Table des matières

Chapitre 1

Introduction

La simulation numérique est un outil incontournable pour l'industrie. Elle rassemble des industriels tels que l'automobile, l'aéronautique, l'énergie et bien d'autres. La simulation numérique apporte la possibilité de réduire les coûts des expériences très coûteuses. Par exemple, lors de la conception de nouvelles voitures, les constructeurs peuvent simuler des milliers de crashs tests avant d'en effectuer dans des conditions réelles. Il en est de même pour les essais en soufflerie, permettant d'optimiser l'aérodynamisme d'une voiture ou bien d'une aile d'avion. Les grandes compagnies pétrolières effectuent des simulations numériques permettant de réduire le risque de faire un forage là où il n'y a pas de pétrole. L'utilisation de la simulation numérique ne permet pas seulement de faire des économies. Elle permet aussi de simuler des expériences dangereuses sans les contraintes qui y sont liées, ou bien d'effectuer des simulations de phénomènes impossibles à expérimenter. C'est le cas du projet DEUS qui a permis de simuler la structure de l'univers observable depuis le Big-Bang [2]. Pour réaliser toutes ces expériences, les scientifiques s'appuient sur la puissance des machines que l'on appelle aujourd'hui les supercalculateurs. La science qui régit ces supercalculateurs est « l'Informatique Haute Performance » ou bien « High Performance Computing » (HPC).

Les supercalculateurs sont constitués de plusieurs machines inter-connectées. Afin de toutes les utiliser pour réaliser un seul calcul, il est nécessaire qu'elles communiquent entre elles. Le standard le plus répandu pour la programmation parallèle est nommé « Message Passing Interface » (MPI). Ce standard spécifie une interface permettant la programmation parallèle principalement par le biais d'envois de messages. Il s'agit de l'interface standard utilisée pour effectuer des communications dans les applications HPC. Le coût des communications est un des problèmes majeurs pour avoir de bonnes performances dans les applications MPI. Pour amortir le coût des communications MPI, les développeurs d'applications ont pour but de les recouvrir par du calcul en utilisant les primitives MPI non-bloquantes, permettant aux communications de s'effectuer en arrière-plan pendant que le calcul s'exécute sur les CPU.

Objectifs et contributions

Initialement, les communications non-bloquantes étaient uniquement disponibles pour les opérations entre 2 processus MPI : les communications point-à-point. L'extension des communications non-bloquantes aux opérations impliquant plus de 2 processus MPI, les opérations collectives, est apparue dans la version 3.0 de du standard MPI en 2012 [1]. Cela a ouvert la possibilité de recouvrir les communications collectives non-bloquantes par du calcul. Cependant, les communications collectives non-bloquantes consomment plus de temps CPU que les communications point-à-point, elles sont donc plus difficiles à faire progresser en arrière-plan. La conséquence est que les codes utilisant les collectives non-bloquantes ont un faible recouvrement (souvent aucun) car les différentes implémentations du standard MPI gèrent mal le placement et l'ordonnancement des threads de progression permettant la progression de ces communications en tâche de fond. En effet, la réalisation de communications complexes avec des dépendances entre les différentes tâches MPI ainsi que les opérations locales (réduction) demande l'existence de ressources de calcul pouvant exécuter ces opérations. Mais en général, le partage des ressources de calcul est quasi prohibé dans les codes HPC. Les collectives MPI non-bloquantes ne sont donc pas souvent utilisées dans des codes de calcul alors que cela pourrait apporter un réel gain de performance.

Dans cette thèse, nous proposons différents algorithmes pour recouvrir les communications avec du calcul sans dégrader les performances. Pour cela, nous proposons d'aborder ce problème sous plusieurs angles. D'une part, nous nous concentrons sur le placement des threads de progression générés par les collectives MPI non-bloquantes. Nous proposons deux algorithmes de placement des threads de progression pour toutes les collectives MPI non-bloquantes. Le premier est de regrouper les threads de progression sur des cœurs libres. Le second est de placer les threads de progression sur les hyper-threads. Pour être plus efficace, nous nous concentrons ensuite sur l'optimisation de deux types d'algorithme utilisés pour les opérations collectives : les algorithmes en arbre et les algorithmes en chaîne.

D'autre part, nous avons aussi étudié l'ordonnancement des threads de progression afin que les threads inutiles à la progression de l'algorithme ne s'exécutent pas. Pour cela, nous proposons d'abord d'utiliser un mécanisme permettant de suspendre l'ordonnancement de ces threads, puis de forcer l'ordonnancement optimal des threads de progression de façon statique à l'aide de sémaphores. Enfin, une politique d'ordonnancement avec des priorités a été mise en place.

Organisation du document

Ce document est organisé en trois parties. La première partie présente le contexte et est constituée du chapitre 2 qui introduit toutes les notions essentielles à la bonne compréhension du document et du chapitre 3 qui présente l'état de l'art et la problématique de cette thèse. La deuxième partie présente les contributions. Nous proposons des algorithmes de placement statique pour les tâches MPI et les threads de progression dans le chapitre 4. Ensuite, nous proposons des algorithmes de placement dynamique pour les threads de progression en fonction des algorithmes de collectives utilisés dans le chapitre 5. Enfin, nous proposons des optimisations et des politiques d'ordonnancement pour les threads de progression sur les cœurs dédiés aux communications dans le chapitre 6. La dernière partie conclut ce document. Nous y résumons nos contributions et proposons une ouverture sur des perspectives à moyen et long terme dans le chapitre 7.

Publications et communications

Les travaux que nous présentons dans ce document ont fait l'objet de plusieurs publications et communications sur l'amélioration du recouvrement des collectives MPI non-bloquantes.

Conférence et Workshop internationaux avec comité de lecture

Publication et communication : Hugo Taboada, Alexandre Denis, Julien Jaeger, Emmanuel Jeannot, Marc Pérache. « Dynamic Placement of Progress Thread for Overlapping MPI Non-Blocking Collectives on Manycore Processor », Dans *EURO-PAR 2018 : 24th International European Conference on Parallel and Distributed Computing*, août 2018, Turin, Italie.

Publication et communication : Hugo Taboada, Alexandre Denis, Julien Jaeger. « Progress Thread Placement for Overlapping MPI Non-Blocking Collectives using Simultaneous Multi-Threading », Dans *COLOC : 2nd workshop on data locality, in conjuction with EURO-PAR 2018*, août 2018, Turin, Italie.

Conférences nationales avec comité de lecture

Publication, poster et communication : Hugo Taboada. « Recouvrement des Collectives MPI Non-Bloquantes sur Processeur Manycore », Dans

Compas 2018 : conférence d'informatique en Parallélisme, Architecture et Système, juillet 2018, Toulouse, France.

Publication et communication : Hugo TABOADA. « Impact du placement des threads de progression pour les collectives MPI non-bloquantes », Dans *Compas 2016 : conférence d'informatique en Parallélisme, Architecture et Système*, juillet 2016, Lorient, France.

Séminaire

Communication : Hugo TABOADA. « Impact du placement des threads de progression pour les collectives MPI non-bloquantes », Dans *Inhp@ct n° 38 (21 janvier 2015) et Inhp@ct n° 47 (9 juin 2016)*, Bruyères-le-châtel, France.

Première partie

Contexte

Chapitre 2

L'informatique haute performance

Sommaire

Pour concevoir les machines utilisées en HPC, il est nécessaire d'assembler des composants de diverses natures. L'architecture de ces machines est le fruit de nombreuses évolutions depuis l'invention du microprocesseur en 1969 par M. Hoff et F. Fagin alors ingénieurs chez Intel.

Le but de ce chapitre est de définir certaines notions indispensables à la bonne compréhension du manuscrit de thèse sans pour autant faire une

description chronologique des avancées technologiques dans le domaine de l'informatique.

2.1 L'évolution des architectures HPC

Avant l'invention du circuit intégré, les processeurs étaient constitués de plusieurs composants électroniques interconnectés. Cette invention permit de placer tous ces composants sur un seul circuit intégré afin de créer le microprocesseur. En 1971, la société américaine INTEL commercialise le premier microprocesseur : l'INTEL 4004.

Depuis, le nombre de transistors dans les microprocesseurs ne cesse de croître. L'augmentation de la finesse de gravure des microprocesseurs a permis d'avoir de plus en plus de transistors par unité de surface disponible sur un microprocesseur, ce qui contribue à l'augmentation de la puissance de calcul des microprocesseurs. En 1965, G. E. MOORE prédit que le nombre de transistors dans un circuit intégré doublerait tous les ans [3]. En 1975, il réévalue sa prédiction et prédit que le nombre de transistors dans un microprocesseur doublerait tous les 18 mois. Nous pouvons voir que cette prédiction s'avère juste sur la figure 2.1.1 représentant la croissance du nombre de transistors dans les microprocesseurs de 1971 à 2016.

Cependant, la finesse de gravure des microprocesseurs atteint bientôt ses limites physiques. En effet, après un certain seuil, il ne sera plus possible d'augmenter la finesse de gravure tout en gardant les propriétés physiques des matériaux qui permettent aux microprocesseurs de fonctionner correctement.

2.1.1 Le parallélisme d'instruction

Afin d'améliorer les performances des microprocesseurs, l'utilisation du parallélisme d'instruction où *Instruction-Level Parallelism* (ILP) est particulièrement importante. La technique est d'utiliser un *pipeline d'instruction* (figure 2.1.2). Cela consiste à diviser le traitement d'une instruction en N étapes et de recouvrir l'étape j de l'instruction i avec l'étape $j+1$ de l'instruction $i-1$. De cette façon, il est possible de traiter plusieurs étapes provenant d'instructions différentes en parallèle. Dans le cas idéal, au cycle N, le pipeline est rempli et le débit d'instructions est maximal. Cependant, le pipeline n'est pas toujours rempli au maximum. Lorsque l'étape d'une instruction ne peut pas être exécutée à cause d'une dépendance de donnée, il se produit un phénomène appelé « bulle » retardant l'exécution des instructions suivantes. Ce phénomène est illustré dans la figure 2.1.3. Plusieurs techniques existent pour limiter les bulles dans le pipeline telles que la prédiction de branchement ou bien le pipeline *out-of-order*. La prédiction de branchement consiste à remplir le pipeline avec les instructions d'un des branchements. Si la prédiction s'avère juste, le

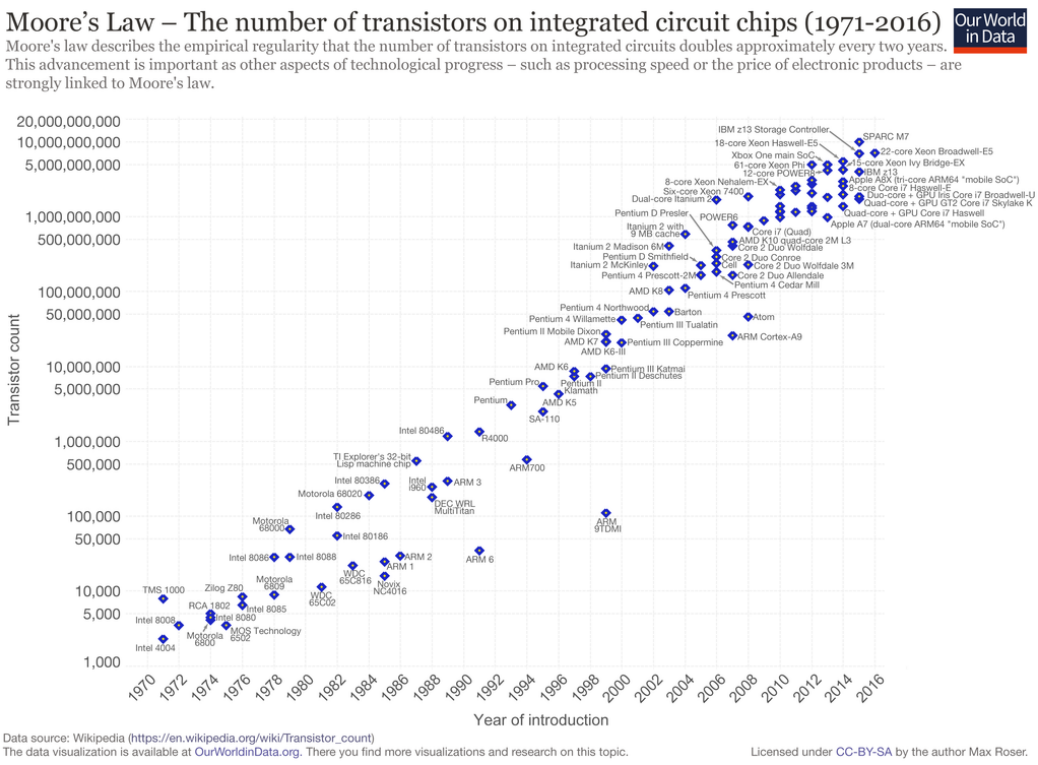


FIGURE 2.1.1 – Croissance du nombre de transistors dans les microprocesseurs par rapport à la loi de MOORE. [4]

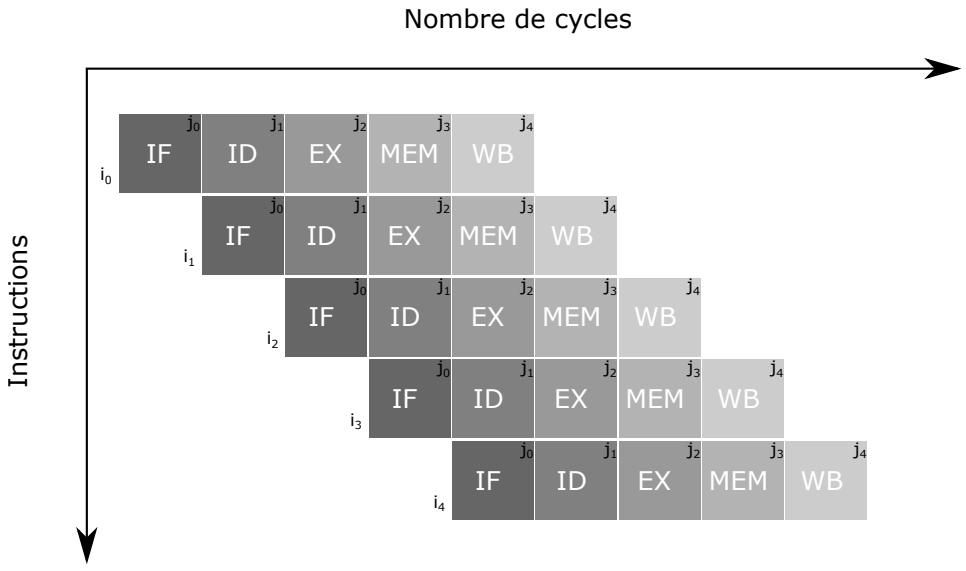


FIGURE 2.1.2 – Illustration d'un pipeline d'instructions à 5 étapes (IF : Instruction Fetch, ID : Instruction Decode, EX : Execute, MEM : Memory access, WB : Register wirte back) de j_0 à j_4. Au $5^{\text{ième}}$ cycle d'horloge, le pipeline est rempli.

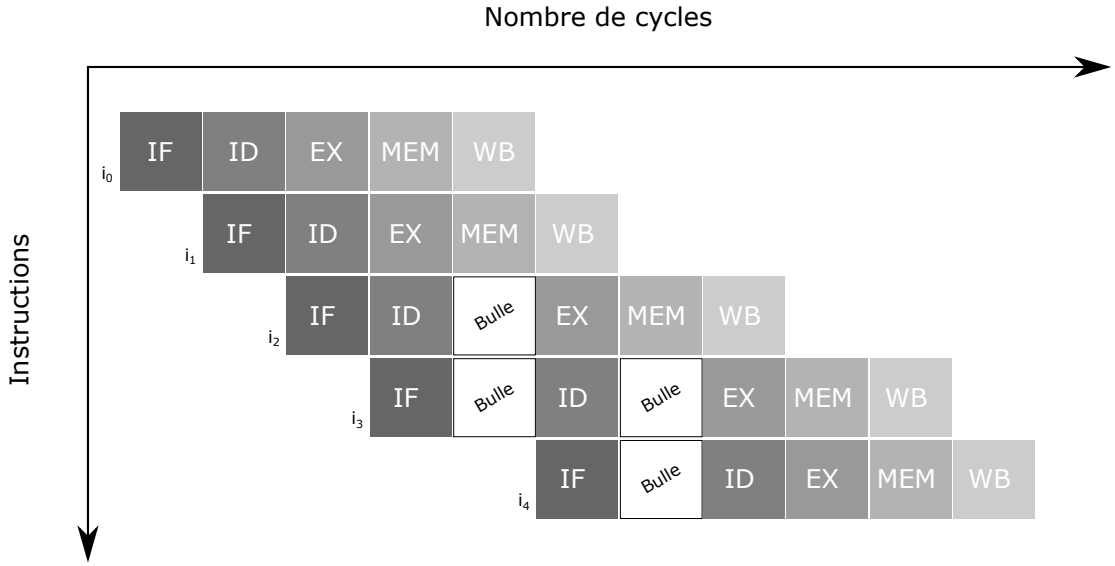


FIGURE 2.1.3 – Illustration d'une bulle se produisant dans un pipeline d'instruction à 5 étapes (IF : Instruction Fetch, ID : Instruction Decode, EX : Execute, MEM : Memory access, WB : Register wirte back).

pipeline restera rempli et aucun temps n'est perdu. Dans le cas contraire, le pipeline est réinitialisé. Le pipeline *out-of-order* optimise l'ordre d'exécution des instructions dans le pipeline afin d'éviter les bulles dans le pipeline. Toutes ces techniques permettent d'améliorer le nombre de cycles par instruction (CPI) du pipeline.

Le temps pris pour exécuter les instructions d'un programme sur un microprocesseur, noté T, peut être décrit par l'équation 2.1.

$$T = N \times CPI \times P \tag{2.1}$$

Il dépend de trois paramètres : le nombre d'instructions du programme (N), le CPI et la durée d'un cycle d'horloge (P). En remplaçant la durée d'un cycle d'horloge par sa fréquence, nous obtenons l'équation 2.2.

$$T = \frac{N \times CPI}{F} \tag{2.2}$$

Une des façons simples pour améliorer les performances d'un microprocesseur est donc d'augmenter sa fréquence d'horloge. C'est la raison pour laquelle les sociétés fabriquant des microprocesseurs ont mené une course à la fréquence pour augmenter leurs performances. Il suffisait donc aux utilisateurs de supercalculateurs d'attendre la génération suivante de processeur pour bénéficier d'une augmentation des performances de leurs programmes. C'était sans compter que la consommation énergétique (E) des microprocesseurs est proportionnelle à la fréquence (F), au carré de la tension (V) et la capacité électrique (C) tel que

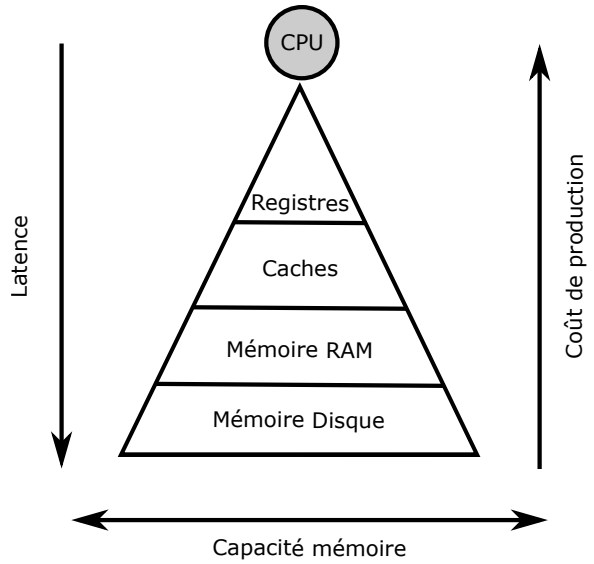


FIGURE 2.1.4 – La hiérarchie de la mémoire

décrit par l'équation 2.3.

$$E = C \times V^2 \times F \tag{2.3}$$

Étant donné que la consommation électrique génère de la chaleur, l'augmentation de la fréquence d'horloge devient problématique. Cela provoque une hausse de la température des microprocesseurs jusqu'à atteindre un seuil où il n'est plus possible d'augmenter la fréquence d'un microprocesseur sans qu'il soit inutilisable.

2.1.2 La hiérarchie de la mémoire

Afin de réaliser des calculs, les microprocesseurs sont dotés d'unités arithmétiques et logiques. Néanmoins, ces calculs doivent être effectués sur des données stockées en mémoire.

2.1.2.1 Les composants matériels

Il existe différents types de mémoires établis dans ce qu'on appelle « la hiérarchie de la mémoire ». Cette hiérarchie est née d'un compromis entre la latence de la mémoire, sa capacité et son coût de production (figure 2.1.4).

Les registres : Accéder à la mémoire coûte des cycles d'horloge. Plus la mémoire est proche du microprocesseur et moins de cycles il faudra pour charger une donnée en mémoire. Afin d'avoir des bonnes performances, il est donc nécessaire d'avoir la mémoire au plus proche du microprocesseur. C'est pourquoi, les unités arithmétiques et logiques ne travaillent que sur les données

stockées dans les registres. Les registres sont au plus haut niveau de la hiérarchie mémoire. Il s'agit d'une mémoire très rapide intégrée aux microprocesseurs et c'est la seule mémoire en accès direct. Le nombre de registres dépend du microprocesseur utilisé mais leur nombre est généralement très limité.

Les caches : La mémoire cache est aussi intégrée aux microprocesseurs. Il s'agit de niveaux de mémoire avec des protocoles de cohérence afin d'avoir des données toujours valides lorsque le CPU a besoin d'une donnée qui y est stockée. En effet, étant donné que le CPU ne travaille qu'avec les registres, il est nécessaire de transférer les données des caches vers les registres.

La mémoire cache est disponible en très faible quantité et ne nécessite que très peu de cycles pour y accéder. Afin d'optimiser les performances des codes de calcul, il est commun de vouloir utiliser efficacement cette mémoire.

La mémoire centrale (RAM) : La mémoire RAM *Random Access Memory* est la mémoire principale d'un ordinateur. C'est là où sont chargés les programmes quand ils sont exécutés. Néanmoins, accéder à cette mémoire coûte approximativement 300 cycles [5]. C'est donc une mémoire lente si nous voulons avoir un programme performant.

Les disques : Les disques sont utilisés pour stocker le système de fichier d'un ordonnateur. Accéder à cette mémoire coûte approximativement 2 millions de cycles [5]. Il existe aussi le stockage sur bande magnétique pour archiver les données. Elles peuvent contenir des données sur de très longue période. Ils ne nécessitent pas de courant pour garder les données en mémoire contrairement aux mémoires que nous avons vues précédemment.

2.1.2.2 Temps d'accès aux données non uniforme

Bien que le temps d'accès aux données varie en fonction de la hiérarchie mémoire (latence), il est possible que le temps d'accès entre deux mémoires du même niveau de la hiérarchie ne soit pas uniforme. Un système NUMA (Non Uniform Memory Access) est constitué de plusieurs microprocesseurs disposant chacun d'une zone de mémoire propre. Comme l'illustre la figure 2.1.5, les CPU sont reliés entre eux par des liens QPI [6] sur les processeurs INTEL, et peuvent utiliser n'importe quelle mémoire de manière cohérente. Cependant, les temps d'accès ne seront pas uniformes suivant que le CPU accède à des données sur la mémoire locale ou bien à des données se trouvant sur un banc mémoire d'un autre CPU.

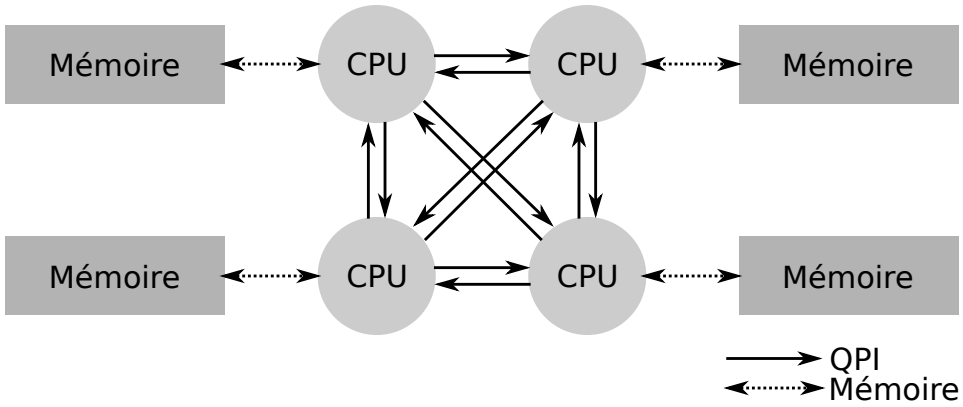


FIGURE 2.1.5 – Exemple d'architecture NUMA

2.1.3 La technologie « Simultaneous Multi-Threading »

La technologie *Simultaneous Multi-Threading* [7], plus connue sous le nom d'Hyper-Threading pour les microprocesseurs d'INTEL, est incluse dans la plupart des microprocesseurs actuels. Il s'agit, pour un cœur, d'exécuter simultanément plusieurs threads matériels se partageant les mêmes unités arithmétiques et logiques. Selon la terminologie d'INTEL, ces threads matériels sont appelés des *Hyper-Threads*.

Concrètement, il s'agit de la duplication des registres de données et de contrôle au sein même d'un cœur permettant l'exécution de plusieurs fils d'exécution ou « threads » sur ce même cœur. Les Hyper-Threads se partagent donc non seulement les mêmes unités arithmétiques et logiques mais aussi la même hiérarchie mémoire dont les caches font partie. Le but est d'exécuter plusieurs threads sur le même pipeline d'instruction. Cela permet d'avoir moins de bulles et d'améliorer l'ILP. C'est pourquoi l'utilisation des Hyper-Threads n'est pas toujours synonyme de performance. En effet, l'utilisation des mêmes unités arithmétiques et logiques par plusieurs Hyper-Threads signifie que si tous les threads ont besoin des mêmes unités au même moment, il ne sera pas possible de le faire.

2.1.4 Les microprocesseurs multi-cœurs

L'augmentation du nombre de transistors a d'abord servi à améliorer l'ILP. Les transistors étaient utilisés pour implémenter le pipeline d'instruction, la prédiction de branchement, le pipeline *out-of-order*, les *Hyper-threads*, les caches et bien d'autres optimisations matérielles décrites dans *Computer Architecture* [8]. Ensuite, dès lors que l'augmentation du nombre de transistors ne permettait plus d'amélioration des performances avec ces optimisations, l'augmentation de la finesse de gravure a laissé le champ libre à l'avènement des microprocesseurs multi-cœurs. En 2001, IBM commercialise le premier microprocesseur

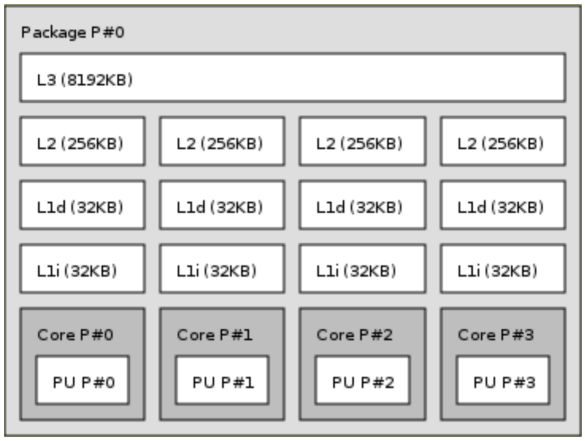


FIGURE 2.1.6 – Topologie d'un microprocesseur multi-cœurs obtenue à l'aide du logiciel hwloc [10].

multi-coeur : le POWER4 [9]. Il s'agit de mettre plusieurs microprocesseurs sur une même puce afin d'augmenter les performances d'une machine. Ces microprocesseurs gravés sur une même puce sont appelés des « cœurs ». Nous pouvons voir la topologie de ce type de microprocesseur sur la figure 2.1.6. Ils se partagent généralement une partie de la hiérarchie mémoire et notamment le dernier niveau de la mémoire cache. La mémoire cache est une mémoire qui est intégrée au microprocesseur. Il existe plusieurs niveaux de cache selon la génération du processeur. Chaque niveau est plus ou moins proche des unités arithmétiques et logiques ou « Arithmetic Logic Units » (ALU). Plus le niveau de cache est proche des unités de calcul et plus le cache est rapide. La capacité de la mémoire cache est cependant très limitée. Cette capacité dépend aussi de la génération du processeur et est de l'ordre du Mega-octet pour les derniers niveaux de cache et du Kilo-octet pour les niveaux de cache les plus proches des unités de calcul.

Les microprocesseurs multi-cœurs peuvent exécuter plusieurs threads et rendent le calcul parallèle possible au sein d'une même puce. Chaque cœur est un microprocesseur complet et possède ses propres unités de calcul contrairement aux *Hyper-Threads* qui doivent se partager l'ALU.

Plusieurs modèles de programmation permettent de programmer sur ce type d'architecture. Nous verrons cela dans la section 2.2.

2.1.5 Les microprocesseurs manycores

Les microprocesseurs manycores sont des architectures avec beaucoup plus de cœurs que les microprocesseurs multi-cœurs mais avec des cœurs moins puissants. Ainsi, ils bénéficient de plus de parallélisme.

Comme nous l'avons vu, les microprocesseurs ne sont pas seulement dotés d'unités de calcul, de mémoires caches, et de registres, mais aussi de circuits

servant à optimiser l'ordre d'exécution des instructions dans le but de réduire le CPI tels que le pipeline *out-of-order*. Ces circuits ne sont pas présents sur tous les microprocesseurs manycores. La raison est que ces circuits consomment de l'énergie et prennent de l'espace sur la puce. La conséquence de l'absence de circuit permettant de réduire le CPI est qu'il reste plus d'espace sur la puce. Ainsi, les concepteurs de microprocesseurs peuvent augmenter le nombre de cœurs sur les puces.

Les Xeon Phi de chez Intel : La première version des Xeon Phi à être commercialisée était dénommée « Intel Knights Corner (*KNC*) » [11] sortie en 2013. Il s'agissait d'un coprocesseur doté de 57 à 61 cœurs cadencés de 1.1 à 1.2 GHz avec 4 Hyper-Threads par cœur. La finesse de gravure était de 22 nm et celui-ci ne disposait pas de pipeline *out-of-order*. Cela permit d'avoir plus de surface sur la puce pour y mettre les cœurs.

La version la plus récente des Xeon Phi est dénommée l'« Intel Knights Landing (KNL) » [12] sortie en 2016. Il s'agit d'un microprocesseur doté de 64 à 72 cœurs cadencés de 1.3 à 1.5 GHz avec 4 Hyper-Threads par cœur. La finesse de gravure est de 14 nm. Ce microprocesseur possède un pipeline *out-of-order* mais le *buffer* utilisé pour réordonnancer les instructions est beaucoup plus petit que dans les processeurs Xeon courants. Le pipeline est donc moins efficace sur les Xeon Phi.

Les mppa de chez Kalray : Il s'agit de microprocesseurs manycores qui peuvent avoir jusqu'à 288 cœurs pour les mppa 2-256 [13] sortis en 2013. Ces cœurs sont disposés en 16 groupes de 16 cœurs plus 1 cœur de contrôle, ainsi que 4 quad-cores. La programmation sur ce microprocesseur s'effectue avec des noyaux de calcul comme pour les processeurs graphiques.

Les accélérateurs graphiques : Il s'agit de microprocesseurs se trouvant sur une carte graphique. Ils se distinguent par un nombre très important de « cœurs ». Cependant, ces cœurs ne sont pas des cœurs classiques que l'on peut retrouver sur les microprocesseurs. Il s'agit d'unités de calcul très simples qui exécutent la même instruction sur toutes les unités de calcul par tic d'horloge permettant la programmation avec le paradigme « Single Instruction Multiple Data » (*SIMD*) selon la taxonomie de Flynn [14] représentée sur la figure 2.1.7. Nvidia appelle cela « Single Instruction Multiple Threads » (SIMT).

Dans une certaine mesure, nous pouvons aussi dire que les processeurs graphiques ou « General Purpose processing on Graphics Processing Units » (*GPGPU*), en anglais, sont des processeurs manycores.

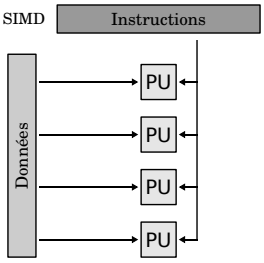


FIGURE 2.1.7 – Paradigme SIMD selon la taxonomie de Flynn.

2.1.6 L'avènement des systèmes massivement parallèles

De nos jours, tous les supercalculateurs sont des systèmes massivement parallèles. Il s'agit de plusieurs ordinateurs appelés « nœuds » que l'on relie entre eux par un réseau d'interconnection. Chacun de ces nœuds dispose de son propre système d'exploitation, généralement Linux sur les supercalculateurs actuels. Sur la figure 2.1.8, nous voyons la transition des systèmes d'exploitation des supercalculateurs d'*Unix* vers *Linux* depuis l'année 2000.

Ces nœuds sont formés de plusieurs microprocesseurs avec leurs mémoires. Certains nœuds disposent même d'accélérateurs tels que les GPGPUs fournissant une très grande capacité de calcul. Nous pouvons voir la topologie d'un nœud composé de microprocesseurs Haswell sur la figure 2.1.9. Ce nœud dispose de deux microprocesseurs différents (« socket » sur la figure) et d'une capacité totale de 128 Go de mémoire RAM. Cette mémoire est répartie sur 4 bancs mémoires différents, 2 par *socket*. Ici chaque *socket*, dispose de 16 cœurs et chacun d'eux possède 2 Hyper-Threads. Les cœurs sont regroupés par 8 et sont plus proches d'un des bancs mémoires et y accèdent donc plus rapidement. Ce couple comprenant plusieurs cœurs associés à son banc mémoire le plus proche est donc un « nœud NUMA ».

Une machine massivement parallèle peut être composée de milliers de nœuds comme celui-ci. Cela exige une programmation spécifique des applications en fonction de la topologie de la machine. Au sein d'un nœud, tous les bancs mémoires sont disponibles pour tous les cœurs. Il faut donc veiller à ce que les cœurs accèdent à leur banc mémoire le plus proche pour avoir de bonnes performances. En inter-nœuds, un programme multi-processus devra faire des communications inter-processus par l'envoi de messages. Nous verrons de quelle façon programmer ce type de machine dans la section 2.2.

2.1.7 Les classements Top500, HPCG et Green500

Les systèmes massivement parallèles font l'objet d'un classement des meilleurs supercalculateurs appelé le « Top500 » [15]. Ce classement est établi tous les 6 mois grâce au test de performance appelé « Linpack » [16]. Ce test résout

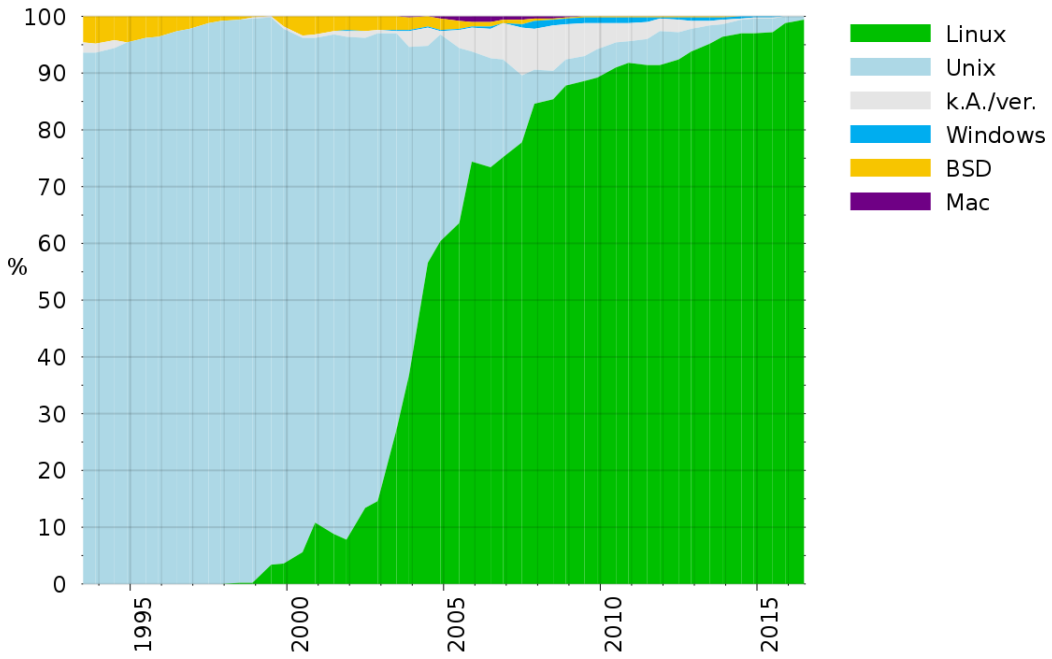


FIGURE 2.1.8 – Transition des systèmes d'exploitation des supercalculateurs d'Unix (en bleu pâle) vers Linux (en vert) (Top500 [15]).

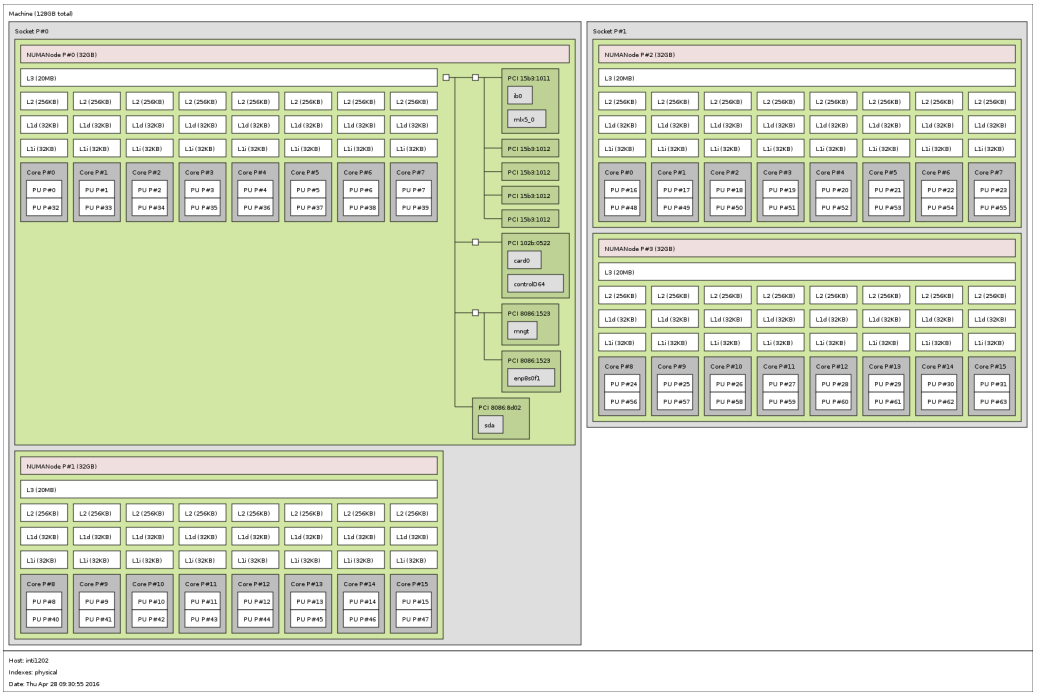


FIGURE 2.1.9 – Topologie d'un nœud Haswell obtenue à l'aide du logiciel hwloc.

Rang	Pays	Nom	Rmax(TFlop/s)
1	US	Summit	122 300,0
2	Chine	Sunway TaihuLight	93 014,6
3	US	Sierra	71 610,0
4	Chine	Tianhe-2A	61 444,5
5	Japon	ABCI	19 880,0
6	Suisse	Piz Daint	19 590,0
7	US	Titan	17 590,0
8	US	Sequoia	17 173,2
9	US	Trinity	14 137,3
10	US	Cori	14 014,7
14	France	Tera-1000-2	12 210,0

TABLE 2.1.1 – Top 10 du classement *Top500* ainsi que le supercalculateur français « Tera-1000-2 » le plus puissant en juin 2018

Rang	Top500	Pays	Nom	Rmax(TFlop/s)	HPCG(TFlop/s)
1	1	US	Summit	122 300,0	2 925,75
2	3	US	Sierra	71 610,0	1 795,67
3	16	Japon	K computer	10 510,0	602,74
4	9	US	Trinity	14 137,3	546,12
5	6	Suisse	Piz Daint	19 590,0	486,40
6	2	Chine	Sunway TaihuLight	93 014,6	480,85
7	12	Japon	Oakforest-PACS	13 554,6	385,48
8	10	US	Cori	14 014,7	355,44
9	14	France	Tera-1000-2	12 210,0	333,76
10	8	US	Sequoia	17 173,2	330,37

TABLE 2.1.2 – Top 10 du classement *HPCG* en juin 2018

un système linéaire dense de n équations à n inconnues. Il permet d'établir le nombre d'opérations flottantes qu'un supercalculateur peut faire par seconde (Flop/s) appelé Rmax. Bien qu'utilisé depuis 1993 pour classer les supercalculateurs au *Top500*, ce test ne reflète pas la réalité de tous les codes de calcul actuels. En effet, certains codes de calcul ne se contentent pas d'effectuer des produits de matrices, opérations se parallélisant très bien. C'est pourquoi un autre test a été proposé. Il s'agit de « HPCG » qui reproduit l'algorithme de gradient conjugué.

