain parallelism and, to a lesser extend, with Coarse-Grain, as highlighted on Figure 3.8: the optimization of OpenMP runtimes, focusing on the creation of threads and their synchronization. In order to reduce this overhead, we describe in the next chapter a new design of an OpenMP runtime meeting constraints encountered on MPI+OpenMP codes, i.e the need of performance on a large spectrum of threads. Then, we describe challenges raised by NUMA architectures, and we show how our design meets these challenges by relying on a tree structure in order to fit underlying topology, and to use it, for example, for thread activation and synchronization. At last, we introduce our contribution, the *Adaptive tree*, which is a tree whose shape automatically

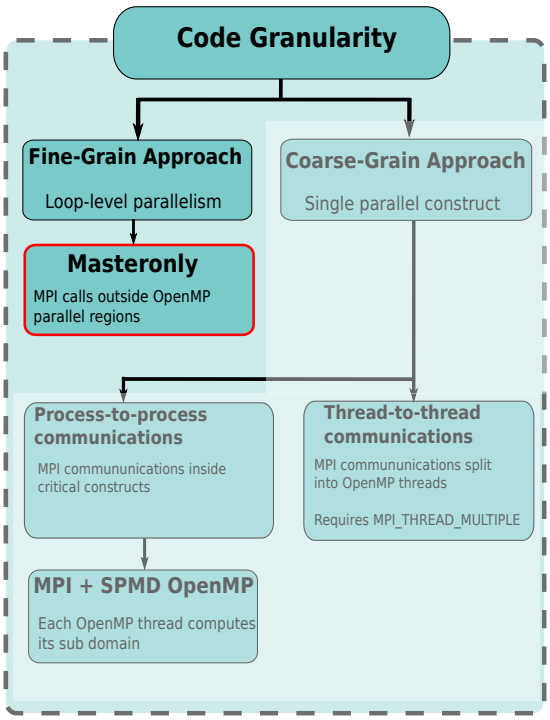


Figure 3.9: Targeted modes for hybridizing MPI collectives

changes depending on the number of launched threads in OpenMP constructs.

3.6.2 Study Collective operations in a hybrid context

In Chapter 5, we propose a global study about MPI collective operations, targeting several bottlenecks.

Hybridization of Collective operations. In our second contribution, we target the problem of sequential component of the total execution time and we search for a better use of compute resources during this time. This problem is especially encountered with *Masteronly* approach (Figure 3.9). So, we analyze what parts of sequential component can be optimized, which leads to focus on MPI collectives. We then investigate literature about optimization of MPI Collectives, through multiple directions: rely either on shared memory or on network topology. We also study different kinds of MPI collectives through their characteristics and discuss how we can hybridize them with OpenMP threads. At last, our contribution is focused on `MPI_Allreduce` and we propose a portable way to optimize this collective with the help of OpenMP threads, and using different algorithms.

Introduce unified collectives. In the third contribution, we tackle the problem of code complexity when mixing models with different semantics. We target Coarse Grain approach (Figure 3.10), and give guidelines to simplify semantics of the Hybrid model, especially when performing MPI communications in a thread safe way.

In the case of MPI + SPMD pattern, we introduce collectives unifying both models, such as reduction. A global barrier is presented as a proof of concept, synchronizing both MPI tasks and OpenMP threads.

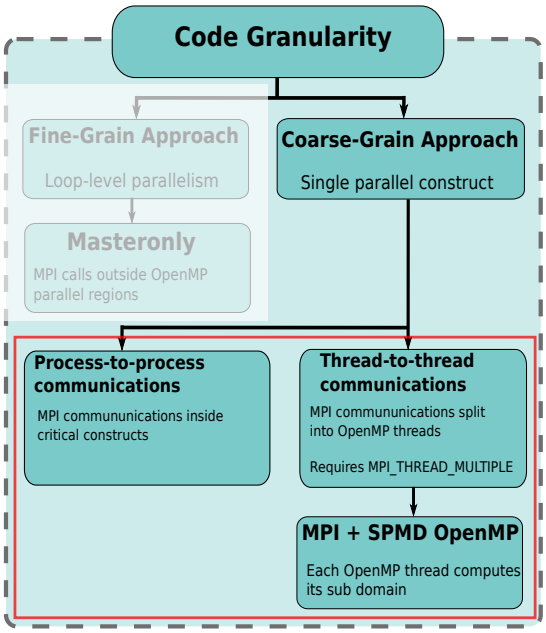


Figure 3.10: Targeted modes for unified collectives

3.6.3 Standard performance analysis of OpenMP codes

In the last part of this thesis, we study eligible tools for performance analysis targeting OpenMP codes. We then describe the newly introduced OpenMP Tools API (OMPT [54]): this describes an interface compatible with the OpenMP standard, and is dedicated to performance analysis. This API provides a set of events dedicated to OpenMP constructs and is to be interfaced between OpenMP runtime and tools for performance analysis. It covers all OpenMP constructs and can be used to identify bottlenecks in OpenMP codes and runtimes. We consider this interface as the last arrow to enable a good profiling of hybrid codes. Then, we describe the implementation of this API into MPC framework.
At last, we study performance analysis using our implementation of OMPT, oriented for both runtimes and hybrid codes, and with the help of TAU, a tool dedicated to performance analysis.

Part II

Contributions

Chapter 4

OpenMP runtimes for MPI+OpenMP applications

Due to its Fork/Join mechanisms, OpenMP implies overheads when starting and terminating threads. The reason is that threads have to be created and activated for parallel execution. Regarding these considerations, runtimes implementing OpenMP model propose optimizations to minimize these overheads as much as possible. In this chapter, we expose on the one hand constraints on OpenMP runtimes implied in hybrid applications, and on the other hand challenges encountered by NUMA systems on OpenMP constructs. Then we describe a NUMA-aware design of our OpenMP runtime, and subsequently our contribution.

4.1 Constraints on OpenMP runtime in hybrid MPI+OpenMP codes

As the top model in hybrid MPI+OpenMP codes, OpenMP is likely to be frequently entered and exited. Moreover, the taxonomy related to hybrid MPI+OpenMP described in last chapter exposes different mixings between MPI and OpenMP, and different code granularities. Constraints on OpenMP runtime in hybrid context are then twofold:

- OpenMP constructs are frequently called
- Space for OpenMP vary among compute node

These constraints can consequently lead to non negligible overheads which can impact performances of applications. They will be detailed in the following sub-sections.

4.1.1 Need of efficient mechanisms for OpenMP constructs

The taxonomy related to hybrid MPI+OpenMP exposed different granularities when mixing MPI and OpenMP. Depending on these granularities, OpenMP runtime will be stressed in different manners. With fine-grain parallelism, code is likely to perform frequent calls to constructs such as `omp parallel region` and its combined version `omp parallel for`. In this configuration, it is critical to be able to enter and exit OpenMP layer on a fast manner. Moreover, efforts have to be focused on threads activation and synchronization when designing OpenMP runtimes. These constraints advocate for a lightweight runtime.
Since we encounter large parallel regions with coarse grain parallelism, we have a lower number of threads activations. We put the stress on constructs inside OpenMP parallel region, such as OpenMP loops, `barrier` or `single` constructs.
Moreover, in oversubscribing configurations, cores are shared between OpenMP threads, or between MPI tasks and OpenMP theads. We then need to be able to perform fast context switches between threads.

4.1.2 Need of a flexible runtime

In chapter 2, several levels of thread placement were described. For example, in `mixed hybrid` configuration, compute node is shared between OpenMP teams. Thus, depending on granularity of thread placement, space available for OpenMP can vary, from the entire node to a socket. These constraints have to be taken into account by OpenMP runtimes and advocate for efficient runtimes for only few threads and numerous threads. These considerations lead to design runtimes allowing flexible handling of threads.

4.2 Impacts of NUMA effects on OpenMP runtimes

Even if, in a hardware point of view, memory units are separated and dedicated to a processor, ccNUMA architectures are considered as shared memory systems at user's level. However, due to their characteristics, inter-sockets communications are expensive in terms of latency. Such latencies are called NUMA effects. These NUMA effects can be encountered when threads located on different NUMA nodes need to interact, or if a thread has to access data on a remote node. To avoid these penalties, a special care has to be taken in the area where data are accessed. We call this data locality. With the problem of data locality comes the problem of memory affinity, i.e. the relationship between threads and data. Memory affinity is enabled when threads and related data are located on the same NUMA node, and thus remote accesses are avoided. Taking NUMA characteristics and data locality into account is critical for scalability of OpenMP codes, and consequently MPI+OpenMP codes. Then, we have to keep in mind NUMA effects and data locality when designing OpenMP runtimes.
A flat representation of threads is no longer sufficient to leverage such architectures. We illustrate the problem of threads interactions by taking the example of basic operations in OpenMP such as threads activation and thread synchronization in the following sub-section.

Impact on OpenMP constructs: thread activation and thread synchronization The basic operations with OpenMP are to activate threads when beginning a parallel region and to synchronize them on a barrier (Fork and Join mechanisms). A classic implementation of thread activation is, for the thread Master (thread of rank 0), to wake itself the other OpenMP threads.

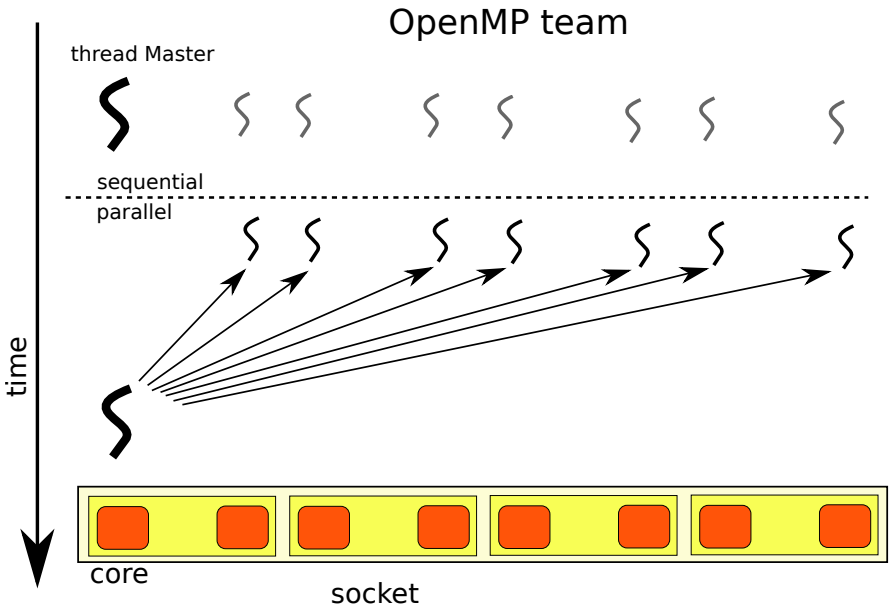


Figure 4.1: Centralized approach for thread activation

We can call this implementation a centralized approach (Figure 4.1). We consider OpenMP threads are equally dispatched on all available cores on the compute node, among all NUMA nodes. If we can assume that activating threads is only fast if all threads are on the same NUMA node, then, activating threads dispatched on all NUMA nodes can be time consuming. The explanation is that the Master thread has to flip variables to launch execution of the entire OpenMP team. Considering that the variables relative to each thread are located on memory banks near their respective NUMA nodes, master thread will have to perform remote access, encountering NUMA penalties. The centralized approach relies on the fact that a single thread is in charge of the entire work of thread activation, which leads to contention. Moreover, by ignoring data locality previously presented, NUMA effects are likely to be encountered, and additional overheads are encountered.

The same problem occurs for thread synchronization. With the centralized approach, master thread increments a counter as threads reach the barrier. When all threads have reached the barrier, the Master thread sets a flag and all threads are free to continue their execution.

Problem of memory affinity Beside the problem of threads interaction encountered with OpenMP constructs, a thread also interacts with data during its execution, for example accessing and writing an array. If we assume that data are allocated near threads (on the same node) when starting the application, thread placement may evolve during execution. This case is encountered with irregular applications such as AMR codes(`Adaptive Mesh Refinement`), in which the memory pattern evolves.

4.3 Related Work

Lightweight OpenMP runtimes. Hybrid codes are designed in a way parallel regions can often be entered and exited, or cores are shared between OpenMP threads. So there is a need to create threads that are able to quickly start their execution. This leads to study contributions proposing light implementations of OpenMP threads. [38] describes a new kind of threads named Filaments, supporting efficient execution of fine grained codes. In [24], authors introduce, inside the runtime MPC (Multi Processor Computing), a new kind of OpenMP threads named `microVP` (micro Virtual Processor). These microVPs schedule their own `microthreads`, which can be considered as lazy threads. This implementation allows faster context switches, since `microthreads` don't have their own stack but microVP ones. Stacks are created on-the-fly at scheduling points (such as barriers). This design achieves a lightweight implementation of OpenMP runtime and ensures good integration within MPI+OpenMP context.

Thread handling and placement in NUMA environment. Some papers focused on optimizing main OpenMP constructs. [86] proposes several implementations of the OpenMP barrier in the OpenUH compiler, describing several algorithms, such as tree, dissemination, or tournament. Experiments revealed that tournament ensured best scalability of the OpenMP barrier.

In [20], authors present ForestGOMP, an OpenMP runtime which extends the libGOMP library, and propose optimizations for NUMA environments. Indeed, it introduces a hierarchical representation of threads, via a component named BubbleSched, which gathers threads belonging to an OpenMP team in bubbles, and dispatchs them among underlying topology. To target irregular applications, some work stealing algorithms are also proposed to dynamically move threads from a core to another, for load balancing purposes.

Memory Affinity In order to control memory affinity between threads and data, ForestGOMP also proposes data migration via its memory manager MaMI[1]. This component proposes new memory policies like `next-touch`, which is a generalization of `first-touch` memory policy: when a thread accesses data on a remote node, data are moved inside the memory system near the thread.

Some other contributions also focus on how to ensure memory affinity in a NUMA context and for OpenMP codes. In [94], authors present Minas, a framework providing explicit or automatic tuning to control memory affinity. Minas contains several components for ensuring memory affinity:

[1] Marcel Memory Interface

- MAi (Memory Affinity interface) is an API providing data allocation and placement. It implements some memory policies following different data distributions (bind, cyclic and random). Minas first presents alternative memory pattern policies than `first-touch` policy, implemented in Linux systems: `next-touch` policy, which allows data movement to close threads. These memory policies deal with both regular and irregular applications.
- MApp is a preprocessor performing source-to-source transformations to insert specific functions belonging to MAi API.
- A third component named Numarch retrieves machine informations, ensuring Minas to offer hardware abstraction and code portability.

4.4 NUMA aware runtime

The characteristics of NUMA architectures and impacts on OpenMP runtime described in the previous section lead us to design a hierarchical representation of threads inside OpenMP runtimes. This work has been implemented in MPC framework and is built on top of the lightweight implementation of OpenMP threads mentioned in [24]. It can also be compared to the contribution made with Forest-GOMP, as it also relies on a hierarchical structure. However, as we will see, it is also oriented towards Hybrid context where MPI and OpenMP are mixed.

4.4.1 Hierarchical tree

Our hierarchical representation of threads consists of a balanced tree, containing a root, nodes and leaves. Threads will be mapped on this tree. To illustrate our design, we can take the example of a machine equipped with 4 processors, each containing 8 cores and coming with its memory bank. This example is depicted on Figure 4.2. If we build a tree following hardware topology, it will consist in 4 nodes, each node representing a NUMA node, and having 8 leaves in its sub-tree. The tree has a total of 32 leaves. Let's then consider an OpenMP team of 32 threads, fully populating the machine, and let's see how these threads are mapped on this tree. The Master thread (thread of rank 0) will be placed on the tree root, on the top left node and top left leaf. Threads of rank 8, 16, and 24 will be placed on the other nodes and top left leaves inside their respective sub-trees.
The Hierarchical data structure allows to delegate part of the activation and synchronization work to those threads located on intermediate nodes. We describe these mechanisms in the following subsections.

4.4.2 Application to thread activation and thread synchronization

Whereas the centralized approach previously described consisted in having all the OpenMP threads activated by the Master thread, our approach allows to delegate parts of the thread activation, as illustrated in Figure 4.2.

Algorithm 2 Hierarchical thread activation

Require: $tree$
1: $node \leftarrow tree.root$
2: **while** $node \neq LEAF$ **do**
3: wait on $node.exec = 1$
4: **for** $i := 0$ **to** $node.nbchildren$ **do**
5: **if** $i > 0$ **then**
6: $node.children[i].exec \leftarrow 1$
7: **end if**
8: **end for**
9: $node \leftarrow node.children[0]$
10: **end while**

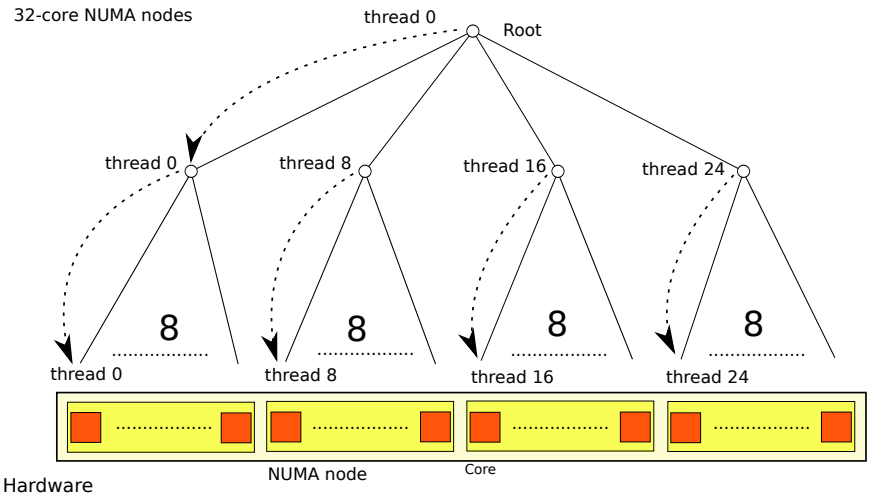


Figure 4.2: Hierarchical thread activation

The principle of thread activation using hierarchical tree, depicted in Algorithm 2, operates as follows: the thread of rank 0 is waiting on the tree root and wakes the threads located on children nodes. Once awaken, these threads are polling on their nodes and take in charge the activation of threads located on their children nodes. This operation is repeated in a recursive way until reaching the tree leaves. As soon as a thread is activated, it starts its execution. As depicted on Figure 4.2, the Master thread (ranked 0) is waiting on the tree root as well as on the left intermediate node, and on the top left leaf. In a similar manner, the threads located on other internal nodes are also broadcasted on each left leaf of respective sub-trees.

We detail the process of thread activation by following its different steps. For better understanding, nodes where threads are waiting for execution are colored in red and those where threads are executing or activating other threads are colored in green.

Figure 4.3 describes the first step of thread activation. We start from the root of the tree with the thread ranked 0. The first job of thread 0 is to wake its children which are thread 8, thread 16 and thread 24. The wakening operation consists in updating value of a flag to 1 in order to inform children threads to start their execution.

Once the threads located on children nodes are awaken, they wake the threads located on the leaves, in their own sub-trees. This is the second step of thread activation process (Figure 4.4. While threads located on intermediate leaves activates their children leaves, the Master thread located on the tree root, switches location from the root to the node located on its left.

Figure 4.5 describes the third step of Fork operation. Once threads ranked 8, 16 and 24 have finished to activate the threads on leaves, they can switch from intermediate nodes to the top left leaves of each sub-tree. The activated children threads, as soon as they get their flag turned on, immediately start their execution. Meanwhile, the Master thread iterates through its children inside the first sub-tree starting from the left, and wakes them.

When the Master thread has finished activating all its children, it switches from top left intermediate node to the top left leaf and executes himself (Figure 4.6).

Algorithm 3 describes the implementation of the OpenMP barrier using a hierarchical tree. Each node of the tree has an intermediate barrier to synchronize threads belonging to its sub-tree. To synchronize all threads relative to its sub-tree, the node contains a counter `barrier` allowing to know how many of its threads have reached this barrier. Each time a thread reaches the barrier, it increments the counter, until its value equals the number of expected threads stored in the sub-tree. We have to make sure this counter is incremented in an atomic way since it is accessed in a concurrent way. When the expected number of threads has reached the intermediate barrier, a flag related to this node named

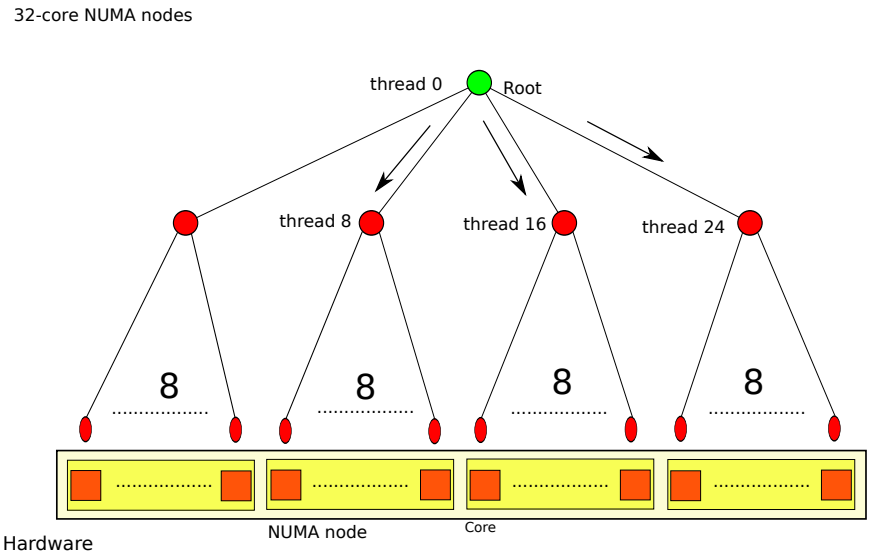


Figure 4.3: Hierarchical thread activation - 1st stage

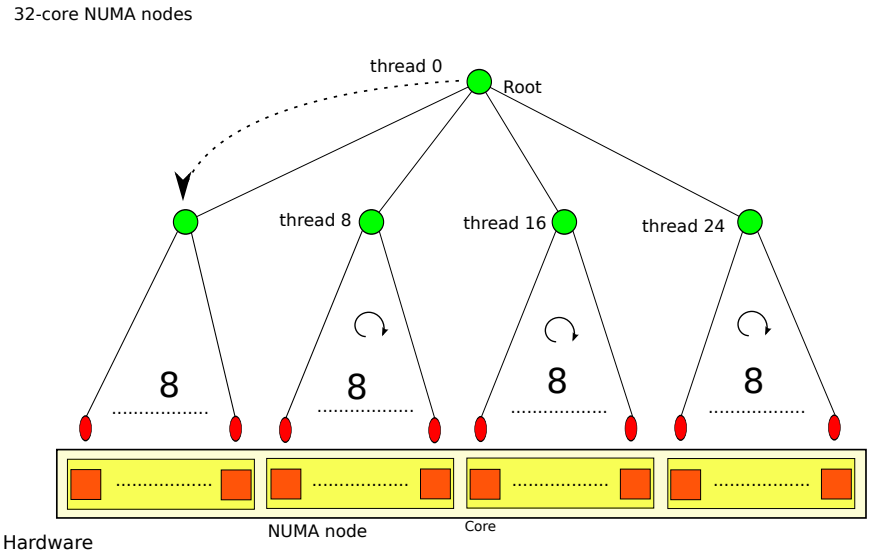


Figure 4.4: Hierarchical thread activation - 2nd stage

`barrier_done` is activated, and the last thread having reached this barrier continues to higher stages. Consequently, we climb the tree in a recursive way until reaching the tree root and its global barrier. Once root is reached, all nodes are informed that the global barrier is reached, and each thread goes down to its leaf through relative nodes and continues its execution.

By using a hierarchical tree taking into account underlying topology, we improve the data locality and reduce the contention by decoupling the work of Fork / Join mechanisms.

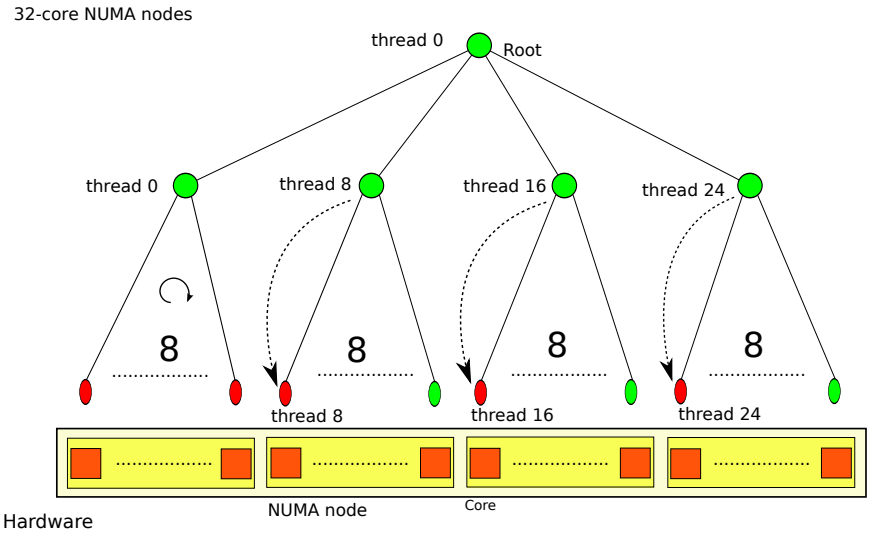


Figure 4.5: Hierarchical thread activation - 3rd stage

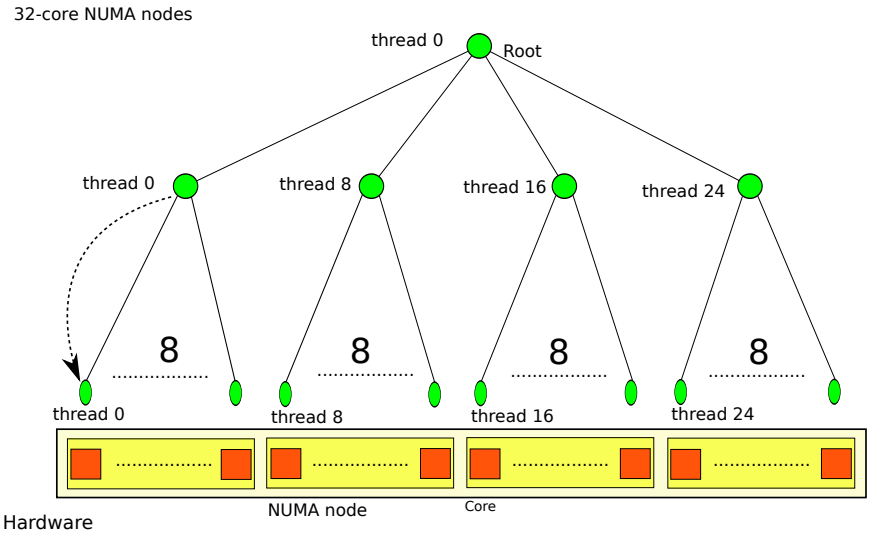


Figure 4.6: Hierarchical thread activation - 4th stage

4.4.3 Explore different tree shapes

Flat tree In the previous part, we took the example of a tree following hardware topology, as we call it `topology tree`. This solution seems to be the most intuitive tree shape, as it adheres to hierarchical architecture and optimizes data locality. However, for a small number of threads, centralized algorithm presented in Section 4.2 is an efficient way for thread activation and synchronization, as it is straightforward. We call this approach `flat tree` (depicted on the left part of Figure 4.7): we deal with a one level tree, with a single thread in charge of activating or synchronizing all other threads. The overhead is minimal for these operations.

Algorithm 3 Hierarchical barrier

Require: $tree, thread$
1: $node \leftarrow thread.father$
2: $bdone \leftarrow node.barrierdone$
3: $b \leftarrow node.barrier$
4: atomic($node.barrier \leftarrow node.barrier + 1$)
5: **while** $b + 1 = node.barriernbthreads$ **and** $node \neq tree.newroot$ **do**
6: $node.barrier \leftarrow 0$
7: $node \leftarrow node.father$
8: $b \leftarrow node.barrier$
9: atomic($node.barrier \leftarrow b + 1$)
10: **end while**
11: **if** $node.father \neq NULL$ **or** $node.father = NULL$ **and** $b + 1 \neq node.bnthreads$ **then**
12: **while** $bdone = node.barrierdone$ **do**
13: wait thread
14: **end while**
15: **else**
16: atomic($node.barrier = 0$)
17: $node.barrierdone = node.barrierdone + 1$
18: **end if**
19: **while** $node.typechildren \neq LEAF$ **do**
20: $node = node.child[thread.ranktree[node.depth]]$
21: $n.barrierdone = node.barrierdone + 1$
22: **end while**

Deep tree At the opposite, for a large number of threads, we can investigate trees deeper than the topology tree, in order to expose more parallelism. Indeed, by increasing the depth of the tree, we increase the number of steps and decrease the number of threads to deal with, which allows to lower the contention at each level. The binary tree is the deepest tree we can study. Since each node degree is minimal, it allows the smallest contention when it needs to deal with its children. But the drawback is that the global overhead increases due to the number of implied steps. These considerations lead to find a tradeoff between exposing parallelism and limiting the number of steps, depending on the number of threads we have to handle.

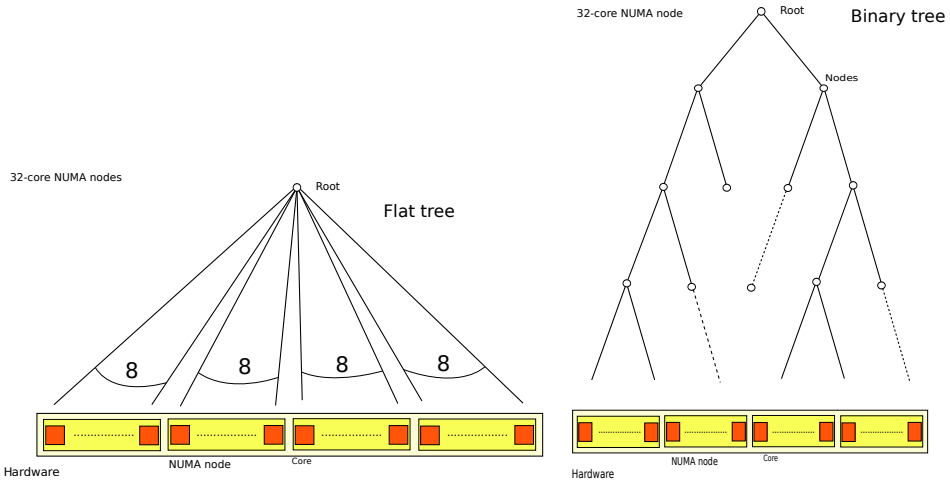


Figure 4.7: Different tree shapes

4.4.4 Handle tree shapes in a dynamic way

We saw that the tree shape is likely to influence performances, depending on the number of launched threads. Furthermore, some codes, as they perform OpenMP parallel regions, require less OpenMP threads than launched at the beginning of the execution. Thus, the number of used OpenMP threads is likely to change during execution time, and we need to dynamically adapt the tree shape. The difficulty here is twofold: elect the best tree shape to ensure best possible performances, and be able to change it in a dynamic way, with low additional overhead.

4.5 Contribution: Adaptive tree

In the last section, we described our design of NUMA-aware OpenMP runtime, consisting in a hierarchical tree where OpenMP threads are mapped, and used to wake or synchronize OpenMP team. The need to keep a minimal overhead for few threads and numerous threads drove to study different tree shapes. But as the number of threads involved in parallel codes can change during their execution, there is a need to tune tree shape in a dynamic manner.
Thus, we introduce here our concept of adaptive tree: we build a `topology tree`, mapping the underlying topology, and exploit it depending on the number of launched threads. Our contribution, extracted from [77], consists in only using a subset of the whole tree when all threads can be contained in this subset. We later present our mechanism to bypass the tree when possible.

4.5.1 Bypassing the tree

In the previous sub-section, we saw there was no unique solution for best tree shape. The need of handling tree shape in a dynamic manner according to the number of launched threads lead us to change the tree shape during execution. Our contribution consists in an adaptive tree, whose the exploitation depends on the number of launched threads during the execution. The principle is the following: we use only a sub-tree provided it can contain all the threads. We do this by comparing the number of launched threads with the number of available cores on the machine. If the number of threads can be contained inside a sub-tree, we set the root on top of this sub-tree.

