A study on progression of MPI communications using dedicated resources

Florian Reynier

To cite this version:

Florian Reynier. A study on progression of MPI communications using dedicated resources. Distributed, Parallel, and Cluster Computing [cs.DC]. Université de Bordeaux, 2022. English. NNT : 2022BORD0206 . tel-04023709

HAL Id: tel-04023709
https://theses.hal.science/tel-04023709v1

Submitted on 10 Mar 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR
DE L'UNIVERSITÉ DE BORDEAUX

ECOLE DOCTORALE DE MATHEMATIQUES ET

D'INFORMATIQUE

Par **Florian REYNIER**

A study on progression of MPI communications using dedicated resources

Sous la direction de : **Emmanuel JEANNOT**
Co-directeur : **Alexandre DENIS**

Soutenue le 24 juin 2022

Membres du jury :

M. Michaël KRAJECKI	Professeur d'université	Université de Reims Champagne-Ardenne	Rapporteur
Mme. Christine MORIN	Directrice de Recherche	INRIA	Examinatrice
M. Raymond NAMYST	Professeur d'université	Université de Bordeaux	Président du jury
M. Alexandre DENIS	Chargé de Recherche	INRIA	Co-Directeur de thèse
M. Emmanuel JEANNOT	Directeur de Recherche	INRIA	Directeur de thèse
M. Julien JAEGER	Docteur	CEA	Encadrant
M. Anthony SKJELLUM	Professeur d'université	University of Tennessee Chattanooga	Rapporteur

A study on progression of MPI communications using dedicated resources

Résumé : De nos jours, MPI est *de facto* le standard pour la programmation à mémoire distribuée pour les supercalculateurs. Les communications non bloquantes sont un des modèles proposés par le standard MPI.
Ces opérations peuvent être utilisées pour recouvrir les communications avec du calcul (ou d'autres communications) afin d'amortir leurs coûts. Cependant, pour être utilisées efficacement, ces opérations nécessitent une progression asynchrone pouvant régulièrement utiliser un montant non négligeable de ressources de calcul (particulièrement les collectives non bloquantes). De plus, partager les ressources de calcul avec l'application peut provoquer un ralentissement global. Les mécanismes utilisés pour cette progression asynchrone parviennent difficilement à concilier un bon recouvrement en gardant un impact minimal sur l'application, ce qui raréfie leur utilisation.
Afin de résoudre ces différents problèmes, nous avons suivi plusieurs étapes. Premièrement, nous proposons une étude approfondie de la progression asynchrone dans les implémentations MPI, en utilisant de nouvelles métriques se concentrant sur l'évaluation des mécanismes de progression et de leur impact sur le système global.
Après avoir exposé les faiblesses de ces implémentations MPI, nous proposons une nouvelle solution pour la progression des collectives non bloquantes en utilisant des cœurs dédiés combinés à des algorithmes de collectives basés sur des évènements. Nous avons mesuré l'efficacité de cette solution en utilisant nos métriques, pour nous comparer avec les implémentations MPI étudiées dans la première étape. Enfin, nous avons développé un modèle permettant de prédire le gain potentiel et le surcout induit par l'utilisation d'opérations non bloquantes avec des cœurs dédiés. Ce modèle peut être utilisé pour évaluer l'utilité de transformer une application basée sur des opérations bloquantes en opérations non bloquantes pour bénéficier du recouvrement. Nous évaluons ce modèle sur plusieurs benchmarks.

Mots-clés : MPI, Progression, Non bloquantes, Collectives

A study on progression of MPI communications using dedicated resources

Abstract: Nowadays, MPI is the *de-facto* standard for distributed-memory parallelism on supercomputers. One of the communication models offered by the MPI standard is MPI nonblocking communications.
These communications can be used to overlap communication with computation (or other communications) in order to reduce their impact. However, to perform efficiently, these operations require asynchronous progression, which can need non negligible amount of computation resources regularly (especially for nonblocking collectives). However, sharing the compute resources with the application may cause an overall slowdown. The current mechanisms used to achieve this asynchronous progression struggle to reconcile a good overlap and minimal impact on the application, which leads to nonblocking collective operations being very seldom used in applications.
To address this issue, we followed several steps. First, we proposed a thorough study of asynchronous progression in MPI implementations using newly defined metrics, focusing on the evaluation of progression mechanisms and their impact on the global runtime. After exposing the shortcomings of these MPI implementations, we propose a new solution for the progression of nonblocking collectives using dedicated cores combined with event-based collective algorithms. We measured the efficiency of this solution using our metrics, to compare ourselves with the MPI implementations studied in the first step. Finally, we developed a model to predict the potential gain and the overhead induced by the use of nonblocking operations with a dedicated core. This model can be used to evaluate the usefulness of transforming an application based on blocking operation to nonblocking ones to benefit from overlap. We evaluate this model on several benchmarks.
Keywords: MPI, Progression, Nonblocking, Collectives

Contents

0.1 Introduction 9
0.1.1 Contributions 9
0.2 Évaluer le recouvrement des collectives non bloquantes 10
0.2.1 Définition et mesure de la performance d'une collective 10
0.2.2 Définition de métriques 10
0.2.3 Étude des mécanismes de progression 12
0.3 Un cœur dédié pour la progression des communications 12
0.3.1 Utilisation d'un cœur dédié pour la progression des communications 12
0.3.2 Implémentation d'algorithmes de collectives non bloquantes . . . 12
0.3.3 Étude de la progression avec cœur dédié 13
0.4 Modèle de performance des applications pour l'utilisation de cœur dédié . 13
0.4.1 Vue globale du modèle 13
0.4.2 Validation du modèle 14

1 Introduction 17
1.1 Objectives of this thesis 18
1.1.1 Organisation of the document 18

2 Context around high performance computing and the problematic 21
2.1 Simulation in of high performance computing 21
2.2 The evolution of hardware 22
2.2.1 Computer-scale developments 22
2.2.2 Advent of distributed computing 24
2.3 The evolution of software 26
2.3.1 Managing multiple models resource usage 26
2.3.2 The fork/join model 26
2.3.3 The task model 29
2.3.4 Message passing model 30
2.4 Problematic of this thesis 35

3 State of the art 37
3.1 Benchmarking MPI 38
3.1.1 Benchmarking of blocking communications 38
3.1.2 Benchmarking of nonblocking communications 42
3.1.3 Overview and link to the progression mechanisms 43
3.2 Progression mechanism for overlap 44
3.2.1 Point-to-point communication overlap 44

3.2.2 Collective communication overlap 46
3.2.3 Overview of progression 46
3.3 Modelling hybrid applications performance 47
3.3.1 Analysing and modelling MPI performance 47
3.3.2 OpenMP scaling of applications 48
3.4 Summary of items to address in contribution 48

4 Evaluating the overlap of nonblocking collectives 49
4.1 Metrics for Overlap and its Impact on Performance 50
4.1.1 Measuring Overlap 50
4.1.2 Interference between Computation and Communication 52
4.1.3 Using these Metrics to Diagnose Bad Overlap 54
4.2 Our methodology for assessing computation/communication overlap of NBC 56
4.2.1 Multiple MPI processes Time Variation 56
4.2.2 Clock Synchronisation and Barriers 56
4.2.3 Fixed Computation and Communication Time 57
4.3 Presentation of the BenchNBC Benchmark 57
4.3.1 Benchmark Implementation 58
4.3.2 Metrics Display 59
4.3.3 Experimental setups 59
4.4 Benchmark cases study 60
4.4.1 Case study: OpenMPI/InfiniBand 60
4.4.2 Case study: MPICH with `MPICH_ASYNC_PROGRESS=1` 63
4.4.3 Comparison with state-of-the-art 65
4.5 Survey of Overlap Benchmark Results with Various MPI Implementations and Networks 68
4.5.1 Results with basic configuration 68
4.5.2 In depth interpretation of these results using complementary metrics 72
4.5.3 Results with asynchronous progression enabled 72
4.5.4 Results with dedicated core for progression 75
4.5.5 Results with hardware-based progression 76

5 A dedicated core mechanism for communication progression 77
5.1 Using a dedicated core for asynchronous communication progression . . . 78
5.1.1 Analysing core dedication relevance for progression 78
5.1.2 Implementation of dedicated cores 79
5.1.3 Dedicated core cohabitation with OpenMP threads 81
5.2 Implementing NBC in NewMadeleine 82
5.2.1 NewMadeleine algorithms for nonblocking collectives 82
5.2.2 Implementation of collectives in NewMadeleine 84
5.3 Pioman dedicated core benchmarking 88
5.3.1 MadMPI dedicated core performance using BenchNBC 88
5.3.2 Analysis of collective behaviour using concurrency and all ranks metrics 88

6 A model for application performance with a dedicated core for communication progression **95**
6.1 Modelling dedicated core impact on hybrid applications 96
6.1.1 Overview of the performance model 96
6.1.2 Impact of a dedicated core on computation 97
6.1.3 Modeling MPI performance with dedicated core 100
6.1.4 Global model with MPI and computation 103
6.2 Gathering applications information to use the model 104
6.3 Evaluation of the model 104
6.4 Using the model 109
6.4.1 Understanding the computation-slowdown/communication-speedup ratio 109
6.4.2 Evaluating the effectiveness of nonblocking communication 110
6.4.3 Potential gain of transforming blocking call to nonblocking 111

7 Conclusion **115**
7.1 Contributions 116
7.1.1 Nonblocking collective benchmarking 116
7.1.2 Dedicated core for progression 117
7.1.3 Hybrid application with dedicated core modelling 117
7.1.4 Answers about the dedicated core usage 117
7.2 Perspectives 118
7.2.1 Use of MPI nonblocking collectives benchmark for MPI library improvement 118
7.2.2 Evolution of the dedicated core impact model 119
7.2.3 Multiple runtime centralisation of resources management 119

Je voudrais tout d'abords remercier mes directeur et co-directeur de thèse Emmanuel et Alexandre, ainsi que Julien pour leur encadrement et disponibilité durant cette thèse. Je voudrais également remercier l'ensemble des doctorants, ingénieurs, stagiaires avec qui j'ai pu échanger et gagner en connaissance. Je remercie tout particulièrement Simon et Anton pour ces trois super années passées avec vous.
Merci aussi à toute ma famille, et particulièrement mes parents Stéphane et Nadia pour leur soutien tout au long de mes études. Enfin, merci à ma compagne Lili qui a toujours été à mes côtés même dans les moments les plus compliqués.

Résumé en français

0.1 Introduction

L'utilisation de communications non bloquantes afin de gagner en performance représente un enjeu du fait de leur complexité. Il est en effet nécessaire de progresser ces communications afin de pouvoir recouvrir leur coût par du calcul. Cependant, la progression requiert du temps CPU habituellement utilisé par l'application. Partager ces ressources entre l'application et le runtime MPI mène le plus souvent à un recouvrement faible voir inexistant, ainsi qu'a des ralentissements du coté de l'application. Enfin avoir du recouvrement nécessite une conception particulière de ces codes. Cette conception n'est le plus souvent pas présente.
L'ensemble de ces raisons à conduit les opérations non bloquantes à n'être utilisées seulement que dans de rares cas, n'étant la plupart du temps pas rentables. La principale motivation de cette thèse est de proposer des solutions pour la progression de ces communications, afin de réunir recouvrement et gestion efficace des ressources de calcul. Cette association à pour but final de permettre un gain de performances grâce a ces opérations dans un plus grand nombre de cas.

0.1.1 Contributions

Pour réaliser ces objectifs, cette thèse propose un ensemble de contributions:

- La conception d'une méthode globale pour évaluer non seulement le recouvrement obtenu, mais aussi les performances des différents mécanismes de progression de l'état de l'art. Nous proposons une étude des méthodes de mesures existantes et de leur fonctionnement. Nous proposons différentes métriques complémentaires pour analyser et comprendre les causes d'un mauvais recouvrement, et plus largement des problèmes de performances rencontrés lors de l'utilisation de mécanismes de progression.

- Nous proposons la modification d'un runtime dans l'objectif de progresser les collectives non bloquantes de manière efficace. Cette modification s'appuie sur un mécanisme de progression optimisé pour s'exécuter sur un cœur physique dédié au runtime MPI, ainsi que sur l'implémentation d'algorithmes de collectives optimisés pour être exécuté sur un cœur dédié.

- Enfin, nous proposons un modèle de performance conçu pour prédire l'efficacité de l'utilisation du cœur dédié avec différentes applications hybrides MPI + OpenMP.

Ce modèle est aussi conçu pour assister la décision d'investir dans une reconception d'une application existante pour l'utilisation d'opérations non bloquantes.

0.2 Évaluer le recouvrement des collectives non bloquantes

Avant de pouvoir développer un mécanisme performant pour la progression des communications non bloquantes, il est nécessaire de définir ce qu'est la performance d'une collective.

0.2.1 Définition et mesure de la performance d'une collective

La durée de la collective est le premier point nécessaire. Cependant, une collective concerne par définition un ensemble de processus MPI. Ainsi chaque processus impliqué dans la collective possède sa propre durée d'exécution.

Afin d'avoir une valeur globale, il faut donc agréger cet ensemble de durées. Les outils de l'état de l'art ont pour cela fait le choix d'utiliser la moyenne des durées des processus. Cette valeur de temps manque cependant les potentiels écarts de variances entre les différents processus.

Le démarrage synchronisé entre les processus est aussi nécessaire afin d'avoir un comportement stable de la collective. Sans synchronisation, le comportement de la collective risque d'être impacté par un processus retardataire. Certains outils de mesures existants reposent sur l'utilisation d'une collective MPI Barrier dont le but est d'attendre l'ensemble des processus avant de continuer l'exécution. Cependant, rien ne garantie que l'ensemble des processus sortent en simultané de cette opération.

Plusieurs benchmarks se reposent sur l'opération MPI de prise de temps MPI Wtime. Cette opération est définie par le standard et implantée par les différentes implémentations MPI existantes. Si le standard définit une précision minimale à respecter, les implémentations peuvent se baser sur différentes méthodes de prise de temps. Ainsi, la comparaison entre différentes implémentations peut être faussée.

La mesure du recouvrement est réalisée par ratio de recouvrement. Ce ratio impose l'utilisation de temps de calculs et de communication équivalent peu similaire aux cas réels d'applications. De plus ces ratios omettent un potentiel ralentissement dû à un mauvais recouvrement.

0.2.2 Définition de métriques

Pour répondre à ces problèmes de mesure, nous proposons un ensemble de métriques pour la mesure des collectives non bloquantes.

Ratio de surcoût

Pour mesurer le recouvrement dans des cas déséquilibrés en tenant compte de potentiels ralentissements, nous utilisons un ratio de surcoût dont la formule est:

$$r_{\text{overhead}} = \frac{t_{\text{measured}} - \max(t_{\text{comp_ref}}, t_{\text{comm_ref}})}{\min(t_{\text{comp_ref}}, t_{\text{comm_ref}})} \tag{1}$$

Ce ratio a pour valeur 0 si les communications et calculs sont totalement recouverts. Il prend la valeur 1 si les communications et calculs sont équivalents ou égaux à une exécution séquentielle. Enfin tout ratio supérieur à 1 exprime un ralentissement par rapport au temps séquentiel.

Ratio d'impact passif

Le runtime MPI a besoin durant l'exécution d'une application de s'exécuter sur les ressources de calculs. Cette exécution peut même se dérouler dans des phases ou il n'y a pas de communications.

$$r_{\text{MPI_impact}} = \frac{t_{\text{comp_passive_mpi}}}{t_{\text{comp_ref}}} \tag{2}$$

Ce ratio vaut 1 si les performances de calculs sont égales avec et sans runtime MPI. Un ratio supérieur à 1 témoigne cependant d'un ralentissement du calcul par le runtime.

Ratios d'impact concurrenciels

Essayer de recouvrir un montant de calcul et de communication peut avoir un impact sur leur durée propre. Cela est dû aux interactions entre les deux. Pour mesurer ce phénomène, nous proposons les métriques suivantes:

$$r_{\text{comp_slowdown}} = \frac{t_{\text{comp}}}{t_{\text{comp_ref}}} \tag{3}$$

Ce premier ratio mesure l'impact sur le calcul. Un ratio de 1 montre qu'il n'y a aucun impact des communications sur le calcul. Un ratio supérieur à 1 exprime le ralentissement du calcul par rapport à une exécution sans recouvrement.

$$r_{\text{comm}} = \frac{t_{\text{call}} + t_{\text{wait}}}{t_{\text{comm_ref}}} \tag{4}$$

Ce deuxième ratio mesure l'impact du calcul sur les communications. Un ratio de 0 montre que les communications ont été recouvertes par le calcul. Un ratio de 1 montre une exécution équivalente au séquentiel. Un ratio supérieur à 1 montre un impact des calculs sur la durée des communications.

0.2.3 Étude des mécanismes de progression

À l'aide de ce *benchmark*, nous avons pu réaliser une évaluation des performances de plusieurs mécanismes de progression de l'état de l'art. Cette étude a permis de tirer les conclusions suivantes:

- Ne pas utiliser de mécanismes de progression ne permet pas de progresser les communications en tâche de fond. Le benchmark montre une exécution séquentielle dans l'ensemble de ces cas.

- L'utilisation de threads de progression est cependant capable de réaliser cette progression pour recouvrir les communications. Cependant, les métriques complémentaires au ratio de surcoût montrent un impact trop important sur les performances pour en tirer bénéfice.

0.3 Un cœur dédié pour la progression des communications

L'évaluation des mécanismes actuels a montré des difficultés à associer un bon recouvrement et des gains de performances. La principale cause de ces problèmes est la complexité de gérer le partage des différentes ressources entre le mécanisme de progression et l'application.

0.3.1 Utilisation d'un cœur dédié pour la progression des communications

Dans le but de pallier le problème de partage des ressources, nous proposons de retirer un cœur de l'application. Ce cœur peut ainsi être utilisé par le runtime MPI pour la progression des communications.
Le principal avantage de cette solution est de cloisonner chacun des runtime sur un ensemble de ressources fixe et défini. De cette manière, les interactions de partages de ressources sont minimisées. Le runtime MPI a, de plus, en permanence la possibilité d'exécuter le travail de progression sans être interrompu.
En contrepartie, l'application est exécutée sur un nombre restreint de ressources. On s'attend donc à observer un ralentissement des calculs. Le principe de ce cœur dédié repose donc sur le pari de recouvrir plus de calcul que le ralentissement provoqué par le cœur dédié.

Nous avons implémenté ce cœur dédié au sein du gestionnaire d'entrée sortie PIOman. PIOman repose sur l'utilisation de tâches légères (ltask) pour l'exécution de handlers d'évènements. Ces tâches sont exécutées par des *workers* utilisant les ressources. Pour tirer parti d'une ressource dédiée, nous avons ainsi développé un *worker* optimisé pour être exécuté dans cette configuration.

0.3.2 Implémentation d'algorithmes de collectives non bloquantes

Afin de pouvoir exécuter le travail de progression à l'aide de notre *worker* PIOman, nous nous reposons sur la bibliothèque de communication NewMadeleine et son interface

MPI Madmpi. Cette bibliothèque repose sur un ensemble d'opérations point-à-point non bloquantes pour ses communications.

L'implémentation d'algorithmes de collectives repose donc sur l'association de ces opérations non bloquantes de manière optimisée. On peut de plus découper l'exécution d'une collective en de multiples étapes composées chacune d'un ensemble de point-à-point. Le principe de fonctionnement mis en œuvre ici est donc de se servir des événements déclenchés par la terminaison des opérations point-à-point d'une étape pour ordonnancer l'étape suivante lorsque tout les handlers ont été exécuté.

0.3.3 Étude de la progression avec cœur dédié

À l'aide du *benchmark* défini dans la section précédente, nous avons évaluer les performances du cœur dédié. Ce mécanisme a montré de bonnes capacités à recouvrir communication et calculs y compris sur des cas déséquilibrés. De plus, les métriques complémentaires montrent un impact minimal sur les performances. Ce mécanisme est donc prometteur pour gagner des performances par recouvrement.

0.4 Modèle de performance des applications pour l'utilisation de cœur dédié

L'utilisation de cœur dédié pour la progression des communications montre de bonnes performances du point de vue du benchmark. Pour valider l'efficacité du cœur dédié, il nous faut maintenant confirmer ces performances sur des cas réels d'utilisation et donc des applications HPC. Ces applications possèdent des profils très différents en termes de proportion de temps MPI, de communication bloquante et non bloquantes ou encore de collective ou point-à-point. Nous proposons donc un modèle de performance permettant à partir de ces profils de déterminer l'impact positif ou négatif du cœur dédié sur ces applications.

0.4.1 Vue globale du modèle

L'idée de ce modèle est de prendre en compte deux phénomènes distincts, la perte de performance sur le calcul causée par le cœur manquant et le potentiel gain dû au recouvrement induit par le mécanisme de progression.

Pour la partie calcul, nous avons étudié le passage à l'échelle de plusieurs applications HPC afin d'observer le coût de voler un cœur. La multiplication du nombre de cœurs sur les processeurs modernes rend ardue leur utilisation à plein potentiel: On observe dans les meilleurs cas un passage à l'échelle linéaire. De multiples applications ont cependant un gain de performance inférieur voir une perte.
Nous modélisons donc le surcoût à l'aide d'une fonction de passage à l'échelle linéaire. De cette manière, nous sommes en mesure de borner le gain potentiel des applications. Cette solution pessimiste nous garanti donc de ne pas sous estimer l'impact du cœur dédié.

Pour la deuxième partie du modèle, nous voulons estimer l'impact sur le temps MPI du cœur dédié. La totalité du temps MPI est cependant composée de diverses parties réagissant différemment au cœur dédié:

- Le temps passé dans les appels bloquants.
- Le temps passé dans les appels non bloquants.
- Le temps passé dans les appels de complétion.
- Le reste du temps MPI.

Si *a priori* les communications bloquantes ne sont pas affectées par le cœur dédié, nous avons choisi de modéliser une potentielle transformation de ces appels en appels non bloquant recouvert. En effet, la plupart des applications ne sont pas conçues pour le recouvrement. Profiter de cette solution requiert ainsi une reconception. Cette modélisation permet ainsi une estimation des gains avant un investissement dans cette reconception.

Les communications non bloquantes sont en revanche affectées par le cœur dédié. Sans cœur dédié le temps passé dans les appels d'initialisation et de complétion non bloquant inclut le temps de ces initialisations et complétions, mais aussi le temps de la progression réalisée. Avec un cœur dédié, ce travail de progression n'est plus fait dans ces appels. On s'attend donc à une réduction du temps passé dans l'ensemble de ces opérations.

Le reste du temps MPI correspond à l'initialisation du runtime, des communicateurs et des *datatypes*. Cette partie n'est pas affectée par le cœur dédié.

Nous avons donc le modèle de performance suivant avec en noir la modélisation du calcul, en rouge la partie affectée de MPI, et en bleu la partie constante de MPI.

$$
\begin{aligned}
t_{\text{dedicated}} = t_{\text{comp}} \times \frac{N_{\text{core}}}{N_{\text{core}} - 1} + \\
N_{\text{nonblocking}} \times t_{\text{minMPInonblock}} + N_{\text{test}} \times t_{\text{minMPItest}} + \\
N_{\text{wait}} \times t_{\text{minMPIwait}} + \alpha \times N_{\text{blocking}} \times (t_{\text{minMPInonblock}} + t_{\text{minMPIwait}}) \\
+ (1 - \alpha) \times N_{\text{blocking}} \times t_{\text{MPIblockingcom}} + t_{\text{MPIother}}
\end{aligned}
$$

0.4.2 Validation du modèle

Afin de valider les bonnes performances du modèle, nous avons réalisé une étude sur plusieurs applications de l'état de l'art. Nous avons pour cela réalisé deux types d'exécutions: une première basée sur une configuration hybride basique avec un *thread* OpenMP par cœur et un processus MPI par nœud. La seconde reprends la configuration mise au point pour le cœur dédié. Un cas test correspond donc à l'ensemble de ces deux exécutions lancé avec les mêmes paramètres de l'application. Le modèle prend ainsi en entrée les exécutions de la configuration par défaut, et cherche à approcher l'exécution avec cœur dédié.