La liste des dix premiers supercalculateurs au *Top500*, en juin 2018, est décrite sur la table 2.1.1. Le premier supercalculateur français se place à la 14$^{\text{ième}}$ place du *Top500*. Nous pouvons voir que les États-Unis dominent largement ce classement en plaçant 6 de leurs supercalculateurs dans le Top 10. Concernant le classement *HPCG* (table 2.1.2), les États-Unis dominent toujours ce Top 10. La France rentre dans ce Top 10 en 9$^{\text{ième}}$ position.

Un autre test basé sur la consommation énergétique existe aussi, il s'agit du *Green500*. Celui-ci classe les supercalculateurs du *Top500* en mesurant leur

Rang	Top500	Pays	Nom	Rmax(TFlop/s)	(kW)	(GFlops/watts)
1	359	Japon	Shoubu system B	857,6	47	18,404
2	419	Japon	Suiren2	798,0	47	16,835
3	385	Japon	Sakura	824,7	50	16,657
4	227	US	DGX SaturnV Volta	1 070,0	97	15,113
5	1	US	Summit	122 300,0	8 806	13,889
6	19	Japon	TSUBAME3.0	8 125,0	792	13,704
7	287	Japon	AIST AI Cloud	961,0	76	12,681
8	5	Japon	ABCI	19 880,0	1 649	12,054
9	255	Espagne	MareNostrum P9 CTE	1 018,0	86	11,865
10	171	Japon	RAIDEN GPU subsystem	1 213,0	107	11,363
20	249	France	Romeo	1 022,0	127	8,047
44	14	France	Tera-1000-2	11 965,5	3 178	3,765

TABLE 2.1.3 – Top 10 du classement *Green500* ainsi que les supercalculateurs français « Romeo » et « Tera-1000-2 » en juin 2018.

efficacité énergétique en *GFlops/watt*. En effet, limiter la consommation des supercalculateurs est primordiale pour atteindre l'exascale (1 exaFlop/s : 1 milliard de milliards d'opérations flottantes par seconde). Nous pouvons voir le Top 10 du *Green500* sur la table 2.1.3. Cette fois-ci, le Japon domine ce Top 10 avec 7 supercalculateurs dont les trois premiers. La France n'arrive qu'en 20$^{\text{ième}}$ position avec le supercalculateur *Romeo*. Le supercalculateur français le plus puissant du *Top500* se place 44$^{\text{ième}}$. Il est à noter que le nouveau supercalculateur américain premier du *Top500* et troisième de *HPCG*, se place 5$^{\text{ième}}$ du *Green500*. Cela démontre que les nouvelles machines du *Top500* tendent à consommer moins d'énergie.

La programmation sur de telles architectures n'est pas simple car les développeurs sont amenés à utiliser des architectures complexes comprenant plusieurs niveaux de mémoires. Parfois, ces architectures sont hétérogènes et nécessitent des langages spécifiques. C'est pourquoi l'utilisation de plusieurs modèles de programmation permet la programmation de ces machines.

2.2 Les modèles de programmation

En *HPC*, l'architecture des machines est complexe. Les développeurs utilisent donc plusieurs modèles de programmation pour programmer ces systèmes. Généralement les modèles de programmation sont séparés en deux grandes catégories : les modèles à mémoire partagée pour l'intra-nœud et modèles à mémoire distribuée pour l'inter-nœuds.

2.2.1 Modèle à mémoire partagée

Les nœuds sont souvent composés de plusieurs microprocesseurs avec leurs mémoires respectives formant des nœuds NUMA. Au sein d'un nœud, il est commun d'utiliser un modèle de programmation à mémoire partagée. Ce modèle

de programmation implique d'avoir un accès à une mémoire partagée par toutes les entités voulant communiquer ensemble. Plusieurs techniques existent pour permettre d'avoir le même espace d'adressage telles que les processus multi-threadés, les segments de mémoire partagée et les *Partitioned global address space* (PGAS) [17] où la mémoire est vue comme étant partagée alors qu'elle peut être distribuée.

Pour s'échanger des données, chaque thread lit les données dont il a besoin dans la mémoire partagée. Ce mécanisme nécessite de prendre des précautions lorsque deux threads veulent écrire sur la même case mémoire. Ce problème est connu sous le nom de « race condition » [18]. Il s'agit de savoir quelle est la valeur d'une case mémoire si celle-ci est en mémoire partagée et plusieurs threads écrivent une valeur en même temps. La compilation ne va pas générer d'erreur, mais le code n'aura pas le comportement souhaité si le développeur n'a pas mis en place des mécanismes de synchronisation telles que les régions critiques pour maintenir la cohérence de la mémoire.

Une région critique est une portion de code qui est exécutée par un seul thread à la fois. Ce mécanisme permet de garder la cohérence de la mémoire au sein d'un programme multi-threadé à l'aide d'opérations atomiques, c'est-à-dire d'opérations qui ne peuvent pas être interrompues. Un programme est dit « thread safe » si nous pouvons garantir la bonne exécution du code en maintenant la cohérence de la mémoire.

Il existe plusieurs façons d'utiliser des threads au sein d'un processus. Nous allons détailler les deux plus répandues dans le calcul scientifique.

2.2.1.1 Les threads POSIX

Les threads POSIX [19] sont un sous-standard de la norme POSIX. Celle-ci décrit une interface de programmation permettant la gestion des threads sur les systèmes UNIX. Nous allons en expliquer les principales fonctionnalités indispensables à la bonne compréhension des prochaines sections. Pour cela, nous allons d'abord expliquer ce qu'est un processus UNIX.

Un processus est un programme en cours d'exécution. Sa création est effectuée au sein du système d'exploitation pour encapsuler toutes les ressources que requiert le programme telles que les identifiants (processus, groupe de processus, utilisateur, etc.), les variables d'environnement, le compteur ordinal, les registres, la pile, le tas, les signaux d'action, etc.

Chaque processus est donc mappé dans une zone de mémoire virtuelle. L'unité de gestion de la mémoire ou « Memory Management Unit » (MMU) est chargée de transcrire les adresses virtuelles en adresses physiques. Sur la figure 2.2.1, nous pouvons voir de quelle façon un processus utilise la mémoire. Cette figure est non exhaustive mais aide à la compréhension de l'utilisation de la mémoire par un processus. La zone entre l'adresse 0x00000000 [1]

1. Sur les systèmes récents, cette adresse est choisie aléatoirement par mesure de sécurité.

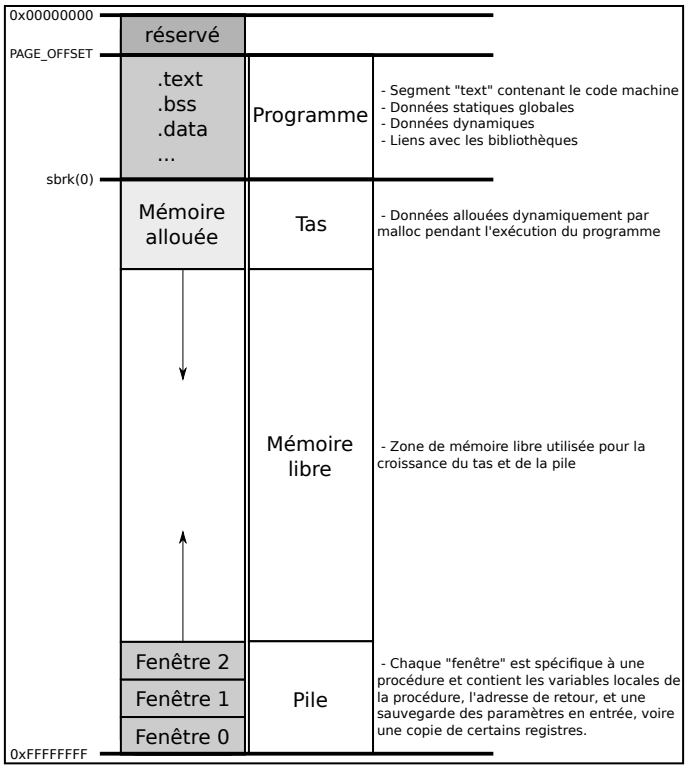


FIGURE 2.2.1 – Utilisation de la mémoire pour un processus

et PAGE_OFFSET est réservée au système d'exploitation. Ensuite, il y a une zone contenant le code de l'application ainsi que les données statiques du programme, puis le tas contient toutes les données allouées par la fonction *malloc*. La pile est composée de *fenêtres* qui représentent les fonctions d'un programme. Elle contient toutes les variables locales d'un programme. Il est à noter qu'un processus contient nécessairement un thread par défaut.

Dans un processus multi-threadé, tous les threads s'exécutent de manière indépendante. Pour cela, les threads possèdent des ressources qui leur sont propres tels que le compteur ordinal, les registres, la pile, certains signaux et leur Thread Local Storage (TLS) : mémoire spécifique et locale à un thread.

La création des threads est donc souvent plus rapide que celle de processus du fait que les threads partagent une partie des ressources du processus. De la même manière, les changements de contexte des threads sont généralement plus rapides que ceux des processus en partie car l'éviction des TLB [2] n'est pas nécessaire. Il est donc très intéressant de les utiliser dans un programme parallèle à mémoire partagée. Néanmoins, il est nécessaire de protéger l'accès à une même variable par des threads différents car ils partagent le même espace d'adressage. Les sémaphores, les verrous ainsi que les variables de condition

2. Translation Lookaside Buffer : C'est une mémoire cache du processeur utilisée par la MMU dans le but d'accélérer la traduction des adresses virtuelles en adresse physiques

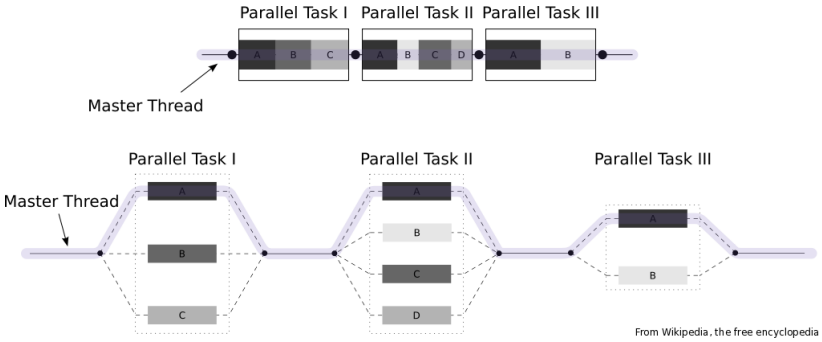


FIGURE 2.2.2 – Le modèle Fork/Join

permettent aux développeurs d'effectuer la synchronisation des threads pour éviter les *race conditions*.

L'utilisation des threads POSIX est très bas-niveau et demande une certaine aisance en programmation. Ces threads sont généralement utilisés dans les bibliothèques de fonctions utilisant des threads. Cela permet d'être compatible avec tous les systèmes POSIX. Plusieurs implémentations de ces threads existent. L'environnement de programmation « Multi-Processor Computing » (MPC) que nous présenterons dans la section 2.4 en possède une.

2.2.1.2 Le standard OpenMP

OpenMP est une interface de programmation pour le calcul parallèle. Son utilisation est relativement simple étant donné que l'API se présente sous la forme de directives préprocesseur ainsi que d'une bibliothèque de fonctions. Plusieurs modèles de parallélisme peuvent être utilisés, dont le modèle fork/join décrit sur la figure 2.2.2. Nous avons un thread maître qui, pour chaque région parallèle, crée plusieurs autres threads pour exécuter des instructions en parallèle.

Nous pouvons aussi utiliser un modèle de parallélisme à base de tâches où celles-ci sont liées entre elles par des dépendances à respecter. Un algorithme d'ordonnancement de tâches propre à OpenMP sera chargé de les exécuter dans le bon ordre.

Depuis OpenMP 4, l'utilisateur peut exprimer le parallélisme de son application sur un accélérateur en utilisant des directives autour des boucles qu'il voudrait paralléliser. OpenMP est une API très utilisée dans le domaine du calcul scientifique en ce qui concerne les codes à mémoire partagée. Cependant, si nous voulons exploiter la puissance des systèmes massivement parallèles, il est indispensable d'utiliser en plus un modèle de programmation à mémoire distribuée.

2.2.2 Modèle à mémoire distribuée : Message Passing Interface

« Message Passing Interface » (MPI) est une norme spécifiant une interface permettant la programmation parallèle par le biais d'envois de messages. Il s'agit du standard pour programmer sur des systèmes à mémoires distribuées tels que les supercalculateurs. Plusieurs implémentations de cette norme existent avec des implémentations open source comme Open MPI [20], MPICH [21], [22], MVAPICH [23], NewMadeleine [24], MPC [25] ou propriétaire comme INTEL-MPI. Ces supports exécutifs permettent d'abstraire les difficultés liées aux échanges de messages.

Un programme développé avec MPI est constitué de plusieurs *processus MPI*. Chaque *processus MPI* est généralement encapsulé dans un *processus système* et possède un identifiant. Néanmoins, les *processus MPI* peuvent aussi être encapsulés dans des threads, c'est le cas pour l'implémentation de MPC (section 2.4). Afin d'éviter la confusion entre les *processus MPI* et les *processus systèmes*, **nous appellerons les *processus MPI* des « tâches MPI »**.

Les tâches MPI peuvent communiquer entre elles par l'envoi et la réception de messages au sein du même communicateur. Un communicateur désigne un ensemble de tâches MPI pouvant communiquer ensemble.

Il existe trois possibilités pour réaliser des communications avec MPI :

— Les communications point-à-point.
— Les communications collectives.
— Les communications unilatérales (*One-sided*).

Néanmoins, les communications *One-sided* étant hors du cadre de la thèse, nous ne détaillerons pas leur fonctionnement.

2.2.2.1 Les communications point-à-point

Une communication point-à-point permet à une tâche MPI d'envoyer un message à une autre tâche MPI. Ce message est composé d'une *enveloppe* qui contient les champs nécessaires aux échanges de messages et d'une partie *données* qui peuvent être :

— Des scalaires : un nombre entier ou flottant.
— Des tableaux : Une suite de scalaires.
— Un type défini par l'utilisateur.

Ces communications peuvent être bloquantes ou non-bloquantes.

Les communications point-à-point bloquantes et non-bloquantes : Selon la section 3.4 de la norme MPI intitulé *Communication Modes*, l'appel à une routine de communication bloquante (e.g. MPI_Send) suspend l'exécution de la tâche MPI appelante jusqu'à ce que le *buffer* contenant le message puisse être réutilisé. C'est-à-dire qu'en sortant d'une routine de com-

munication bloquante, la norme MPI nous garantit que le *buffer* qui contenait le message est réutilisable.

Les communications point-à-point peuvent aussi être non-bloquantes. Elles permettent au calcul de recouvrir les communications. Pour cela, la technique est de séparer l'appel à une communication en deux phases distinctes. La première permet d'initialiser la communication, par exemple à l'aide des routines MPI_Isend et MPI_Irecv. À partir de ce moment, la communication pourra s'effectuer n'importe quand. La seconde consiste à attendre que le *buffer* soit réutilisable à l'aide d'une routine de terminaison telle que MPI_Wait. Il est aussi possible de vérifier si la communication est terminée en effectuant un appel à la routine MPI_Test. Quand ce cas s'avère vérifié, il n'est plus nécessaire de faire appel à MPI_Wait. Du code peut ainsi être inséré entre l'appel d'initialisation et celui de terminaison dans le but de recouvrir la communication par du calcul. La communication sera effectuée en arrière-plan, c'est ce qu'on appelle la « progression » des communications que nous détaillerons dans la section 3.2. Cette progression n'est pas automatique et a fait l'objet de nombreux travaux que nous détaillerons dans le chapitre 3.

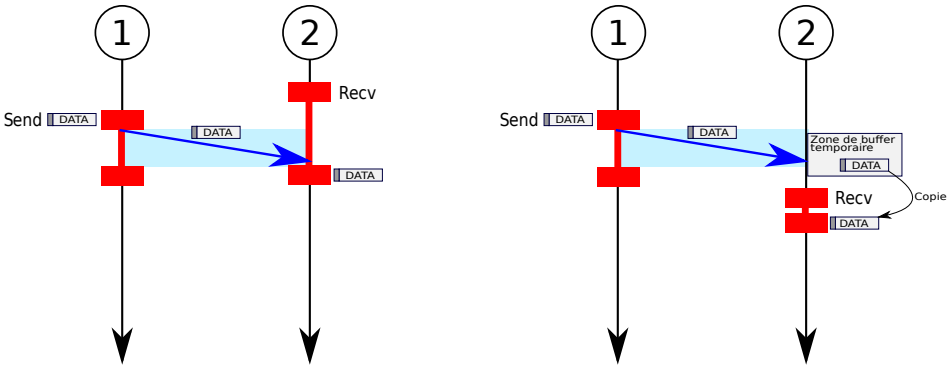
Il existe plusieurs protocoles d'envoi de messages qui ne sont pas liés au caractère bloquant ou non-bloquant des communications.

Le protocole « Eager » : Le protocole *Eager* est généralement utilisé pour les petites tailles de messages, dont le seuil dépend entièrement de l'implémentation MPI. Il consiste à envoyer directement le message à son destinataire sans son autorisation au préalable. C'est une communication asynchrone, c'est-à-dire, sans synchronisation entre les deux tâches MPI s'échangeant un message. De ce fait, ce protocole a l'avantage de réduire le délai d'envoi de message, car la tâche émettrice n'a pas à attendre l'approbation de la tâche réceptrice pour envoyer son message. Il existe deux cas de figure :

— La réception du message est déjà postée : le message est directement copié dans le *buffer* de réception. La figure 2.2.3a décrit ce comportement.
— La réception du message n'est pas encore postée : le message est copié dans un *buffer* temporaire pour que la tâche MPI envoyant le message puisse être débloquée. Lorsque le Recv est posté, le *buffer* temporaire est copié dans le *buffer* de réception. La figure 2.2.3b décrit ce comportement.

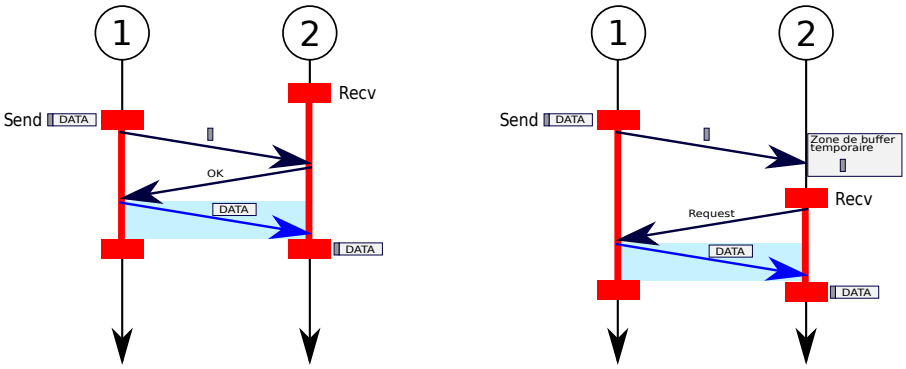
Néanmoins, ce protocole a besoin de *buffers* temporaires pour stocker les messages. Les temps de transferts pour effectuer les copies des *buffers* deviennent trop coûteux lorsque la taille des *buffers* est grande. De plus, l'utilisation du CPU par la tâche MPI réceptrice pour effectuer toutes les copies de buffers peut prendre du temps.

Le protocole « Rendez-vous » : Le protocole *Rendez-vous* est généralement utilisé pour les messages plus volumineux. Il consiste à établir une synchronisation entre les deux tâches MPI pour l'envoi et la réception du



(a) Recv posté avant le Send (b) Recv posté après le Send

FIGURE 2.2.3 – L'envoi d'un message avec le protocole « Eager »



(a) Recv posté avant le Send (b) Recv posté après le Send

FIGURE 2.2.4 – L'envoi d'un message avec le protocole « Rendez-vous »

message. Ce protocole permet de ne pas faire de copies des messages dans des *buffers* temporaires, ce qui serait désastreux pour des messages volumineux car en plus d'utiliser plus de mémoire, les temps de transfert seraient très longs. En contrepartie, un délai supplémentaire sera subi par la tâche MPI émettrice dû à l'établissement de la connexion entre les deux tâches MPI. Comme pour le protocole « Eager », il existe aussi deux cas de figure :

— La réception du message est déjà postée : la tâche MPI émettrice du message demande un rendez-vous à la tâche MPI qui doit réceptionner le message. Lors de la réception de cette demande, la tâche MPI accepte la demande de rendez-vous et le message est envoyé par la tâche MPI émettrice. La figure 2.2.4a décrit ce comportement.

— La réception du message n'est pas encore postée : la tâche MPI émettrice du message demande un rendez-vous à la tâche MPI qui doit réceptionner le message. Cette demande est stockée dans un *buffer* temporaire. Lorsque le Recv est posté, une notification est envoyée à la tâche MPI émettrice et le message est envoyé. La figure 2.2.4b décrit ce comportement.

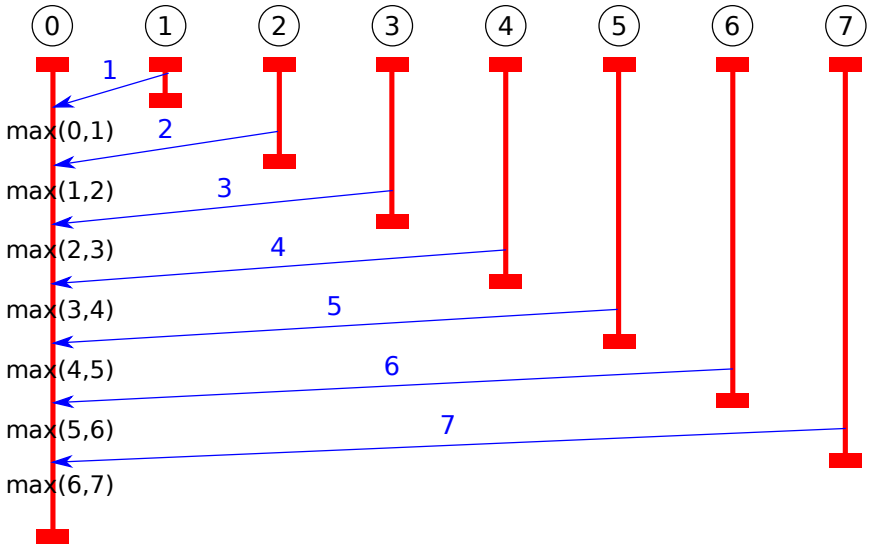


FIGURE 2.2.5 – Illustration de la collective MPI_Reduce calculant la valeur maximale d'un tableau distribué sur 8 tâches MPI dans lequel chaque tâche MPI possède une valeur du tableau correspondant à son rang. Les flèches bleues correspondent aux envois de messages.

2.2.2.2 Les communications collectives

Les communications collectives permettent à toutes les tâches MPI d'un même communicateur de participer à une même communication. Par exemple, la routine MPI_Reduce permet d'effectuer une réduction à partir de toutes les tâches MPI d'un même communicateur et d'envoyer le résultat à une tâche MPI « racine ». Sur la figure 2.2.5, nous pouvons voir le déroulement d'une implémentation linéaire d'une réduction avec 8 tâches MPI qui calculent la valeur maximale d'un tableau. Ce tableau est distribué sur 8 tâches MPI dans lequel chaque tâche MPI possède une valeur du tableau correspondant à son rang. Les flèches bleues correspondent aux envois de messages. Ici, chaque tâche MPI envoie sa valeur à la tâche MPI 0 qui est la « racine ». Celle-ci calcule le maximum de son maximum courant et des valeurs qu'elle reçoit. Dans cet algorithme, la tâche MPI racine exécute beaucoup plus d'instruction que les autres tâches MPI. Cependant, il existe plusieurs algorithmes de communication pour exécuter la même opération collective.

Sur la figure 2.2.6, nous pouvons voir le déroulement de la même opération collective que précédemment, mais cette fois-ci avec des communications en arbre binomial : chaque tâche MPI calcule le maximum entre les valeurs reçues et celle qu'elle possède, puis l'envoie à son destinataire. Cet algorithme répartit la charge de travail sur plus de tâche MPI. Parfois, un algorithme est meilleur jusqu'à un certain seuil dépendant du nombre de nœuds, de la topologie, et de la taille des *données*, ensuite un autre algorithme devient plus performant.

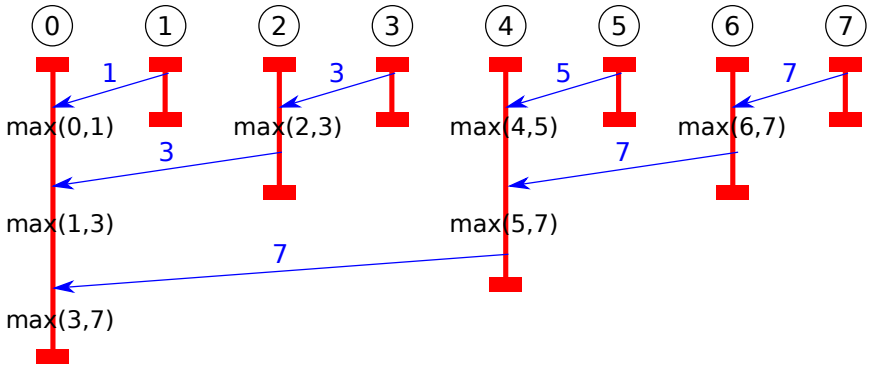


FIGURE 2.2.6 – Illustration de la collective MPI_Reduce avec un arbre de communication binomial calculant la valeur maximale d'un tableau distribué sur 8 tâches MPI dans lequel chaque tâche MPI possède une valeur du tableau correspondant à son rang. Les flèches bleues correspondent aux envois de messages.

Liste des opérations collectives bloquantes : Il existe 17 opérations collectives différentes que nous pouvons classer en 4 types d'opérations :

1. **Tous-vers-Tous** : Toutes les tâches MPI d'un même communicateur contribuent au résultat et toutes le reçoivent (MPI_Barrier, MPI_Allgather, MPI_Allgatherv, MPI_Allreduce, MPI_Reduce_scatter, MPI_Alltoall, MPI_Reduce_scatter_block, MPI_Alltoallv, MPI_Alltoallw).
2. **Tous-vers-Un** : Toutes les tâches MPI d'un même communicateur contribuent au résultat mais une seule (la racine) reçoit le résultat (MPI_Gather, MPI_Gatherv, MPI_Reduce).
3. **Un-vers-Tous** : Une seule tâche MPI (la racine) contribue au résultat mais toutes les tâches MPI du même communicateur reçoivent le résultat (MPI_Bcast, MPI_Scatter, MPI_Scatterv)
4. **Autre** : Opération Collective qui ne rentre dans aucune des catégories précédentes (MPI_Scan, MPI_Exscan)

Toutes ces collectives ont leur équivalent en version non-bloquantes.

Les communications collectives non-bloquantes : Depuis la norme MPI 3.0 [1] datant de septembre 2012, les communications collectives peuvent aussi être non-bloquantes. Comme pour les communications point-à-point non-bloquantes, il s'agit de séparer l'appel à une communication collective en deux phases distinctes. Cela permet d'insérer du code entre ces appels dans le but de recouvrir le temps de communications par du calcul.

Nous expliquerons en détail les problématiques liées à la progression et au recouvrement des communications non-bloquantes dans le chapitre 3. Avant cela, nous allons présenter différents modèles d'exécution.

2.3 Modèle d'exécution

Un modèle d'exécution est un modèle qui décrit la façon dont un programme s'exécute. Cela est différent d'un modèle de programmation qui décrit comment un programme doit être implémenté. Nous allons d'abord présenter le fonctionnement de l'ordonnanceur puis présenter différentes bibliothèques de threads implémentant plusieurs modèles d'exécution pour un même modèle de programmation : la programmation multi-threadée.

2.3.1 L'ordonnancement

L'ordonnanceur est l'entité décidant quel thread doit s'exécuter parmi tous les threads d'un système. En effet, il y a généralement plus de threads que de ressources. Il faut donc choisir quels threads peuvent s'exécuter sur les ressources à un moment donné. L'ordonnanceur est un ensemble de procédures permettant d'attribuer les ressources (e.g. les cœurs) selon un algorithme que l'on appelle « l'algorithme d'ordonnancement ». Selon A. S. Tanenbaum et H. Bos [26], ces algorithmes peuvent être divisés en deux catégories :

— Les algorithmes préemptifs. Ce sont des algorithmes qui permettent des interruptions matérielles afin de changer de thread à exécuter.
— Les algorithmes non-préemptifs. Ce sont des algorithmes qui ne permettent pas les interruptions matérielles. Lorsqu'un thread est exécuté, il ne rendra pas la main tant qu'il n'aura pas fini son travail ou qu'il rende la main pour être coopératif.

L'algorithme par défaut sur les systèmes Linux depuis la version 2.6.23 en 2007 est l'algorithme préemptif « Completly Fair Scheduler » (CFS). Il repose sur une structure de donnée particulière qui garantit d'attribuer du temps sur le CPU de manière équitable à tous les threads. Ainsi, chaque thread s'exécutera plus ou moins longtemps suivant son temps d'exécution actuel ainsi que les temps d'exécution des autres threads.

Dans un contexte HPC, cet algorithme n'est pas forcément le plus approprié. En effet, sur un supercalculateur nous aimerions maximiser l'exécution des threads effectuant du calcul ou bien donner la priorité à des threads de progression pendant un laps de temps. C'est pourquoi il existe des techniques permettant d'avoir le contrôle sur l'ordonnancement sans avoir à changer l'algorithme d'ordonnancement du noyau Linux qui demande d'avoir les privilèges d'administrateur. Ces techniques reposent sur différentes implémentations de bibliothèques de threads.

2.3.2 Les bibliothèques de threads

Les bibliothèques de threads implémentent le modèle d'exécution permettant la programmation multi-threadée. Il existe différents types de bibliothèques de

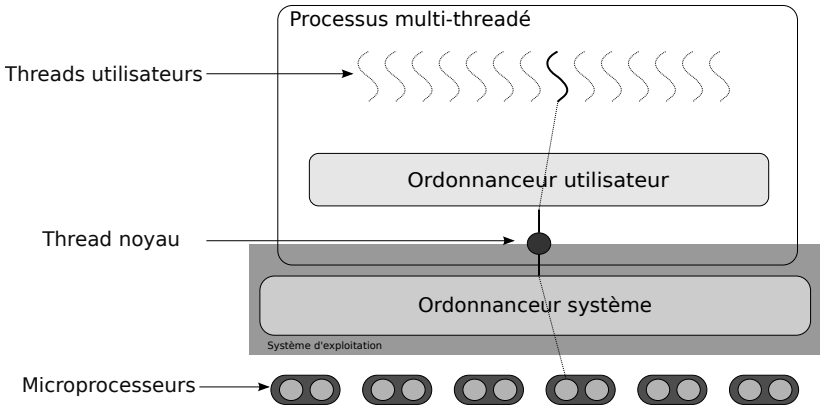


FIGURE 2.3.1 – Bibliothèque de thread utilisateur

threads ayant chacun leurs avantages et leurs inconvénients que nous allons détailler.

2.3.2.1 Bibliothèque de thread utilisateur

Les bibliothèques de thread utilisateur sont entièrement en espace utilisateur. Le noyau n'a pas connaissance de l'existence de ces threads. En effet, les structures de données permettant la gestion des threads (table des threads) sont sauvegardées dans la mémoire du processus. La figure 2.3.1 illustre leur fonctionnement. Le thread utilisateur à exécuter est choisi par un ordonnanceur en espace utilisateur. Cela permet l'exécution de plusieurs threads par processus de façon non simultanée.

L'avantage d'une bibliothèque comme celle-ci est que les changements de contexte entre les threads sont très rapides. En effet, la gestion des structures propres aux threads étant locale au processus, le changement de contexte se fait avec des fonctions en espace utilisateur.

L'autre avantage est de permettre d'avoir un algorithme d'ordonnancement propre à chaque processus. En effet, un processus A pourra avoir un algorithme d'ordonnancement différent d'un processus B selon ses propres besoins.

Malgré les avantages offerts par les bibliothèques de thread utilisateur, il subsiste des inconvénients à ne pas négliger. Les appels systèmes bloquants provoquent une attente des threads pourtant prêts à être exécutés. Les appels systèmes sont aussi compliqués à gérer, notamment les appels liés aux défauts de pages. En effet, lorsqu'un thread veut accéder à une donnée qui n'est pas en mémoire, il se produit un défaut de page. Le système n'ayant pas connaissance de l'existence des threads, il doit bloquer tout le processus et donc tous les threads pour traiter le défaut de page. De plus, les bibliothèques de thread utilisateur ne sont pas adaptées aux microprocesseurs multi-cœurs car un seul thread utilisateur est exécuté par processus.

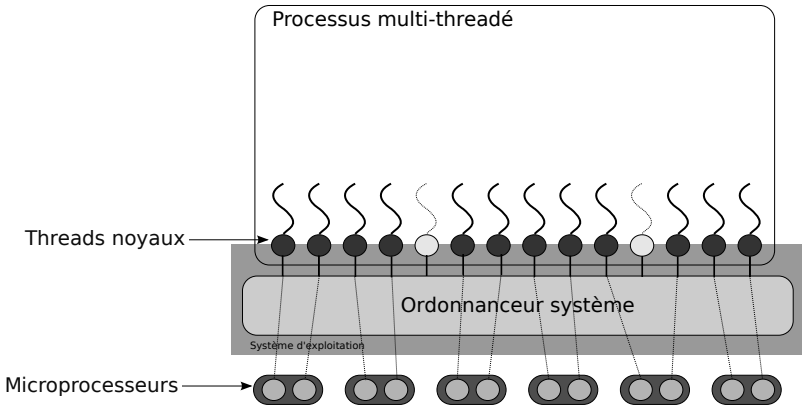


FIGURE 2.3.2 – Bibliothèque de thread système

2.3.2.2 Bibliothèque de thread système

Les bibliothèques de threads système sont entièrement implémentées dans le noyau du système d'exploitation. Les threads créés sont des *threads noyaux*, c'est-à-dire que les structures de données indispensables à leur fonctionnement sont stockées dans le noyau et gérés par le système d'exploitation. La figure 2.3.2 en illustre le fonctionnement.

L'avantage de cette implémentation est que comme c'est le noyau qui gère tous les threads, lorsqu'un thread est bloqué, l'ordonnanceur du système peut choisir un autre thread à exécuter provenant ou non du même processus. De plus, un processus multi-threadé pourra s'exécuter sur plusieurs cœurs au même moment car chaque thread du même processus peut avoir accès à un cœur.

En revanche, l'inconvénient est que les coûts de création, de destruction et de changements de contexte des threads sont élevés, dus à la gestion des structures de données liées aux *threads noyaux* dans le processus multi-threadés.

2.3.2.3 Bibliothèque thread mixte

Les bibliothèques de threads mixtes consistent à combiner les avantages des threads utilisateurs avec ceux des *threads noyaux*. Cela consiste à créer des *threads noyaux* et d'ordonnancer des threads utilisateurs sur ces *threads noyaux* comme l'illustre la figure 2.3.3. Ce modèle est très flexible car le développeur peut choisir le nombre de *threads noyaux* et de threads utilisateurs. Les threads utilisateurs sont ordonnancés sur les threads noyaux avec les avantages que cela apporte (création de thread rapide, changement de contexte rapide, etc.). Un processus multi-threadé pourra s'exécuter sur plusieurs cœurs en même temps. Ce qui n'était pas possible avec les bibliothèques de threads utilisateurs classiques.

Dans la prochaine partie, nous allons présenter le framework MPC qui repose sur l'utilisation d'une bibliothèque de thread mixte.

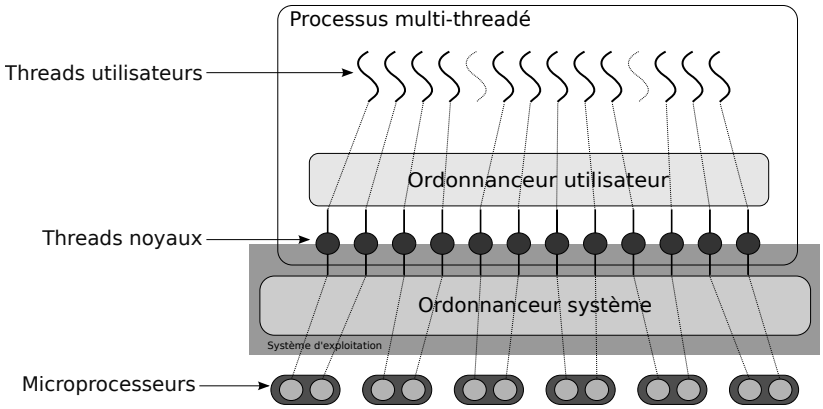


FIGURE 2.3.3 – Bibliothèque de thread mixte

2.4 Le framework MPC

MPC (MultiProcessor Computing) [25] est un environnement de programmation dédié à l'Informatique Haute Performance unifiant plusieurs modèles de programmation. Cet environnement de programmation est développé par le CEA. C'est également un logiciel libre sous licence CeCILL-C [27].

2.4.1 Caractéristiques

Comme nous l'avons vu dans les sections 2.2.1.2 et 2.2.2, OpenMP et MPI sont des standards de programmation parallèle sur supercalculateur. MPC est un environnement de programmation pensé pour la programmation hybride consistant à mélanger plusieurs modèles de programmations rencontrés dans les codes scientifiques. Il possède :

- Une bibliothèque de threads mixte avec une implémentation Pthread.
- Une implémentation MPI 3.1 à base de thread incluant l'intégration de la libNBC pour la gestion des collectives non-bloquantes.
- Une implémentation d'OpenMP 3.1.
- Un allocateur mémoire topologique.
- Des outils de débogage et d'aide au développeur.

2.4.2 La bibliothèque de threads mixte

MPC possède un ordonnanceur unifié. Tous les threads créés par les différentes bibliothèques connues sont vus comme des threads utilisateurs de la bibliothèque de threads interne à MPC.

Sur la figure 2.4.1, nous pouvons observer le fonctionnement global de l'ordonnanceur de MPC. Les threads MPC sont ordonnancés sur des threads systèmes qui représentent des processeurs virtuels.

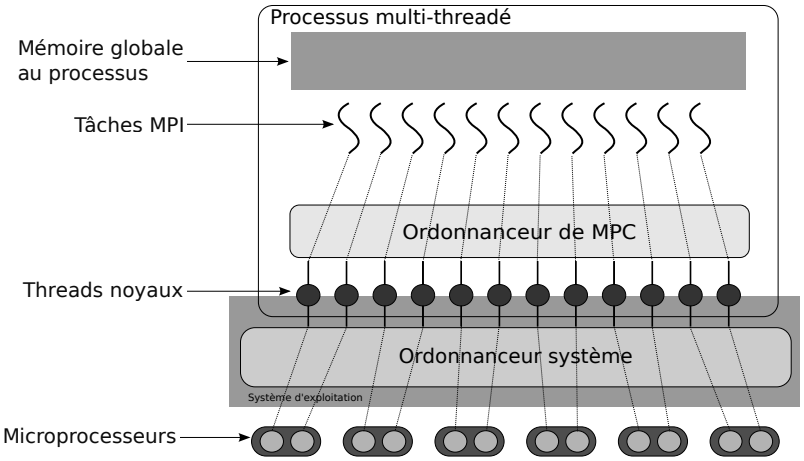


FIGURE 2.4.1 – Implémentation MPI basée sur les threads (MPC)

2.4.3 L'implémentation MPI

MPC intègre une implémentation MPI compatible avec la norme 3.1. La particularité de celle-ci est de proposer deux implémentations MPI. La première est une implémentation MPI à base de processus et la seconde est une implémentation basée sur les threads (figure 2.4.1). Les tâches MPI ne sont pas encapsulées dans des processus UNIX comme dans d'autres implémentations MPI (figure 2.4.2) telles que MPICH, Open MPI, INTEL-MPI et d'autres, mais dans des threads utilisateurs proposés par la bibliothèque de threads mixte de MPC. Cette façon de procéder possède de nombreux avantages comme le fait de pouvoir se partager des données volumineuses en mémoire. Cela peut réduire considérablement l'utilisation de la mémoire d'une application lorsque de grandes quantités de données sont uniquement utilisées en lecture [28]. La gestion des variables globales s'effectue lors de la compilation. En effet, la privatisation automatique de celles-ci est effectuée lors de la compilation avec l'option *mpc-privatize* disponible dans les compilateurs gcc et icc.