Algorithm 4 Bypassing

Require: $tree, numthreads$
1: $node \leftarrow tree.root$
2: **while** $node.typechildren \neq LEAF$ **and** $numthreads \leq node.children[0].maxindex$ **do**
3: $\quad node \leftarrow node.children[0]$
4: **end while**
5: $tree.newroot \leftarrow node$
6: **return** $tree.newroot$

Algorithm 4 describes how Adaptive tree works through what we call a Bypassing mechanism. We start at the tree root, and take it as the current root. The algorithm then compares the number of launched threads with the total number of leaves in the sub-tree of the top left child node of the tree (given by the field `maxindex`). If the number of threads is lower or equal to the number of leaves, then the child node located at the top left becomes the root. This process is recursively repeated until reaching the leaves. At the end, Algorithm 4 returns the new computed root.

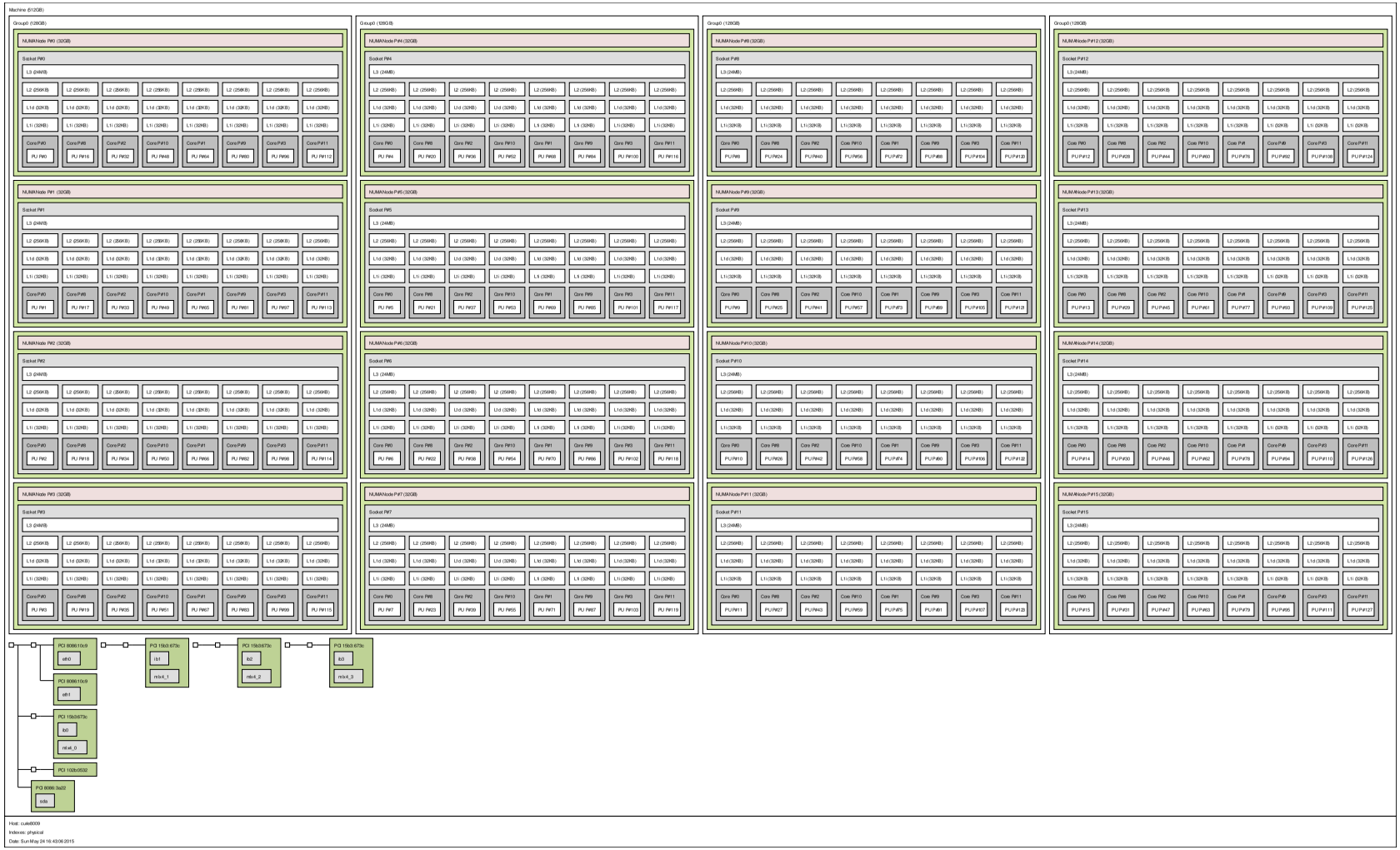


Figure 4.8: Logic view of a 128-core node

To illustrate our approach, we take the example of an application launched with 8 OpenMP threads and running on a large compute node. The compute node is a 128-core Bull Coherency Switch node, composed of 4 modules, containing each 4 8-core processors (Figure 4.8).

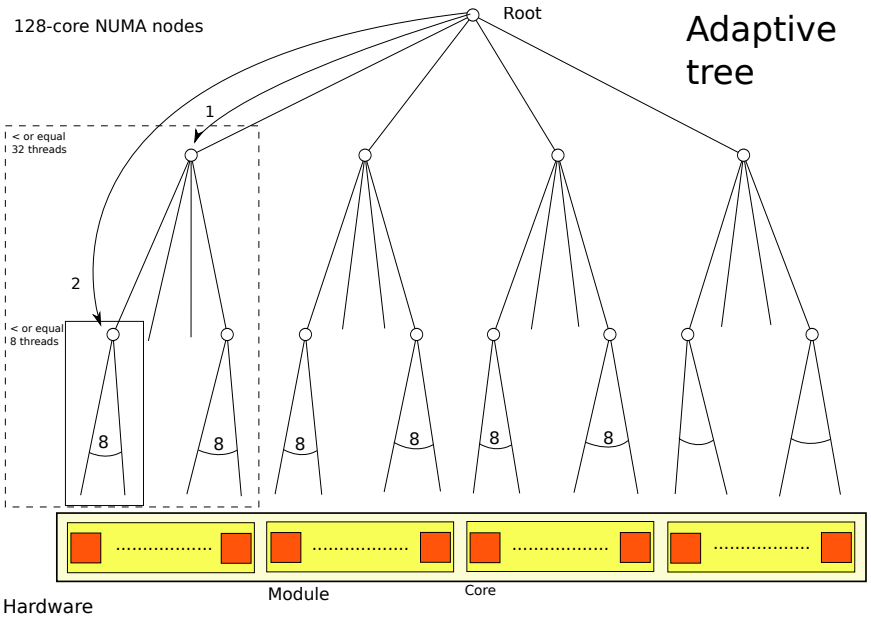


Figure 4.9: Adaptive tree

The tree depicted on Figure 4.9 maps the underlying topology of the node.

We can apply the bypassing algorithm to this configuration. The current node is the root of this tree. We start by considering the top left sub-tree, framed in a dashed rectangle. We compare the number of launched threads with the total number of leaves contained by this sub-tree. Since the sub-tree gathers a total of 32 leaves, then we tag the top left child node as the root. Again, we repeat this operation by examining its own top left sub-tree, in plain rectangle, and test if the number of its children exceeds or is equal to the number of threads. This time, the number of threads and leaves match. So we can place current root on the root of the top left sub-tree.

4.5.2 Apply bypassing to thread activation and thread synchronization

The mechanisms of thread activation and synchronization remain the same. The only difference is that bypassing algorithm returns a different root. So the thread activation and synchronization take the root returned by the bypassing algorithm as an input, and are therefore constrained inside the deduced sub-tree.

In order to make clear how bypassing algorithm works, we illustrate the concept with a code (Figure 4.10) involving several consecutive OpenMP parallel regions. Each region is launched with a different number of threads. Inside the runtime, the bypassing algorithm will be executed as soon as the parallel region is encountered. The first encountered parallel region is launched with 128 threads, which corresponds to the total number of available threads. So the computed root of the algorithm will be the real root of the tree. This computed root applies to the algorithms of parallel region and barrier. Once the first parallel region is passed, the computed root is reset. Consequently, all trees will be used and all threads will be activated.

When encountering the second parallel region, 32 threads are required. The Bypassing algorithm computes the new root and tags the left child node as the new root: all its leaves can contain the requested threads (Figure 4.11). So only the threads contained in the left sub-tree are waiting to be activated.

At the end of the parallel region, the barrier applies only on this sub-tree.

```
int main(int argc, char **argv)
{

 /* Sequential code */

#pragma omp parallel num_threads(128)
{
 /* Code of parallel region */
}

#pragma omp parallel num_threads(32)
{
 /* Code of parallel region */

 #pragma omp barrier

 /* Code of parallel region */

}

#pragma omp parallel num_threads(8)
{
 /* Code of parallel region */

 #pragma omp single
 {
   /* Body of omp single construct */
 }

 /* Code of parallel region */
}

#pragma omp parallel num_threads(128)
{
 /* Code of parallel region */
}

 return EXIT_SUCCESS;
}
```

Figure 4.10: OpenMP regions with different threads

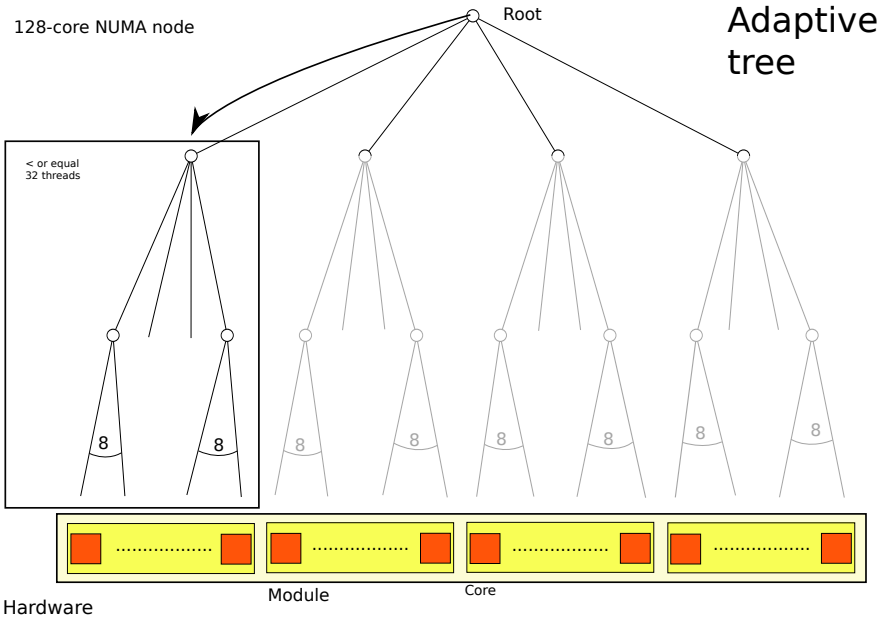


Figure 4.11: Adaptive tree with 32 threads

The third parallel region is to be launched with 8 threads. Starting from the tree root, we recursively cross the tree and take the top left node of the smallest sub-tree as the root. We then launch the parallel region.

But the flexibility provided by the Bypassing approach allows to go back to a big number of threads (i.e. last parallel region).

This example showed that our bypassing algorithm offered a flexible way to efficiently launch OpenMP constructs with varying numbers of threads, and on consecutive parallel regions. The computed root is reset when the parallel construct is finished. This approach allows to minimize the impact for thread activation and synchronization implementations.

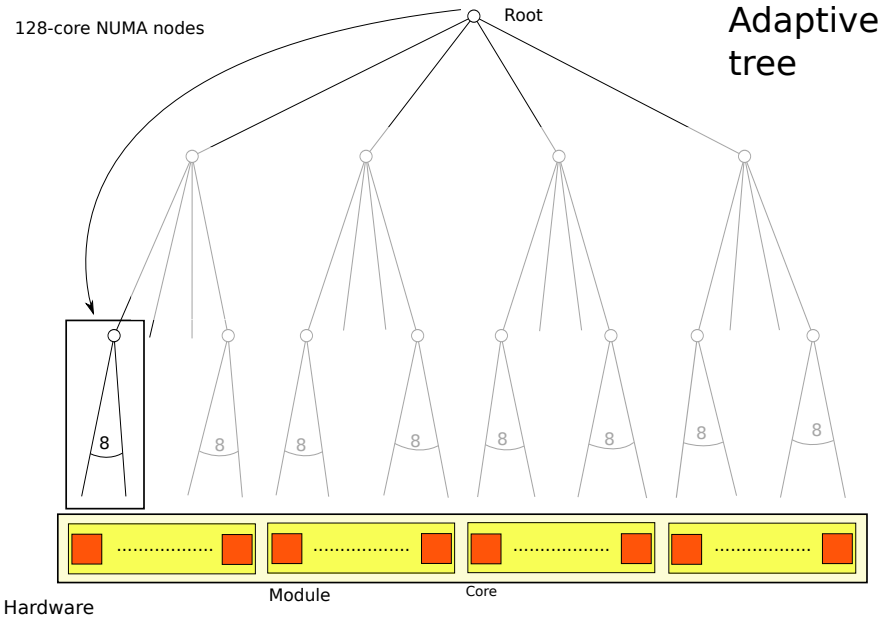


Figure 4.12: Adaptive tree with 8 threads

The main advantage with the Bypassing algorithm is that this approach is transparent for thread activation and barrier: the algorithms just have to take the input root into account, as a starting point or as a limit.

4.5.3 Implementation

Hierarchical tree, thread activation and synchronization, and Adaptive tree have been implemented in MPC 2.5.0. To be able to build the topology tree, we needed to know about underlying topology. So we used HWLOC library (Hardware LOCality). This library allowed to discover the hardware topology and therefore to build our tree in a recursive way. The tree was built once at the beginning of the application execution, when first encountering an OpenMP parallel region. Each time a parallel region is encountered, the Bypassing algorithm was executed and returned a new computed root. This computed root was taken as the input for thread activation and synchronization.

4.5.4 Experimental results

We evaluated the adaptive tree on two hardware configurations:

- one 32-core node composed of 4 processors Nehalem EX X7550 clocked at 2.00 GHZ (Tera-100 supercomputer)
- a 128-core Bull Coherency Switch node containing 16 processors Intel Xeon E7-4800. These processors are dispatched on 4 modules linked with a BCS interconnect system, each composed of 4 NUMA nodes (Curie supercomputer)

EPCC microbenchmarks First experiments were performed on EPCC [21], a suite of microbenchmarks the goal of which is to estimate overhead of OpenMP constructs. We focused on `omp parallel` and `omp barrier` constructs, and compared our approach with other tree shapes and OpenMP runtimes.

Different OpenMP runtimes were compared:

- ICC 12.1

- GCC 4.6.1 on 32-core node
- GCC 4.4.4 on 128-core node
- MPC 2.5.0

GCC compiler includes libGOMP library.

The different tree shapes compared with the adaptive tree were a ”4-8” tree, a flat tree on the 32-core node, a ”4-4-8” tree and a ”4-32” tree on the large node.

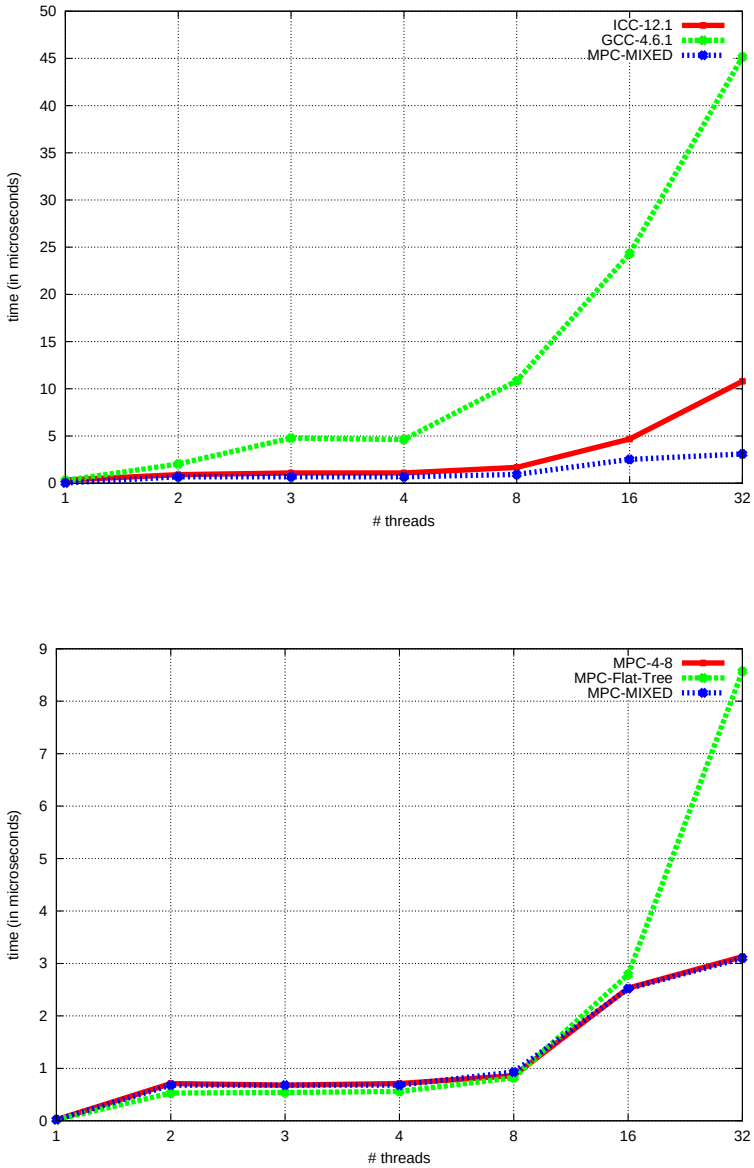


Figure 4.13: Adaptive tree - Parallel region overhead on 32-core node

Results are depicted on Figures 4.13, 4.14, 4.15 and 4.16. On the first hardware configuration, we compared performances between a flat tree, a topology tree and our adaptive tree, with MPC. We notice that on 32-core node, the flat tree performs best with a few threads but performances fall with more than 16 threads. The ”4-8” tree and adaptive tree have similar performances on 32-core node. We observe GCC performs poorly on all configurations, especially with a large number of threads, probably due to

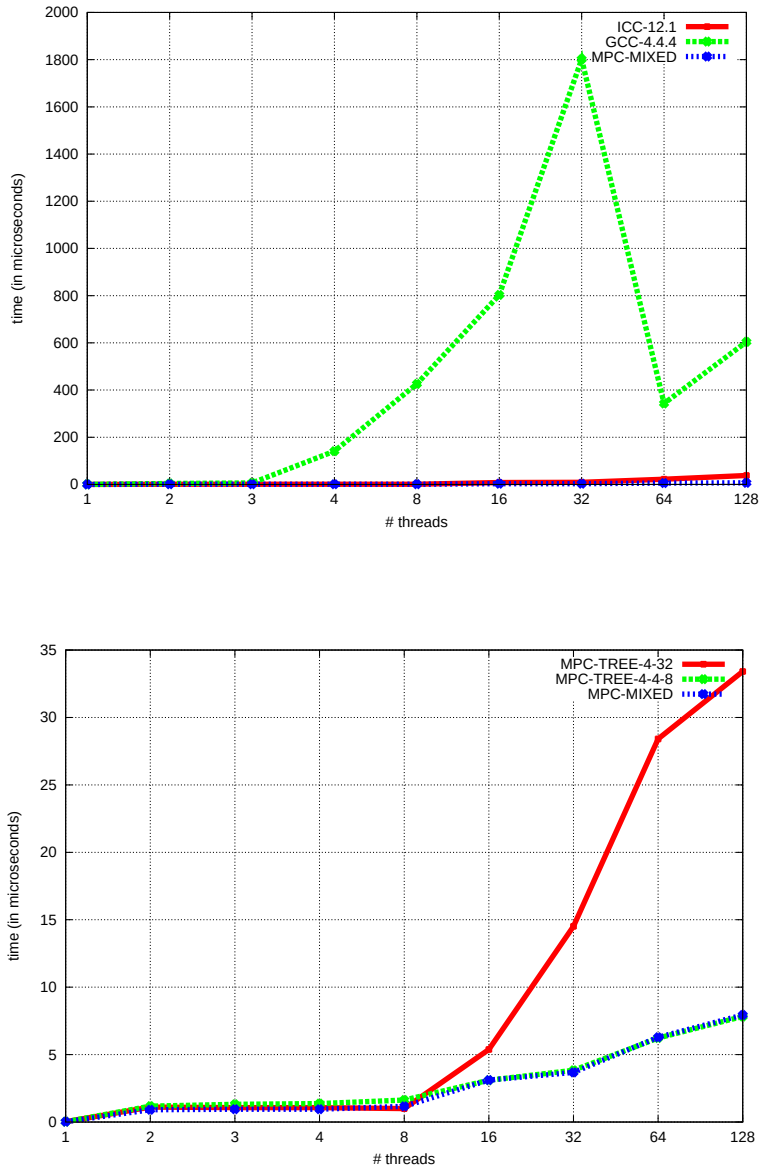


Figure 4.14: Adaptive tree - Parallel region overhead on 128-core node

the absence of support of NUMA environment. MPC with the Adaptive tree is able to better perform than with the ICC 12.1, on both hardware configurations.

4.6 Discussion about hierarchical work stealing

Some applications are called irregular due to their sparsity. Such applications cause load imbalance between threads, in the case of OpenMP applications. Load imbalance is an obstacle to scalability of codes, and some techniques enable to correct it, in particular work stealing. Work stealing corrects imbalance by allowing idle threads to look for compute work to be executed by other threads. There are opportunities for work stealing with OpenMP loops and OpenMP tasks.

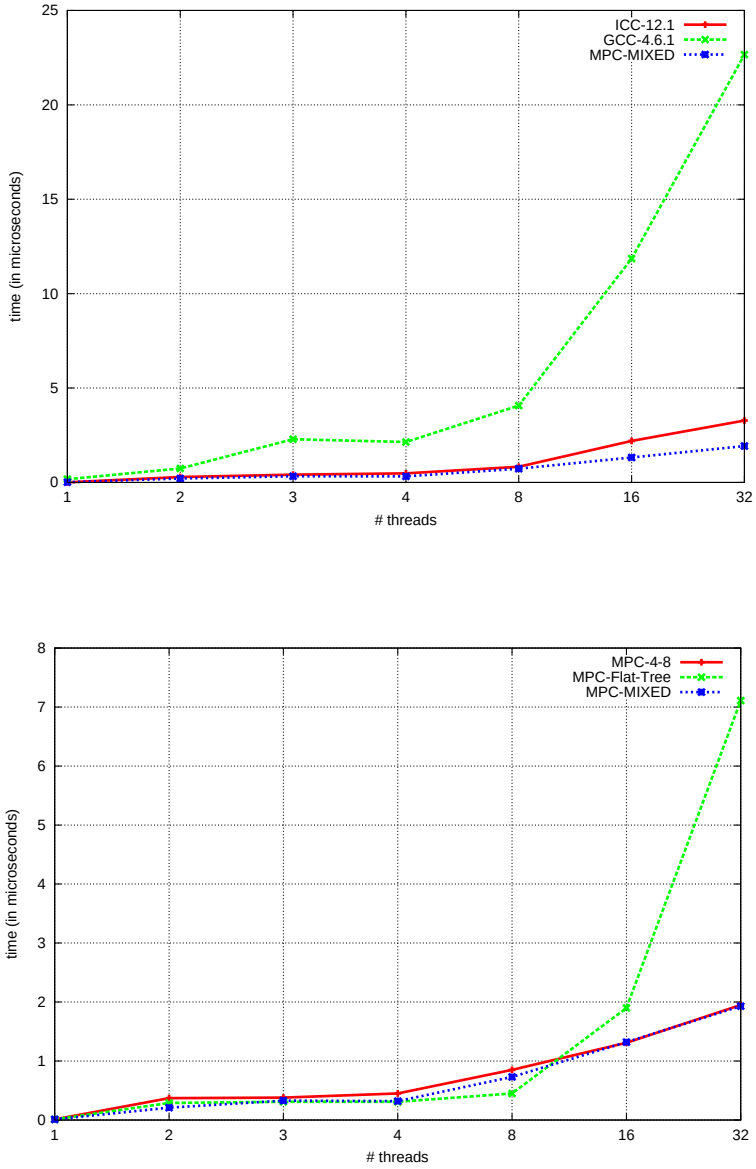


Figure 4.15: Adaptive tree - Barrier overhead on 32-core node

OpenMP loops OpenMP standard allows, via constructs like `#pragma omp for`, to split their iteration space involved in loops into chunks, to be dispatched on threads and executed. There is also a possibility to tune granularity of chunks. Several scheduling policies such as `static` and `dynamic` are described to schedule chunks on threads. Whereas `static` policy doesn't allow correcting load imbalance, `dynamic` policy gives room for work stealing.

This is why OpenMP runtimes propose features such as work stealing to correct this imbalance. When a thread has no more chunks to execute and becomes idle, he looks for other chunks to execute in the neighbor threads.

OpenMP tasks OpenMP 3.0 revision introduced the concept of explicit tasks. Developers can express these explicit tasks via `#pragma omp task` construct, as soon as this feature is supported by OpenMP runtime. This task is associated with a function or a portion of code. A thread encountering this construct

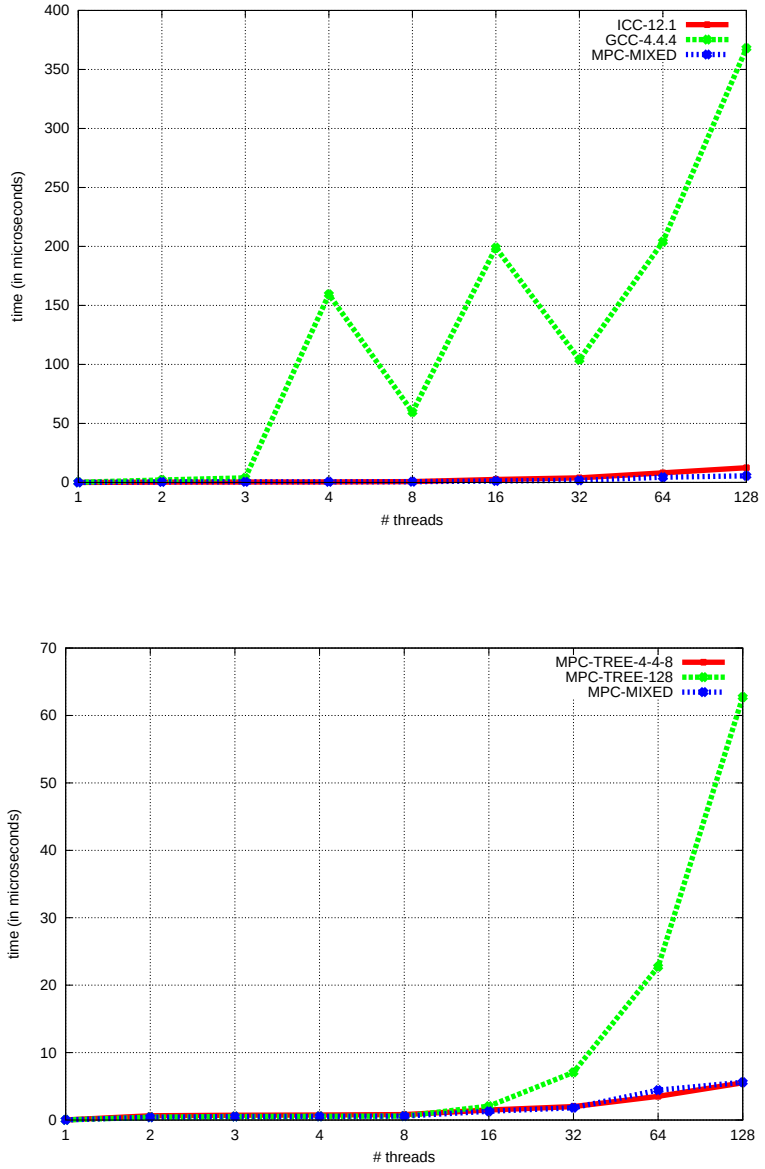


Figure 4.16: Adaptive tree - Barrier overhead on 128-core node

is in charge of executing the associated task before reaching a scheduling point. Thus, during execution time, several tasks related to a thread are likely to be waiting for execution. There are different possible implementations to store OpenMP tasks waiting for execution. Examples of implementations are a pool of tasks shared by all threads or a private queue for each thread.

In the following sub-sections, we will study different strategies and implementations of work stealing.

4.6.1 Strategies for work stealing

Work stealing strategies have been widely investigated and numerous algorithms have been proposed in the literature. Emergence of NUMA architectures lead to perform work stealing in a hierarchical way. In [28], the authors adopt a hierarchical representation of OpenMP task scheduling and adapt work

stealing strategies to this representation. [82] introduces HotSLAW, a task library proposing several work stealing mechanisms such as:

- Hierarchical Victim Selection: steals from the nearest neighbor and gradually climbs locality hierarchy.
- Hierarchical Chunk Selection: tunes chunk size to steal depending on the distance between the thief thread and the victim thread

Another contribution extracted from [31] enables a technique to split task queues between victim and thief threads. This technique allows a scalability of work stealing on 8,192 cores.

4.6.2 Implementations of hierarchical work stealing

We investigated several designs for work stealing, relying on hierarchical structure previously presented.

A first implementation of hierarchical work stealing is to store the entire tree (e.g nodes and leaves) in a stack. To parse the tree, the thread has to pop the stack.

An alternate design allows to avoid parsing the tree.

Algorithm 5 Compute index

Require: $DEPTH, nbleaves, treeShape[DEPTH], threadPos[DEPTH]$
1: **for** $i := 0$ **to** $DEPTH$ **do**
2: $absPos[i] \leftarrow 0$
3: $currentPos[i] \leftarrow threadPos[i]$
4: **end for**
5: **while** $iter < nbleaves$ **do**
6: **for** $i := 0$ **to** $DEPTH$ **do**
7: $currentPos[i] \leftarrow (absPos[i] + threadPos[i]) \mathbf{mod} treeShape[i]$
8: **end for**
9: $i \leftarrow DEPTH - 1$
10: **while** $i \geq 0$ **and** $absPos[i] = treeShape[i] - 1$ **do**
11: $absPos[i] \leftarrow 0$
12: $i \leftarrow i - 1$
13: **end while**
14: **if** $i \geq 0$ **then**
15: $absPos[i] \leftarrow absPos[i] + 1$
16: **end if**
17: $iter \leftarrow iter + 1$
18: **end while**
19: **return** $currentPos$

As described in Algorithm 5, it consists, for each thread, in starting from the leaf of current thread and in parsing all leaves by beginning with the next leaf. The algorithm takes three parameters as input: the tree depth, the tree shape as an array, and the thread position relative to the tree. The tree shape (i.e the number of children at each level) is stored in an array the size of which corresponds to the tree depth. Another array, *threadPos* contains the path from the tree root to the location of the current thread, and is used to compute the next targeted leaf, modulo the tree shape. This process is iterated until parsing all leaves. This algorithm gives each thread a unique parsing in order to avoid contention, and first explores the neighbors first, before looking for other works to steal.