Les résultats obtenus montrent que le modèle est capable de discriminer les cas gagnants et perdants. Le modèle est donc capable de définir pour quels cas utiliser les cœur dédié ou non.

Chapter 1

Introduction

Experimenting is at the base of innovation and development in numerous field of works. For the industry, running such tests can rapidly become expansive or hard to implement. Some of these experiments indeed would require to use prototypes requiring to afford raw materials and building costs. As an example, the development of a rocket need to ensure that it would respect constraints needed to operate. It should be expected to launch multiple prototypes before the real one.
Sometimes, experiments are just impossible to drive considering danger issues or non reproducible conditions. For example, trying to reproduce the entry landing of a capsule in Mars atmosphere is not possible in real conditions as conditions are not present on Earth.
To bypass these issues, the industry can rely on multiple tools and particularly on the simulation. The purpose of simulating phenomena is to computationally reproduce the characteristics of the experiments such as objects, environmental and physical conditions to fit the reality. These simulations are thus able to reduce the costs of implementation, and reproduce impossible parameters.
However, virtually running these simulations with a precision high enough to fit the reality require to take account of millions of variables. To be able to run such simulation within a reasonable amount of time, it is necessary to develop specific powerful hardware to compute these numerous variables. These big an powerful machines are call supercomputers, and the field of work created to operate them is called the "High Performance computing" field (HPC).

Nowadays supercomputers are composed of multiple machines working together called *nodes*, over a high performance network. To be able to run a global simulation on such distributed resources, it is necessary to split this global workload among the machines composing the supercomputer. Moreover, to make these machines working together, a software layer is required to communicate intermediary results between these machines. The most common software used to perform these communications is "Message Passing Interface" (MPI). This standard use messages to communicate the data needed by nodes. The cost of these communications remains however higher than memory transfer used inside each node. Thus, the communication performance is a major issue as it is usually a bottleneck in the overall performance of the system.
One solution to reduce the impact of the communication on the performance is to hide these communications with simultaneous computation. MPI propose nonblocking primitives

allowing the application developer to perform computation between the communication initiation and its termination. Such simultaneous computation on nodes and communication on the network is called *communication and computation overlap*.

1.1 Objectives of this thesis

The complexity of these communication operations brought issues when trying to use it in the purpose of getting a performance gain. They indeed require to be progressed to perform overlap using the computation resources typically dedicated on the application. This resource sharing have for main consequence of either get poorly overlapped operations or have a performance drop on the application side. Moreover, the applications are not designed to enable overlap in most of the cases.
All these factors have made these operations to not being used often, as it does not bring performance gain most of the time. The main idea of this thesis is to propose solutions to progress communications in order to conciliate overlap and an efficient management of computation resources with the final purpose of improving performances.

Contributions

In this thesis, we thus propose some elements to answer the need for effective progression of communications with:

- The design of a global method to assess the performance of state-of-the-art progression mechanisms used and designed to progress nonblocking collectives. We make a survey of existing benchmarks and how they measure overlap and communication performance. We propose complementary metrics to be used to understand the causes of bad overlap and more widely the causes of bad performances usually encountered when using nonblocking collective and progression mechanism. We give an implementation of these metrics taking into account multiple issues met when measuring collectives.

- We propose a full runtime modification to efficiently progress nonblocking collective, including a progression mechanism based on dedicating a physical core to the MPI runtime in order to run the progression engine. We also propose new collective algorithms designed to be efficiently progressed using this dedicated core.

- We finally propose a model designed to predict the efficiency of this solution on real applications with hybrid MPI + OpenMP design. This model can then be used to give a clue on the relevancy of investing a re-design of applications to use nonblocking collectives.

1.1.1 Organisation of the document

This document is split in six subsequent chapters. The chapter 2 introduce all of the prerequisites to fully understand the context and the problematic discussed in this thesis. The chapter 3 present some relevant work made around the problematic and the existing solutions already implemented.

The next three chapters focus on the contributions made during this thesis with respectively: The chapter 4 introducing the new metrics and the implementation of the benchmark. The chapter 5 presenting the implementation of the dedicated core solution and the collective algorithms in the NewMadeleine communication library. The chapter 6 give the conception of the predicting model, its validation and use cases on common HPC applications. Finally, the chapter 7 summarise the work presented in this document and open to other perspective to develop in the future.

Chapter 2

Context around high performance computing and the problematic

Contents

2.1 Simulation in of high performance computing 21

2.2 The evolution of hardware 22

2.2.1 Computer-scale developments 22

2.2.2 Advent of distributed computing 24

2.3 The evolution of software 26

2.3.1 Managing multiple models resource usage 26

2.3.2 The fork/join model 26

2.3.3 The task model 29

2.3.4 Message passing model 30

2.4 Problematic of this thesis 35

To have a good understanding of the issues addressed in this thesis, it is necessary to introduce some global ideas around the High Performance Computing field (HPC). To have faster and effective computation, the high performance computing field always relied on the evolution of technologies. The following sections will introduce the basics from the evolution of processors and supercomputers to the software stack developed to operate them.

2.1 Simulation in of high performance computing

Science and industry are fields where experimentation is one of the starting point of innovation. However, the complexity of implementing some experiments due to environmental, technical, or financial issue can hinder the progress of knowledge. Since the start of computer science, the development of computer technology have made possible to bypass those issues by modelling and simulate those complex experiments.

The seek for performance toward simulating various phenomena is at the origin of the high performance computing field. This research of performance is powered by enhancing computers on multiple sides from the physical components (hardware) to the algorithms and their implementation (software). In this first chapter we introduce both hardware and software knowledge needed for the good understanding of this thesis.

2.2 The evolution of hardware

One of the main factor of improving performance of computer is the evolution of the hardware composing it. Cutting edge machines used in high performance computing are then in constant evolution to follow the increasing need of computation power. In this section we will first focus on computer-wide improvements made, then presenting the distributed computing advent.

2.2.1 Computer-scale developments

At the scale of one computer, multiple enhancements have been done on its components to improve the performance, among which we can especially cite the processor or the memory.

Enhancement of microprocessors

At the origins of the computer, there is a need of fast and automatic computation. To answer this need, the computer relies on its processor or *central processing unit* (CPU) which is designed to execute various operations. From the base of circuit boards using transistors, the evolution made nowadays microprocessors with more and more computation capability and power efficiency.

In the pursuit of performance since the seventies, the microprocessor has become faster and smaller. This evolution of hardware has been theorised first by Gordon E. Moore in 1965 as doubling the transistors density every year. This pace was revised ten years later in 1975 as doubling every two years. Looking at the actual evolution of microprocessors transistor number, the pace follow this forecast (figure 2.1).

This multiplication of transistor was possible due to the miniaturisation of these components. The Dennard scaling state that with the reduced transistor dimension, the voltage is reduced proportionally which is at the origin of increasing the clock frequency. Until around 2000-2005, the improvements were essentially based on this increasing clock rate. During this period, the reduction of voltage started to hit a physical limit. As a consequence, since the middle of 2000's, the clock rate increasing pace stagnates. As of 2021, microprocessors used in industry do not exceed 5Ghz.

Notion of parallel computing

With the stagnation of the frequency of processors, an idea was to make multiple processors work together. Multiple methods have been developed to allow multiple simultaneous executions context to work together on. This can be defined as multiple execution units

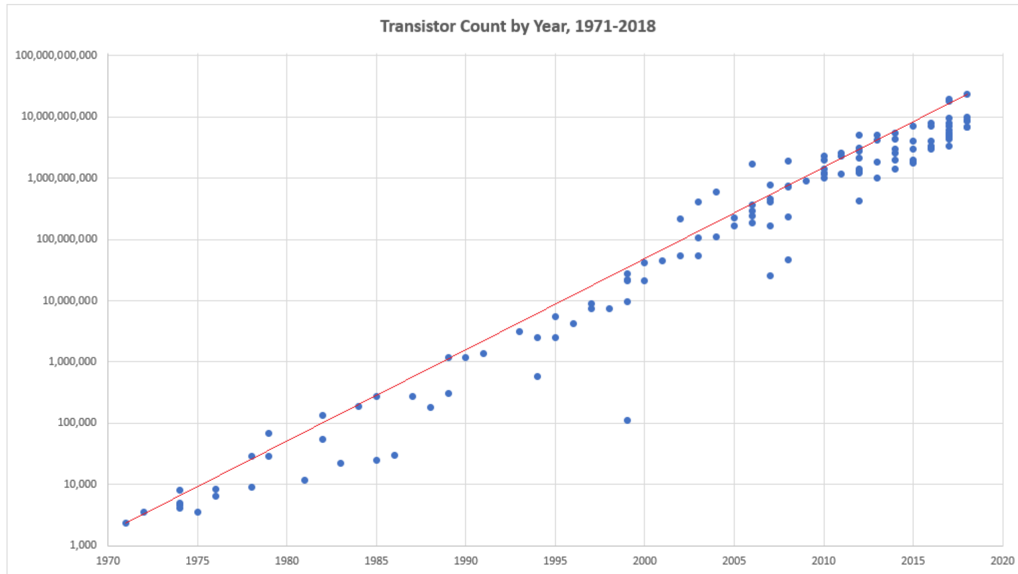


Figure 2.1

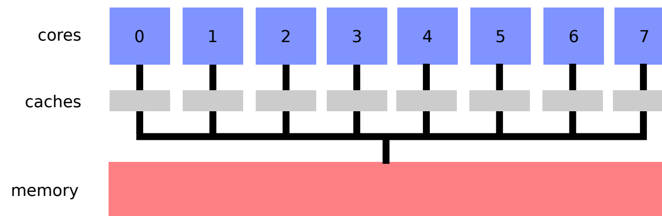


Figure 2.2: Example of an SMP architecture with 8 cores

working simultaneously on one big problem. This problem must be split between all these execution units. Usually, these units must share resources depending on the type of parallelism.

Inside a machine, multiple processor can then work on top of a memory accessed via one or multiple bus.

Parallelism with shared memory

The computer architecture is then adapted to permit the access to memory for multiple processors. The Symmetric shared memory processor (SMP) rely on multiple processors on top of one single memory(figure 2.2). This memory must be protected against concurrent writings at the same memory address. Also, the algorithm must be developed to ensure that multiple parts of the code can be executed by multiple workers.

Multi-core microprocessors

The number of instruction per cycle(IPC) represent the average number instruction done in one clock cycle. The evolution of microprocessors also reached physical limits preventing the increase of IPC. To keep increasing the performance of processor it was necessary to

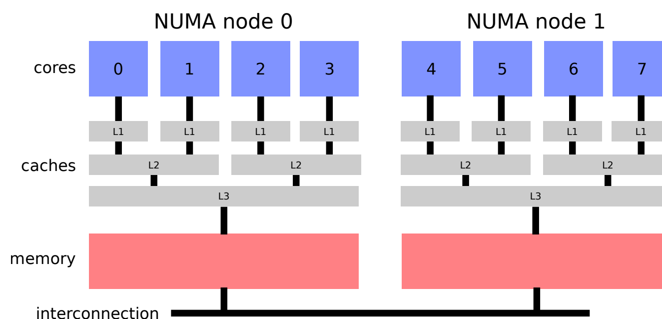


Figure 2.3: Example of an NUMA architecture with 8 cores

rely on new technologies.

In 2001, IBM released the POWER4 which was the first commercially available multi-core processor. The "core" refers to a physical execution unit that are on the same chip. Multicore processors have made it possible to maintain the evolution of the computing capacities with the multiplication of PU on chips.

As of today, modern processor feature numerous cores on one chip. The multiplication of cores increased the complexity of programming. To address these issues and benefit from multiple cores, several programming models have been created. We will introduce one of the most used in section 2.3.2.

The non-uniform memory access

The multiplication of processing units on the SMP architecture eventually bring some issues. The multiplication of execution units accessing the unique memory from one single bus causing congestion and degrading the performance. To address this issue, a new architecture was designed were each core have its own local memory (figure 2.3). The memory as a whole is still sharedi, but the non-local memory is longer to access. This type of architecture allow the user to develop software respecting constraints of locality in order to reduce the congestion as each core use his own memory in priority and access non-local memory only when needed.

2.2.2 Advent of distributed computing

In 1989 with the creation of PVM [1], the exponential growing need of computation power was at the origin of another form of parallelism. As the simulations and mathematical applications became greedier, the use of multiple processors and thus of multiple machines became necessary.

Massively parallel clusters

These sets of machine are linked using networking technologies. They can then communicate intermediate results and work together on one problem. Thus, this group of machine is considered as one computer and is called a cluster computer. Inside those cluster, each

physical machine is called a node. To take advantage of those distributed node, a global workload is split between them. Structurally, this can be seen as a unique application software designed to be separated on multiple machines. This introduced a different level of parallelism to be exploited, the distributed memory parallelism, and therefore new programming models such as MPI presented in section 2.3.4.

Parallelism with distributed memory

The distributed memory architecture is based on the use of each node memory from a global point of view. As each node can only access its own memory, it needs to communicate necessary data between them. This data transfer is performed on the network using message passing protocol.
In this way, the global cluster is able to split a global problem on each node and gather all parts to get the global result.

Evolution of network interfaces

The idea of getting high performance cluster based on network linked node instead of one specific supercomputer emerged from the mid 1990s and the Beowulf cluster architecture [2]. This evolution was completed with the first high speed dedicated networks such as Myrinet [3].

With the evolution of clusters, being able to communicate efficiently between nodes became a key challenge. As each node became faster and faster due to improvements introduces in section 2.2.1, the communication time must be the fastest possible to minimise the CPU waiting for incoming data. Otherwise, the network communication will make the CPU not operate at maximum capacity, and then slow the execution. More generally, having performances of the entire system limited by the slowest component is known as a bottleneck issue and are a major problem in HPC.

In terms of network interface performance the effectiveness will be measured with its bandwidth i.e, the amount of data transferred over time, and its data transfer time. Multiple solution are used, from high-end common hardware such as Gigagbyte Ethernet card, to specialised hardware such as Infiniband or Omnipath. This kind of hardware are designed to achieve high bandwdith and most importantly low data transfer time.

These fast networks also rely on specific technologies such as zero-copy communications. This technology make it possible from network interface to directly transfer the data from and to the application memory of each process. Without it, the data must be copied from the application to a specific network buffer to be transferred.

During the last decade, one of the most important changes in network hardware is the increasingly complex integration of components allowing the management of communications by the network interface to reduce the impact on application computation. This kind of hardware such as Atos BXI is able to fully offload communication, i.e, the communication progression work is fully done by the BXI card. The processor is then relieved from this work, and fully available for the application.

2.3 The evolution of software

With the hardware evolution and the creation of new architectures, it was necessary to bring a software layer to abstract the growing complexity of systems. The main idea behind these softwares is to free an application or simulation designer from the change of hardware with the use of an application programming interface (API) or a standard.

To comply with the different parallel architectures exposed in the last section, multiple programming model have been developed. This section present different of these programming model. The shared memory architecture is usually exploited using models such as task or fork/join model, and the distributed memory architecture rather rely on message passing model.

2.3.1 Managing multiple models resource usage

In most situations, to have an optimal use of the available hardware, HPC applications chose to mix multiple programming models to cover both shared and distributed memory. To be able to work, each programming model will require to use resources during the execution. As a consequence, each model only know its own existence on the system. Thus, by default, they all see the entire set of resource to be executed on. Without the specific intervention of the user, each model can then try to use the same resources at the same time and cause overall performance loss.

The user can manually manage the resources using built-in tools of each programming model, to restrain the resources allowed for each one. Another solution is to have a framework managing multiple programming models to have a global vision of the multiple programming model used. This is solution used by MPC [4] which can have its own MPI and OpenMP implementation. These two standards cover both share and distributed memory, and we will then introduce them in section 2.3.4 and section 2.3.2.

A need for topology aware management

HPC clusters may usually highly differ in terms of hardware constitution. The combinations of CPU, network adapters, accelerators give for each cluster its own architecture. Thus, the multiple programming models running on these systems have to get a global vision of the components constituting each compute node. This hardware constitution is referred as the machine topology and describe the number of components in nodes such as cores, NUMA nodes, sockets.
The hwloc [5] framework is conceived to gather information about HPC platforms, and describe the topology on the running machine. It can bes used via the `lstopo` command or directly in applications in C.

2.3.2 The fork/join model

To exploit multicore processors, different models of parallel execution were designed. One of the most used is the fork-join model. The fork join model relies on taking a sequential

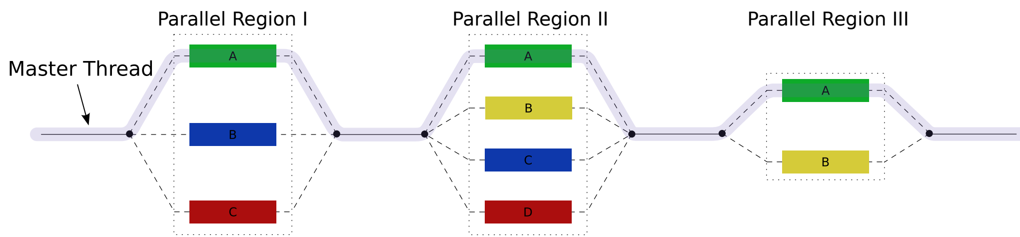


Figure 2.4: Example of a fork/join program

execution and creating multiple thread working simultaneously on a predetermined parallel region (fork). At the end of this parallel region, resume to a sequential execution (join). This cycle can then be repeated to have multiple parallel regions when needed(figure 2.4). The sequential part of the execution is executed by the designated master thread. The repeating cycles of fork/join does not require creating threads each time. The threads are only created once and are asleep on the sequential part. These threads kept available for further tasks are called a thread pool.

OpenMP is a programming model for shared-memory multiprocessing programming implementing the fork/join model. It supports C/C++ and Fortran language. The first version of OpenMP was released in 1997. Current version is 5.2 released in 2021. OpenMP is used in HPC field to exploit intra-node parallelism of multicore processors. Multiple OpenMP implementations exist such as GNU OpenMP (GOMP) [6] for gcc compiler, LLVM and icc also have their own OpenMP implementation.

OpenMP core elements

OpenMP allows users to manage parallel regions, thread creation, data sharing and thread synchronisation. It features different directives to declare how to parallelize the code.
An execution thread is a software unit with an execution context. Each thread has its own execution context but shares the memory with other threads. The figure 2.5 show an example code with creation of a parallel region. In these regions, multiple thread will execute the code in that zone. More directives help to control the threads behaviour and data management. For example the global workload can be split between different threads. One most common way to do so is to split independent iterations from a loop like in the example code (figure 2.6).

Resource sharing and concurrency

Having simultaneous execution of code sharing the same memory is yet not innocuous. As an example, letting multiple threads incrementing a single variable once with no control will eventually break the intended behaviour. This happens as basic increment operation is not atomic, i.e, multiple step are required to perform an increment: reading the value, add one to this value, and write the new value.

With no control, threads can read the same value and then write the same increment multiple times. The figure 2.7 show an example of using non-atomic incrementation operations. The operation is divided in three different atomic sub operations, read (green),

```
int main(int argc, char** argv)
{
  #pragma omp parallel
  {
    //some parallel code
    //each thread will execute
  }
}
```

Figure 2.5: creation of parallel region

```
int main(int argc, char** argv)
{
  int tab[1000];
  #pragma omp parallel for
    for (int i = 0; i < 1000; i++)
    {
      tab[i] = i*i;
    }
}
```

Figure 2.6: splitting workload in "parallel for"

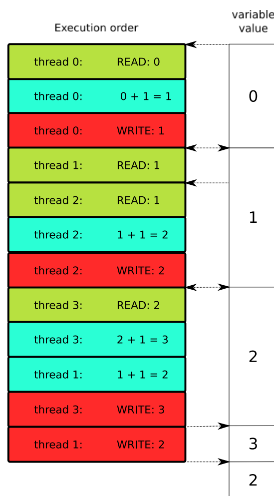


Figure 2.7: Basic increment with four threads

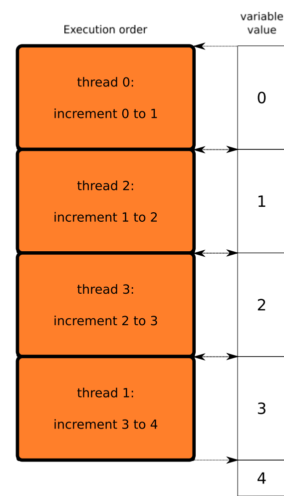


Figure 2.8: Atomic increment with four threads

arithmetic operation (light blue), and finally write (red). Without control when used on the same memory address, these operations can be mixed and give various behaviour. In this case a four threads incrementation from zero would expect a value of four. This example show a possibility where the final value is two.

To address this issue, the concurrent programming model introduces different tools. One is to lock the variable for one thread during the execution. One other is to create atomic operation which cannot be interrupted during their execution. The figure 2.8 show the same execution as before, but where all operations, read, arithmetic operation and write, respectively green, light blue and red are one indivisible operation represented as the big orange box. This operation ensure that read and write cannot be interleaved and thus ensure that the result is always the expected value of four.

Thread placement and management

OpenMP threads need to be scheduled on cores to be executed. Sometimes the number of threads is greater than the number of core, in this situation, threads need to share the resources. This resource sharing implies that thread will need to alternatively sleep to let other threads run on cores. This eventually leads to slowdown with the concerned threads. In some situations, threads that are placed on nearby cores will share memory registers, i,e, fastest but smallest temporary memory for computation, and erase other thread intermediate results to stock their ones. This is known as cache trashing and have a huge impact on overall performances.

To avoid this kind of issues, OpenMP let the user control the thread number and placement on each core. `OMP_NUM_THREADS` global variable control the total number of thread to be created by OpenMP. It is also possible to restrict the cores on which threads will be executed using `OMP_PLACES`. Finally, threads can migrate from one core to another, this can be activated or deactivated using `OMP_PROC_BIND`. The combination of all these variable allow the user to tune the OpenMP runtime and make it coexist with other programming models.

When focusing on a more general scope, OpenMP creates threads that exist in a global environment. The cores needed to execute threads have to be shared by all software running on the machine. Having multiple cores executing threads allow the application to parallelise the global workload. However, even in the application/simulation scope, threads from other programming models may exist. Having multiple threads trying to execute on the same core will cause slowdown as threads will execute alternately and lose the benefits of parallelism.

2.3.3 The task model

The task model is a different way of programming where work to do is separated in independent chunks. Each chunk of work is a task, and each task can be scheduled when it is ready to execute. A task is considered ready when all of these dependencies are fulfilled. The advantage of task is to be easily executed by any thread in the runtime. The figure 2.9 show that each application thread can create tasks, tasks are put in a task list and task engine threads can take tasks from the list. It is also possible to have multiple tasks list with a specific granularity, this can be done to take profit from the hardware topology. For example with a manycore node, it is possible to have a task list per NUMA node as, inter NUMA memory exchange are more expensive.
To control the task availability, task engine typically rely on dependencies with task requiring the end of parent tasks to be executed. It is also possible to schedule task based on specific events.

StarPU [7] is a unified platform based on task model designed for task scheduling. It allows user to develop task based application to take advantage of heterogeneous multicore architecture.

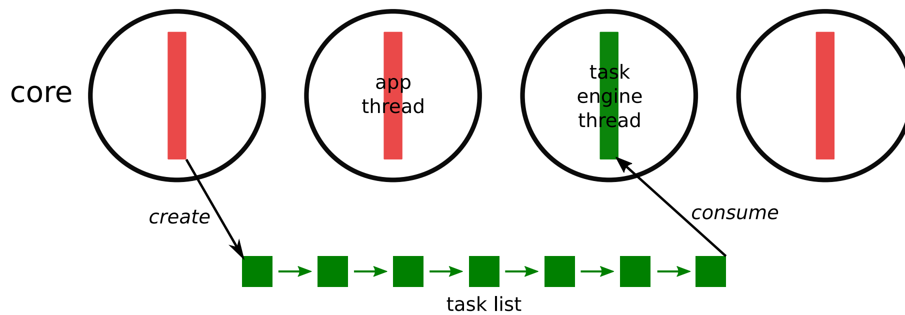


Figure 2.9: Example of task model runtime with four cores

OpenMP tasks

OpenMP proposes directives using the task model to parallelise code. Each OpenMP parallel region have a task pool. The threads executing the parallel region execute tasks from the pool when they reach a barrier. The directive `#pragma omp task` is used to create a task and add it to the task pool or execute it directly.
Tasks are used in OpenMP to have a better sharing of execution resources concerning the work to do, as each thread can execute any task when available.