2.4.4 L'implémentation OpenMP

MPC est un environnement de programmation qui a pour but d'unifier les supports exécutifs afin de programmer avec plusieurs modèles de programmation de façon efficace. MPC possède sa propre implémentation OpenMP. Cette implémentation permet la programmation MPI+OpenMP au sein du même support d'exécution. Les tâches MPI et les threads OpenMP sont des threads utilisateurs gérés par le même ordonnanceur dans le même processus. De cette façon, il est possible d'optimiser le placement et l'ordonnancement de tous les threads gérés par MPC à l'aide d'algorithme prenant en compte les différents types de threads.

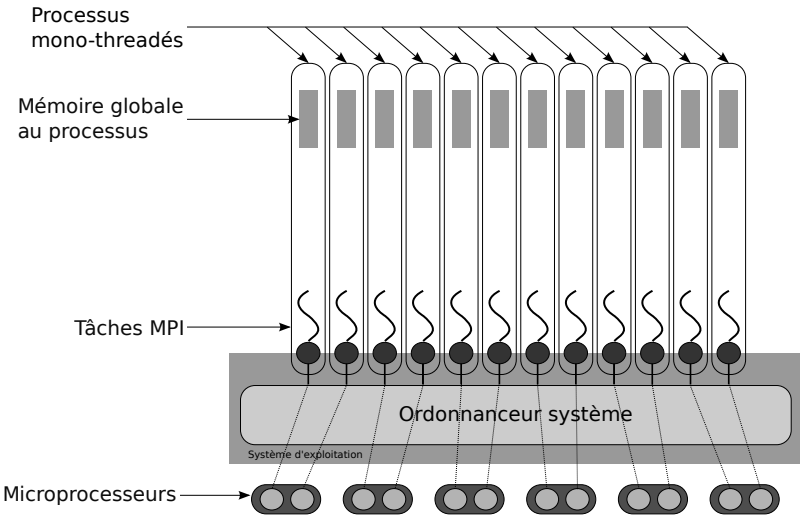


FIGURE 2.4.2 – Implémentation MPI basée sur les processus

2.5 Conclusion

Ce chapitre recense l'essentiel à savoir sur le HPC dans le cadre de cette thèse. Nous avons décrit comment l'évolution de l'architecture des microprocesseurs a conduit aux architectures actuelles. Ensuite, les modèles de programmation à mémoire partagée et à mémoire distribuée ont été expliqués. Ce chapitre termine sur les différents modèles d'exécution ainsi que la présentation de MPC sur lequel nous nous sommes appuyés pour implémenter la plupart de nos algorithmes développés pendant cette thèse.

Dans le chapitre suivant, nous nous concentrerons sur la problématique de cette thèse, à savoir, la progression des communications collectives MPI non-bloquantes. Nous verrons en quoi la progression des communications est un problème difficile en détaillant l'état de l'art répondant à ces problématiques. Enfin nous dégagerons la problématique de la thèse.

Chapitre 3

État de l'art et problématique

Sommaire

Les spécifications MPI proposent les routines non-bloquantes, décrites dans les sous-sections 2.2.2.1 et 2.2.2.2, permettant le recouvrement des communications par du calcul. Dès lors, les développeurs d'applications tentent d'obtenir un bon recouvrement dans les applications. Quelques questions se posent alors :

— Comment faire la progression des communications en arrière-plan ?
— Comment obtenir un bon recouvrement des communications par du calcul ?
— Comment limiter l'impact des communications sur le calcul et vice-versa ?

Ce chapitre pose les bases de la problématique de cette thèse en se reposant sur l'état de l'art. Nous verrons d'abord quels sont les problèmes liés à la progression des communications point-à-point non-bloquantes, puis ceux liés à la progression des communications collectives non-bloquantes. Ensuite, nous verrons comment fonctionne la progression des communications dans MPC. Enfin, nous détaillerons la problématique de cette thèse : le recouvrement des collectives MPI non-bloquantes.

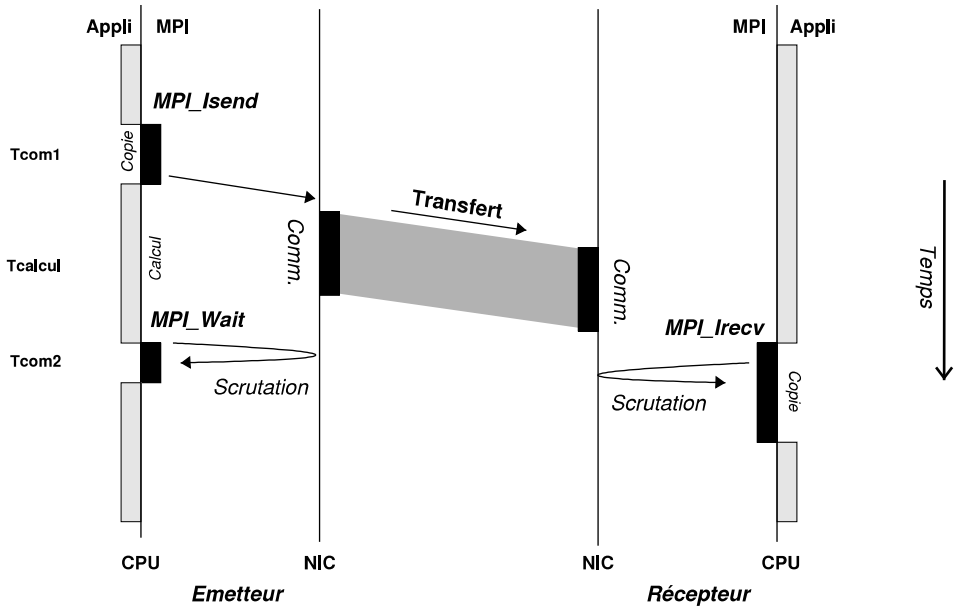


FIGURE 3.1.1 – Illustration du recouvrement d'une communication par du calcul. (Source de la figure : [29])

3.1 Liens entre recouvrement, progression et ressources matérielles

Les performances de certaines applications scientifiques dépendent fortement du temps des communications MPI. Afin de masquer ce temps de communications, les développeurs utilisent les opérations non-bloquantes. Du code de calcul peut ainsi être inséré entre les deux phases des communications non-bloquantes (e.g. MPI_Isend, MPI_Wait).

Afin de bénéficier d'un bon recouvrement, il est nécessaire d'exécuter le code de calcul et les communications simultanément. La figure 3.1.1 représente une communication non-bloquante entre deux tâches MPI avec le protocole « Eager » et illustre ce comportement. Nous pouvons voir qu'il est nécessaire d'avoir des mécanismes de progression des communications pour obtenir du recouvrement. Lors de l'envoi du message, l'émetteur utilise le CPU pour copier son message sur la carte réseau (NIC) qui effectue le transfert du message vers la carte réseau du récepteur. Pendant ce temps, l'émetteur et le récepteur recouvrent le temps de communication par du calcul. Le récepteur effectue une scrutation du réseau pour savoir si un message est arrivé. Lors de la réception du message, celui-ci doit le copier dans son *buffer*.

Tous ces mécanismes utilisent des ressources (CPU, NIC) pour effectuer les copies de *buffers* et la scrutation du réseau pour savoir si la communication a été effectuée. C'est pourquoi, il y a un lien très fort entre le recouvrement, la progression et les ressources matérielles.

Bien que la norme MPI ait été conçue pour envoyer des messages à travers le réseau, avec l'arrivée des microprocesseurs multi-cœurs et manycore, certaines tâches MPI sont placées sur le même nœud. Dans ce cas, les communications sont généralement faites par des copies de *buffers* en mémoire sans passer par le réseau bien qu'elles puissent être réalisées en passant par le réseau.

3.2 Le recouvrement des communications point-à-point

Les communications MPI point-à-point non-bloquantes sont utilisées pour recouvrir les communications par du calcul en arrière-plan. Plusieurs techniques existent pour obtenir un bon recouvrement : la progression *matérielle* et la progression *logicielle*.

3.2.1 La progression *Matérielle*

La progression *Matérielle* consiste à avoir du matériel spécialisé dans la progression des communications. Par exemple, les cartes réseaux permettent de faire des *Direct Memory Access* (DMA). Il s'agit de cartes permettant de copier directement le *buffer* à envoyer dans le *buffer* de réception. Cependant, le protocole de rendez-vous doit encore se faire de façon logicielle. Il y a aussi les microcontrôleurs intégrés à certaines cartes réseaux, pour réaliser la progression des communications. Le recouvrement est possible étant donné que le calcul s'exécutera sur le CPU tandis que la progression des communications s'exécutera sur le matériel spécifique.

L'utilisation de la technologie matérielle appelée « Remote Direct Memory Access » (RDMA) [30], [31] permettant à une carte réseau de copier directement un message sur la mémoire distante sans l'intervention du processeur est aussi une technique utilisée. Cela permet de ne pas utiliser le CPU distant et donc de recouvrir la communication avec du calcul. En revanche, le récepteur n'est pas notifié de l'arrivée d'un message.

3.2.2 La progression *Logicielle*

La progression *logicielle* consiste à effectuer la progression des communications sans l'aide de matériel spécifique à cette fonction. Il existe plusieurs techniques pour faire progresser les communications point-à-point de façon *logicielle*. Il s'agit d'effectuer la progression par le biais d'un appel de fonction explicite (MPI_Test) ou bien d'utiliser d'autres mécanismes reposant sur des threads de progression.

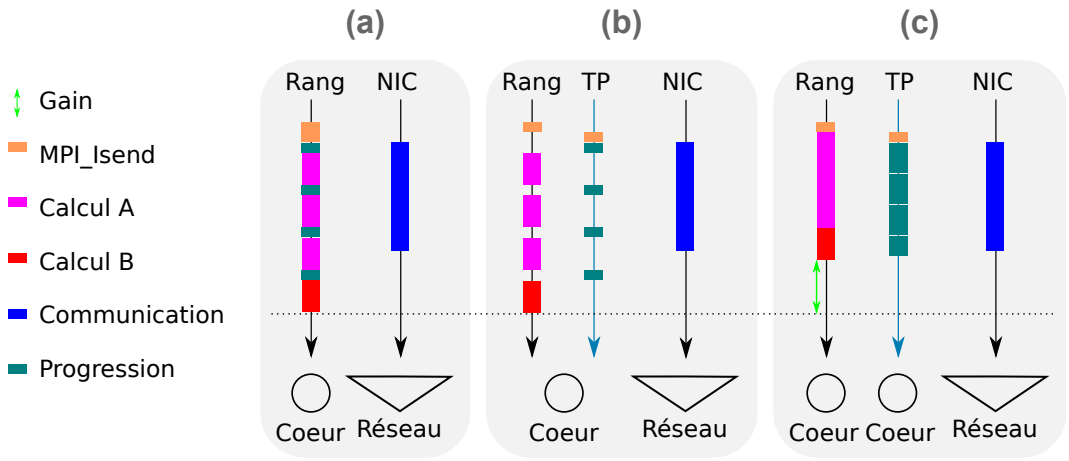


FIGURE 3.2.1 – Progression des communications non-bloquantes. (a) à l'aide d'appels à MPI_Test, (b) à l'aide d'un thread de progression, (c) à l'aide de thread de progression sur un cœur dédié.

3.2.2.1 La progression manuelle

La progression manuelle des communications consiste à utiliser une fonction explicite pour faire progresser les communications. Il s'agit de l'appel à la fonction MPI_Test (figure 3.2.1(a)). Grâce à cet appel, la bibliothèque MPI va scruter si la communication a été effectuée ou si le *buffer* a été copié dans un *buffer* temporaire interne à la bibliothèque MPI. L'idée est de tester si la communication est terminée. Les implémentations MPI profitent aussi du fait d'avoir la main pour faire progresser l'ensemble des communications en attente lorsqu'une fonction MPI est appelée. Cependant, il n'est pas toujours possible de faire des appels à MPI_Test. En effet, lorsque le calcul utilisé pour recouvrir les communications provient d'une bibliothèque externe telle que la MKL, dont le code source n'est pas disponible, il n'est pas possible d'insérer des fonctions pour faire progresser les communications. Il n'est pas non plus envisageable de modifier toutes les bibliothèques de fonction pour y insérer des appels à MPI_Test. De plus, cette technique nécessite que les développeurs d'applications sachent où placer ces appels de fonctions dans le but de l'appeler le moins possible inutilement.

3.2.2.2 Les threads de progression

La progression des communications peut aussi s'effectuer avec des threads ou des processus de progression. T. HOEFLER et A. LUMSDAINE [32] démontrent l'utilité des threads de progression pour effectuer les communications non-bloquantes. Pour cela, un thread de progression est créé pour effectuer la progression des communications (figure 3.2.1(b)). Cette technique permet aux développeurs de ne pas avoir à faire de MPI_Test dans le code. Ainsi la tâche

MPI appelante peut recouvrir les communications avec du calcul sans que le développeur d'application n'ait à se soucier de faire de la progression manuelle. Néanmoins, il reste le problème du placement des threads de progression ainsi que de la compétition entre les threads de progression et/ou les tâches MPI pour les ressources.

P. Lai, P. Balaji, R. Thakur et D. Panda [33] proposent de faire la progression des communications en allouant un sous-ensemble de cœurs d'un microprocesseur multi-cœurs ou manycores à un processus de progression. L'idée est de faire progresser les communications MPI dans un sous-ensemble de cœurs dédiés par un processus et non par un thread de progression. Cette technique évite d'avoir la compétition des threads de progression avec les threads de calcul car nous avons des cœurs dédiés à la progression.

Cependant, l'utilisation d'un thread ou d'un processus de progression n'est pas toujours synonyme de performance. En effet, si un cœur n'est pas attribué au thread de progression, la progression des communications ne sera pas efficace car comme nous l'avons vu dans la section 3.1, la progression des communications nécessite d'avoir des ressources pour effectuer les copies de *buffers*. Lorsque des ressources sont attribuées pour les threads de progression, les communications peuvent être recouvertes. Ce phénomène est illustré sur la figure 3.2.1(c).

3.2.2.3 Ordonnancement opportuniste des threads de progression

D'autres travaux reposant sur des techniques logicielles permettent une meilleure progression des communications. A. Denis et F. Trahay présentent PIOMan [29], [34] qui utilise le modèle de tâches pour exécuter les communications. Le moteur interne de PIOMan ordonnance les tâches sur les cœurs des nœuds de façon opportuniste, c'est-à-dire en essayant de ne pas perturber le calcul de l'application. Cependant, aucune implémentation pour les collectives MPI non-bloquantes n'existe.

D'autres travaux sur l'ordonnancement opportuniste permettent d'améliorer le recouvrement des communications par du calcul. M. Si et al. [35] présentent MT-MPI. Il s'agit d'une implémentation MPI se concentrant sur les architectures manycores. Le but est de cibler les applications hybrides MPI+OpenMP utilisant les modes de protection FUNNELED ou le mode SERIALIZED. MPI_FUNNELED autorise qu'un processus MPI soit multi-threadé mais seul le thread maître peut faire des appels MPI. MPI_SERIALIZED autorise qu'un processus MPI soit multi-threadé et que n'importe quel thread puisse faire un appel MPI tant qu'il y en a qu'un seul à la fois. Dans ces cas, lorsqu'une communication MPI est effectuée, les threads OpenMP n'utilisent pas les ressources qui leurs sont attribuées. MT-MPI permet d'ordonnancer de manière opportuniste des threads aidant à faire progresser les communications MPI sur ces cœurs inutilisés. De la même manière, M. Sergent et al. utilisent Bull Hybrid Communication Optimizer (BHCO) [36] pour faire la progression des

communications sur les cœurs qui ne sont pas utilisés lorsqu'un thread d'une région parallèle OpenMP finit avant les autres.

3.3 Le recouvrement des communications collectives

Les communications collectives MPI non-bloquantes sont plus difficiles à faire progresser en arrière-plan que les communications point-à-point. En effet, elles requièrent non seulement de faire progresser le transfert des données comme pour les communications point-à-point mais aussi de faire progresser l'algorithme de collective en lui-même. De plus, si l'algorithme de progression n'est pas ordonnancé au bon moment, il est possible d'accumuler du retard et de voir le retard se propager de nœud en nœud. Cela rend la progression des communications collectives beaucoup plus complexe.

3.3.1 La progression *Matérielle*

Comme pour les communications point-à-point, les communications collectives MPI bénéficient aussi de travaux portant sur la progression matérielle des communications.

Des travaux spécifiques tels que l'utilisation de la progression assistée par le matériel sur Blue Gene ont été effectués. G. Almasi et al. [37] présentent des optimisations des collectives non-bloquantes mais qui sont liées à leur machine Blue Gene. En effet, les algorithmes proposés reposent sur la présence de microcontrôleurs dédiés à la progression des communications. Cela pose des problèmes de flexibilité pour choisir un algorithme en fonction de la taille des données et du nombre de tâches MPI impliquées.

W. Yu et al. [38] proposent un algorithme permettant à une barrière de s'exécuter sur une carte réseau utilisant Quadrics [39] ou Myrinet [40]. Cependant, les auteurs n'ont pas proposé d'algorithmes pour d'autres collectives.

S. Derradji et al. [41] proposent l'architecture d'inter-connection BXI. Il s'agit d'une carte réseau utilisant l'API Portals 4 [42] permettant la programmation d'algorithmes de collective pouvant être exécutés entièrement sur la carte. De cette façon, les communications et le calcul n'utilisant pas les mêmes ressources peuvent progresser indépendamment.

Néanmoins, les techniques reposant sur le matériel sont spécifiques aux machines utilisant ce matériel.

3.3.2 La progression *Logicielle*

L'utilisation de la progression *Logicielle* est plus flexible et cible toutes les architectures de machines.

3.3.2.1 Module noyau

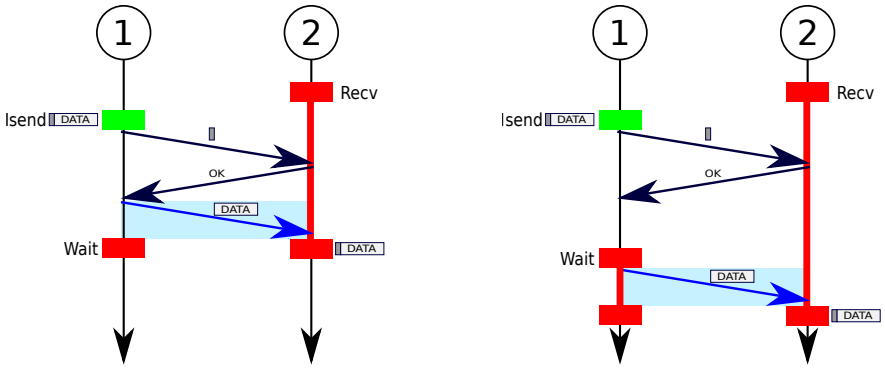
La plupart des implémentations MPI sont basées sur des processus Unix, les différents rangs MPI ne partagent donc pas le même espace d'adressage. Afin de s'envoyer des messages sans passer par le réseau, la progression des communications collectives en intra-noeud s'effectue généralement à l'aide de copies de *buffers* en mémoire. Pour cela, les implémentations MPI utilisent généralement les segments de mémoire partagée. Il s'agit d'une zone mémoire accessible à plusieurs processus et dans laquelle le processus voulant envoyer un message copie le *buffer* dans cette zone et le processus destinataire fera une copie depuis cette zone vers son *buffer*. Il est donc nécessaire d'effectuer deux copies du même *buffer* pour procéder à un envoi de message.

Pour effectuer une seule copie au lieu de deux copies de *buffer*, T. Ma et al. [43] proposent des algorithmes de collectives en mémoire partagée reposant sur le module noyau knem [44]. Le noyau ayant accès à la mémoire de tous les processus, une seule copie de *buffer* est nécessaire pour effectuer les copies mémoire nécessaires à la progression des algorithmes en mémoire partagée. Néanmoins, les auteurs ont seulement traité les performances obtenues sur les opérations collectives bloquantes et pas la progression des collectives non-bloquantes.

3.3.2.2 Une implémentation des collectives non-bloquantes : LibNBC

Avant la norme MPI 3.0 [1], seules les communications point-à-point pouvaient être non-bloquantes. Pour pallier ce problème, la libNBC [45] a été créée. C'est l'implémentation de référence des collectives non-bloquantes. Elle utilise un thread de progression, avec certaines améliorations [46] pour améliorer le recouvrement sur InfiniBand.

Dans cette implémentation, une collective MPI non-bloquante est décomposée en plusieurs appels aux opérations point-à-point non-bloquantes effectuant l'algorithme de collective. Quand une collective MPI non-bloquante est appelée, chaque tâche MPI crée un *schedule* contenant les requêtes des opérations point-à-point non-bloquantes à effectuer pour faire sa partie de l'algorithme de communication collective. Ensuite, chaque tâche MPI attache le *schedule* à son thread de progression associé. Ainsi, le thread de progression prendra en charge les communications décrites dans le *schedule* pendant que la tâche MPI continuera d'exécuter du calcul. Néanmoins, il reste le problème du placement des threads de progression ainsi que de la compétition entre les threads de progression et/ou les tâches MPI pour les ressources.



(a) Comportement attendu (b) Comportement obtenu

FIGURE 3.4.1 – L'envoi d'un message non-bloquant avec le protocole « Rendez-vous »

3.3.2.3 La progression des communications non-bloquantes dans MPC

MPC intègre la libNBC [45] pour implémenter les collectives MPI non-bloquantes apparues lors de la version 3 de la norme MPI. Un thread de progression est créé pour chaque tâche MPI. Ainsi, avec la version MPI à base de threads, l'ordonnanceur de MPC possède la connaissance de toutes les tâches MPI ainsi que de tous les threads de progression présents sur un nœud. Nous avons la possibilité de placer et d'ordonnancer les threads de progression en ayant une vision globale de tous les threads mis en jeu.

3.4 Problématique de la thèse

La problématique de la thèse est de faire progresser les communications collectives MPI non-bloquantes de manière efficace. Ce qu'espère un développeur lorsqu'il fait appel aux opérations non-bloquantes est que le calcul et la communication s'effectuent simultanément (figure 3.4.1a) dans le but de recouvrir les communications par du calcul.

Ce qui peut être obtenu lorsqu'il n'y a pas de progression des communications, est que la communication est effectuée au moment de l'appel à MPI_Wait (figure 3.4.1b). Ceci implique que le temps de transfert des données ne sera pas recouvert par du calcul.

Comme nous l'avons vu, beaucoup de travaux portent sur la progression des communications point-à-point non-bloquantes. Cependant, les communications collectives non-bloquantes sont plus difficiles à faire progresser car cela nécessite non seulement de faire progresser les communications point-à-point (protocole de rendez-vous, etc.) participant à l'algorithme de collective mais aussi de faire progresser l'algorithme de la collective en soi. En effet, si une des tâches MPI

participant à l'algorithme de collective est en retard, cela va retarder toutes les autres tâches MPI. De plus, si plusieurs tâches MPI sont en retard, les retards seront additionnés et l'opération collective sera d'autant plus lente.

Le recouvrement est lié au volume de calcul ainsi qu'au volume de communication. Néanmoins, un bon recouvrement n'est pas synonyme d'un bon temps d'exécution. En effet, nous pouvons avoir un recouvrement de 100% et avoir un temps d'exécution plus long. Il est à noter qu'il s'agit de deux problèmes différents.

Comme nous l'avons vu dans la sous-section 3.2.2.2, certaines implémentations utilisent des threads de progression prenant en charge la progression des communications. Ils se retrouvent ainsi en concurrence non seulement avec les threads de calcul mais aussi avec d'autres threads de progression. Les changements de contextes engendrés ainsi que le partage des ressources de calcul peuvent dégrader les performances du calcul et/ou des communications.

Afin d'améliorer le recouvrement des communications par du calcul avec des threads de progression, nous utiliserons deux leviers permettant d'interagir avec les threads de progression :

1. Le placement des threads :
   - Placements statiques des tâches MPI et des threads de progression.
   - Placements dynamiques où les opérations constituant les collectives MPI non-bloquantes sont déplacées sur différentes ressources en fonction de l'algorithme utilisé.
2. L'ordonnancement des threads de progression.

Dans cette thèse, nous verrons d'abord comment améliorer le recouvrement des communications par du calcul grâce aux placements statiques d'une part et dynamiques d'autre part, respectivement dans les chapitres 4 et 5. Ensuite nous verrons comment l'ordonnancement des threads de progression permet aussi d'améliorer la progression des communications collectives MPI non-bloquantes dans le chapitre 6.

Deuxième partie

Contributions

Chapitre 4

Placement statique des tâches MPI et des threads de progression

Sommaire

Le recouvrement des collectives MPI non-bloquantes doit permettre d'améliorer les performances des codes de calcul. Nous avons vu dans l'état de l'art que plusieurs techniques existent pour recouvrir les communications point-à-point MPI avec du calcul. Nous avons aussi vu qu'il est plus difficile de faire progresser les communications collectives MPI non-bloquantes.

Dans ce chapitre, nous nous intéresserons plus particulièrement au placement des threads de progression dans le but d'améliorer la progression des communications ainsi que leur recouvrement.

Dans la section 4.1, nous présentons de quelle façon nous évaluons le taux de recouvrement des collectives MPI non-bloquantes. Ensuite, dans la section 4.2, nous proposons plusieurs algorithmes de placement des tâches MPI ainsi que des threads de progression dans le but d'optimiser la progression des communications. Enfin, dans la section 4.3, nous étudions l'impact du placement des threads de progression sans dédier de ressources supplémentaires pour la progression des communications en utilisant la technologie SMT.

4.1 Outils d'évaluation des performances

Afin de réaliser des études sur le placement et l'ordonnancement des collectives MPI non-bloquantes, il est nécessaire d'avoir des outils d'évaluation de performances. Ces évaluations sont indispensables dans le domaine du HPC et permettent de rendre compte des performances des codes de calcul afin de les optimiser.

4.1.1 Mesure du taux de recouvrement et du temps d'exécution

Dans notre étude, une des mesures intéressantes est celle du taux de recouvrement des collectives MPI non-bloquantes obtenue en calculant le pourcentage du temps de communication et du temps de calcul s'effectuant en parallèle. Pour cela, trois temps sont mesurés comme l'illustre la figure 4.1.1 : le temps de communication seul (t_{comm}), le temps de calcul seul (t_{comp}) et le temps de la communication en concurrence avec un temps de calcul (t_{ovrl}). Le taux de recouvrement utilisé dans les *IMB-NBC* [47] que nous présenterons dans la sous-section 4.1.2 est donné par l'équation 4.1. Un taux de recouvrement de 100% correspond à un temps d'exécution égal au temps de calcul ou au temps de communication, c'est-à-dire qu'il y a eu un recouvrement total des communications ou du calcul. À l'inverse, un taux de 0 signifie que nous n'avons aucun recouvrement : le temps d'exécution correspond à la somme du temps de calcul et du temps de communication.

$$taux = 100 \times \max(0, \min(1, \left(\frac{t_{comm} + t_{comp} - t_{ovrl}}{\min(t_{comm}, t_{comp})}\right)) \tag{4.1}$$

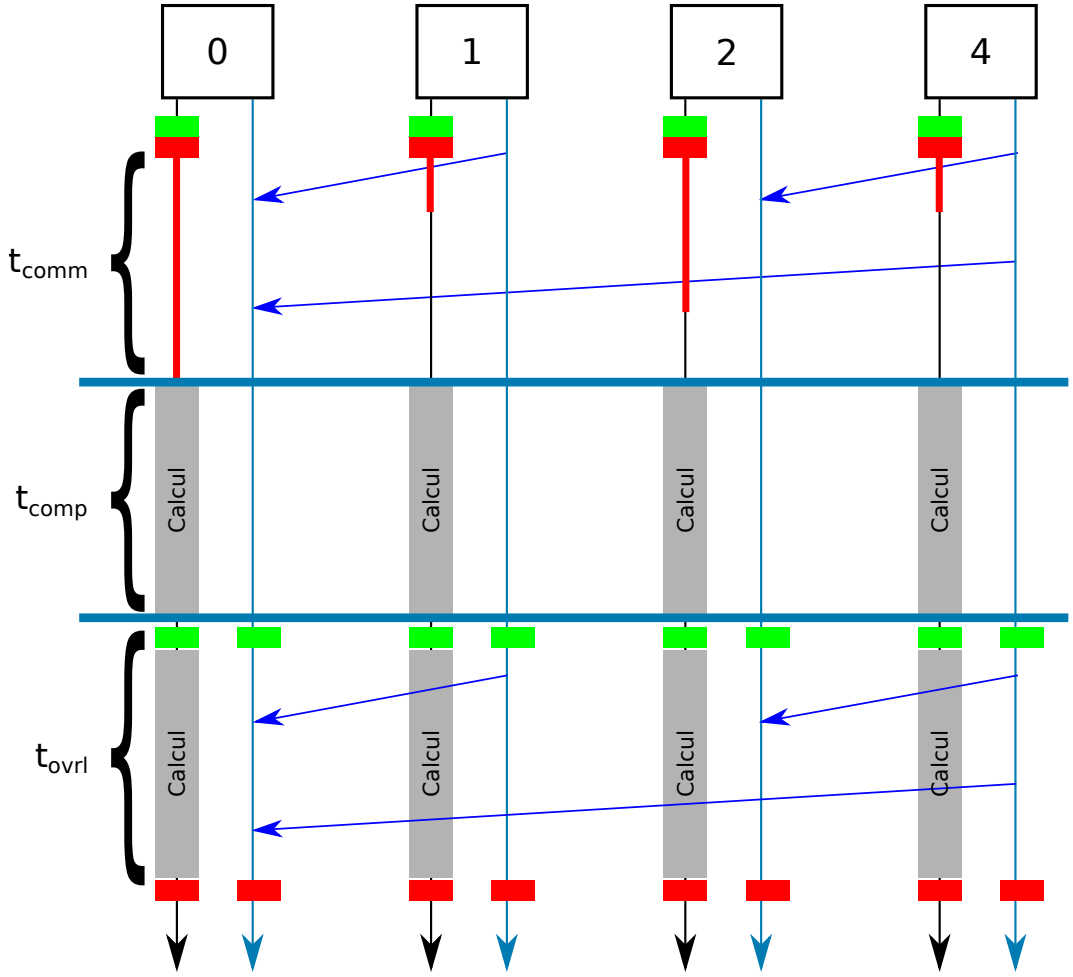


FIGURE 4.1.1 – Illustration du placement « default » avec le taux de recouvrement associé.

L'algorithme 1 décrit le fonctionnement des suites de tests mesurant le taux de recouvrement illustré par la figure 4.1.1. *MPI_Icollective* représente une des 17 collectives MPI que nous souhaitons tester.

Algorithme 1 : Mesure du taux de recouvrement et du temps d'exécution

MPI_Barrier()

$t_{begin} \leftarrow$ MPI_Wtime()
MPI_Icollective(buffer_size)
MPI_Wait()
$t_{comm} \leftarrow$ max of (MPI_Wtime() $- t_{begin}$) for all MPI tasks

MPI_Barrier()

$t_{begin} \leftarrow$ MPI_Wtime()
Computation(compute_size)
$t_{comp} \leftarrow$ max of (MPI_Wtime() $- t_{begin}$) for all MPI tasks

MPI_Barrier()

$t_{begin} \leftarrow$ MPI_Wtime()
MPI_Icollective(buffer_size)
Computation(compute_size)
MPI_Wait()
$t_{ovrl} \leftarrow$ max of (MPI_Wtime() $- t_{begin}$) for all MPI tasks

retourner $t_{comm}, t_{comp}, t_{ovrl}$

Le cas idéal permettant de faire une mesure pertinente est d'avoir un temps de calcul équivalent au temps de communication et d'avoir un taux de recouvrement de 100%. Dans ce cas, non seulement le taux de recouvrement est au maximum, mais nous aurons aussi une diminution du temps d'exécution.

Dans les sous-sections 4.1.2 et 4.1.3, nous présentons d'une part la suite de tests *IMB-NBC* connue pour être le benchmark de référence concernant la mesure du taux de recouvrement. Ensuite nous proposons notre propre suite de tests permettant non seulement de mesurer le taux de recouvrement mais aussi d'étudier le temps d'exécution en fonction du nombre de tâches MPI.

4.1.2 Intel MPI Benchmarks : IMB-NBC

Pour évaluer les performances des implémentations de bibliothèques MPI, Intel a mis en place une suite de tests appelée « Intel MPI Benchmarks » (IMB). Cette suite de tests permet de mesurer les performances des bibliothèques

MPI, notamment pour les opérations point-à-point ainsi que les opérations collectives.

Afin de mesurer le taux de recouvrement, les *IMB-NBC* [47] sont utilisés. Il s'agit d'un composant des IMB conçu pour mesurer le taux de recouvrement des communications collectives non-bloquantes.

Le taux de recouvrement calculé dans *IMB-NBC* est donné par l'équation 4.1. Un taux de recouvrement de 100% correspond à un taux de recouvrement optimal des communications par le calcul. À l'inverse, un taux de 0 signifie que nous n'avons aucun recouvrement. Le calcul et les communications sont alors faits l'un après l'autre. Il est aussi possible qu'un taux de recouvrement de 0% cache le fait que les performances sont dégradées bien plus que si le calcul et les communications étaient effectués l'un après l'autre. C'est une des raisons pour laquelle une autre métrique a été proposée par Alexandre Denis et François Trahay [48] permettant de voir à quel point les performances sont dégradées.

Les *IMB-NBC* ont été conçus de manière à ce que le temps de calcul soit toujours du même ordre de grandeur que le temps de communication. La fonction IMB_cpu_exploit [49] est utilisée pour générer un temps de calcul très proche du temps de communication mesuré précédemment.

Dans cette suite de tests, plusieurs tailles de messages peuvent être testées. Ainsi, nous pouvons voir comment évolue le comportement des algorithmes suivant la taille du message, notamment les changements d'algorithmes de communication lorsque la taille du message augmente. Suivant les implémentations MPI, ce seuil peut varier mais généralement, pour les petites tailles de *buffer*, le protocole *Eager* est utilisé. Pour les tailles de *buffer* plus grandes, le protocole *rendez-vous* est privilégié. Ces protocoles sont décrits dans la section 2.2.2.1.

Cependant, cette suite de tests ne suffit pas pour tous les besoins. En effet, à notre connaissance, il n'est pas possible de fixer la taille du problème et de faire varier le nombre de tâches MPI afin d'avoir un volume de calcul global constant pour un nombre différent de tâches MPI. Le but étant de pouvoir mesurer le surcoût dû au fait de retirer des cœurs initialement prévus pour exécuter du calcul, pour ensuite les dédier à exécuter les threads de progression générés par les tâches MPI pour améliorer le recouvrement des communications par du calcul.

C'est pour cette raison que nous proposons notre propre suite de tests dédiée aux collectives MPI, que nous allons présenter dans la section suivante.

4.1.3 MPC-NBC-Bench : une suite de tests dédiés aux collectives MPI

Pour chaque collective MPI, nous créons un test implémentant l'algorithme 1 en langage C. Celui-ci permet de mesurer le taux de recouvrement ainsi que le temps d'exécution de la collective MPI testée.

La fonction *Icollective* correspond à une des 17 collectives MPI implémentées dans notre suite de tests. La fonction *Computation* correspond au calcul avec lequel nous allons recouvrir la communication collective. Celle-ci est là uniquement dans ce but et le calcul que nous réalisons est indépendant par tâche MPI.

Afin d'avoir plus de flexibilité dans nos tests, nous pouvons faire varier plusieurs paramètres tels que :

— *La taille des communications* : Il s'agit de la taille des *buffers* échangés lors des communications MPI en octet. Cela a pour conséquence d'augmenter ou de diminuer le temps de communication. Ce paramètre correspond à une variable d'environnement de notre suite de tests.

— *La taille du calcul* : Il s'agit de la taille des matrices données pour le calcul. Cela a pour conséquence d'augmenter ou de diminuer le temps de calcul. Cette taille correspond à une variable d'environnement de notre suite de tests.

— *Le noyau de calcul* : Il s'agit de savoir combien d'opérations sont réalisées par rapport au volume de données nécessaire à réaliser ces opérations. Par exemple, une addition de matrice a une intensité arithmétique plus faible que celle d'un produit de matrice car pour la même quantité de données, un produit de matrice nécessite plus d'opérations. Nous pouvons donc choisir quel est le type de calcul qui sera exécuté pour faire varier l'intensité arithmétique.

— *Le nombre de répétitions du test* : Il s'agit de savoir combien de fois le test est lancé afin calculer le temps moyen, médian, minimal et maximal du test.

La grande différence avec *IMB-NBC* est que nous pouvons activer certains modes de compilation qui activent ou désactivent certaines options de notre suite de tests. Le mode *Equivalent Compute*, permet de calculer la taille des matrices données à la fonction *Computation* pour chaque tâche MPI. Le but est de pouvoir comparer le temps d'exécution avec un nombre de tâches MPI différent tout en gardant un volume de calcul global constant. Les tailles des matrices locales à chaque rang MPI sont calculées à partir de l'équation 4.2 sous l'hypothèse que le calcul est un produit de matrices carrées ou une addition de matrices carrées.

$$\text{compute\_size\_local} = \begin{cases} \sqrt[3]{\dfrac{\text{compute\_size}^3}{\text{MPI\_Comm\_size}}} & \text{si : produit de matrice} \\ \sqrt{\dfrac{\text{compute\_size}^2}{\text{MPI\_Comm\_size}}} & \text{si : addition de matrice} \end{cases} \tag{4.2}$$

Il existe aussi le mode *CPU Exploit* donnant la possibilité de faire comme pour *IMB-NBC*, générer un calcul qui mettra le même temps que l'opération collective pour avoir un temps de calcul et un temps de communication

équivalent.

Notre suite de tests permet aussi de vérifier si l'implémentation de la collective non-bloquante nous donne les résultats attendus. Pour cela, nous comparons le résultat des collectives MPI non-bloquantes avec le résultat que nous auraient donné les collectives MPI bloquantes.

4.1.4 Discussion

Les suites de tests que nous avons présentées sont indispensables. Elles permettent d'évaluer les bibliothèques MPI dans le but d'améliorer leur performance. Dans cette section, nous avons présenté la suite de test la plus connue pour mesurer le recouvrement des collectives non-bloquantes. Néanmoins, elle ne convient pas quand il s'agit de fixer la taille du problème pour garder une charge de travail constante avec un nombre différent de tâches MPI. C'est la raison pour laquelle nous avons développé notre propre suite de tests.

Dans la section suivante, nous allons étudier l'impact du placement des threads de progression pour les collectives MPI non-bloquantes.

4.2 Impact du placement des threads de progression pour les collectives MPI non-bloquantes

Dans cette section, nous nous concentrons sur les communications collectives non-bloquantes apparues lors de la version 3.0 de la norme MPI [1]. Nous proposons un placement de threads permettant un meilleur recouvrement.

4.2.1 Placement des tâches MPI

Le placement des tâches MPI sur les cœurs de calcul est un sujet largement étudié dans la littérature. Celui-ci contribue à obtenir de bonnes performances sur un supercalculateur. En effet, si nous ne plaçons pas les tâches MPI sur des cœurs bien définis, il est possible que toutes les tâches MPI se retrouvent sur le même cœur. Si cela arrive, le programme ne bénéficiera pas du parallélisme, bien que le programme soit exécuté par plusieurs tâches MPI. Un cas moins extrême mais tout aussi problématique est que les threads allouent de la mémoire sur un nœud NUMA et soient exécutés sur un autre nœud NUMA. Ces threads ne bénéficieront pas de la localité de la mémoire et auront donc les problèmes de performance qui y sont liés.

Le placement « *scatter* », qui consiste à écarter les tâches MPI au maximum sur un nœud de calcul, est généralement celui qui est utilisé par défaut. C'est le cas dans l'implémentation MPI de MPC. Celui-ci permet de maximiser le nombre de cœurs disponibles par tâches MPI lors de l'utilisation d'OpenMP comme modèle de programmation intra-nœud. Néanmoins, ce placement n'est

pas le plus approprié pour le placement des threads de progression, car il ne prend pas en compte les différents nœuds NUMA au sein d'un nœud. Nous voulons que les threads de progression associés à une tâche MPI soient liés sur le même nœud NUMA.

Notre approche est de prendre en compte la topologie de la machine sous-jacente pour placer les différents threads mis en jeu. Pour cela, notre idée est de placer les tâches MPI de manière à les écarter au maximum au sein du même nœud NUMA. Ce placement que nous nommons « $scatter_{numa}$ » permet d'assurer ensuite un bon placement des threads de progression.