With Figure 4.17 we take an example of a balanced tree and apply the algorithm to it. The depth of the given tree is equal to 3 and its shape can be expressed as *treeShape = 3,2,3* . Let's consider the current thread is located on the second upper sub-tree, its second sub-tree and on the second leaf. We will apply the algorithm to this thread. Its location following the tree is thus given as *threadPos = 1,1,1* . The location of this thread is marked with an arrow. When starting the algorithm, *absPos* is initialized to *0,0,0* and *currentPos* is set to 1,1,1. The ordering of the parsing is tagged with the

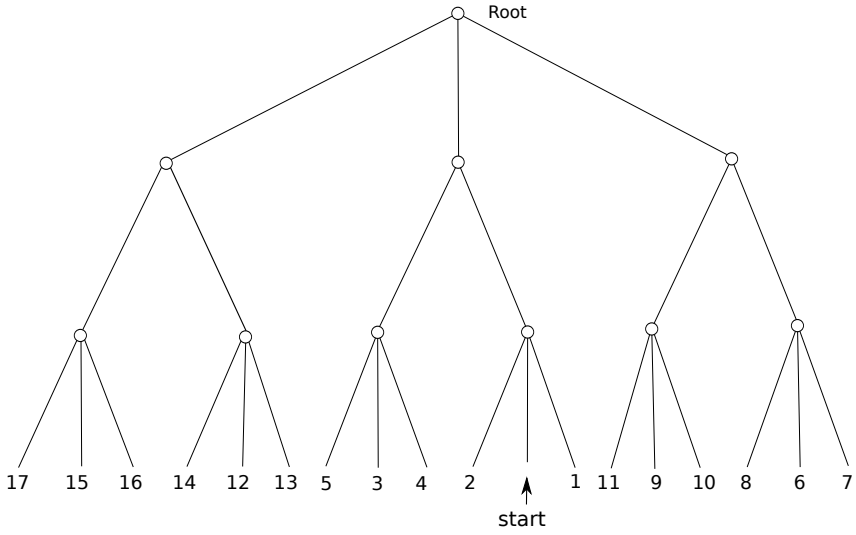


Figure 4.17: Example of implementation of compute index for work stealing

numbers below the leaves. The algorithm then iterates through the leaves and the first step is the last leaf of the sub-tree (step 1). For the next step, *currentPos* is computed taking into account *absPos* and modulo *treeShape* , then it is gives 1,1,0. So on step 2, the current thread looks after the first leaf of the sub-tree. On the following step, we parsed the entire starting sub-tree, the next leaf to be visited is the second one of the neighbor sub-tree, wich is included in the containing sub-tree. After iterating through this sub-tree, the current thread parses the top right sub-tree, starting by the second leaf, which is still computed following *absPos* and *treeShape* . When finishing the algorithm, *absPos* equals *treeShape* .

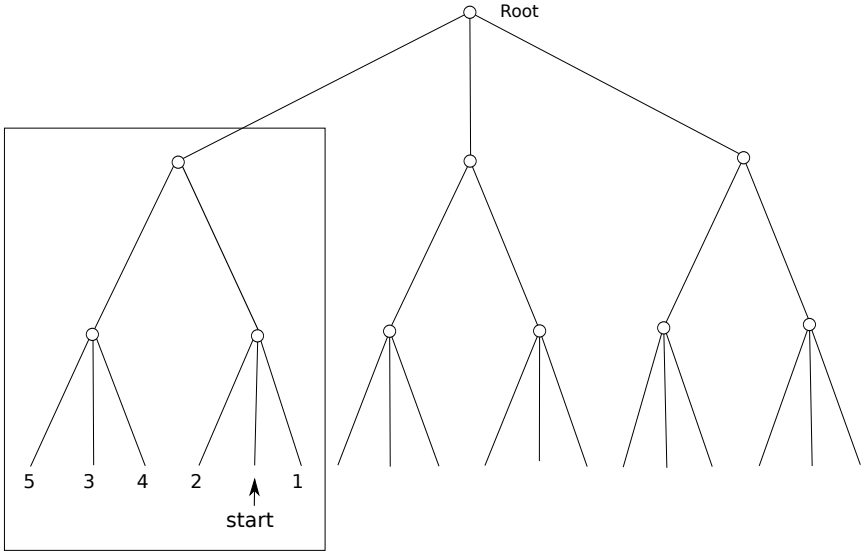


Figure 4.18: Apply Bypassing algorithm to the Work Stealing with compute index algorithm

We can also use the Adaptive tree for Work Stealing purposes and combine it with the Compute Index algorithm. Let's apply the Bypassing algorithm to the left sub-tree (Figure 4.18) and run the Compute Index algorithm again. The value of *treeShape* will be *2,3*. The current thread is located on the second lowest sub-tree, on the second branch (*threadPos = 1,1*). *absPos* and *currentPos* are respectively initialized to *0,0* and *1,1*. Then, the algorithm parses all leaves like in the first case, but it only applies for the given sub-tree.

4.7 Conclusion

The goal of this chapter was to highlight constraints on OpenMP runtimes encountered in hybrid MPI+OpenMP codes, and running on NUMA architectures. We presented a state-of-the art design to take into account hierarchical parallelism implied by NUMA systems, with application to most encountered OpenMP constructs such as thread activation and barrier. We introduced our first contribution, the Adaptive tree, that offers a flexible way to reduce the overhead of OpenMP runtime, through its Bypassing algorithm. We demonstrated through several examples that Bypassing algorithm was transparent to other features such as thread activation, synchronization or Work stealing algorithms. This contribution was implemented inside MPC runtime and experiments showed this approach was a promising solution to meet previously enumerated constraints. At last, we discussed about features coming with OpenMP implementations such as Work stealing and impacts of hierarchical topology on Work stealing.

Chapter 5

Study Collective operations in hybrid MPI+OpenMP context

In Chapter 4, we identified the overhead of OpenMP runtimes as a limiting factor of scalability in hybrid MPI+OpenMP codes, following Fine Grain pattern. We then presented a design in order to reduce this overhead on NUMA architectures.

But other issues have been highlighted and are other bottlenecks for hybrid applications. These issues are such:

- Ratio of sequential and communication parts in total execution, for Fined Grained codes
- How to reuse idle cores outside OpenMP parts, in Masteronly codes
- Problem of code complexity with coarse grained applications, and reduce programming effort

In this chapter, we conduct a global study of collective operations, starting by focusing on existing MPI Collectives and describe different ways to optimize them in *Masteronly* mode. We then introduce hybridization of such collective operations by exploiting idle cores and idle OpenMP threads, and see how we could apply the state-of-the-art optimizations with threads. After, we introduce our contribution consisting in optimizing `MPI_Allreduce` with OpenMP threads, following several techniques.

In the second part of the chapter, we recall characteristics of Coarse grained codes and see how to propose common constructs between MPI and OpenMP in order to alleviate the programming effort in this pattern. By recalling combination between MPI and OpenMP SPMD, we propose unified constructs coordinating both MPI tasks and OpenMP threads. We introduce a unified barrier as a proof of concept, synchronizing MPI tasks and OpenMP threads, and show its performances againts a regular implementation of this feature.

5.1 Tackle the problem of sequential time in Fine Grain pattern: Focus on MPI collectives

When studying the taxonomy of Hybrid model in Section 3.3 of Chapter 3, we listed the drawbacks of Fine Grain approach:

- The first one is the difficulty to maintain the compute cores active during the whole execution time. Indeed, all compute units are busy only when OpenMP threads are launched. This is particularly true with *Masteronly* style. Figure 5.1 illustrates this shortcoming with a code launched with 4 MPI tasks and a total of 16 OpenMP threads, and compares occupation of compute cores, inside and outside OpenMP parts. In the last case, only a subset of cores is used, which leads to a loss of parallelism.
- The second shortcoming is that sequential and communication components (i.e. when only executing MPI code outside OpenMP constructs) are far from being negligible. For example, the

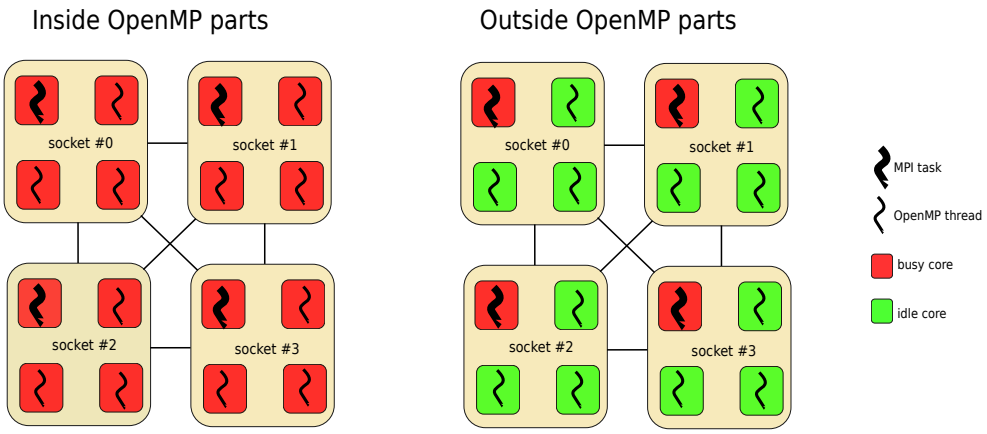


Figure 5.1: Hardware view of Master approach

authors in [90] showed that communications in this hybrid scheme could reach 15% of global execution time.

As sequential and communication times are such obstacles to the scalability hybrid codes, we studied how to minimize them, and have looked for elements that could be optimized.

MPI Collectives are good candidates for optimization because they are well encountered in high performance codes and are often concerned by communications by involving multiple MPI ranks at the same time. Moreover, they represent a non negligible percentage of total execution time, and consequently in sequential time. To illustrate this point, we studied an hybrid MPI + OpenMP application simulating interactions between particles with a Monte Carlo algorithm, involving multiple `MPI_Allreduce` operations [36]. We launched it with 4 MPI tasks and 128 OpenMP threads per team, on a representative configuration of 4 Bull Coherency Switch 128-core nodes, on the Curie supercomputer. The time spent in such collective represents more than 50% of the global execution time of the application.

In the next section, we study the state-of-the-art optimizations of MPI collectives, using shared memory systems and network topology.

5.1.1 Related Work on optimizing MPI collectives and reduction collectives

We expose in this section the state-of-the-art regarding optimizations of MPI collectives, with a focus on reduction collectives. These optimizations take advantage of two aspects: (i) shared memory available in current compute nodes, and (ii) underlying topology including network topology.

Exploit shared memory

Shared memory has been strongly exploited to design powerful algorithms implementing MPI collectives. In [89], the authors improve the collective operations of MPICH runtime, and present new algorithms for Reduction operations. Two algorithms are presented to implement Allreduce collective: *Recursive Halving and Doubling*, and *Binary Blocks*. With both algorithms, Allreduce operation is actually decomposed in 2 operations: `Reduce_scatter` and `Allgather`. `Reduce_scatter` is an irregular collective, which performs a regular reduce operation, but, instead of sending the result to the root like allreduce, it scatters it among processes. With *Recursive Halving and Doubling* algorithm, the input vectors are recursively halved and one half of the vector is sent to the neighbor process. At each step, the distance between neighbor processes is doubled. When each process has a reduced element, `Reduce_scatter` operation is achieved. Then an `Allgather` operation is performed by recursively concatenating elements, but the distance between neighbor processes is recursively halved. This algorithm

implies an overhead when the number of MPI processes is not a power of 2. The *Binary Blocks* algorithm reduces the load imbalance implied by the *Recursive Halving and Doubling* algorithm when the number of MPI tasks is not a power of 2. Indeed, it splits MPI processes into power of 2 chunks.

HMPI [42, 68], presented in Section 2.5 of Chapter 2, optimizes `Allreduce` collective with several algorithms. On algorithm, *Tiled Reduce-Broadcast*, splits input vectors into multiple chunks and each MPI task reduces a chunk into a temporary buffer. Tiling is used only within sockets. Then temporary buffers are reduced between sockets. This algorithm shows good performances for large vectors.

The article [79] presents two variants of the same algorithm to parallelize `Allreduce` through shared memory. In the first flavor, each process starts by dividing its own block of elements into several sub-blocks (one sub-block per process available on the node). Then the ith process on the node reduces the ith block of each MPI rank, storing the intermediate result into block 0. The second flavor optimizes the memory traffic by shifting the work of each process in a cyclical way.

In [43] the problem of double copy between MPI processes is mentioned, and this problem is solved by allocating a buffer for communications between processes. MPI Reduce collective is implemented by splitting buffers in chunks, on which processes perform operations

Rely on topology

Other contributions highlight the importance of having knowledge about the topology and separate intra-node from inter-node communications, by specifying sub-communicators and electing leaders.

The authors in [61] discuss about exploiting the network topology to design topology-aware algorithms for two MPI collectives: *Scatter* and *Gather*. These algorithms involve sub-communicators, created at MPI initialization, gathering processes according to their position in the topology hierarchy. These optimizations allow to improve these collectives by almost 54%.

Other contributions focus on separating inter- and intra-node communications.
In [74], the HierKNEM framework includes algorithms for `MPI_Bcast` and `MPI_Reduce`, using hierarchical layers, and one-sided primitives. [104] introduces some methodologies to separate inter-node from intra-node communications, using sub-communicators. The authors also use a cache blocking technique which consists in tiling the message and storing it in L1 or L2 cache, evaluating different segment sizes.

In [63], considering one single leader per node as a limitation, the authors propose a multi-leader approach, a maximum of one per socket, and several implementations, including one exploiting shared memory. This approach leads to a performance improvement of 60% for small messages, and 70% for medium-sized messages.

The article [62] exposes encountered problematics when executing MPI collectives in heterogeneous environment, including Many Integrated Cores (MIC) co-processors[1]. Especially, it identifies a bottleneck between MIC and network adapter. Then, authors introduce a framework decoupling execution of MPI collectives in heterogeneous environment. An example is described with the implementation of `MPI_Bcast` collective, where different aspects are considered, as communications between MIC and CPU, intra-node and inter-node communications. For intra-MIC communications, they present several algorithms to implement reduction, and one with a tree-based approach, using OpenMP threads. Those threads distribute the work for the computational part of the reduction.

5.2 Discussion about hybridization of MPI collectives

Previous Related Work showed different techniques to optimize MPI Collective operations, with a focus on reduction operations. One of the techniques consisted in decoupling communications whether they happen inside or between nodes.

In this section we describe how we can take advantage of the Hybrid model to accelerate the work made by Collective operations, and how we apply encountered techniques using them. We define hybridization of Collective operations by using idle cores to split work done by such operations. This can be done following multiple directions, which have been explored in some Related Work:

- Use threads to separate inter node from intra node communications
- Split messages

- Parallelize both communications and computations

The following sub-sections describe these different flavors using several algorithms, which can apply to both rooted and non rooted MPI collectives.

5.2.1 Split communicators with threads

MPI Collectives such `MPI_Bcast`, `MPI_Gather`, `MPI_Allgather` or `MPI_Scatter` are purely communications oriented, as they do not perform computations in their data. Then, all we can do with spare cores is to parallelize communications between MPI tasks. Some Related Work showed that separating inter-node from intra-node communications was a promising way to optimize Collectives.

Algorithm 6 Split intra-node and inter-node communications

Require: $message, size, nbNodes, nbProcsOnNode, communicator, MPIRank, numthreads, threadRank$

1: $distantComm[nbProcsOnNode]$
2: $localComm[nbNodes]$
3: $MPIMasterRank \leftarrow \frac{MPIRank}{nbProcsOnNode} * nbProcsOnNode$
4: **for** $i = 0 \rightarrow nbProcsOnNode$ **do**
5: $localComms[i] \leftarrow MPIMasterRank + i$
6: **end for**
7: **for** $i = 0 \rightarrow nbNodes$ **do**
8: $distantComms[i] \leftarrow i * nbProcsOnNode$
9: **end for**
10: **for** $i = 0 \rightarrow numthreads$ **do**
11: **if** $threadRank = 0$ **then**
12: **if** $MPIRank = MIPMasterRank$ **then**
13: collective($message, size, distantComm$) {Inter node communications with OpenMP Master thread}
14: **end if**
15: **else if** $threadRank = 1$ **then**
16: collective($message, size, localComm$)
17: **end if**
18: **end for**

Algorithm 6 depicts how we can use threads to separate communications within a node from communications between nodes. In the first part of the algorithm, we create one communicator per node, *localComm*, and another communicator, *distantComm*, gathering each leader of each node. The leader of each node, *masterRank*, is computed depending on the current rank, and the number of MPI ranks per node. In the second part, depending on the MPI rank, we search whether it is a local communication or an inter-node communication. We parallelize this part using 2 threads.

Algorithm 7 Split communications between nodes

Require: $nbNodes, numthreads, nbProcs, communicator, MPIRank$
1: $nbProcsPerBlock \leftarrow \frac{nbProcs}{numthreads}$ {Provided $nbProcs$ can be divided by $numthreads$}
2: $block \leftarrow \frac{MPIRank}{nbProcsPerBlock}$
3: $splitComms[numthreads]$
4: **for** $i = 0 \rightarrow numthreads$ **do**
5: **for** $j = 0 \rightarrow nbProcsPerBlock$ **do**
6: $splitComms[i][j] \leftarrow (block * nbProcsPerBlock) + j$
7: **end for**
8: **end for**
9: **for** $i = 0 \rightarrow numthreads$ **do**
10: collective($sendbuf, recvbuf, root, splitComms[i]$) {Parallel communications}
11: **end for**

Another possibility, depicted in Algorithm 7, consists in splitting communications between nodes. The number of involved MPI tasks is divided into blocks by the number of launched threads, in order to create as many sub-communicators as there are threads. We then fill sub-communicators using computed blocks. Then we perform communications inside a parallel loop.

This algorithm allows to use more threads than the previous one.

5.2.2 Message tiling and parallelizing computations using threads

As a case study, we developed a code to validate the approach of using OpenMP threads to optimize `MPI_Broadcast` collective. We separate inter node from intra node communications.

We saw in [74], that tiling was used for a better use of caches. We can also add that messages exchanged between MPI tasks are bigger in hybrid context due to bigger sub-domains. So this technique can be promising.

Algorithm 8 Hybridize collective with $numthreads$ given

Require: $sendbuffer \geq 0$, $recvbuffer \geq 0$, $numelements$, $numthreads \geq 0$
1: $size \leftarrow \frac{numelements}{numthreads} * datasize$
2: **if** $numelements \mod numthreads = 0$ **then**
3: **for** $i = 0 \rightarrow numthreads$ **do**
4: collective($sendbuffer + i * size, recvbuffer + i * size, \frac{numelements}{numthreads}, communicator$)
5: **end for**
6: **else**
7: collective($sendbuffer, recvbuffer, numelements, communicator$) {Parallel communications}
8: **end if**

Algorithm 8 describes this technique. The call to the collective is surrounded by a loop, and `numthreads` calls are performed, `numthreads` being the number of involved OpenMP threads. Each thread computes an offset and deduces a chunk of `sendbuffer` and `recvbuffer`. Each thread performs the collective operation with its chunk. This algorithm is also interesting for collectives performing computations such as reduction collectives (`MPI_Reduce` and `MPI_Allreduce`). Inded, threads are then used to parallelize reduction.

We will base our contribution on this algorithm.

5.3 Contribution: Use OpenMP threads to optimize MPI_Allreduce

In Related work, we observed in [79] some techniques to parallelize *Allreduce* collective by decomposing input vectors with the help of other MPI tasks. In the last section, we studied how to use threads to optimize collectives, by separating for example inter-node from intra-node communications. Our contribution is extracted from [76] and extends beyond[79] using OpenMP threads.

5.3.1 Hybrid Allreduce approach

Our approach of hybridizing *Allreduce* consists in splitting the vectors of a `MPI_Allreduce` operation in multiple chunks so that each available core will be in charge of reducing one chunk. Thus, each OpenMP thread would process a subset of the input vector by performing a smaller MPI reduction, leading to the parallelization of the computational and communication parts of the whole operation with independent operations.

Figure 5.2 depicts the example of an application running on a computational node containing four 4-core processors for a total of 16 cores. Let us consider now a parallel MPI + OpenMP application running with 4 MPI processes. Considering that MPI processes should be dispatched among the whole node, the regular approach would be to place one of these processes per multicore processor, letting 4 cores per MPI rank to launch an OpenMP parallel region. The left part of Figure 5.2 illustrates the behavior of the application when performing an *Allreduce* communication outside any OpenMP region.

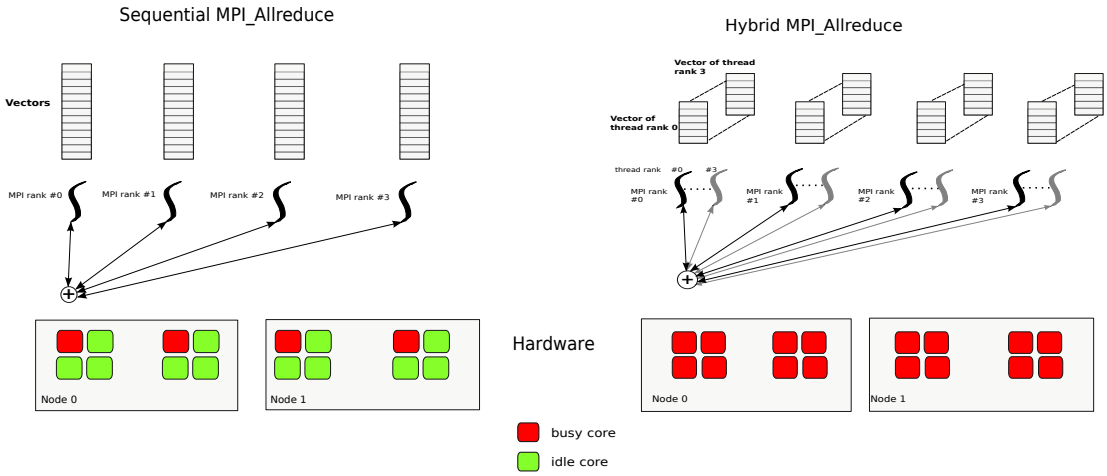


Figure 5.2: Hybrid MPI_Allreduce

Because this operation is not done inside a parallel block, only 4 cores will help processing this operation. It leads to a total of 12 idle cores. Our solution is described on the right part of this figure: it shows how the spare cores are used when performing an hybrid `MPI_Allreduce` operation. We can see that for each MPI rank, 4 OpenMP threads are launched, each working on its subset of the initial vector, and thus all cores of the node are active.

The algorithm presented in Figure 5.3 describes how this new version of `MPI_Allreduce` is done with OpenMP. Let *numthreads* be the number of OpenMP threads defined when running the application. The first step is to take care of the communicator to allow multiple threads to perform sub-reductions. The input communicator *comm* is duplicated in *numthreads* identical sub-communicators, so that each thread can have its own. This is done inside the `manage_comms` function. But duplicating the input communicator each time `MPI_Allreduce` is called may imply a large overhead: therefore we implemented a cache to keep the sub-communicators available. We motivate this optimization by observing a repeating pattern in current scientific codes, meaning the same communicators are reused with different calls of MPI collectives. Indeed, in numerous high performance applications, a main loop is performed to iterate physical system and MPI collectives are called at each iteration of this loop, using identical communicators.
The principle of this cache is quite simple: the captured communicator is split into sub-communicators and stored. If the communicator for the next reduction operation is identical, the previous subcommunicators are used, avoiding additional duplication. Note this algorithm depicts a cache of size 1, but more optimized structures can be used to cache multiple main communicators. After dealing with communicators, the initial vector is split into *numthreads* chunks, each thread computes its offset based on the vector size, and then works on its chunk. This is done with a simple parallel for directive with a static schedule. Finally, instead of doing one call to `MPI_Allreduce`, our approach performs *numthreads* smaller reductions. The algorithm tries to balance the charge among threads. If the number of elements to reduce cannot be divided by the number of threads, we straighten the remaining elements between them.

5.3.2 Rank Shifting

The main effect of the algorithm Figure 5.5 is to duplicate the calls to the MPI runtime (as many calls to `MPI_Allreduce` as the number of OpenMP threads within a team) with exactly the same configuration.

```
MPI_Comm newcomm[numthreads];
MPI_Comm lastcomm;

int initcomm = 0;

/* Create a cache to optimize communicators handling */
void manage_comms(MPI_Comm comm)
{
 if((lastcomm != comm) || (!initcomm)) {
    if(initcomm) {
     for(i = 0 ; i < numthreads ; i++)
      PMPI_Comm_free(&newcomm[i]);
    }

    for(i = 0 ; i < numthreads ; i++)
     PMPI_Comm_dup(comm, &newcomm[i]);

    initcomm = 1;
 }

 lastcomm = comm;
}

/* Entry point fo reduction */
int MPI_Allreduce(void *sendbuf, void *recvbuf,
int numelements, MPI_Datatype datasize, MPI_Op op, MPI_Comm comm)
{
  int size, i;

  manage_comms(comm);

  size = (numelements / numthreads) * datasize;

#pragma omp parallel for schedule(static)
  for(i = 0 ; i < numthreads ; i++)
  {
      if (i < numelements % numthreads)
        PMPI_Allreduce(sendbuf+i*size+datasize, recvbuf+i*size+datasize,
        numelements/numthreads, newcomm[i]);
      else
        PMPI_Allreduce(sendbuf+i*size, recvbuf+i*size,
        numelements/numthreads, newcomm[i]);
  }
}
```

Figure 5.3: Algorithm for Hybrid MPI_Allreduce

To illustrate this behavior, let us consider a very simple implementation of `MPI_Allreduce` structured in 3 steps: (i) each MPI task sends its data to a destination MPI task, (ii) this destination task performs the whole reduction of the received data and (iii) it broadcasts the final results to all MPI processes involved in the input communicator. In such a case, our approach will allow parallel execution of the main reduction within the destination task, but the communication pattern remains the same. Indeed, instead of performing one reduction, the algorithm will spawn multiple smaller reductions, but the underlying MPI library will eventually perform the same communications to send data to the destination task, on the same core. The reason is that each thread has a copy of the same input communicator, with the same ordering of MPI ranks. So all threads in a team have the same behavior when performing reduction. Thus this scheme is likely to generate memory contention and lead to communication imbalance, as all data involved in reduction are sent to a same NUMA node.

To tackle this issue, a second technique consists in giving a different role to each thread, by shifting their rank in sub-communicators.

Figure 5.4 explains this technique by describing behavior of MPI ranks involved in a reduction. It shows how the ordering of communicator implied in the reduction impacts the operation. We have here 4 MPI tasks, each containing a vector of elements. In the first example, the communicator contains the ranks of the tasks, starting from 0. So the task of rank 0 will be the root, and will reduce the value. If we shift the ordering of the communicator to the left, then the task of rank 1 will be the root. A rotation is performed so that rank 0 is the last one in the communicator. We run through the example, until the last case (where the first number in the communicator is 3). Then all tasks reduce to the rank 3. This example is relevant for our case because it addresses the first part of an *Allreduce* operation.

Thus each thread belonging to the same team will have a different rank inside their communicator. It can be seen as the extension to hybrid of the cyclical algorithm proposed by [79].

We illustrate this approach with Figure 5.5 exposing the same hardware configuration as Figure 5.2. In this example, each thread, with OpenMP rank equal to 0, will perform the same sub-reduction as the one on the right part of Figure 5.2. But the second thread of each team (those with OpenMP rank equal to 1) will use the second MPI rank as destination of their sub-reduction. We continue with this approach until we reach the last MPI rank. If the number of threads exceeds the number of MPI tasks, we do a

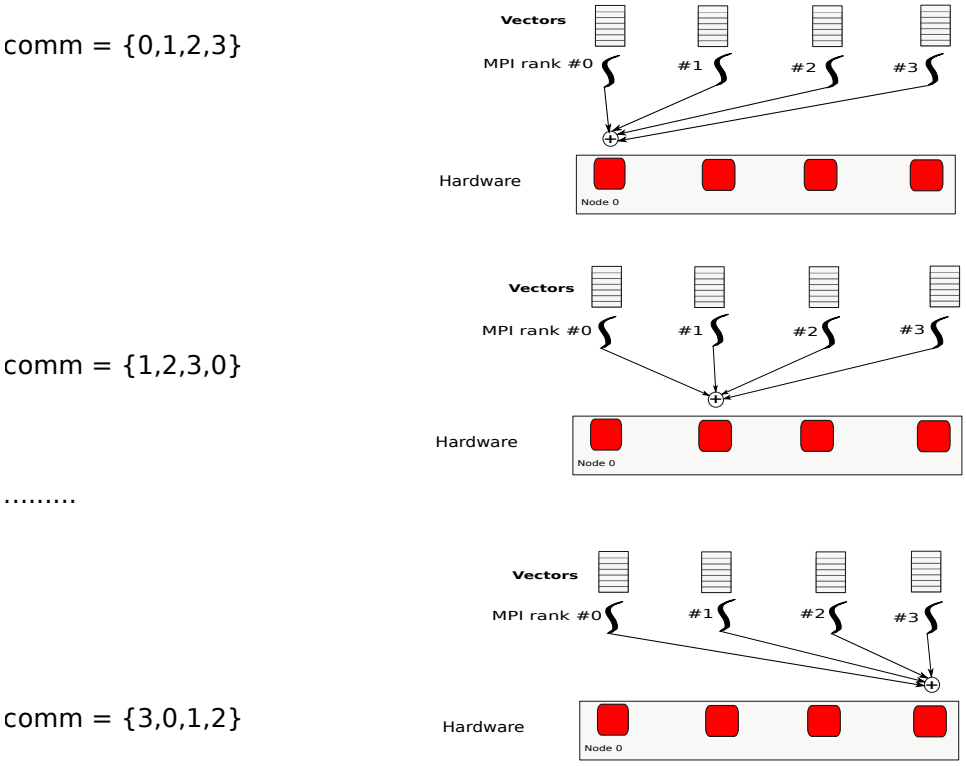


Figure 5.4: Different behavior of the reduction following ordering of the communicator

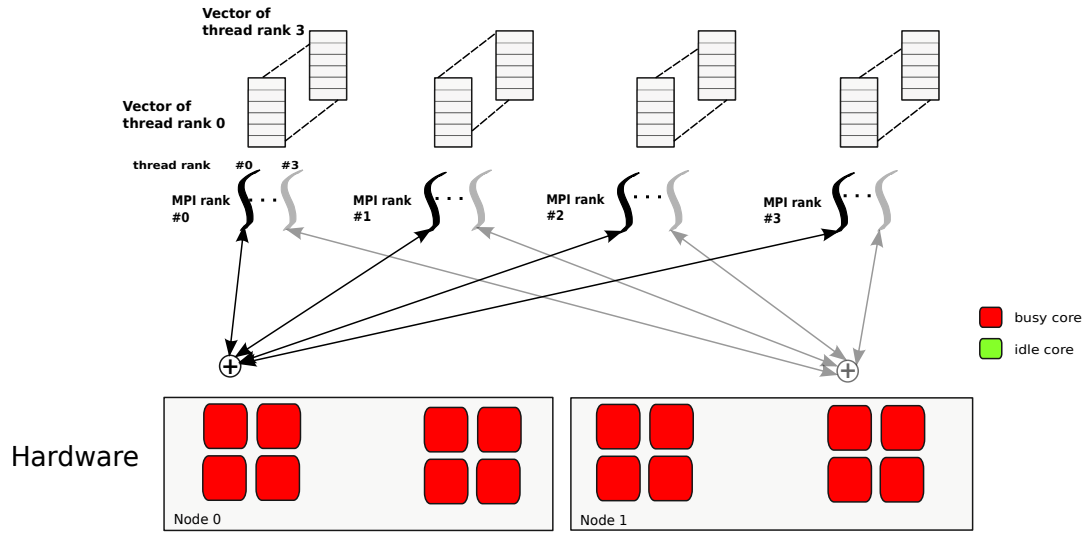


Figure 5.5: Rank Shifting of Hybrid MPI_Allreduce

rotation and go back to first task. In this example, we shifted the MPI rank of each OpenMP thread by 1. It is also possible to change the shifting value to adapt the memory traffic and the inter-node commu-

```
MPI_Comm newcomm[numthreads];
MPI_Comm lastcomm;

int initcomm = 0;
int rank, mpisize;

/* Reorder communicator for each thread */
void manage_comms(MPI_Comm comm)
{
 PMPI_Comm_rank(comm, &rank);
 PMPI_Comm_size(comm, &mpisize);

 if((lastcomm != comm) || (!initcomm)) {
   if(initcomm) {
    for(i = 0 ; i < numthreads ; i++)
     PMPI_Comm_free(&newcomm[i]);
   }

   for(i = 0 ; i < numthreads ; i++)
   {
    key = (rank+i)%mpisize;
    color = i;
    PMPI_Comm_split(comm, color, key, &newcomm[i]);
   }
   initcomm = 1;
 }

 lastcomm = comm;
}
```

Figure 5.6: Algorithm for Rank Shifting

nication pattern. It is important to add that this technique only works for non rooted MPI collectives.