2.3.4 Message passing model

With distributed memory architecture, the memory cannot be directly accessed from one node to another. The message passing model uses message send over the network to communicate data necessary for work. Program using this model sends messages to other processes to pilot their code execution. This can be used to achieve a global work as each process can be set to a specific sub-task of this work. This model is used for example in the NewMadeleine [8] library, which is a multithreaded communication library. The most used message passing programming model in HPC is Message Passing interface (MPI).

Message passing interface

Message Passing Interface (MPI) is a standard designed for communication in distributed memory. It allows user to send data between different processes in the same node or in different nodes with message passing technique. It was originally released in 1994 and last version is MPI 4.0 released in 2021. MPI is based on MPI process and communicators. MPI process are initialised using `MPI_Init` and each MPI process is part of one or several communicators with an identifier (rank) inside each communicator. The base communicator with all MPI processes registered is `MPI_COMM_WORLD`.
This standard has multiple implementations which are open source such as OpenMPI [9], mpich [10],mpc mpi [4], madmpi [11] and mvapich [12]. Some proprietary implementation also exists as intelMPI [13].

Point-to-point Blocking communication

MPI proposes many communication modes. The simplest one are blocking point-to-point where two processes want to exchange data, from a sender to a receiver. Two primitives

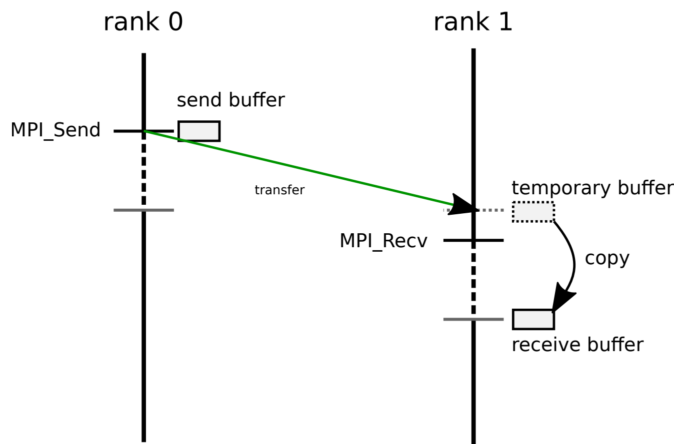


Figure 2.10: Scheme of eager protocol with the use of temporary buffer

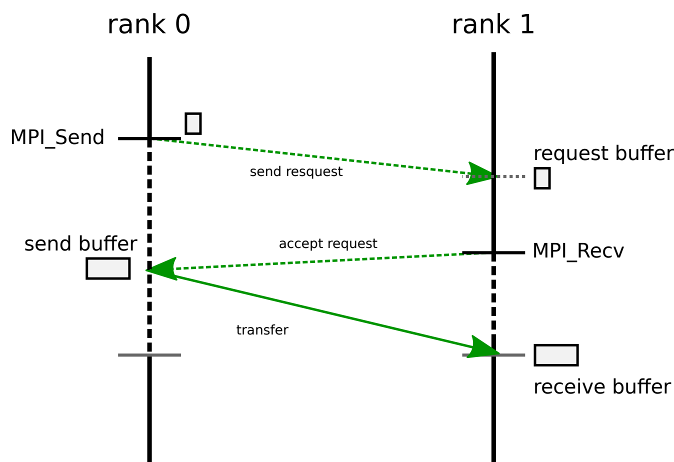


Figure 2.11: Scheme of rendezvous protocol with pending request

are needed to perform such communication: `MPI_Send` called by the sender and `MPI_Recv` by receiver. The processes implicated in a blocking communication call their respective primitive and then, the runtime initiate the communication by allowing buffers, copy data to the network interface and then wait for the communication to be done before returning and continue the application execution (figure 2.14)

Communication protocols

According to the amount data to be transferred, multiple communication protocols can be used to be more efficient. The eager protocol is particularly used when message size is small. With this protocol, the communication is done asynchronously as the sender directly transfer the data to the receiver without pre agreement. If the receive is posted before the send, the data may directly be sent in the destination buffer. This may not be the case when data require a header to know the packet content, or in function of the message matching.
In other cases, the data is transferred in a temporary buffer which is copied to the destination buffer when receive is posted(figure 2.10).

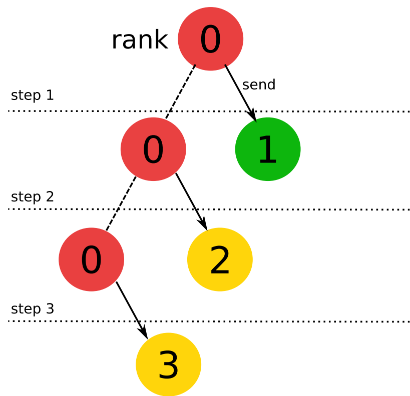


Figure 2.12: Basic `MPI_Bcast` from rank 0 with 4 MPI process

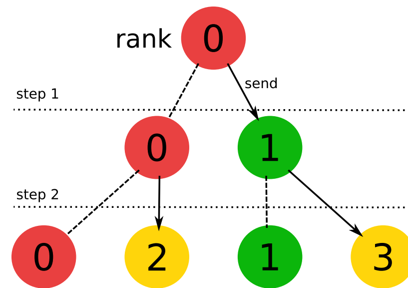


Figure 2.13: Tree-based `MPI_Bcast` from rank 0 with 4 MPI process

With large messages, this protocol quickly become ineffective as the size of the temporary buffers get bigger. The copy could also become more expensive if the memory is faster than the network. The rendezvous protocol is used for these large messages. With this protocol the message is transferred synchronously: the sender send a transfer request to the receiver which can be accepted by the receiver. The communication can then be directly done without in a zero copy way without the need of a temporary buffer (figure 2.11).

Collective operations

Along with point-to-point operations, MPI proposes communicator-wide operations. These operations are named collective operations and bring most common action needed to communicate between all nodes in a cluster. The most basic example is `MPI_Bcast` which performs a broadcast, i.e, one node sends data to all processes in the communicator.
Collective operations also necessitate more complex algorithms to be implemented than point-to-point operations. These algorithms rely on multiple point-to-point operations optimised to perform the global operation. To return to the example of the broadcast, for a n process broadcast, $n-1$ point-to-point communications will need to be done. More than that, collectives are optimised to be as fast as possible.
The broadcast typically can use a tree-based algorithm to save time by using all rank with data available at each step. The figure 2.12 show the most basic algorithm broadcast on 4 MPI processes: The root rank just send the value to every other ranks one by one. With 4 rank a total of 3 steps is required to complete the collective. The figure 2.13 perform the very same operation using a tree-based algorithm where at each step, every process having the value send it to a missing value process. The value can thus be propagated among the entire communicator in only 2 steps. It is important to note that the more processes in the communicator the greater the difference between these two algorithms will be. For n processes the first one will take $O(n)$ steps and the second one will take $O(log_2(n))$ steps.

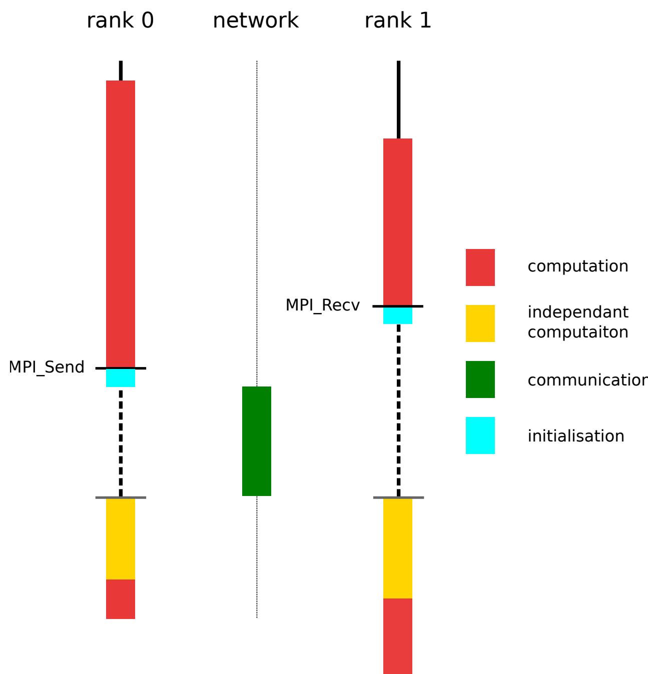


Figure 2.14: Blocking send/receive in MPI

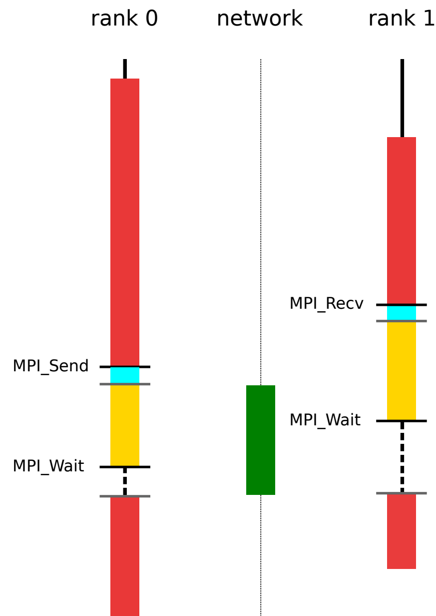


Figure 2.15: Nonblocking send/receive in MPI

Overlap of communications

One of the main limit of blocking communication is by design the waste of CPU time implied when the communication is performed. After the communication initialisation and the actual end of network transfer, blocking calls wait for the buffer to be usable again end losing precious CPU cycles.
MPI introduces nonblocking operations which will let the application continue some computations not related to the data transfer until a defined point where the data transferred will be needed. During the network transfer controlled by the network adapter, the independent computation is done by the CPU in application. This simultaneous work done by both network and CPU on computation and communication is called overlap. The overlap of communication and computation allows hiding the communication with computation to amortise its cost on the execution time.

For example, `MPI_Isend` and `MPI_Irecv` which are respectively nonblocking send and receive only initiates communication and return. The application then can continue its execution until the data transferred are needed. The user either can test if the communication is complete using `MPI_Test` or wait for it using `MPI_Wait` (figure 2.15).

MPI 3 introduced nonblocking collectives (NBC). Just like their blocking version, NBC have complex algorithms and require initiating multiple communications in the collective lifetime. These multiple initialisations are spread along the collective. For both point-to-point and collectives nonblocking operations, modification in the runtime are necessary to be executed efficiently. This is especially true for NBC due to their more complex structure.

Progression in MPI

Indeed, the nonblocking call cannot be overlapped without resources and specific mechanism designed for it, as the initialisation and data transfer to the network adapter need CPU to be performed. Without further mechanism to take care of this work, the runtime will only be able to do it when having control of the execution resources. This will likely end by eventually execute this progression task in the final `MPI_Wait`. In this situation, the communication is performed after the end of the computation and thus no overlap is done.
The progression will have a different impact for the different communication protocols used. The rendezvous protocol presented in section 2.3.4 used in a nonblocking call requires progression to be efficient as the call may return before the end of negotiation. Without a progression engine, the communication itself likely will start after the computation. The NBC with their multiple communications to be initialised during the collective add a reactivity constraint. At the end of one communication, the faster the next one is initialised, the better the performance will be.

This need of progression can be defined manually in the application. The use of multiple `MPI_Test` placed between independent computation parts will let the MPI runtime the opportunity to progress potentially pending communication initialisations. However, this requires to modify the code and introduce slowdown as some tests will have nothing to do but returning as communication is not over.
One other way is to use a background progression mechanism to test the collective state, initiate intermediate communications, and send it to the network interface. Modern, network adapter as defined in section 2.2.2 can themselves initiate the communication. With NBC, the progression work is much more important and require using the CPU to perform efficiently. The runtime then can create a specific thread regularly scheduled to do the progression work. These progression threads are effective for the progression but tends to slow the application down due to sharing processing units or cores with application threads.

Combination with OpenMP for hybrid shared distributed parallelism

To be able to benefit from a whole system using both shared and distributed memory, it is a common practise to mix multiple programming models. As an example, MPI can be associated with other programming models such as OpenMP, this type of conception is known as hybrid programming. In this case, multiple threads can call MPI primitives. MPI standard define multiple thread level support, from `MPI_THREAD_SINGLE` which deactivate hybrid support, to `MPI_THREAD_MULTIPLE` which gives no restriction. This association of MPI + OpenMP is very common as it can cover both distributed (MPI) and shared (OpenMP) memory programming. This is one of the most used combinations in HPC, and will also be used in this thesis.

2.4 Problematic of this thesis

We presented the global context needed to consider the problematic of this thesis, including the multiple programming model used in this field and the different issues concerning resource sharing and hardware interactions. Especially, the cohabitation of multiple programming models makes it difficult to manage resources for each programming model used. As an example, let's focus on hybrid MPI + OpenMP applications. One of the best case scenario is to have, for each compute node, one OpenMP thread per core, with an MPI process mainly using the network card with very few interferences on the computation. However, some issues with computation resources occupation may occur when nonblocking communications are used. As we described in section 2.3.4, nonblocking communications, especially collectives, need an efficient progress mechanism to offer performance. Such progress mechanism can be the use of a progress thread. In this case, the progress thread will run among the OpenMP threads, hence causing some oversubscribing on the computation resources. This interference between MPI-generated threads and OpenMP-generated threads may lead to a degradation of the application performance instead of the speedup expected with the use of nonblocking communications.

The problematic of this thesis is to study the progression of MPI communication using dedicated resources. The progression of communication and especially of collectives requires setting up specific mechanisms to have overlap. The cohabitation of these mechanisms and other programming models such as OpenMP also require to efficiently manage resources to avoid bad resource sharing. On the other side, the complexity of these runtime mixing both shared and distributed memory make the performance hard to measure. Also, the dedication of resources modify the application set of resources to be executed on, and the impact must be evaluated. Finally, the MPI + OpenMP applications design does not always suit the requirements to get overlap, as most of them are derived from blocking operation based codes.

In this thesis, we will answer the following issues included in this problematic:

- How to develop reliable metrics to assess the global behaviour of MPI runtime concerning background progression ?
- What are the interactions between computation managed by an OpenMP runtime and the progression mechanism ?
- The dedicated core could be effective for progression and limit the impact on computation by canceling application/ runtime resource sharing ?
- Is the re-design of an existing applicative code worth it to benefit from overlap compared to the overhead caused by a dedicated core ?

To this end, we first study the impact of progression mechanisms existing in current MPI implementations on the communication and computation of hybrid applications. We design new metrics adapted to this study and implement a set of benchmarks based on these metrics. Then, we use our benchmarks to profile the progression of nowadays MPI

implementations. This whole process is presented in chapter 4.

We then introduce a new progression mechanism using a dedicated core on top of task-based collective algorithm for the progression of NBC. In the chapter 5, we first introduce the task-based collective algorithms, before presenting the implementation of a dedicated core worker and the management of resources to ensure its exclusive on the dedicated core. We also measure its performance using the metrics introduced in the chapter 4.
Finally, in the chapter 6, we propose a model to predict the efficiency of transforming a hybrid MPI+OpenMP application based on blocking calls into an application using nonblocking calls. We detail the different part of the model from the overhead estimation based on OpenMP scaling model, to the estimation of overlap gain based on MPI primitive timings.

Chapter 3

State of the art

Contents

3.1 Benchmarking MPI 38

3.1.1 Benchmarking of blocking communications 38

3.1.2 Benchmarking of nonblocking communications 42

3.1.3 Overview and link to the progression mechanisms 43

3.2 Progression mechanism for overlap 44

3.2.1 Point-to-point communication overlap 44

3.2.2 Collective communication overlap 46

3.2.3 Overview of progression 46

3.3 Modelling hybrid applications performance 47

3.3.1 Analysing and modelling MPI performance 47

3.3.2 OpenMP scaling of applications 48

3.4 Summary of items to address in contribution 48

The measurement of collectives, and especially nonblocking collectives is not straightforward. Depending on multiple factors such as the type of progression mechanism used, if MPI processes are on the same node or not, different benchmarks can implement different measuring methods. Usually, these methods are developed to be pertinent in their scope, but may not be well suited for different usages.
In this chapter we will first introduce the existing benchmarking tools designed to measure MPI and more specifically the overlap of communication and computation.
The progression of MPI nonblocking operations is necessary to get overlap. We will then present the existing mechanisms which have been developed to answer the need for asynchronous progression.
Finally, the progression itself is not sufficient to gain time with overlapped communication and computation. The internal design of MPI + OpenMP hybrid application is often not suited to gain time from overlap. It can be helpful to model the parallel areas of a hybrid application to find out if the current algorithms in this application is compatible with efficient overlap. The last section of this chapter will introduce the work done around MPI applications, OpenMP application and hybrid applications modelling.

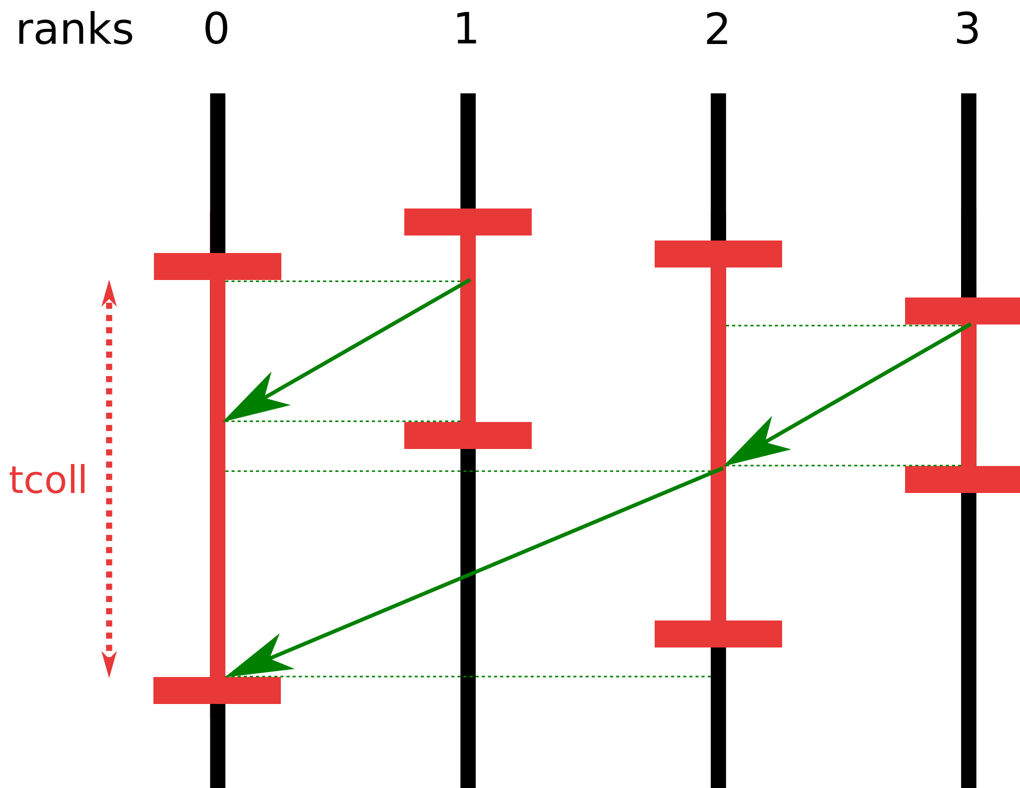


Figure 3.1: "Ping-pong" protocol scheme for point-to-point send/receive

3.1 Benchmarking MPI

To be able to improve the performance of a runtime, it is crucial to base the measure of the performance on reliable benchmarking tools. MPI benchmarks consist of a set of metrics conceived to gather the performance of different features in the specified MPI runtime. Typically, measures done with such MPI benchmarks on several MPI implementations are compared to each other, with the one having the best results being considered better than the other MPI implementations. As such, one can consider that MPI benchmark are used to *define* what are good performances. To realistically *define* the best MPI implementation, metrics must be exhaustive, must be able to detect every variation between executions and must be representative of every execution behaviour.

However, assessing the performance of MPI primitives, and especially the most complex ones, may be discussed. The definition of what is measured can have multiple interpretations, thus have different implementations in multiple benchmark suites.
In this section we will introduce the existing methods of MPI benchmarking and focus on the potential lacks these methods have.

3.1.1 Benchmarking of blocking communications

Blocking point-to-point communication are the simplest type of MPI communications, as there is only two MPI processes involved in the communication. The performance can be described as the amount of transferred data over a period of time (bandwidth) between these two MPI processes, and how fast it is transferred (data transfer time).

Several benchmarks exist for measuring blocking point-to-point MPI communications. NetPipe [14] is a network performance evaluator which have a module to measure MPI and MPI-2 operations. The Intel MPI benchmark (IMB) [15] and OSU microbenchmark [16] proposes different metrics such as network data transfer time and throughput. The measure protocol is based on a "ping-pong" between two MPI processes. One process will send a message to the other one, then the latest will send back a message. The measured data transfer time is the mean of the total duration of the two messages (figure 3.1). This "ping pong" method have the advantage of measuring the time on only one of the two processes involved in the communication. With internode communications (when the two processes are located on different nodes) this allows the measure to be done without taking care of synchronising clocks on each node.

Benchmarking of blocking collectives also relies on bandwidth and data transfer time metrics. However, the introduction of multiple ranks and thus potentially multiple nodes make it impossible to use a "ping-pong" method to have only single rank timing.

OSU microbenchmark and IMB also define methods to measure blocking collectives. Both of these benchmarks run their measurement on multiple iterations to avoid the overhead observed for first iterations. They also test a specific range of message sizes to observe the scaling of data amount on the network.
With collectives and multiple MPI processes involved, the measurement become much harder than for point-to-point. The following paragraphs will introduce some issues concerning collective measurement, and the way state-of-art answer to them.

Multi rank measurement aggregation

Opposed to point-to-point operations, collective cannot rely on a "ping pong" method. The presence of more than two ranks implies to find other methods involving all MPI processes in the collective to measure a representative performance of said collective. Moreover, the start and the end of a collective can be calculated with different methods considering each MPI process starts and ends at different time. Thus, using the average the minimum or the maximum to define the start and the end of the collective gives different valid collective times.
OSU microbenchmark and IMB both evaluate collectives by measuring the duration of a performed collective on each MPI process. However, reporting the timings for all MPI processes in the collective can be overwhelming. Thus, they both only report the average data transfer time by default. This can be flawed as tree-based algorithms may induce very different workloads on each rank. Using the average in this situation can lead to the same value for different behaviour, as the average do not take account of variance in MPI process time.
As an example, A 4 rank collective being executed in 20ms on two ranks and in 10ms on the two other will have an average execution time of 15ms. If the execution is done in 15ms on each rank, the average execution time will also be 15ms. However, the first execution is less effective. The faster rank will eventually have to wait for the slower ones and the overall execution will be closer to 20ms.

OSU microbenchmark add an option to also report the minimum and maximum data

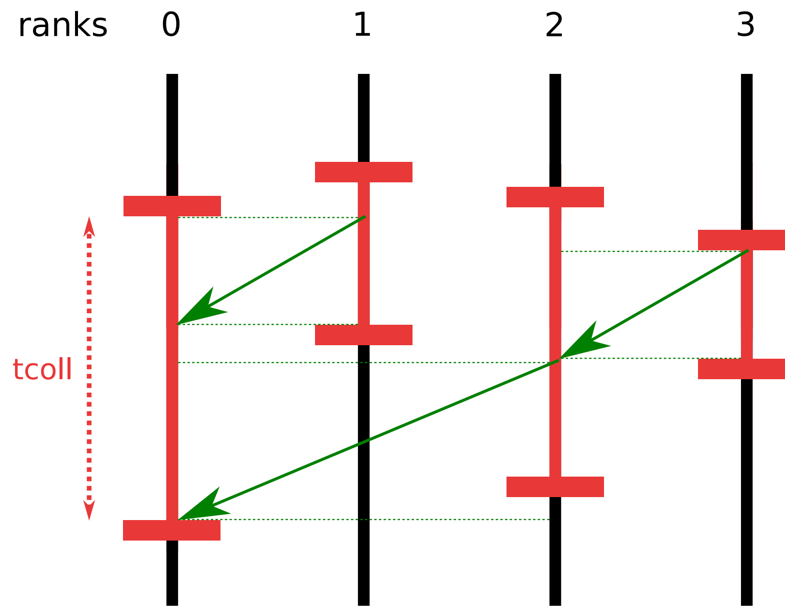


Figure 3.2: Shifted start of `MPI_Reduce` impact

transfer time. Looking at the average coupled with minimum and maximum allow to differentiate those cases. Thus, to correctly use the measures reported by such benchmark to perform a performance comparison, all of these values have to be used, and not only the average time.