Notre démarche a d'abord été de compter le nombre de nœuds NUMA au sein d'un nœud (N_{numa}). Nous avons ensuite divisé le nombre de tâches MPI (N_{mpi}) par N_{numa} afin de savoir combien de tâches MPI devaient être placées par nœud NUMA. Sachant cela, il nous a suffi de placer les tâches MPI de façon à les écarter au maximum (politique « *scatter* ») au sein d'un nœud NUMA pour donner le placement « $scatter_{numa}$ ».

Ce placement est défini par l'algorithme 2, où r représente le rang MPI courant, $Ncpu_{numa}$ représente le nombre de cœurs par nœuds NUMA, id_{numa} représente l'identifiant de chaque nœud NUMA, id_{local} représente l'identifiant de chaque tâche MPI locale à un nœud NUMA, n_{local} représente le nombre de tâches MPI dans le nœud NUMA local, pos_{local} représente le placement de la tâche MPI dans le nœud NUMA local, et pos_{global} représente l'identifiant du cœur où sera placée la tâche MPI de rang r.

Algorithme 2 : Algorithme de placement des tâches MPI « $scatter_{numa}$ »

Entrées : r, N_{mpi}, N_{numa}, $Ncpu_{numa}$

$id_{numa} \leftarrow \lfloor \frac{r \times N_{numa}}{N_{mpi}} \rfloor$

$id_{local} \leftarrow r - \lceil \frac{id_{numa} \times N_{mpi}}{N_{numa}} \rceil$

$n_{local} \leftarrow \lceil \frac{(id_{numa}+1) \times N_{mpi}}{N_{numa}} \rceil - \lceil \frac{(id_{numa}) \times N_{mpi}}{N_{numa}} \rceil$

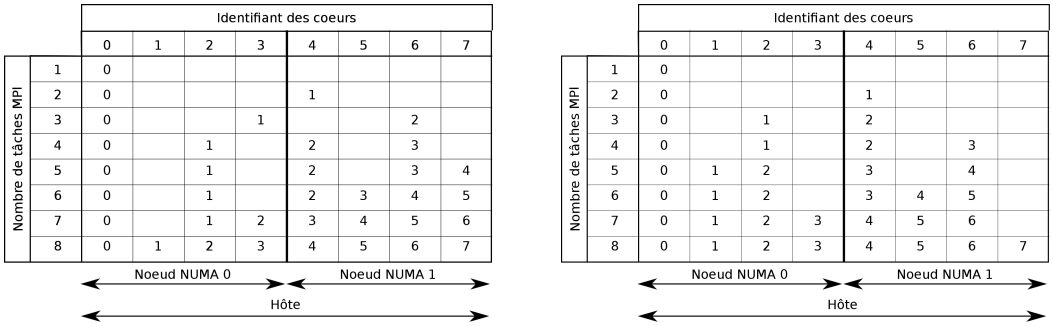
$pos_{local} \leftarrow \lfloor \frac{id_{local} \times Ncpu_{numa}}{n_{local}} \rfloor$

$pos_{global} \leftarrow pos_{local} + (id_{numa} \times Ncpu_{numa})$

retourner pos_{global}

Les figures 4.2.1a et 4.2.1b représentent les différents placements engendrés par les algorithmes de placement de threads sur un nœud de 8 cœurs avec 2 nœuds NUMA. Nous y voyons la répartition des tâches MPI sur les cœurs. Chaque ligne représente le placement des tâches MPI en fonction du nombre de tâches MPI choisi lors du lancement du programme MPI.

La figure 4.2.1a illustre l'algorithme de placement par défaut dans MPC. Il s'agit d'un placement écartant les tâches MPI au maximum au sein d'un nœud pour permettre de peupler les cœurs libres avec des threads OpenMP. Cet algorithme ne prend pas en compte les différents nœuds NUMA au sein d'un nœud.



(a) *scatter* (b) $scatter_{numa}$

FIGURE 4.2.1 – Placement des tâches MPI sur un nœud de 8 cœurs avec 2 nœuds NUMA en fonction du nombre de tâches MPI.

Sur la figure 4.2.1b, nous avons le placement que nous avons mis en place pour tenir compte des nœuds NUMA. Nous plaçons les threads de façon à avoir un nombre de tâches MPI équilibré entre les nœuds NUMA. Cela permet non seulement de placer les threads de progression par nœud NUMA mais aussi de répartir la charge de calcul sur les nœuds NUMA et de bénéficier au maximum des effets de caches.

4.2.2 Placement des threads de progression

Rappelons que pour recouvrir les communications par du calcul, le calcul et les communications doivent s'exécuter en parallèle. Pour cela, nous dédions des cœurs aux threads de progression. Le problème est de savoir comment placer les threads de progression sur les cœurs qui leurs ont été attribués.

Notre proposition est de placer les threads de progression en fonction du placement des tâches MPI. La figure 4.2.2 reprend les placements des tâches MPI donnés sur la figure 4.2.1b. Le placement des threads de progression générés par leur tâche MPI est affiché en bleu.

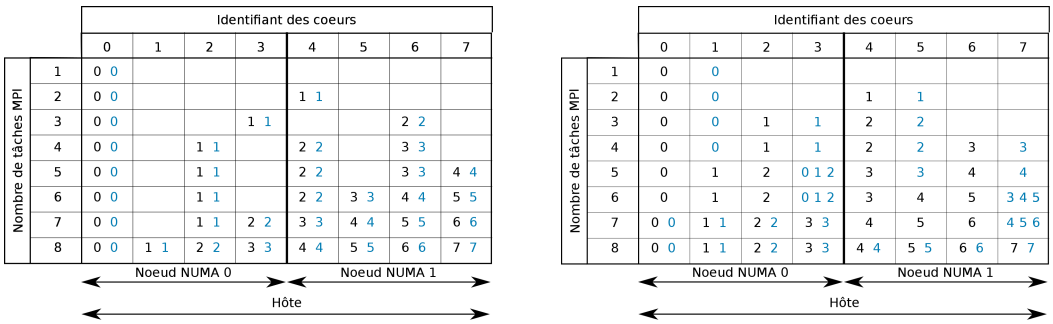
La figure 4.2.2a illustre le placement des threads de progression lorsqu'ils sont liés sur les mêmes cœurs que les tâches MPI qui les ont générés (placement « *bind* »). Ce placement est le placement des threads de progression par défaut dans MPC. La figure 4.2.2b décrit le placement « numa » résultant de l'algorithme 3 que nous proposons pour permettre une meilleure progression où N_{numa} représente le nombre de nœuds NUMA au sein d'un nœud, $Ncpu_{numa}$ représente le nombre de cœurs par nœuds NUMA, n_{local} représente le nombre de tâches MPI dans le nœud NUMA local, pos_{global} représente l'identifiant du cœur où est placée la tâche MPI courante et $pos_{threadProgression}$ représente l'identifiant du cœur où sera placé le thread de progression associé à la tâche MPI courante.

Nous plaçons les threads de progression sur les cœurs libres répartis au sein du même nœud NUMA. De cette manière, les threads de progression bénéficieront

toujours de la localité des données. S'il ne reste aucun cœur de libre, les threads de progression seront liés sur les mêmes cœurs que les tâches MPI comme pour le placement « *bind* ».

Algorithme 3 : Algorithme de placement des threads de progression « numa »

Entrées : N_{numa}, $Ncpu_{numa}$, n_{local}, pos_{global}
si $n_{local} \geq Ncpu_{numa}$ **alors**
  $pos_{threadProgression} \leftarrow pos_{global}$
sinon
  $\delta \leftarrow \frac{Ncpu_{numa}}{Ncpu_{numa} - n_{local}}$
  $pos_{threadProgression} \leftarrow \lceil (\lfloor \frac{pos_{global}}{\delta} \rfloor + 1) \times \delta \rceil - 1$
fin
retourner $pos_{threadProgression}$



(a) *bind* (b) *numa*

FIGURE 4.2.2 – Placement des tâches MPI en noir et des threads de progression en bleu sur un nœud de 8 cœurs avec 2 nœuds NUMA en fonction du nombre de tâches MPI.

Sur la troisième ligne de la figure 4.2.2b, les trois tâches MPI sont réparties sur les nœuds NUMA et les threads de progression associés sont sur les cœurs libres sur le même nœud NUMA. Le but de notre méthode est de placer les threads de progression sur le même nœud NUMA que leurs tâches MPI associées. Pour 7 tâches MPI, nous avons donc un comportement similaire sur le nœud NUMA. Sur le premier nœud NUMA, il n'y a pas de cœurs de libres alors les threads de progression sont liés sur les mêmes cœurs que les tâches MPI. En revanche, sur le second nœud NUMA, un cœur est libre alors tous les threads de progression de ce nœud NUMA vont y être liés.

4.2.3 Implémentation

Nous avons ajouté ces algorithmes de placement des tâches MPI au sein du support exécutif MPC. MPC utilise *hwloc* [10] pour avoir une vision globale de la topologie de la machine. Pour rappel, les tâches MPI sont des threads dans MPC. Il intègre son propre ordonnanceur de threads. Cela nous permet d'avoir un contrôle très fin sur le placement de tous les threads utilisés dans MPC. Ces threads peuvent être les tâches MPI, les threads OpenMP ou bien les threads de progression générés par les collectives MPI non-bloquantes. Nous pouvons donc définir le placement des tâches MPI et des threads de progression à leur création avec les algorithmes 2 et 3 implémentés dans MPC.

4.2.4 Évaluation du taux de recouvrement

Afin de mesurer l'impact du placement des threads de progression, nous utilisons *IMB-NBC* que nous avons présentée dans la section 4.1.2. Cette suite de test a été conçue pour mesurer le taux de recouvrement des communications collectives non-bloquantes. Nous avons lancé la suite de tests sur 4 nœuds possédant 2 processeurs quad-cores INTEL Xeon X5560 à 2.80 GHz constituant chacun 2 nœuds NUMA avec un réseau InfiniBand. Nous testons toutes les collectives MPI non-bloquantes avec la configuration par défaut de MPC que nous nommerons « default » ainsi qu'avec la configuration que nous avons mise en place. Cette configuration permet d'améliorer le recouvrement des communications ; nous la nommerons « numa ». Pour chaque collective, nous faisons varier le nombre de tâches MPI impliquées dans celle-ci ainsi que la taille des données à utiliser lors de ces communications.

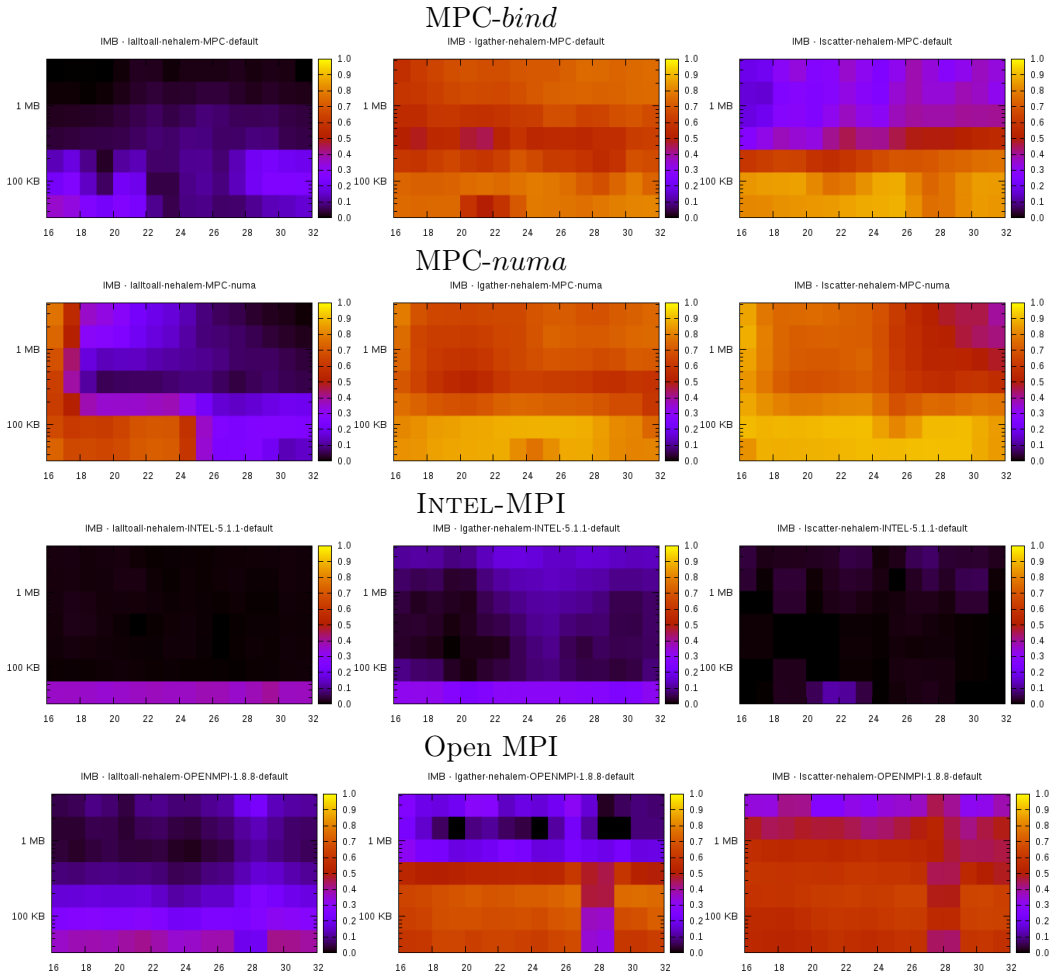
Lors de l'utilisation de *IMB-NBC*, le temps de calcul est toujours du même ordre de grandeur que le temps des communications dû à l'utilisation de la fonction IMB_cpu_exploit [49] générant un temps de calcul proche du temps de communication. Le taux de recouvrement est donné par l'équation 4.1 à la page 48.

Sur la figure 4.2.3, un taux de 1 correspond à un taux de recouvrement de 100%. À l'inverse, un taux de 0 signifie que nous n'avons aucun recouvrement. Cette figure illustre les résultats des tests sur les collectives non-bloquantes pour trois types d'opérations :

— **Tous-vers-Tous** : MPI_Ialltoall
— **Tous-vers-Un** : MPI_Igather.
— **Un-vers-Tous** : MPI_Iscatter.

Ces opérations sont testées avec plusieurs implémentations MPI telles que INTEL-MPI, Open MPI avec leurs options par défaut ainsi qu'avec MPC avec le placement « bind » et « numa ». Pour chaque collective, nous mesurons le taux de recouvrement en utilisant *IMB-NBC*. Nous avons le nombre de tâches MPI en abscisse et la taille des données (en octets) utilisée pour les

communications en ordonnée. La couleur représente le taux de recouvrement des communications. La couleur jaune représente un taux de recouvrement de 100% tandis que le noir correspond à un taux de 0%.



(a) Ialltoall (b) Igather (c) Iscatter

FIGURE 4.2.3 – Taux de recouvrement des collectives Ialltoall, Igather et Iscatter sur 32 cœurs en fonction du nombre de tâches MPI et de la taille des messages en octets sur 4 nœuds de 8 cœurs

Sur les résultats correspondant à « MPC-*bind* » et « MPC-*numa* », le seuil de 24 tâches MPI correspond à un cœur de libre pour chaque nœud NUMA dans notre cas test. Le calcul n'est donc jamais perturbé par les communications avec le placement des threads de progression « *numa* ». Au-delà de 24, certains threads de progression sont liés sur le même cœur que les tâches MPI qui les ont générés s'il n'y a plus de cœurs de libres au sein de leur nœud NUMA.

Les seuils observés lorsque nous dépassons une taille 128 Ko pour MPC et

64 Ko pour INTEL-MPI correspondent à un changement d'algorithme. C'est le moment où est mis en œuvre le protocole de « rendez-vous ». Au-delà de ce seuil nous avons un taux de recouvrement inférieur. Néanmoins, lorsque nous avons un cœur dédié à chaque thread de progression, nous pouvons observer un très bon taux de recouvrement malgré le changement d'algorithme pour le cas « MPC-*numa* ». De plus, le taux de recouvrement est de l'ordre de 80% lorsque nous avons des ressources dédiées aux threads de progression et que nous sommes avec des tailles de messages inférieures au seuil de changement de protocole.

Nous pouvons voir qu'il est très compliqué de faire progresser les communications pour la collective MPI_Ialltoall pour INTEL-MPI », « Open MPI » ainsi qu'avec MPC avec l'algorithme *bind* utilisé par défaut. En effet, il s'agit de la fonction qui génère le plus de messages car toutes les tâches MPI doivent envoyer un message à toutes les autres tâches MPI. Néanmoins, notre algorithme de placement des threads de progression permettant de bénéficier des cœurs libres permet d'obtenir un taux de recouvrement de l'ordre de 60% lorsque chaque thread de progression possède des cœurs dédiés ou bien quand nous avons un cœur dédié aux threads de progression par nœud NUMA avec le protocole « Eager ».

Les résultats des tests que nous avons effectués démontrent l'importance du placement des threads de progression.

4.2.5 Évaluation du temps d'exécution

Afin d'évaluer le temps d'exécution du calcul/communication, nous utilisons *MPC-NBC-Bench* que nous avons présenté dans la section 4.1.3. Nous avons vu que quand les threads de progression partageaient le même cœur que les threads de calcul, il en résultait un taux de recouvrement de l'ordre de 0 à 20% comme l'illustre la figure 4.2.4. Sur la gauche de cette figure, nous voyons le placement des tâches MPI et des threads de progression. Sur la droite, nous voyons une représentation du taux de recouvrement comme sur la figure 4.2.3.

Lorsqu'au contraire, chaque thread possède un cœur qui lui est dédié pour effectuer la progression des communications, nous obtenons un taux de recouvrement de l'ordre de 80 à 90% (figure 4.2.5). Pour obtenir ces bons taux de recouvrement il est nécessaire d'avoir plus de cœurs pour les dédier aux threads de progression ou bien de faire moins de tâches MPI pour laisser des cœurs de libres.

Néanmoins, il est impensable de réserver la moitié des cœurs disponibles uniquement pour faire progresser les communications. C'est pourquoi nous avons testé un compromis entre les deux approches qui est de réserver moins de cœurs pour les threads de progression. Nous avons donc testé de réserver 1 cœur par nœud NUMA et nous avons obtenu de bons taux de recouvrement avec le protocole « Eager » (figure 4.2.6).

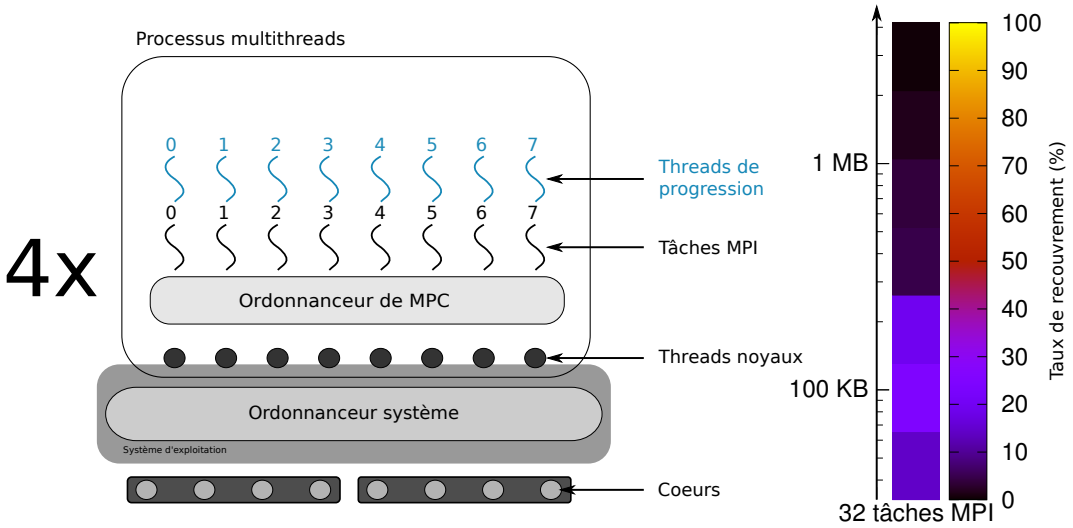


FIGURE 4.2.4 – Illustration du placement « bind » avec le taux de recouvrement associé pour la collective Ialltoall.

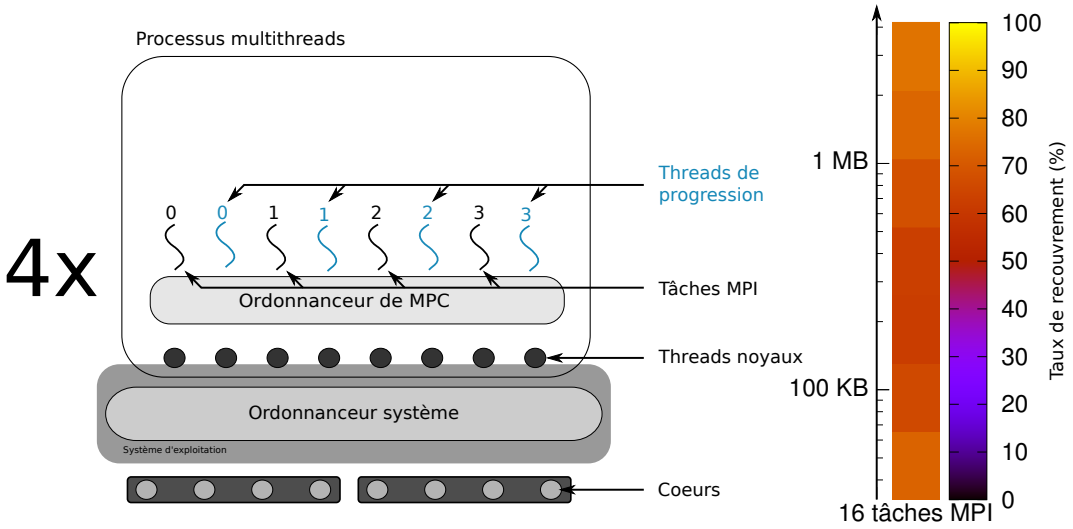


FIGURE 4.2.5 – Illustration du placement « numa » quand nous avons un cœur libre pour chaque thread de progression avec le taux de recouvrement associé pour la collective Ialltoall.

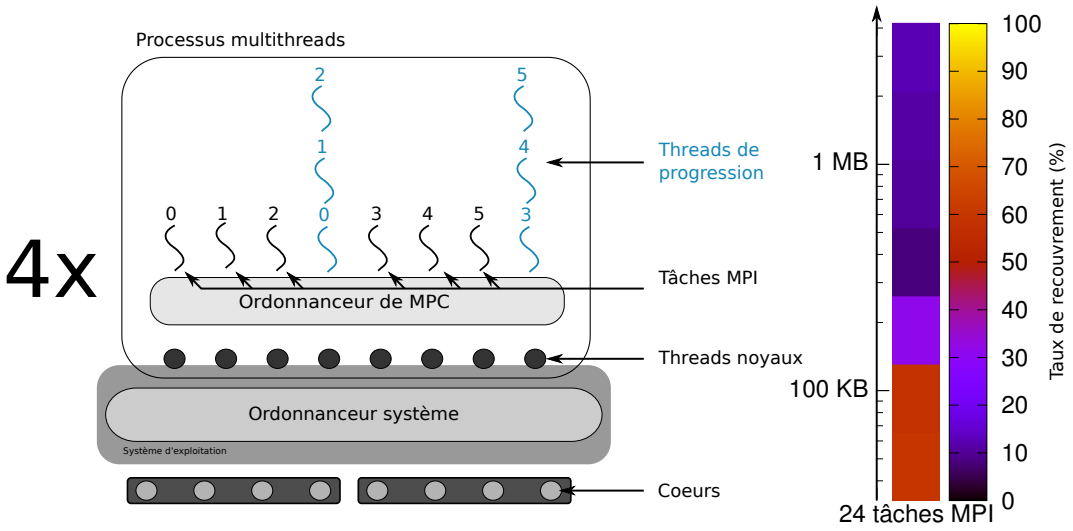


FIGURE 4.2.6 – Illustration du placement « numa » quand nous avons un cœur libre dans chaque noeud NUMA pour y placer les threads de progression avec le taux de recouvrement associé pour la collective Ialltoall.

Afin de voir l'effet du placement des threads de progression sur le temps d'exécution, nous avons exécuté la collective MPI_Ialltoall sur un KNL de 68 cœurs avec « MPC-NBC-bench » décrit dans la section 4.1.3 avec le mode *Equivalent Compute*. Nous testons deux algorithmes de placement des threads de progression différents : « bind » et « numa ». Les résultats sont illustrés sur la figure 4.2.7.

Nous y voyons plusieurs courbes. Sur l'axe des abscisses, nous avons le nombre de tâches MPI. Sur l'axe des ordonnées de droite, nous avons les mesures du taux de recouvrement tandis que sur l'axe des ordonnées de gauche, nous avons les mesures du temps d'exécution. La courbe grise (Tcpu) représente le temps de calcul pour les deux algorithmes. En revanche, les temps de communications (Tcomm-bind et Tcomm-numa) ne sont pas identiques. Le temps de communication pour le placement « bind » (Tcomm-bind) augmente linéairement quand nous avons plus de tâches MPI. Ceci est dû au fait que plus nous avons de tâches MPI, plus nous avons de messages à échanger.

Le temps de communication avec l'algorithme de placement « numa » (Tcomm-numa) augmente très rapidement quand le nombre de tâches MPI augmente. Ceci est dû au fait que les threads de progression sont placés sur les cœurs restants libres, c'est-à-dire que plus le nombre de tâches MPI augmente, moins nous avons de cœurs libres pour les threads de progression.

Néanmoins, nous pouvons voir que le temps d'exécution avec le placement « numa » (Tovrl-numa) est plus petit que celui du placement « bind » (Tovrl-bind) jusqu'à 52 tâches MPI. Ceci s'explique par le fait que nous avons un taux de recouvrement de l'ordre de 80% (Recouvrement-numa) avec le place-

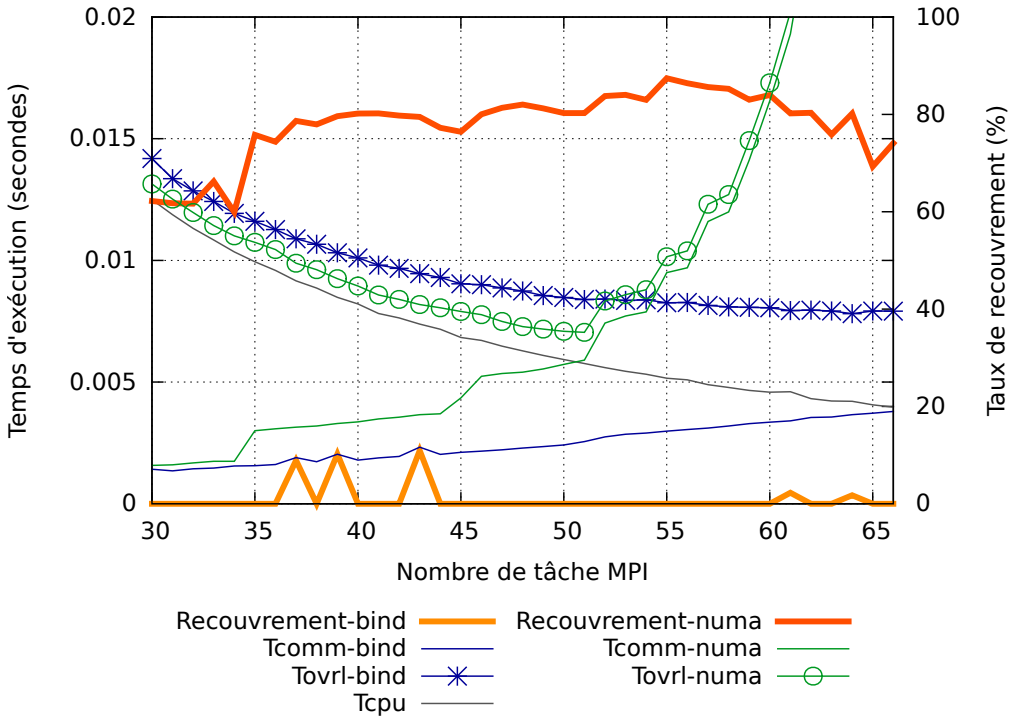


FIGURE 4.2.7 – Temps d'exécution, temps de calcul et temps de communication pour un Ialltoall avec les placements « bind » et « numa ».

ment « numa » alors qu'avec le placement « bind », nous avons un taux de recouvrement de 0% (Recouvrement-bind).

Après 52 tâches MPI, les threads de progression ne disposent plus d'assez de cœurs libres pour permettre d'obtenir un bon temps d'exécution car toutes les communications s'effectuent sur trop peu de cœurs dédiés. Nous avons donc un meilleur temps d'exécution avec l'algorithme « numa » quand 17 cœurs sont dédiés à la progression sur une machine disposant de 68 cœurs. C'est-à-dire que nous avons dédié un quart de la machine pour les threads de progression afin d'avoir un meilleur temps d'exécution.

4.2.6 Conclusion

Nous avons vu qu'il était nécessaire d'exécuter les communications et les calculs en parallèle pour recouvrir les communications par du calcul. Des cœurs sont donc dédiés aux threads de progression. Nous proposons des algorithmes de placement des threads MPI et des threads de progression montrant qu'utiliser un placement par nœud NUMA nous permettait de mieux recouvrir les communications par du calcul. Cependant, il est nécessaire d'allouer des ressources pour les threads de progression afin d'avoir un taux de recouvrement supérieur à 80%.

Nous constatons aussi que la plupart des machines actuelles disposent de microprocesseurs dotés de la technologie SMT qui est rarement utilisée dans le contexte HPC car les Hyper-Threads se partagent les même unités de calcul. Néanmoins, la progression des communications n'utilisent pas les mêmes unités de calcul mis à part pour les réductions. C'est pourquoi nous étudions l'impact du placement des threads de progression sur la progression des communications en utilisant la technologie SMT dans le but de ne pas allouer de cœurs pour les threads de progression.

4.3 Étude du placement des threads de progression sur les Hyper-Threads

Allouer des cœurs pour les threads de progression n'est pas la seule option pour exécuter les threads de progression en même temps que les threads de calcul. Dans la section 2.1.3, nous avons vu que la plupart des microprocesseurs actuels utilisent les SMT, plus connu sous le nom d'Hyper-Threading pour les microprocesseurs d'INTEL. Certaines applications scientifiques n'utilisent pas les *Hyper-Threads* à cause des problèmes de performances décrits dans la section 2.1.3 à la page 13 : l'utilisation des *Hyper-Threads* pour faire plus de calcul génère de la contention sur les unités arithmétiques et logiques. Cependant, les threads de progression n'ont pas besoin de ces unités de calcul pour effectuer la progression des communications, ou très peu si la communication nécessite une

réduction. Ainsi, le placement des threads de progression sur les *Hyper-Threads* semble être une bonne idée pour faire progresser les communications sans ralentir le calcul. Étant donné que la plupart des communications n'utilisent pas les unités arithmétiques, nous espérons faire la progression des communications sans surcoût en plaçant les threads de progression sur les *Hyper-Threads*. L'idée est d'utiliser les SMT pour ne pas avoir à allouer de cœurs pour les threads de progression.

Afin d'évaluer l'impact de l'utilisation des *Hyper-Threads* sur la progression des collectives MPI non-bloquantes, nous allons d'abord étudier le placement des threads de progression sur les *Hyper-Threads* en distinguant deux cas :

— Le cas des communications inter-nœuds utilisant le réseau pour faire les communications, où les threads de progression exécutent seulement l'algorithme pour l'opération collective, le protocole de rendez-vous, la programmation des *DMA* sur le réseau, mais qui ne consomment pas beaucoup de cycles CPU.

— Le cas des communications intra-nœud se partageant la mémoire pour les communications, où les transferts de données sont essentiellement des copies mémoires (*memcpy*) consommant plus de cycles CPU.

Ensuite, nous expliquerons comment les effets de caches influent sur le recouvrement des communications par du calcul ainsi que sur le temps d'exécution quand nous plaçons les threads de progression sur les *Hyper-Threads* pour les communications intra-nœuds.

4.3.1 Description de la méthode de test

Nous avons testé plusieurs politiques de placement pour les threads de progression, avec ou sans les *Hyper-Threads*, ainsi que des communications inter-nœuds et intra-nœuds.

Afin d'évaluer l'utilisation des *Hyper-Threads* pour effectuer la progression des communications collectives non-bloquantes, nous avons utilisé notre suite de tests présentée dans la sous-section 4.1.3. Pour rappel, cette suite de tests exécute une collective non-bloquante entrelacée avec du calcul décrit par l'algorithme 1. Dans ce cas, le calcul est une multiplication de matrice. Nous utilisons le mode *Equivalent Compute* permettant de comparer le temps d'exécution en gardant une charge globale de calcul constante lorsque nous utilisons différents algorithmes de placement des threads de progression. Nous avons choisi une taille de *buffer* de 2 Mo afin d'avoir un *buffer* assez gros pour augmenter le temps de communication afin que le recouvrement des communications par le calcul ait un sens. Ensuite, nous avons ajusté la charge de calcul pour obtenir un taux de recouvrement de 100% quand nous avons des cœurs dédiés aux threads de progression.

Nous avons lancé les tests sur une architecture multi-cœurs. Il s'agit de l'Intel Xeon E5-2698 v3 @2.30GHz avec 32 cœurs par nœud, et 128 Go de

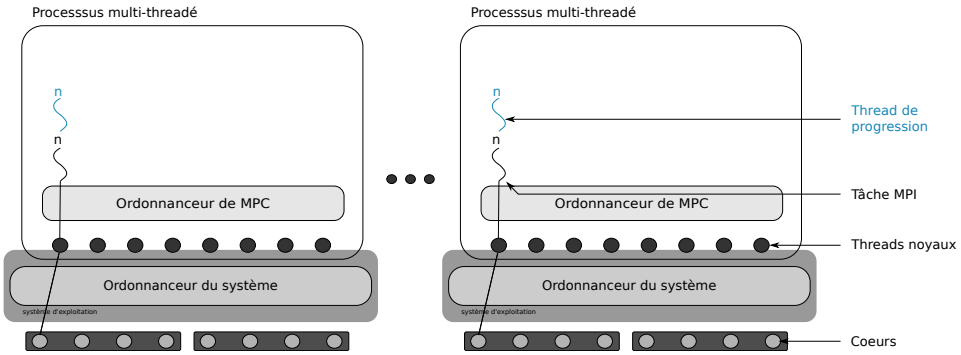


FIGURE 4.3.1 – Placement « no-smt-bind »

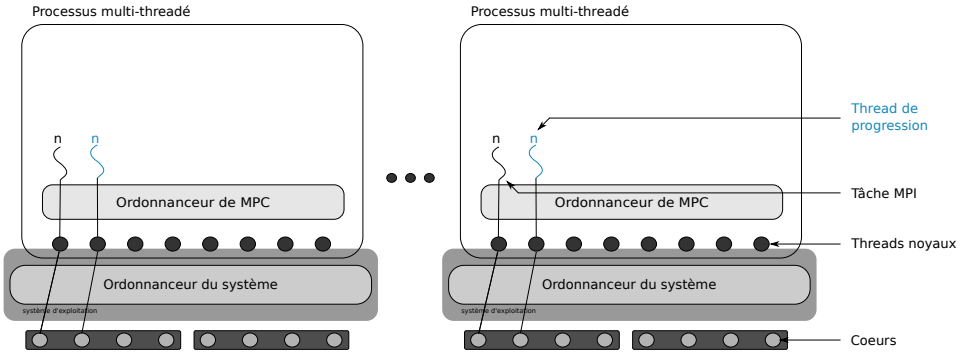


FIGURE 4.3.2 – Placement « dedicated-core »

RAM (Haswell).

Les tâches MPI sont placées avec la politique « $scatter_{numa}$ » vue dans la sous-section 4.2.1 à la page 53. Nous testons trois politiques de placement des threads de progression différentes.

— « no-smt-bind » : les threads de progression sont liés sur le même cœur que celui de la tâche MPI qui lui est associée et l'*Hyper-Threading* est désactivé (figure 4.3.1).
— « dedicated-core » : chaque thread de progression est lié sur un autre cœur mais en utilisant deux fois plus de cœurs que pour les autres cas (figure 4.3.2).
— « smt » : chaque thread de progression est lié sur le même cœur que celui de la tâche MPI qui lui est associée mais sur un autre *Hyper-Thread* (figure 4.3.3). Nous distinguerons deux cas « smt-default » et « smt-sleep » que nous expliquerons.

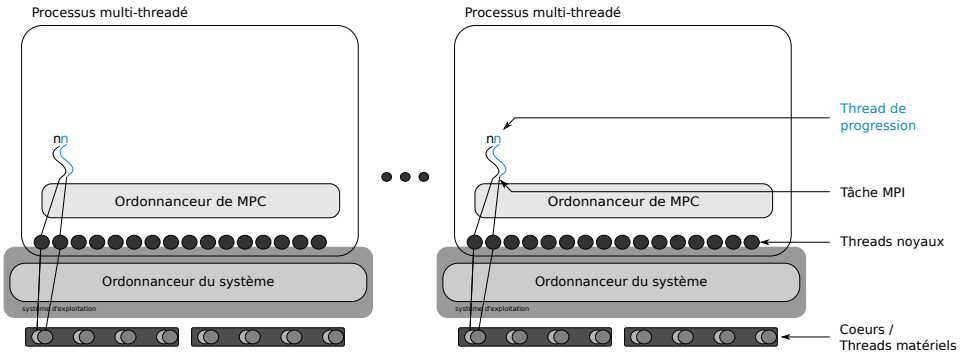


FIGURE 4.3.3 – Placement « smt »

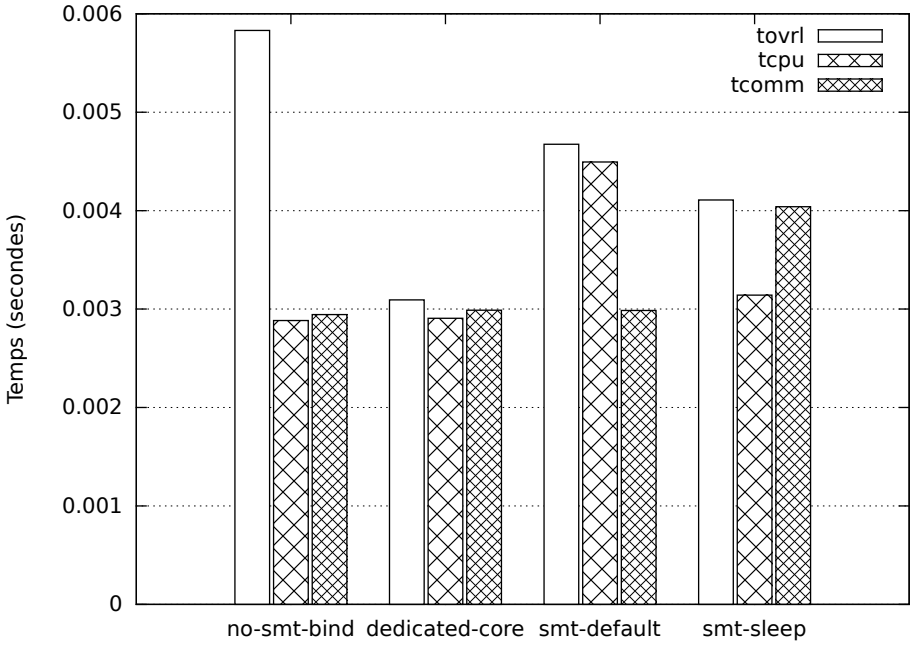
4.3.2 Utilisation des Hyper-Threads pour les communications inter-nœuds

Pour étudier l'impact de l'utilisation de l'*Hyper-Threading* dans le cas des communications inter-nœuds, nous avons exécuté notre suite de tests sur 8 nœuds Haswell avec seulement une tâche MPI par nœud. Il s'agit de l'utilisation usuelle lorsque MPI est utilisé avec un autre modèle de programmation pour les communications intra-nœud tel qu'OpenMP.

Les résultats pour les communications inter-nœuds sont décrits dans la figure 4.3.4. Pour le placement « no-smt-bind », nous ne voyons aucun recouvrement et le temps d'exécution est de $5,8\,ms$. C'est le comportement attendu. Étant donné que MPC est non-préemptif, le calcul et les communications sont exécutés l'un après l'autre si les threads de progression et les threads de calcul sont placés sur les mêmes cœurs. Nous observons que les communications ont besoin d'utiliser le CPU pour faire progresser les communications, pas nécessairement pour faire progresser les communications dans le réseau mais au moins pour faire progresser l'algorithme de collective et le protocole de rendez-vous quand les messages dépassent une certaine taille.