The rank shifting appears inside the `manage_comms` function of the algorithm in Figure 5.3: when creating a sub-communicator for each OpenMP thread, we shift the target MPI rank, modulo the total number of MPI ranks, and store the result in a key. Thus, at the end of this function, each thread has its own communicator and each thread belonging to the same OpenMP team will have a different MPI rank inside their own communicator. Figure 5.6 illustrates the updated version of the `manage_comms` function.

5.3.3 Implementation

We implemented the algorithms of Figure 2 and 4 in a wrapper, which captures the calls to `MPI_Allreduce` collectives using the MPI Profiling Interface (PMPI [3]). This wrapper is compiled as a shared library and preloaded using `LD_PRELOAD` environment variable when running the target hybrid application. With this approach, our implementation is independent from any MPI implementation and works with all OpenMP implementations and MPI runtimes. The only requirement is related to the thread-level support in the target MPI library: in our wrapper, the calls to the MPI library are performed inside OpenMP regions, as depicted in Figure 2, it therefore requires the highest multithreading support level `MPI_THREAD_MULTIPLE`. Moreover our design guarantees a full support of regular types, derived data types, and communicators passed to the MPI_Allreduce collective.

5.3.4 Experimental environment

We evaluated our concept on the Curie supercomputer, with the following hardware configurations: the first configuration consists in 4 Bull Coherency Switch nodes, containing each 16 8-core CPU Nehalem-EX, clocked at 2.27GHZ. These processors are distributed on 4 modules, each containing 4 sockets. The second hardware environment is composed of 8 nodes, containing each 2 8-core CPU Sandy Bridge clocked at 2.7GHZ.

Experiments were performed on micro benchmarks and one real-world application (MC), using IntelMPI 4.1.3.048, MPC 2.4.1 (Multi Processor Computing [24]) and BullxMPI 1.1.16.6 (based on Open MPI 1.5.4.a1 [47]) libraries. Notice that the first ones (IntelMPI and MPC) both support the required `MPI_THREAD_MULTIPLE` multithreading level with Infiniband network while BullxMPI does not. For the OpenMP part, we used Intel OpenMP for IntelMPI (with the `KMP_AFFINITY` variable set to compact to keep the threads as close as possible) and the OpenMP runtime integrated to MPC for the experiments with MPC.

5.3.5 Microbenchmarks

We first consider the Intel MPI Benchmarks (IMB[2]) suite 3.2, which measures the time spent in MPI operations, for various message sizes. Because our target is an hybrid application performing MPI collective communications outside OpenMP region, we selected the message size on a range between 1MB and 16MB.

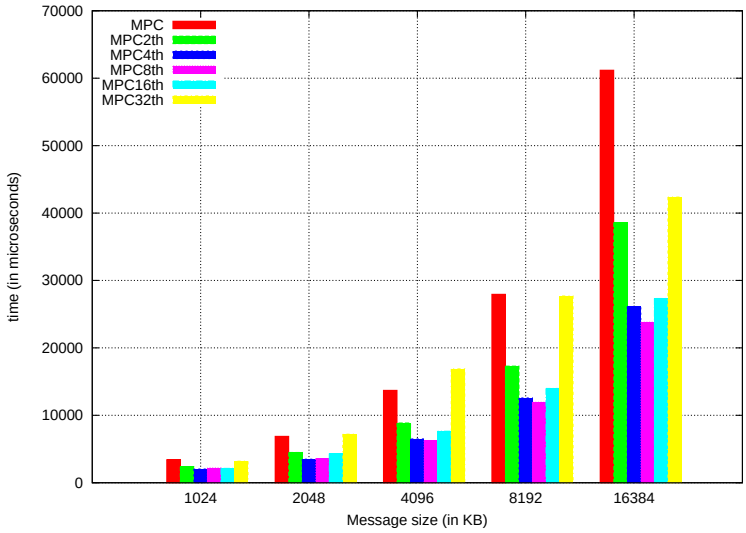


Figure 5.7: IMB - Hybrid MPI_Allreduce with MPC (4 128-core nodes, 1 MPI task per node)

Figure 5.7 depicts the latency of an hybrid `MPI_Allreduce` operation according to the message size with MPC and different numbers of OpenMP threads(see algorithm Figure 5.3). The most efficient combination is reached with 8 OpenMP threads leading to a speedup of 2.57 against regular implementation of `MPI_Allreduce` on a 16 MB message. This result can be explained because the size of the smallest NUMA node is 8. With more than 8 threads, we have penalties dues to OpenMP inter-socket communications and MPI NUMA effects.

Figure 5.8 shows the same experiment with the best thread configuration (8 threads) and rank shifting. In this test, the MPI rank of each OpenMP thread in the corresponding sub-communicator is shifted by 1 (see algorithm Figure 5.6). It provides an additional speedup of 18,6%.

The best hybrid configuration with IntelMPI is obtained with 90 OpenMP threads which leads to a speedup of 2.55 $\times$ compared to regular `MPI_Allreduce` (see Figure 5.9). This large number of threads can be explained by the fact that the Intel OpenMP implementation is well optimized for multicore processors.

Figure 5.10 summarizes the results with MPC, BullxMPI and IntelMPI within the same hardware configuration. Furthermore, this graph plots the best combination for MPC (8 OpenMP threads with shifted ranks) and IntelMPI (90 OpenMP threads). The best performance is reached by MPC providing a speed up of 41% against IntelMPI and 3.75 $\times$ compared to the reference MPI implementation BullxMPI.

5.3.6 Real World application

The second test case, MC, is extracted from [36]. MC is an hybrid MPI + OpenMP application solving the transport equation for neutronics, using Monte Carlo methods. The application is launched on an Adaptive Mesh Refinement grid. The code works on an unidimensional domain, which is replicated among MPI processes. It also contains an iterative loop where the system is computed and updated,

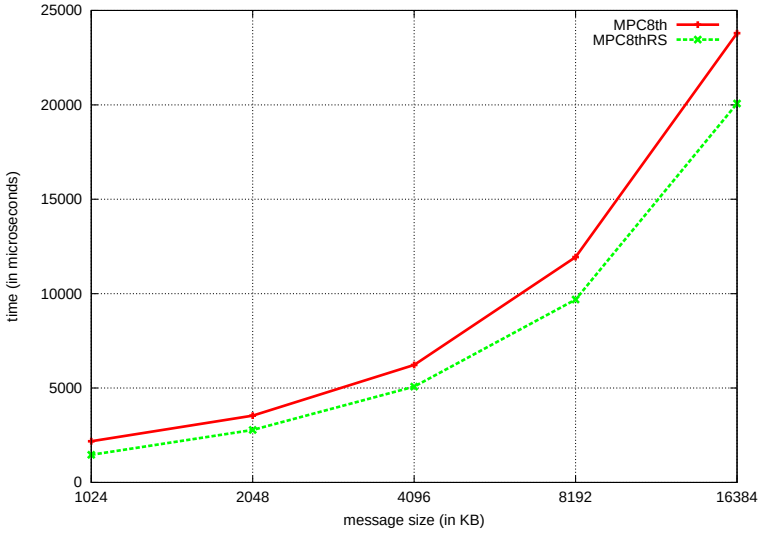


Figure 5.8: IMB - Hybrid MPI_Allreduce w/ and w/out rank shifting on MPC (4 128-core nodes, 1 task per node)

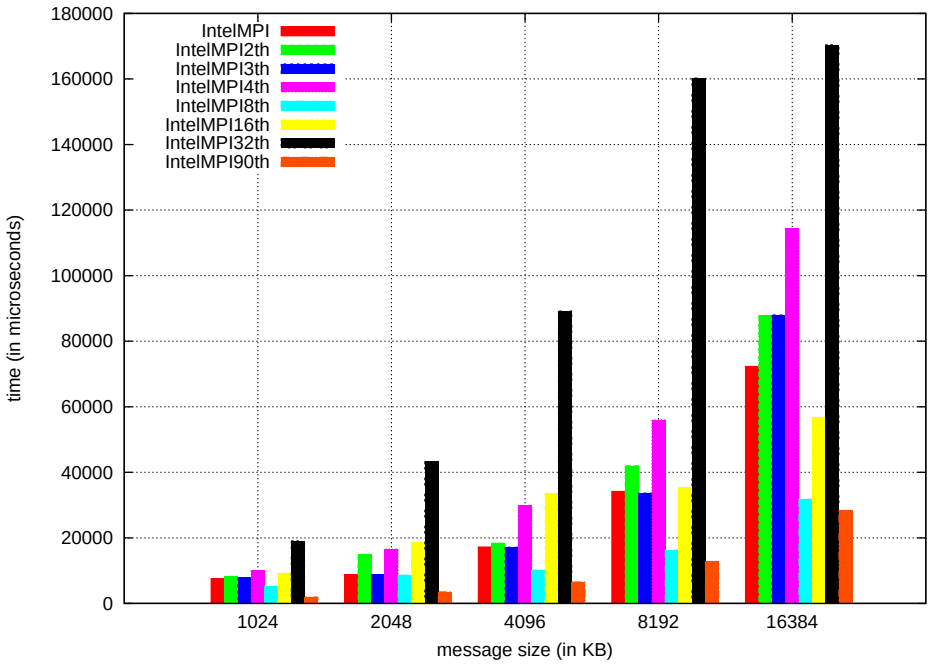


Figure 5.9: IMB - Hybrid MPI_Allreduce with IntelMPI (4 128-core nodes, 1 MPI task per node)

and lasts as long as particles exist on the domain. An MPI Allreduce collective is performed to update the number of available particles at each step. We tested our hybrid approach on this `MPI_Allreduce` collective operation, each MPI task containing a 128KB vector. We conducted our experiments on the same configuration as for the micro benchmark, we fully populated the nodes with OpenMP threads,

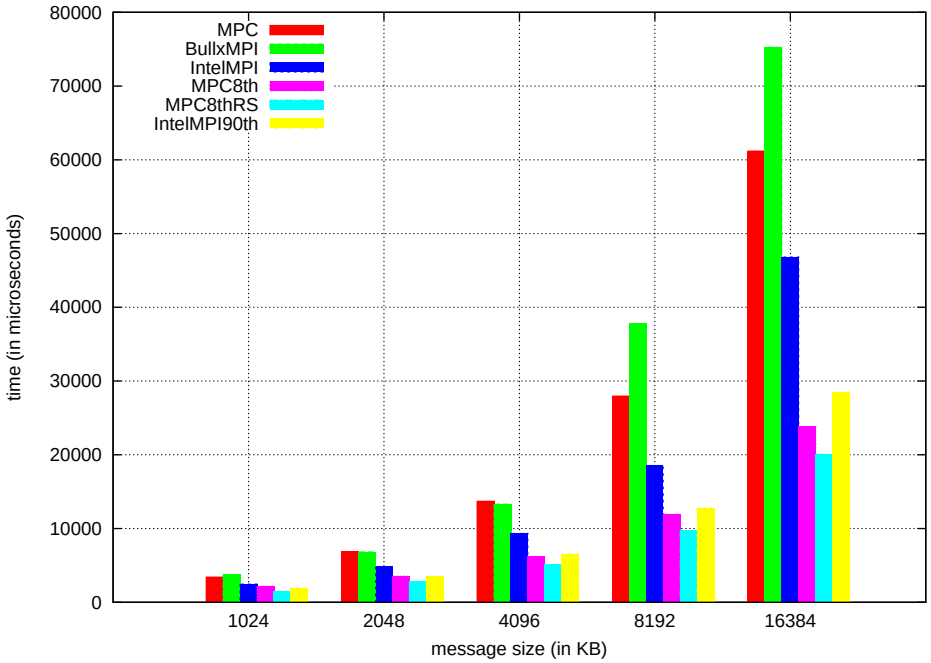


Figure 5.10: IMB - Comparison between MPC, BullxMPI, IntelMPI, and the best hybrid combination (4 128-core nodes, 1 MPI task per node)

and tuned the number of threads for the hybrid `MPI_Allreduce` collective.

Figure 5.11 depicts comparative execution times of the MC application, with 4 MPI tasks launched on 4 large nodes, on all MPI libraries. With MPC, using 8 OpenMP threads provides the best performance for the hybrid `MPI_Allreduce`. But an additional speedup of 68% is achievable by applying rank shifting (5 th bar on Figure 9), which gives a total speedup of 5.29. For IntelMPI, the best combination is obtained with 9 threads, leading to a speedup of 3.54. The lowest execution time is reached with MPC (3.9 better than OpenMPI) by performing hybrid `MPI_Allreduce` with 8 threads and appying a rank shifting.

Figure 5.12 presents the same study on eight 16-core nodes, with 1 MPI task per node. Once again, even if the total execution time of the application is larger with MPC, parallelizing the `MPI_Allreduce` collective operation with 8 threads leads to the best performance among the other configurations. Indeed, the performance increase is around 81% and shifting the rank gives an additional improvement of 41%. Similarly with IntelMPI, our approach is able to reduce the execution time from 242 seconds to 147 seconds using 8 threads for the hybrid `MPI_Allreduce` part. In all the test cases, no additional speedup could be obtained for IntelMPI by coupling rank shifting with the best optimizations of MPI Allreduce.

5.3.7 Conclusion and future work about optimizing MPI collectives

With this approach, we investigated the topic of optimizing MPI collectives in hybrid applications. We proposed a way to extend existing optimizations, targeting MPI + OpenMP applications, to parallelize the computational and communication parts, using spare cores. Experimental results showed a 2.57 speedup on micro benchmarks and a speedup of 5.29 on a real world application. Furthermore, this approach is portable to any MPI implementation which provides the support of `MPI_THREAD_MULTIPLE`

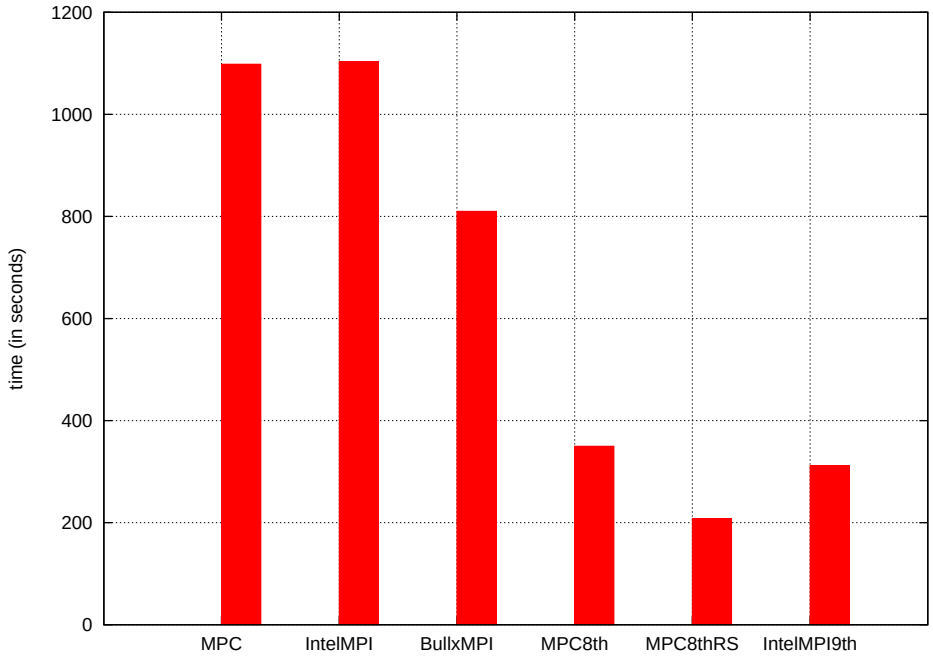


Figure 5.11: MC - Comparison between MPC, IntelMPI, BullxMPI, and best hybrid combination for MPC and IntelMPI (4 128-core nodes, 1 task per node)

and any OpenMP runtime.

5.4 Designing common constructs between MPI and OpenMP

The hybrid taxonomy introduced in Chapter 3 described different programming styles, including Coarse grain parallelism. While providing better performances than Fine Grain approach by a better exploitation of resources, it requires more programming effort, and is prone to bugs. Indeed, it is the responsibility of the programmer to launch both MPI tasks and OpenMP threads, and to ensure that MPI functions are called only once using synchronization directives.

In this section, we discuss opportunities for designing constructs mixing MPI and OpenMP features, in coarse grained codes. We then focus on codes mixing MPI and OpenMP, using SPMD method, and we show how common constructs can be relevant in this case.

We mentioned in Chapter 2 that PGAS languages mixed several paradigms for parallelizing computations, and included Collective operations in [6] for example. But to the best of our knowledge, no investigation has been made about designing common constructs implying different parallel programming models.

Need of simpler constructs with Coarse Grain parallelism. Coarse Grain parallelism, while overcoming limitations of Fine Grain approach, requires special care for code correctness. Examples of this specific attention can be to call functions or directives to start MPI tasks and OpenMP threads, or to surround MPI calls by OpenMP blocks, or else to use threads to split point to point or collective communications, etc. Moreover, since a single OpenMP parallel region is open in this mode, MPI tasks and OpenMP threads are launched approximately at the same time. But they both require their own constructs to launch tasks and threads. Consequently, there are some opportunities to simplify the code

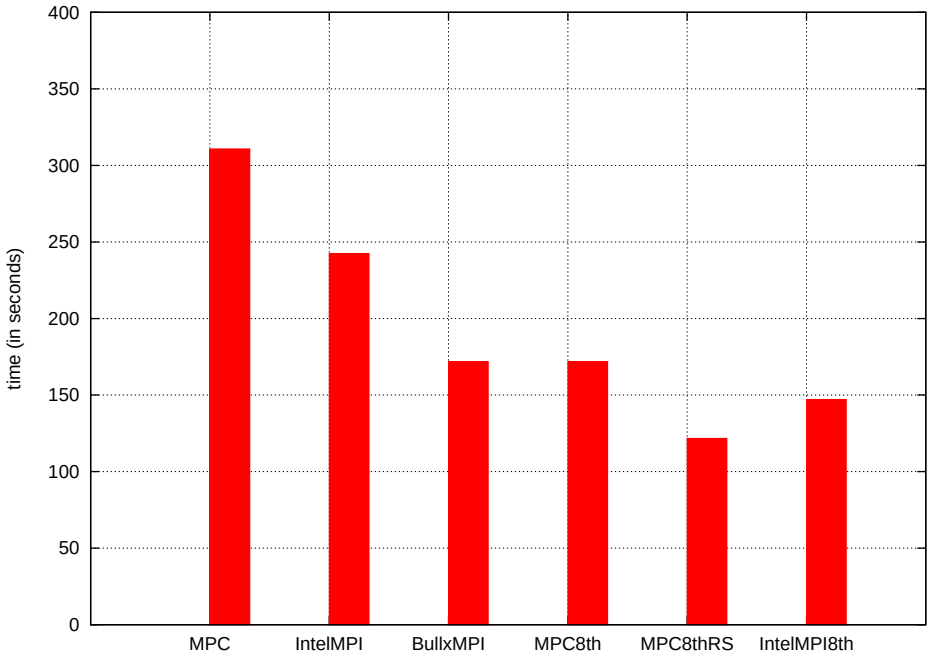


Figure 5.12: MC - Comparison between MPC, IntelMPI, BullxMPI, and best hybrid combination for MPC and IntelMPI (8 16-core nodes, 1 task per node)

and reduce the risks of introducing bugs. These examples call for a new language proposing features such as launching MPI tasks and threads at the same time, calling MPI primitives in a thread safe way, etc. This supposes to transfer such work as ensuring the correctness of the code and as activating MPI and OpenMP from the developer to the runtime.

We will consider that with the Coarse Grain approach, different paradigms co-exist, by mixing distributed memory parallelism with shared memory parallelism.

Cooperation between MPI and OpenMP SPMD programming. SPMD fashion, introduced in Hybrid taxonomy of chapter 2, describes a special kind of parallelism, as it allows to use OpenMP following a distributed memory paradigm. In a way similar to MPI, each thread computes its work in an independent manner. But this mode prevents from using regular OpenMP constructs. For example, when we want to perform a reduction using SPMD technique, instead of using the proper clause `reduction` combined with parallel loops, each thread computes a local reduction. These local reductions are then accumulated in a shared variable.

Some contributions in the literature show OpenMP can be used to implement features traditionally proposed by MPI. For example, in [65] the authors introduce different versions of NAS benchmarks, including one implemented with SPMD fashion, and being derived from MPI version. Operations such as point-to-point, reduction, or All-to-All are implemented with OpenMP and SPMD programming. To implement All-to-All operations with OpenMP, a global shared array is used.

A pure MPI version of the reduction is implemented using `MPI_Allreduce`, and is compared to others using OpenMP, including one implemented following SPMD fashion. They build the SPMD OpenMP version from pure MPI version of NAS Benchmarks. So with SPMD programming, we are able to implement some operations traditionally performed by MPI. This contribution shows we can use OpenMP with SPMD programming to implement operations such All-to-All, Gather or Broadcast, traditionnally implemented by MPI collectives.

If we go back to MPI + SPMD OpenMP mode, we observe that both models can do these operations

at the same time, but without cooperation between them. For example, each model would perform a reduction operation using the same patterns, but without sharing the results between MPI ranks and OpenMP threads. So, beyond reducing the code complexity, it would be of strong interest for the programmer to use constructs implementing the mentioned operations, and implying both MPI and OpenMP at particular steps of code execution.

```
int main(int argc, char **argv)
{

 MPI_Init(&argc, &argv);

 int iter;
 int size = 100;

 /* Iteration loop */
 for( iter = 0 ; iter < niter ; iter++)
 {

#pragma omp parallel
  {

    int a[100];
    int myOMPRank = omp_get_thread_num();
    int nbOMPThreads = omp_get_num_threads();
    int nbnLoc = n / nbOMPThreads;
    int iDeb = 1 + myOMPRank*nbnLoc;
    int iFin = iDeb + nbnLoc - 1;

    if(myOMPRank == nbOMPThreads - 1)
      iFin = n;

    /* Code */

    for( i = iDeb ; i < iFin ; i++)
    {
      a_in[i] = a_in[i] + a * niter;
    }

  } /* End of parallel region */

 /* Code */

     MPI_Reduce(a_in, a_out, n, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

 } /* End of iteration loop */

 MPI_Finalize();

}
```

Figure 5.13: Combination of MPI and OpenMP used as SPMD

What we learn from this contribution advocates for common constructs simplifying these operations and gathering both MPI and OpenMP. Indeed, combining MPI with OpenMP using SPMD programming will result in executing models with same paradigms and similar algorithms.

Prerequisites in runtimes. Designing constructs involving MPI tasks and threads in an application imply gathering MPI calls and OpenMP blocks inside the same function. Strong cooperation of both models are then required at runtime level.

For example, when considering Coarse Grain parallelism, a programmer has to handle multiple paradigms within the same code, like ensuring thread safety when calling MPI functions, or managing communications with several threads. Simplifying this kind of programming would consist, for the runtime, to take in charge serialization of MPI calls. In this case, we need to guarantee that the corresponding MPI internal is called with a single OpenMP thread, inside the runtime. To be able to design this kind of constructs, MPI and OpenMP can be implemented inside their own runtime. But it is necessary for each model to access the API one from the other. Most vendors propose separate implementations of MPI and OpenMP. This is the case of IntelMPI and Intel OpenMP runtimes. The ideal case is provided with MPC. Indeed, this framework offers a very easy way to design such constructs as it implements MPI and OpenMP inside the same runtime. This framework ensures a good cooperation between the different models. Thus it is simple to extend existing constructs.

The most convenient case is to have both MPI and OpenMP implemented inside the same runtime, so that we have a common view of both models.

5.5 Contribution: Unified collectives in runtime

In the previous section, we recalled characteristics coming with Coarse-Grained codes, and focused on the problematics when combining MPI and OpenMP used the following SPMD fashion. We observed that there was a need to simplify this kind of codes on the one hand, and a lack of cooperation between MPI and OpenMP for collective operations, on the other hand. To the best of our knowledge, no common constructs mixing different programming models have been proposed yet.

We introduce here our concept of unified collectives, which consists in proposing common constructs for MPI and OpenMP achieving traditional MPI collective operations. Examples of these collectives are:

- Reduction
- Barrier
- All-to-all

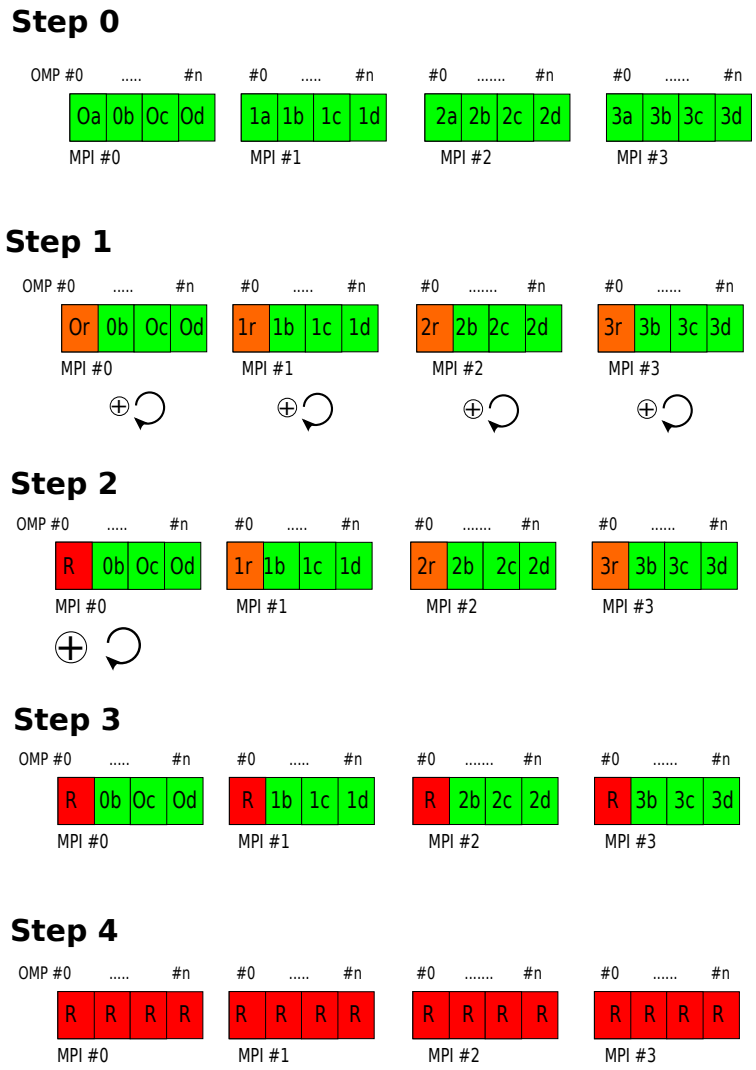


Figure 5.14: Example of unified Allreduce operation

Let's start by describing a specific example with *Allreduce* operation. We take the case of an application running 4 MPI tasks, each containing a team of 4 OpenMP threads, as depicted on Figure 5.14. The

```
int main(int argc, char **argv)
{

 MPI_Init(&argc, &argv);

#pragma omp parallel
 {

 /*  Code to be executed */

#pragma omp barrier

#pragma omp single
  {
    MPI_Barrier();
  }

 /* Code to be executed */

 }

 MPI_Finalize();

}
```

Figure 5.15: Global MPI+OpenMP barrier without optimization

figure describes the different steps implied by *Allreduce* operation, involving MPI ranks and OpenMP threads. We will consider the reduction must concern one element per MPI task and OpenMP thread, and MPI task of rank 0 is the temporary root of all tasks. Step 0 is the initial stage. Each OpenMP thread has computed a local reduction. At step 1, each MPI rank computes a temporary reduction from all local ones of its OpenMP team. At the end of the operation, Master threads of all OpenMP teams have their variables `0r`, `1r`, `2r` and `3r` updated with computed temporary reduction. At step 2, all MPI ranks perform a global reduction (as would be done by `MPI_Reduce()`) starting from values of `0r`, `1r`, `2r` and `3r`. At the end of the global reduction, the MPI task of rank 0 contains the global reduction `R`. In step 3, the root MPI task broadcasts the result of the reduction to all MPI tasks. Step 2 and 3 describe a classic `MPI_Allreduce` operation, so that these steps can be fused. In the last step, the reduced value `R` is shared between all OpenMP threads of each team. The *Allreduce* operation is finished.

This example showed how MPI ranks and OpenMP threads could cooperate to perform global operation, combining basic operations.

5.5.1 Case study of unified barrier in hybrid MPI+OpenMP context

We presented unified collectives in order to perform global operations implying all tasks, with the example of the reduction. To show the feasibility of this contribution, we present here a proof of concept which is a unified barrier. Unified barrier consists in synchronizing at the same time all MPI tasks and all OpenMP threads of each team, with a single construct.

The goal of our prototype is to perform this synchronization with one single directive, and compare it with a regular synchronization, i.e. a barrier implying MPI tasks and threads with traditional constructs. Algorithm 5.15 describes a global barrier involving regular MPI and OpenMP constructs. In an OpenMP parallel region, several MPI tasks are launched, each executing a team of threads. We first encounter an OpenMP barrier synchronizing all threads. To make sure only one OpenMP thread of each team executes`MPI_Barrier()`, the call to `MPI_Barrier()` is surrounded by a `#pragma omp single`. An implicit barrier is done for each OpenMP team at the end of `single` construct.

But these synchronization points do not ensure that all threads of all MPI ranks are synchronized at the same time. To understand this, let's consider all threads of a single team. They will all execute single construct, and the first thread encountering the construct will go inside and will execute `MPI_Barrier()`. Other threads will wait at the barrier terminating `single` construct. But what happens in the case the threads within the same team are distant from each other ? All teams will not finish implicit OpenMP barrier at the same time, so the global barrier is not respected. Consequently, we have to insert an additional OpenMP barrier before executing `single` construct, in order to ensure that all threads are well balanced before executing `MPI_Barrier()`.

We saw that implementing a unified barrier with regular constructs is not straightforward and offers room to introduce bugs. We propose here to replace this complex implementation by a single function

call which is our optimized hybrid barrier.

5.5.2 Implementation

We chose to develop our optimized hybrid barrier inside MPC framework, because it offered a very convenient way to mix MPI and OpenMP internals inside the same function. We implemented it starting from current implementation of OpenMP barrier.

Algorithm 9 Global MPI+OpenMP barrier

Require: $tree, thread$
1: $node \leftarrow thread.father$
2: $bdone \leftarrow node.barrierdone$
3: $b \leftarrow node.barrier$
4: atomic($node.barrier \leftarrow node.barrier + 1$)
5: **while** $b + 1 = node.barriernbthreads$ **and** $node \neq tree.root$ **do**
6: $node.barrier \leftarrow 0$
7: $node \leftarrow node.father$
8: $b \leftarrow node.barrier$
9: atomic($node.barrier \leftarrow b + 1$)
10: **end while**
11: **if** $node.father \neq NULL$ **or** $node.father = NULL$ **and** $b + 1 \neq node.nbthreads$ **then**
12: **while** $bdone = node.barrierdone$ **do**
13: wait thread
14: **end while**
15: **else**
16: atomic($node.barrier = 0$)
17: $node.barrierdone = node.barrierdone + 1$
18: call MPI_Barrier()
19: **end if**
20: **while** $node.typechildren \neq LEAF$ **do**
21: $node = node.child[thread.ranktree[node.depth]]$
22: $n.barrierdone = node.barrierdone + 1$
23: **end while**

Algorithm 9 is based on another one, from OpenMP barrier presented in Chapter 4. It describes the implementation of our optimized global barrier. This algorithm, initially implementing the barrier in a hierarchical manner, is also tuned to synchronize MPI tasks. We insert a call to `MPI_Barrier` at a specific point of the algorithm, and ensure that one single thread executes this function. To explain how we chose the location to call MPI barrier, let's remind the hierarchical implementation. Each thread, when entering the barrier, will climb the tree by encountering intermediate barriers when reaching nodes. When the root is reached, only one thread is executing the algorithm at this point. Thus, this thread calls `MPI_Barrier()`, all threads cross the tree to the leaves and continue their execution.

To understand the exposed algorithm, let's detail the different stages of the algorithm of our unified barrier. We take here the example of 2 MPI tasks and 2 OpenMP teams containing each 8 threads, encountering the global barrier.