Synchronised start need

MPI processes execute independently of each other. Each MPI process will experience its own set of system interruptions, concurrent computations, and other events that will affect their execution.

Thus, when performing a collective operation, they may call the initiating primitive at different times, and then either directly start working or wait, depending on their role in the collective operation. This may cause some shifting between all MPI processes.
This shift when starting the collective also cannot be just corrected by offsetting the timing at then end as the whole collective may be influenced by the initial shift. The figure 3.2 shows a `MPI_Reduce` with a shifted start. The MPI reduce wait for all value to be reduced in the root(here rank 0). The shifted start from the rank 3 will here delay the global gathering of reduced value. This chain effect, will finally cause a slowdown of the entire collective. The figure 3.3 shows the expected behaviour with a simultaneous start, which lead to a faster execution.

Moreover, the more MPI process are involved, the largest the slowdown chain effect could be. This makes the measurement of numerous MPI processes collective even harder.

To have a consistent measure, it is then necessary to have a synchronised start. The use of `MPI_Barrier` to synchronise the MPI processes does not imply that each rank will start the measured collective at the same time, as all the MPI processes may not leave the `MPI_Barrier` simultaneously.

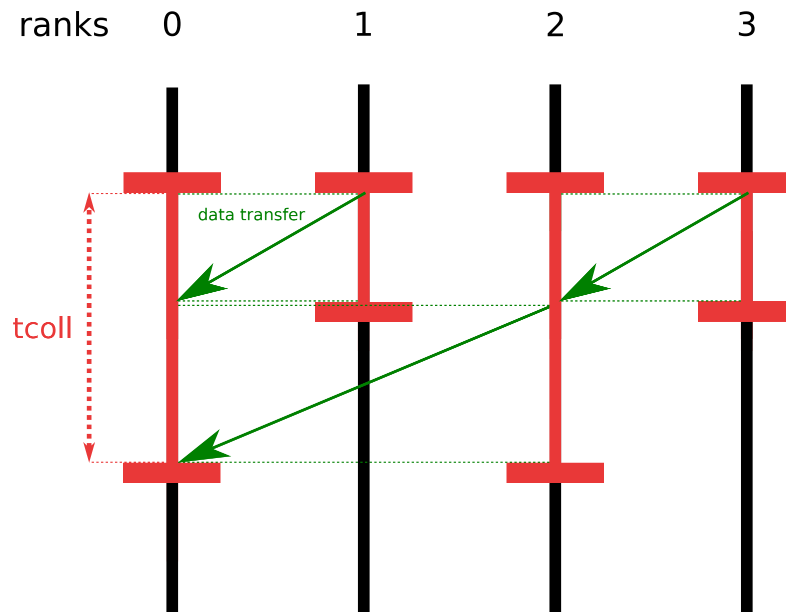


Figure 3.3: Synchronised start of `MPI_Reduce`

SkaMPI benchmark [17] uses a window-based starting protocol. The window-based starting protocol consist of all ranks agreeing on a date in the future to start simultaneously. This can lead to some issue as the date can be passed when all ranks get it. This is especially true when a lot of rank are implied in the collective.

OSU microbenchmark use a `MPI_Barrier` to synchronise the start of the collective. The standard define this operation as waiting for each MPI process before returning and continuing the execution. However, nothing is done to make all process leave this collective at the same time.

Synchronised start handling

Having multiple nodes involved in a collective implies that each start time defined on different nodes will not be comparable. Each node will rely on its processor clock to get time. However, all nodes may not have been started at the same time, and they may have different speed over time due to power consumption. Thus, time coming from different clock cannot be mixed as nothing guarantee the same origin. To avoid this kind of problem, it is possible to set a shared origin point to all clock and do a clock synchronization [18]. This clock synchronisation is achieved by gathering the start time of each node and compute the delta between them. These deltas can then be applied to correct each time from the reference.

It has also been demonstrated [19, 20] that correcting clock offsets at the start is not sufficient to keep a good synchronisation. The time of the CPU clocks may diverge between nodes after the correction, and this skewness needs to be taken into account for the end of the time measure. This divergence is caused by the physical condition of the clock preventing a perfect precision. This precision can for example vary from temperature variation in the processor. One solution to this issue is to resynchronise each

clock periodically between measures. If the measure is too long and time start to diverge before the end of the measure, another solution is to model each clock skew to correct it. However, such method will delay the end of the measure, hence invalidate the measured collective timing.

Analysing timing primitives

Timing issues may also come directly from the timing primitives. A recent paper, trying to achieve overlap through overloading, highlighted the problem of using traditional MPI timer calls (e,g,. `MPI_Wtime()`) with this configuration [21]. The `MPI_Wtime()` primitive is also implementation dependant. As each MPI runtime can have its own version with different characteristics, using these primitives prevent the comparison between multiple runtimes in addition to change benchmark characteristics.
It is then needed to use an independent timing methods. Most common and known primitives such as `gettimeofday` can work for coarse measures, but the precision is not high enough for an accurate benchmark.
A solution is to rely on high precision integrated mechanism. For example, x86 and x86-64 processors have the `RDTSC` or `RDTSCP` instructions.

3.1.2 Benchmarking of nonblocking communications

If data transfer time and bandwidth are sufficient to efficiently measure blocking communications, nonblocking ones require other metrics to assess their global behaviour. Indeed, with the possibility of computation/communication overlap, a slower but perfectly overlapable nonblocking communication can accelerate the global application. To have a complete measure, the nonblocking collective must then have a combination of overlap metric and data transfer time/bandwidth metric.

The pure data transfer time of the nonblocking communication is the total duration of the communication without any computation in between the MPI call and the corresponding wait. It can be measured with the same method as for blocking communications, with the exception of adding an `MPI_Wait` just after the nonblocking call. This is a mandatory reference time as it can be used as comparison to evaluate a potential slowdown due to computation/communication interaction. It can also be used to measure the slowdown against the equivalent blocking call.

On the contrary, the measurement of overlap introduces new methods and parameters. The main difference being the necessary introduction of computation to have overlap. The benchmarks cited above propose their measurement method for overlap and nonblocking collectives.

Analysing benchmark structure and metric relevance

Some benchmarks implement metrics working on a very specific case. OSU microbenchmark uses an overlap ratio designed as the time spent in nonblocking calls with simul-

taneous computation, relative to the pure communication time (i.e, the measure of the collective data transfer time alone). This metric is designed to measure the overlap of equal computation and communication amount, and then cannot evaluate the impact of unbalanced computation/communication on overlap. This metric is also based on the pure communication time based only on the mean between all MPI processes. As stated in section 3.1.1, this can lead to missing a potential variance difference as it do not include min and max in the ratio. OSU microbenchmark also doesn't compute a fixed amount of work but perform while communication is not over. This ensures that computation and communication have a similar execution time but miss the potential slowdowns due to communication and computation interactions.

OSU microbenchmark is structurally designed to measure the active progression of a runtime and perform a defined number of `MPI_Test` during the measure. The number of `MPI_Test` can be set with the "-t" option but cannot be zero. This is suitable for measuring manual progression of nonblocking collective. However, when focusing on background progression, the presence of `MPI_Test` will alter the measure and hinder the background progression mechanism.

The Intel MPI Benchmark (IMB) [15] use a combination of an overlap ratio and the *pure* communication duration to evaluate MPI nonblocking collective performance. OSU microbenchmarks [16] propose a similar approach with the addition of possible GPU computation.
NBCBench [22] uses a similar overlap measurement, but it focuses on the measurement of one issued collective at a time, instead of pipe-lining N collectives. It also divides the communication time in two parts: an overlappable time and a non-overlappable time, which allows measuring the overlap efficiency only on the overlappable time. The benchmark was later updated to also check the CPU overhead [23], based on lessons learned from the SkaMPI benchmarks [17] which does not handle nonblocking collectives.

MadMPI benchmark [24] relies on an *overhead ratio* to measure the overlap of point-to-point communications. This overhead ratio is a more flexible metric than the overlap ratio used in other benchmarks, as it is usable even when communication and computation do not take the same time, and it highlights cases where the overlap slows down computations or communications. However, it requires to be adapted to work for collective timing.

3.1.3 Overview and link to the progression mechanisms

In this section, we presented the most used MPI benchmarks. The state-of-the-art benchmarks implementing methods to measure the nonblocking collectives have shown weaknesses to highlight the real behaviour of nonblocking collectives. Especially the calculation of metrics to evaluate the overlap capacity remains incomplete. We highlighted cases where existing metrics were not sufficient to fully assess the performance:

- Collectives are designed to be executed on numerous MPI processes, gathering the performance of every part of this collective is sometimes incomplete. In particular,

the calculation of metrics conceived to evaluate specific parts of MPI such as overlap ratio are only based on average time.

- The parallel execution of collective may cause a shifted start between all MPI processes involved. This shift of start time influence the course of the collective algorithm and impact the performance. To get rid of this, it is necessary to synchronise the start on each MPI process. Multiple state-of-the-art benchmarks use `MPI_Barrier` which do not synchronise the processes.

- To measure the local time, methods must take care of internodes measurements as each node rely on a local clock to get time. To efficiently synchronise the start of the collective, it is necessary to set a global time by using clock synchronisation system. However, state-of-the-art benchmarks which relies on `MPI_Barrier` are based on locally calculated durations.

- Finally, the timing primitives used have a great impact on the performance measurement. Multiple benchmarks use `MPI_Wtime` to get time, which is a MPI implementation dependant tool. Thus, evaluating MPI implementation using their own tools is not reliable.

The section 2.3.4 introduced the necessity to add specific mechanism to progress the nonblocking communications in the purpose of getting overlap.

3.2 Progression mechanism for overlap

In this section we present the different mechanism that have been developed to address the need for progression. We discuss how efficient they are to progress the communication and contrast it with their performance impact on the global applications.

The need of progression for the different nonblocking operations is not necessarily equivalent. We will first introduce mechanisms done for point-to-point progression before introducing more complex ones. For both purposes, the solutions can be classified in two categories: they can directly rely on specialised hardware or develop specific software to perform the progression work.

3.2.1 Point-to-point communication overlap

Nonblocking communications have been available in MPI for point-to-point communication since the first version of the standard [25] for more than two decades. Even the progression of point-to-point communication is not straightforward. Different solutions have been released both on hardware and software side.

Hardware based progress of point-to-point

Hardware solutions relies on physical components developed to relieve general hardware such as CPU. Specific integrated microcontrollers are able to perform basic operations on network adapter. This is used to progress communication without the need of the CPU. Some network adapters such as Infiniband have been specifically designed to copy send

buffer to receive buffer with direct access memory (DMA). Some work have been done to write directly to the receiver memory using the "Remote Direct memory access" (RDMA) technology [26],[27].
All these hardware technologies have the advantage of not using the CPU, hence all the CPU capability can focus on the application during the communication progress. This allows the application to have efficient overlap. However, these hardware solutions require having the specific network adapter to use these technologies. Moreover, to follow the enhancement of these hardware capacities, it is necessary to change this network adapter to the newest version. Thus, this solution is hardly expandable to collective operations as microcontrollers are not able to manage collective messages.

Software based progress of point-to-point

To have a more generic and easily applicable solution, it is necessary to use software. These solutions can be applied without the necessity of changing supercomputer hardware. The most common mechanism developed for communication and computation overlap are progress threads. T. Hoefler and A. Lumsdaine [28] showed that creating a thread in MPI runtime make the communication progress without requiring `MPI_Test` in the application. The use of a progress thread have a counterpart as it require having access to computation resources to be executed. These resources must then be shared with the application. The overall performance can then be degraded [24].

It is also possible to build multithreaded MPI runtime [29]. This solution also schedule threads in an opportunistic way for MPI + OpenMP hybrid application on manycore architecture. The MPC framework [4] also rely on multithreaded MPI runtime by implementing user threads for multiple programming models.

Tasklets can be used to manage the point-to-point progression. PIOMan [30] is an I/O manager used to opportunistically schedule progression tasks on available cores. This allows a better cohabitation with the application threads as the progress engine can poll every task. The context needed for the progression is included in these tasks, so the engine does not have to migrate to get it. This limitation of migration reduces the bad interactions with application threads. However, The use of tasklets require specific algorithms to perform operations, and they are currently developed only for point-to-point. New implementations for collectives must then be designed to be efficiently progressed using tasklets.

Work have been done to use hyperthreading to progress communication [31],[32]. The hyperthreading use the unused computation facilities of a core to execute multiple execution context on one core at the same time. This is particularly efficient with different tasks. The thread use for progress can then be executed with less impact on application thread. However, if the same logical units are used, the performance will decrease as both application and progression mechanism will compete for resources.

3.2.2 Collective communication overlap

The advent of nonblocking collectives [33] has pushed asynchronous progression to its limit. The nonblocking collective need specific progression mechanisms which tends to have a greater impact on applications, as stated in section 2.3.4.

Hardware based progress of collectives

Hardware solution for collectives are less effective than for point-to-point. As the collectives represent much more complex algorithms, the integrated microcontrollers need to be enhanced to scale for the efficient progression of these primitives. Some solutions have been developed: Derradji et Al. proposed the BXI [9] network adapter designed to allow a full offload of communication primitives and especially nonblocking collectives on hardware. The main difference with other network adapter is to not only offload the communication work required by the collectives, but to run the entire algorithm on the card. BXI, thus support tree-based, pipeline and ring algorithms. However, this is limited to systems including this specific hardware.

Software based progress of collectives

The libNBC [22] is a portable implementation of nonblocking collective. It works using a progress thread with specific fine tuning to have a better management of spread progression along the communication duration. Progress threads are efficient for collective progression but tends to slow the application down due to threads sharing resources.

To ensure actual background progression, it has been proposed to use multi-threading [32], using a dedicated core [28], using special tasks or threads in the runtime [34, 35, 36] or in the application [37], with some optimizations such as running the thread only when a nonblocking communication operation is in progress [38].All these solutions relies on threads to progress, the possible interactions with application thread must then be handled to remains efficient.

Threading support is usually not a well-supported feature of MPI implementations regarding performances [39]. To allow for an efficient mixing of MPI with a threading models, two approaches were studied. The first approach aims to mitigate the full OS process implementation of usual MPI libraries. Several MPI implementations relying on threading or tasking were developed with the purpose of improving intranode communications [40, 41] or leveraging better runtime stacking support [42, 43]. The second approach aims to improve the support of multi-threaded communications in standard process-based implementation [44, 45]. It leads to the adoption of *Partitioned communications* in the latest MPI standard [46].

3.2.3 Overview of progression

The overlap of communication requires progression to be efficient. The collective communications are especially complex to overlap as they require more computation power more often than point-to-point. In this section, we showed the different state-of-the-art

mechanisms implemented to answer this problem. Most of these mechanisms still have remaining issues:

- Some mechanism such as progress threads are able to progress collective with good reactivity, and thus are able to get overlap. However, they have issues with sharing the available resources of the system. Since the application compete for the same resources, progress threads tends to globally slow the application down.
- Opportunistically-scheduled tasklets have also been used as progression mechanisms. They have demonstrated a good efficiency for overlap but currently do not implement collective algorithms.
- Hardware-based progression is also an existing solution. They are able to fully offload the collective algorithm to allow the CPU to fully focus on the application. However, these solutions needs to rely on specific hardware and are expansive to update, as it require to directly update the hardware.

3.3 Modelling hybrid applications performance

Most of the application using NBC today are modified blocking communication based application. The conception of applications using nonblocking communication require thinking about the independent computation, i.e, the computation not linked to the performing communications. On the contrary, applications usually have computation parts followed by a communication part based on the computation results. To have an efficient progress, application thus need deep redesign.
Specific work has been done in some applications [37, 47, 48] to utilize nonblocking communications so as to get overlap. However, the deep redesign of application is not simple nor fast. These applications often rely on old compute kernel code which need to be modified in order to discloses computation part available for overlap. Indeed, one the main factor of profitability when transforming to nonblocking operations is the amount of computation which will be available to overlap.
Understanding the characteristics of applications is then a good way to help to take the decision of transforming a code or not. Modelling applications gives the opportunity to extract these characteristics from executions. As applications may mix programming models, we will introduce in this section modelling work on both OpenMP and MPI runtime to focus on hybrid applications.

3.3.1 Analysing and modelling MPI performance

Modelling MPI may be executed using numerous approaches. Different models have been created using a general approach of modelling[26],[49],[50] which are not specifically designed to simulate overlap. These models focus on the global functioning of MPI application, but the counterpart is that they tend to be very complex and not designed to assess precise behaviour.

An alternative solution is to model the specific part of the runtime wanted to be improved. These models are simpler to implement and use and are by nature focused on

the interesting point. Previous work exists to estimate the gain obtained from overlapping communications and computation. They either estimate [51] the potential overlap from an algorithmic point of view, or focus [52] on networks with offloading capabilities and non-threaded applications. However, These models stay on MPI modelling only and cannot simulate the specific interactions with OpenMP runtime such as slowdown. The section 2.3.1 introduced that multiple programming models have an effect on each other. As a consequence, missing these interactions in modelling will make it difficult to accurately simulate the exact behaviour of the application.

3.3.2 OpenMP scaling of applications

Modelling OpenMP application try to predict the evolution of performance with a given amount of work and compute resources. The evolution of those parameters are important to handle as this is the key to predict the OpenMP scaling. Work has already been published [53],[54],[55],[56],[57] about the scalability of OpenMP applications. These models propose very complete and complex simulation of OpenMP. However, to simulate a specific behaviour, simpler models focused on that part would be enough.

3.4 Summary of items to address in contribution

Numerous models exist concerning MPI and OpenMP runtimes. We showed that these models are great to simulate a global runtime. The simulation of a whole system including multiple of these runtimes remains difficult. The chapter 6 proposes to simulate the precise part of hybrid application we worked on. This would be achieved by combining simpler models of theses runtimes. The state-of-the-art showed many possibilities on the way of getting overlap for MPI collectives operations. Theses solution either have a good overlap capability but tends to have a non-negligible impact on the global system such as progress threads, or are not sufficiently evolved to efficiently progress complex collectives such as hardware progression. The use of taskslets for progression showed a great potential for progression, but do not implement collectives. In the chapter 5, we will introduce new collective implementation using tasklets, and introduce a method based on dedicating a core to execute these tasklets. Benchmarking MPI, and especially collectives still have some issues presented in section 3.1.3. In the following chapter 4, we introduce a new benchmarking method trying to address these issues. We propose new metrics to assess the performance and the overall behaviour of collectives. We then propose an implementation for this benchmark. Finally, we give a survey of most used MPI implementation and especially ones using the progression mechanism presented in section 3.2.

Chapter 4

Evaluating the overlap of nonblocking collectives

Contents

4.1 Metrics for Overlap and its Impact on Performance 50
4.1.1 Measuring Overlap 50
4.1.2 Interference between Computation and Communication 52
4.1.3 Using these Metrics to Diagnose Bad Overlap 54
4.2 Our methodology for assessing computation/communication overlap of NBC 56
4.2.1 Multiple MPI processes Time Variation 56
4.2.2 Clock Synchronisation and Barriers 56
4.2.3 Fixed Computation and Communication Time 57
4.3 Presentation of the BenchNBC Benchmark 57
4.3.1 Benchmark Implementation 58
4.3.2 Metrics Display 59
4.3.3 Experimental setups 59
4.4 Benchmark cases study 60
4.4.1 Case study: OpenMPI/InfiniBand 60
4.4.2 Case study: MPICH with `MPICH_ASYNC_PROGRESS=1` 63
4.4.3 Comparison with state-of-the-art 65
4.5 Survey of Overlap Benchmark Results with Various MPI Implementations and Networks 68
4.5.1 Results with basic configuration 68
4.5.2 In depth interpretation of these results using complementary metrics 72
4.5.3 Results with asynchronous progression enabled 72
4.5.4 Results with dedicated core for progression 75
4.5.5 Results with hardware-based progression 76

The problematic of this thesis is to study the progression of MPI communication using dedicated resources. Evaluating the performances of asynchronous progression implies to study the behaviour of applications using nonblocking communications, through a set of metrics and measurement methods. As described in section 3.1, benchmarks and metrics to evaluate such communications already exist. Unfortunately, they either have shortcomings, or are not suited for collective operations. Moreover, if they give a general overview of the behaviour, they do not allow to understand why the observed behaviour is happening, especially in the case of bad performances. In this chapter we propose a method to not only measure the overlap of computation and communication, but also their impact on the global system. The main idea of this method is then to be used not only to measure the performance of progression mechanisms, but to give the causes of bad performances.
We will first introduce a new set of metrics, not only designed for the performance measurement of collective but also to identify the causes of bad performances. We will then introduce our implementation of our benchmarks using the new metrics. We will take into account the benchmarking problems showed in section 3.1.3 such as the management of multiple ranks value, the synchronisation of clocks for internodes operations and the simultaneous start. To show the pertinence of these metrics, we will then showcase situations measured with both our benchmarks and common state of the art benchmark to analyse the information extracted from each one. Finally, we will present a survey of various MPI implementations and progression mechanisms, used on multiple networks and hardware.

4.1 Metrics for Overlap and its Impact on Performance

In this section, we define our metrics to assess communication / computation overlap, and to measure the impact of overlap on both communication and computation performance.