Le placement « dedicated-core » permet de mieux recouvrir les communications par du calcul, avec un taux de recouvrement de 96% pour un temps d'exécution de $3,0\,ms$. Il s'agit du comportement attendu puisque dédier un cœur pour chaque thread de progression permet de faire progresser les communications en arrière-plan sans aucune difficulté. Le calcul est exécuté sur un cœur et la progression des communications sur un autre cœur. Cela permet d'avoir un recouvrement parfait. Cependant, cette configuration utilise deux fois plus de ressources que les autres configurations.

Le placement « smt-default » avec les paramètres par défaut conduit à un taux de recouvrement de 94% pour un temps d'exécution de $4,6\,ms$. Quand les communications et le calcul se recouvrent bien, nous observons aussi que t_{cpu} augmente significativement. Cela est dû à notre implémentation MPI. Quand l'*Hyper-Threading* est activé, MPC crée un *thread noyau* pour chaque *Hyper-*



Taux de recouvrement :

- no-smt-bind : 0%
- dedicated-core : 96%
- smt-default : 94%
- smt-sleep : 98%

FIGURE 4.3.4 – t_{ovrl}, t_{comm} et t_{comp} pour une opération *Ialltoall* avec un *buffer* de taille constante de 2 Mo sur 8 nœuds avec 8 tâches MPI.

Threads. Par défaut, ce thread est peuplé avec un *thread utilisateur* nommé *thread idle* qui exécute une boucle d'attente active en attendant une tâche à exécuter. Comme rien n'est prévu pour ce thread, il va entraver le CPU avec de l'attente active. La conséquence est de ralentir les threads qui partagent le même cœur sur l'autre *Hyper-Thread*.

Pour évaluer ce comportement, nous avons inséré un appel à la fonction *usleep* de $2\,\mu s$ pour diminuer l'impact de l'attente active sur le *thread idle*.

Avec cette version, appelé « smt-sleep » sur la figure 4.3.4, nous observons une amélioration de t_{ovrl} par un facteur de $1,42$ par rapport à la configuration par défaut de MPC (no-smt-bind) et un taux de recouvrement de 98%. Dans cette version, t_{cpu} est très peu impacté, cela montre que cette amélioration atténue la contention entre les communications et le calcul. En effet, depuis que le *thread idle* est endormi la plupart du temps, le thread de calcul n'est plus ralenti et le temps de calcul revient à la normale. Cependant, quand nous avons besoin de faire de la progression, les appels à la fonction *usleep* réduisent les performances de la progression et le temps de communication augmente. Par conséquent, il est possible de trouver un compromis.

En résumé, les ralentissements observés lors des expériences étant dus à l'ordonnancement des threads dans MPC et non aux threads de progression, placer les threads de progression sur les *Hyper-Threads* améliore le recouvrement ainsi que le temps d'exécution pour les communications inter-nœuds passant par le réseau. Cela soulage le besoin de cœurs supplémentaires pour faire progresser les communications. Dans l'idéal, il faudrait avoir une implémentation sans attente active. Il reste à patcher MPC pour que le *thread idle* fasse une attente passive.

4.3.3 Utilisation des Hyper-Threads pour les communications intra-nœuds

La façon courante d'effectuer les communications intra-nœud est de faire des copies de *buffers* depuis la source vers sa destination. Pour une implémentation MPI à base de processus, tels qu'Open MPI, MPICH, MVAPICH, Intel-MPI ou Mad-MPI, cela peut se faire avec un segment de mémoire partagée entre tous les processus du même nœud. Cette technique permet à tous les rangs MPI de copier les *buffers* directement sur le segment de mémoire partagée pour faire des transferts de messages. Néanmoins, cela nécessite deux copies de *buffers* (source vers segment de mémoire partagée puis segment de mémoire partagée vers destination) au lieu d'une seule pour les implémentations MPI à base de threads.

Dans la version à base de threads de MPC, toutes les tâches MPI sont des threads. Cela implique que toute la mémoire est partagée au sein d'un processus MPC. Par conséquent, toutes les tâches MPI ainsi que tous les threads de

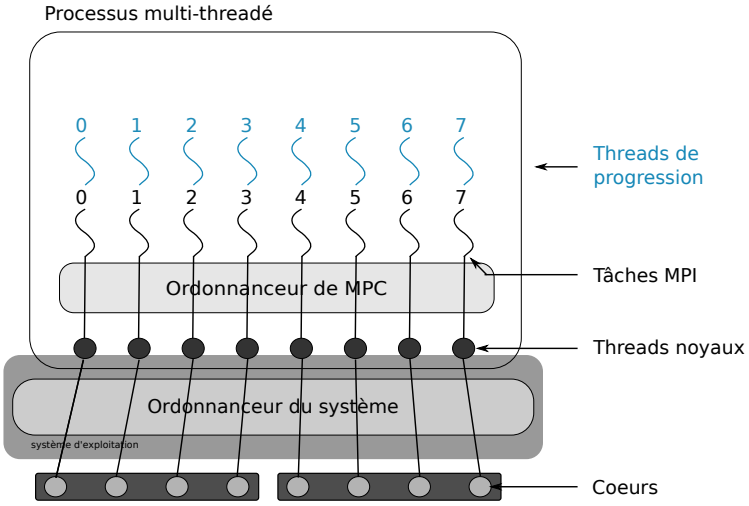


FIGURE 4.3.5 – Placement « no-smt-bind »

progression se partagent le même espace d'adressage. La copie de *buffer* est donc directement réalisée avec un appel à la fonction *memcpy*.

Nous exécutons notre suite de tests sur un seul nœud Haswell, avec une tâche MPI par cœur. Nous testons deux configurations de placement différentes : le placement « no-smt-bind » (figure 4.3.5) et le placement « smt » (figure 4.3.6) que nous avons déjà présenté dans la sous-section 4.3.1 à la page 64.

Pour chaque configuration, nous mesurons le temps de calcul (t_{cpu}), le temps de communications (t_{comm}) et le temps total de l'exécution (t_{ovrl}) quand le calcul et les communications sont exécutés en parallèle.

Les résultats sont illustrés sur la figure 4.3.7. Pour les deux configurations de placement « no-smt-bind » et « smt », nous observons que $t_{ovrl} = t_{cpu} + t_{comm}$, ce qui veut dire que nous n'avons aucun recouvrement. Nous observons aussi une augmentation de 44% sur le temps total d'exécution quand nous plaçons les threads de progression sur les *Hyper-Threads*. Cela est dû à une augmentation très importante du temps de calcul. C'est un comportement complètement différent de celui que nous avons observé pour les communications inter-nœuds car dans le cas inter-nœuds, nous avons un taux de recouvrement supérieur à 90% alors que dans le cas intra-nœud, nous avons un taux de recouvrement de 0%.

Ces observations nous montrent clairement que placer les threads de progression sur les *Hyper-Threads* a un énorme impact sur les performances du thread de calcul quand les communications sont en mémoire partagée, car les échanges de messages sont des copies mémoire qui utilisent le CPU. Donc les threads de communications s'exécutant sur les *Hyper-Threads* perturbent l'exécution des threads de calcul.

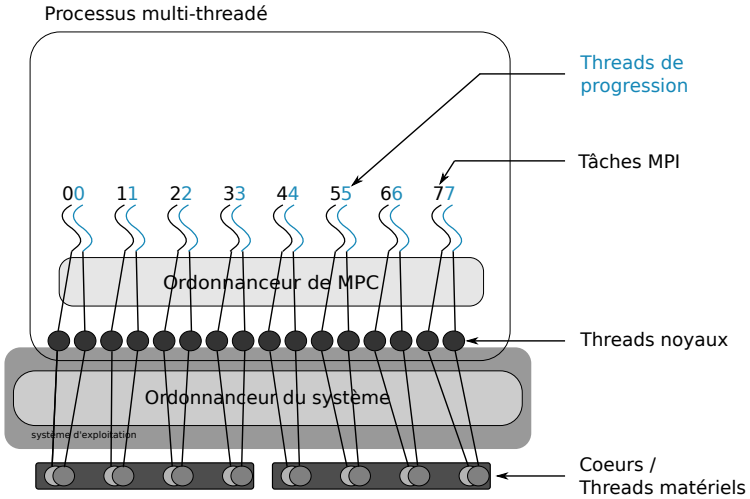
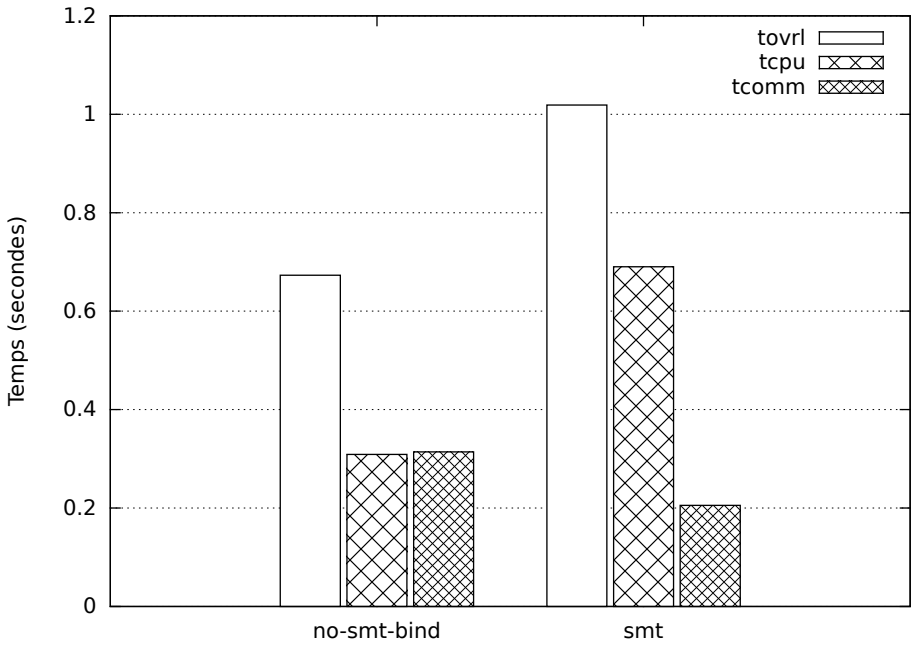


FIGURE 4.3.6 – Placement « smt »



Taux de recouvrement :

— no-smt-bind : 0%
— smt : 0%

FIGURE 4.3.7 – t_{ovrl}, t_{comm} et t_{comp} pour une opération *Ialltoall* avec un *buffer* de taille constante de 2 Mo sur 1 nœud avec 32 tâches MPI.

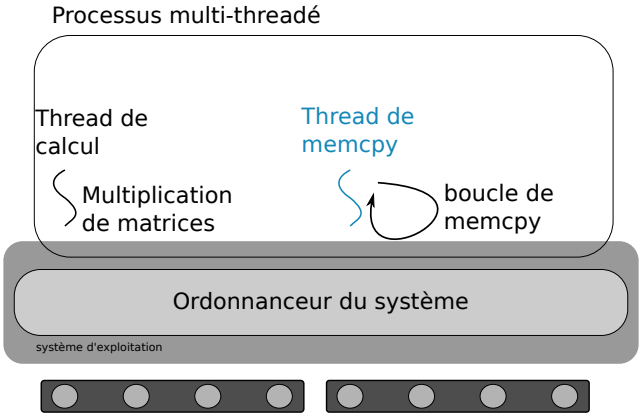


FIGURE 4.3.8 – Illustration du micro-test

4.3.4 Influence des effets de cache lors de l'utilisation des Hyper-Threads

Nous cherchons à savoir pourquoi le placement des threads de progression sur les *Hyper-Threads* a un impact négatif sur les performances du thread de calcul s'exécutant sur le même cœur. Nous nous sommes concentrés sur les effets de cache provoqués par la compétition entre les *Hyper-Threads* du même cœur voulant occuper la même ligne de cache. Cet effet est connu sous le nom de *cache thrashing* [50].

Nous avons implémenté un micro-test pour confirmer nos hypothèses sur les effets de caches provoqués quand l'*Hyper-Threading* est utilisé pour faire la progression des communications intra-nœuds. Afin d'isoler le problème, nous avons exécuté ce test en dehors du framework MPC. Dans ce test, un thread exécute une multiplication de matrice 1024×1024 que nous appellerons *thread de calcul*. Un autre thread est créé pour simuler la progression des communications en intra-nœuds en exécutant une boucle d'appel à la fonction *memcpy* que nous appellerons *thread de memcpy* (figure 4.3.8). Nous nous concentrons sur l'impact du *thread de memcpy* sur le *thread de calcul*.

Nous testons trois configurations de placement différentes :

— « cache-not-shared » : le *thread de calcul* est lié sur un cœur et le *thread de memcpy* est lié sur un autre socket. Ces deux threads ne partagent aucune ligne de cache (figure 4.3.9).

— « no-smt-bind » : le *thread de calcul* est lié sur un seul cœur et le *thread de memcpy* est lié sur le même cœur. L'*Hyper-Threading* est désactivé (figure 4.3.10).

— « smt » : le *thread de calcul* est lié sur un seul cœur et le *thread de memcpy* est lié sur le même cœur mais sur un autre Hyper-Thread (figure 4.3.11).

Pour chaque configuration, nous exécutons nos tests avec trois tailles de *buffers* différentes pour le *thread de memcpy*, $4\,Ko$, $128\,Ko$ et $2\,Mo$ sur un

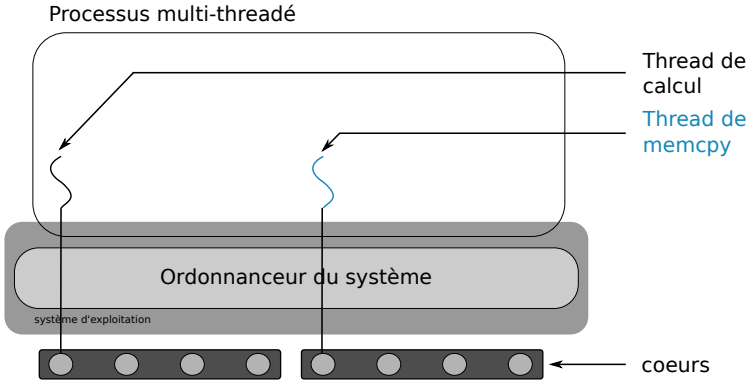


FIGURE 4.3.9 – Placement « cache-not-shared »

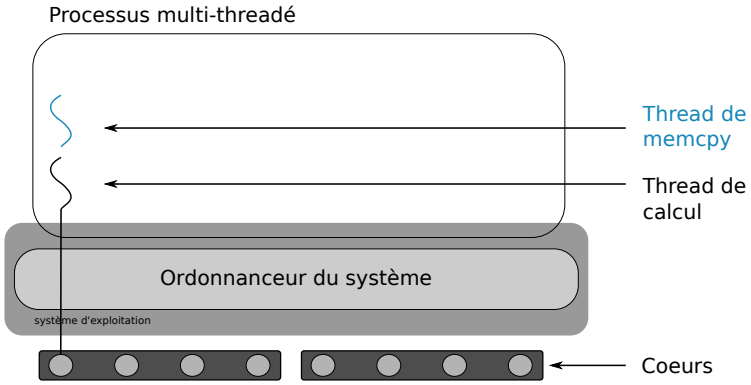


FIGURE 4.3.10 – Placement « no-smt-bind »

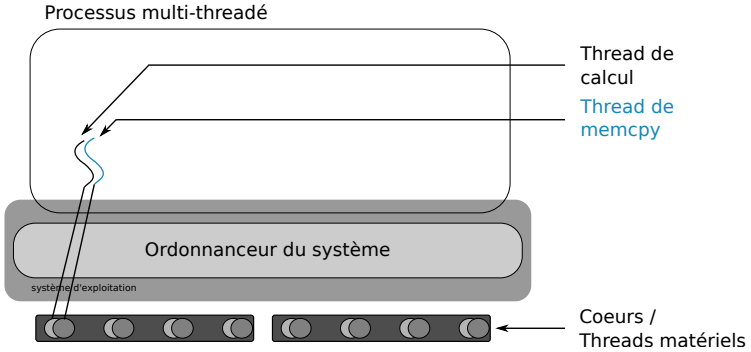


FIGURE 4.3.11 – Placement « smt »

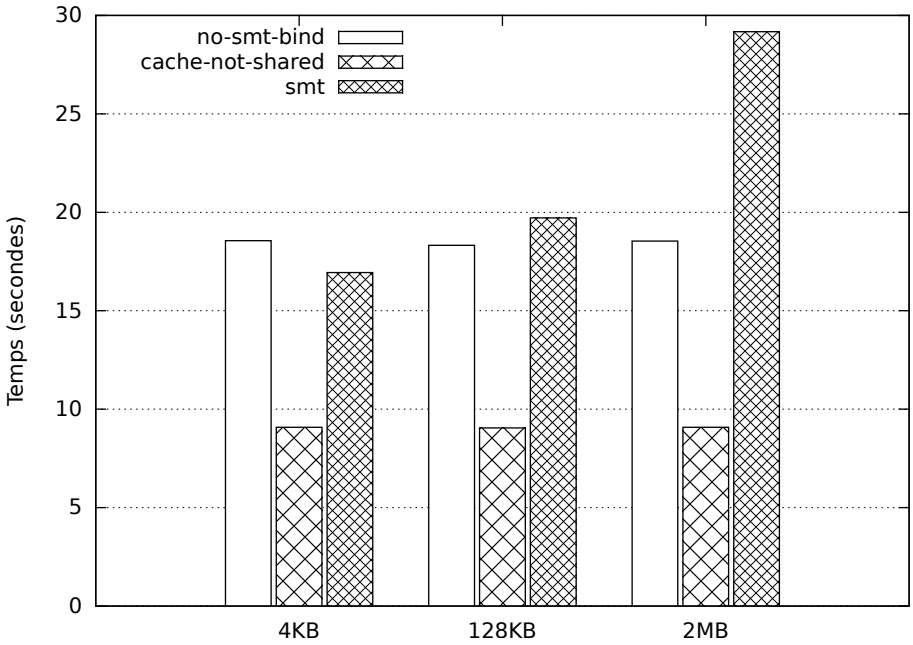


FIGURE 4.3.12 – Temps d'une multiplication de matrice 1024×1024 pour des tailles de *buffers* de $4\,Ko$, $128\,Ko$ et $2\,Mo$ sur un processeur dual socket Haswell avec 16 cœurs par socket et 2 *Hyper-Threads* par cœur

processeur bi-socket Haswell avec 16 cœurs par socket et 2 *Hyper-Threads* par cœur.

Nous mesurons le temps de calcul pour ces trois configurations de placement avec différentes tailles de *buffers*. Nous observons sur la figure 4.3.12 que pour toutes les tailles de *buffers*, nous obtenons un temps d'exécution de $9\,s$ avec le placement « cache-not-shared ». Ce temps double quand nous utilisons le placement « no-smt-bind ». La raison est que nous utilisons deux fois plus de cœurs pour le cas « cache-not-shared ». Le *thread de calcul* et le *thread de memcpy* ne partagent pas le même cœur. Ainsi, nous avons un taux de recouvrement de 100%. De plus, les deux premiers placements ne sont pas en compétition pour l'accès aux caches. Dans le premier cas, le *thread de calcul* et le *thread de memcpy* ne sont pas sur le même socket. Dans le second cas, les deux threads sont sur le même cœur mais sans l'utilisation de l'*Hyper-Threading*, les deux threads s'exécutent l'un après l'autre en effectuant des changements de contexte. Par conséquent, même si des données peuvent être supprimées des lignes de cache après un changement de contexte, la compétition pour les lignes de caches entre les deux threads qui sont exécutés est à la granularité du changement de contexte.

Pour le placement « smt », le *thread de calcul* est lié sur le même cœur que le thread *thread de memcpy* mais sur un autre *Hyper-Threads* et nous

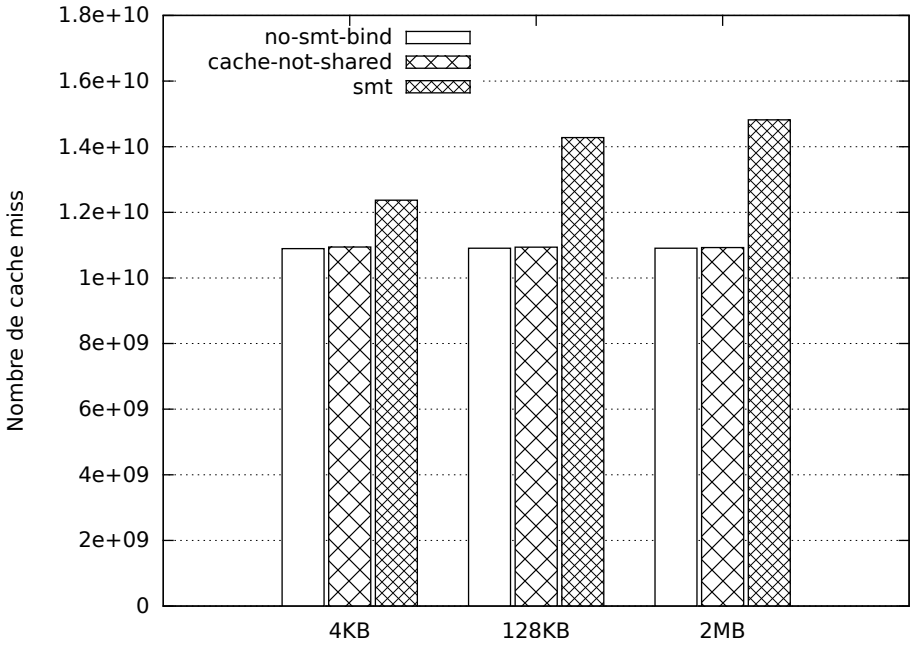


FIGURE 4.3.13 – Nombre de cache L1 miss pour des tailles de *buffers* de 4 Ko, 128 Ko and 2 Mo sur un processeur dual socket Haswell avec 16 cœurs par socket et 2 Hyper-Threads par cœur

observons un comportement différent. Le temps d'exécution augmente quand la taille du *buffer* augmente tandis que le temps d'exécution demeure constant entre les tailles de *buffers* pour les autres placements (« cache-not-shared » et « no-smt-bind »).

L'augmentation du temps de calcul lorsqu'un thread manipule des données de plus en plus grande est le symptôme typique du *cache thrashing* résultant de la compétition pour la même ligne de cache à la granularité de l'instruction car les deux threads partagent les mêmes lignes de cache.

Pour évaluer cette hypothèse, nous utilisons l'outil *Performance Application Programming Interface* (PAPI) [51] pour collecter le nombre de *cache miss* sur le cache de niveau 1. Nous observons dans la figure 4.3.13 que le nombre de *cache miss* est constant entre les placements « no-smt-bind » et « cache-not-shared ». C'est le comportement attendu parce que le *thread de calcul* et le *thread de memcpy* ne partagent pas les caches pour le placement « cache-not-shared ». Pour le placement « no-smt-bind », ces threads sont ordonnancés l'un après l'autre et aucun *cache miss* additionnel n'apparait.

Pour le placement « smt », nous observons des *cache miss* additionnels comparé aux deux stratégies de placement précédentes. Cela est dû à l'utilisation de l'*Hyper-Threading*. Les deux threads sont exécutés sur le même cœur simultanément utilisant le même bus mémoire et partageant le même cache.

Chacun des threads a besoin de récupérer une ligne de cache pour exécuter son travail. Cependant, il y a de la contention qui apparait et conduit à des *cache miss* additionnels, car le *thread de memcpy* évince les lignes de cache du *thread de calcul* et vice versa.

Ces résultats expliquent pourquoi l'utilisation des *Hyper-Threads* pour la progression des communications en mémoire partagée dégradent les performances du calcul s'exécutant sur le même cœur.

4.4 Conclusion

Dans ce chapitre, nous nous sommes intéressés plus particulièrement au placement des threads de progression dans le but d'améliorer le recouvrement ainsi que le temps d'exécution d'une application. Nous avons présenté notre propre suite de test permettant de fixer la taille du problème pour garder une charge de travail constante avec un nombre différent de tâches MPI.

Plusieurs algorithmes de placement de threads tenant compte des effets NUMA et améliorant le recouvrement des communications liées aux collectives MPI non-bloquantes ont été proposés. Nous avons comparé les taux de recouvrement de plusieurs bibliothèques MPI entre elles afin de savoir lesquelles proposaient un bon recouvrement des communications par du calcul.

Ensuite, nous nous sommes concentrés sur l'évaluation du placement des threads de progression sur les *Hyper-Threads*. Cela nous a permis de voir que l'utilisation de cette technologie est efficace pour les communications inter-nœuds mais qu'au contraire, son utilisation pour les communications intra-nœuds dégrade les performances. Nous avons expliqué que cela est dû au *cache thrashing*.

Le placement le plus efficace des threads de progression effectuant des communications inter-nœuds et des communications intra-nœuds n'est pas le même. Pour les communications inter-nœuds, utiliser les Hyper-Threads est une bonne solution. En revanche, pour les communications intra-nœuds, des cœurs doivent être dédiés pour les threads de progression. C'est pourquoi nous nous intéressons particulièrement aux communications intra-nœud dans le chapitre suivant.

Chapitre 5

Placement dynamique des threads de progression en fonction des algorithmes de collectives utilisés

Sommaire

Dans ce chapitre, nous proposons deux algorithmes de placement dynamique des threads de progression en intra-nœud dans le but d'améliorer le recouvrement des communications. Le premier se concentre sur les opérations collectives basées sur un arbre de communication. Le second vise à optimiser les opérations basées sur les communications en chaîne.

Nous avons vu une manière de faire progresser les communications collectives MPI non-bloquantes en proposant des algorithmes de placement des threads de progression de manière statique. Nous avons vu qu'il était possible d'utiliser les Hyper-Threads pour réaliser la progression des communications inter-nœuds.

En revanche, la progression des communications en intra-nœud se révèle plus compliquée.

5.1 L'algorithme « split-tree » pour les collectives MPI non-bloquantes en arbre

Dans cette section, nous proposons un algorithme permettant d'améliorer le recouvrement des communications collectives MPI non-bloquantes pour les opérations basées sur un arbre de communication (broadcast, reduce, scatter, gather, allreduce). L'idée est de répartir les communications entre les cœurs dédiés à la progression des communications et les cœurs alloués à l'application en fonction de l'étape de l'algorithme de collective basé sur un arbre.

5.1.1 L'algorithme « split-tree »

L'algorithme « split-tree » s'applique sur les communications intra-noeuds sur une machine manycore. Pour recouvrir les communications par du calcul, les communications et le calcul doivent s'exécuter en parallèle. Pour cela, les threads de calcul et les threads de progression doivent s'exécuter sur des cœurs indépendants, car les échanges de messages sont des copies mémoires et utilisent le CPU comme nous l'avons vu dans le chapitre 4. Notre proposition est de faire progresser les communications en arrière-plan en dédiant certains cœurs aux communications car l'impact de la perte de quelques cœurs pour le calcul est négligeable sur ce type de microprocesseur. Ainsi, quelques cœurs exécutent les tâches MPI (appelés par la suite « cœurs applicatifs »), tandis que les cœurs restants seront dédiés aux threads de progression (appelés par la suite « cœurs de communication »).

Cependant, les algorithmes de communications collectives nécessitent une quantité élevée de communications point-à-point. Quand les threads de progression placés sur les cœurs de communication exécutent toutes les communications au nom de toutes les tâches MPI s'exécutant sur les cœurs applicatifs, l'arbre de communication est *replié* sur les cœurs de communication et les communications provenant d'une étape donnée de l'algorithme peuvent être sérialisées. Par conséquent, quand nous replions les threads de progression sur peu de cœurs de communication, les communications collectives s'exécutent plus lentement que lorsqu'elles sont exécutées comme une collective bloquante sur tous les cœurs applicatifs.

Les étapes d'une collective basée sur des communications en arbre binomial sont représentées sur l'exemple de la figure 5.1.1. Chaque niveau de l'arbre est une étape de l'algorithme, des feuilles vers la racine. Le rang de chaque tâche MPI à différentes étapes est représenté par les sommets. Les sommets reliés par une arête en pointillés représentent les mêmes tâches MPI. Seuls les

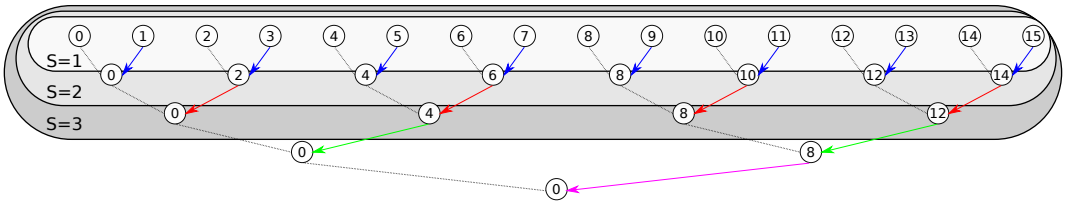


FIGURE 5.1.1 – L'arbre de communication pour la collective reduce avec 16 rangs MPI. S est le nombre d'étapes (niveau de profondeur de l'arbre) qui s'exécute sur les cœurs applicatifs. Les arcs pleins représentent les communications. Les sommets représentent les rangs MPI.

arcs pleins et colorés impliquent une communication. Sur l'exemple présenté en figure 5.1.1, pour 16 tâches MPI, nous avons 15 communications réparties sur 4 étapes représentées par les arcs bleus, rouges, verts et magenta. Si nous replions ces communications sur un seul cœur de communication, nous aurons 15 étapes, ce qui est 4 fois plus lent. Si nous replions ces communications sur 2 cœurs de communication, nous aurons 8 étapes, ce qui est 2 fois plus lent. La figure 5.1.2 illustre le deuxième cas.

Sur les algorithmes basés sur une topologie en arbre, nous observons que le nombre de communications est très irrégulier suivant les étapes de l'algorithme. Une partie importante des communications s'effectue à la première étape, représenté par $S = 1$ avec les étapes numérotées depuis les feuilles. S'il n'y a que les communications de la première étape qui s'exécutent sur les cœurs applicatifs, et que le reste des communications s'exécute sur les cœurs de communication, le temps nécessaire est 2 fois moins élevé que de tout exécuter sur les cœurs de communication. Cependant, la première étape de l'algorithme ne peut pas être recouverte par du calcul. Nous sacrifions donc du recouvrement potentiel pour avoir un nombre d'étapes moins important et, à priori un meilleur temps d'exécution.

L'algorithme que nous proposons est une généralisation de ce principe pour avoir un compromis entre les performances des communications et le recouvrement : scinder l'arbre de communication avec une partie s'exécutant sur tous les cœurs applicatifs pour bénéficier du parallélisme et une autre partie sur les cœurs de communication pour bénéficier du recouvrement des communications. La figure 5.1.3 illustre ce comportement pour deux étapes de l'arbre s'exécutant sur les cœurs applicatifs ($S = 2$).

Soit S le nombre d'étapes (étages de l'arbre) s'exécutant sur les cœurs applicatifs. $S = 0$ est équivalent à exécuter toutes les communications sur les cœurs de communication. L'algorithme exécute S étapes de l'arbre sur les cœurs applicatifs comme décrit dans la figure 5.1.1. Quand $S = 1$, l'algorithme exécute la plus petite partie en nombre d'étapes, mais la plus lourde en communications sur les cœurs applicatifs. En revanche, la partie la plus longue en nombre d'étapes mais la plus légère en communications est exécutée sur un ou plusieurs

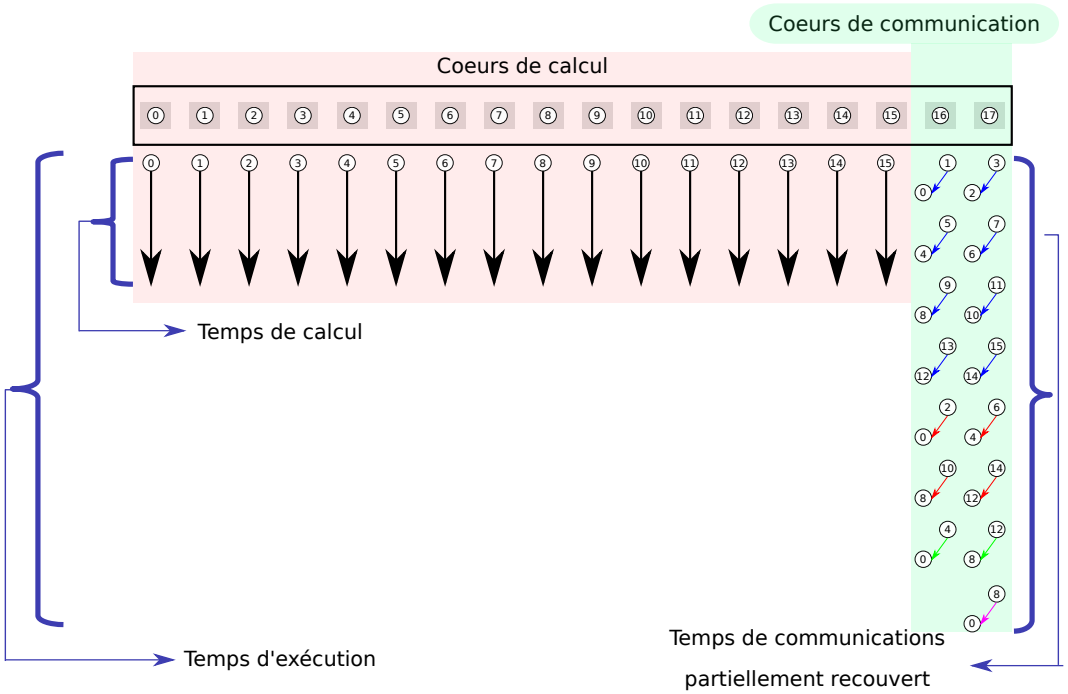


FIGURE 5.1.2 – Exemple pour la collective reduce avec 16 rangs MPI les cœurs applicatifs (zone rouge) et les threads de progression repliés sur les cœurs de communication (zone verte).

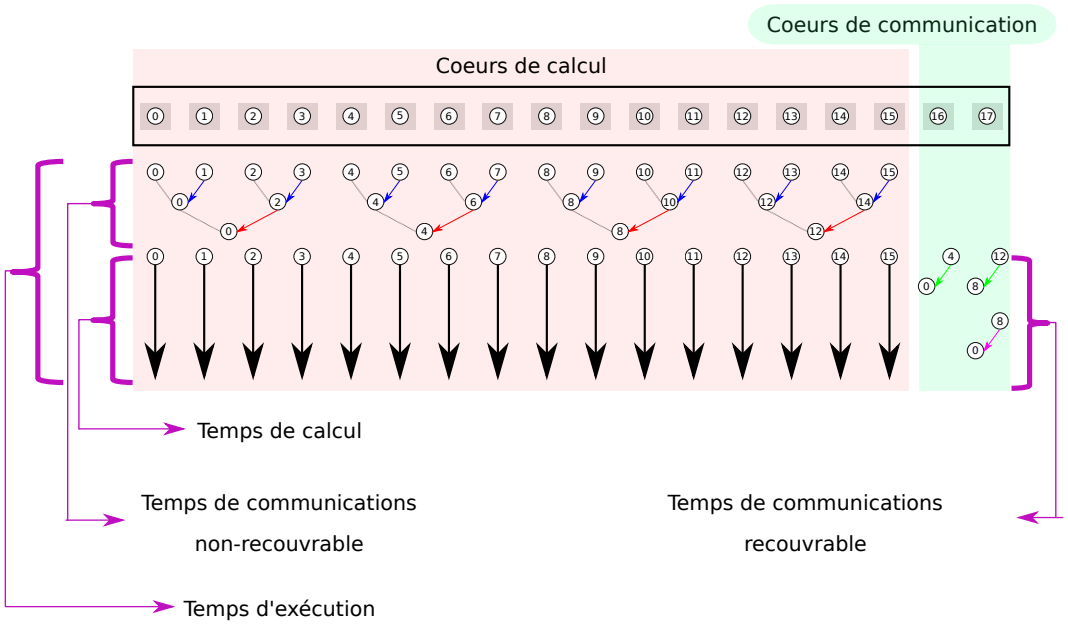


FIGURE 5.1.3 – Exemple de l'algorithme « split-tree » pour la collective reduce avec 16 rangs MPI avec 2 étapes sur les cœurs applicatifs (zone rouge) et 2 étapes sur les cœurs de communication (zone verte).

cœurs de communication. Toutes les communications qui sont exécutées sur les cœurs applicatifs ne bénéficient pas du recouvrement par du calcul, car elles sont exécutées sur le même cœur. Cependant, cette partie de l'arbre est celle qui concentre le plus de communications et l'exécuter sur un petit nombre de cœurs de communication compromettrait les performances des communications. La partie s'exécutant sur les cœurs de communication bénéficie d'un recouvrement total de ses communications.

Si S augmente, l'algorithme perd de sa capacité à recouvrir les communications mais peut augmenter les performances des communications et donc des performances globales de l'application, si le ratio calcul/communications le permet. Nous devons avoir un compromis entre le recouvrement et les performances globales.

5.1.2 Modélisation

Dans cette partie, nous proposons un modèle de performance de l'algorithme décrit dans la section 5.1.1, afin de montrer sa pertinence et d'ajuster ses paramètres.

5.1.2.1 Modèle pour les opérations collectives

Soit N_{proc} le nombre total de cœurs et N le nombre de cœurs applicatifs (c.-à-d. nombre de tâches MPI), alors le nombre de cœurs dédiés aux communications est $P(N) = N_{proc} - N$.

Nous considérons seulement les opérations collectives en arbre. Nous modélisons le coût des communications comme étant linéaire avec la taille des données, car avec des tailles de messages suffisantes pour que le recouvrement ait un sens en intra-noeud, la latence est négligeable et nous négligeons volontairement les effets de cache pour simplifier. Nous prenons comme unité le temps de transfert d'un *buffer* de la taille considérée dans l'opération collective pour une opération point-à-point. Nous avons d'abord étudié les opérations avec une taille de *buffer* constante à travers tout l'arbre de communication (reduce, broadcast). Nous avons ensuite étendu cet algorithme prenant en compte des tailles de *buffer* variables selon les étapes (gather, scatter). La profondeur de l'arbre est donnée par l'équation 5.1.

$$H(N) = \lceil log_2(N) \rceil \tag{5.1}$$

Dans le cas d'une opération bloquante où les communications sont exécutées simultanément par tous les cœurs applicatifs, nous obtenons l'équation 5.2 décrivant le temps d'exécution en fonction du nombre de tâches MPI N.

$$T_{blocking}(N) = H(N) = \lceil log_2(N) \rceil \tag{5.2}$$

Soit $C(N)$ le temps de calcul sur N cœurs. Pour modéliser le calcul et le recouvrement des communications, nous considérons que le développeur de

l'application a essayé d'obtenir un recouvrement parfait des communications, et que la charge de calcul est suffisante pour que le temps du calcul soit équivalent au temps d'une opération collective bloquante sur tous les cœurs, c'est ce qui est décrit par l'équation 5.3.

$$C(N_{proc}) = T_{blocking}(N_{proc}) \tag{5.3}$$

Afin de faire varier le nombre de cœurs disponible pour les tâches MPI exécutant du calcul, nous faisons l'hypothèse que la charge de calcul a une accélération linéaire, le temps de calcul sur N cœurs est représenté par l'équation 5.4.

$$C(N) = \frac{N}{N_{proc}} \times C(N_{proc}) \tag{5.4}$$

5.1.2.2 Modèle pour l'algorithme proposé

Nous proposons deux modèles pour l'algorithme « split-tree ». Le premier modélise l'algorithme « split-tree » avec des tailles de *buffer* fixes (e.g. Reduce). Le second le modélise avec des tailles de *buffer* variables (e.g. Gather).

Modèle avec des tailles de *buffer* fixes : Modélisons l'algorithme « split-tree » en lui-même. Comme défini dans la section 5.1.1, S est le nombre d'étapes s'exécutant sur les cœurs applicatifs. Le temps nécessaire à exécuter ces étapes correspond à S dans notre premier modèle puisque l'unité de temps est la communication élémentaire. Ensuite, l'algorithme ordonnance les opérations de communication des $H(N) - S$ dernières étapes repliées sur les $P(N)$ cœurs de communication. Soit $R(N) = N - 2^{\lfloor log_2(N) \rfloor}$ le nombre de feuilles qui ne sont pas dans le plus grand sous-arbre binaire complet de l'arbre. Soit $F(N, i)$ (équation 5.5) le nombre de communications à exécuter, pour N tâches MPI, pour l'étape i avec i compris entre 1 et $H(N)$.

$$F(N, i) = 2^{\lfloor log_2(N) \rfloor - i} + \left\lfloor \frac{R(N) + 2^{(i-1)}}{2^i} \right\rfloor \tag{5.5}$$

Sur la figure 5.1.4, nous pouvons voir le nombre de communications à effectuer par étape i de l'arbre de communication correspondant aux valeurs de $F(N, i)$ pour un nombre de tâches MPI allant de 1 à 64. Nous pouvons voir qu'à chaque étape, le nombre de communications diminue de moitié comme nous l'expliquions dans la section 5.1.1.