At first stage (Figure 5.16), the threads of each OpenMP team encounter the barrier relative to each team. Then, they climb the tree until reaching the tree root.

Once the last thread of each team reached the tree root, they call the MPI barrier (Figure 5.17), and wait until MPI barrier finishes its job.

When MPI barrier is done, calling OpenMP threads go down inside their respective tree, freeing other threads until reaching the leaves, and threads continue their execution (Figure 5.18).

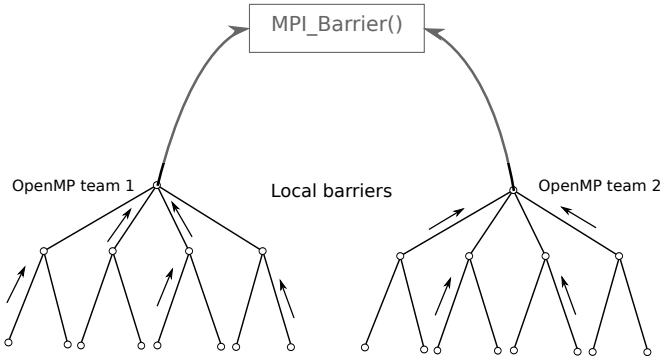


Figure 5.16: First stage of unified barrier: Barrier of OpenMP teams

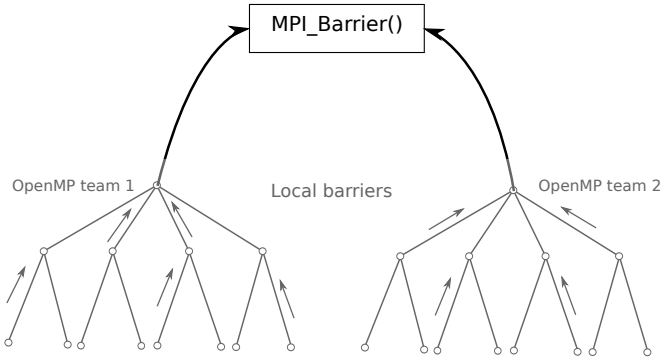


Figure 5.17: Second stage of unified barrier: Call MPI barrier

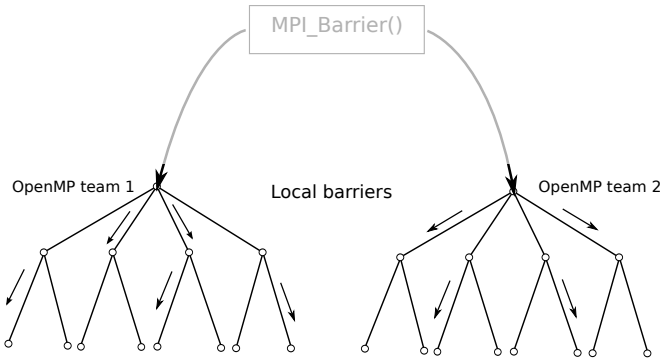


Figure 5.18: Third stage of unified barrier: Release OpenMP teams

5.5.3 Experiments

To validate our concept of hybrid barrier, we used EPCC microbenchmarks and added new tests, one executing an hybrid barrier involving regular MPI and OpenMP constructs as well as our optimized version calling the associated construct. We compared both versions on the Curie supercomputer, on one 128-core Bull Coherency Switch node. The range of launched MPI tasks varied from 2 to 64, populating other cores with OpenMP threads.

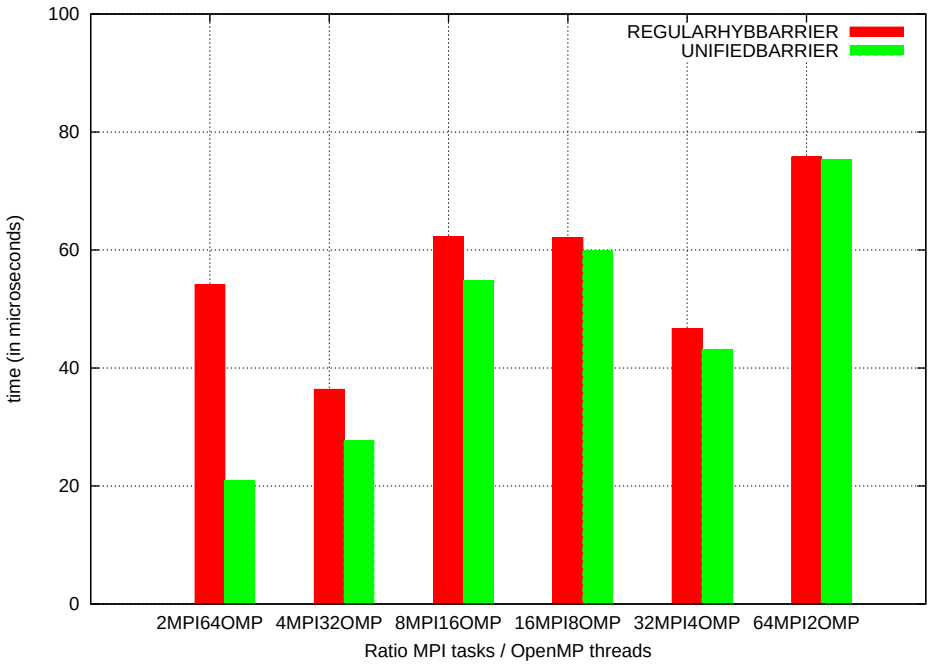


Figure 5.19: Comparison between hybrid barrier and optimized hybrid barrier implemented in MPC (1 128-core node)

Figure 5.19 presents the performances of our optimized unified barrier against the regular one. We gain speedup on all configurations, but we observe that performances of both versions are somehow similar with 64 MPI tasks, whereas the biggest difference of performances is seen with 2 MPI tasks.

In order to analyse the results, we described the different components of regular and optimized versions of the unified barrier through a timeline, in figure 5.20. With the regular one, each OpenMP team has to pass a first OpenMP barrier. We assume there is a small overhead due to `#pragma omp single` construct, and then, we call the MPI barrier. At last, all OpenMP teams have to pass the last OpenMP barrier related to `single` construct. But with the optimized unified barrier, OpenMP threads have just to pass half of the barrier before reaching the tree root. Then one thread executes MPI barrier, and all pass the other half of the tree.

So, when analysing general performances, it is not suprising that our optimized construct gives better performances, since optimized barrier avoids the call of an additional OpenMP barrier. We also assume that time spent in MPI barrier increases with the number of involved tasks in the barrier, and that its cost is bigger than the one of OpenMP barrier, even if the MPI barrier was optimized for shared memory in MPC. This is why, with 2 MPI tasks and 64 OpenMP threads, we observe the biggest discrepancy between the two versions. Indeed, this configuration highlights the advantages of our version against the regular one. But if we consider the configuration involving 64 MPI tasks and 2 OpenMP threads,

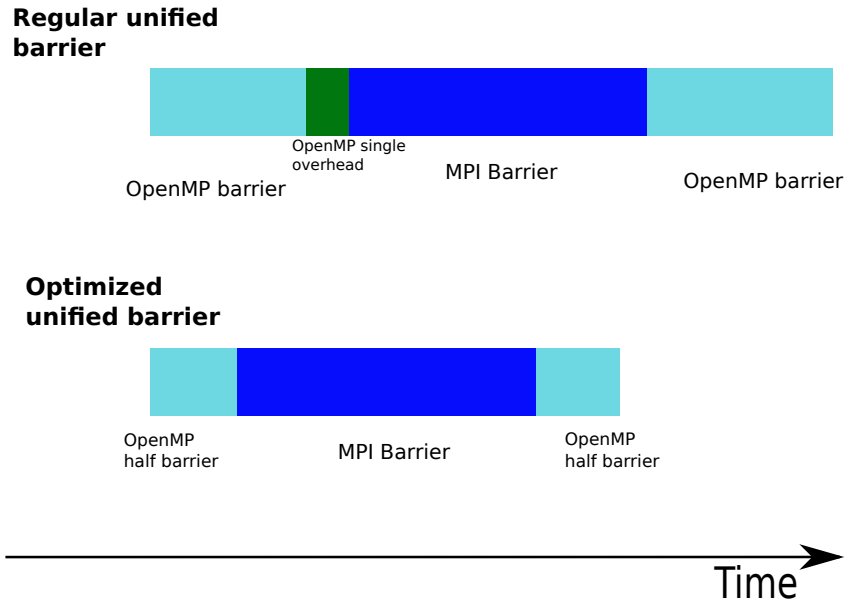


Figure 5.20: Timeline comparing execution of regular and optimized unified barrier, with the different stages

we can say overhead due to OpenMP barriers is negligible and most of the time is spent in MPI barrier. Furthermore, if we assume that calling an MPI barrier is much more expensive than an OpenMP barrier, the optimization of our concept can be neglected regarding the cost of `MPI_Barrier` with 64 tasks.

5.5.4 Conclusion and Future work about Unified Collectives

We recalled characteristics of Coarse grained codes, and especially MPI mixed with OpenMP used in a SPMD style. While this pattern gives better performances than Fine grained model, one shortcoming is the complexity of programming applications with such modes. We highlighted the behavior of OpenMP in SPMD approach and its similarities with MPI, each OpenMP thread having a particular role. Literature showed it was possible to design collectives involving pure OpenMP codes. We then introduced constructs gathering both MPI tasks and OpenMP threads for collective operations traditionally encountered with MPI. An example of such unified constructs was given with *Allreduce* operation. We showed the feasibility of Unified Collectives by giving the example of hybrid barrier. This proof of concept consisted of synchronizing all threads and all MPI ranks with one single directive, implemented inside the MPC runtime. Its implementation was very simple, thanks to MPC framework, and gave speedup against a regular implementation of MPI+OpenMP barrier.

5.6 Conclusion of studying Collective operations in hybrid context

In this chapter, we tackled various issues preventing from reaching high performances in Hybrid programming, including:

- Idle cores in Fine Grain scheme, outside OpenMP regions
- Part of sequential and communications components in total execution time
- Complexity of code in Coarse Grain approach
- Necessity of cooperation between OpenMP and MPI in SPMD programming

We gave solutions to these shortcomings by conducting a global study of collective operations, for different purposes. We first investigated optimization of MPI collectives and applied several techniques for Hybrid programming, in order to reuse idle cores and reduce communication time. We then studied characteristics of SPMD programming using OpenMP and common operations between MPI and OpenMP.

Our contributions were twofold:

- Optimize performances of regular MPI collectives in the context of Fine Grain parallelism by reusing inactive threads outside OpenMP constructs. Our contribution on parallelizing `MPI_Allreduce` operation with threads showed promising results. It should be extended by investigating deeper presented techniques, and should address other MPI collectives (reduction, MPI Broadcast, etc). At last, activating OpenMP threads inside the runtime instead of within a wrapper would allow to parallelize the collective in a transparent way for the developer.

- Introduce unified collectives which consist in making MPI ranks and OpenMP threads work together in collective operations. This contribution is relevant in SPMD programming style where the programming effort is the most important. Our optimized unified barrier as proof of this concept allowed to gain speedup compared to the regular unified barrier. With these constructs, we seek out to reduce the programming effort in the SPMD pattern and to make the runtime take in charge operations such as synchronizing the calls to MPI primitives or splitting off the MPI calls with threads. This could lead to a new language mixing models with different paradigms.

Chapter 6

Analyze performance of MPI+OpenMP applications at user level and runtime level

The hybrid taxonomy introduced in Chapter 3 highlighted the complexity of hybrid codes. Depending on the various programming configurations, the performance pitfalls can be of various natures, and may come from both the application side or the runtime side. Moreover, due to the mix of the two programming models, they are difficult to isolate. Having proper tools to detect such pitfalls is critical to enable hybrid programming.

In this chapter, we focus on the performance bottlenecks that can exist in hybrid codes, we introduce performance analysis as a research field, and we detail its different components. We then present existing tools targeting both MPI and OpenMP codes. Afterwards, we focus on the instrumentation part and look for ways to instrument both models. If MPI proposes a standard API via `MPI_T` for high performance codes, this was not the case for OpenMP. Then, we present the newly introduced OpenMP Tools API, included in the OpenMP standard, and its features enabling an instrumentation of OpenMP codes and an insight in OpenMP runtimes. At last, we present our contribution, which first consists first in the implementation of OMPT inside the MPC framework, and then in analyzing the performances of hybrid MPI+OpenMP application using two different implementations of OpenMP Tools, including ours.

6.1 Scalability of high performance codes

We estimate performance of High Performance codes by their scalability, in other words how they can take advantage of numerous compute cores. But it is hard to reach scalability on massively parallel machines, as the current supercomputers approximately reach one million core counts. These difficulties can come, for a large part, from the characteristics of these machines: their hardware topology, the multiple levels of parallelism, their network latency, etc. Optimizing parallel codes to efficiently exploit such architectures is a long and difficult process, and the performance bottlenecks have to be identified.

These bottlenecks of parallel codes can be:

- Limited extensibility of algorithms implemented in applications
- Load imbalance in parallel codes
- Inefficiency of load balancing in OpenMP runtimes
- Implementation of MPI algorithms in MPI runtimes
- MPI library not optimized for intra node communications
- Bad data locality

- Memory consumption

As we can observe in this list, the performance pitfalls are numerous and can come from various directions. Limitations can come from both application or runtime level.

Moreover, adding OpenMP pragmas inside MPI codes means adding another level of parallelism, and establish coexistence between two runtimes. This leads to increase potential bottlenecks. These are then difficult to pinpoint and require some expertise.

Performance tools are necessary to get insight of parallel codes and runtimes.

6.2 Overview of performance analysis infrastructure

The previous section showed the difficulties encountered by the developers to track potentiel bottlenecks in numerical codes, which revealed the necessity to use proper tools for performance analysis.

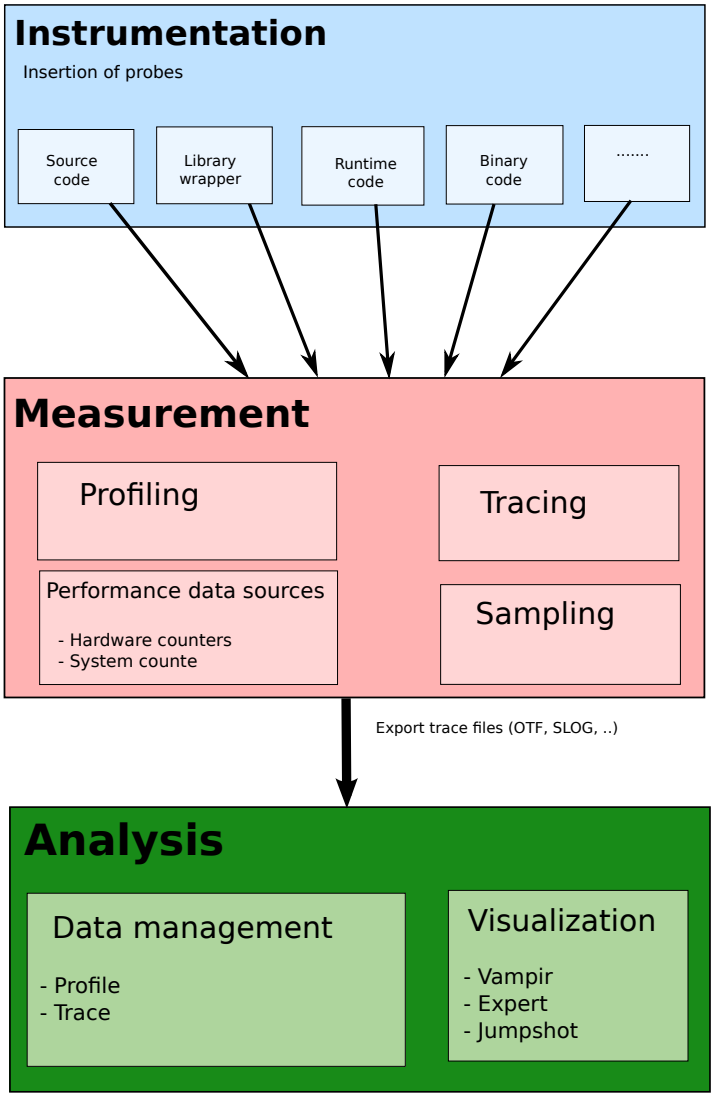


Figure 6.1: Infrastructure of performance analysis

Figure 6.1 describes the infrastructure of performance analysis regarding high performance applications. We can split the chain in three stages: (i) instrumenting application at several levels, (ii) measuring parts of the application using different techniques and performance metrics, and (iii) analysis of the measurement.

6.2.1 Instrumentation

In order to gain information from an application for performance analysis, some probes have to be inserted. This stage is called instrumentation. While the program is executed, events related to these probes are activated and detected by the tool which will perform measurements. We will see here that the instrumentation can be done at several levels of the code, and different kinds of instrumentations have to be distinguished:

- **Source-level instrumentation**: This technique consists in inserting function calls in the source code for instrumentation. The source code has to be transformed by a source-to-source compiler which will insert probes at specific points. For example, in the case of an OpenMP code, constructs will be surrounded by function calls. This is the case with OPARI2 [83] which uses the functions provided by the POMP interface.

- **Compiler instrumentation**: In this case, the instrumentation probes are added by the compiler, inside the object code. This kind of instrumentation benefits from a higher level of details and allows fine grain analysis. However, it can prevent from other optimizations usually offered by the compiler, since they can corrupt measurements.

- **Library wrapper**: This kind of instrumentation consists in interposing a library between user code and runtime. If we take the case of MPI codes, MPI calls are intercepted by the library, which is in charge of calling internals. PMPI is an example of used interface to be plugged to a library. The advantage is that this technique is transparent for the developer.

- **Binary instrumentation**: Another instrumentation method consists in patching binary codes, and inserting probes at instruction level, allowing finer grained instrumentation. Examples of tools proposing this kind of instrumentation are PIN[92] and MAQAO[32].

6.2.2 Measurement

Instrumentation helps the tools to measure performance of applications using various methods and metrics. At measurement stage, the events are linked with performance data.

We distinguish several kinds of measurement:

- **Profiling**: A first method is profiling, which consists in aggregating performance metrics in order to characterize behavior of the targeted application. One example can be to measure the time spent in the different functions composing the studied application (inclusive or exclusive time).

- **Tracing**: Whereas profiling is able to summarize performance of applications via metrics, another method named tracing consists in building a view of the evolution of the application through a time-line, and in studying its behavior through spatial and temporal aspects. Tracing method generally relies on events capturing activity of the application.

- **Sampling**: Sampling consists in periodically collecting statistics from the program through execution time, via inserted interrupts.

In order to provide metrics, performance tools can rely on performance data sources, such as hardware counters. Hardware counters are low-level performance metrics and are integrated in current CPUs. They can give more insight about the behavior of studied applications: examples of those counters can be the number of cache hits or misses, or the number of executed instructions per clock cycle,.... PAPI[73] is an example of interface giving access to hardware counters. To be portable across platforms, it comes with its own API.

When measurements such as profiling or tracing have been built, they can be sent to the analysis component. This can be done by exporting results in files using formats such as Open Trace Format (OTF)[64] or SLOG[26].

Measurement overhead. Events allow to observe the behavior of the targeted code, and to measure performance of key parts. The selection and the number of these events depend on the performance resolution the user wants to enable. But instrumenting parallel codes is not free and can insert a bias in measuring, as instrumentation represents a performance intrusion. And the bigger the number of inserted events in instrumented code is, the higher the probability is to alter behavior of applications. This is called performance perturbation ([78]). But most performance tools provide techniques to reduce the measurement overhead.

6.2.3 Analysis

The role of the analysis component is to enable an empirical performance evaluation of high performance applications, and to display it. Depending on the tools, features coming with the analysis part can be more or less rich, and include a large set of hints in order to characterize the studied codes. At first, outputs from measurement component such as profile and trace data are merged to build a comprehensive view of the application behavior. With some tools, databases are also included to compare different executions of the same application. For example, the Performance Data Management Framework (PerfDMF) [53] allows storing, querying and analyzing performance data from multiple executions or application versions. At last, several utilities are proposed for display purpose. Vampir[85] allows to display traces of MPI based applications. It also offers statistic features. CUBE [100] is a performance analyzer relying on a data model and provides performance metrics. Other examples are Paraprof[15] or Jumpshot[109].

6.2.4 Existing performance tools for hybrid codes

Several performance tools gather numerous features and kinds of instrumentations, measurements, and analysis capabilities, among those described in the previous sub-section.

Presentation of TAU. TAU (Tuning and Analysis Utilities [98]) provides the full chain (instrumentation, measurement and analysis) for performance analysis of hybrid MPI+OpenMP codes. This tool is able to be plugged to various interfaces such OPARI2, or OMPT, an interface which will be studied later. So we see that various instrumentation techniques available such as source to source or binary instrumentations. TAU also includes several measurement techniques such as profiling and tracing. It can also rely on hardware performance counters. At last, it comes with several tools such as paraprof or jumpshot for visualization purposes.

Scalasca. Scalasca[44] is another example of performance tool, able to analyze hybrid codes. This tool proposes several analysis features such as runtime summarization by accumulating statistics and postmortem analysis with the help of event traces. For visualization purposes, Scalasca comes with Cube.

6.3 Prerequisites in instrumenting MPI+OpenMP codes

Performance bottlenecks of hybrid codes can come from numerous directions, as difficulties to write efficient applications are leveraged by the mixing of different programming models. In order to have a full observation of hybrid codes, it is necessary to have tools that are able to instrument both MPI and OpenMP codes.

Instrumentation of MPI codes. MPI Tool Information Interface (`MPI_T`) offers a standardized mechanism to gain information from MPI runtimes. It was introduced with MPI 3.0 revision and works through PMPI interface [3]. `MPI_T` communicates with the MPI runtime through control and performance variables. Control variables allow tuning operations such as eager limits, or the selection of algorithms implementing collectives. With performance variables, the user can get information such as memory consumption of MPI library.

Instrumentation of OpenMP codes. As standard interfaces are available to instrument MPI codes, this was not the case with OpenMP codes until recently. Different approaches exist for OpenMP instrumentation at different levels. The next section exposes the related work concerning the existing tools to instrument OpenMP codes.

6.4 Related Work about OpenMP tools for performance analysis

As mentioned in the previous section, different methods exist for instrumenting OpenMP codes.

[57] presents a binary instrumentation methodology to monitor runtime events, integrating the MAQAO performance tool into Score-P [7]. It introduces a new interface named SMOMP, which provides a set of probe functions for OpenMP regions.

Still within MAQAO, the authors in [14] describe features consisting in inserting probes at the instruction level of the codes for static analysis of OpenMP codes.

In [54], the authors compare different approaches to instrument OpenMP codes:

- A first method, OpenMP Pragma And Region Instrumentor (OPARI) [83], is a source-to-source translator which locates the OpenMP constructs and inserts instrumentation via the POMP interface. POMP OpenMP Performance Monitoring Interface was also introduced in [83] and defines a portable API for performance tools. Opari is then able to provide tools for OpenMP activity, but without any insight inside the runtimes. At last, source instrumentation is likely to generate some overhead and prevent compiler optimizations.
- A second approach, built into OpenUH compiler, is introduced, OpenMP Runtime API [56], commonly known as Collector API, and allows performance tools to interact with OpenMP runtimes. Callbacks are to be registered for event transitions in the runtime, and thus get insight into it. ORA is supported by TAU.
- The authors also present a method consisting in a library interposed between the application and libGOMP, the runtime library coming with GCC[45]. This technique is possible via a utility coming with TAU, which generates libraries.
- At last, their paper describes OpenMP Tools API (OMPT), which provides a set of events and states. It is based on the same principle than ORA but provides more events. It is integrated in the OpenMP standard.

OMPT takes advantage of these different approaches: it can be considered as a superset of ORA, since more events are provided for tools. Moreover, it is now integrated into the OpenMP standard. In the next section, we will study this instrumentation tool, its features and how we can take advantage of it to measure both application and runtime performances.

6.5 OpenMP Tools API

We recall the need of a standardized interface in order to analyze the performance of OpenMP codes. Starting from the observation of a gap between OpenMP applications and the behavior of OpenMP implementations, the OpenMP Architecture Review Board introduced OMPT as an extension of the OpenMP standard. OMPT (OpenMP Tools API [37]) is an Application Programming Interface, oriented towards performance analysis. This API was defined to provide the measurement of OpenMP applications and mirror the behavior of OpenMP runtimes. For this purpose, it collects performance measurements for tools, and provides a set of events and states.

Events are associated to OpenMP constructs, and are, for most part, organized according to begin/end pairs: they have to be placed from creation to completion of associated OpenMP constructs. Also they are split into two classes: mandatory or optional, and are notified when threads encounter OpenMP constructs such as parallel regions. Performance tools are then able to build profilings and tracings, based on these events. But to receive them, the tools have to interact with the runtimes via

OpenMP construct	Associated events
void	ompt_event_thread_begin / ompt_event_thread_end
#pragma omp parallel	ompt_event_parallel_begin / ompt_event_parallel_end
#pragma omp for	ompt_event_loop_begin / ompt_event_loop_end
#pragma omp barrier	ompt_event_barrier_begin / ompt_event_barrier_end
#pragma omp single	ompt_event_single_in_block_begin / ompt_event_single_in_block_end
	ompt_event_single_others_begin / ompt_event_single_others_end
#pragma omp master	ompt_event_master_begin / ompt_event_master_end
#pragma omp task	ompt_event_task_begin / ompt_event_task_end
#pragma omp taskwait	ompt_event_taskwait_begin / ompt_event_taskwait_end

Figure 6.2: Corresponding OMPT events to OpenMP constructs

callbacks: they register the callbacks associated with the events they want to be notified.

OMPT also describes some states of the runtime, referring to the state information of the executing thread, whether the current thread is executing serial code or parallel code, or is idle. . . . These states are implemented as variables which are updated in accordance the evolution of the current thread, and are not linked to any temporal information. They can be used for sampling techniques for example.

At last, OMPT provides unique identifiers to threads, parallel regions, and tasks, and also provides some inquiry functions to get these identifiers.

Thus, this API is able to feed the tools with thread activites and status, for profiling, tracing or sampling purposes.

Unlike other instrumenters like OPARI2 which performs source-to-source transformations, OMPT is to be implemented inside OpenMP runtimes. It is thus not visible from the source code but can require changes inside the compiler if the ABI[1] supported by the runtime does not fit the OMPT events. We will detail this last point later.

6.5.1 Targeted OpenMP constructs and provided events

Events provided by OMPT cover all OpenMP constructs and are classified into two categories: mandatory and optional. The most common provided events are `ompt_event_thread_begin` / `ompt_event_thread_end` which notify when a thread is created and terminated.

Here is a list of the main events provided by OMPT, with the corresponding OpenMP constructs:

- Creation and termination of threads: `ompt_event_thread_begin` / `ompt_event_thread_end`
- Creation and termination of parallel regions: `ompt_event_parallel_begin` / `ompt_event_parallel_end`
- Entering and exiting a parallel loop: `ompt_event_loop_begin` / `ompt_event_loop_end`

Figure 6.3 presents the example of an OpenMP parallel region and shows where related event callbacks have to be inserted. For a better understanding, we listed a user code, but in a real case, callbacks should be inserted in the runtime code. We see that a parallel region is open, and the callback related to `ompt_event_parallel_begin` is placed at the beginning of the parallel region, while the one related to `ompt_event_parallel_end` is placed just before the closing bracket. These callbacks have to be executed by the master thread, in charge of creating and terminating the parallel region.

A pair of events is also provided and has to be used when the current thread starts executing its associated code and when exiting it.

OMPT also supplies events in order to measure how much time a given thread waits inside a construct (like barriers, or OpenMP locks).

At last, some events are available to get insight of the runtime. One of these provided events defines an implicit task as the code associated to a parallel region, which corresponds to the code

[1] Application Binary Interface

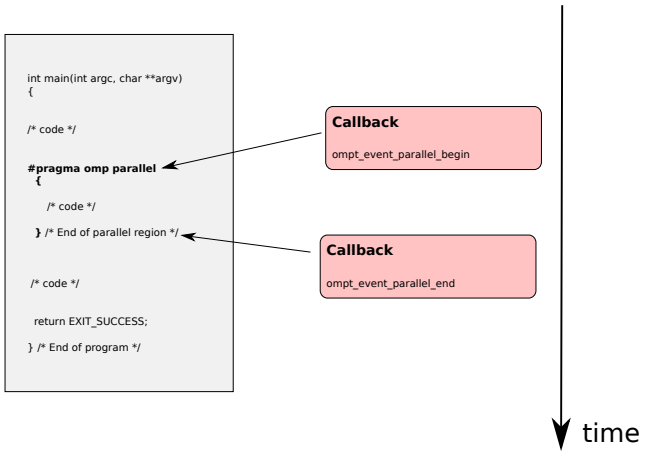


Figure 6.3: Insert event callbacks related to parallel regions

inside brackets. This implicit task is not visible from user code, and can be instrumented by the pair `ompt_event_implicit_task_begin` / `ompt_event_implicit_task_end`

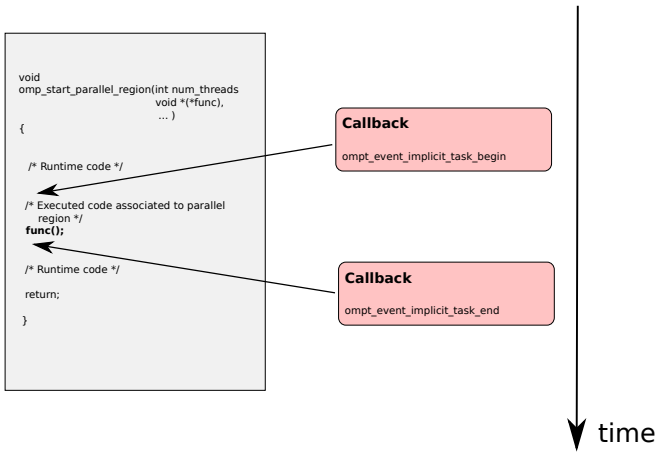


Figure 6.4: Insert event callbacks related to implicit tasks inside OpenMP runtime

We show where the event pair regarding the implicit task has to be inserted with Figure 6.4. To explain the interest of this event pair, we listed the skeleton of an implementation of an OpenMP parallel region inside a runtime. The parallel region is implemented by the function `omp_start_parallel_region`, coming with arguments such as the number of threads, and a pointer to the code inside the parallel region. The callbacks associated to the implicit tasks surround the call to the code `func`.

6.5.2 How OMPT Works

Now we evocated the different features proposed by OpenMP Tools, let's describe how it interacts with the runtime and the performance tools.

Figure 6.5 describes the chain composed of the application running with the OpenMP runtime, the implementation of OMPT, a performance tool, and the interactions between all these components. The runtime implements OpenMP directives of the application and supports its execution. OMPT implementation can be considered as part of the runtime: data structures defined by OMPT have to be implemented inside it and be linked to runtime structures. Hooks (e.g., event callbacks) have to be inserted inside it, in the proper locations. These hooks are used to know when targeted OpenMP constructs are encountered. When a tool wants to be plugged to the OpenMP runtime, it has to communicate through

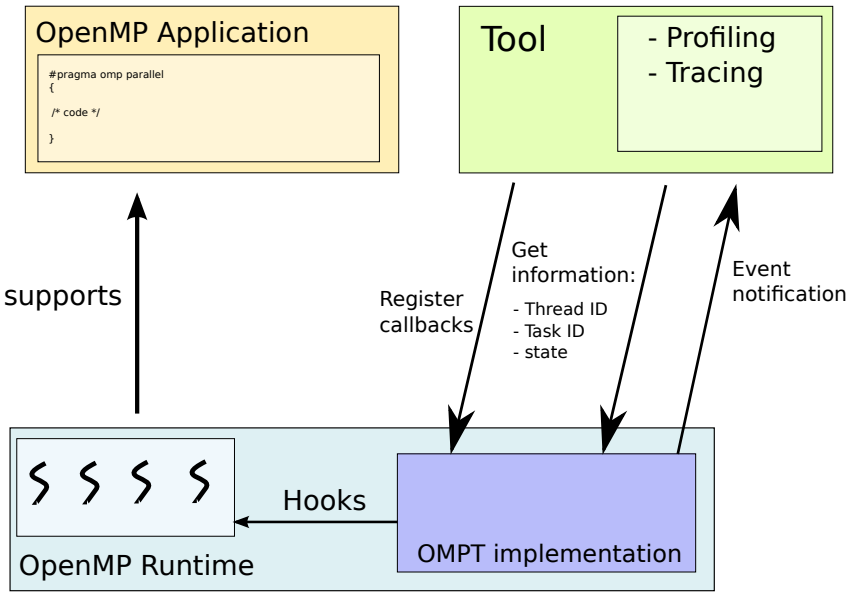


Figure 6.5: Interaction between OMPT, runtime and tool

the OMPT API. So events are then notified to the plugged tool. To receive events from the runtime, the tool has first to register callbacks. To retrieve informations such as thread identifiers, the tool has to call the corresponding inquiry functions. Once callbacks have been registered, the runtime has to notify the events associated to callbacks to the tool.