4.1.1 Measuring Overlap

To measure overlap, we use the *overhead ratio* metric already defined [24] for point-to-point communication. This metric shows the overhead of the actual performance when overlapping, compared to the ideal case with perfect overlap. This ratio is actually not specific to point-to-point communication and can be used for nonblocking collectives too. It is defined as follows and as depicted in Figure 4.1. Consider $t_{\text{comp_ref}}$ the reference time of computation and $t_{\text{comm_ref}}$ the reference time of communication, without overlap. Then, in case of overlap, the ideal total time assuming perfect overlap is:

$$t_{\text{ideal}} = \max(t_{\text{comp_ref}}, t_{\text{comm_ref}})$$

The *overhead* of the actual measured performance is then:

$$\begin{aligned}\Delta_{\text{measured}} &= t_{\text{measured}} - t_{\text{ideal}} \\ &= t_{\text{measured}} - \max(t_{\text{comp_ref}}, t_{\text{comm_ref}})\end{aligned}$$

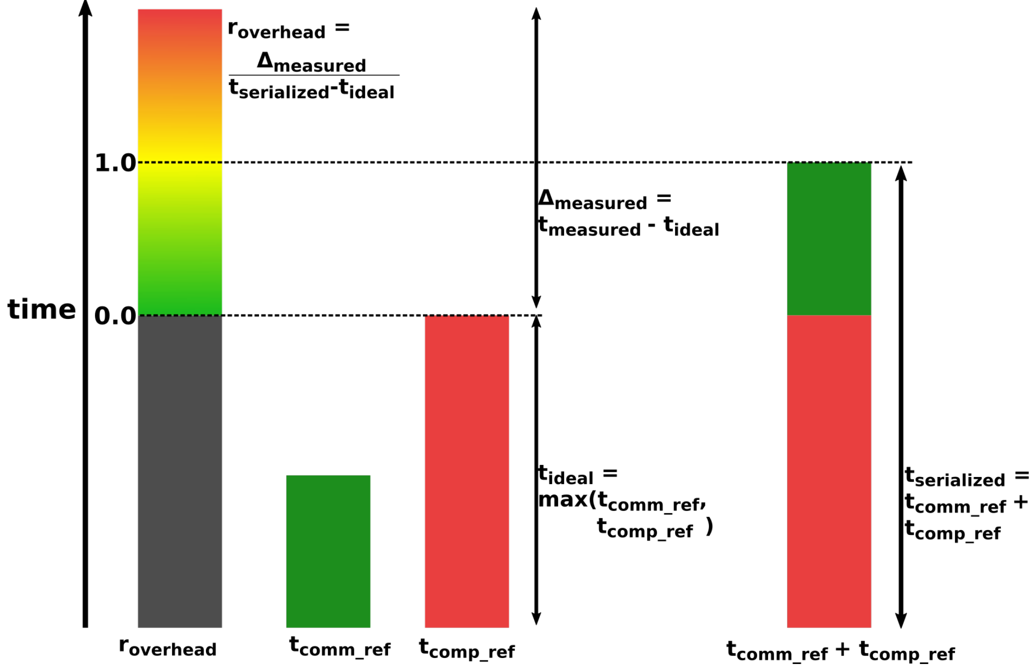


Figure 4.1: Overhead ratio metrics details

This value is an absolute time, hard to interpret since it depends on network speed and computation time. We *normalize* it relative to the serialized case, *i.e.* the case where computation and communication are run in sequence without overlap. The overhead for the serialized time is defined as:

$$\begin{aligned}\Delta_{\text{serialized}} &= t_{\text{serialized}} - t_{\text{ideal}} \\ &= t_{\text{comm_ref}} + t_{\text{comp_ref}} - \max(t_{\text{comp_ref}}, t_{\text{comm_ref}}) \\ &= \min(t_{\text{comp_ref}}, t_{\text{comm_ref}})\end{aligned}$$

We thus define the *overhead ratio* as:

$$r_{\text{overhead}} = \frac{\Delta_{\text{measured}}}{\Delta_{\text{serialized}}}$$

and thus we get:

$$r_{\text{overhead}} = \frac{t_{\text{measured}} - \max(t_{\text{comp_ref}}, t_{\text{comm_ref}})}{\min(t_{\text{comp_ref}}, t_{\text{comm_ref}})} \tag{4.1}$$

In case no overlap happens and computation and communication have been done sequentially, we have $t_{\text{measured}} = t_{\text{comp_ref}} + t_{\text{comm_ref}}$ and thus $r_{\text{overhead}} = 1$. In the case of perfect overlap, we have $t_{\text{measured}} = \max(t_{\text{comp_ref}}, t_{\text{comm_ref}})$, and thus $r_{\text{overhead}} = 0$. A ratio greater than 1 means that t_{measured} is slower than sequential execution, which may happen in case of interference between computation and communication. Finally, in some cases, we get a negative ratio, which happens if the measured time is faster than expected for perfect overlap; it is mostly observed when there are imprecisions in the measure of reference time for computation and/or communication.

4.1.2 Interference between Computation and Communication

Low performance when overlapping computation and communication is not always a sign of bad communication progression in the background. It may be caused by performance degradation of either computation or communication (or both!) because of contention or interferences. It may be especially the case when the mechanisms used for communication progression steal CPU cycles to computation.

While all cases of bad overlap performance will be captured by the *overhead ratio* defined in the previous section, this ratio cannot show whether the degradation is caused by lack of communication progression or by computation or communication slowdown. Thus, we propose additional metric that will help diagnose the cause of low overlap performance.

Impact of Inactive MPI Runtime on Computation

First, the MPI runtime system itself may have an impact on computation even without any communication, by the sole effect of the network polling mechanisms running in the background. It may be especially the case with MPI libraries that rely on an asynchronous progress thread for communication progression. Such a thread will use CPU time, which may slow down computation or cause load imbalance.

To quantify the slowdown of the application caused by background MPI execution, we propose a metric called the *MPI impact ratio*. We define it as the ratio between the computation time with and without the MPI library loaded and initialized.

To measure it, we first measure the reference computation time without MPI $t_{\text{comp_ref}}$. Then, we run the exact same computation code with the addition of enclosing `MPI_Init` and `MPI_Finalize`, and compiled and linked with the `mpicc` wrapper provided by the tested MPI implementation. We then get $t_{\text{comp_passive_mpi}}$, the computation time with the MPI runtime. The time is computed by running the largest square matrix multiplication in a given time on one core. Each core determines its own time by running one OpenMP thread. We define the *MPI impact ratio* as follows:

$$r_{\text{MPI_impact}} = \frac{t_{\text{comp_passive_mpi}}}{t_{\text{comp_ref}}} \tag{4.2}$$

If the presence of the MPI runtime causes no performance penalty, the ratio is 1. Otherwise, a ratio higher than 1 indicates that the active MPI runtime impacts the computation.

Impact of Active Progression and Computation Concurrency

Actual background progression of communication is likely to have more impact on computation than only polling. Running nonblocking collectives operations involve more than data transfer that can be offloaded to the Network Interface Card (NIC): it executes the collective algorithm. If the NIC does not directly support the offload of collective communications [58], the algorithm will take CPU time to execute, and it must be periodically scheduled to trigger all the transfers at the right time. Thus, it is expected to actually consume more CPU time than progression of point-to-point nonblocking communication.

To assess this phenomenon, we propose a metric to measure the slowdown caused on computation by the progression of communication in the background. Not all applications are expected to suffer from the same impact, we can nonetheless evaluate the runtime tendency to disturb the application, and conversely the communication disturbed by computation.

We suppose we have a fixed amount of computation between a nonblocking MPI initialisation call and the related `MPI_Wait`, and measure the time spent in each part of the execution, as depicted in Figure 4.2:

- t_{call} the time taken by the nonblocking MPI call itself,
- t_{comp} the time of the compute function with communication running in the background,
- t_{wait} the time spent in `MPI_Wait` after computation. It must be noted that if communication did not progress in the background at the same time as computation, this time may be significantly high.

Figure 4.2: Definition of times with overlap

To evaluate the slowdown of computation and communication caused by overlap, we compare these times with reference time of computation and communication without overlap. We use the same reference values as defined in section 4.1.1, namely $t_{\text{comm_ref}}$ the collective communication time without overlapping computation, and $t_{\text{comp_ref}}$ the computation time without overlapping communication, as depicted in Figure 4.3.

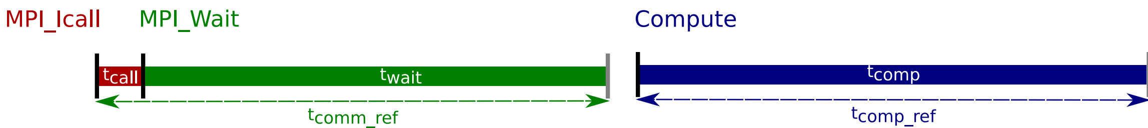


Figure 4.3: Definition of reference times

To measure the *slowdown* of computation in the presence of overlapping communication, we define the *computation slowdown ratio* as:

$$r_{\text{comp_slowdown}} = \frac{t_{\text{comp}}}{t_{\text{comp_ref}}} \tag{4.3}$$

This ratio measures the time taken by the computation to complete while the communication is running. It does not tell whether the computation is actually slower (because of contention, etc.) or whether CPU cycles have been stolen by the MPI library to make communication progress hence delaying the end of the computation. A slowdown in the computation is not necessarily an overwhelming obstacle since it may be counterbalanced by the time gained thanks to an efficient overlap, *i.e.* less time spent in the `MPI_Wait` function to finish the communication.

We couple this computation slowdown ratio with another metric used to evaluate the time spent in communication primitives compared to the reference situation. We use the exact same principle as for computation slowdown: we compare the time spent in MPI primitives $t_{call} + t_{wait}$ in the overlap case, with the communication time without overlap t_{comm_ref}.

$$r_{comm} = \frac{t_{call} + t_{wait}}{t_{comm_ref}} \tag{4.4}$$

It is expected that, in the case of overlap, most of the communication is done in the background at the same time as computation. Thus, the time taken by MPI primitives should be shorter. Since this ratio measures the time spent in the MPI primitives, in case of good overlap, r_{comm} is expected to be significantly lower than 1 thanks to a much lower t_{wait}. A ratio greater than 1 is always the sign that communication is slower than without overlap.

Finally, a ratio around 1 can have two explanations. The first reason to have such ratio is that no communication overlap happened, hence the time for the communication t_{comm_ref} is directly split between the initiation call time t_{call} and the completion call time t_{wait}. It must also be noted that a ratio around 1 may reveal that communication is slower than without overlap, caused by interference with computation. In this case, the extra time taken by the communication is hidden in the computation time due to actual overlap, hence it is not included in $t_{call} + t_{wait}$.

4.1.3 Using these Metrics to Diagnose Bad Overlap

In this section, we will show how the metrics we just defined may be used to diagnose cases of bad computation/communication overlap. Bad overlap is defined by an overhead ratio, as given by equation 4.1, greater than 0. With good overlap, we expect $r_{comp_slowdown} \approx 1$ and $r_{comm} \approx 0$.

No progression

The most common case of bad overlap happens when the MPI library does not have any progression mechanism to make communication actually progress in the background. The same behaviour may happen with an MPI library that features a progression thread for communications, but because of thread placement, the progression thread was not scheduled at the right time and thus did not have the opportunity to make communication progress.

Without any communication progression running alongside the computation function, the whole communication is ultimately done in the `MPI_Wait` call, which results in an execution as depicted in Figure 4.4. In this case, $r_{comm} \approx 1$ as defined in equation 4.4, which is roughly the same situation as in the *serialized* case, despite the use of nonblocking communications. In this scenario, nothing disturbs computation and thus $r_{comp_slowdown} \approx 1$.

In some cases, even without progression mechanisms, some overlap happens thanks to DMA: the first step of the collective algorithm is started in the MPI call and is executed by the NIC in the background. However, the following steps in the algorithm need the CPU

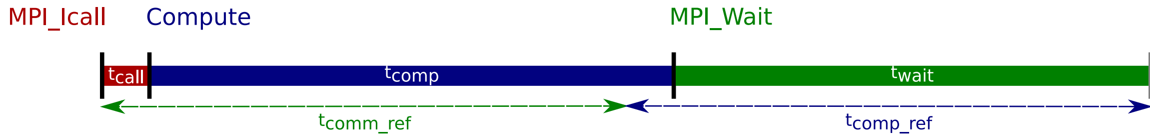


Figure 4.4: Protocol execution with no overlap

to be performed, and they will be started only once the communication is done, hence once the CPU is free, if there is no progression. In this case, r_{comm} is a little bit lower than 1, but converges to 1 for large number of nodes, while overlap becomes negligible.

Computation slowdown

Another cause of inefficient overlap is when the progression mechanism actually does overlap, but hinders the computation to a point where the overall time is higher than serialized computation and communication. The main clue to detect this problem is looking up the impact on the computation overhead: since the progression takes CPU time, it slows down the computation, which results in $r_{\text{comp_slowdown}} > 1$.

Since communication progresses alongside the computation, in this case communication is already over when we reach `MPI_Wait` and thus we have $r_{\text{comm}} \approx 0$. This case is depicted in Figure 4.5.

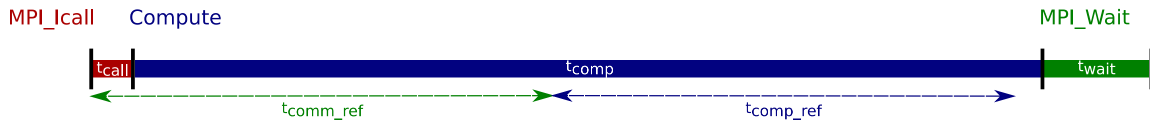


Figure 4.5: Protocol execution with communication significantly hindering computation

The bottom line is: in the presence of communication progression mechanisms, computation is slower, and thus overlap may not be worth it.

Contention without overlap

The last possible case is both $r_{\text{comp_slowdown}} > 1$ and $r_{\text{comm}} > 1$. A possible scenario for this case is communication was serialized, without any overlap, but at the same time computation was slowed down. MPI process imbalance with an aggressive progress thread can lead to such a result. If the progression thread is scheduled very often, it will greatly delay the completion of the computation. However, it is possible that while being scheduled, the progress thread has nothing to progress. If the mandatory first operation of the algorithm on the local rank is to receive data, the progress thread may wait for these data each time it is scheduled. In case of MPI process imbalance, the data might be sent very late by the other MPI rank. If the expected data arrive once the computation is done on the local MPI process, then the whole collective communication will happen without any overlap, and the computation time would have been hindered by the progress thread.

4.2 Our methodology for assessing computation/communication overlap of NBC

In this section, we describe some points about methodology on how to measure performance and overlap on nonblocking collectives.

4.2.1 Multiple MPI processes Time Variation

By design, collective operations usually involve more than two MPI processes. Depending on the algorithm used for the collective operations, all MPI processes do not have necessarily the same amount of work. Some collectives will be very unbalanced such as `MPI_Gather` or `MPI_Bcast` when they use tree-based algorithms. In a *gather* implemented with a tree-based algorithm, the root will be receiving data on each step of the algorithm, while the leaves will only send data at operation start. These different workload will lead to different completion times on each MPI process.

Since many MPI applications are loosely based on the BSP model, or at least rely on collective communications to synchronize MPI processes, any late MPI process will delay the other ones. Thus, the overall performance depends on the slowest MPI process. We thus define the completion time of a collective as the time the last MPI process involved in this collective completes it locally.

Conversely, the start time of a collective is defined as the time where the first MPI process enters the MPI initiation call.

Hence, we define the collective global time as $\max(\text{end_times}) - \min(\text{start_times})$.

4.2.2 Clock Synchronisation and Barriers

To measure the time taken by the collective on each node, and to implement the aformentioned synchronized barrier, we need a global clock, *i.e.* a clock that gives the same time across nodes, as much as possible. In practice, each node has its own clock. Local clocks of each node may be set to different times, and even if they are set to the same time, they may drift (the time flows at a different speed) like any clock even though they are quartz-based. This problem is usually solved at the sytem level using NTP [59] to synchronize clocks. However, the synchronisation provided by NTP is not precise enough to measure communication times in the microsecond range. To effectively be able to synchronize all MPI processes across the multiple nodes used by the application, one must use a global clock synchronized between nodes.

We implement a global synchronized clock, taking into account both clock offset and drift using a method similar to the state of the art [19, 20]. Our method uses the clock of node #0 as reference clock and computes offset and skew for the clock of each other node. At initialization time, each node go through a calibration phase where it exchange its local time with node #0 with thousands of roundtrips. We are then able to compute the offset between the local clock and the reference clock, as well as the network latency to compensate for latency in clock exchanges.

Then, to compute global clocks, we use two different methods depending on the context:

- for timestamps used only in post-mortem analysis, like the duration of collectives, we perform another calibration phase at exit, to precisely determine the offset of clocks at the end of the benchmark in addition to offsets at the beginning. Then we are able to *interpolate* each date in local time, by knowing its offset with the reference clock at the beginning and at the end, if we assume that the drift is constant.

- for timestamps that we need to translate to global clock in real time, as used for synchronizing barriers, we perform a calibration phase at the beginning of the barrier, then we *extrapolate* each local time, by knowing its offset relative to reference clock at initialisation and at the beginning of the barrier, and assuming the time on each node will continue to flow at the same speed in the next seconds. To get a result precise enough, we ensure that the time between the barrier and initialisation is large enough so that extrapolation does not amplify errors. We insert sleep phases if needed.

With this mechanism, we are able to synchronize all nodes with node #0. It uses a simple loop for now, but could be extended to be hierarchical [60] in future work.

4.2.3 Fixed Computation and Communication Time

To thoroughly study nonblocking communications *overhead ratio* defined in Section 4.1.1, it is better to perform experiments with varying computation amount and communication data size, hence communication time. However, it can be very hard just looking at computation and communication sizes to assess their time, and which pair of sizes actually offer a fair matching. The time of communication depends on the number of nodes, the considered collective operation, the network hardware, just to name a few parameters. It is even worse when studying multiple MPI implementations. Each of them can use different algorithms, protocols or network interface. It can render a valid comparison point for a specific MPI implementation completely useless for another one.

Since it is not relevant to assess overlap in the case of computation time and communications time that are different by several orders of magnitude, we propose to have both scales, for computation and communication, use times rather than the number of operations for computation, and data size for communication. That way, we ensure the explored parameter space is relevant and it makes the results easier to interpret since we always know whether computation is shorter or longer than communication.

In our benchmark described in the next section, the size of data in computation and the exchanged data size in the collective are calibrated so as to reach the wanted duration.

4.3 Presentation of the BenchNBC Benchmark

In this Section, we present BenchNBC [61], the implementation of our NBC overlap benchmark, to measure the metrics defined in Section 4.1 and following the methodology presented in Section 4.2. We also describe how each metric is presented.

4.3.1 Benchmark Implementation

Estimation of Reference Times We measure the reference communication time $t_{\text{comm_ref}}$ first. It is defined as the duration of an MPI NBC initialisation call directly followed by an `MPI_Wait`. The wait ensures that the communication has completed. We use a NBC followed by a wait and not its blocking counterpart to be sure the same algorithm is used for the calibration and for the overlap benchmark. Then, as described in Section 4.2.3, we automatically find the data size to give to the collective so that it takes a given target time. We execute multiple runs and keep the median of all runs. Using the size and the time obtained this way, we approach the target time iteratively. We consider that this step has converged as soon as the error is below 10 %, as a compromise between precision and estimation time. It is important to note that to compute all the metrics defined in Section 4.1, we take for $t_{\text{comm_ref}}$ the actual measured communication time, not the target time.

The second reference time needed for this benchmark is the reference computation time $t_{\text{comp_ref}}$. The computation in the benchmark is supposed to be representative of a typical HPC application. We use a multi-threaded workload, with one OpenMP thread per core. Each thread executes a sequential GeMM. We measure the time on each thread. Then, as for communication, we automatically find the matrix size so that the matrix multiplication of two square matrices takes the target time (as reference we use the time on the slowest core). Then, for $t_{\text{comp_ref}}$ we take the actual measured computation time, not the target time.

In all of our benchmark, we did not use hyperthreading and kept one thread per physical core.

Overlap Benchmark The overlap benchmark itself is a combination of the same code used to measure communication and computation reference times except they are now interleaved. The goal is to have both communication and computation running concurrently.

Algorithm 1: Algorithm for overlap benchmark

```
Input: S1  // collective data size
Input: S2  // computation data size
t1 ← get_Time();
MPI_Icollective(S1);
t2 ← get_Time();
Compute(S2);
t3 ← get_Time();
MPI_Wait();
t4 ← get_Time();
```

The actual implementation is shown in Algorithm 1. Using the data and matrix size estimated with the technique described above, we first call the MPI collective function and the compute function just after. Finally, we call the `MPI_Wait` at the end of the last one.

The purpose of this test is to evaluate the ability of the runtime to make background progression. Thus, there is no call to explicit progression functions such as `MPI_Test`.

Then, using the time measured in the overlap benchmark shown in Algorithm 1, we are able to compute the inputs for our metric: $t_{\text{measured}} = t_4 - t_1$, $t_{\text{comp}} = t_3 - t_2$, $t_{\text{call}} = t_2 - t_1$, $t_{\text{wait}} = t_4 - t_3$.

The benchmark is designed to run with 1 MPI process per node, with all cores occupied with OpenMP threads. It is possible to reduce the number of OpenMP threads to leave one (or several) cores free for any MPI progress thread.

4.3.2 Metrics Display

We display the metrics described in Section 4.1 with different methods.

Overhead ratio display The main metric we show is the *overhead ratio* r_{overhead} as defined in Section 4.1.1. We represent it using a 2-D heat-map, with communication time as X-axis and computation time as Y-axis. The color map ranges from green ($r_{\text{overhead}} = 0$, perfect overlap) to yellow ($r_{\text{overhead}} = 1$, serialized execution), and further to red ($r_{\text{overhead}} > 1$, slowdown). Blue means $r_{\text{overhead}} < 0$ (faster than reference time), which usually indicates measure imprecision[1].

X-axis and Y-axis follow the same scale, hence the bottom left to top right diagonal represents cases where computation and communication times are equal. In the opposite corners, the two timings differ from several orders of magnitude ($500\,\mu s$ v.s. $1\,s$). Hence, the measures and their ratios are very sensible to execution noise, which leads to less meaningful observations. We didn't crop them for completeness, but they may be safely ignored when looking the 2-D heat-maps.

Impact on computation display The impact on computation, $r_{\text{MPI_impact}}$ is shown as a histogram, as in Figure 4.10; the red line highlights a ratio of 1 (no impact).

Concurrency ratios display *Concurrency ratios*, r_{comm} and $r_{\text{comp_slowdown}}$, are also presented as 2-D heat-maps, with the same axes as for the *overhead ratio*. The color map ranges from blue for the best case (for communication ratio, perfect overlap with $r_{\text{comm}} = 0\%$; for computation ratio, no impact with $r_{\text{comp_slowdown}} = 100\%$) to red for the worst case (communication slowdown with $r_{\text{comm}} > 100\%$; impact on computation with $r_{\text{comp_slowdown}} > 100\%$). We show these ratios in pairs, with communication ratio heat-map on the left, and computation ratio heat-map on the right, as in Figure 4.7.

4.3.3 Experimental setups

We present here our experimental setups for the tests with our benchmark. We run experimentations on various MPI implementations and network hardware. Our test platforms are:

[1]Warning to Mac users. The Apple Quartz engine tends to blur the figures displayed in this section, even for printing. To correctly view and print the figures, use a non-Quartz PDF renderer (*e.g.* Adobe or Chrome).

- *inti/skylake*: 32 nodes, with dual-socket Intel Xeon Platinum 8168, each with 24 cores at 2.7 GHz, equipped with Mellanox MT27700 (Connect-IB) InfiniBand boards;
- *inti/haswell*: 8 nodes, with dual-socket Intel Xeon E5-2698, each with 16 cores at 2.3 GHz, equipped with Mellanox MT27600 (Connect-IB) InfiniBand boards;
- *inti/sandy*: 64 nodes with dual-socket Intel Xeon E5-2680, each with 8 cores at 2.7 GHz, equipped with Mellanox MT25400 (ConnectX-2) InfiniBand boards;
- *irene/bxi*: 64 nodes, with Intel Xeon Phi 7250 with 68 cores at 1.4 GHz, equipped with Bull BXI v1.2 network adapters;
- *bora/omnipath*: 8 nodes with dual-socket Xeon Gold 6240 each with 18 cores at 2.60 GHz, equipped with Omni-Path HFI Silicon 100 Series network adapters.

We run one thread per physical core, without hyperthreading. We tested four different collectives for each configuration: `MPI_Ibcast`, `MPI_Ireduce`, `MPI_Iallgather` and `MPI_Ialltoall`.

4.4 Benchmark cases study

In this section, we present some benchmark results in two typical cases: OpenMPI/InfiniBand in *basic* configuration and MPICH with *async progress*; and detail how to use our metrics to diagnose pathological behaviours. We then compare these results and the approach of our benchmark with state-of-the-art benchmarks presented in section 3.1: IMB [15] and OSU [16]. We will then apply these benchmarks to the same cases.

4.4.1 Case study: OpenMPI/InfiniBand

We present here results for the specific case of OpenMPI 4.0.2 on the two machines *inti/haswell* (using 8 nodes) and *inti/sandy* (using 64 nodes), both with an InfiniBand network.

Figure 4.6 shows *overhead ratio* results for `MPI_Ibcast` and `MPI_Ireduce`. In most cases, we observe no overlap at all (yellow areas), with $r_{\text{overhead}} \approx 1$, and even *slowdown* in some cases (red areas, with $r_{\text{overhead}} > 1$. This poor performance may be surprising for some users, but it gets easily explained: OpenMPI does not feature mechanisms for asynchronous progression. It is expected to observe no overlap in case there is no background progression mechanisms in the MPI library, like described in Section 4.1.3.