Ces étapes sont déroulées les unes après les autres pour respecter les dépendances entre les communications. Puisque chaque étape i contient $F(N, i)$ communications, le temps de communication prend $\lceil F(N, i)/P(N) \rceil$ quand l'étape est repliée sur $P(N)$ cœurs de communication. Le temps d'exécution d'une collective non-bloquante avec notre algorithme qui scinde l'arbre des communications est décrit par l'équation 5.6 suivante :

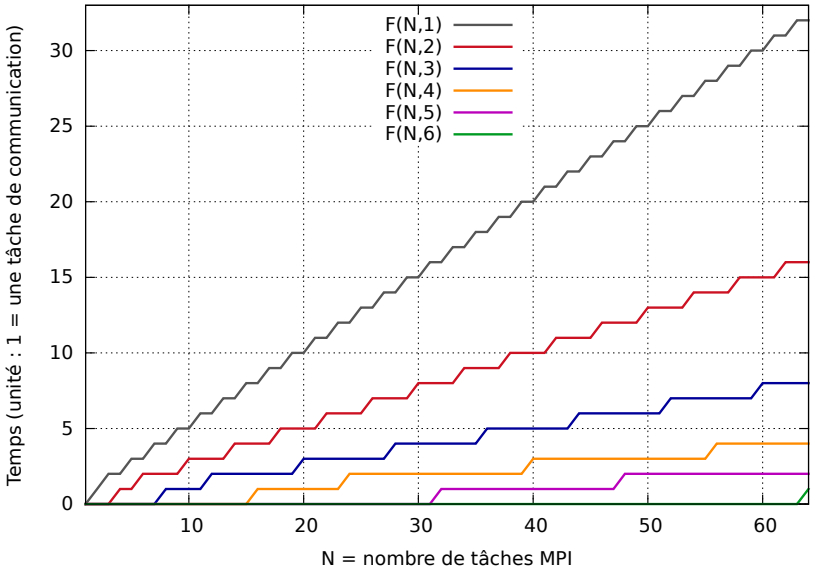


FIGURE 5.1.4 – Le nombre de communications par étape de l'arbre de communication pour une collective en arbre avec N allant de 1 à 64.

$$T_{non-blocking}(S,N) = \underbrace{\min(S,H(N))}_{\text{premières } S \text{ étapes}} + \underbrace{\sum_{i=S+1}^{H(N)} \left\lceil \frac{F(N,i)}{P(N)} \right\rceil}_{\text{dernières } H(N)-S \text{ étapes restantes}} \tag{5.6}$$

Le recouvrement des communications par du calcul provenant d'une collective non-bloquante est effectué seulement sur la partie s'exécutant sur les cœurs de communication. La partie des communications s'exécutant sur les cœurs applicatifs n'est pas recouvrable. Le temps d'exécution ($calcul + communication$) est donné par l'équation 5.7 suivante :

$$T_{overlapped}(S,N) = \underbrace{\min(S,H(N))}_{\text{communications non-recouvrables}} + \underbrace{\max\left(C(N), \sum_{i=S+1}^{H(N)} \left\lceil \frac{F(N,i)}{P(N)} \right\rceil\right)}_{\text{communications recouvrables}} \tag{5.7}$$

Les courbes représentant $C(N)$ (Calcul), $T_{blocking}$ (Comms, algorithme no-overlap), et $T_{non-blocking}(N,S)$ (Comms, split-tree ($S = i$)) pour différentes valeurs de S avec $N_{proc} = 64$ (KNL) sont représentées sur la figure 5.1.5.

Notons d'abord que le temps de calcul n'est pas constant car quand nous augmentons le nombre de tâches MPI N, le temps de calcul diminue pour conserver une charge de travail globale constante pour n'importe quelle valeur de N.

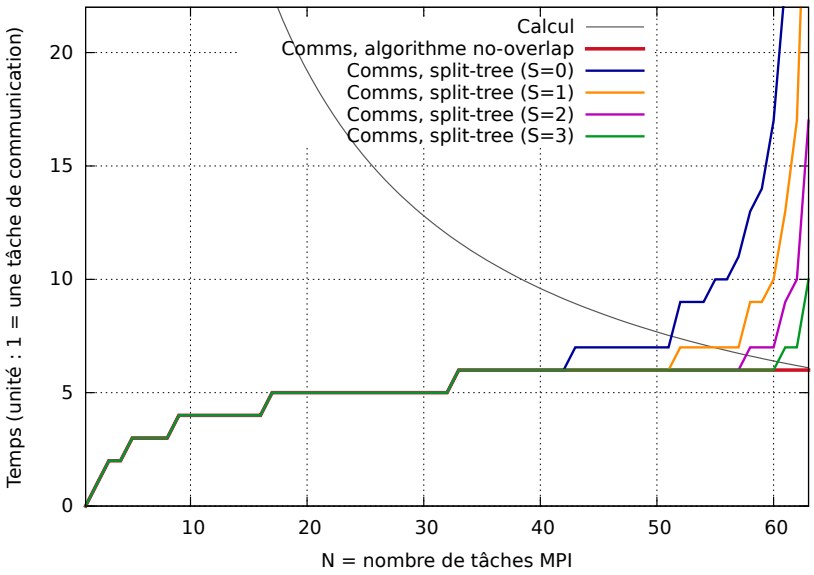


FIGURE 5.1.5 – Modèle de coût des communications avec un *buffer* de taille constante (reduce, bcast) sur 64 cœurs (KNL).

Nous observons que la valeur optimale de N quand toutes les communications s'effectuent sur les cœurs dédiés ($S = 0$) est de 51 et qu'après $N = 56$ le temps de communication devient important, car nous n'avons plus assez de cœurs dédiés aux communications. En effet, le nombre de communications pour $N = 56$ est de 55 ($\sum_{i=1}^{H(56)} \lceil F(56, i) \rceil = 55$) communications à répartir sur seulement 8 cœurs dédiés à la progression. Pour $N = 57$, nous aurons plus de communications ($\sum_{i=1}^{H(57)} \lceil F(57, i) \rceil = 56$) à répartir sur moins de cœurs dédiés ($P(57) = 7$). C'est pourquoi, pour des grandes valeurs de N, le temps estimé est très élevé pour $S = 0$ car toutes les communications sont réalisées sur peu de cœurs de communication. Le coût des communications se réduit quand le nombre d'étapes S de l'arbre de communications augmente. En effet, le nombre de communications à effectuer sur les cœurs dédiés dépend du nombre d'étapes effectués sur les cœurs applicatifs. Quand $S = 1$, le nombre de communications à exécuter sur les cœurs dédiés est divisé par deux, car la première étape de l'arbre de communication est effectuée sur les cœurs applicatifs.

Sur la figure 5.1.6, le temps total du recouvrement calcul/communications est représenté dans le cas où le calcul et les communications s'effectuent sans aucun recouvrement (Calc + comms, algorithme no-overlap), puis dans le cas où nous utilisons notre algorithme avec différentes valeurs de S (Calc + comms, split-tree ($S = i$)).

Nous observons que, pour les petites valeurs de N, augmenter la valeur de S augmente le temps d'exécution du *calcul* + *communications*. Ceci est dû à l'exécution de S étapes de l'arbre de communication sur les cœurs applicatifs.

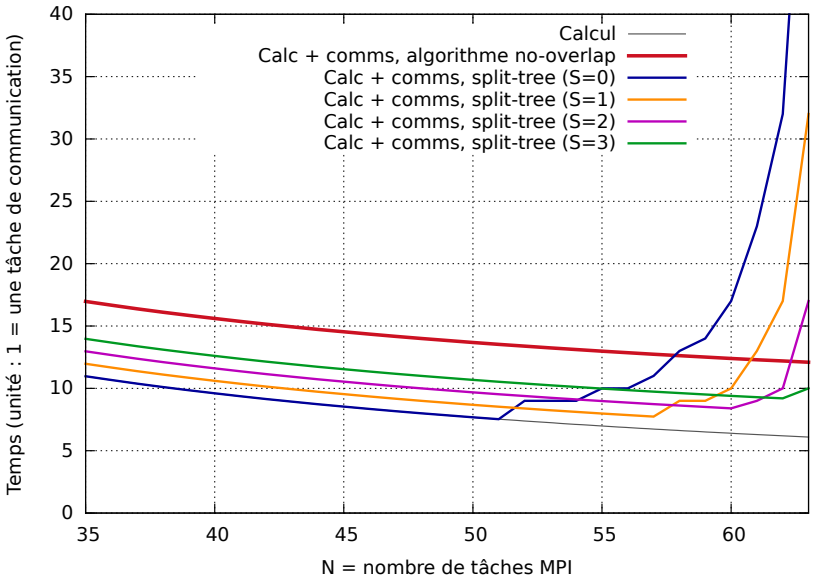


FIGURE 5.1.6 – Modèle de coût du recouvrement calcul/communications avec un *buffer* de taille constante (reduce, bcast) sur 64 cœurs (KNL).

Ces communications ne sont pas recouvrables par du calcul. Le temps de ces communications est donc ajouté au temps de calcul et explique l'augmentation du temps du $calcul + communications$. Néanmoins, ce coût est amorti pour les grandes valeurs de N où le temps total est dominé par le coût des communications repliées sur peu de cœurs de communication. Nous pouvons donc observer que l'algorithme « split-tree » permet de conserver des bonnes performances en réservant peu de cœurs pour les threads de progression et qu'il existe une valeur de S optimale en fonction de N. En effet, nous pouvons choisir le nombre d'étapes S en fonction du nombre de tâches MPI N et du nombre de cœurs dédiés aux threads de progression ($P(N)$). Pour cela, il nous suffit de calculer le temps donné par l'équation 5.7 pour les valeurs de S possibles entre 0 et $log_2(N)$ et de choisir la meilleure valeur de S possible pour chaque valeur de N (et donc $P(N)$). Cela correspond au croisement des courbes représentants les temps de communication avec la courbe représentant le calcul sur la figure 5.1.5.

Le modèle sur Skylake est représenté sur la figure 5.1.7. On y observe le même comportement malgré le fait que le Skylake n'a que 48 cœurs comparé au KNL qui en a 64.

Modèle avec des tailles de *buffer* variables : Nous pouvons étendre le modèle proposé aux opérations collectives avec un poids non constant sur les arêtes, telles que gather et scatter. Le poids des arêtes double à chaque étape de l'arbre en partant des feuilles vers la racine.

Soit $W_H(N) = \sum_{i=1}^{H(N)} 2^{(i-1)}$ le poids total des communications avec le poids

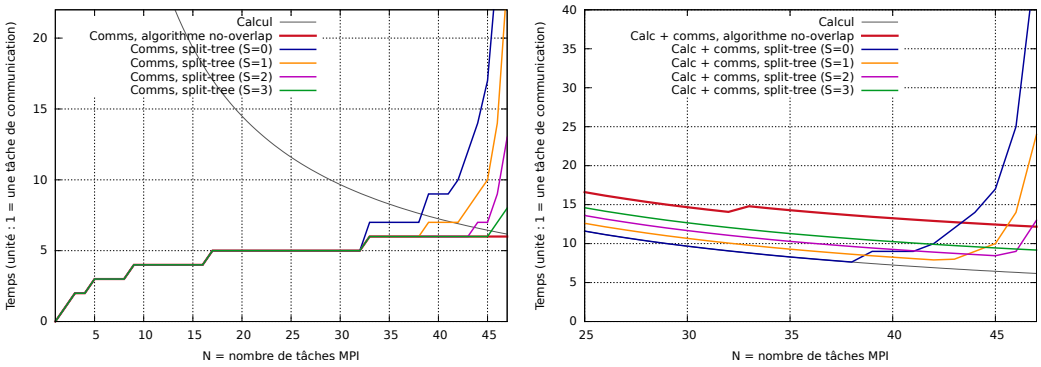


FIGURE 5.1.7 – Modèle de coût des communications (à gauche) et du recouvrement calcul/communications avec un *buffer* de taille constante (reduce, bcast) sur 48 cœurs (Skylake).

des arêtes qui double à chaque étape.

Soit $W(N,S) = \sum_{i=1}^{S} 2^{(i-1)}$ le poids de l'étape 1 à S avec le poids des arêtes qui double à chaque étape.

Le temps d'exécution d'une collective non-bloquante avec notre algorithme qui scinde l'arbre des communications avec des tailles de *buffer* variables est donc décrit par l'équation 5.8 suivante :

$$T_{non-blocking}(S,N) = \underbrace{min(W(N,S), W_H(N))}_{\text{premières } S \text{ étapes}} + \underbrace{\sum_{i=S+1}^{H(N)} \left\lceil \frac{F(N,i)}{P(N)} \right\rceil \times 2^{(i-1)}}_{\text{dernières } H(N)-S \text{ étapes restantes}} \tag{5.8}$$

L'équation 5.9 décrit le temps d'exécution pendant le recouvrement des communications par du calcul provenant d'une collective non-bloquante (gather ou scatter). Le recouvrement est effectué seulement sur la partie s'exécutant sur les cœurs de communication. La partie des communications s'exécutant sur les cœurs applicatifs n'est pas recouvrable.

$$T_{overlapped}(S,N) = \underbrace{\min(W(N,S), W_H(N))}_{\text{communications non-recouvrables}} + \underbrace{\max\left(C(N), \sum_{i=S+1}^{H(N)} \left\lceil \frac{F(N,i)}{P(N)} \right\rceil \times 2^{(i-1)}\right)}_{\text{communications recouvrables}} \tag{5.9}$$

Ce modèle est très similaire au modèle précédent avec un *buffer* de taille fixe. Nous pouvons voir le modèle de coût des communications sur la figure 5.1.8 et le modèle de coût du recouvrement calcul/communication sur la figure 5.1.9.

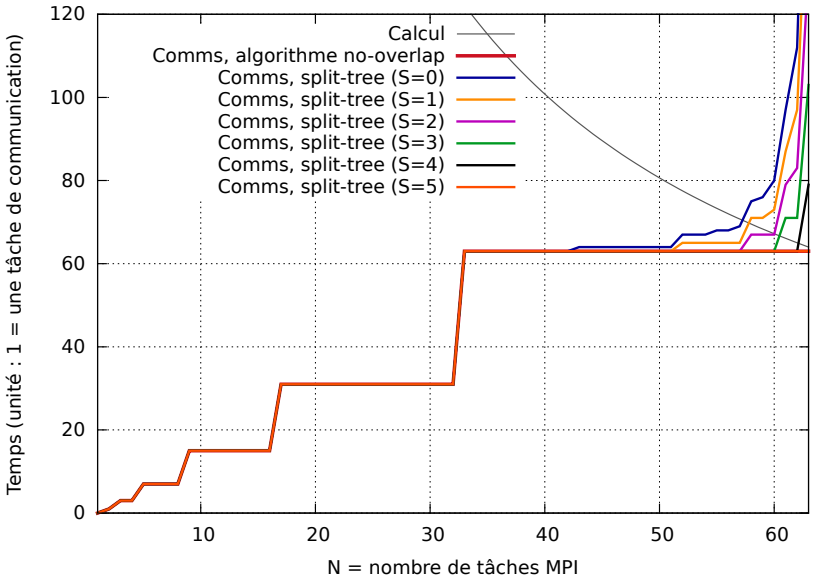


FIGURE 5.1.8 – Modèle de coût des communications avec un *buffer* de taille variable (gather, scatter) sur 64 cœurs.

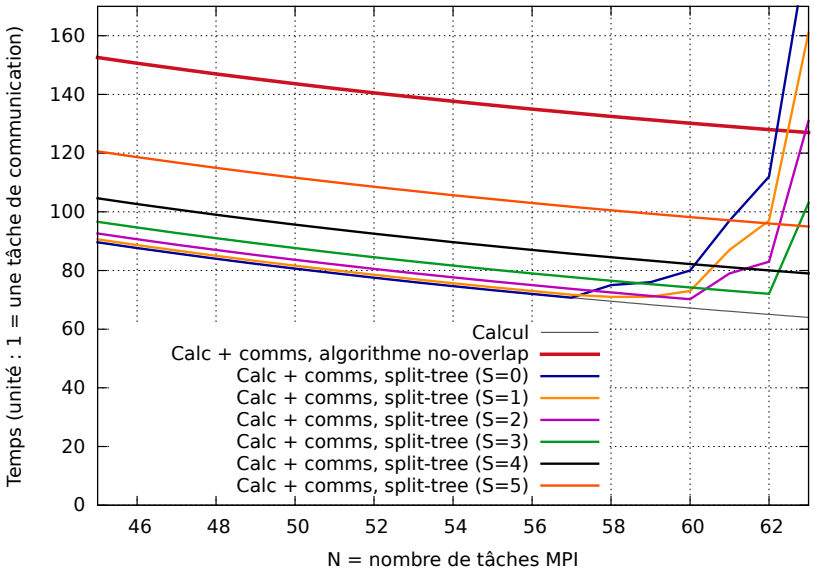


FIGURE 5.1.9 – Modèle de coût du recouvrement calcul/communications avec un *buffer* de taille variable (gather, scatter) sur 64 cœurs.

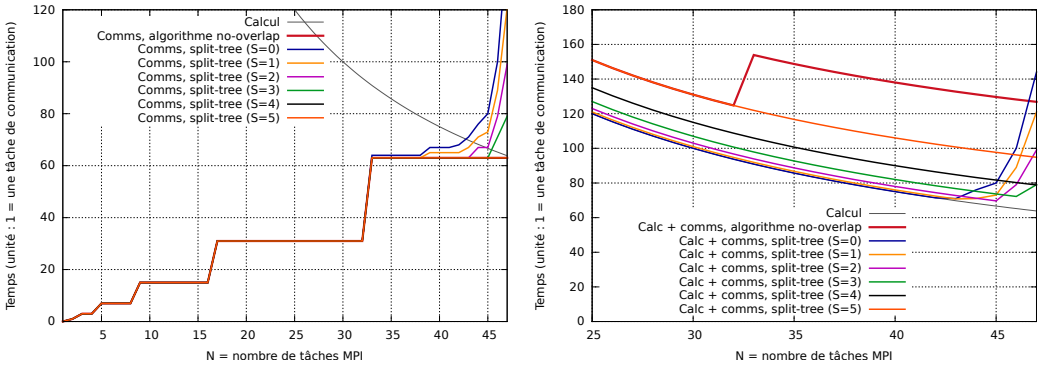


FIGURE 5.1.10 – Modèle de coût des communications (à gauche) et du recouvrement calcul/communications avec un *buffer* de taille constante (gather, scatter) sur 48 cœurs (Skylake).

Le modèle sur Skylake est représenté sur la figure 5.1.10. On y observe le même comportement.

5.1.2.3 Conclusion

Nous proposons un modèle de coût pour l'algorithme « split-tree ». Ce modèle nous permet de prédire les performances du temps d'exécution *calcul* + *communications* quand nous scindons l'arbre de communication pour exécuter un nombre d'étapes S sur les cœurs applicatifs et un nombre d'étapes $H(N) - S$ sur les cœurs de communications. Étant donné que le modèle ne dépend que de la valeur de N et pas des performances du matériel, celui-ci peut-être implémenté directement dans le code sans avoir besoin d'échantillonnage des performances préalables.

5.1.3 Implémentation

Cet algorithme a été implémenté dans MPC. Le fonctionnement de la progression des communications a été abordé dans la section 3.3.2.3. Nous avons vu qu'un thread de progression était créé pour chaque tâche MPI. Ainsi chaque tâche MPI peut donner des communications à effectuer à son thread de progression dédié par le biais d'une structure de donnée appelé *schedule*. Les threads de progression et les tâches MPI sont placés suivant un algorithme de placement que nous avons défini dans la section 4.2. Nous avons donc les tâches MPI qui sont placées sur les cœurs de manière à les écarter au maximum au sein du même nœud NUMA pour assurer un bon placement des threads de progression. Les threads de progression sont placés sur les cœurs libres les plus proches.

Pour implémenter nos algorithmes, nous définissons le paramètre S correspondant aux niveaux de l'arbre de communications que nous souhaitons voir

s'exécuter sur les cœurs applicatifs. Trois cas se présentent alors :
- Les algorithmes Tous-vers-Un (reduce, gather).
- Les algorithmes Un-vers-Tous (broadcast, scatter).
- Les algorithmes Tous-vers-Un-vers-Tous (allreduce).

Tous-vers-Un : Pour ces algorithmes, nous exécutons les premiers S étages de l'arbre de communication directement par les tâches MPI en utilisant des communications point-à-point bloquantes. Ensuite, chaque tâche MPI crée une structure *schedule* pour les $H(N) - S$ étapes restantes. Enfin, nous envoyons les *schedules* créés par les tâches MPI à leur thread associé. Ainsi, la première partie de l'arbre de communication est exécutée dans les cœurs applicatifs et la seconde partie de l'arbre est exécutée dans les cœurs de communication.

Un-vers-Tous : Pour ces algorithmes, nous commençons par créer les structures *schedules* pour les $H(N)-S$ premières étapes de l'arbre de communication et nous les envoyons aux threads de progression. Enfin, nous implémentons les S dernières étapes de l'arbre de communication dans la fonction MPI_Wait qui est exécutée par la tâche MPI. Ainsi, la première partie de l'arbre de communication est exécutée dans les cœurs de communication et la seconde partie de l'arbre est exécutée dans les cœurs applicatifs.

Une optimisation possible lors l'appel à MPI_Wait serait de prendre en charge toutes les communications en cours afin de ne pas attendre l'exécution des S premières étapes sur peu de cœurs dédiés.

Tous-vers-Un-vers-Tous : Il s'agit là d'implémenter l'algorithme allreduce. Nous l'avons implémenté très simplement. Nous appelons nos deux algorithmes précédents. C'est-à-dire que nous avons implémenté notre allreduce avec les fonctions reduce et broadcast que nous avons modifié pour implémenter nos algorithmes.

5.1.4 Résultats expérimentaux

Nous avons utilisé notre suite de tests présentée dans la section 4.1.3. Concernant les architectures de tests utilisées, il s'agit d'un Intel Xeon Phi Knights Landing à 1.4GHz avec 64 cœurs (KNL) et d'un bi-socket Intel Xeon Platinum Skylake à 2.7GHz avec 48 cœurs.

Comparaison de l'algorithme « split-tree » au placement « numa ». Replier les threads de progression sur des cœurs dédiés apporte de bonnes performances quand le nombre de cœurs dédiés est important. Cependant, les performances se dégradent quand trop de threads de progression sont repliés sur le même cœur. Ce comportement est illustré par la courbe bleue sur les

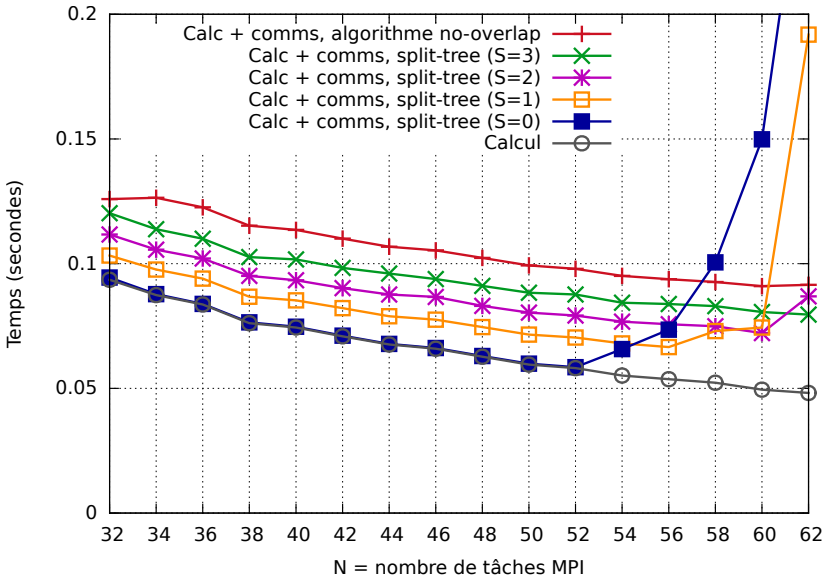


FIGURE 5.1.11 – Résultat pour l'algorithme « split-tree » avec différentes valeurs de S, pour MPI_Ireduce avec une taille de tampon constante de 2MB sur KNL.

figures 5.1.11 et 5.1.12, étiquetée « Calc + comms, split-tree (S=0) » où toutes les communications sont exécutées sur les cœurs de communication pour la collective Ireduce respectivement sur KNL et Skylake.

Grâce à l'algorithme « split-tree », nous sommes capables d'équilibrer les communications entre les cœurs applicatifs et les cœurs de communication. Les courbes orange, violette et verte respectivement étiquetées « Calc + comms, split-tree (S=1,2,3) » montrent les performances d'une même collective quand 1, 2 ou 3 étapes de l'arbre de communications sont effectuées sur les cœurs applicatifs. Quand il y a assez de cœurs dédiés aux threads de progression, l'algorithme « split-tree » est moins performant. Cependant, cet algorithme permet de maintenir de bonnes performances pour moins de cœurs dédiés aux threads de progression.

Comparaison des résultats expérimentaux aux modèles Les résultats expérimentaux obtenus sur KNL sur la figure 5.1.11 sont très proches du modèle représenté sur la figure 5.1.6. Le changement du meilleur nombre d'étapes S s'effectue au même moment sur le modèle et sur nos résultats. Ceci nous permet de sélectionner automatiquement la meilleure valeur de S en tenant compte de la collective et du nombre de cœurs dédiés aux threads de progression et de l'implémenter dans MPC.

Les résultats expérimentaux obtenus sur Skylake (figure 5.1.12) par rapport au modèle présenté sur la figure 5.1.10 sont un peu moins précis. Nous pensons

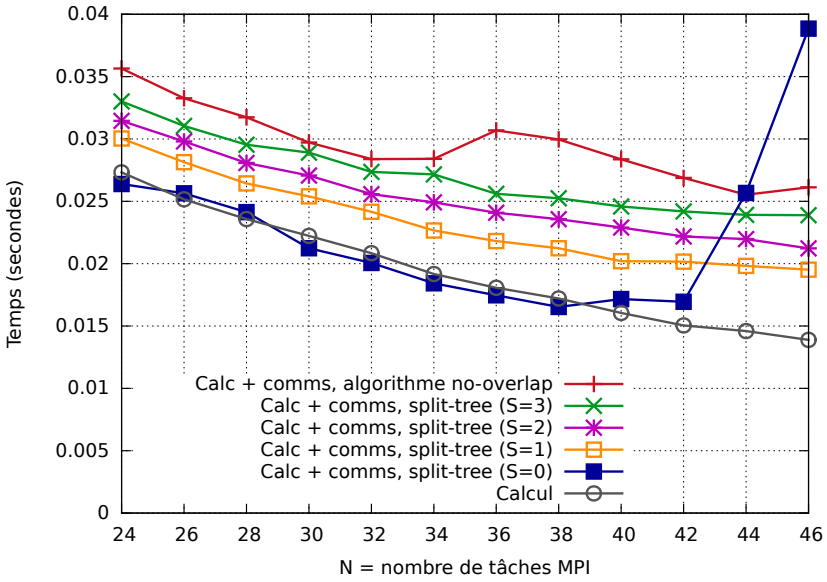


FIGURE 5.1.12 – Résultat pour l'algorithme « split-tree » avec différentes valeurs de S, pour MPI_Ireduce avec une taille de tampon constante de 2MB sur Skylake.

que cela est dû au fait que notre modèle ne prend pas en compte certains effets, tels que le cache. Néanmoins, le modèle nous permet de sélectionner automatiquement une valeur de S permettant d'améliorer les performances.

Comparaison des implémentations MPI. Nous avons comparé notre algorithme « split-tree », avec les valeurs de S données par notre modèle, à d'autres implémentations MPI telles qu'INTEL-MPI et Open MPI. Sur KNL, nous changeons de $S = 0$ à $S = 1$ pour 52 tâches MPI, de $S = 1$ à $S = 2$ pour 58 tâches MPI et de $S = 2$ à $S = 3$ pour 62 tâches MPI (respectivement 12, 6 et 2 cœurs de libres).

Les résultats des autres implémentations MPI testées sont représentés sur les figures 5.1.13 et 5.1.14 respectivement sur le KNL et le Skylake.

Par souci d'équité, nous avons activé les flags permettant d'avoir la progression asynchrone pour la bibliothèque Intel-MPI (*I_MPI_ASYNC_PROGRESS* et *I_MPI_ASYNC_PROGRESS_PIN*), mais ces flags réduisent les performances au lieu de les améliorer. Nous pensons que cela est dû à un mauvais placement des threads de progression.

Nous observons que notre algorithme (MPC-split-tree – courbe verte) est toujours meilleur que celui d'OpenMPI et d'Intel-MPI sauf pour 46 tâches MPI sur Skylake.

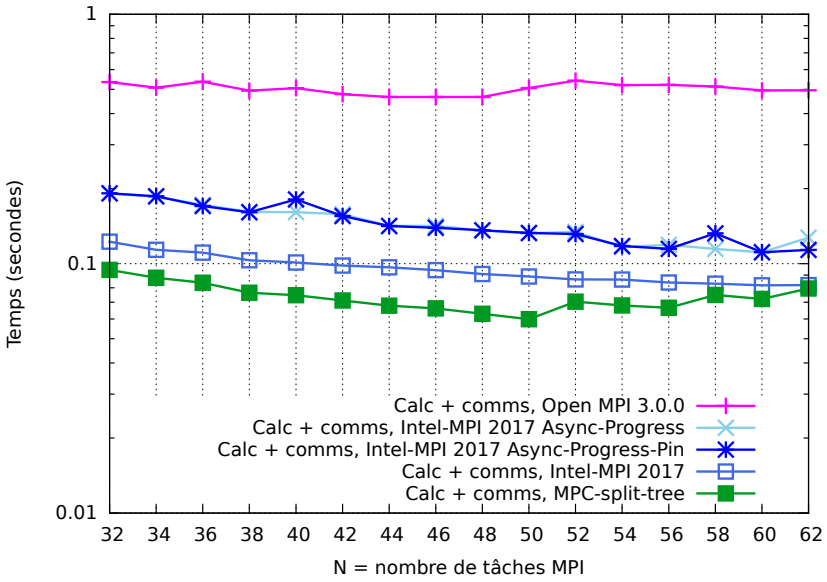


FIGURE 5.1.13 – Résultat pour plusieurs implémentations MPI pour MPI_Ireduce avec une taille de tampon constante de 2MB sur KNL.

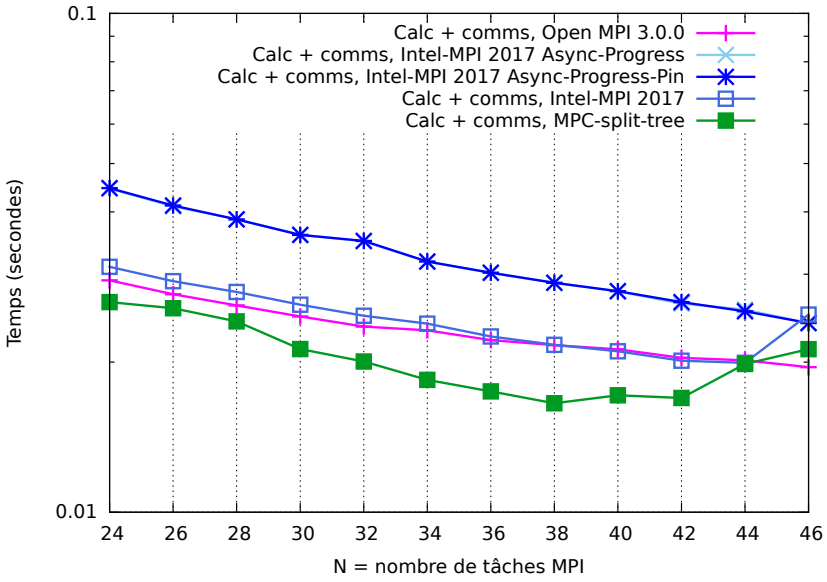


FIGURE 5.1.14 – Résultat pour plusieurs implémentations MPI pour MPI_Ireduce avec une taille de tampon constante de 2MB sur Skylake.

5.1.5 Discussion

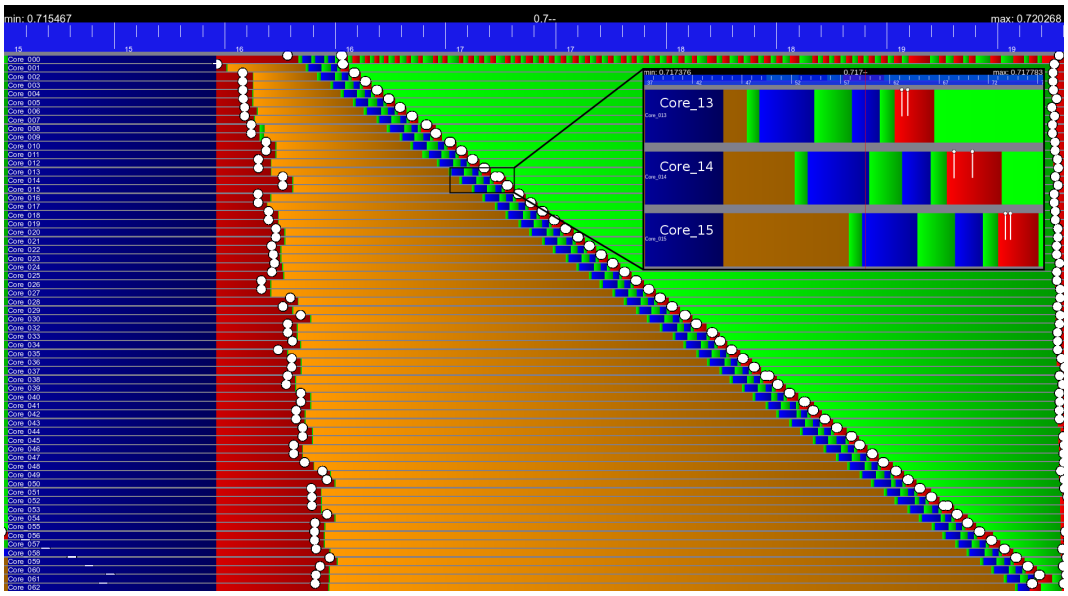
Recouvrir les communications par du calcul est la clé pour amortir le coût des communications, spécialement pour les communications collectives qui sont plus gourmandes en temps CPU que les communications point-à-point. Les approches consistant à faire de la progression avec un thread de progression par tâche souffrent de la compétition entre les communications et le calcul. Les approches consistant à allouer des cœurs dédiés aux communications souffrent d'une baisse des performances des communications quand une collective est repliée sur peu de cœurs dédiés.

Nous avons présenté un nouvel algorithme qui combine le meilleur des deux mondes. Il scinde l'arbre des communications et exécute la partie la plus légère en communications mais représentant le plus grand nombre d'étapes sur les cœurs dédiés afin de les recouvrir par du calcul tandis que la partie la plus lourde en communications mais représentant peu d'étapes est exécutée sur les cœurs applicatifs pour bénéficier de plus de parallélisme.

Nous avons modélisé l'algorithme pour démontrer sa pertinence et affiner ses paramètres. Nous l'avons implémenté dans MPC et évalué ses performances sur des processeurs manycore (Intel KNL et Skylake). Grâce à la précision de notre modèle, nous somme capable de trouver le meilleur compromis entre utiliser les cœurs dédiés ou les cœurs applicatifs et par conséquent d'avoir une meilleure performance que d'autres implémentations MPI de l'état de l'art. De plus, il est important de noter que notre solution n'est pas liée au framework MPC et peut être implémenté sur n'importe quelle implémentation MPI avec des threads de communications.

5.2 Le placement « pair-impair » pour les collectives MPI non-bloquantes en chaîne

Dans cette section, nous nous concentrons sur l'optimisation des collectives MPI non-bloquantes en chaîne. Il s'agit des collectives MPI_Iscan et MPI_Iexscan qui effectuent une réduction partielle inclusive et exclusive respectivement. L'hypothèse est qu'un seul cœur dédié aux threads de progression est suffisant pour effectuer toutes les communications car chaque communication dépend de la communication précédente. Il serait donc possible d'effectuer les opérations les unes après les autres sur un seul cœur. Ainsi, nous devrions obtenir un taux de recouvrement de 100% en dédiant qu'un seul cœur pour la progression des communications.



Légende :
Tâches MPI : Calcul
Threads de progression : Communication
Thread interne à MPC : Attente active
Thread idle : Attente passive

FIGURE 5.2.1 – Placement « bind » des threads de progression.

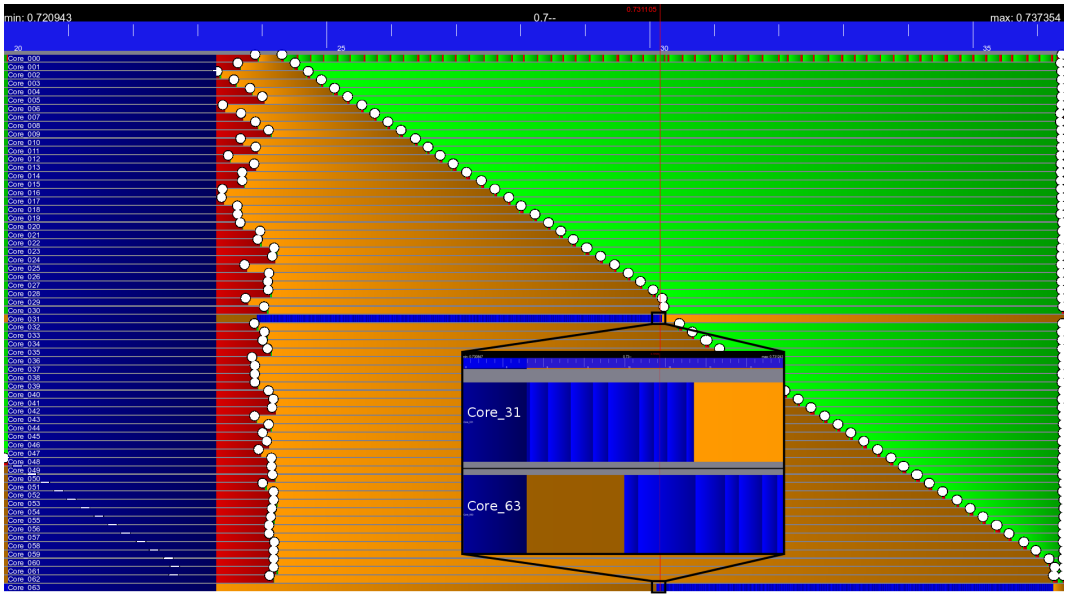
5.2.1 Étude des algorithmes en chaîne

Afin d'avoir une vision de ce qui se passe réellement lorsque les threads sont exécutés, nous avons mis au point une prise de trace au format Pajé [52] dans l'ordonnanceur de MPC. Ensuite ces traces sont visualisées avec un logiciel compatible avec ce format. Nous utilisons le logiciel ViTE [53] permettant de visualiser des traces au format Pajé.

À l'aide de ces traces, nous avons pu comprendre que notre hypothèse de départ, à savoir qu'il suffit d'un seul coeur pour faire progresser toute la communication, était erronée. Sur la figure 5.2.1, nous pouvons voir le déroulement de la collective MPI_Iscan avec 62 tâches MPI sur un KNL de 64 cœurs pour le placement « bind » vu dans la section 4.2.2.

Les couleurs correspondent aux types des threads s'exécutant sur les cœurs :

— La couleur rouge correspond à l'exécution d'une tâche MPI qui effectue du calcul.
— La couleur bleue correspond à l'exécution d'un thread de progression des collectives MPI non-bloquantes.
— La couleur verte correspond à l'exécution d'un thread interne à MPC qui effectue de l'attente active. Dans notre cas, il s'agit de la progression de la fonction MPI_Barrier après l'exécution de MPI_Iscan.