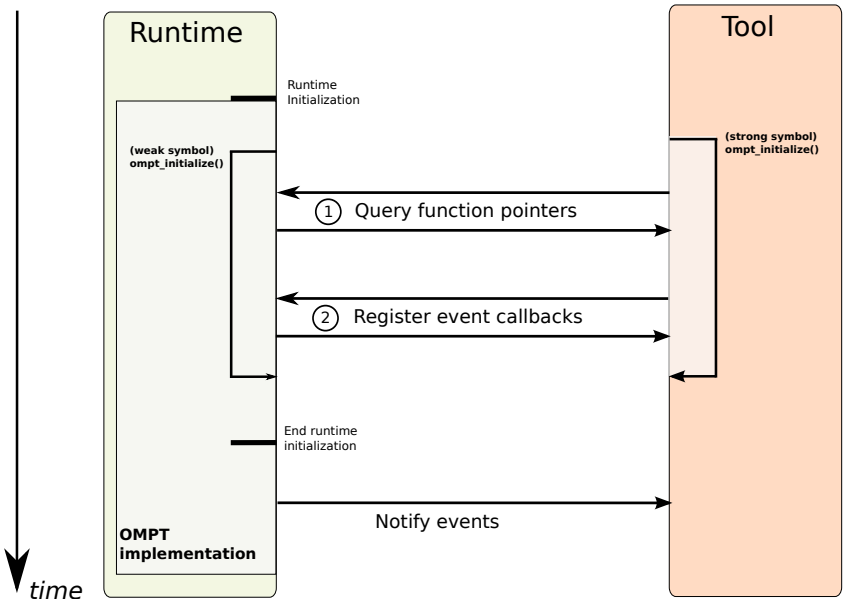


Figure 6.6: Initialization of OMPT

Initialization of OMPT and interactions between runtime and tool are described in Figure 6.6. OMPT is started when runtime is initialized and is turned off at runtime termination. The API defines a function `ompt_intialize` for tool initialization. The prototype of this function is as follows:

```
extern "C"
```

```
{
int ompt_initialize(ompt_function_lookup_t lookup,
const char *runtime_version,
unsigned int ompt_version);
}
```

One of the arguments coming with `ompt_initialize` function is a callback named `lookup` allowing the tool to query pointers. These pointers will be used by the tool to interrogate the runtime.

Both runtime and tool provide an implementation of `ompt_initialize` function, but runtime has to implement it with a weak symbol, whereas the one at tool side must be as strong symbol. This means that the tool-supplied implementation of the function overrides the runtime one. If `ompt_intialize` returns 0, the OpenMP runtime does not need to maintain state information and won't perform any callback.

The tool's implementation of `ompt_initialize` follows two main steps:

- Query function pointers by calling `lookup` function. These pointers are necessary for the tool to register callbacks and get information such as thread identifiers via inquiry functions.
- Register callbacks to receive notifications of events, using `ompt_set_callback` function

For example, to obtain a pointer to get the identifier of current thread, the lookup function has to be used as follows:

```
ompt_interface_fn_t ompt_get_thread_id_ptr = lookup("ompt_get_thread_id")
```

The return code of the function at runtime side is overriden.

Procedure frames. Along its lifetime, any thread switches from the user space to the runtime space: when it encounters a parallel region, the Master thread enters in the runtime, activates the slave threads and executes the body of the region. It then exits the runtime to execute the user's procedure. OMPT distinguishes user's procedures from runtime procedures by dividing the thread execution into procedure frames. For that purpose, OMPT standard defines a structure containing pointers to these procedure frames, associated to the current thread. This structure is updated whether the current thread enters or exits the runtime.

Figure 6.7 illustrates this approach with a code containing nested parallel regions, and executed by 2 threads. Code A, B and C are respectively the initial task (sequential), the code executing the outer parallel region, and the one executing inner parallel region. The user's view of the code execution and the call stacks of threads are compared. When the first thread encounters the outer parallel region, it calls the routine associated with OpenMP parallel region and enters in the runtime to create a new parallel region. A frame f2 is created and corresponds to the runtime routine called by frame f1. The field `reenter_runtime_frame` is set to frame f2 and the `ompt_frame_t` structure is labeled r1.

Before exiting the runtime and starting the implicit task associated to the parallel region "b", the field `exit_runtime_frame` is set to f4.

The second thread starts from frame f3, and is first idle, as shown with the frame f6. When thread 2 is able to execute its work, it enters frame 7. Before exiting the runtime to execute its implicit task associated to code B, field `exit_runtime_frame` of `ompt_frame_t` (labeled r3) is set to frame f7.

At last, when thread 2 encounters parallel region "c", the runtime fills `reenter_runtime_frame` of the structure labeled r3 to frame f9, which is the first frame to be executed after the implicit task associated to code B. Before invoking the implicit task associated to code C, the field `exit_runtime_frame` of structure labeled r4 is set to frame f11.

We can see how pointers of the frame structure are maintainted, in order to distinguish frames dedicated to user's procedures from those for OpenMP runtime routines.

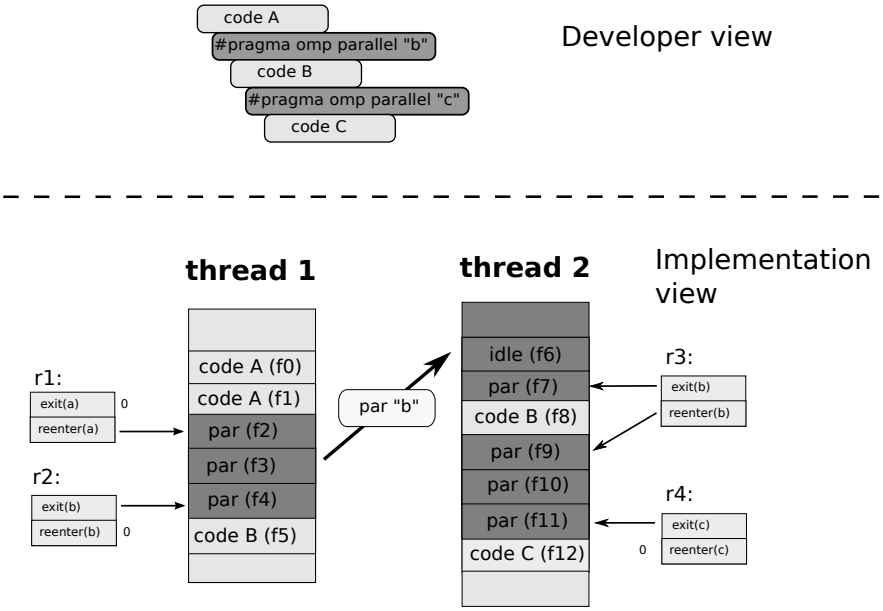


Figure 6.7: Frame management with OMPT

6.5.3 Implementations of OpenMP Tools

OpenMP Tools API is supported by several compilers.
Intel opensourced its OpenMP runtime [5] and a first version of OMPT has been implemented. IMB also supports OMPT in a version of its XL compiler. OMPT is also expected to be supported in the GOMP library of GCC compiler.

6.6 Contribution: Performance analysis of OpenMP applications and OpenMP runtimes using OMPT

The previous section described the OpenMP Tools API, including its possibilities. In this section, we describe our contribution which consists in the implementation of OMPT inside the OpenMP runtime of MPC framework and in the insertion of events. Secondly, we will show how OMPT can be exploited for performance analysis at both application and runtime sides.

```
#pragma omp parallel for schedule(dynamic) private(ip)
  for(ip = 0 + shift ; ip < max + (shift / 2); ip++)
  {
        if (ens_partic->particules[ip].p_enable)
        {
            const int ic = ens_partic->particules[ip].p_nc; // Indice de la couche

            const real_t wmc = ens_partic->particules[ip].p_wmc;

            const real_t dw = (1 - expo) * wmc;

            // Le poids retire' de la particule est depose'
            // dans la couche dans laquelle la particule se deplace

            ens_partic->particules[ip].p_wmc -= dw;

            c_wa[ic]  += dw;
        }

  } /* End for loop */
```

Figure 6.8: Parallelize the loop of the absorption function in MC code

For this purpose, we developed a fine grained version of hybrid application MC [36], by parallelizing loops, and using `pragma omp parallel for` constructs. Figure 6.8 shows a sample code of MC appli-

OpenMP construct	Associated events
ompt_event_thread_begin / ompt_event_thread_end	IMPLEMENTED
ompt_event_idle_begin / ompt_event_idle_end	NOT IMPLEMENTED
ompt_event_parallel_begin / ompt_event_parallel_end	IMPLEMENTED
ompt_event_loop_begin / ompt_event_loop_end	IMPLEMENTED
ompt_event_barrier_begin / ompt_event_barrier_end	IMPLEMENTED
ompt_event_wait_barrier_begin / ompt_event_wait_barrier_end	IMPLEMENTED
ompt_event_single_in_block_begin / ompt_event_single_in_block_end	IMPLEMENTED
ompt_event_single_others_begin / ompt_event_single_others_end	IMPLEMENTED
ompt_event_master_begin / ompt_event_master_end	NOT IMPLEMENTED
ompt_event_task_begin / ompt_event_task_end	IMPLEMENTED
ompt_event_taskwait_begin / ompt_event_taskwait_end	IMPLEMENTED
ompt_event_wait_taskwait_begin / ompt_event_wait_taskwait_end	IMPLEMENTED
ompt_event_taskgroup_begin / ompt_event_taskgroup_end	NOT IMPLEMENTED
ompt_event_wait_taskgroup_begin / ompt_event_wait_taskgroup_end	NOT IMPLEMENTED
ompt_event_sections_begin / ompt_event_sections_end	IMPLEMENTED
ompt_event_implicit_task_begin / ompt_event_implicit_task_end	IMPLEMENTED
ompt_event_initial_task_begin / ompt_event_initial_task_end	NOT IMPLEMENTED
ompt_event_workshare_begin / ompt_event_workshare_end	NOT IMPLEMENTED
ompt_event_tasks_switch	NOT IMPLEMENTED
ompt_event_wait_critical	IMPLEMENTED
ompt_event_acquired_critical	IMPLEMENTED
ompt_event_release_critical	IMPLEMENTED
ompt_event_wait_atomic	NOT IMPLEMENTED
ompt_event_acquired_atomic	NOT IMPLEMENTED
ompt_event_release_atomic	NOT IMPLEMENTED
ompt_event_init_lock	IMPLEMENTED
ompt_event_destroy_lock	IMPLEMENTED
ompt_event_release_nest_lock	NOT IMPLEMENTED
ompt_event_init_nest_lock	NOT IMPLEMENTED
ompt_event_destroy_nest_lock	NOT IMPLEMENTED
ompt_event_acquired_nest_lock_first	NOT IMPLEMENTED
ompt_event_acquired_nest_lock_prev	NOT IMPLEMENTED
ompt_event_release_nest_lock_prev	NOT IMPLEMENTED
ompt_event_release_nest_lock_last	NOT IMPLEMENTED
ompt_event_acquired_nest_lock_next	NOT IMPLEMENTED
ompt_event_flush	NOT IMPLEMENTED

Figure 6.9: Implemented and not implemented OMPT events inside MPC

cation, which is a parallelized loop of the absorption function, one component of the application. The loop is tuned with a dynamic scheduling policy. We then conduct experiments on this version of the MC code, by first showing how we can use OMPT to guide the loop optimization by tuning scheduling policy and chunk size. For these experiments, we compile MC code with both MPC and Intel OpenMP and rely on both implementations of OMPT. All experiments are launched on one 16-core node from the Curie supercomputer, containing 2 CPUs Sandy Bridge EP, clocked at 2.27GHZ.

In the last part, we rely on some OMPT events to estimate the efficiency of the MPC runtime.

This contribution will show how we can use OMPT to detect bottlenecks of the application, and estimate the efficiency of the OpenMP runtimes.

6.6.1 Implementation inside MPC framework

We implemented OMPT standard inside MPC 2.5.0 and tested it with performance-evaluation tool TAU. Most provided events were added inside the runtime, excepted those related to nested parallelism:

To easily enable and disable OMPT support inside the framework, we defined the macro `OMPT_SUPPORT`.

This allowed to eliminate any overhead when not using OMPT.

Our implementation was tested with the performance tool TAU, which proposes proper interface to communicate with OMPT. It is to be noted that entry and exit functions are required in the runtime to insert pairs of events, for constructs such as parallel regions or parallel loops. Thus technical issues appeared when inserting some events related to OpenMP loop and `pragma omp single` constructs. Since MPC runtime is tied to GCC ABI for runtime internals, there is no generated internal for `pragma omp for` construct with static scheduling, whereas there are some with this construct coming with dynamic scheduling. For `pragma omp single` construct, GCC generates one single routine `mpcomp_do_single()` function. This function returns 0 or 1 whether the current thread executes code associated to the construct or not. These issues required to modify the interface of GCC compiler to generate the functions `mpcomp_static_loop_begin()` and `mpcomp_static_loop_end()` for OpenMP loops with static scheduling, and the function `mpcomp_end_single()` to permit the insertion of the terminating event related to `pragma omp single` construct.

6.6.2 Guiding OpenMP loops tuning using OMPT

Our first study with OMPT is purely oriented towards performance analysis of the MC application. As we parallelized existing loops with the combined construct `pragma omp parallel for`, we wanted to know if we would encounter some load imbalance or if we would get the best performances with a static scheduling. In this part, we then conducted an empirical study of OpenMP loops by tuning the scheduling policy and the chunk size. We then launched the hybrid version of the MC code, using 1 MPI task and 16 OpenMP threads, and generated profilings of various combinations of the loops.

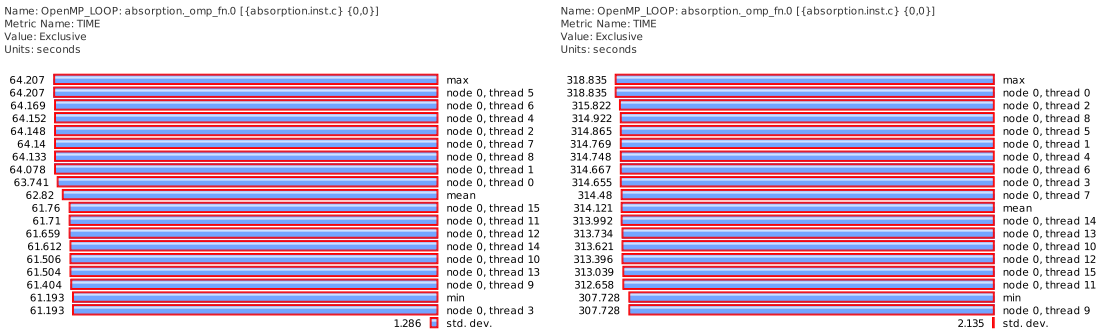


Figure 6.10: MC compiled with MPC - compare performances of static and dynamic scheduled loops

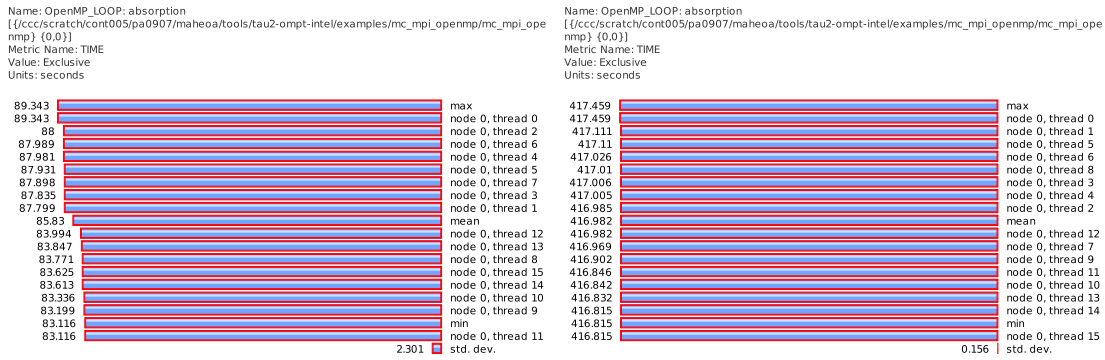


Figure 6.11: MC compiled with Intel OpenMP - compare performances of static and dynamic scheduled loops

With Figures 6.10 and 6.11 we compare the execution times of OpenMP loops with both static and

dynamic scheduling, starting from the absorption function. On each figure, the number of bars equals the number of launched OpenMP threads, multiplied by the number of MPI tasks. There are additional bars regarding the mean and the maximum execution times. We see on the left side of the bars the execution time expressed in seconds. On the right side, we see the identifiers of each OpenMP thread belonging to a MPI task with a given rank number.

On the one hand we notice the version with static scheduling that the loop is well balanced. On the other hand, we observe a huge slowdown when switching to dynamic scheduling: the maximum execution time with static scheduling is around 64 seconds whereas it is 421 seconds with dynamic scheduling, which is approximatively 6.6 $\times$ slower.

There can be multiple reasons explaining such a big gap:

- **Static scheduling vs dynamic scheduling**: In static loops, the iteration space is split between threads, when no chunk size is specified. This policy is then straightforward for the runtime and ensures a minimal overhead. This is thus the best option when the application is well balanced as we can see in this case. Rather, when choosing dynamic policy, each thread gets a chunk, executes it and requests another one until depleting available chunks. We are in the case where no chunk size is defined, so the default is 1, which corresponds to one iteration. This policy involves more work from the runtime, and thus more overhead.
- **Work stealing**: While static scheduling forbids the use of work stealing, such technique is possible with other scheduling policies. But different strategies exist to steal work, which are more or less efficient according to the hardware topology or load imbalance. We saw in Chapter 3 that too aggressive strategies could interfere with the inter-socket bandwidth and disturb compute work, which leads to penalize performances.

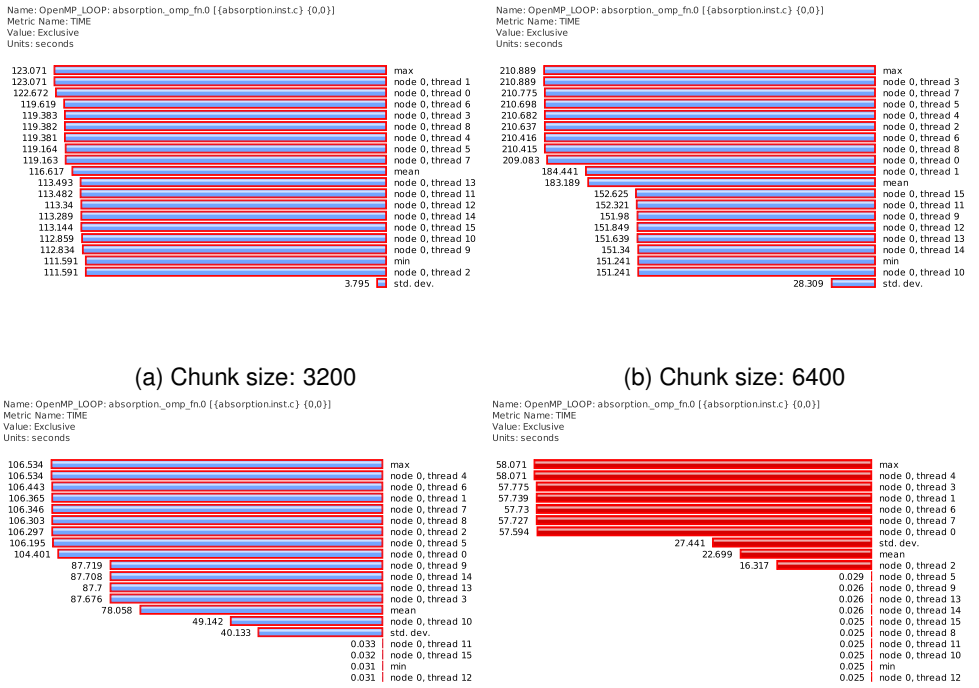
To summarize, several reasons can explain such performance penalties with a dynamic policy, but more insight is required to analyze those results. Unfortunately, OMPT doesn't provide enough coverage to understand the runtime behavior when executing loops.

Figures 6.12 and 6.13 expose the profiling of the parallel loop in the absorption function, scheduled with static policies, running with MPC and Intel OpenMP. We show here how performances evoluate by increasing the chunk size, and how OMPT can help us to find the best optimization. We start from the reference times from Figures 6.10 and 6.11, where maximum times to execute loops were 64 seconds for MPC and 83 seconds for Intel OpenMP, and we try to reduce these times by increasing chunk sizes by a factor 2 at each step. With MPC, we can see that, when running loops with a chunk size 25600, we execute the loops with a maximum time of around 58 seconds, but at a cost of strong imbalance. With Intel OpenMP, a chunk size of 100 gives a maximum time of approximately 53 seconds, with gives a speedup of 60%.

We show with Figures 6.14 and 6.15 the profiling of the same loop, still with MPC and Intel OpenMP, but with a dynamic scheduling. We recall that the maximum times with dynamic scheduling were respectively 318 seconds and 417 seconds with MPC and Intel OpenMP. With MPC, we show different flavors of tuned loops with a chunk size varying from 400 to 3200 iterations. The best result with MPC is obtained with a chunk size of 1600 iterations: maximum time is around 101 seconds, which represents a speedup of 3.16 $\times$. With Intel OpenMP, we reach best performances by tuning chunk size at 800, with a maximum time of approximately 87 seconds (speedup of 4.79 $\times$). Chunk size is increased following the different sub-figures, from 400 to 3200. We can see that minimal execution time is reached with a chunk size of 1600: the maximum execution time reaches approximately 100 seconds, whereas it was 421 seconds without specifying a chunk size (e.g, chunk size equal to 1). We then gained a speedup of 4.21.

We see with these examples that OMPT, coupled with TAU, can help us to tune parallel loops in order to obtain the best performances, through successive optimizations. Our observations are the following:

- Dynamic scheduling is far from being the best scheduling policy against static one and can give surprising slow downs.



(a) Chunk size: 3200

(b) Chunk size: 6400

(c) Chunk size: 12800

(d) Chunk size: 25600

Figure 6.12: Tune loops of the absorption function depending on the chunk size, with static scheduling, running with MPC - with the help to OMPT and TAU

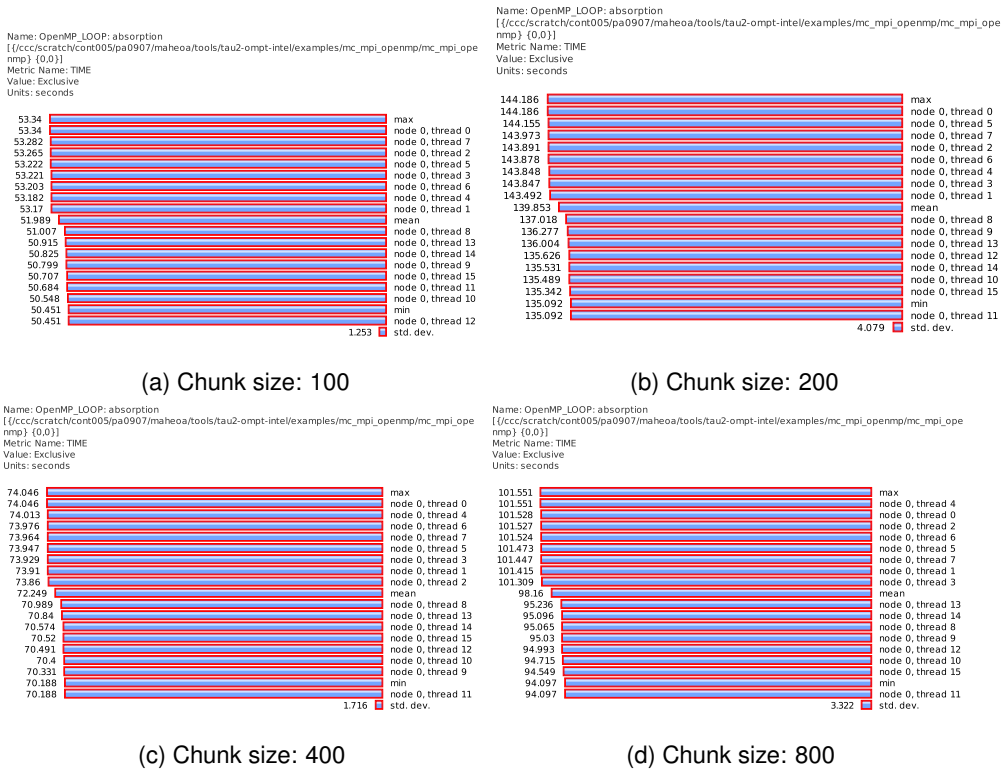
- An increasing chunk size can give huge speedups, especially when coupled with dynamic scheduling. We explain this observation by the fact that when the chunk size is increased, the pool of chunks to be executed is reduced, then each thread has less work to do. Consequently, the overhead implied in the runtime is lowered.

6.6.3 Estimate overhead of OpenMP runtimes

Whereas we could characterize an OpenMP code and optimize its loops using OMPT and profiling tool, we will demonstrate here that we can get a better use of this interface and estimate the overhead of the runtime. Indeed, the event pair `ompt_event_implicit_task_begin` and `ompt_event_implicit_task_end` allows to separate the body of parallel regions from threads activation and synchronizations.

We can see in Figure 6.16 the inclusive execution time of the different components of the MC application (on the first sub-figure). Orange bars represent the time spent between the event pair relative to the OpenMP parallel regions. Green bars next to the orange ones correspond to the time spent in implicit tasks. The overhead of the runtime can be estimated in several ways: either by the exclusive time of the OpenMP parallel region, or by substracting the execution time of the parallel region from the one of the implicit task. In the last sub-figure we can see that the exclusive time associated to the parallel region is 87 milliseconds, which corresponds to the interval between the moments when the master thread executes the runtime procedure associated to the region construct and when it starts executing associated an implicit task. This interval corresponds to the overhead of the runtime, and has to be compared with the duration of the parallel region (second sub-figure). We can then deduce that this overhead represents 0.013% of the execution of the parallel region.

The runtime overhead could be estimated with MPC only, since no event related to implicit tasks



(a) Chunk size: 100

(b) Chunk size: 200

(c) Chunk size: 400

(d) Chunk size: 800

Figure 6.13: Tune loops of the absorption function depending on the chunk size, with static scheduling, running with Intel OpenMP - with the help to OMPT and TAU

were inserted into the Intel OpenMP runtime.

6.7 Conclusion

In this chapter, we started by depicting the various bottlenecks to be encountered in hybrid codes. The origins of these bottlenecks are from both application and runtime sides, and are therefore difficult to identify. Thus, sophisticated tools are necessary to troubleshoot MPI+OpenMP codes. We then presented an overview of the field of performance analysis and focused on the instrumentation layer. Whereas MPI standard includes primitives for performance measurement, this was not the case for OpenMP until recently. OpenMP Tools API was shortly integrated to OpenMP standard and we then described this instrumentation tool. In our contribution, we implemented OMPT in MPC framework and were then able to instrument OpenMP and hybrid codes, and measure them thanks to TAU performance tool. OMPT helped us driving loop optimizations, by tuning scheduling policy and chunk size.

From our point of view, OMPT can interest both application developers and runtime designers, as it allows isolating bottlenecks from both origins. We saw application and runtime are strongly interlaced, especially in nested parallel codes: switches from code to runtime are frequent. OMPT tackles this aspect with several features. For example, the pair of events related to the implicit tasks allows separating the work spent in activating OpenMP threads and the work dedicated to computations. These events so proved to be very useful to estimate the runtime overhead. Moreover, by proposing procedure frames, user's procedures and runtime procedures are separated.

But procedure frames don't give an exact view of interactions between application and runtime. Indeed, this structure assumes threads executing their implicit tasks (i.e. body of parallel region), don't

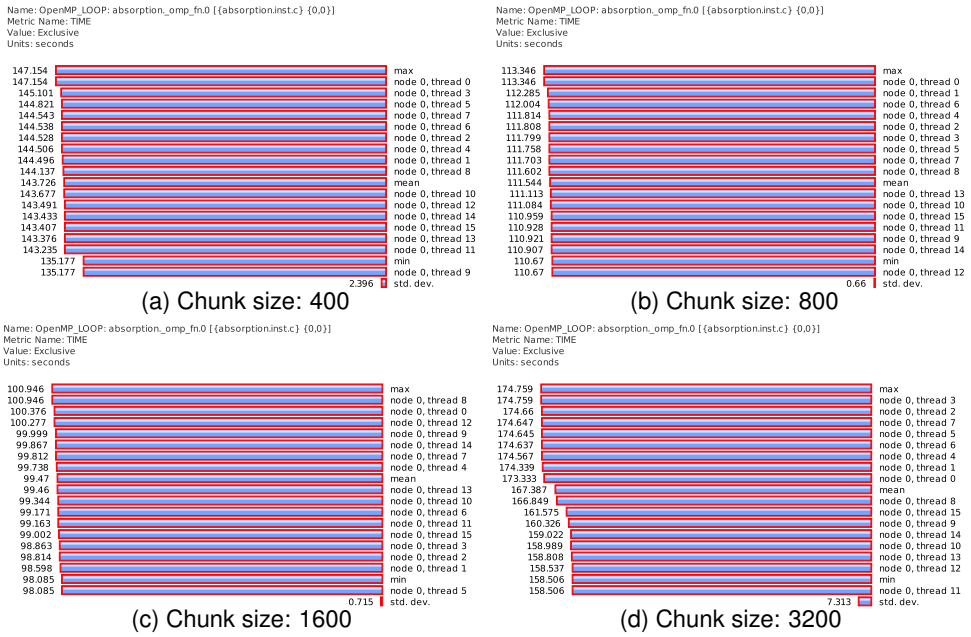


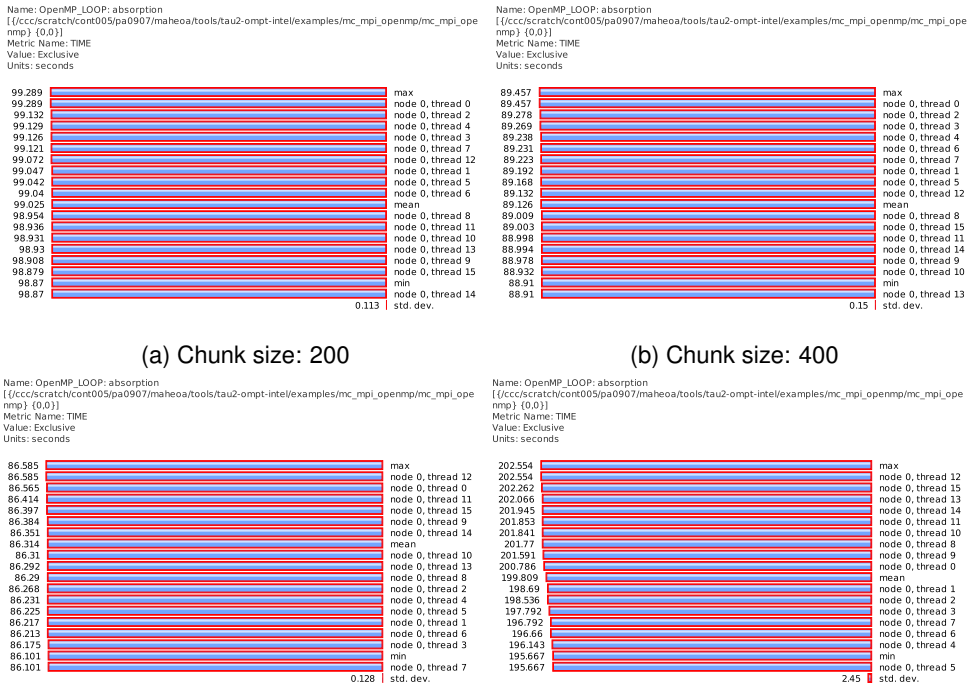
Figure 6.14: Tune loops of the absorption function depending on chunk size, with dynamic scheduling, running with MPC - with the help to OMPT and TAU

reenter the runtime during this time. Actually, parallel regions often include loop or synchronization parts such as `pragma omp for`, `pragma omp barrier` or `pragma omp single`. Executing these constructs implies reentering inside the runtime, which can be a source of additional overheads.