To explain the lack of overlap, Figure 4.7 shows concurrency ratios r_{comm} and $r_{\text{comp_slowdown}}$ for OpenMPI on `MPI_Ibcast` and `MPI_Ireduce`. `MPI_Ibcast` on 8 *inti/haswell* nodes exhibits an r_{comm} typical of a purely sequential behaviour: MPI calls are as long as without overlap, which shows it was not executed in background; the computation is unaffected. For `MPI_Ireduce`, r_{comm} (left) is red, indicating that the communication is slowed down by the computation, while the computation (right) is not impacted. The difference between `MPI_Ireduce` and `MPI_Ibcast` is that `MPI_Ireduce` needs to execute the reduction operation on the CPU; these results show that the reduction

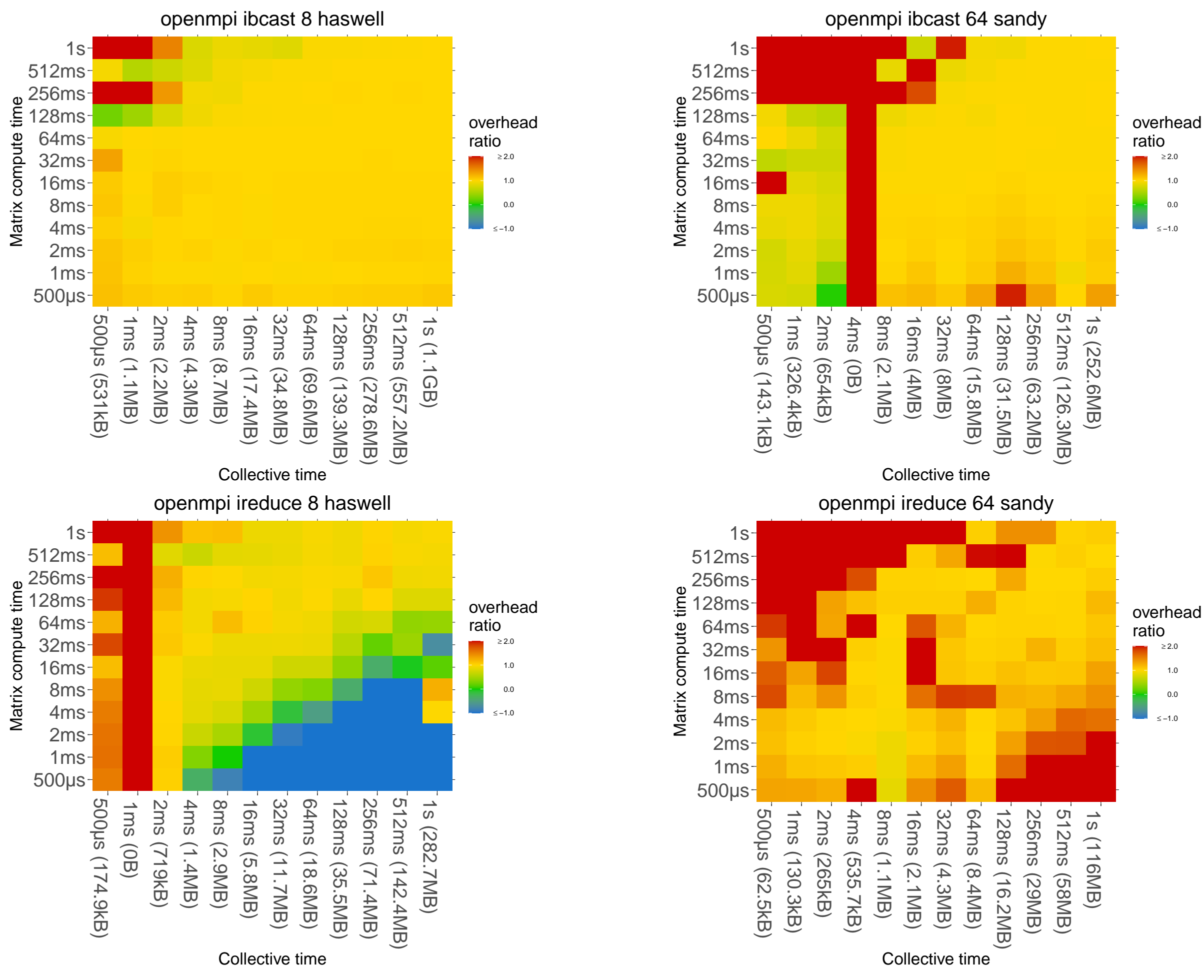


Figure 4.6: OpenMPI 4.0.2 overlap results on InfiniBand for `MPI_Ibcast` and `MPI_Ireduce`

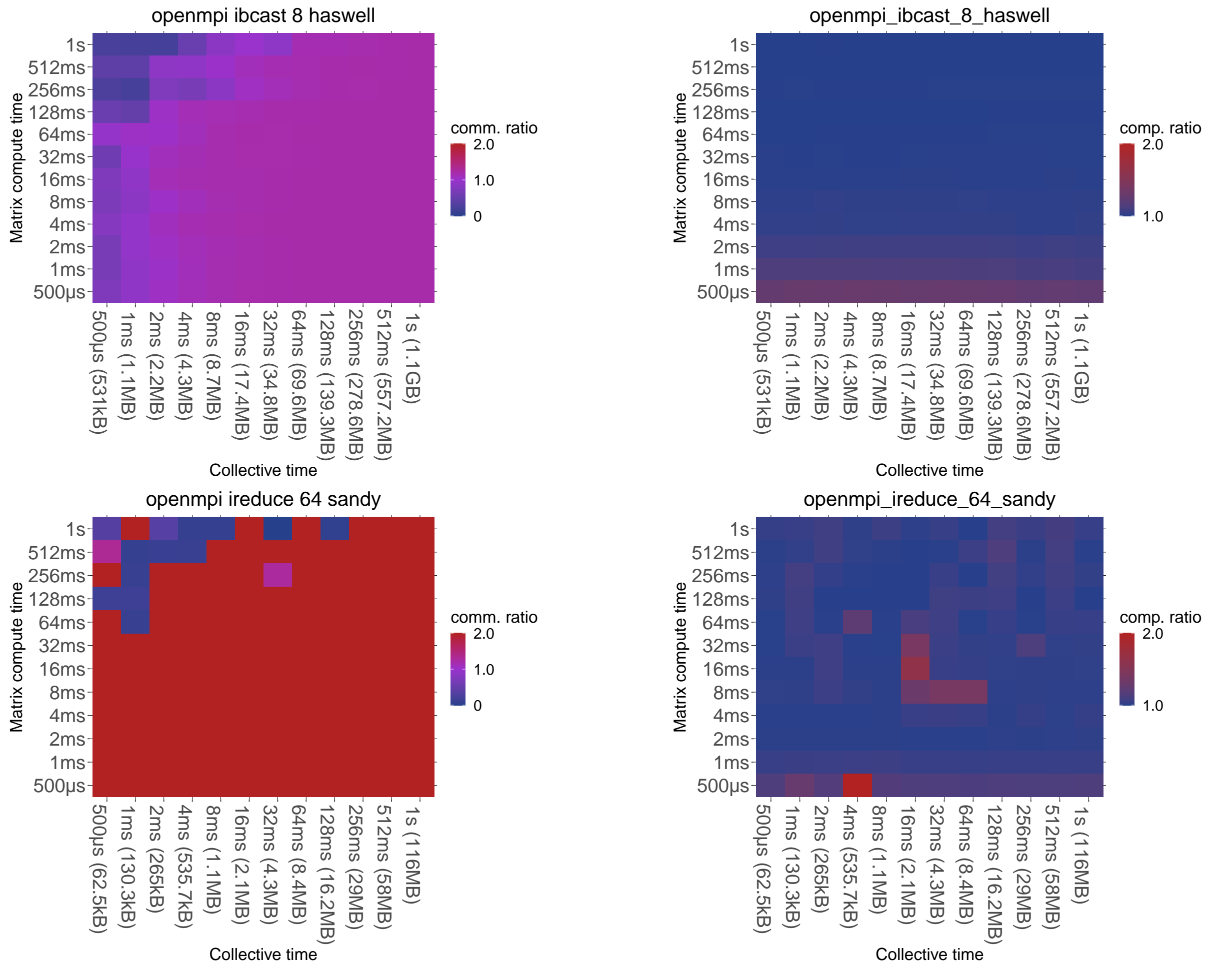


Figure 4.7: OpenMPI concurrency ratios r_{comm} (left) and $r_{\text{comp_slowdown}}$ (right)

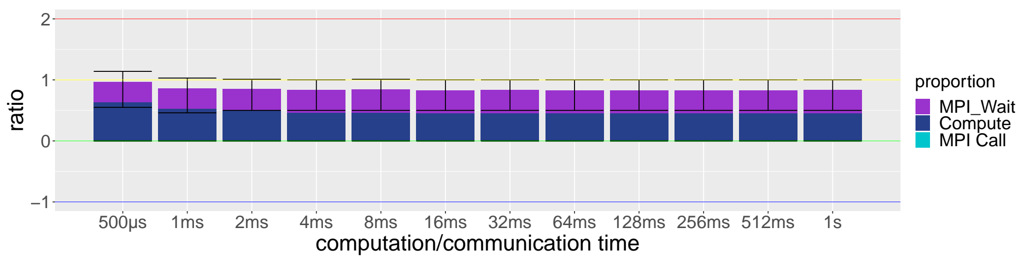


Figure 4.8: OpenMPI all ranks global ratios `MPI_Ibcast` on 8 nodes

speed actually suffers from being run alongside computation. `MPI_Ireduce` on 64 nodes is the case where the communication time is the most impacted.

Moreover, we observe some green areas with $r_{\text{overhead}} < 1$ (or even blue) for `MPI_Ireduce` on *inti/haswell* on Figure 4.6, which indicates successful overlap in this part of the parameters space. The green area is located below the diagonal, thus with computation time larger than communication time. We can guess that a single step of the collective algorithm actually overlaps communications with computation, thanks to hardware progression for point-to-point operations, then the remainder of the collective does not have the opportunity to be scheduled on the CPU while the computation is running and is scheduled only at the end. Such mechanism only works when the computation is shorter than a single step of the collective algorithm, so much shorter than the full collective time (computation 4 times shorter than communication, which corresponds to a single level in a reduction tree on 8 nodes). Obviously, the more nodes, the more insignificant this phenomenon becomes.

The figure 4.8 feature the all rank global ratios. This figure features detailed and gathered results for a sub part of heatmap results. Each bar represent a case along the diagonal of equals computation and communication times. The total height of the bar represent the median overhead ratios for all ranks. There is also an upper extreme representing the highest overhead ratio among all ranks and lower extreme for the smallest overhead ratio among all ranks. Finally, the colours composing the bar plot gives the proportion of time spent in MPI call (light blue), `MPI_Wait` (purple) and computation (dark blue). For example in this case, median overhead ratio is below 1.0, meaning that most of the rank actually overlap a bit. The max overhead ratio is equal to 1.0 for most cases, so for each execution at least one rank is not able to overlap which make the whole collective not winning time. The proportion of the bar is half blue and purple meaning that half of time have been spent in computation and wait. As the experiment used the same computation and communication time, we can then conclude that the execution was serialized.

As a conclusion, in most cases no overlap is observed with OpenMPI, which is not surprising given that it does not implement asynchronous progression mechanisms. We have shown that there is overlap in some anecdotal cases with very short computation and a low nodes count. We have exhibited pathological cases (*e.g.* `MPI_Ireduce`) where the communication is *slower* than if no overlap would have been attempted. In the general case, the observed performance is roughly the same as if computation and communication would have been run sequentially.

4.4.2 Case study: MPICH with `MPICH_ASYNC_PROGRESS=1`

MPICH features an optional progress thread that can be activated through the `MPICH_ASYNC_PROGRESS` environment variable. Results for the overhead ratio are shown on Figure 4.9. As expected, we observe successful overlap in some cases. However, it only works for large compute sizes and large communication sizes. With smaller sizes, there is a significant slowdown, worse than without the progress threads. Actually, the behaviour looks like it is *chaotic*, with intermixed zones with $r_{\text{overhead}} \gg 1$ and zones with $r_{\text{overhead}} < 0$. This is due to huge variability of performance, with standard deviation between 4 ms and

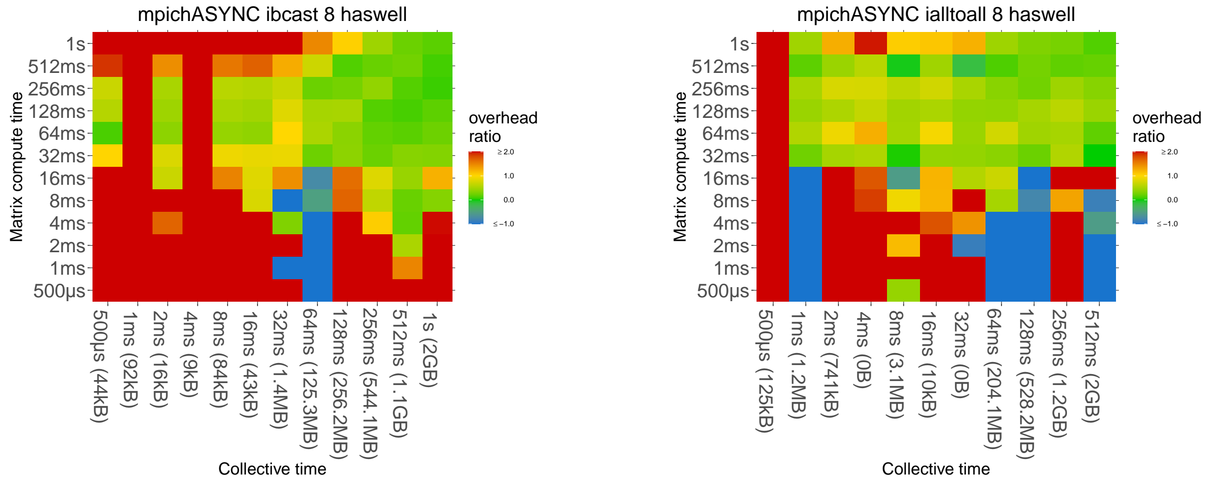


Figure 4.9: MPICH overlap results with async progress thread

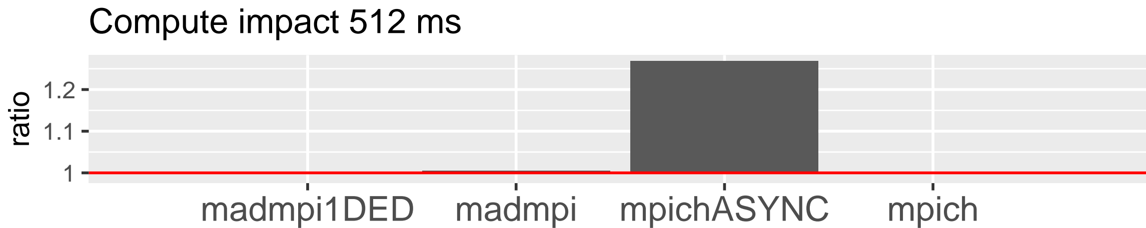


Figure 4.10: Runtime impact comparison on 512 ms computation time, for MadMPI (left 2 bars - dedicated core and no dedicated core) and MPICH (right 2 bars - async progress enabled and disabled)

11 ms. For larger sizes, these variations become negligible and overlap can be seen. For small sizes, these variations prevent from having meaningful times and observations. We assume that the high variability of performance comes from perturbations from the MPI runtime with its progression thread.

To confirm this hypothesis, we measured the impact of the MPI implementation on the computation, using the $r_{\text{MPI_impact}}$ introduced in Section 4.1.2. The ratio is shown in Figure 4.10 for estimated compute times of 512 ms, on the right two bars for MPICH with asynchronous progression and without progression. Overall, for all estimated compute times, MPICH with an asynchronous progress thread has a huge impact on the computation, even without actually doing any MPI communications, with a computation slowdown up to 50 % for the worst case. This is explained by the progress thread being scheduled by the kernel on cores performing computation, stealing CPU cycles and thus delaying computation.

To understand the impact of MPICH progress thread in the case of overlap, we take a look at the concurrency ratios r_{comm} and $r_{\text{comp_slowdown}}$, as introduced in Section 4.1.2, for the `MPI_Ireduce` case, in Figure 4.11. We observe that the bad overhead ratio observed for small compute sizes and small communication sizes comes from two different reasons. For small compute times, the overhead comes from the impact on computation. It confirms that the MPICH runtime in this configuration has a huge impact on the computation. Hence, the slowdown induced on the computation jeopardizes the global performance,

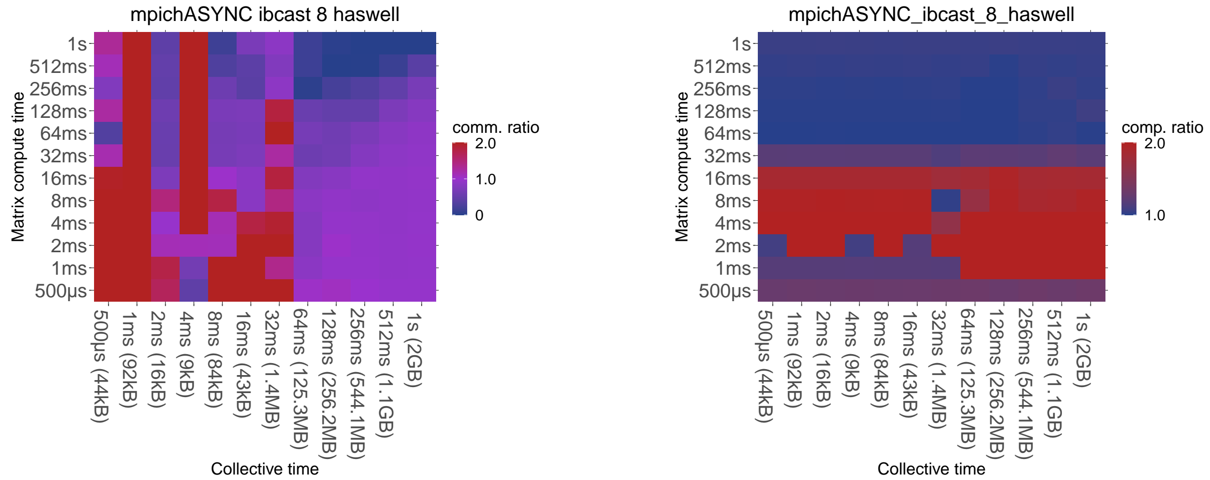


Figure 4.11: MPICH concurrency ratios with async progress thread r_{comm} (left) and $r_{\text{comp_slowdown}}$ (right)

even with perfect overlap. For communications, above a 8 ms threshold, we observe that the larger the times, the smaller the communication time spend in the `MPI_Wait` function, until the communication time is completely overlapped by the computation. For small communication sizes, the observed overhead comes from communication slowdown. In these cases, the extra cost necessary to use a progress thread is too high to be hidden by the communication overlap gain when CPU computations are required.

4.4.3 Comparison with state-of-the-art

We compare here our metrics and benchmark with state-of-the-art benchmarks for non-blocking collectives: IMB [15] and OMB (OSU microbenchmark) [16]. To do so, we compare the results on both cases studied in sections 4.4.1 and 4.4.2.

General idea. The general principle of both OMB and IMB is the use of an *overlap ratio*, defined as the ratio of time spent in the MPI primitives with overlap, relative to the same time with blocking MPI calls. Overlap is done with a computation running approximately the same time as the reference communication time. This way, they evaluate how much time spent in the MPI primitives has decreased. In essence, this is very similar to our r_{comm} metric.

In contrast, the *overhead ratio*, our main metric, compares what we get on the overall time with what we expected if perfect overlap would happen. As a consequence, our benchmark captures behaviours where the progression mechanisms interfere with computations and make them less efficient.

Comparison of benchmarks using OpenMPI/InfiniBand. The first case for our comparison is OpenMPI as described in section 4.4.1. The results obtained with OSU benchmark v5.7 for 1MB `MPI_Ibcast` and `MPI_Ireduce` are depicted in Figure 4.12. It shows a 41.70% overlap ratio for the broadcast and 23.79% for the reduction. In comparison, our results for the same reduction case shown in Figure 4.6 are in 2-D,

```
OSU MPI Non-Blocking Broadcast Latency Test v5.7
# Size           Overall(us)       Compute(us)    Pure Comm.(us)     Overlap(%)
 1048576            665.10            424.78            412.18           41.70

# Size           Overall(us)       Compute(us)    Pure Comm.(us)     Overlap(%)
 1048576           1226.44            706.79            681.85           23.79
```

Figure 4.12: OSU benchmark results for OpenMPI on 8 nodes

```
OSU MPI Non-Blocking Broadcast Latency Test v5.7
# Size           Overall(us)       Compute(us)    Pure Comm.(us)     Overlap(%)
 1048576           2098.49           1888.42           1832.79          88.54

# Size           Overall(us)       Compute(us)    Pure Comm.(us)     Overlap(%)
 1048576            851.56            663.64            643.58           70.80
```

Figure 4.13: OSU benchmark results for MPICH on 8 nodes with asynchronous progression

with computation time $t_{\text{comp_ref}}$ and communication time $t_{\text{comm_ref}}$ not necessary equal. We observe on our graphs that results are non-uniform, thus it is valuable to measure overlap for a computation time different than the communication time. In practice, an application programmer tries to overlap communications with computation, but he/she has no guarantee their duration will be equal, depending on the CPU speed, network speed, and number of nodes, performance is hard to predict. Our results cover a wider range of values, though the conclusion is not radically different: overlap is not very good in this case.

Because we display various computation time and communication time, we can observe that a better *overhead ratio* is available for the reduction case (middle left 2-D heat-map in Figure 4.6, labeled *openmpi ireduce 8 haswell*). If the communication time is greater than the computation, then we have a greener diagonal, indicating a better overlap. This is easily explained thanks to the two *concurrency ratios*, r_{comm} and $r_{\text{comp_slowdown}}$, as described in Section 4.4.1. In the bottom 2-D heat-maps of Figure 4.7, r_{comm} (left) is red, indicating that the communication is slowed down by the computation, while the computation (right) is not impacted. Hence, to have a perfect overlap, the *slowed* communication should not take more time than the computation. Because the slowed communication takes twice (or more) the amount of time of the communication without computation, having an initial communication time smaller than the computation time will bring a better overlap. If a user can influence the nonblocking communication/computation time ratio in its program, having such information can help leverage better performance.

Such information cannot be deduced from the IMB or OSU measurements, but only with our extended metrics.

Comparison of benchmarks using MPICH + asynchronous progression. The second case is MPICH with asynchronous progress thread. We consider `MPI_Ialltoall` and `MPI_Ireduce`, with 1MB of data, on 8 nodes of the *inti/skylake* platform. Our results are described in section 4.4.2 (Figure 4.9), and OSU benchmark and IMB results are shown in Figure 4.13 and Figure 4.14 respectively.

```
#    Intel(R) MPI Benchmarks 2018, MPI-NBC part
#bytes    #repet.    t_ovrl[usec]   t_pure[usec]  t_CPU[usec]   overlap[%]
 1048576       40         2281.20         1888.82     2052.86          80.89

#bytes    #repet.    t_ovrl[usec]   t_pure[usec]  t_CPU[usec]   overlap[%]
 1048576       40           981.55          790.55        803.90          76.24
```

Figure 4.14: IMB benchmark results for MPICH on 8 nodes with asynchronous progression

We can see that there is a huge difference between OSU/IMB-NBC and our benchmark: our benchmark shows that overlap causes an overall 10× *slow down* ($r_{\text{overhead}} = 10.96$ for `MPI_Ialltoall` and 13.07 for `MPI_Ireduce`, on the considered data size), while other benchmarks tend to show it overlaps successfully (*overlap* = 70.80% to 88.54%). It is explained by multiple factors:

- nature of metric: our metric, which is an *overhead* relative to perfect overlap, is an open scale; it can express the fact that an overlap leads to worse performance than if no overlap would have been attempted.
- realistic computation: the computation method in our benchmark uses OpenMP, and thus exploits all cores, compared to the single-threaded computation in other benchmarks. We believe our approach is closer to real applications, and is able to show thread interaction (applications threads and MPI progress threads) that other benchmarks would overlook.
- computation fixed time v.s. fixed amount: our benchmark runs a predefined amount of work, so as to be able to compare the duration of computation with and without overlap, and thus be able to detect the impact of overlap on computation speed. For computation, the other benchmarks use a loop that runs for a given time, without being able to tell whether the amount of computation was hindered by communication progression mechanisms. Our metric $r_{\text{comp_slowdown}}$ is designed to pinpoint this issue.
- timing issues: other benchmarks take the average of time between nodes as the collective time, while we take the time of the last rank to complete the collective. It is important especially for collectives implemented as trees, with a huge load-imbalance between nodes. Moreover, thanks to our synchronized clocks, we take the median time of a number of iterations, while others use simply the average.