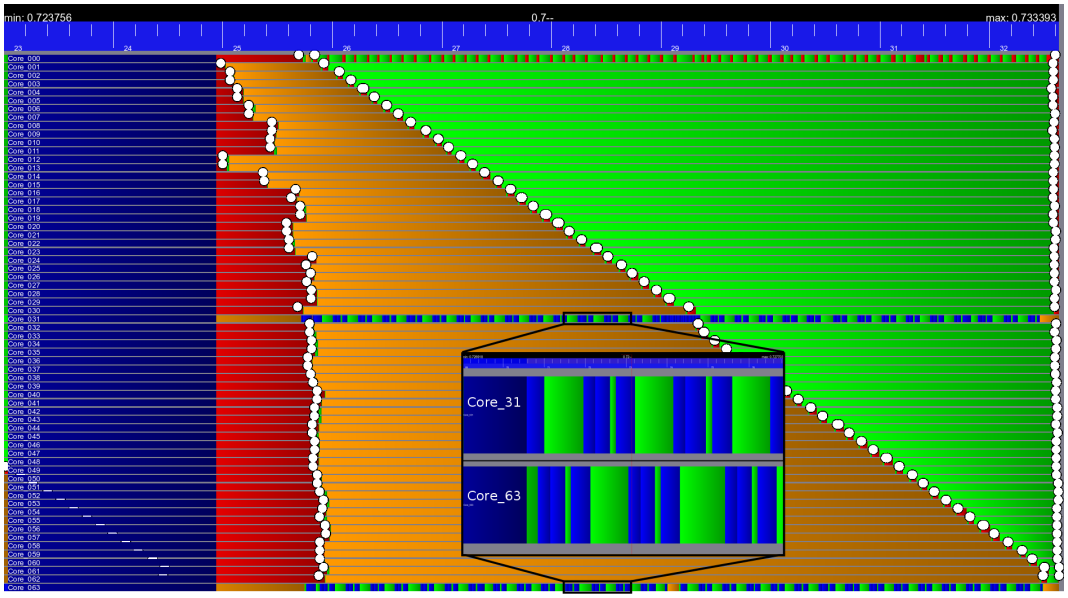
Légende :
Tâches MPI : Calcul
Threads de progression : Communication
Thread interne à MPC : Attente active
Thread idle : Attente passive

FIGURE 5.2.2 – Placement « numa » des threads de progression.

— La couleur orange correspond à l'exécution du thread « idle ». C'est une routine s'exécutant lorsque qu'il n'y a plus de threads à exécuter dans la liste des threads prêt, dans l'ordonnanceur.

Les threads de progression associés aux rangs MPI 1 à $n-2$ effectuent la réception d'un message, puis l'envoi d'un message au rang MPI suivant. Le thread de progression associé au rang MPI 0 n'effectue qu'un envoi de message tandis que le thread de progression associé au rang MPI $n-1$ n'effectue qu'une réception de message. La diagonale bleue que nous observons correspond à ces échanges de messages. Rappelons que comme les threads de progression s'exécutent sur les mêmes cœurs que les tâches MPI, il n'est pas possible de recouvrir les communications par du calcul.

C'est pourquoi nous avons mis en place le placement « numa » dans la sous-section 4.2.2 à la page 55. En utilisant ce placement pour *scan* et *exscan*, nous observons un temps de communication plus lent. Nous allons expliquer cela sur le cas avec 62 tâches MPI et 2 cœurs dédiés à la progression, illustré sur la figure 5.2.2. Les threads de progression des rangs MPI 0 à 30 sont placés sur le cœur 31 tandis que les threads de progression des rangs MPI 31 à 61 sont placés sur le cœur 63 ; avec les cœurs numérotés de 0 à 63. Les cœurs 31 et 63 ne sont donc utilisés que la moitié du temps. De plus, nous observons aussi que lorsque le thread de progression associé au rang MPI 30 s'exécutant



Légende :
Tâches MPI : Calcul
Threads de progression : Communication
Thread interne à MPC : Attente active
Thread idle : Attente passive

FIGURE 5.2.3 – Placement « pair-impair » des threads de progression.

sur le cœur 31 envoie un message ; le thread de progression associé au rang MPI 31 est déjà en train de s'exécuter sur le cœur 63 pour la réception du message. Certaines parties de l'algorithme peuvent s'exécuter simultanément. Cette observation nous montre qu'un seul cœur n'est pas suffisant pour effectuer la progression des communications basée sur une chaîne puisqu'elle ne permet pas ce recouvrement. En effet, si nous n'avions qu'un seul cœur pour les threads de progression, 2 threads de progression ne pourraient pas s'exécuter en même temps.

5.2.2 Le placement « pair-impair »

Nous avons donc élaboré un nouvel algorithme de placement des threads de progression dédiés aux collectives basées sur des chaînes de communication (scan et exscan). Ce placement nommé « pair-impair » nécessite d'avoir deux cœurs à dédier aux threads de progression. Le placement est le suivant : les threads de progression associés aux rangs MPI pairs sur un cœur dédié et les threads de progression associés aux rangs MPI impairs sur un autre cœur. Ce placement est illustré sur la figure 5.2.3.

Nous pouvons voir que contrairement au placement « numa » qui ne permettait de recouvrir qu'une petite partie des communications entre l'envoi du

message provenant du rang MPI 30 au rang MPI 31, le placement « pair-impair » permet de recouvrir tous les envois des tâches MPI de rang pair (respectivement impair) avec les réceptions des tâches MPI de rang impair (respectivement pair). Ce phénomène est illustré sur la partie zoomée de la figure 5.2.3. Nous pouvons voir que nous retrouvons le comportement du placement « bind ». En effet, nous retrouvons les mêmes temps d'attentes entre les échanges de messages, partiellement recouvert par l'exécution des threads de progression précédents et suivants.

5.2.3 Résultats expérimentaux

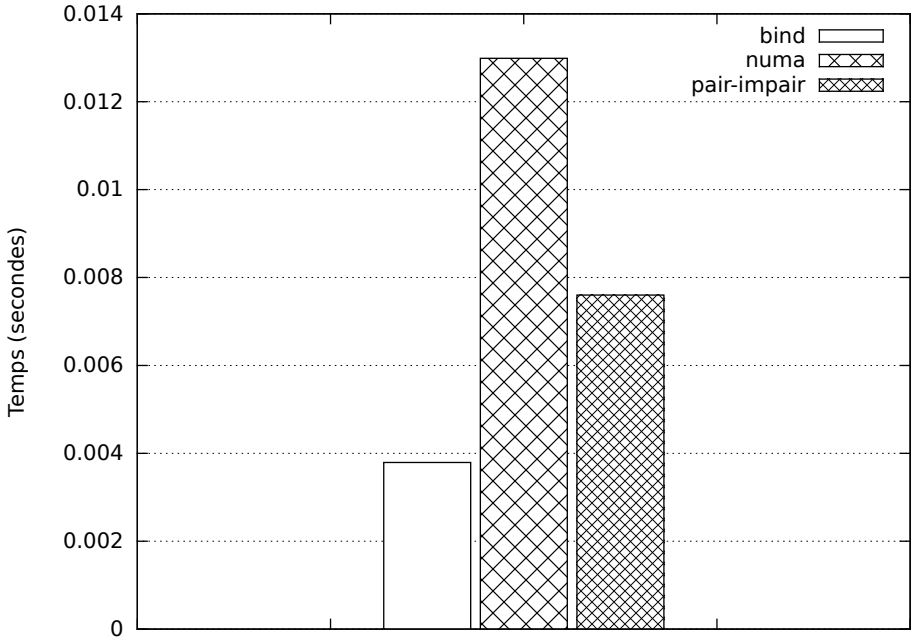
Nous avons mesuré le temps d'une communication avec les trois placements suivants :

— Le placement « bind » : chaque thread de progression est placé sur le même cœur que sa tâche MPI associée. Il n'y a pas de recouvrement entre les communications et le calcul car les tâches MPI et les threads de progression se partagent le même cœur sans préemption.
— Le placement « numa » : chaque thread de progression est associé au cœur le plus proche. Le recouvrement des communications est possible car les threads de progression et les tâches MPI ne se partagent pas les mêmes cœurs.
— Le placement « pair-impair » : les threads de progression provenant des rangs MPI pair se partagent le même cœur dédié tandis que les threads de progression provenant des rangs MPI impair se partagent un autre cœur dédié. De la même manière que le placement précédent, le recouvrement des communications est possible car les threads de progression et les tâches MPI ne se partagent pas les mêmes cœurs.

Sur la figure 5.2.4, nous pouvons voir le temps de communication de la collective MPI non-bloquante MPI_Iscan avec 62 tâches MPI sur un KNL de 64 cœurs.

Nous observons un gain de 41.5% sur le temps de communication de la collective MPI_Iscan avec le placement « pair-impair » par rapport au placement « numa ». Ceci est dû au fait que nous pouvons recouvrir les envois de messages avec les réceptions avec ce placement des threads de progression.

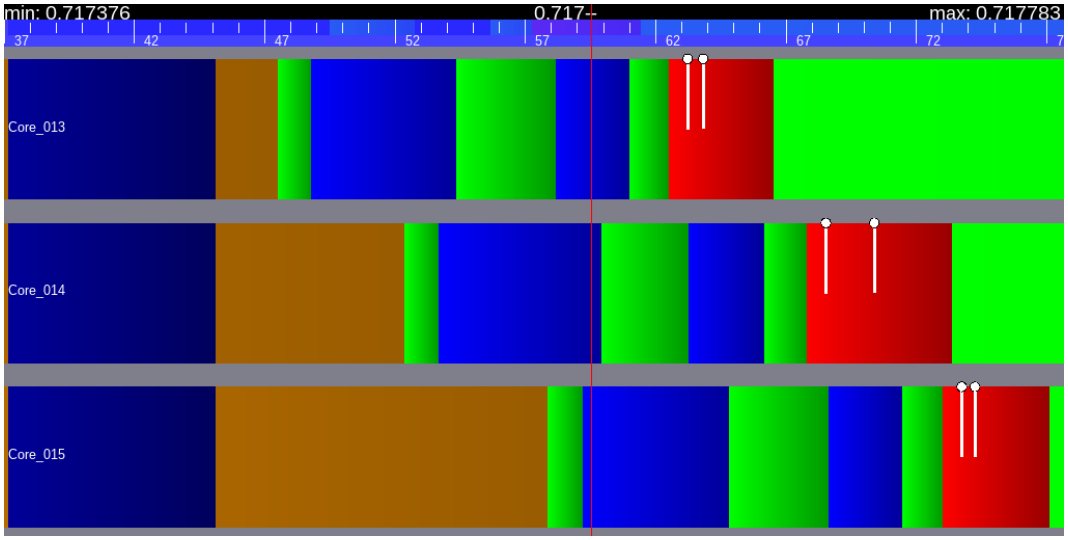
Bien que le placement « bind » ne permette pas de recouvrir les communications avec du calcul, nous observons un meilleur temps d'exécution quand on ne recouvre pas. Nous ne parvenons pas à atteindre ce temps de communication avec le placement « pair-impair » car comme nous pouvons le voir sur la figure 5.2.5 représentant les traces d'exécution sur les cœurs 13, 14 et 15 ; le thread de progression s'exécute deux fois, une première fois pour effectuer la réception et l'envoi de messages et une seconde fois pour vérifier qu'il n'y a plus rien à faire. Ce temps là est recouvert par le thread de progression s'exécutant sur le cœur 14 ainsi que par le thread de progression sur le cœur 15 quand



Recouvrement :

- bind : temps de commununication *non recouvrable*
- numa : temps de commununication *recouvrable*
- pair-impair : temps de commununication *recouvrable*

FIGURE 5.2.4 – Temps de communication de MPI_Iscan pour 62 tâches MPI sur 64 cœurs



Légende :
Tâches MPI : Calcul
Threads de progression : Communication
Thread interne à MPC : Attente active
Thread idle : Attente passive

FIGURE 5.2.5 – Zoom pour les cœurs 13, 14 et 15 pour le placement « bind » des threads de progression.

nous avons le placement « bind ». Au contraire, quand nous avons le placement « numa » ou « pair-impair », le thread de progression associé au rang MPI 13 ne peut pas être recouvert par le thread de progression associé au rang MPI 15 puisqu'ils sont sérialisés sur le même cœur. En regardant la trace de plus près, nous nous rendons compte que l'exécution du thread de progression associé au rang MPI 15 s'intercale entre les deux phases de l'exécution du thread de progression associé au rang MPI 13. Cela entraîne un décalage dans l'exécution complète du thread de progression associé au rang MPI 13, et par extension de tous les threads de progression sérialisés, expliquant ainsi la différence de temps de communication entre le placement « bind » et le placement « pair-impair ».

5.2.4 Conclusion sur l'algorithme « pair-impair »

Dans cette section, nous proposons l'algorithme de placement « pair-impair » pour le placement des threads de progression provenant des collectives MPI non-bloquantes en chaîne. Ce placement permet d'obtenir un gain de 41.5% sur le temps de communication par rapport au placement « numa ». Bien que le placement « bind » permette d'avoir un meilleur temps de communication, celui-ci ne permet pas de recouvrir le temps de communication par du temps de calcul.

5.3 Conclusion

Dans ce chapitre, nous nous sommes intéressés aux placements dynamiques des threads de progression en fonctions des algorithmes de collectives utilisés. Nous avons proposé un algorithme permettant d'améliorer le recouvrement des communications collectives MPI non-bloquantes pour les opérations basées sur un arbre de communication (broadcast, reduce, scatter, gather, allreduce). Ensuite nous avons vu comment améliorer le temps de communication des algorithmes en chaîne (scan et exscan) par rapport à nos précédents algorithmes de placement de threads de progression. Nous avons vu que pour obtenir un bon temps d'exécution, il faut faire un compromis entre le recouvrement des communications par le calcul et la performance de communication.

Tous nos travaux se sont concentrés sur le placement des threads de progression. Nous avons vu que l'utilisation de cœurs dédiés pour les threads de progression aidait à mieux recouvrir les communications par du calcul. Néanmoins, tous ces threads de progression se retrouvent en concurrence sur les cœurs dédiés. Le chapitre suivant, portant sur la mise en place de politique d'ordonnancement des threads de progression sur les cœurs dédiés, apporte des solutions à cette problématique.

Chapitre 6

Politiques d'ordonnancement des threads de progression sur les cœurs dédiés

Sommaire

Dans ce chapitre, nous nous intéressons à l'ordonnancement des threads de progression qui surchargent les cœurs dédiés. Nous avons vu dans les chapitres 4 et 5 que le placement des threads de progression permettait d'améliorer la progression des communications MPI non-bloquantes. Nous avons vu qu'allouer des cœurs pour les threads de progression permettait de mieux recouvrir les communications par du calcul. Afin de ne pas allouer trop de cœurs pour les

threads de progression, nous avons proposé l'algorithme *split-tree* permettant de répartir les communications sur les cœurs applicatifs et les cœurs dédiés aux threads de progression. Néanmoins, toutes nos techniques de placement des threads de progression provoquent une surcharge des cœurs dédiés aux threads de progression.

Nous proposons une optimisation permettant de ne pas exécuter les threads de progression lorsqu'ils n'ont plus de travaux à exécuter. Nous proposons plusieurs méthodes permettant d'améliorer l'ordonnancement des threads de progression sur les cœurs dédiés aux communications MPI non-bloquantes à l'aide de sémaphores. Enfin, nous proposons une nouvelle politique d'ordonnancement basée sur la priorité des threads de progression. Les méthodes proposées dans ce chapitre sont des preuves de concept permettant d'avoir une première approche sur l'ordonnancement des threads de progression.

6.1 Problématique de la surcharge des cœurs par les threads de progression

Les travaux que nous avons menés jusqu'à présent montrent la nécessité de dédier des cœurs aux threads de progression. Cependant, il n'est pas concevable de dédier la moitié des cœurs d'une machine pour les threads de progression. C'est pourquoi, nous avons mis au point l'algorithme de placement « numa » permettant d'avoir une solution lorsque nous avons peu de cœurs à dédier aux threads de progression. Même si l'algorithme « split-tree » que nous avons mis en place permet de relâcher cette pression pour les algorithmes en arbre, les threads de progression générés par les tâches MPI se retrouvent tous repliés sur les mêmes cœurs dédiés. Une surcharge de ces cœurs est donc à prévoir.

L'ordonnancement des threads de progression sur les cœurs dédiés est donc un levier pour obtenir de meilleures performances. En effet, chaque collective MPI non-bloquante repose sur un algorithme de collective basé sur des envois de message point-à-point. Ces échanges de messages forment un graphe de dépendance.

Lorsque les threads de progression sont exécutés sur les cœurs dédiés, ils sont ordonnancés les uns après les autres sans prendre en compte le fait qu'ils exécutent des algorithmes bien définis. Les threads s'exécutant sur les cœurs dédiés sont donc ordonnancés avec la politique d'ordonnancement standard de MPC, à savoir « round-robin ». Cette politique d'ordonnancement n'est pas forcément la plus adéquate pour les threads de progression exécutant des algorithmes bien spécifiques. En effet, cet algorithme génère un nombre de changements de contexte important. Or, un ordonnancement efficace des threads de progression permettrait d'obtenir un gain de performance. Le but est donc de ne pas perdre de temps à exécuter des threads de progression qui ne sont pas prêts à exécuter leur travail, car ils dépendent de l'exécution d'un

autre thread de progression.

Dans notre cas, nous nous appuyons sur l'ordonnanceur de MPC. La principale contrainte posée par cet ordonnanceur est qu'il s'agit d'un ordonnanceur non-préemptif. C'est-à-dire que lorsque un thread s'exécute sur un cœur, il est impossible de le préempter (l'interrompre) pour en exécuter un autre. Nous sommes donc dans l'obligation d'attendre que ce thread relâche le processeur de son plein gré. Ensuite l'ordonnanceur est chargé de choisir quel est le thread suivant à exécuter sur le cœur.

Afin de traiter le problème provoqué par la surcharge des cœurs dédiés à la progression des communications, nous avons commencé par suspendre les threads de progression ayant fini leur travail.

6.2 Suspension des threads de progression inutiles

Dans cette section, nous proposons une optimisation dans l'ordonnancement des threads de progression s'exécutant sur les cœurs dédiés à la progression des communications collectives MPI non-bloquantes. Cette optimisation vise à suspendre les threads de progression ayant déjà terminé de contribuer à leur algorithme de collective dans le but de ne pas ordonnancer des threads inutilement. Pour cela, nous utilisons un mécanisme interne à MPC permettant de retirer les threads de progression de la liste des threads prêts à être ordonnancés.

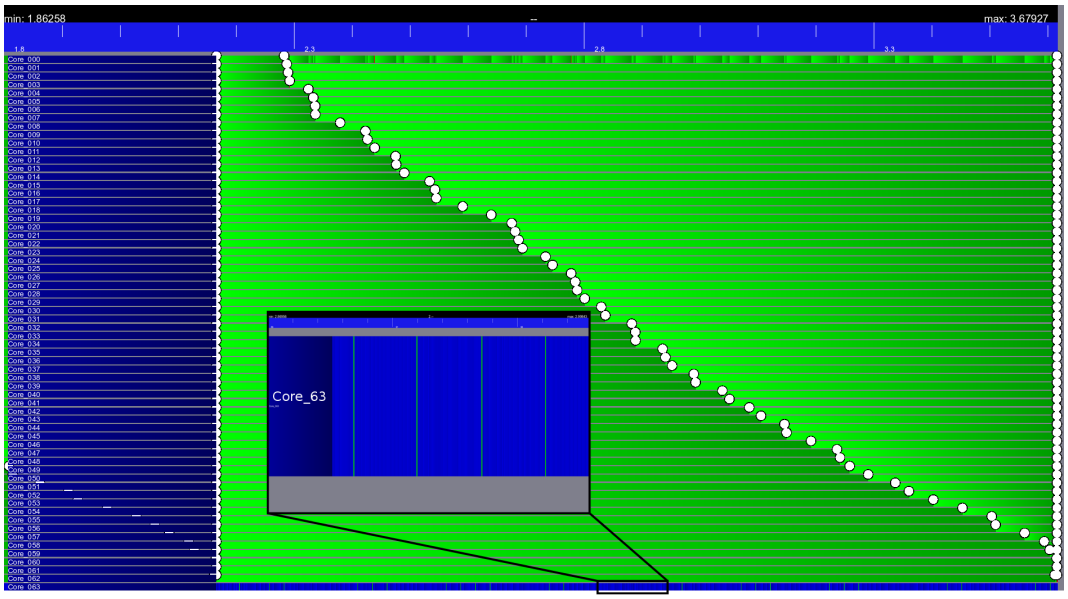
6.2.1 Mécanisme de progression interne à MPC

Ce mécanisme repose sur la fonction *wait_for_value_and_poll* que nous abrègerons « *wfvap* ». Lors de l'appel à cette fonction, le thread appelant sera bloqué jusqu'à ce qu'une condition soit vérifiée. Cette vérification repose sur un thread interne à MPC scrutant toutes les conditions soumises par différents threads ayant fait appel à ce mécanisme.

Dans l'ordonnanceur de MPC utilisé pour effectuer nos tests, chaque cœur possède une liste de threads prêts à être exécutés. L'algorithme par défaut est l'algorithme « round-robin », c'est-à-dire que tous les threads sont exécutés les uns à la suite des autres. Parmi ces threads, il y a ce thread interne à MPC.

6.2.2 Implémentation

Dans notre cas, nous utilisons le mécanisme *wfvap* pour retirer les threads de progression ayant déjà effectué leur part de l'algorithme de collective. Pour cela, un compteur est utilisé pour chaque thread de progression. Ce compteur permet de savoir si un thread de progression a encore une collective à faire progresser.



Légende :
Tâches MPI : Calcul
Threads de progression : Communication
Thread interne à MPC ou thread idle : Temps d'attente

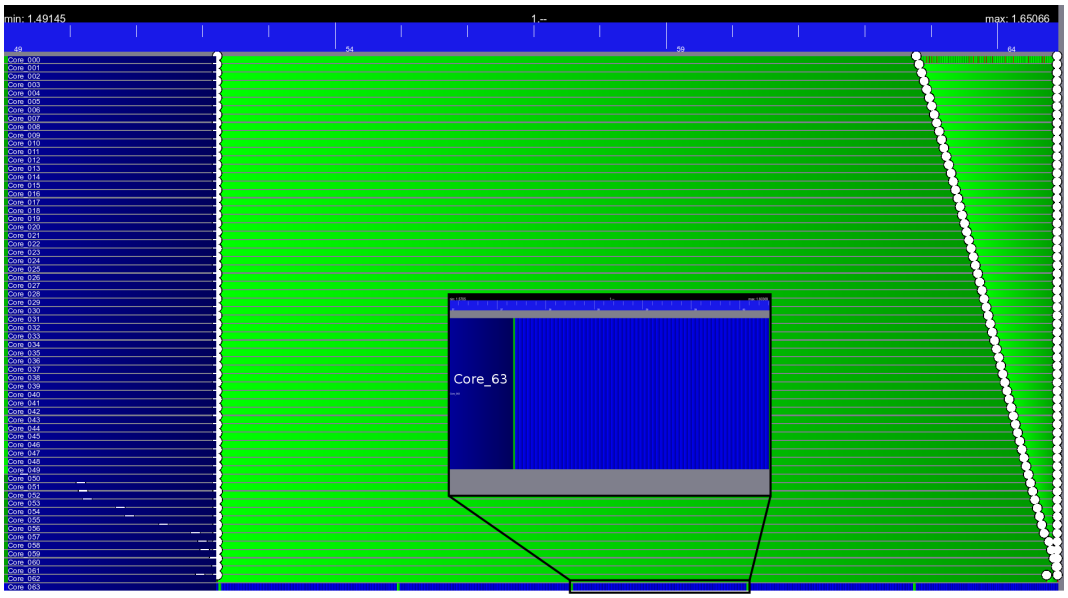
FIGURE 6.2.1 – Trace d'exécution de l'algorithme MPI_Iscan avec 63 tâches MPI sur un KNL de 64 cœurs en utilisant le placement « numa » **sans** notre optimisation (sans-wfvap).

Dans le cas contraire, le thread de progression fera appel à la fonction *wfvap*. Cela aura pour effet de bloquer le thread de progression tant que le compteur vaudra 0. Lorsqu'une autre opération collective sera appelé, le compteur sera incrémenté et le thread interne à MPC réveillera le thread de progression en question. Ainsi, l'ordonnanceur de MPC ne continuera pas à exécuter un thread de progression qui n'a plus de collective à faire progresser.

6.2.3 Résultats expérimentaux

La figure 6.2.1 représente une trace d'exécution de la collective MPI_Iscan suivi d'une MPI_Barrier avec 63 tâches MPI sur un KNL de 64 cœurs en utilisant le placement « numa » **sans** notre optimisation. L'exécution de la collective est faite par les threads de progression qui sont tous placés sur le cœur 63.

Pour chaque cœur, représenté par une ligne dans notre figure, nous pouvons voir en vert le thread interne à MPC et en bleu l'exécution des threads de progression. Nous avons 3 points blancs par ligne représentant, de gauche à droite, le début de MPI_Iscan, la fin de MPI_Iscan et le début de MPI_Barrier et enfin la fin de MPI_Barrier.



Légende :
Tâches MPI : Calcul
Threads de progression : Communication
Thread interne à MPC ou thread idle : Temps d'attente

FIGURE 6.2.2 – Trace d'exécution de l'algorithme MPI_Iscan avec 63 tâches MPI sur un KNL de 64 cœurs en utilisant le placement « numa » **avec** notre optimisation (avec-wfvap).

Nous pouvons voir que l'exécution des threads sur le cœur 63 s'effectue avec l'algorithme « round-robin ». Les 63 threads de progression s'exécutent les uns après les autres suivi du thread interne à MPC en vert, puis de nouveau, l'exécution des threads de progression et ainsi de suite.

Le problème qui intervient lorsque le thread de progression associé au rang MPI 0 finit d'envoyer son message est qu'il perturbe l'exécution des autres threads de progression n'ayant pas encore effectué leur échange de message. Il en est de même pour les autres threads lorsqu'ils ont effectué leur part de la collective. Ils ne devraient plus être ordonnancés.

La figure 6.2.2 représente le même cas test mais **avec** notre optimisation. Notons que l'échelle de temps représentée sur cette figure est plus petite que sur la figure 6.2.1. Sur la partie zoomée, nous pouvons voir que les threads de progression sont tous ordonnancés avec l'algorithme « round-robin » mais qu'ils ne sont tous ordonnancés que 5 fois pendant l'exécution de la fonction MPI_Iscan. Il y a donc beaucoup moins de changements de contextes entre les threads de progression.

Nous pouvons voir les temps d'exécution correspondant à nos tests sur la figure 6.2.3. Nous observons que le temps d'exécution de la collective MPI_Iscan avec notre optimisation est 11.4 fois plus rapide. Ceci est dû au fait que les

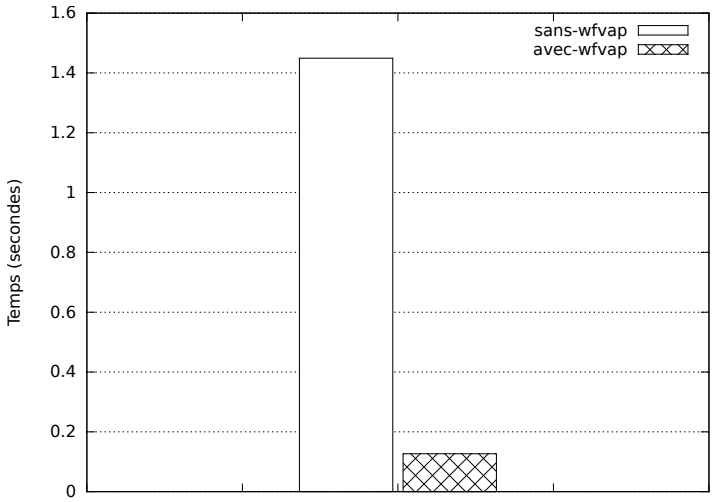


FIGURE 6.2.3 – Temps d'exécution de l'algorithme MPI_Iscan avec 63 tâches MPI sur un KNL de 64 cœurs en utilisant le placement « numa ».

threads de progression qui n'ont plus de collective à faire progresser ne se retrouvent plus dans la liste des threads prêts à être exécutés.

6.2.4 Conclusion

Nous avons montré l'importance de la gestion des threads de progression s'exécutant sur les cœurs dédiés. Nous avons mis en place un mécanisme permettant d'optimiser l'ordonnancement des threads de progression, ce qui a permis un *speedup* de 11.4 pour le temps d'exécution pour la collective MPI_Iscan. Néanmoins, il est encore possible d'améliorer les performances.

6.3 Ordonnancement statique à l'aide de sémaphores

Nous ne prenons pas en compte les dépendances entre les communications point-à-point formant l'algorithme de collective pendant son exécution. C'est pourquoi nous proposons une gestion de l'ordonnancement des threads de progression de façon statique reposant sur les dépendances entre les communications point-à-point formant l'algorithme de collective dans cette section. Nous avons donc mis en place un ordonnancement statique à l'aide de sémaphores.

6.3.1 Gestion des threads de progression avec des sémaphores

La gestion des threads de progression à base de sémaphores pour la collective MPI_Iscan présente l'avantage d'être rapide à implémenter. Nos recherches se sont concentrées sur les algorithmes où nous avons une chaîne de communications (Scan, Exscan) afin d'avoir une preuve de concept extensible aux autres collectives.

Hormis la première tâche MPI devant envoyer le premier message et la dernière tâche MPI devant recevoir le dernier message, chaque tâche MPI doit recevoir un message, puis envoyer un message afin de former une chaîne de communications.

Fonctionnement des sémaphores : Rappelons que les sémaphores s'utilisent avec les fonctions *sem_wait* et *sem_post*.

Lorsque la fonction *sem_wait* est appelée, la valeur du sémaphore est décrémentée si la valeur du sémaphore est supérieure à 0. Si la valeur du sémaphore est égale à 0, l'appel bloquera. Dans MPC, cela placera le thread de progression dans une liste de threads bloqués ayant pour effet de désactiver le thread et de le retirer de la liste des threads prêt à être exécuté.

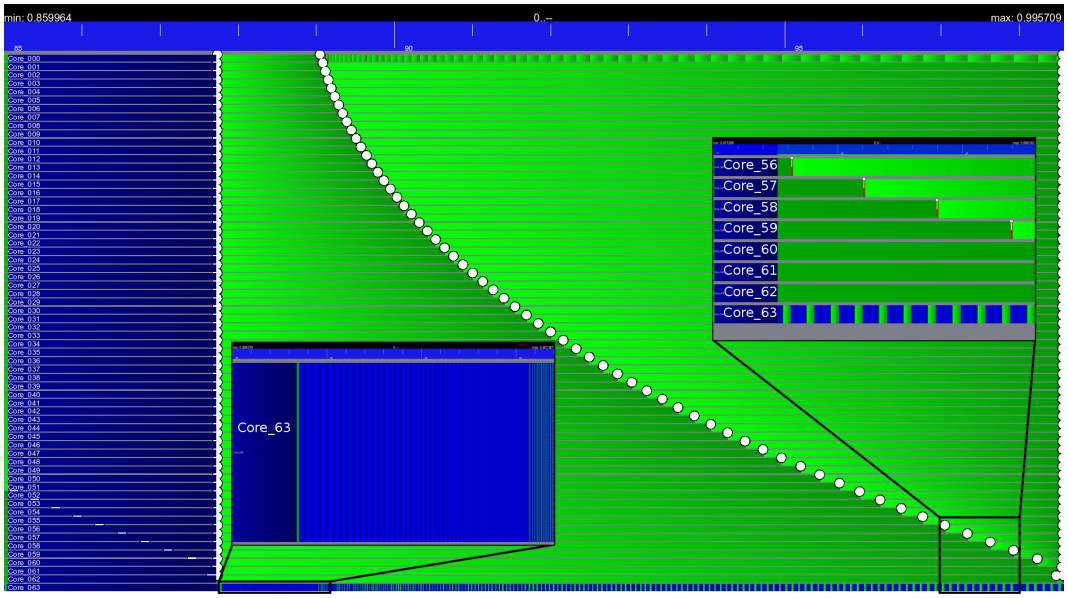
Lorsque la fonction *sem_post* est appelée, la valeur du sémaphore est incrémentée. Si à la suite de cet incrément, la valeur du sémaphore devient supérieur à 0, le thread de progression bloqué par le sémaphore sera réveillé.

Implémentation : La gestion des dépendances avec les sémaphores pour l'algorithme Scan se fait de la façon suivante :

- Chaque thread de progression i associé à sa tâche MPI possède un sémaphore sem_i.
- À chaque nouvel appel à une collective, les sémaphores de toutes les tâches sont initialisés par la tâche MPI appelante.
- Le thread de progression du rang MPI 1 se voit attribuer un sémaphore de valeur 1.
- Tous les autres threads de progression se voient attribuer un sémaphore de valeur 0.

Ensuite, l'ordonnancement des threads de progression se fait de façon statique dans le code source à l'aide des fonctions *sem_wait* et *sem_post*. Chaque thread de progression i de 1 à $n-1$ effectue *sem_post*(sem_{i-1}) puis *sem_wait*(sem_i) après chaque réception de message. Chaque thread de progression i de 1 à $n-1$ effectue *sem_post*(sem_{i+1}) puis *sem_wait*(sem_i) avant chaque envoi de message puis *sem_post*(sem_{i+1}) après l'envoi de son message. Le thread de progression 0 n'effectue que *sem_post*(sem_{i+1}) après l'envoi de son message.

L'ordonnancement des threads de progression est le suivant :



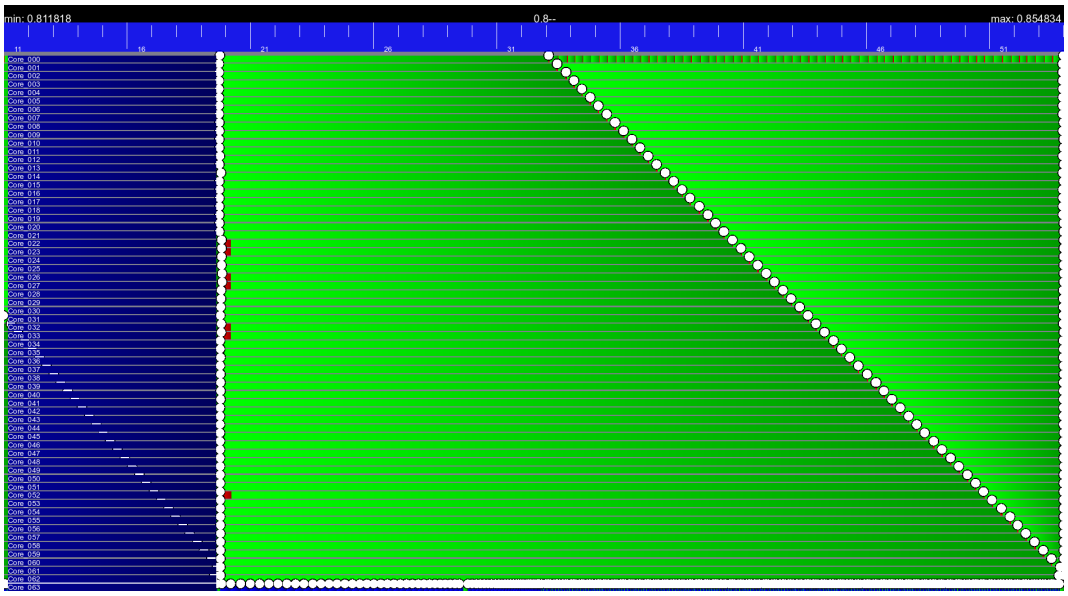
Légende :
Tâches MPI : Calcul
Threads de progression : Communication
Thread interne à MPC ou thread idle : Temps d'attente

FIGURE 6.3.1 – Trace d'exécution de l'algorithme MPI_Iscan avec 63 tâches MPI sur un KNL de 64 cœurs en utilisant le placement « numa » **avec** l'optimisation vue dans la section 6.2 et la gestion de l'ordonnancement **avec** les sémaphores (avec-wfvap-avec-sem).

1. Irecv(1 from 0)
2. Isend(0 to 1)
3. Irecv(2 from 1)
4. Isend(1 to 2)
5. Irecv(i from i-1)
6. Isend(i-1 to i+1)
7. Irecv(i+1 from i)
8. Isend(i to i+1)
9. Irecv(n-1 from n-2)
10. Isend(n-2 to n-1)

6.3.2 Résultats expérimentaux

La figure 6.3.1 représente une trace d'exécution de la collective MPI_Iscan suivi d'une MPI_Barrier avec 63 tâches MPI sur un KNL de 64 cœurs en utilisant le placement « numa » **avec** l'optimisation consistant à suspendre les threads de progression inutiles à l'aide de la fonction *wfvap* et la gestion de l'ordonnancement avec les sémaphores.



Légende :
Tâches MPI : Calcul
Threads de progression : Communication
Thread interne à MPC ou thread idle : Temps d'attente

FIGURE 6.3.2 – Trace d'exécution de l'algorithme MPI_Iscan avec 63 tâches MPI sur un KNL de 64 cœurs en utilisant le placement « numa » **sans** l'optimisation vue dans la section 6.2 et la gestion de l'ordonnancement **avec** les sémaphores (sans-wfvap-avec-sem).

Nous pouvons voir qu'au départ, tous les threads de progression s'exécutent les uns après les autres avec l'algorithme « round-robin ». Ensuite, grâce à l'utilisation des sémaphores, nous avons l'ordre d'exécution des threads qui est parfait. C'est la raison pour laquelle les threads de rang 0 à 32 sont exécutés très rapidement. Néanmoins, les threads de progression sont ensuite suspendus à l'aide de la fonction *wfvap* au fur et à mesure car ils n'ont plus de collective à faire progresser. Cela a pour effet de saturer le thread interne de MPC et ralonge son temps d'exécution. C'est ce que nous observons sur la figure à partir du rang 32 à 62.

La figure 6.3.2 représente le même cas test mais **sans** l'utilisation de *wfvap* qui produit une surcharge de travail sur le thread interne à MPC.

L'ordonnancement des threads de progression est effectué uniquement avec les sémaphores et nous observons que nous n'avons plus de surcharge du thread interne à MPC dû à l'utilisation intensive de la fonction *wfvap*. Les threads de progression sont donc ordonnancés uniquement lorsqu'ils ont du travail à effectuer.

Sur la figure 6.3.3, nous observons le temps d'exécution correspondant à nos tests. Nous pouvons voir que l'ordonnancement statique « sans-wfvap-avec-sem »

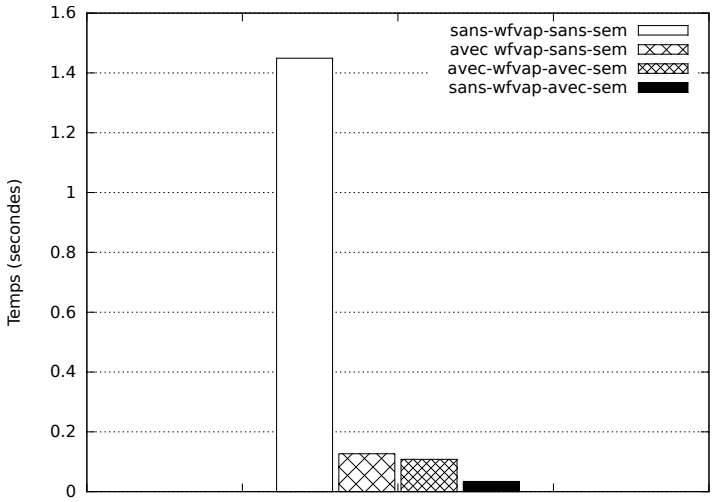


FIGURE 6.3.3 – Temps d'exécution de l'algorithme MPI_Iscan avec 63 tâches MPI sur un KNL de 64 cœurs en utilisant le placement « numa ».

améliore le temps d'exécution de 3 par rapport au cas « avec-wfvap-avec-sem » et de 42 fois par rapport au cas « sans-wfvap-sans-sem ».

6.3.3 Conclusion

Cette section vise à montrer que les performances des collectives non-bloquantes peuvent être améliorées à l'aide d'un ordonnancement statique des threads de progression. Les travaux présentés dans cette section mettent en évidence qu'il est nécessaire d'ordonnancer correctement les threads de progression lorsqu'ils sont en concurrence sur les mêmes cœurs dédiés. Dans la section suivante, nous allons voir que l'ordonnancement des threads de progression est indispensable lorsque nous n'avons pas de cœurs dédiés aux threads de progression.

6.4 Ordonnancement dynamique à l'aide de priorité

Dans cette section, nous proposons une politique d'ordonnancement dynamique basée sur des priorités. L'idée est de montrer que gérer les threads de progression lorsqu'ils sont en concurence avec d'autre threads de progression, ou des tâches MPI dans le cas où nous n'avons pas de coeurs dédiés, est indispensable. Pour cela, nous proposons de modifier l'ordonnanceur de MPC afin d'intégrer des priorités dynamiques.

6.4.1 Gestion des threads de progression avec des priorités

Nous avons modifié l'ordonnanceur de MPC en ajoutant un type à chaque thread en fonction de sa nature (tâche MPI, thread OpenMP, thread de progression, etc.). Ensuite, nous avons mis en place un algorithme d'ordonnancement avec une priorité par type de thread. Ainsi, lorsque les threads de progression doivent faire progresser une collective MPI non-bloquante, nous avons la possibilité d'augmenter la priorité des threads concernés. Rappelons que l'ordonnanceur de MPC est non-préemptif. La seule façon que nous avons de gérer l'ordonnancement des threads de progression avec des priorités est de faire en sorte que le choix du prochain thread à exécuter soit défini par les priorités.