The work of threads synchronization is also included in the runtime overhead. The event pair `ompt_event_barrier_begin` and `ompt_event_barrier_end` can be used to measure the cost of an OpenMP barrier. But the load imbalance accounts for a large part to this cost, and some threads have to wait for the last one to reach the barrier. Thus, we can separate a synchronization construct into two components for each thread, regarding MPC implementation:

- The barrier overhead: to climb the tree until reaching the root, and go back to the leaves before continuing execution
- The barrier idling: make the current thread wait until all threads reach the barrier

Fortunately, OMPT also includes events measuring how much time each thread is idle inside the barrier (`ompt_event_wait_barrier_begin` / `ompt_event_wait_barrier_end`). We can then isolate overhead induced by the work of synchronization, by substracting the sum of waiting times from the time spent inside the barrier.



(a) Chunk size: 200

(b) Chunk size: 400

(c) Chunk size: 800

(d) Chunk size: 1600

Figure 6.15: Tune loops of the absorption function depending on chunk size, with dynamic scheduling, running with Intel OpenMP - with the help to OMPT and TAU

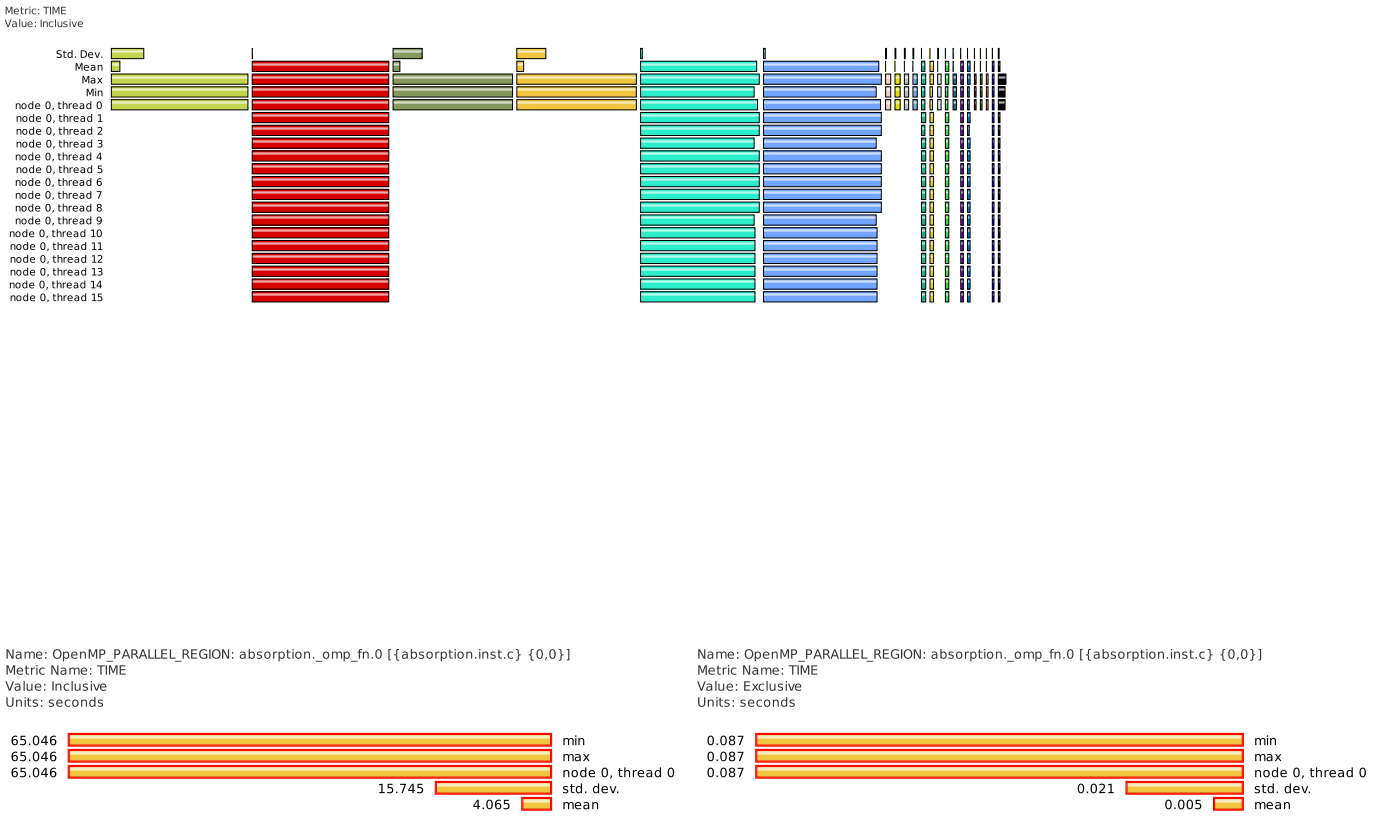


Figure 6.16: Deduce overhead of OpenMP runtime based on implicit task

Chapter 7

Conclusion and Perspectives

In this thesis, we addressed some problematics coming with supercomputers in the field of High Performance Computing, especially its programming aspects. As parallel architectures evolved with the introduction of multicore processors and the advent of heterogeneous environments, it becomes clear that the MPI model is no longer the unique solution to enable code scalability. Different solutions exist, and one consists in combining MPI with a thread based model such as OpenMP. This direction is more and more adopted in parallel codes.
But switching to the Hybrid programming is complex and generates several bottlenecks following the ways MPI and OpenMP are combined. Thus, switching from pure MPI model to Hybrid programming implies making the whole software stack to evolve (applications, runtimes). We recall here our contributions presented in this thesis, and the perspectives, e.g. some directions to deeper dig inside Hybrid programming.

7.1 Contributions

We recall here our contributions, tackling different aspects of Hybrid programming.

Overhead of OpenMP in Hybrid programming

Mixing several programming models generally implies cooperation between different runtimes. This approach can be interesting only if each model obtains a good performance. But OpenMP, due to its execution model, generates an overhead when creating and stopping threads. This overhead can impact performances of MPI+OpenMP codes, especially in Fine-Grain mode (i.e, when OpenMP is used to parallelize loops). To maintain an OpenMP runtime as efficient as possible, the characteristics of NUMA architectures have to be taken into account. This is why the OpenMP runtime of MPC framework evolved to integrate a hierarchical activation and synchronization of OpenMP threads. But with hybrid codes, computational resources are shared between MPI tasks and OpenMP threads, and following the thread placement, the space dedicated to a single OpenMP team can be limited. This is why our first contribution, the *Adaptive tree*, aimed at minimizing the overhead of the runtime as much as possible, by offering a flexible way to launch and stop threads. This contribution only requires few modifications of some OpenMP constructs inside the MPC framework. This technique was tested with microbenchmarks against mainstream compilers such as GCC and ICC and showed interesting results on a 128-core NUMA node.

Cooperation between MPI and OpenMP for Collective operations at application and runtime levels

A MPI based application can be decomposed into sequential and parallel parts. Computational resources are then fully exploited only when the parallel component is executed. Combining MPI with OpenMP consists in adding a level of parallelism, but as compute cores are shared between tasks and threads, only a fraction of cores is used when executing only MPI, leading to waste resources.

This is especially true with the *Masteronly* approach, where cores dedicated to threads are idle outside OpenMP parallel regions. Our second contribution aimed at tackling this drawback by targeting MPI Collective operations. We then used OpenMP threads to keep cores busy as much as possible and accelerate Collective operations. We enhanced this new algorithm with the design of the *Rank Shifting* technique. By adapting the way each OpenMP thread processes a sub-part of the MPI collective, the memory and network bandwidth can be exploited to improve performance. This approach was validated on several runtimes such as MPC or IntelMPI and allowed improvement of the overall time up to a factor of 5.29 $\times$ on a real world application.

In a second part, we discussed some ways to build constructs gathering MPI and OpenMP. We focused on an OpenMP used in a SMPD fashion and highlighted similarities with the MPI model. We then proposed new collective operations implying both MPI tasks and OpenMP threads, implemented inside the MPC runtime, and introducing a unified barrier as a case study.

Performance analysis of OpenMP codes and OpenMP runtime using an instrumentation tool

Obtaining scalability of parallel applications on thousands of processors is a complex work, and bottlenecks can be difficult to detect, as they can come from the application level or the runtime level. Thus, numerous tools have been developed to help the users to solve performance issues. But efficient instrumentation is required to analyze performances of both MPI and OpenMP models. OMPT was introduced as an extension of the OpenMP standard, proposing features to instrument OpenMP codes. In our last contribution, we implemented *OMPT* inside MPC framework. We then relied on our implementation to profile hybrid codes, and got insight of the behavior of our runtime.

7.2 Perspectives

We discuss in this section the main architectural trends which will be encountered in future compute nodes, and their impacts in MPI+OpenMP codes. Then, we will give concrete examples of short term evolutions, based on our contributions, and long term perspectives.

7.2.1 Towards heterogeneous nodes

Based on architectural trends, compute nodes should integrate more and more hardware accelerators, such as Intel Xeon Phi or GPUs. Current architectures presented in Section 2.2 of Chapter2, and composed of NUMA nodes, will then be extended to include these devices.

Figure 7.1 shows how traditional CPUs and hardware accelerators are articulated. We define a NUIOA[1] node as composed of a NUMA node linked to an hardware Accelerator via a bus (PCI Express bus in this example). This is the building block of heterogeneous nodes which will spread in future supercomputers. Each Hardware accelerator is equipped with its own memory; when the user wants to execute a program on the device, the data to be computed have to be transferred to its memory.

Parallel programming models thus evolved to exploit these devices. Specific parallel models appeared to program GPUs such as CUDA[2][29] introduced by Nvidia, or OpenCL[84]. But porting large scientific codes using these models is costly. Thus, more recent parallel models such as OpenACC[107] or OpenMP 4.0[70] were released to provide simpler interfaces based on directives offloading computations on hardware accelerators. This simplification leads to allow an easier adoption by legacy codes. Some efforts have been made to translate OpenMP into CUDA, with OpenMPC for example [67]. But from our point of view, the next step would be to port MPI and OpenMP codes on GPUs. But some runtimes such as StarPU propose an execution model relying on tasks and allowing execution of code and data transfers on GPUs in a transparent way [10, 106].

[1] Non Uniform Input Output Access
[2] Compute Unified Device Architecture

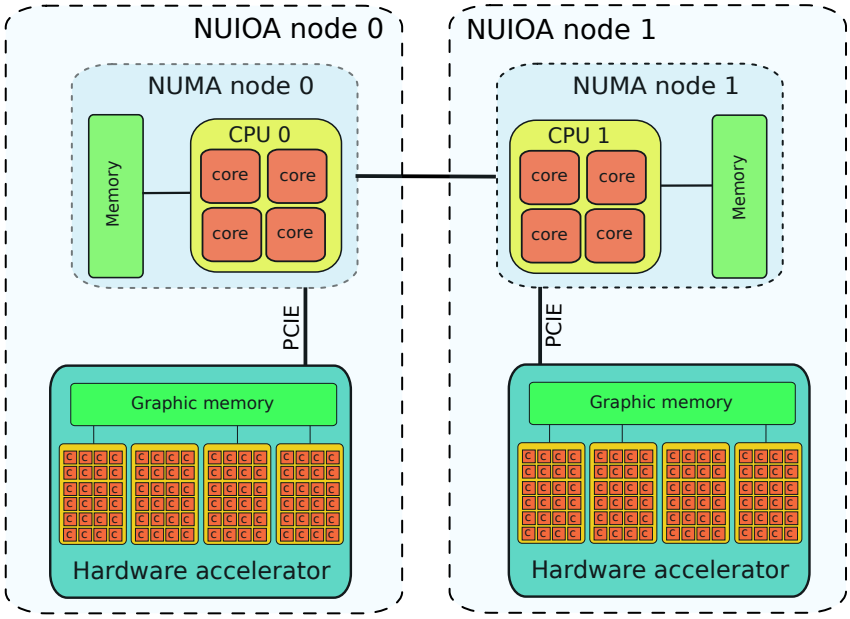


Figure 7.1: Non Uniform Input Output Access architecture gathering a NUMA node and a hardware accelerator, communicating through a PCI Express bus

Xeon Phi coprocessors support native programming models such as MPI or OpenMP. Therefore, parallel codes do not have to be completely rewritten to run on such accelerators. However, as current coprocessors can run several hundreds of threads, both MPI and OpenMP libraries should be adapted to these architectures.

To summarize, future machines should mix distributed memory, shared memory, and heterogeneous systems. They should be more complex to program, and will require efforts from the research community to be efficiently exploited. Therefore, the Hybrid model should evolve following at last two aspects:

- **MPI and OpenMP for manycore**: Optimize both MPI and OpenMP libraries to get good performances on manycore architectures
- **Heterogeneous taxonomy**: The taxonomy was defined for MPI+OpenMP codes running on traditional clusters. it should evolve, taking the coprocessors into account inside the Thread placement (1 MPI task per compute node, 1 task per NUOIA node, several tasks per accelerator,...).

7.2.2 Short term evolutions

Going back to our contributions, we propose here several ideas to enhance them. We will also focus on how to port our contributions to manycore architectures.

Overhead of OpenMP runtimes on manycore environment

We optimized the OpenMP runtime of the MPC framework to efficiently start and stop threads on large compute nodes, containing up to 128 cores. These nodes were designed following a topology implying several levels of NUMA nodes, exposing hierarchies of parallelism. Compared to the large NUMA nodes, the architecture of Xeon Phi coprocessor contains less hierarchical levels. Moreover, to be able to run on such an architecture, OpenMP runtimes will have to deal with 250 to 300 threads, which can generate a huge overhead. Thus, porting OpenMP codes on a manycore environment such as Intel Xeon Phi implies optimizing the runtime for this platorm. The hardware topology of this device should then be studied.

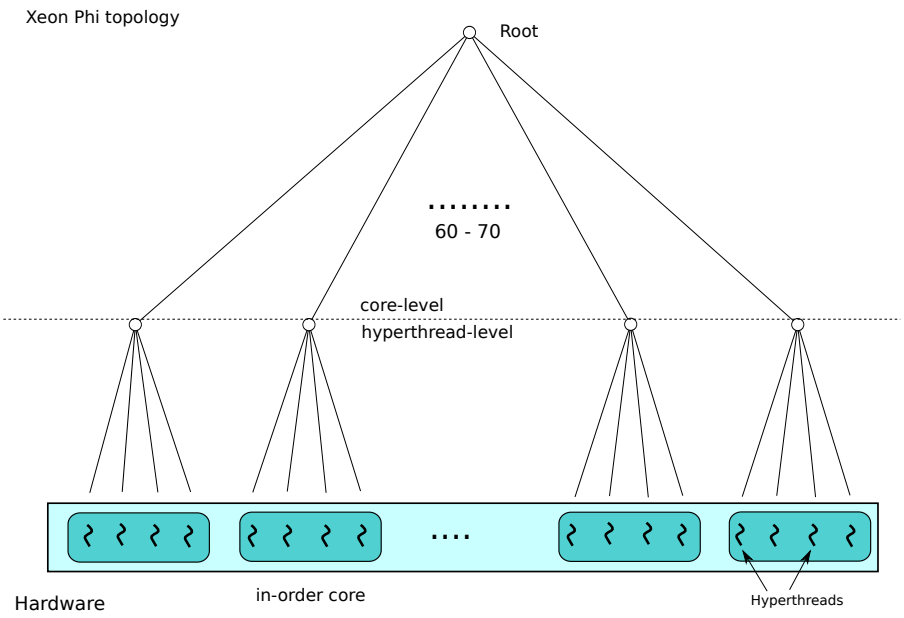


Figure 7.2: Topology of Xeon Phi architecture

Figure 7.2 exposes the topology view of a Xeon Phi processor. As we can see, two hierarchical levels exist:

- **core level**: Depending on the version of the processor, we find between 60 and 70 in-order cores uniformly distributed inside the socket. These cores are interconnected by a bidirectional ring bus.
- **hyperthread level**: Each core supports four hardware threads.

If we map the underlying topology (Figure 7.2), we should use a two-level tree to deal with OpenMP threads: the root would have 70 children and with 4 children for each sub-tree, since each core can contain four hyperthreads. Moreover, building the tree would be straightforward as HWLOC can detect the topology of Intel Xeon Phi. But in the case the coprocessor is fully populated, the Master thread would have to deal with more than 70 threads, which should generate a huge contention.

Thus, we should decorrelate the tree from the topology and add additional levels, in order to reduce the work of the Master thread. Figure 7.3 illustrates a tree allowing to decouple the thread activation. There are two levels at the core-level, then the Master thread only has to activate six children, and each child wakes the threads. We could even insert an additional level to keep the degrees of the sub-trees as minimal as possible. We assume this kind of tree would give better performances than just using a tree respecting the underlying topology.

Optimization of MPI Collective operations using threads

As we only focused on `MPI_Allreduce` operation, this contribution could be extended towards different directions:

- **Generalize to other MPI collectives**: Our approach is not specific to `MPI_Allreduce` collective and could be experimented with collectives such as `MPI_Reduce` and be generalized to other ones such as `MPI_Bcast` or `MPI_Gather`. We remind that *Rank Shifting* doesn't work with rooted collectives (Reduce, Broadcast, etc).
- **Performance model**: For now, we validated our concept with an empirical approach. We plan to predict the best hybrid combination using heuristics and to establish a performance model, based on criteria such as vector length and hardware topology. This model would allow to launch the application with an automatically calculated number of threads.

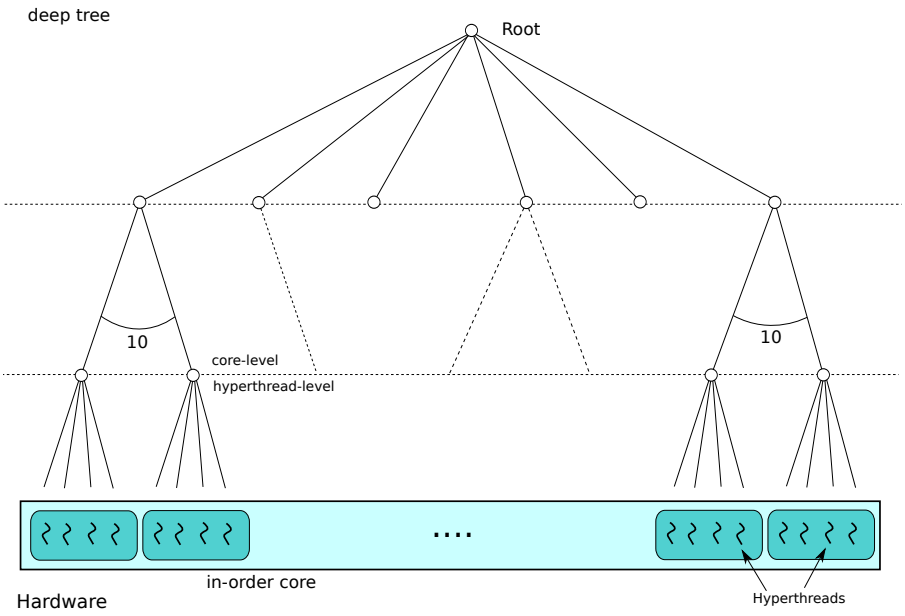


Figure 7.3: Increase depth of the tree for Xeon Phi

- **Rank Shifting**: For the moment, *Rank Shifting* is implemented to shift MPI rank with a step equal to one, ignoring underlying topology and placement of MPI tasks. Indeed, the implemented policy can fit if there is only one MPI task per socket or per node. But with several tasks per socket, step should be increased. Thus, this technique should be more deeply investigated by testing different policies for Rank Shifting, like increasing step for example, or shifting from one NUMA node to another, in order to take memory balancing into account and avoid NUMA effects as much as possible.
- **Oversplitting**: At this stage, initial vector of each MPI task is split between the OpenMP threads. Another optimization can be studied, i.e. splitting vectors in chunks smaller than required so that each thread performs several calls instead of only one. This solution could lead to better cache reuse. Moreover, since each thread would have more than one chunk to compute, scheduling policies other than static could be experimented.
- **Enlarge cache**: In order to avoid duplicating communicators each time `MPI_Allreduce` is called and generate additional overhead, we implemented a cache of size 1 storing the input communicator. But if the input communicator changes, the old one is trashed and a new communicator will be duplicated. We would like to enlarge the cache to be able to store multiple communicators, for more complex applications.
- **Parallelize collective inside runtime**: For now, the developer has to preload a wrapper in order to re use idle threads. To enable full transparency for the developer, hybridization could be achieved inside the runtime, by tuning implementation of MPI Allreduce. Two limitations have to be mentioned: it requires a common vision of MPI and OpenMP inside the runtime, and it makes us dependent from the runtime.
- **Experiment on Xeon Phi**: At this stage, we tested our approach on NUMA environments and proved we could have interesting results in this context. It would be interesting to test hybridization of Allreduce on Xeon Phi co-processor [1] and see how our approach scales on numerous cores and how we can take advantage of existing shared memory.

Unified Collective operations

We introduced Unified collective operations to bridge MPI and OpenMP used in SPMD fashion. For future work, we target several aspects of Coarse grain approach:

- **Unified barrier**: When all OpenMP threads have passed their barrier, the threads reaching the root of their respective trees call `MPI_Barrier()`. All threads are then idle until MPI barrier returns and makes them free. Several possibilities can be studied about keeping them busy during this time (execute loop iterations or OpenMP tasks for example).
- **Unified reduction**: MPI includes several primitives performing reduction such as `MPI_Reduce()` or `MPI_Allreduce()`. OpenMP allows a similar feature by the clause `reduction` coming with `#pragma omp for` construct. This calls for a Unified reduction gathering MPI ranks and OpenMP threads.
- **Hybrid parallel loop**: `#pragma omp for` allows loop level parallelism by distributing the chunks among the threads. But it is limited to OpenMP teams and a single compute node. We could extend this construct by distributing iterations to other MPI tasks, by using Scatter collective for example.
- **Other Collectives**: We observed with [65] that collective operations could be realized with OpenMP. We can then extend our concept to other operations such as All-to-all, Broadcast, etc.
- **Critical constructs**: As seen in chapter 2, in Coarse Grain parallelism synchronization constructs are required to surround MPI primitives called inside OpenMP regions, for correctness purposes. Developer is still in charge to use proper synchronization directives to ensure MPI primitives are called only once. It seems straightforward to tune such MPI primitives at runtime level in that they are called by only one thread.
- **Multi-threaded Collectives**: Splitting off collectives is a technique allowing communications overlap, but is heavy to implement and often leads to bugs. Another variant of enhanced collective would be to parallelize communications at runtime level.
- **Initialization**: By studying hybrid taxonomy, we saw that in Coarse grained codes, one unique OpenMP parallel region was open, initializing OpenMP threads and MPI tasks approximately at the same time. We could improve simplicity of the code by proposing one unique construct initializing and terminating both MPI tasks and OpenMP threads. `MPI_Init()` function is required to launch MPI tasks. A common construct initializing both the MPI tasks and the threads could be interesting for Coarse Grain mode, where MPI and OpenMP are both enabled at the beginning of the execution.

Performance analysis with OMPT

We pinpointed some limitations in OMPT standard that prevents from a complete analysis of OpenMP runtimes. So, for future work we plan to dig more into loop optimizations, and better identify roles of both application and runtime in performance bottlenecks. To drive loop tuning, one direction would consist in building a decision tree to automatically deduce the best optimization of OpenMP loops, by tuning both scheduling policy and chunk size.
As we noticed a huge slowdown when switching from static to dynamic scheduling, we would like to get more insight into the runtime, and isolate work stealing from execution of iterations. Causes of this slowdown could be inefficient work stealing strategies or bad implementation of OpenMP loops. But proposed events by OMPT prevent us from investigating inside the runtime, and from identifying sources of slow down. For that purpose, we propose the creation of a new set of events dedicated to work stealing, and allowing to estimate the ratio of time spent in work stealing when executing loops on the one hand, and the percentage of success in the work stealing (how often a thread can steal one or several chunks) on the other hand. We could also imagine finer grained ways to instrument OpenMP loops, such as creating events for each chunk. But all these possibilities have to be carefully considered, as they could generate too much measurement overhead.

7.2.3 Long term perspectives

Based on first observations on future heterogeneous environment and short term evolutions, we propose here some long term perspectives relying on two main ideas:

- **Runtime stacking**: We propose to adapt our contribution about Unified collectives to several models.
- **Hybrid scheduling in an heterogeneous environment**: Challenges exposed by heterogeneous clusters will highlight new bottlenecks for the Hybrid codes. Also, features such as asynchronous communications or transfers from CPUs to hardware accelerators will require better more sophisticated schedulers.

Generalization of the unified barrier with nested parallel models

We described the operation consisting in performing a unified barrier involving MPI tasks and OpenMP threads. MPI and OpenMP are nested: one MPI task creates a team of *t* OpenMP threads. So each OpenMP team performs the first part of a barrier, then the MPI tasks are synchronized, and then each team does another half barrier.

This example involved two programming models, but more models could be involved in this operation (MPI, OpenMP, and POSIX threads for example). Now let's extend this operation and consider *n* nested programming models:

$$M_0 \rightarrow M_1 \rightarrow ...M_{n-1}$$

For example, M_0 is the MPI model, and M_1 is OpenMP.

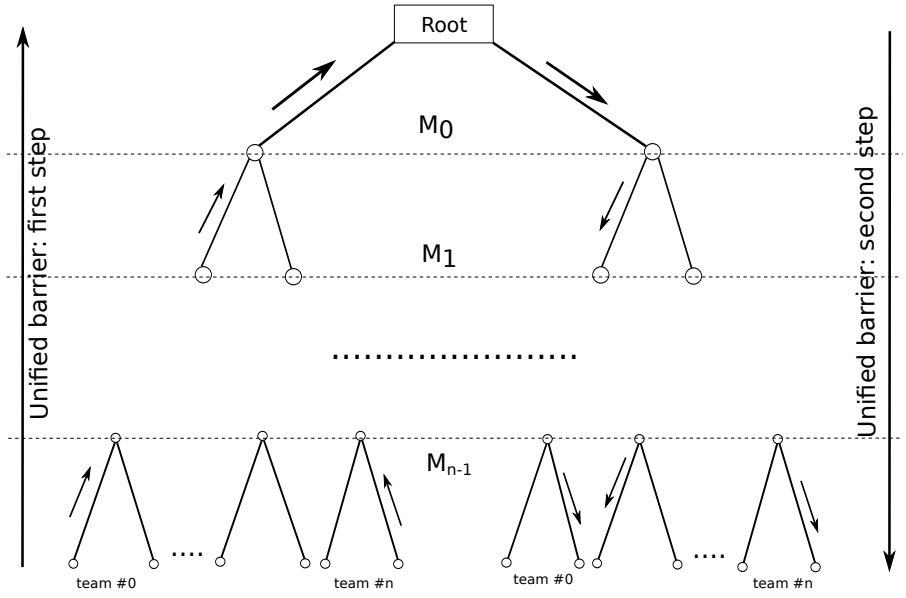


Figure 7.4: Generalization of the unified barrier with n models

Figure 7.4 shows how the unified barrier would be executed, if involving n models: The model M_{n-1} first performs a half barrier, and the one that called M_{n-1} executes its half barrier, and we continue until the model M_0 does a complete barrier. Then all nested models repeat another half barrier but in the other direction, until reaching M_{n-1}. This construct implies that M_i is able to enable M_{i+1}.

We need to mention some constraints to enable unified constructs implying n models:

- Each model excepted M_0 needs to be able to decompose a barrier into two steps: (i) all tasks reach the scheduling point and (ii) they can continue their execution once the scheduling point is reached.

- M_i has to be able to efficiently activate M_{i+1}. This implies using lightweight mechanisms to activate the inner model.
- **Topology**: The inner model M_{i+1} has to be aware of its restricted topology given by M_i. This implies that either all models share a global view of the hardware topology, or each model M_i knows its restricted topology.
- **Affinities**: Data locality is not taken into account with the MPI model (M_0). So the ranks are usually dispatched among compute cores inside a node. But with the thread-based models, the threads are kept close to each other, following their affinities. But in the case such model activates another one, it has to let some cores idle for the inner model. So policies have to be defined to guarantee that the nester and the nestee models can both keep their threads close to each other.

Scheduling MPI tasks and OpenMP threads.

With the Hybrid MPI+OpenMP codes, execution time is shared between computations and communications, and cores are shared between the tasks and the threads according to the thread placement. Along this thesis, we took a static approach about the utilization of compute cores by tasks and considered that each MPI task or thread was statically bound to a compute core. This approach is available when MPI performs blocking communications.

But MPI standard also introduced Non Blocking Communications, whether they are point-to-point (`MPI_ISend`, `MPI_IRecv`) or collective (`MPI_IBarrier`, `MPI_IReduce`). These asynchronous communications allow performing operations before communications have finished. Thus, a compute core used by a task taking part to such communications can be reused by another MPI task or another thread. These features offer the possibility to share the cores between several tasks and introduce the necessity to schedule them. To enable a fair utilization of available cores, a smart scheduling of MPI tasks and OpenMP threads is required. This leads to open questions: should we favor computations or communications ? One technique consists in creating additional threads to accelerate progression of MPI communications. But a special care has to be given to the placement of this kind of threads, since they should not disturb computations.

But asynchronous MPI communications are not the only example motivating this technique. In heterogeneous context, programming models such as OpenMP 4.0 enable transfers between CPUs and hardware accelerators. Another kind of threads would be in charge to handle these transfers.

Thus, the scheduler could have different kinds of threads to handle, and some options are available about the placement of these specialized threads:

- Gather them on a dedicated core or dispatch them inside the compute node ?
- Gather the progression threads in blocks according to their speciality or build blocks mixing threads of different types
- Schedule in priority specialized threads
- Dedicate a number of cores to these threads: 1 core, or several

It is to be noted that the MPC framework includes a scheduler providing a global vision of MPI tasks and OpenMP threads. So this could be an eligible platform to conduct these works.