Our interpretation on this MPICH +progression case, as explained in section 4.5.3, is that activating the progress thread makes him collide with OpenMP application threads, causing a huge overhead: the progress thread needs to be woken up, then it competes with application threads, leading to slow computation, and slow communication. We observe that this slowed-down communication is actually overlapped, even though this is probably not what the user wants. All these phenomenons are overlooked by other benchmarks that simply conclude that there is overlap.

More generally, IMB and OSU benchmarks try to measure whether there is overlap or not in the case of perfectly matching computation and communication time. Our

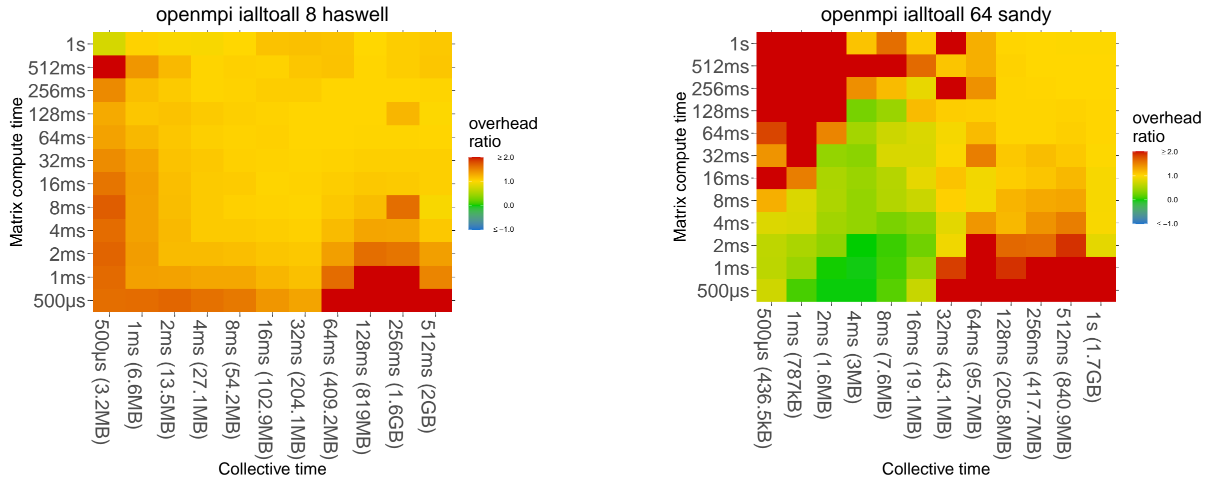


Figure 4.15: OpenMPI 4.0.2 overlap results on InfiniBand for `MPI_Ialltoall`

benchmark consider more realistic cases (various computation and communication times) and tries not only to assess overlapping but to diagnose pathological behaviours. It may be useful for MPI library developers, but for end-users too, *e.g.* to diagnose thread conflicts between the MPI library and the application.

4.5 Survey of Overlap Benchmark Results with Various MPI Implementations and Networks

We present here a survey of the results obtained with our benchmarks on a large set of configurations and MPI libraries described in Section 4.3.3. See [61] for more complete results.

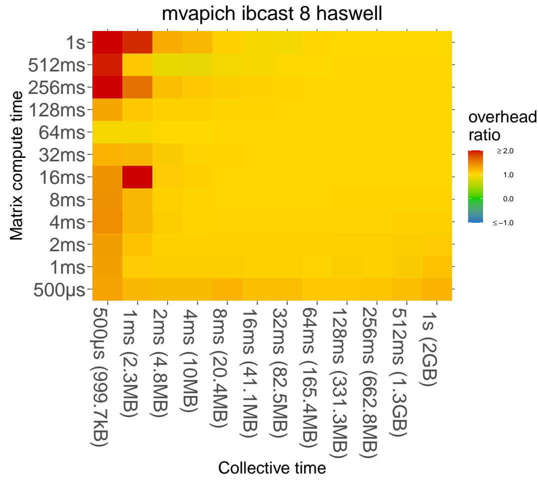
We distinguish three different deployment configurations:

- *basic* with computation on all cores, and default configuration of MPI library;
- *progress* with computation on all cores, and asynchronous progression mechanisms enabled if available;
- *dedicated* with computation on $n-1$ cores, with progress thread enabled and bound to the free core if possible.

4.5.1 Results with basic configuration

We present here results for the *overhead ratio* we get with widespread MPI implementations and their default configuration. Due to space constraints, we cannot include results for all MPI implementations, on all machines, for all collective operations. The selected results are typical of what we obtained.

These results are obtained on InfiniBand network, on machines *inti/haswell* and *inti/sandy*. Figure 4.15 shows results for OpenMPI 4.0.2, in addition to Figure 4.6 already studied in Section 4.4.1, Figure 4.16 for MVAPICH 3.2.1, Figure 4.17 for MPICH 3.3, and Figure 4.18 for MPC.



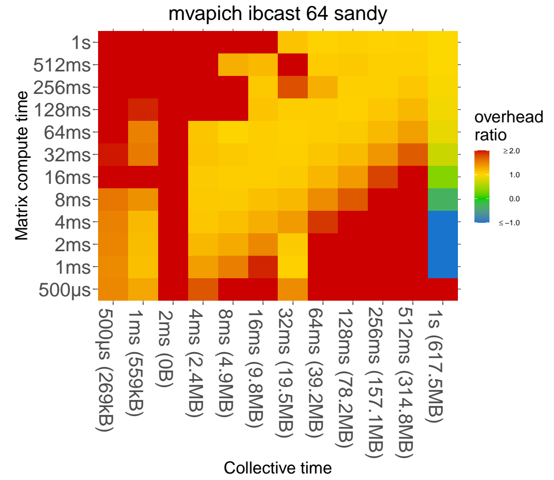
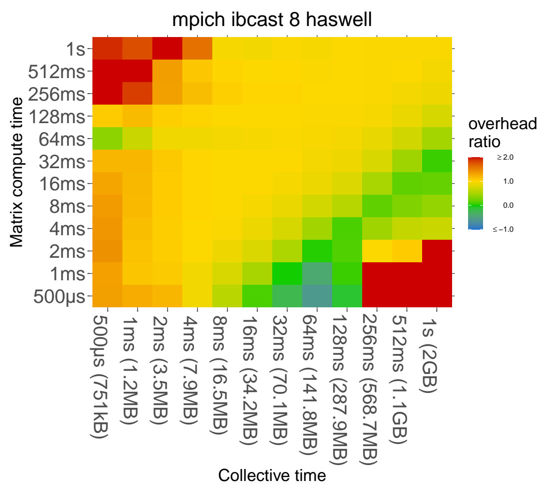


Figure 4.16: MVAPICH 3.2.1 overlap results on InfiniBand for `MPI_Ibcast`



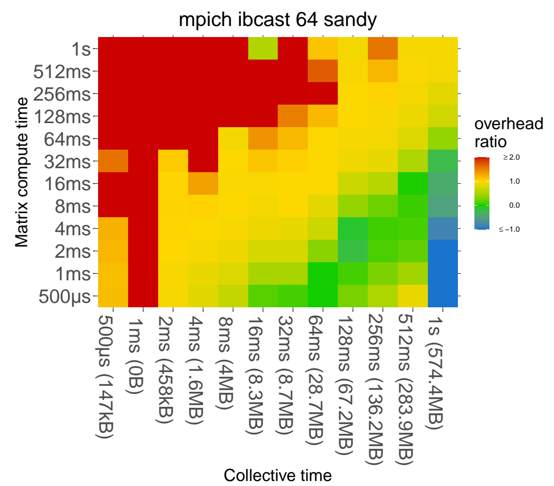
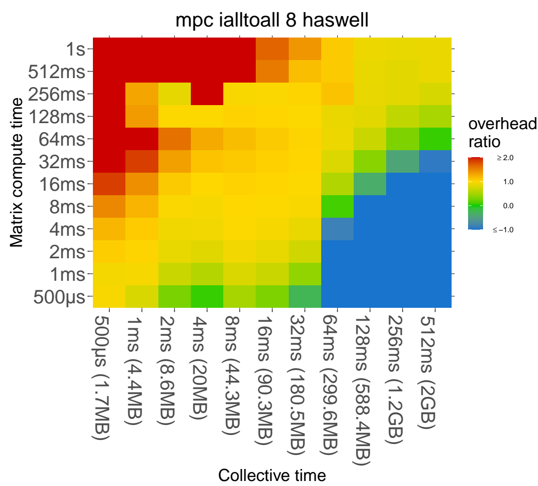


Figure 4.17: MPICH 3.3 overlap results on InfiniBand for `MPI_Ibcast`



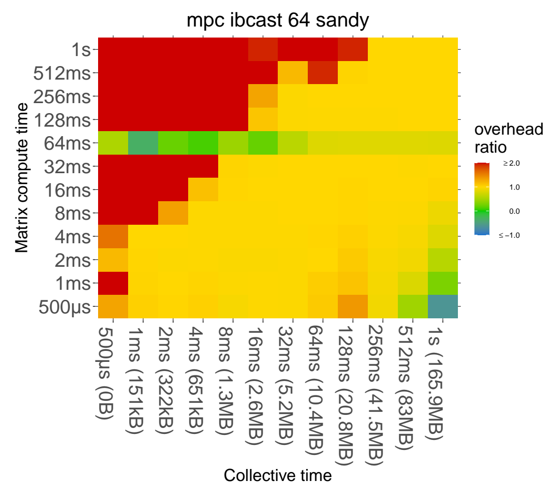


Figure 4.18: MPC overlap results on InfiniBand for `MPI_Ialltoall` and `MPI_Ibcast`

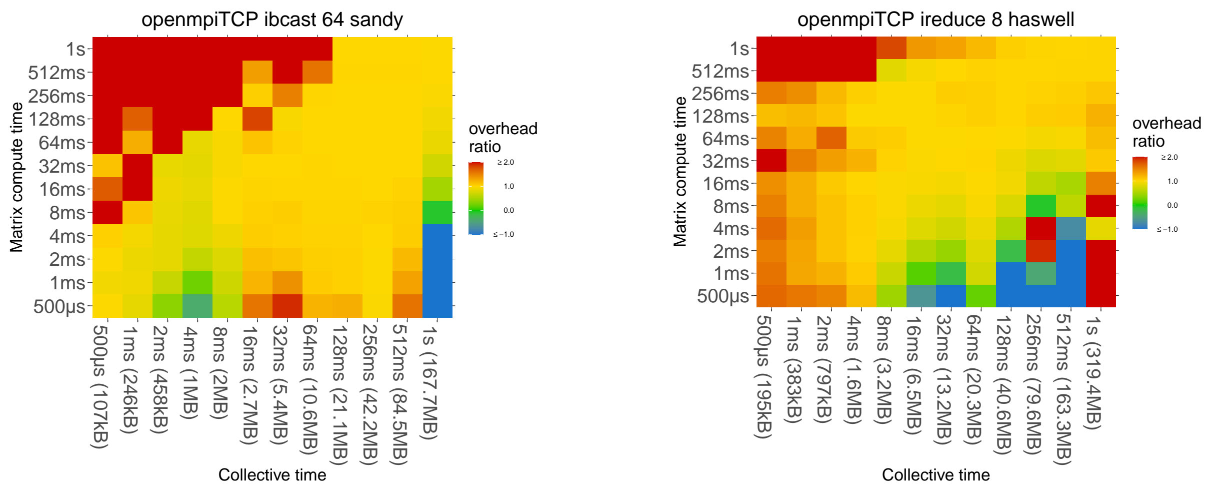


Figure 4.19: OpenMPI 4.0.2 overlap results on TCP-Ethernet for `MPI_Ibcast` and `MPI_Ireduce`

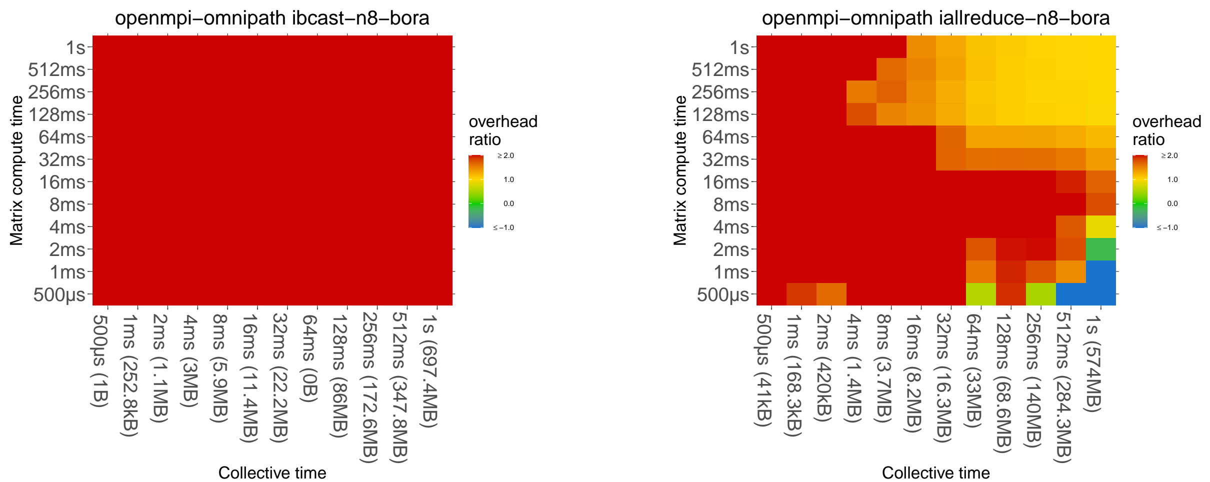


Figure 4.20: OpenMPI 4.0.2 overlap results on Omni-Path for `MPI_Ibcast` and `MPI_Iallreduce`

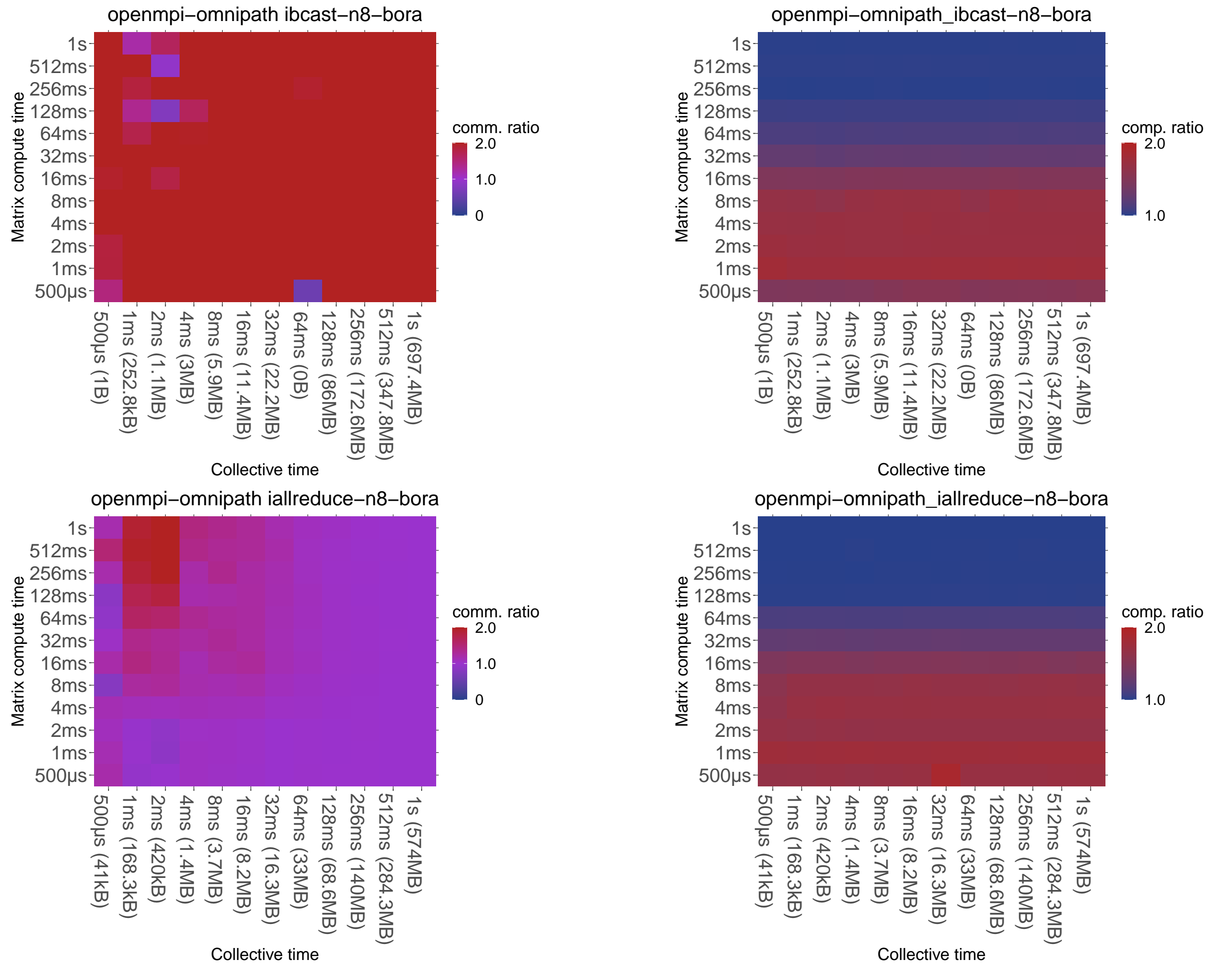


Figure 4.21: OpenMPI concurrency ratios r_{comm} (left) and $r_{\text{comp_slowdown}}$ (right) on Omni-Path

We also present results obtained for OpenMPI with TCP-Ethernet on *inti/sandy* on Figure 4.19, and with Omni-Path on *bora/omnipath* on Figure 4.20.

Since we are in the basic configuration here, the MPI implementations do not feature mechanisms for asynchronous progression. Most of these results are similar to the case studied in Section 4.4.1: no progression is observed and communication and computations are serialized (yellow areas), or they are contending with each other so the overall performance is slower than serialized (red areas). We get the same green areas with $r_{\text{overhead}} < 1$, which is a sign of successful overlap, in the bottom right portion of graphs, where computation is much shorter than communication.

We observe another green area for OpenMPI on small communication and short computation for `MPI_Ialltoall` on sandy (Figure 4.15), probably because without *rendezvous* and without dependency between messages as in `MPI_Ialltoall`, progression is performed fully by the hardware as independent point-to-point operations.

The overlap results for Omni-Path are worse than the other OpenMPI results. The corresponding concurrency ratios in Figure 4.21 show that both communication and computation are impacted on this machine, jeopardizing any chance of performance improvement through overlap.

As a conclusion, with basic configuration, for a large set of MPI libraries (OpenMPI, MVAPICH, MPICH, MPC), on various networks (InfiniBand, TCP-Ethernet, Omni-Path), no overlap is observed except in some insignificant cases. We even observe performance degradation in some cases.

4.5.2 In depth interpretation of these results using complementary metrics

The overhead ratio is able to tell us if we gain time or not in an overlap situation. For example, the figure 4.15 with 8 haswell node tell us that there is no gain nor loss time, i.e, that the total execution time is equivalent to a serialized execution. This can be caused by no progression being done in background causing an actual serialized execution, but it can also be caused by a progress mechanism slowing down the computation to actually take the same time as the serialized execution. These two cases must be discriminated as they need different solution to be addressed. Respectively, enhancing the progression engine in the first case, and managing the resource sharing in the second case.

The figure 4.22 show the concurrency ratios for the same execution. These results show that the global time spent in `MPI_Wait` for the overlapped situation is equal to the reference communication time. Also, the computation is not impacted at all. All these results lead to the conclusion that the execution is finally serialized. In the second case where the progression mechanism allow overlap but slow the application down, the communication concurrency would be closer to 0 with blue zones and the computation concurrency ratio higher with red zones. The all rank global ratio figure lead to the same interpretation with overhead ratio equal to 1 and time proportion split equitably between computation and communication.

Finally, the figure 4.23 shows the same ratios for the 64 nodes execution of `MPI_Ialltoall`. The communication concurrency ratio shows the limit between both *eager* and *rendezvous* protocol, as the vertical frontier between 16ms and 32ms. The all rank overhead ratio figure shows that for value under this frontier, the majority of time spent is in the computation part. This reinforce the hypothesis that for these small cases, `MPI_Ialltoall` is separated in point to point operation successfully handled by hardware.

4.5.3 Results with asynchronous progression enabled

We present here results with asynchronous progression enabled.

MPICH with `MPICH_ASYNC_PROGRESS=1`

MPICH with `MPICH_ASYNC_PROGRESS=1` has been detailed previously in Section 4.4.2. In some cases, overlap is successful; in other cases, it causes contention between the progress thread and computation, which causes a huge performance drop.