6.4.2 Implémentation

Pour chaque cœur, nous avons une liste de threads prêts à être exécutés appelée : *ready_list*. Nous définissons une échelle avec les priorités pouvant aller de 0 à 20. Une priorité de base de 0 est donnée à chaque thread lors de sa création. Chaque thread possède une *priorité de base* et une *priorité courante*.

La *ready_list* consistera en une liste triée de threads prêts à être ordonnancés. Lorsqu'un thread est choisi pour être exécuté, sa *priorité courante* est diminuée de 1. Si le thread a une priorité de 0, sa *priorité courante* est réinitialisée à sa valeur de *priorité de base*. Ainsi, si tous les threads conservent une priorité de 0, nous retombons sur l'algorithme « round-robin ».

L'implémentation des priorités dans l'ordonnanceur de MPC s'est faite en modifiant principalement deux fonctions. Il s'agit des fonctions permettant de choisir le nouveau thread à exécuter, que nous appellerons *get_from_list* et de la fonction permettant de rajouter les threads dans la *ready_list* que nous appellerons *add_to_list*.

La fonction *get_from_list* est très simple : étant donné que la *ready_list* est toujours triée, nous choisissons toujours la tête de liste. La complexité du choix du thread à exécuter est donc en temps constant.

Pour la fonction *add_to_list*, Il suffit de comparer la *priorité courante* du thread à ajouter à la *ready_list*, à celle de la *priorité courante* de la tête de la *ready_list*.

Lorsque nous souhaitons augmenter la priorité d'un thread, il suffit d'augmenter la *priorité courante* ainsi que la *priorité de base* du thread. Il est aussi possible d'augmenter les priorités d'un certain type de thread en même temps.

Il est possible d'améliorer l'implémentation naïve que nous avons utilisée en changeant la structure de la *ready list*. Actuellement, nous avons une liste qui implique que la complexité de l'ajout d'un élément dans la liste est en $O(n)$ tandis que la suppression est en temps constant ($O(1)$). L'utilisation

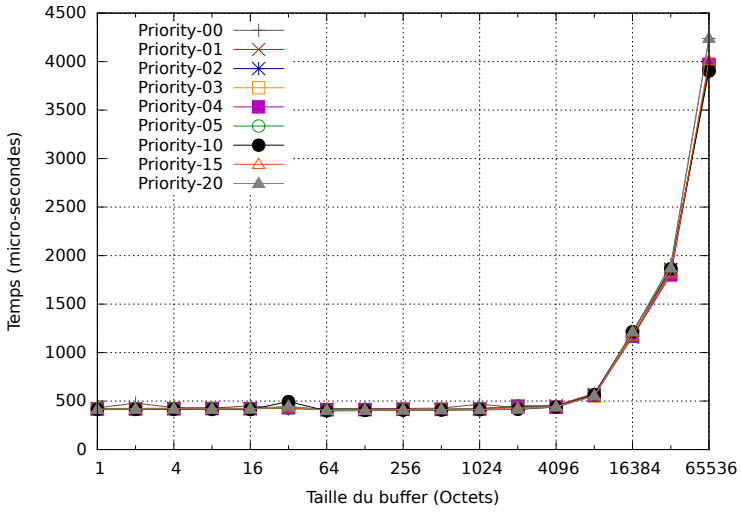


FIGURE 6.4.1 – Temps d'exécution de l'algorithme MPI_Ialltoall avec 64 tâches MPI sur un KNL de 64 cœurs en utilisant le placement « bind »

d'une structure permettant l'ajout dans la liste en $O(1)$) et la suppression en $O(\log(n))$ [54] est possible. Néanmoins, lorsque la *ready list* comporte très peu d'éléments, cela ne change rien aux résultats.

6.4.3 Résultats expérimentaux

Nous lançons les Intel MPI Benchmark avec les algorithmes MPI_Ialltoall, MPI_Igather et MPI_Iscatter avec 64 tâches MPI sur un KNL de 64 cœurs en utilisant le placement des threads de progression « bind », c'est-à-dire que chaque thread de progression est lié sur le même cœur que la tâche MPI auquel il est associé. Les valeurs des priorités des tâches MPI sont définies à 0. Ensuite, différentes valeurs de priorités sont testées pour les threads de progression. Le temps d'exécution pour les algorithmes MPI_Ialltoall, MPI_Igather et MPI_Iscatter est représenté sur les figures 6.4.1, 6.4.2 et 6.4.3 pour différentes tailles de *buffers*. Nous faisons varier différentes valeurs de priorité allant de 0 à 20.

Pour la figure 6.4.1, nous voyons que l'ajout de priorités n'apporte pas de gain de temps par rapport à une politique d'ordonnancement « round-robind » classique. En effet, l'algorithme utilisé pour faire MPI_Ialltoall effectue beaucoup de communications car tous les rangs MPI doivent communiquer entre eux. À chaque fois qu'un thread de progression est ordonnancé, il a une communication à faire.

Pour les figures 6.4.2 et 6.4.3, l'augmentation de la priorité diminue le temps d'exécution. Ceci est dû au fait qu'avec les priorités, les threads de progression sont exécutés plus souvent et peuvent donc faire progresser l'arbre

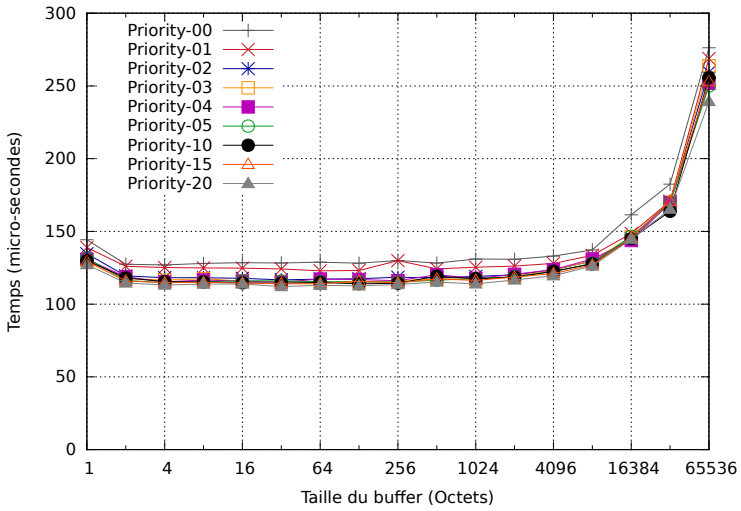


FIGURE 6.4.2 – Temps d'exécution de l'algorithme MPI_Igather avec 64 tâches MPI sur un KNL de 64 cœurs en utilisant le placement « bind »

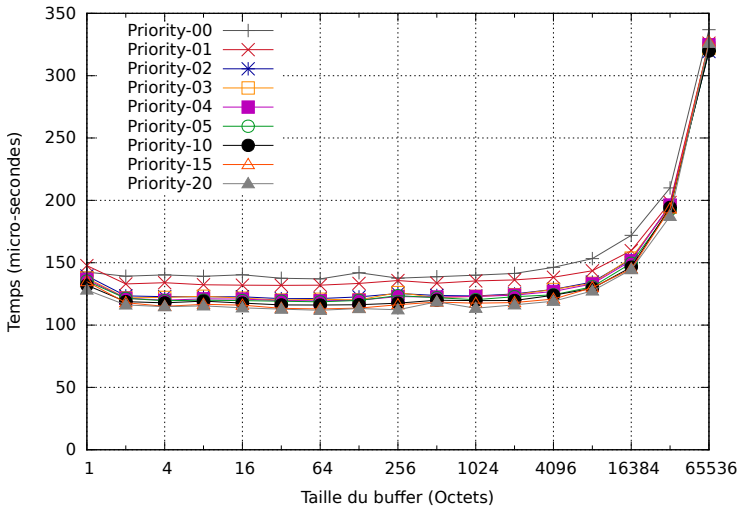


FIGURE 6.4.3 – Temps d'exécution de l'algorithme MPI_Iscatter avec 64 tâches MPI sur un KNL de 64 cœurs en utilisant le placement « bind »

de communication plus rapidement. Nous pouvons aussi voir que le temps d'exécution augmente lorsque la taille des *buffers* augmente.

6.4.4 Conclusion

Dans cette section, nous avons proposé une preuve de concept sur l'utilisation des priorités pour ordonnancer les threads de progression. Nous avons implémenté une politique d'ordonnancement au sein de MPC afin de voir si les priorités avaient un impact sur l'ordonnancement des threads de progression. Nous avons aussi vu que les collectives en arbre *gather* et *scatter* bénéficient de cette politique d'ordonnancement.

6.5 Conclusion

Ce chapitre introduit la nécessité d'ordonnancer les threads de progression afin d'améliorer le recouvrement des communications MPI non-bloquantes. Pour cela, nous proposons une preuve de concept permettant d'avoir une première approche sur l'ordonnancement des threads de progression. Nous avons vu que surcharger les cœurs dédiés à la progression posait un problème car les threads de progression se retrouvent ordonnancés sans aucun contrôle et beaucoup de threads sont exécutés alors qu'ils n'ont rien à faire.

Afin d'améliorer l'ordonnancement des threads de progression sur les cœurs dédiés, nous avons d'abord utilisé un mécanisme permettant de suspendre les threads de progression lorsqu'ils n'ont plus à faire progresser les communications collectives.

Ensuite, nous avons proposé d'ordonnancer les threads de progression de manière statique à l'aide de sémaphores. Nous avons vu que cette façon de faire diminuait le temps d'exécution. Du plus, cela a montré les limites de la première méthode surchargeant la liste de tâches du thread interne à MPC utilisé pour le mécanisme permettant de suspendre les threads de progression. Maintenant, les threads de progression sont suspendus de manière passive, sans attente active.

Enfin, nous avons montré la possibilité de gérer l'ordonnancement des threads de progression de manière dynamique à l'aide de priorité. Cependant, cela nécessiterait de modifier les algorithmes de collective pour y insérer les augmentations de priorités comme nous le faisons avec les sémaphores.

Nous avons vu que le placement des threads de progression permettait d'avoir une meilleure progression dans les chapitres 4 et 5. Nous avons vu dans ce chapitre que nous pouvions améliorer la progression des communications avec un meilleur ordonnancement. Il faudrait donc combiner les deux approches.

Troisième partie

Conclusion

Chapitre 7

Conclusion et perspectives

Les supercalculateurs utilisés dans le HPC sont constitués de plusieurs machines inter-connectées. Généralement, elles sont programmées à l'aide du standard MPI spécifiant une interface de programmation permettant d'échanger des messages entre les machines. Les opérations MPI non-bloquantes ont été proposées pour recouvrir les communications par du calcul afin d'en amortir le coût. Initialement, ces opérations étaient uniquement disponibles pour les opérations entre 2 processus MPI : les communications point-à-point. L'extension des communications non-bloquantes aux opérations impliquant plus de 2 processus MPI, les opérations collectives, est apparue dans la version 3.0 de la norme MPI en 2012 [1]. Cela a ouvert la possibilité de recouvrir les communications collectives non-bloquantes par du calcul. Cependant, ces opérations consomment plus de temps CPU que les opérations point-à-point. L'utilisation d'un seul CPU dédié aux threads de progression n'est donc pas efficace et rend les communications lentes. D'un autre côté, si les communications sont exécutées sur les cœurs applicatifs, aucun recouvrement n'est obtenu.

Nous avons commencé par décrire comment l'évolution de l'architecture des microprocesseurs a conduit aux architectures actuelles. Nous avons ensuite expliqué comment les différents modèles de programmation étaient utilisés pour programmer ces machines ainsi que les différents modèles d'exécutions utilisés dans ce contexte. Nous avons ensuite présenté MPC, qui est l'implémentation MPI sur laquelle nous avons implémenté nos algorithmes.

Nous avons vu que différentes techniques matérielles et logicielles sont utilisées pour faire progresser les communications point-à-point. Cependant, ces techniques ne suffisent pas pour faire progresser les communications collectives MPI non-bloquantes efficacement.

Dans cette thèse, nous proposons différents algorithmes pour améliorer la progression des communications ainsi que leur temps d'exécution. Pour cela, nous avons abordé ce problème sous plusieurs angles. Nous nous sommes concentrés sur le placement des threads de progression générés par les collectives MPI non-bloquantes. Nous avons proposé un placement des tâches MPI dans

le but de pouvoir placer les threads de progression en tenant compte de la topologie de la machine. Nous avons ensuite étudié le placement des threads de progression sur les *Hyper-Threads*. Cette étude a montré que placer les threads de progression sur les *Hyper-Threads* pour des communications intra-nœud résultait à une perte de performance. C'est pourquoi, nous nous sommes concentrés sur les communications intra-nœuds. Nous avons proposé l'algorithme « split-tree » permettant d'améliorer la progression des collectives en arbre. Nous avons aussi proposé un algorithme permettant d'améliorer les opérations collectives en chaîne.

Tous ces travaux reposent sur le placement des threads de progression sur des cœurs dédiés à la progression des communications. Il y a généralement plus de threads de progression que de cœurs dédiés, c'est pourquoi les cœurs dédiés se retrouvent surchargés par les threads de progression. Nous proposons des mécanismes permettant d'améliorer l'ordonnancement des threads de progression. Ces mécanismes étudiés sur des algorithmes en chaîne sont des preuves de concept. Nous pouvons les étendre à d'autres opérations collectives.

Contributions

Les contributions de cette thèse s'articulent autour de trois grands thèmes : le placement statique, le placement dynamique et l'ordonnancement des threads de progression.

Placement statique

Nous avons d'abord proposé deux algorithmes de placement dans le but d'améliorer le recouvrement des communications par du calcul ainsi que le temps d'exécution des applications utilisant les collectives MPI non-bloquantes. L'un pour les tâches MPI, afin d'avoir un placement prenant en compte les nœuds NUMA, l'autre pour placer les threads de progression sur les cœurs libres par nœuds NUMA. Nous avons pu observer que ces placements ont permis de mieux recouvrir les communications par du calcul ainsi qu'un meilleur temps d'exécution. Néanmoins, il est nécessaire d'avoir un certain nombre de cœurs dédiés aux communications.

Ensuite, nous nous sommes concentrés sur l'évaluation du placement des threads de progression sur les *Hyper-Threads*. Cela nous a permis de voir que l'utilisation de cette technologie est efficace pour les communications inter-nœuds mais qu'au contraire, son utilisation pour les communications intra-nœuds dégrade les performances. Nous avons expliqué que cela est dû au *cache thrashing*.

Le placement optimal des threads de progression effectuant des communications inter-nœuds, et des communications intra-nœuds, n'est pas le même.

Pour les communications inter-nœuds, utiliser les Hyper-Threads est une bonne solution. En revanche, pour les communications intra-nœuds, des cœurs doivent être dédiés pour les threads de progression.

Placement dynamique

Nous nous sommes intéressés aux placements dynamiques des threads de progression en fonctions des algorithmes de collectives utilisés.

Algorithmes en arbre : Nous avons d'abord proposé un algorithme permettant d'améliorer le recouvrement des communications collectives MPI non-bloquantes pour les opérations basées sur un arbre de communication (broadcast, reduce, scatter, gather, allreduce).

Pour recouvrir le coût des communications collectives par du calcul, les approches consistant à utiliser un thread de progression par tâche MPI souffrent de la compétition entre les communications et le calcul. Celles qui consistent à allouer des cœurs dédiés aux communications souffrent d'une baisse des performances du temps de communication quand les communications d'une collective sont repliées sur peu de cœurs dédiés.

Nous avons proposé l'algorithme « split-tree » qui combine le meilleur des deux approches. Il scinde l'arbre des communications et exécute la partie la plus lourde en communications sur les cœurs applicatifs, pour bénéficier de plus de parallélisme. En revanche, la partie la plus légère en communications, mais occupant le plus grand nombre d'étapes, est exécutée sur les cœurs de communications afin de les recouvrir par du calcul. Nous avons modélisé l'algorithme pour démontrer sa pertinence et affiner ses paramètres. Nous l'avons implémenté dans MPC et évalué ses performances sur des processeurs manycores (Intel KNL et Skylake).

Nous avons proposé ensuite un modèle de coût pour l'algorithme « split-tree ». Ce modèle nous permet de prédire les performances du temps d'exécution $calcul + communications$ quand nous scindons l'arbre de communication pour exécuter un nombre d'étapes S sur les cœurs applicatifs et un nombre d'étapes $H(N) - S$ sur les cœurs de communications.

Algorithmes en chaîne : Nous avons ensuite vu comment améliorer le temps d'exécution des algorithmes en chaîne (scan et exscan) par rapport à nos précédents algorithmes de placement de threads de progression. Il s'agit du placement « pair-impair ».

Nous avons vu que pour obtenir un bon temps d'exécution, il faut faire un compromis entre le recouvrement des communications par le calcul et le temps d'exécution.

Ordonnancement des threads de progression

Il est nécessaire d'ordonnancer les threads de progression afin d'améliorer le recouvrement des communications MPI non-bloquantes. Nous avons vu que surcharger les cœurs dédiés à la progression posait un problème car les threads de progression se retrouvent ordonnancés sans aucun contrôle et beaucoup de threads sont exécutés alors qu'ils n'ont rien à faire.

Afin d'améliorer l'ordonnancement des threads de progression sur les cœurs dédiés, nous avons d'abord utilisé un mécanisme permettant de suspendre les threads de progression lorsqu'ils n'ont plus à faire progresser les communications collectives.

Ensuite, nous avons proposé d'ordonnancer les threads de progression de manière statique à l'aide de sémaphores. Nous avons vu que cette façon de faire diminuait le temps d'exécution. De plus, cela a montré les limites de la première méthode qui surchargeait de travail le thread interne à MPC. Celui-ci est utilisé pour le mécanisme permettant de suspendre les threads de progression. Maintenant, les threads de progression sont suspendus de manière passive, sans attente active.

Enfin, nous avons montré la possibilité de gérer l'ordonnancement des threads de progression de manière dynamique à l'aide de priorités. Cependant, cela nécessiterait de modifier les algorithmes de collective pour y insérer les augmentations de priorités comme nous le faisons avec les sémaphores.

Perspectives

Ces travaux reposant sur le placement et l'ordonnancement des threads de progression ouvrent des perspectives intéressantes sur le moyen et long terme.

« Split-tree adaptatif » Dans la section 5.1, nous avons proposé l'algorithme « split-tree » permettant de prédire les performances du temps d'exécution $calcul + communications$ quand nous scindons l'arbre de communication pour exécuter S étapes sur les cœurs applicatifs et $H(N) - S$ étapes sur les cœurs de communications. Cependant, pour modéliser le calcul et le recouvrement des communications, nous considérons que le développeur de l'application a essayé d'obtenir un recouvrement parfait des communications, et que la charge de calcul est suffisante pour que le temps du calcul soit équivalent au temps d'une opération collective bloquante sur tous les cœurs. Nous pouvons étendre l'algorithme « split-tree » pour avoir un algorithme adaptatif au volume de calcul à recouvrir.

Prenons l'exemple d'un algorithme itératif utilisant une collective MPI non-bloquante en arbre. Lors d'une itération, nous pouvons calculer le temps de calcul entre l'appel à la collective MPI non-bloquante et l'appel à MPI_Wait. Ensuite, nous pouvons utiliser ce temps dans notre modèle afin de prendre

en compte le volume de calcul par rapport au volume de communication. Si nous avons un temps de calcul supérieur au temps que mettrait la même collective de manière bloquante, notre modèle prédira d'utiliser que les cœurs de communications et de ne pas utiliser les cœurs applicatifs. À l'inverse, si le temps de calcul est très inférieur au temps de communication, notre modèle de coût prédira de n'utiliser que les cœurs applicatifs puisqu'il n'y aura pas de calcul à recouvrir. Dans le cas où le temps de calcul et le temps de communication seraient proche, notre modèle prédira une valeur de S comprise entre 1 et $H(N)$. Ainsi nous aurons un modèle qui s'adapte au temps de calcul.

Placement et ordonnancement générique : Tout au long de cette thèse, nous avons cherché à améliorer le recouvrement des collectives MPI non-bloquantes. Nous avons optimisé le placement des threads de progression pour les algorithmes basés sur des communications en arbre et en chaîne. Néanmoins, il serait judicieux de faire la même chose pour toutes les collectives MPI non-bloquantes.

Sur le moyen terme, il est possible d'utiliser les sémaphores pour forcer l'ordonnancement des threads de progression au bon moment afin qu'ils ne se perturbent pas entre eux. Sur le long terme, nous pensons plutôt qu'il faut réaliser toute la progression des communications collectives non-bloquantes à l'aide de tâches car elles sont plus flexibles à ordonnancer que les threads.

Progression des communications avec un moteur de tâches en intra-noeud : Pour faire progresser les communications collectives MPI non-bloquantes en intra-noeud, nous travaillons avec des threads de progression. Cette technique atteint ses limites lorsqu'il s'agit d'ordonnancer certains algorithmes. Par exemple, si nous comparons deux algorithmes de communications basés sur un arbre, *reduce* et *broadcast*, en effectuant la progression des communications par des threads ou par des tâches, l'ordonnancement optimal n'est pas le même.

Sur la figure 7.0.1, nous pouvons voir l'ordonnancement de l'arbre de communications pour 16 tâches MPI pour la collective *broadcast*.

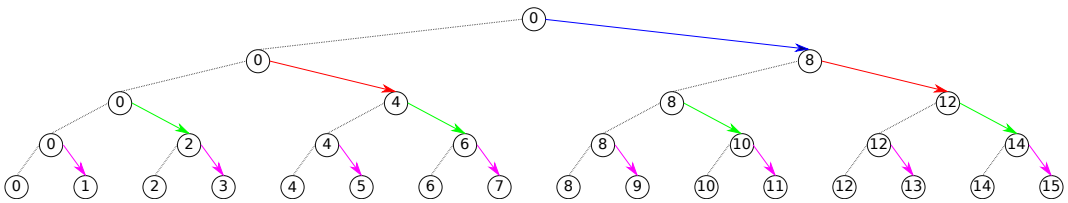


FIGURE 7.0.1 – Arbre de progression des communications pour l'algorithme *broadcast* avec des threads. Les flèches représentent un envoi de message. Les couleurs des flèches : bleues, rouges, vertes, et mangenta représentent respectivement les 4 étapes nécessaires à l'exécution de l'algorithme

Sur la première étape représentée par les flèches bleues, nous pouvons voir que nous avons d'abord le rang 0 qui envoie un message au rang 8, puis que le rang 0 et le rang 8 envoie respectivement un message au rang 4 et au rang 12 à la deuxième étape représentée en rouge. Ensuite les rangs MPI 0, 4, 8 et 12 envoient un message respectivement aux tâches MPI 2, 6, 10 et 14 à la troisième étape représentée en vert. Enfin, les rangs 0, 2, 4, 6, 8, 10, 12 et 14 envoient des messages respectivement aux rangs 1, 3, 5, 7, 9, 11, 13 et 15 à la quatrième et dernière étape représentée en magenta. Nous pouvons voir que l'ordonnancement optimal de cet algorithme nécessite 8 cœurs dédiés aux threads de progression.

En revanche, si nous avions un moteur de tâches qui s'occuperait de faire la progression des communications entre les tâches MPI, nous aurions l'arbre de progression illustré par la figure 7.0.2.

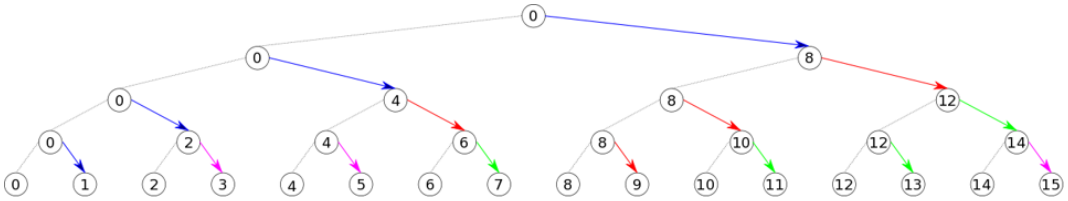


FIGURE 7.0.2 – Arbre de progression des communications pour l'algorithme *broadcast* des tâches. Les flèches représentent un envoi de message. Les couleurs des flèches : bleues, rouges, vertes, et magenta représentent respectivement les 4 étapes nécessaires à l'exécution de l'algorithme

Les étapes ne seraient pas décidées par le fait qu'un même thread de progression ne peut pas envoyer un message en même temps (memcpy en intra-nœud). Mais bien par les dépendances entre les échanges de messages. Nous remarquons qu'avec cet ordonnancement des tâches de progression, l'ordonnancement optimal ne nécessite que 4 cœurs dédiés aux tâches de progression pour exécuter l'algorithme *broadcast* alors qu'avec des threads de progression, nous aurions besoin de 8 cœurs dédiés.

Nous n'avons pas ce phénomène lorsque nous ordonnançons l'algorithme *reduce* (figure 7.0.3) car l'ordonnancement avec des threads ou des tâches de progression nous donne le même arbre de progression.

Les tâches de progression sont donc plus flexibles dans leur ordonnancement et nous pensons que c'est la bonne solution pour faire progresser les collectives MPI non-bloquantes. Cependant cette technique ne fonctionne qu'en intra-nœud car nous avons besoin d'avoir l'arbre de communication définissant les dépendances entre les tâches dans sa globalité.

Progression des communications avec un moteur de tâches en inter nœuds : Étant donné que nous ne pouvons pas avoir un arbre de progression

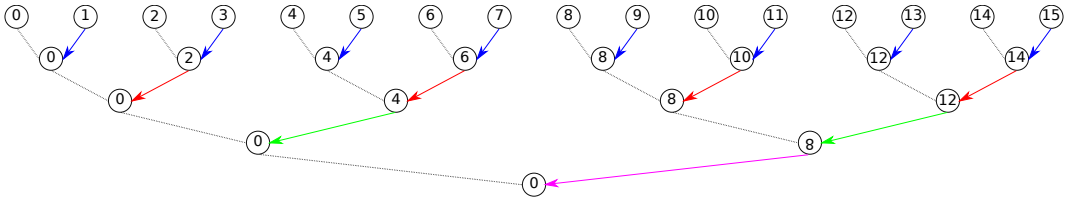


FIGURE 7.0.3 – Arbre de communications pour l'algorithme *reduce* avec des threads ou des tâches. Les flèches représentent un envoi de message. Les couleurs des flèches : bleues, rouges, vertes, et magenta représentent respectivement les 4 étapes nécessaires à l'exécution de l'algorithme.

globale sur tous les nœuds, nous pensons que faire progresser les communications MPI non-bloquantes à l'aide de tâches de communication réagissant aux évènements sur le réseau est une bonne solution. Les tâches seraient des tâches élémentaires de communication constituant les communications point-à-point (flèches colorées sur la figure 7.0.2). En pratique, nous pensons intégrer ce mécanisme dans PIOMAN [29], [34] qui gère déjà les communications point-à-point en ayant des tâches qui scrutent le réseau appellant un *callback* pour réagir à un évènement réseau. Nous pourions utiliser ce mécanisme pour réaliser les collectives MPI non-bloquantes. Par exemple, lors d'un broadcast, lorsque les nœuds intermédiaires recevraient les données de leur père, cela déclencherait l'envoi des données à leurs fils. De cette manière, l'ordonnancement des tâches serait plus flexible.

Bibliographie

[1] MPI Forum, "MPI : A Message-Passing Interface Standard Version 3.0", 2012.

[2] J.-M. Alimi, V. Bouillot, Y. Rasera, V. Reverdy, P.-S. Corasaniti, I. Balmès, S. Requena, X. Delaruelle et J.-N. Richet, "First-ever Full Observable Universe Simulation", in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, sér. SC '12, Salt Lake City, Utah : IEEE Computer Society Press, 2012, 73 :1–73 :11, isbn : 978-1-4673-0804-5. adresse : http://dl.acm.org/citation.cfm?id=2388996.2389096.

[3] G. E. Moore, *Cramming more components onto integrated circuits, Electronics, Vol. 38, No. 8*, avr. 1965.

[4] M. Roser et H. Ritchie, *Technological Progress, Published online at OurWorldInData.org. Retrieved from : 'https ://ourworldindata.org/technological-progress' [Online Resource]*, 2018.

[5] M. K. Qureshi, V. Srinivasan et J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology", in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, sér. ISCA '09, Austin, TX, USA : ACM, 2009, p. 24–33, isbn : 978-1-60558-526-0. doi : 10.1145/1555754.1555760. adresse : http://doi.acm.org/10.1145/1555754.1555760.

[6] B. Mutnury, F. Paglia, J. Mobley, G. K. Singh et R. Bellomio, "QuickPath Interconnect (QPI) design and analysis in high speed servers", in *19th Topical Meeting on Electrical Performance of Electronic Packaging and Systems*, oct. 2010, p. 265–268. doi : 10.1109/EPEPS.2010.5642789.

[7] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm et D. M. Tullsen, "Simultaneous multithreading : a platform for next-generation processors", *IEEE Micro*, t. 17, n° 5, p. 12–19, sept. 1997, issn : 0272-1732. doi : 10.1109/40.621209.

[8] J. L. Hennessy et D. A. Patterson, *Computer Architecture, Fifth Edition : A Quantitative Approach*, 5th. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2011, p. 148–156, isbn : 012383872X, 9780123838728.

[9] J. D. Warnock, J. M. Keaty, J. Petrovick, J. G. Clabes, C. J. Kircher, B. L. Krauter, P. J. Restle, B. A. Zoric et C. J. Anderson, "The circuit and physical design of the POWER4 microprocessor", *IBM Journal of Research and Development*, t. 46, n° 1, p. 27–51, jan. 2002, ISSN : 0018-8646. DOI : 10.1147/rd.461.0027.

[10] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault et R. Namyst, "hwloc : a Generic Framework for Managing Hardware Affinities in HPC Applications", in *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, Pisa, Italia : IEEE Computer Society Press, fév. 2010, p. 180–186. DOI : 10.1109/PDP.2010.67. adresse : http://hal.inria.fr/inria-00429889.

[11] J. Jeffers et J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*, 1st. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2013, ISBN : 9780124104143, 9780124104945.

[12] J. Jeffers, J. Reinders et A. Sodani, *Intel Xeon Phi Processor High Performance Programming : Knights Landing Edition 2Nd Edition*, 2nd. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2016, ISBN : 0128091940, 9780128091944.

[13] Kalray, *Kalray MPPA 2-256*, 2013. adresse : https://www.kalrayinc.com/products/.

[14] M. J. Flynn, "Some computer organizations and their effectiveness", *IEEE Transactions on Computers*, t. C-21, n° 9, p. 948–960, sept. 1972, ISSN : 0018-9340. DOI : 10.1109/TC.1972.5009071.

[15] Top500, *Top500*. adresse : https://www.top500.org/.

[16] J. Dongarra, "The linpack benchmark : an explanation", in *Proceedings of the 1st International Conference on Supercomputing*, London, UK, UK : Springer-Verlag, 1988, p. 456–474, ISBN : 3-540-18991-2. adresse : http://dl.acm.org/citation.cfm?id=647970.742568.

[17] T. El-Ghazawi et L. Smith, "Upc : unified parallel c", in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, sér. SC '06, Tampa, Florida : ACM, 2006, ISBN : 0-7695-2700-0. DOI : 10.1145/1188455.1188483. adresse : http://doi.acm.org/10.1145/1188455.1188483.

[18] R. H. B. Netzer et B. P. Miller, "What are race conditions ? : some issues and formalizations", *ACM Lett. Program. Lang. Syst.*, t. 1, n° 1, p. 74–88, mar. 1992, ISSN : 1057-4514. DOI : 10.1145/130616.130623. adresse : http://doi.acm.org/10.1145/130616.130623.

[19] I. 9.-1. IEEE POSIX 1003.1c-1995, "POSIX 1003.1c Threading", 1995. adresse : https://standards.ieee.org/findstds/standard/1003.1c-1995.html.

[20] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham et T. S. Woodall, "Open MPI : goals, concept, and design of a next generation MPI implementation", in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, sept. 2004, p. 97–104.

[21] MPICH Team, "MPICH", 1992. adresse : https://www.mpich.org/.

[22] W. 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*, London, UK, UK : Springer-Verlag, 2002, p. 7–, isbn : 3-540-44296-0. adresse : http://dl.acm.org/citation.cfm?id=648139.749473.

[23] D. Panda, K. Tomko, K. Schulz et A. Majumdar, "The mvapich project : evolution and sustainability of an open source production quality mpi library for hpc", nov. 2013.

[24] O. Aumage, E. Brunet, N. Furmento et R. Namyst, "NewMadeleine : a Fast Communication Scheduling Engine for High Performance Networks", in *Workshop on Communication Architecture for Clusters (CAC 2007), workshop held in conjunction with IPDPS 2007*, Long Beach, California, United States, mar. 2007. adresse : https://hal.inria.fr/inria-00127356.

[25] M. Pérache, H. Jourdren et R. Namyst, "MPC : A Unified Parallel Runtime for Clusters of NUMA Machines", in *the 14th International Euro-Par Conference*, Springer, éd., sér. LNCS, t. 5168, Las Palmas de Gran Canaria, Spain, août 2008, p. 78–88. doi : 10.1007/978-3-540-85451-7_9. adresse : https://hal.inria.fr/inria-00422229.

[26] A. S. Tanenbaum et H. Bos, *Modern Operating Systems*, 4th. Upper Saddle River, NJ, USA : Prentice Hall Press, 2014, p. 152–153, isbn : 013359162X, 9780133591620.

[27] CEA, CNRS, INRIA, "CeCILL-C FREE SOFTWARE LICENSE AGREEMENT", 2013. adresse : http://www.cecill.info/licences/.

[28] M. Tchiboukdjian, P. Carribault et M. Pérache, "Hierarchical Local Storage : Exploiting Flexible User-Data Sharing Between MPI Tasks", in *IEEE International Parallel and Distributed Processing (IPDPS'12)*, 2012.

[29] F. Trahay, "De l'interaction des communications et de l'ordonnancement de threads au sein des grappes de machines multi-cœurs", 2009BOR13870, thèse de doct., 2009. adresse : http://www.theses.fr/2009BOR13870/document.

[30] S. Sur, H. Jin, L. Chai et D. Panda, "RDMA read based rendezvous protocol for MPI over InfiniBand : design alternatives and benefits", in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, ACM New York, NY, USA, 2006, p. 32–39.

[31] M. J. Rashti et A. Afsahi, "Improving communication progress and overlap in MPI Rendezvous protocol over RDMA-enabled interconnects", in *High Performance Computing Systems and Applications, 2008. HPCS 2008. 22nd International Symposium on*, IEEE, 2008, p. 95–101.

[32] T. Hoefler et A. Lumsdaine, "Message Progression in Parallel Computing - To Thread or not to Thread ?", in *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, Tsukuba, Japan : IEEE Computer Society, oct. 2008, isbn : 978-1-4244-2640.

[33] P. Lai, P. Balaji, R. Thakur et D. Panda, "ProOnE : A General Purpose Protocol Onload Engine for Multi- and Many-Core Architectures", None, juin 2009.

[34] A. Denis, "pioman : a Generic Framework for Asynchronous Progression and Multithreaded Communications", Anglais, in *IEEE International Conference on Cluster Computing (IEEE Cluster)*, Madrid, Espagne, sept. 2014. adresse : http://hal.inria.fr/hal-01064652.

[35] M. Si, A. Peña, P. Balaji, M. Takagi et Y. Ishikawa, "MT-MPI : multithreaded MPI for many-core environments", in *Proceedings of the International Conference on Supercomputing*, juin 2014, isbn : 978-1-4503-2642-1.

[36] M. Sergent, M. Dagrada, P. Carribault, J. Jaeger, M. Pérache et G. Papauré, "Efficient communication/computation overlap with mpi+openmp runtimes collaboration", in *Euro-Par 2018 : Parallel Processing*, M. Aldinucci, L. Padovani et M. Torquati, éds., Cham : Springer International Publishing, 2018, p. 560–572, isbn : 978-3-319-96983-1.

[37] G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow et Y. Zheng, "Optimization of MPI Collective Communication on BlueGene/L Systems", in *Proceedings of the 19th Annual International Conference on Supercomputing*, sér. ICS '05, Cambridge, Massachusetts : ACM, 2005, p. 253–262, isbn : 1-59593-167-8. doi : 10.1145/1088149.1088183. adresse : http://doi.acm.org/10.1145/1088149.1088183.

[38] W. Yu, D. Buntinas, R. L. Graham et D. K. Panda, "Efficient and scalable barrier over quadrics and myrinet with a new nic-based collective message passing protocol", in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, avr. 2004, p. 182–. DOI : 10.1109/IPDPS.2004.1303191.

[39] F. Petrini, W.-c. Feng, A. Hoisie, S. Coll et E. Frachtenberg, "The quadrics network : high-performance clustering technology", *IEEE Micro*, t. 22, n° 1, p. 46–57, jan. 2002, ISSN : 0272-1732. DOI : 10.1109/40.988689.

[40] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic et W.-K. Su, "Myrinet : a gigabit-per-second local area network", *IEEE Micro*, t. 15, n° 1, p. 29–36, fév. 1995, ISSN : 0272-1732. DOI : 10.1109/40.342015.

[41] S. Derradji, T. Palfer-Sollier, J.-P. Panziera, A. Poudes et F. Wellenreiter, "The BXI Interconnect architecture", in *High-Performance Inter-connects (HOTI)*, 2015 IEEE 23th Annual Symposium, 2015.

[42] R. B. Brightwell, "Portals 4 : enabling application/architecture co-design for high-performance interconnects.", août 2012.

[43] T. Ma, G. Bosilca, A. Bouteiller, B. Goglin, J. M. Squyres et J. J. Dongarra, "Kernel Assisted Collective Intra-node MPI Communication Among Multi-core and Many-core CPUs", in *40th International Conference on Parallel Processing (ICPP-2011)*, IEEE, éd., Taipei, Taiwan, sept. 2011. DOI : 10.1109/ICPP.2011.29. adresse : https://hal.inria.fr/inria-00602877.

[44] B. Goglin et S. Moreaud, "Knem : a generic and scalable kernel-assisted intra-node mpi communication framework", *J. Parallel Distrib. Comput.*, t. 73, n° 2, p. 176–188, fév. 2013, ISSN : 0743-7315. DOI : 10.1016/j.jpdc.2012.09.016. adresse : http://dx.doi.org/10.1016/j.jpdc.2012.09.016.

[45] T. Hoefler, A. Lumsdaine et W. Rehm, "Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI", in *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*, Reno, USA : IEEE Computer Society/ACM, nov. 2007.

[46] T. Hoefler et A. Lumsdaine, "Optimizing non-blocking Collective Operations for InfiniBand", in *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, CAC'08 Workshop*, Miami, FL, avr. 2008, ISBN : 978-1-4244-1694-3.

[47] Intel, "IMB-NBC benchmarks", Accessed : 2018-03-29. adresse : https://software.intel.com/fr-fr/node/561946.

[48] A. Denis et F. Trahay, "Mpi overlap : benchmark and analysis", in *2016 45th International Conference on Parallel Processing (ICPP)*, août 2016, p. 258–267. doi : 10.1109/ICPP.2016.37.

[49] Intel, "Measuring Communication and Computation Overlap", Accessed : 2018-03-29. adresse : https://software.intel.com/fr-fr/node/561947.

[50] J. Meng et K. Skadron, "Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling", in *2009 IEEE International Conference on Computer Design*, oct. 2009, p. 282–288. doi : 10.1109/ICCD.2009.5413143.

[51] D. Terpstra, H. Jagode, H. You et J. Dongarra, "Collecting Performance Data with PAPI-C", in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz et W. E. Nagel, éds., Berlin, Heidelberg : Springer Berlin Heidelberg, 2010, p. 157–173, isbn : 978-3-642-11261-4.

[52] J. C. de Kergommeaux et B. de Oliveira Stein, "Pajé : an extensible environment for visualizing multi-threaded programs executions", in *Euro-Par 2000 Parallel Processing*, A. Bode, T. Ludwig, W. Karl et R. Wismüller, éds., Berlin, Heidelberg : Springer Berlin Heidelberg, 2000, p. 133–140, isbn : 978-3-540-44520-3.

[53] K. Coulomb, M. Faverge, J. Jazeix, O. Lagrasse, J. Marcoueille, P. Noisette, A. Redondy, S. Thibault et C. Vuchener, *Visual Trace Explorer*, nov. 2016. adresse : http://vite.gforge.inria.fr/.

[54] M. L. Fredman, R. Sedgewick, D. D. Sleator et R. E. Tarjan, "The pairing heap : a new form of self-adjusting heap", *Algorithmica*, t. 1, n° 1, p. 111–129, 1er nov. 1986, issn : 1432-0541. doi : 10.1007/BF01840439. adresse : https://doi.org/10.1007/BF01840439.