Appendix A

Hierarchical barrier inside MPC

```
/* Barrier for all threads of the same team */
void
__mpcomp_internal_full_barrier (mpcomp_mvp_t *mvp)
{
        mpcomp_node_t *c;
        long b;
        long b_done;
        mpcomp_node_t * new_root ;

#if 0
                        /* TODO DEBUGGING PURPOSE */
                        if ( mvp->threads[0].for_dyn_shift != NULL &&
                                        mvp->threads[0].for_dyn_shift[1] != 0 ) {
                                fprintf( stderr, "[0] __mpcomp_internal_full_barrier: shift[1] = %d\n",
                                         mvp->threads[0].for_dyn_shift[1]        ) ;
                                // not_reachable() ;
                        }
#endif



        c = mvp->father;
        sctk_assert( c != NULL ) ;

        new_root = c->instance->team->info.new_root ;
        sctk_assert( new_root != NULL ) ;

        /* TODO: check if we need sctk_atomics_write_barrier() */

        /* Step 0: TODO finish the barrier whithin the current micro VP */

        /* Step 1: Climb in the tree */
        b_done = c->barrier_done; /* Move out of sync region? */
        b = sctk_atomics_fetch_and_incr_int(&(c->barrier));

        while ((b+1) == c->barrier_num_threads && c != new_root ) {
                sctk_atomics_store_int(&(c->barrier), 0);
                c = c->father;
                b = sctk_atomics_fetch_and_incr_int(&(c->barrier));
        }

        /* Step 2 - Wait for the barrier to be done */
        if (c != new_root || (c == new_root && (b+1) != c->barrier_num_threads)) {
                /* Wait for c->barrier == c->barrier_num_threads */
                while (b_done == c->barrier_done) {
                        sctk_thread_yield();
#if MPCOMP_TASK
                        __mpcomp_task_schedule(); /* Look for tasks remaining */
#endif /* MPCOMP_TASK */
                }
        } else {

#if MPCOMP_TASK
                __mpcomp_task_schedule(); /* Look for tasks remaining */
#endif /* MPCOMP_TASK */
                sctk_atomics_store_int(&(c->barrier), 0);

#if MPCOMP_COHERENCY_CHECKING
                __mpcomp_for_dyn_coherency_end_barrier() ;
                __mpcomp_single_coherency_end_barrier() ;
#endif

                c->barrier_done++ ; /* No need to lock I think... */
        }

        /* Step 3 - Go down */
        while ( c->child_type != MPCOMP_CHILDREN_LEAF ) {
                c = c->children.node[mvp->tree_rank[c->depth]];
                c->barrier_done++; /* No need to lock I think... */
        }
}
```

Figure A.1: Implementation of the hierarchical barrier inside MPC framework

Appendix B

Implementation of OpenMP Tools API inside MPC

```
void
__mpcomp_start_parallel_region(int arg_num_threads, void *(*func)
    (void *), void *shared)
{
  mpcomp_thread_t * t ;
        mpcomp_new_parallel_region_info_t info ;

  sctk_nodebug(
      "__mpcomp_start_parallel_region: === ENTER PARALLEL REGION ===" ) ;

  /* Initialize OpenMP environment */
  __mpcomp_init() ;

  /* Grab the thread info */
  t = (mpcomp_thread_t *) sctk_openmp_thread_tls ;
  sctk_assert( t != NULL ) ;

#if OMPT_SUPPORT
  ompt_frame_t frame;
  ompt_frame_t *parent_task_frame;
  ompt_task_id_t parent_task_id;
  uint32_t team_size;

  /* Generate parallel id if not yet generated */
   parallel_id = OMPT_gen_parallel_id();
   t->children_instance->team->ompt_parallel_info.parallel_id = parallel_id;
   t->children_instance->team->ompt_parallel_info.parallel_function = func;
   t->instance->team->ompt_parallel_info.requested_team_size = arg_num_threads;

  /* Check parallel region depth */
  if(t->instance->team->depth == 0) { /* not nested parallel region : so we generate a parent task team (illicit) */
    frame.reenter_runtime_frame = __builtin_frame_address(0);
    parent_task_frame = &frame;
  } else { /* Nested parallel region */
    t->mpc_ompt_thread.parent_task_frame.reenter_runtime_frame = __builtin_frame_address(0);
    parent_task_frame = &(t->mpc_ompt_thread.parent_task_frame);
  }

  parent_task_id = OMPT_gen_task_id(); /* No nested parallel region handled, so generate parent_task_id */
  cb = OMPT_Callbacks[ompt_event_parallel_begin];

  if(cb && OMPT_enabled())
   ((ompt_new_parallel_callback_t)cb)(parent_task_id, parent_task_frame, parallel_id, t->instance->team->ompt_parallel_info.requested_team_size, (void
         *)func);
#endif /* OMPT_SUPPORT */


        __mpcomp_new_parallel_region_info_init( &info ) ;
        info.func = func ;
        info.shared = shared ;
        // info.icvs = t->info.icvs ;
        info.combined_pragma = MPCOMP_COMBINED_NONE;

        __mpcomp_internal_begin_parallel_region( arg_num_threads, info ) ;

  sctk_nodebug(
                  "__mpcomp_start_parallel_region: calling in order scheduler..."
                       ) ;

#if OMPT_SUPPORT
  ompt_task_id_t task_id = OMPT_gen_task_id();
  ompt_frame_t task_frame;
  task_frame.exit_runtime_frame = __builtin_frame_address(0);
  task_frame.reenter_runtime_frame =  0;

  t->mpc_ompt_thread.state = ompt_state_work_parallel;
  t->children_instance->team->ompt_task.task_id = task_id;
  t->mpc_ompt_thread.parent_task_frame = task_frame;

  cb = OMPT_Callbacks[ompt_event_implicit_task_begin];

  if(cb && OMPT_enabled())
   ((ompt_parallel_callback_t)cb)(t->children_instance->team->ompt_parallel_info.parallel_id, task_id);
#endif /* OMPT_SUPPORT */


        /* Start scheduling */
        in_order_scheduler(t->children_instance->mvps[0]) ;

#if OMPT_SUPPORT
   cb = OMPT_Callbacks[ompt_event_implicit_task_end];
  if(cb && OMPT_enabled())
   ((ompt_parallel_callback_t)cb)(t->children_instance->team->ompt_parallel_info.parallel_id, task_id);
#endif /* OMPT_SUPPORT */


  sctk_nodebug(
                  "__mpcomp_start_parallel_region: finish in order scheduler..."
                       ) ;

        __mpcomp_internal_end_parallel_region( t->children_instance ) ;

#if OMPT_SUPPORT
  cb = OMPT_Callbacks[ompt_event_parallel_end];

  if(cb && OMPT_enabled())
   ((ompt_parallel_callback_t)cb)(parallel_id, parent_task_id);
#endif /* OMPT_SUPPORT */

  sctk_nodebug(
      "__mpcomp_start_parallel_region: === EXIT PARALLEL REGION ===\n" ) ;

}
```

Figure B.1: Implementation of OMPT events inside OpenMP parallel region function

```
int __mpcomp_dynamic_loop_begin (long lb, long b, long incr,
                                 long chunk_size, long *from, long *to)
{
        mpcomp_thread_t *t ;    /* Info on the current thread */

        /* Handle orphaned directive (initialize OpenMP environment) */
        __mpcomp_init();

        /* Grab the thread info */
        t = (mpcomp_thread_t *) sctk_openmp_thread_tls ;
        sctk_assert( t != NULL ) ;

#if OMPT_SUPPORT
       ompt_task_id_t task_id;
       ompt_parallel_id_t parallel_id;
       void *parallel_func;

       if(t->rank == 0) { /* Check if we are master thread or not */
         task_id = t->instance->team->ompt_task.task_id;
         parallel_id = t->instance->team->ompt_parallel_info.parallel_id;
       } else {
         task_id = t->mvp->mpc_ompt_thread.task_id;
         parallel_id = t->instance->team->ompt_parallel_info.parallel_id;
       }
       parallel_func = t->instance->team->ompt_parallel_info.parallel_function;

       cb = OMPT_Callbacks[ompt_event_loop_begin];

       if(cb && OMPT_enabled())
        ((ompt_new_workshare_callback_t)cb)(parallel_id, task_id, parallel_func);
#endif /* OMPT_SUPPORT */

#if 0
        /* TODO FOR DEBUGGIN PURPOSE */
        if ( t->for_dyn_shift != NULL ) {
                if ( t->for_dyn_shift[1] != 0 ) {
                        fprintf( stderr, "[%d] __mpcomp_dynamic_loop_begin: "
                                         "error rank %d has shift[1] = %d\n",
                                         t->rank, t->rank, t->for_dyn_shift[1] ) ;
                }
        }
#endif

        /* Initialization of loop internals */
        __mpcomp_dynamic_loop_init(t, lb, b, incr, chunk_size);

        /* Return the next chunk to execute */
        return __mpcomp_dynamic_loop_next(from, to);
}
```

Figure B.2: Insertion of OMPT events inside implementation of OpenMP loops - starting loops

```
void
__mpcomp_dynamic_loop_end_nowait ()
{
        mpcomp_thread_t *t ;    /* Info on the current thread */
        mpcomp_team_t *team_info ;      /* Info on the team */
        int index;
        int num_threads;
        int nb_threads_exited ;

        /* Grab the thread info */
        t = (mpcomp_thread_t *) sctk_openmp_thread_tls ;
        sctk_assert( t != NULL ) ;

#if OMPT_SUPPORT
       ompt_task_id_t task_id;
       ompt_parallel_id_t parallel_id;

       if(t->rank == 0) { /* Check if we are master thread or not */
         task_id = t->instance->team->ompt_task.task_id;
         parallel_id = t->instance->team->ompt_parallel_info.parallel_id;
       } else {
         task_id = t->mvp->mpc_ompt_thread.task_id;
         parallel_id = t->instance->team->ompt_parallel_info.parallel_id;
       }

       cb = OMPT_Callbacks[ompt_event_loop_end];

       if(cb && OMPT_enabled())
        ((ompt_parallel_callback_t)cb)(parallel_id, task_id);
#endif /* OMPT_SUPPORT */


        sctk_debug( "[%d] __mpcomp_dynamic_loop_end_nowait: entering...",
            t->rank ) ;

#if 0
                        /* TODO DEBUGGING PURPOSE */
                        if ( t->for_dyn_shift[1] != 0 ) {
                                fprintf( stderr, "[%d] __mpcomp_dynamic_loop_end_nowait: begin - shift[1] = %d\n",
                                        t->rank, t->for_dyn_shift[1]    ) ;
                                // not_reachable() ;
                        } else {
                                if ( t->rank != 0 ) {
                                        // sleep(1);
                                }
                        }
#endif

        /* Number of threads in the current team */
        num_threads = t->info.num_threads;

        if (num_threads == 1) {
                /* In case of 1 thread, re-initialize the number of remaining chunks
                 * but do not increase the current index */
                index = (t->for_dyn_current) % (MPCOMP_MAX_ALIVE_FOR_DYN + 1);
                sctk_atomics_store_int( &(t->for_dyn_remain[index].i), -1 ) ;
                return ;
        }

        /* Get the team info */
        sctk_assert (t->instance != NULL);
        team_info = t->instance->team ;
        sctk_assert (team_info != NULL);

        /* Compute the index of the dynamic for construct */
        index = (t->for_dyn_current) % (MPCOMP_MAX_ALIVE_FOR_DYN + 1);

        sctk_assert( index >= 0 && index < MPCOMP_MAX_ALIVE_FOR_DYN + 1 ) ;

        sctk_nodebug("[%d] __mpcomp_dynamic_loop_end_nowait: begin", t->rank);

        /* WARNING: the following order is important */
        t->for_dyn_current++ ;
        sctk_atomics_store_int( &(t->for_dyn_remain[index].i), -1 ) ;

        /* Update the number of threads which ended this loop */
        nb_threads_exited = sctk_atomics_fetch_and_incr_int(
                        &(team_info->for_dyn_nb_threads_exited[index].i ) ) ;

        sctk_nodebug(
                        "[%d] __mpcomp_dynamic_loop_end_nowait: Exiting loop %d: %d -> %d",
                        t->rank, index, nb_threads_exited, nb_threads_exited + 1 ) ;

        sctk_assert( nb_threads_exited >= 0 && nb_threads_exited < num_threads ) ;

        if ( nb_threads_exited == num_threads - 1 ) {
                int previous_index ;

                /* WARNING: the following order is important */
                sctk_atomics_store_int(
                                &(team_info->for_dyn_nb_threads_exited[index].i),
                                MPCOMP_NOWAIT_STOP_SYMBOL ) ;

                previous_index = (index - 1 + MPCOMP_MAX_ALIVE_FOR_DYN + 1 ) %
                        ( MPCOMP_MAX_ALIVE_FOR_DYN + 1 ) ;

                sctk_assert( previous_index >= 0 && previous_index < MPCOMP_MAX_ALIVE_FOR_DYN + 1 ) ;

                sctk_nodebug( "__mpcomp_dynamic_loop_end_nowait: Move STOP symbol %d -> %d", previous_index, index ) ;

                sctk_atomics_store_int(
                                &(team_info->for_dyn_nb_threads_exited[previous_index].i),
                                0 ) ;
        }

}
```

Figure B.3: Insertion of OMPT events inside implementation of OpenMP loops - ending loops

Bibliography

[1] Intel MIC architecture. http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

[2] Intel MPI benchmarks. https://software.intel.com/en-us/articles/intel-mpi-benchmarks.

[3] MPI profiling interface. http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/jumpshot-2/node11.html.

[4] TOP500 supercomputer site.

[5] Intel open source openmp runtime. http://www.openmprtl.org, 2014.

[6] George Almsi, Paul Hargrove, Ilie Gabriel, and Tnase Yili Zheng. Upc collectives library 2.0. In *In Fifth Conference on Partitioned Global Address Space Programming Models*, 2011.

[7] Dieter an Mey, Scott Biersdorff, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knüpfer, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Christian Rössel, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A unified performance measurement system for petascale applications. In *Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany, June 2010*, pages 85–97. Gauß-Allianz, Springer, 2012.

[8] Stathopoulos Anastasios. Mixed mode programming on hector, August 2010.

[9] Daniel Atkins. 2.1 the x10 programming language.

[10] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 863–874, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, March 2009.

[12] Dinshaw S. Balsara and Charles D. Norton. Highly parallel structured adaptive mesh refinement using parallel language-based approaches. *Parallel Comput.*, 27(1-2):37–70, January 2001.

[13] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: The architecture and performance of roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 1:1–1:11, Piscataway, NJ, USA, 2008. IEEE Press.

[14] Denis Barthou, Andres Charif Rubial, William Jalby, Souad Koliai, and Cédric Valensi. Performance tuning of x86 openmp codes with MAQAO. In *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, pages 95–113, 2009.

[15] Robert Bell, Allen D. Malony, and Sameer Shende. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In Harald Kosch, Lszl Bszrmnyi, and Hermann Hellwagner, editors, *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 17–26. Springer, 2003.

[16] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82(1):64–84, May 1989.

[17] Fran Berman, Geoffrey Fox, and Anthony J. G. Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[18] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. Technical report, Cambridge, MA, USA, 1996.

[19] Francois Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. Hwloc: A generic framework for managing hardware affinities in hpc applications. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 180–186, Washington, DC, USA, 2010. IEEE Computer Society.

[20] Franois Broquedis, Nathalie Furmento, Brice Goglin, Pierre-Andr Wacrenier, and Raymond Namyst. ForestGOMP: an efficient OpenMP environment for NUMA architectures. *International Journal on Parallel Programming, Special Issue on OpenMP; Guest Editors: Matthias S. Mller and Eduard Ayguad*, 38(5):418–439, 2010.

[21] J. M. Bull. Measuring synchronisation and scheduling overheads in openmp, 1999.

[22] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[23] Franck Cappello and Daniel Etiemble. Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.

[24] Patrick Carribault, Marc Pérache, and Hervé Jourdren. Enabling low-overhead hybrid mpi/openmp parallelism with mpc. In *Proceedings of the 6th International Conference on Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, IWOMP'10, pages 1–14, Berlin, Heidelberg, 2010. Springer-Verlag.

[25] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[26] Anthony Chan, William Gropp, and Ewing Lusk. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, 16(2-3):155–165, 2008.

[27] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[28] Jérôme Clet-Ortega, Patrick Carribault, and Marc Pérache. Evaluation of openmp task scheduling algorithms for large NUMA architectures. In *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, pages 596–607, 2014.

[29] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[30] G. Dhatt, E. Lefrançois, and G. Touzot. *Finite Element Method*. ISTE. Wiley, 2012.

[31] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, NY, USA, 2009. ACM.

[32] Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean thomas Acquaviva, and William Jalby. Maqao: Modular assembler quality analyzer and optimizer for.

[33] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad Van Der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011.

[34] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: Past, present, and future. concurrency and computation: Practice and experience. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003.

[35] J.J. [Tennessee Univ. Dongarra, R. [Gesellschaft fuer Mathematik und Datenverarbeitung mbH Bonn Hempel, A.J.G. [Southampton Univ. (United Kingdom). Dept. of Electronics Hey, Computer Science], and D.W. [Oak Ridge National Lab. Walker. *A proposal for a user-level, message passing interface in a distributed memory environment*. Feb 1993.

[36] David Dureau and Gaël Poëtte. Hybrid parallelism models for neutron monte-carlo solver in an AMR framework. In *Proceedings of the Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013 (SNA + MC 2013)*, 2013.

[37] Alexandre E. Eichenberger, John M. Mellor-Crummey, Martin Schulz, Michael Wong, Nawal Copty, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. Ompt: An openmp tools application programming interface for performance analysis. In Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Mller, editors, *IWOMP*, volume 8122 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2013.

[38] Dawson R. Engler, Gregory R. Andrews, and David K. Lowenthal. Filaments: Efficient support for fine-grain parallelism. Technical report, 1994.

[39] Jianbin Fang, Ana Lucia Varbanescu, Henk J. Sips, Lilun Zhang, Yonggang Che, and Chuanfu Xu. An empirical study of intel xeon phi. *CoRR*, pages –1–1, 2013.

[40] Wu-chun Feng and Kirk Cameron. The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12):50–55, December 2007.

[41] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, Sep. 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[42] Andrew Friedley, Greg Bronevetsky, Torsten Hoefler, and Andrew Lumsdaine. Hybrid mpi: Efficient message passing for multi-core systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 18:1–18:11, New York, NY, USA, 2013. ACM.

[43] Juan Antonio Rico Gallego, Juan Carlos Díaz Martín, Carolina Gómez-Tostón Gutiérrez, and Álvaro Cortés Fácila. Improving collectives by user buffer relocation. In *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, EuroMPI'12, pages 287–288, Berlin, Heidelberg, 2012. Springer-Verlag.

[44] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 22(6):702–719, April 2010.

[45] GNU Project. GCC, the GNU Compiler Collection.

[46] Brice Goglin and StéPhanie Moreaud. Knem: A generic and scalable kernel-assisted intra-node mpi communication framework. *J. Parallel Distrib. Comput.*, 73(2):176–188, February 2013.

[47] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. OpenMPI: A flexible high performance MPI. In *Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics*, PPAM'05, pages 228–239, Berlin, Heidelberg, 2006. Springer-Verlag.

[48] William Gropp. Mpich2: A new start for mpi implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–, London, UK, UK, 2002. Springer-Verlag.

[49] William Gropp and Rajeev Thakur. Thread-safety in an mpi implementation: Requirements and analysis. *Parallel Comput.*, 33(9):595–604, September 2007.

[50] Shinichi Habata, Kazuhiko Umezawa, Mitsuo Yokokawa, and Shigemune Kitawaki. Hardware system of the earth simulator. *Parallel Comput.*, 30(12):1287–1313, December 2004.

[51] Georg Hager, Gabriele Jost, and Rolf Rabenseifner. Communication characteristics and hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. 2009.

[52] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. Implementation and performance analysis of non-blocking collective operations for mpi. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 52:1–52:10, New York, NY, USA, 2007. ACM.

[53] Kevin A. Huck, Allen D. Malony, and Alan Morris. Design and implementation of a parallel performance data management framework. In *Proceedings of the 2005 International Conference on Parallel Processing*, ICPP '05, pages 473–482, Washington, DC, USA, 2005. IEEE Computer Society.

[54] Kevin A. Huck, Allen D. Malony, Sameer Shende, and Doug W. Jacobsen. Integrated measurement for cross-platform openmp performance analysis. In *International Workshop on OpenMP*, Salvador, Bahia, Brazil, 09/2014 2014. Springer, Springer.

[55] Yasuhiro Idomura and Sébastien Jolliet. Performance evaluations of gyrokinetic eulerian code gt5d on massively parallel multi-core platforms. In *State of the Practice Reports*, SC '11, pages 4:1–4:9, New York, NY, USA, 2011. ACM.

[56] Marty Itzkowitz, Oleg Mazurov, Nawal Copty, and Yuan Lin. An openmp runtime api for profiling.

[57] Julien Jaeger, Peter Philippen, Eric Petit, Andres Charif Rubial, Christian Rössel, William Jalby, and Bernd Mohr. Binary instrumentation for scalable performance measurement of openmp applications. In Michael Bader, Arndt Bode, Hans-Joachim Bungartz, Michael Gerndt, Gerhard R. Joubert, and Frans J. Peters, editors, *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 783–792. IOS Press, March 2014.

[58] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[59] Hyun-Wook Jin and Dhabaleswar K. Panda. Limic: Support for high-performance mpi intra-node communication on linux cluster. In *Proceedings of the 2005 International Conference on Parallel Processing*, ICPP '05, pages 184–191, Washington, DC, USA, 2005. IEEE Computer Society.

[60] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, July 2005.

[61] K. Kandalla, H. Subramoni, A. Vishnu, and Dhabaleswar K. P. Designing topology-aware collective communication algorithms for large scale infiniband clusters: Case studies with scatter and gather. 2010.

[62] K. Kandalla, A. Venkatesh, K. Hamidouche, S. Potluri, D. Bureddy, and D. K. Panda. Designing optimized mpi broadcast and allreduce for many integrated core (mic) infiniband clusters. In *Proceedings of the 2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, HOTI '13, pages 63–70, Washington, DC, USA, 2013. IEEE Computer Society.

[63] Krishna Kandalla, Hari Subramoni, Gopal Santhanaraman, Matthew Koop, and Dhabaleswar K. Panda. Designing multi-leader-based allgather algorithms for multi-core clusters. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[64] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the open trace format (otf). In *Proceedings of the 6th International Conference on Computational Science - Volume Part II*, ICCS'06, pages 526–533, Berlin, Heidelberg, 2006. Springer-Verlag.

[65] Géraud Krawezik. Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '03, pages 118–127, New York, NY, USA, 2003. ACM.

[66] Christoph Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40:40–40:51, July 2013.

[67] Seyong Lee and Rudolf Eigenmann. Openmpc: Extended openmp programming and tuning for gpus. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[68] Shigang Li, Torsten Hoefler, and Marc Snir. NUMA-aware shared-memory collective communication for MPI. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 85–96, New York, NY, USA, 2013. ACM.

[69] Chunhua Liao, Oscar Hern, Barbara Chapman, Wenguang Chen, and Weimin Zheng. Openuh: An optimizing, portable openmp compiler. In *In 12th Workshop on Compilers for Parallel Computers*, 2006.

[70] Chunhua Liao, Yonghong Yan, Bronis R de Supinski, Daniel J Quinlan, and Barbara Chapman. Early experiences with the openmp accelerator model. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 84–98. Springer, 2013.

[71] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High performance rdma-based mpi implementation over infiniband. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 295–304, New York, NY, USA, 2003. ACM.

[72] Zhenying Liu, Barbara Chapman, Tien-Hsiung Weng, and Oscar Hernandez. Improving the performance of openmp by array privatization. In *Proceedings of the OpenMP Applications and Tools 2003 International Conference on OpenMP Shared Memory Parallel Programming*, WOMPAT'03, pages 244–259, Berlin, Heidelberg, 2003. Springer-Verlag.

[73] Kevin London, Shirley Moore, Philip Mucci, Keith Seymour, and Richard Luczak. The papi cross-platform interface to hardware performance counters. In *Department of Defense Users Group Conference Proceedings*, pages 18–21, 2001.

[74] Teng Ma, George Bosilca, Aurelien Bouteiller, and Jack Dongarra. HierKNEM: An adaptive framework for kernel-assisted and topology-aware collective communications on many-core clusters. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 970–982, Washington, DC, USA, 2012. IEEE Computer Society.

[75] D. J. C. MacKay. Learning in graphical models. chapter Introduction to Monte Carlo Methods, pages 175–204. MIT Press, Cambridge, MA, USA, 1999.

[76] Aurèle Mahéo, Patrick Carribault, Marc Pérache, and William Jalby. Optimizing collective operations in hybrid applications. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 121:121–121:122, New York, NY, USA, 2014. ACM.

[77] Aurèle Mahéo, Souad Koliaï, Patrick Carribault, Marc Pérache, and William Jalby. Adaptive openmp for large numa nodes. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 254–257, Berlin, Heidelberg, 2012. Springer-Verlag.

[78] Allen D. Malony. Event-based performance perturbation: A case study. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, pages 201–212, New York, NY, USA, 1991. ACM.

[79] Amith R. Mamidala, Rahul Kumar, Debraj De, and D. K. Panda. MPI collectives on modern multicore clusters: Performance optimizations and communication characteristics. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '08, pages 130–137, Washington, DC, USA, 2008. IEEE Computer Society.

[80] Henry Markram, Karlheinz Meier, Thomas Lippert, Sten Grillner, Richard S. Frackowiak, Stanislas Dehaene, Alois Knoll, Haim Sompolinsky, Kris Verstreken, Javier DeFelipe, Seth Grant, Jean-Pierre Changeux, and Alois Saria. Introducing the human brain project. *Procedia CS*, 7:39–42, 2011.

[81] D. McMorrow and Mitre Corporation. *Technical Challenges of Exascale Computing*. MITRE Corporation, 2013.

[82] Seung-jai Min, Costin Iancu, and Katherine Yelick. Hierarchical Work Stealing on Manycore Clusters. pages 1–10, 2011.

[83] Bernd Mohr, Allen D. Malony, Sameer Shende, and Felix Wolf. Design and prototype of a performance tool interface for openmp. *J. Supercomput.*, 23(1):105–128, August 2002.

[84] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011.

[85] W. E. Nagel, A. Arnold, M. Weber, H.-Ch. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. *Supercomputer*, 12:69–80, 1996.

[86] Ramachandra Nanjegowda, Oscar Hernandez, Barbara Chapman, and Haoqiang H. Jin. Scalability evaluation of barrier algorithms for openmp. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 42–52, Berlin, Heidelberg, 2009. Springer-Verlag.

[87] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[88] Marc Pérache, Patrick Carribault, and Hervé Jourdren. Mpc-mpi: An mpi implementation reducing the overall memory consumption. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 94–103, Berlin, Heidelberg, 2009. Springer-Verlag.

[89] Rolf Rabenseifner. Optimization of collective reduction operations. In *Computation Science - ICCS 2004*, ICCS ' 2004, 2004.

[90] Rolf Rabenseifner and Gerhard Wellein. Communication and optimization aspects of parallel programming models on hybrid architectures. *IJHPCA*, 17(1):49–62, 2003.

[91] Y. Rasera, J-M. Alimi, J. Courtin, F. Roy, P-S. Corasaniti, A. Fuzfa, and V. Boucher. Introducing the Dark Energy Universe Simulation Series (DEUSS). *AIP Conference Proceedings*, 1241:1134–1139, 2010. `http://www.deus-consortium.org/`.

[92] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. Pin: A binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture*, WCAE '04, New York, NY, USA, 2004. ACM.

[93] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[94] Christiane Pousa Ribeiro, Márcio Castro, Jean-François Méhaut, and Alexandre Carissimi. Improving memory affinity of geophysics applications on numa platforms using minas. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, pages 279–292, Berlin, Heidelberg, 2011. Springer-Verlag.

[95] Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21(1):63–72, January 1978.

[96] Jose Carlos Sancho, Fabrizio Petrini, Kei Davis, Roberto Gioiosa, and Song Jiang. Current practice and a direction forward in checkpoint/restart implementations for fault tolerance. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18 - Volume 19*, IPDPS '05, pages 300.2–, Washington, DC, USA, 2005. IEEE Computer Society.

[97] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[98] Sameer Shende, Allen D. Malony, Wyatt Spear, and Karen Schuchardt. Characterizing I/O performance using the TAU performance system. In *Applications, Tools and Techniques on the Road to Exascale Computing, Proceedings of the conference ParCo 2011, 31 August - 3 September 2011, Ghent, Belgium*, pages 647–655, 2011.

[99] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[100] Fengguang Song, Felix Wolf, Nikhil Bhatia, Jack Dongarra, and Shirley Moore. An algebra for cross-experiment performance analysis. In *Proceedings of the 2004 International Conference on Parallel Processing*, ICPP '04, pages 63–72, Washington, DC, USA, 2004. IEEE Computer Society.

[101] Rajeev Thakur and William Gropp. Test suite for evaluating performance of mpi implementations that support mpi_thread_multiple. In *Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, PVM/MPI'07, pages 46–55, Berlin, Heidelberg, 2007. Springer-Verlag.

[102] James E. Thornton. The cdc 6600 project. *IEEE Annals of the History of Computing*, 2(4):338–348, 1980.

[103] Wendy Torell and Victor Avelar. Mean time between failure: Explanation and standards.

[104] B. Tu, M. Zou, J. Zhan, X. Zhao, and J. Fan. Multi-core aware optimization for MPI collectives. 2008.

[105] UPC Consortium. Upc language specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[106] J.Y. Vet. *Parallélisme de tâches et localité de données dans un contexte multi-modèle de programmation pour supercalculateurs hiérarchiques et hétérogènes*. 2013.

[107] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc: First experiences with real-world applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.

[108] Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid mpi/openmp applications. *J. Syst. Archit.*, 49(10-11):421–439, November 2003.

[109] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with jumpshot. *Int. J. High Perform. Comput. Appl.*, 13(3):277–288, August 1999.

Résumé

Afin de répondre aux besoins de plus en plus importants en puissance de calcul de la part des applications numériques, les supercalculateurs ont dû évoluer et sont ainsi de plus en plus compliqués à programmer. Ainsi, en plus de l'apparition des systèmes à mémoire partagée, des architectures dites NUMA (Non Uniform Memory Access) sont présentes au sein de ces machines, fournissant plusieurs niveaux de parallélisme. Une autre contrainte, la diminution de la mémoire disponible par coeur de calcul, doit être soulignée. C'est ainsi que des modèles parallèles tels que MPI (Message Passing Interface) ne permettent plus aux codes scientifiques haute performance de passer à l'échelle et d'exploiter efficacement les machines de calcul, et doivent donc être combinés avec d'autres modèles plus adaptés aux architectures à mémoire partagée. OpenMP, en tant que modèle standardisé, est un choix privilégié pour être combiné avec MPI. Mais mélanger deux modèles avec des paradigmes différents est une tâche compliquée et peut engendrer des goulets d'étranglement qui doivent être identifiés. Cette thèse a pour objectif d'aborder ces limitations et met en avant plusieurs contributions couvrant divers aspects. Notre première contribution permet de réduire le surcoût des supports exécutifs OpenMP en optimisant le travail d'activation et de synchronisation des threads OpenMP pour les codes MPI+OpenMP. Dans un second temps, nous nous focalisons sur les opérations collectives MPI. Notre contribution a pour but d'optimiser l'opération MPI_Allreduce en réutilisant des unités de calcul inoccupées, et faisant intervenir des threads OpenMP. Nous introduisons également le concept de collectives unifiées, impliquant des tâches MPI et des threads OpenMP dans une même opération. Enfin, nous nous intéressons à l'analyse de performance et plus précisément l'instrumentation des applications MPI+OpenMP, et notre dernière contribution consiste en l'implémentation et l'évaluation de l'outil OpenMP Tools API (OMPT) dans le support exécutif OpenMP du framework MPC. Cet outil nous permet d'instrumenter des constructions OpenMP et de conduire une analyse axée aussi bien du côté des applications que des supports d'exécution.

Mots-clés: MPI, OpenMP, Modèle hybride, support exécutif, OMPT

Abstract

To provide increasing computational power for numerical simulations, supercomputers evolved and are now more and more complex to program. Indeed, after the appearance of shared memory systems emerged architectures such as NUMA (Non Uniform Memory Access) systems, providing several levels of parallelism. Another constraint, the decreasing amount of memory per compute core, has to be mentioned. Therefore, parallel models such as Message Passing Interface (MPI) are no more sufficient to enable scalability of High Performance applications, and have to be coupled with another model adapted to shared memory architectures. OpenMP, as a de facto standard, is a good candidate to be mixed with MPI. The principle is to use this model to augment legacy codes already parallelized with MPI. But hybridizing scientific codes is a complex task, bottlenecks exist and need to be identified. This thesis tackles these limitations and proposes different contributions following various aspects. Our first contribution reduces the overhead of the OpenMP layer by optimizing the creation and synchronization of threads for MPI+OpenMP codes. On a second time, we target MPI collective operations. Our contribution consists in proposing a technique to exploit idle cores in order to help the operation, with the example of MPI_Allreduce collective. We also introduce unified Collectives involving both MPI tasks and OpenMP threads. Finally, we focus on performance analysis of hybrid MPI+OpenMP codes, and our last contribution consists in the implementation of OpenMP Tools API (OMPT), an instrumentation tool, inside the OpenMP runtime of MPC framework. This tool allows us to instrument and profile OpenMP constructs and allows the analysis of both runtime and application sides.

Keywords: MPI, OpenMP, Hybrid model, runtime, OMPT