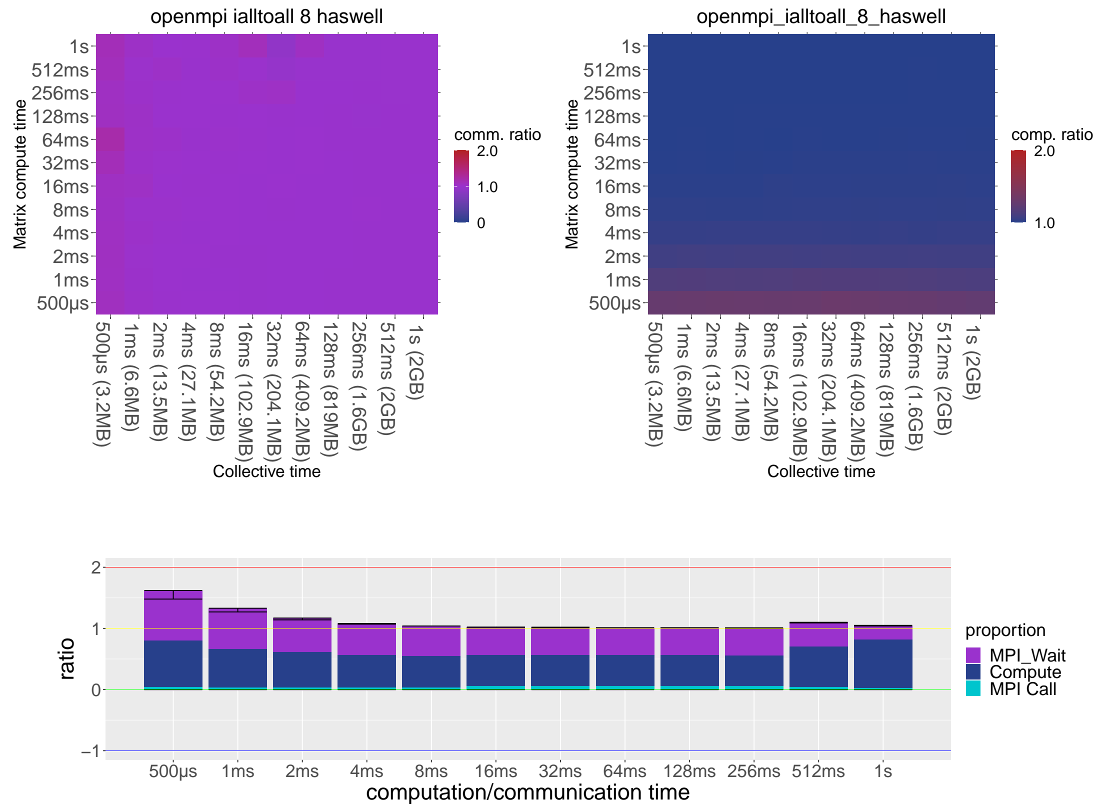


Figure 4.22: Concurrency ratios and all ranks overhead ratio for `MPI_Ialltoall` on 8 Haswell nodes

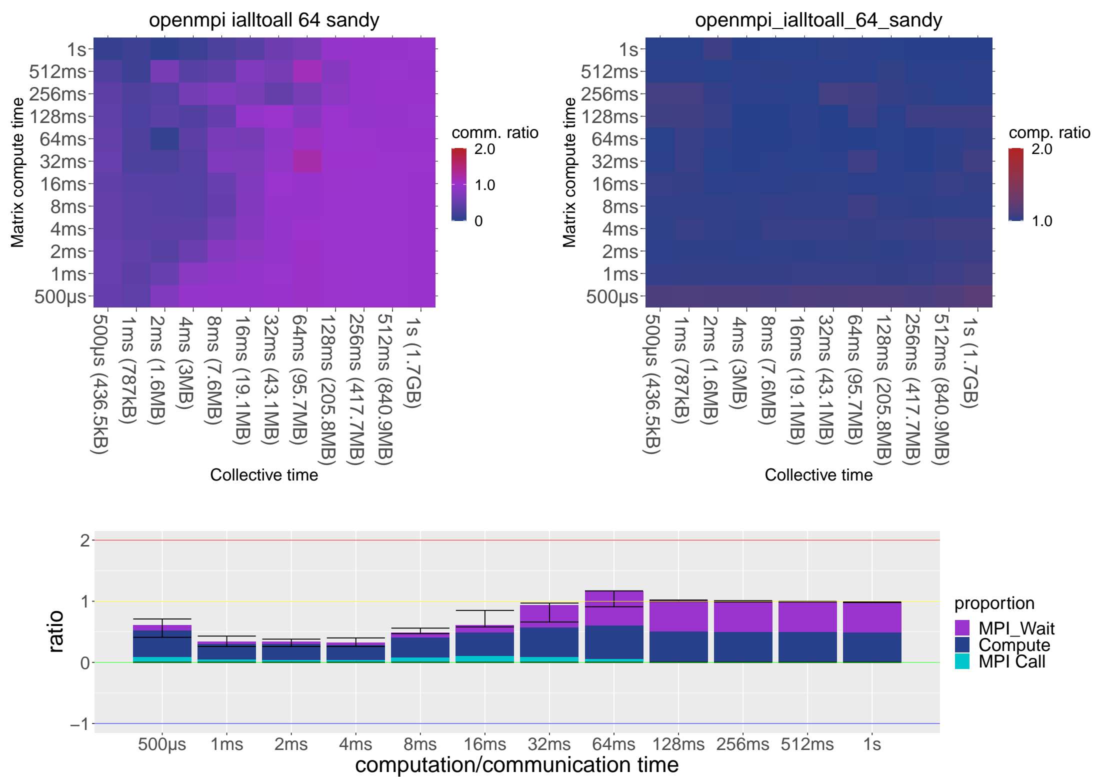


Figure 4.23: Concurrency ratios and all ranks overhead ratio for `MPI_Ialltoall` on 64 Sandy Bridge nodes

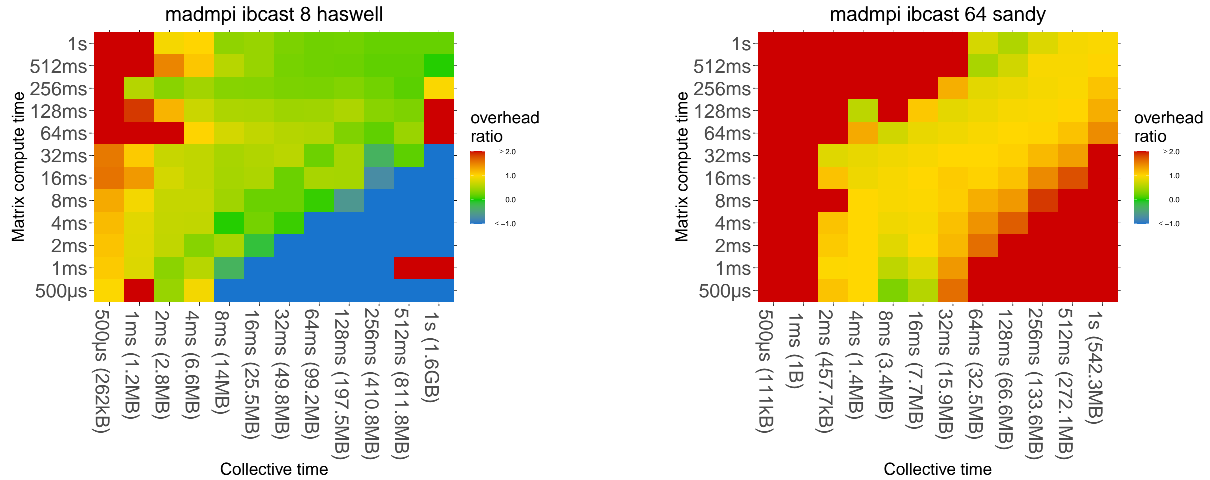


Figure 4.24: MadMPI overlap results with progress thread

MVAPICH on InfiniBand

MVAPICH also provides a version supporting progress threads: MVAPICH2-X. Unfortunately, it is distributed as binary-only and we were unable to correctly install and use MVAPICH2-X 3.2 on our test machines, thus we cannot provide any result with it.

MPC

Recently, MPC exhibited good performances when activating its progress thread [62]. However, this support appears to be broken in the tested version, thus no results can be provided.

MadMPI with default progression

MadMPI enables progression mechanisms [30] by default. Figure 4.24 shows its overhead ratio for `MPI_Ibcast`. We can observe successful overlap on 8 nodes, but no overlap or even slowdown on 64 nodes.

Figure 4.10 (in Section 4.4.2) shows that MadMPI progression mechanism has nearly no impact on computation when no communications occur, since its background progression mechanisms are designed to have more gentle polling strategy than busy-waiting. Progression mechanisms in MadMPI were initially designed to handle nonblocking point-to-point. The workload of collective being heavier, sometimes its progression mechanisms are not enough to achieve good overlap.

Figure 4.25 displays the concurrency ratios for `MPI_Ibcast` on 64 nodes. We observe on these figures that the reason for the bad overhead ratio for this case is twofold. First, with a r_{comm} around 100% for the most part, it seems communications are not overlapped at all. For the cases where communications are overlapped, the purplish spots for $r_{\text{comp_slowdown}}$ indicates computation is impacted. Since on this part of the figure, computation time is larger than communication times, no performance gain is visible.

This behaviour may be explained by the fact that in this configuration, progression is performed by a thread with uncontrolled placement, thus colliding with computation

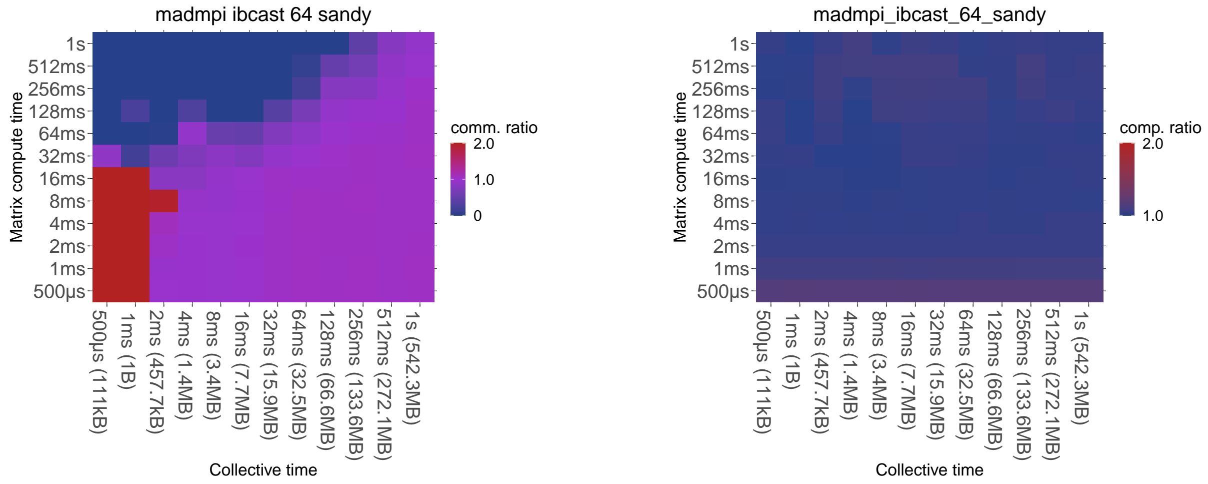


Figure 4.25: MadMPI concurrency ratios

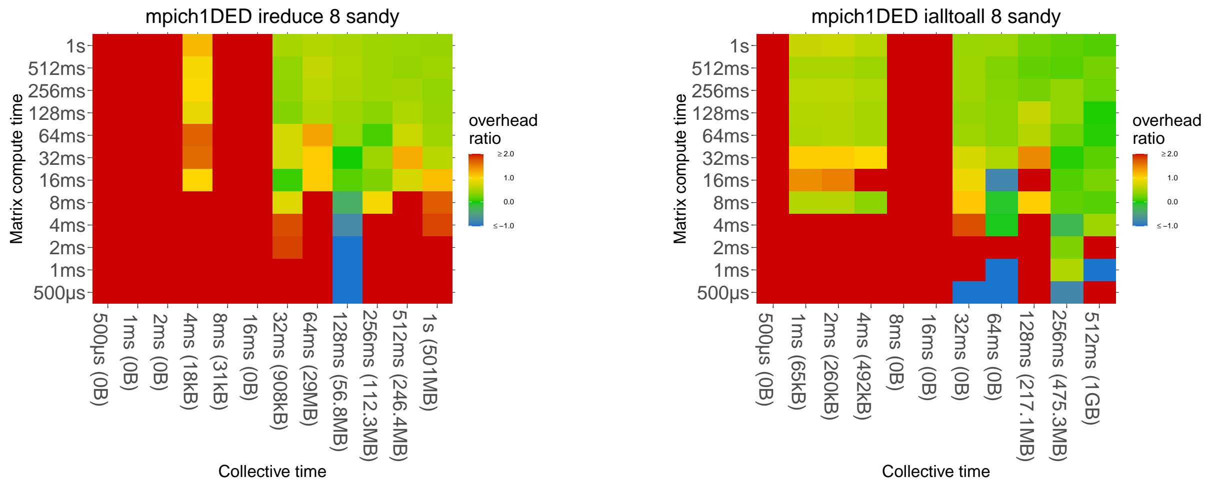


Figure 4.26: MPICH overlap results with dedicated core

threads.

4.5.4 Results with dedicated core for progression

We present here results with a core dedicated to communication progression.

MPICH on InfiniBand with dedicated core

We tried to dedicate a core to MPI progression with MPICH, by running computation on all cores except one, and enabling `MPICH_ASYNC_PROGRESS=1`. Unfortunately, we found no mechanism to bind the progress thread to the dedicated core, so its placement was handled by the kernel thread scheduler. The observed overhead ratios are shown in Figure 4.26. They are very similar to the ones depicted in Figure 4.9 without the dedicated core.

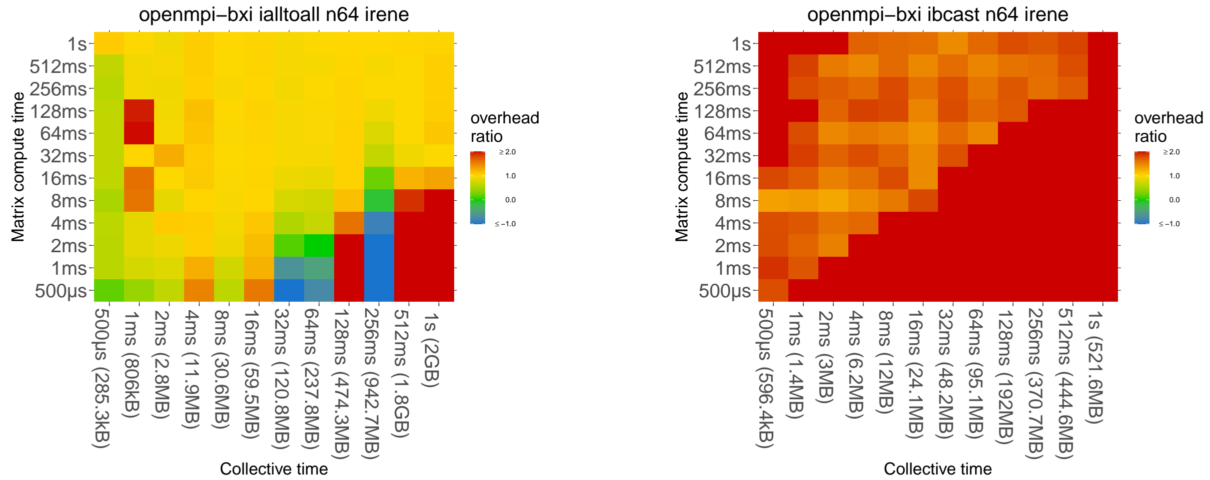


Figure 4.27: OpenMPI overlap results on BXI

4.5.5 Results with hardware-based progression

Bull BXI [58] network is designed to handle progression in hardware. Overhead ratios for Bull BXI v1.2 interconnect with OpenMPI 2.0.4.5-bull on machine irene/bxi are shown on Figure 4.27.

We observe mostly a sequential behaviour on `MPI_Ialltoall` and other collectives. The worst case happens with `MPI_Ibcast`, where the ratios show a slowdown compared to sequential execution. This strange behaviour was observed only for this collective.

BXI is designed to handle collective offload to the network card. Unfortunately BXI v1.2 cards do not properly support this feature. BXI v2 cards have been announced, and should fix support for hardware-assisted progression for collectives, so as to reach better overlap in the future.

Metrics use to address the existing mechanisms issues

In this chapter, we introduced metrics designed to evaluate the performance of progression mechanisms and their impact on the overall performances. The section 4.5 showed that existing progression mechanism fail to associate a good overlap and a restricted impact on the application. Without addressing these two issues in a single mechanism, using overlap to gain performance stays difficult.
In the next chapter, we use these metrics to conceive a progression mechanism with a reduced impact on the overall and a good overlap capability in order to gain performances.

Chapter 5

A dedicated core mechanism for communication progression

Contents

5.1 Using a dedicated core for asynchronous communication progression . . . 78

5.1.1 Analysing core dedication relevance for progression 78

5.1.2 Implementation of dedicated cores 79

5.1.3 Dedicated core cohabitation with OpenMP threads 81

5.2 Implementing NBC in NewMadeleine 82

5.2.1 NewMadeleine algorithms for nonblocking collectives 82

5.2.2 Implementation of collectives in NewMadeleine 84

5.3 Pioman dedicated core benchmarking 88

5.3.1 MadMPI dedicated core performance using BenchNBC 88

5.3.2 Analysis of collective behaviour using concurrency and all ranks metrics 88

The progression mechanisms benchmarked in the previous chapter have shown issues when trying to get overlap. Efficiently progressing nonblocking collective communications remains a problem with existing mechanisms, especially without degrading performance on the application computation part. It needs to often use CPU time to execute collective communication algorithms, and it also needs reactivity as the progression required for the algorithm execution is spread along the communications.

We saw in 4.5.4 that dedicating a core itself does not always implies good performances. However, the use of appropriate collective implementations and a specific progression engine should allow the dedicated core to perform efficiently. In this chapter, we will study the effect of a dedicated core on the progression. Based on this, we propose an event-based implementation of collective algorithms, and a progression engine for dedicated core.

5.1 Using a dedicated core for asynchronous communication progression

To ensure overlap of communications with computation, HPC application programmers use MPI nonblocking primitives to let communication operations perform in background while the computation is running on the CPU. However, the MPI specification [63] only guarantees that these primitives will not block; it does not ensure that progression will actually happen in background.

The actual behaviour depends on the implementation of the MPI library, but generally, progression is poor as stated in chapter 4. If no explicit mechanism for progression is implemented in the MPI library, then progression only occurs inside the calls to the MPI library. Therefore, if a user calls `MPI_Isend`, then performs some computation, and finally calls `MPI_Wait` to check for completion, chances are high that communication will actually *begin* in the `MPI_Wait`, which jeopardizes any tentative of communication and computation overlap.

The idea of dedicating a core to the progression is to steal one core usually allocated for the application computation to restrict its use to the MPI runtime only. Thus, the runtime is always up to progress communications. The use of a core dedicated to communications is of interest since it is an application-independent mechanism. It is a background progression mechanisms as it does not require specific calls (e.g., `MPI_Test`) inside the application source code to progress communications.

5.1.1 Analysing core dedication relevance for progression

The section 4.5 showed that state-of-the-art progression mechanisms for nonblocking collectives often do not associate a good overlap and an optimised sharing of resources with the application threads. The polling have the tendency to slow the overall down due to the frequent need for resources used by application.

The introduction of an additional thread beside a multi-threaded runtime typically causes some performance degradation due to conflicts in thread scheduling as shown in section 4.5.3. Dedicating a core to this progress thread is an elegant solution to this issue. Dedicating a core resolve this resource sharing as it isolate each threads on different resources. However, the section 4.5.4 showed that dedicating a core is not a trivial solution. The implementation of optimised workers and a good management of available resources can lead to successfully associate good overlap and performance gain.

Resource need for collective operation overlap

It has been shown [24] that what hinders progression is not a large amount of work to run on a CPU, but instead very small tasks to be scheduled when needed. Thus, a progression mechanism does not require a lot of computation power at one time, but needs to execute multiple small tasks with reactivity.
Moreover, this mechanism requires being effectively isolated from potential threads created by the application. This imply to restrict the resource used by both MPI progression engine and application threads to non-intersecting resources.

Effect of a dedicated core on progression

Having a full dedicated core following these instructions ensures that the MPI library will be able to schedule these tasks at any time, whenever required, and should guarantee that progress is maximized. This solution ensure that the engine can always be up and execute available tasks with reactivity.
From the application point of view, the problem is that it is executed on pre-determined smaller amount of core. The hindrance between progress threads and application threads causes chaotic and consequent slowdown. As they originate from the scheduling policy of the system, they are hardly predictable. In contrast, the slowdown created by removing a core from the application is accurately predictable using scaling models and should remains moderate.

5.1.2 Implementation of dedicated cores

Although the "dedicated core" refers to the resource itself, the objective is to have a software worker using it to take care of the progression. This worker is in charge of executing the progression work. It must then take care of each MPI implementations characteristics concerning how the progression communication is handled. For example, for a task based dedicated core worker, the worker can perform active waiting to execute each newly available task with reactivity, as the resource is only used for this purpose.

A generic approach of dedicated core

The dedicated core is thus a generic concept which can be applied to every MPI implementation. One idea was to adapt one of the most common MPI implementation to use dedicated core. Different possibilities were available to work on, the major ones being MPICH and OpenMPI.

The considered solution was to rely on the progress threads used in those implementations to modify their binding to restrict it to the dedicated resource, and then tune the worker used on progress thread to run efficiently on a dedicated core. OpenMPI had an option to explicitly enable a progress thread, but this option was discarded in the 1.4 version. MPICH feature a progress thread that can be enabled using the environment variable `MPICH_ASYNC_PROGRESS`, however, there is no binding method for this thread and thus no easy way to restrict it to the dedicated resource. Moreover, the section 4.4.2 showed that this progress thread does not allow smooth progression.

To be able to reunite the best conditions to have an efficient dedicated core, we chose to rely on the Madmpi MPI implementation. This implementation is an interface to the NewMadeleine communication library which is closely integrated with the PIOman I/O manager. The use of this association of PIoman and NewMadeleine allows to accurately bind resources and benefit from the implementation of point-to-point operations used as a base for collectives.

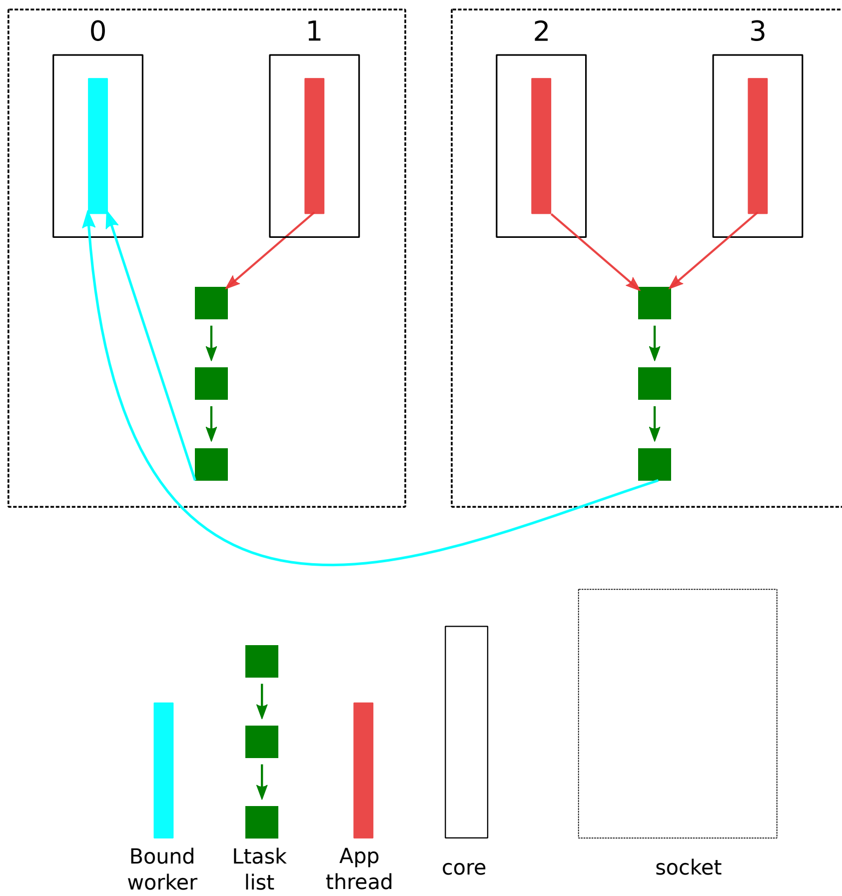


Figure 5.1: Pioman dedicated core and tasks lists on 4 cores processor

Use of tasks and task lists

PIOman relies on ltasks (user space tasklets equivalent) to organise the submitted work. Ltasks are chunk of code with their own context made to be executed in user space. When ltasks are available for execution they are put in lists to be executed by PIOman workers. Multiple lists are used for locally polling from workers if there is active waiting and for contention issues.
PIOman wait for various incoming events such as data transmission completion to make ltasks available.

A Pioman worker for dedicated core

The base principle of this dedicated core implementation is to be associated with the event-based algorithms we will present in section 5.2.2. Pioman is an event manager relying on tasklets to execute event handlers. The Pioman engine already feature different workers used to execute pending tasks. However, these workers were developed to respect constraints concerning their scheduling as they are likely to disturb other threads from the application on the same resource. Thus, the `timer worker` is designed to ensure a periodical polling every several milliseconds in order to avoid famine. The `idle worker` is configured with the `SCHED_IDLE` policy from pthread created for very low priority threads able to opportunistically poll tasks.
Using a dedicated resource remove these scheduling constraints as the resource we want to run our worker on is by definition dedicated for this purpose. However, the worker must be bound to the dedicated resources to prevent the scheduler to migrate it to another place. This new type of worker is then called a `bound worker`. Without any restriction in terms of scheduling, the worker remains very simple. The worker get the list of task associated with the topology, then the worker can poll tasks without restrictions. This allows this worker to be ready to execute newly submitted task with reactivity, unlike a `timer worker` which do it every several milliseconds.

5.1.3 Dedicated core cohabitation with OpenMP threads

As stated in last section, the bound worker is designed to use the resource it is placed on without restriction. However, with hybrid applications, other programming models such as OpenMP may schedule thread on the dedicated resource. Indeed, to actually have a dedicated resource, each programming model must be configured to avoid the resource chosen for dedication. To focus on the common example of MPI + OpenMP hybrid association, the OpenMP runtime must be configured to not impinge on the dedicated core space.

To do so, each OpenMP implementation propose its own binding threads facility. The different OpenMP implementations propose multiple non-standard solutions with a more easy to use conception than the standard one we will present here. The choice of using this standard solution have been made as it is applicable to every implementation.

The first thing we do to dedicate a core is to restrict the chosen resource from the OpenMP resource set. OpenMP define *places* where threads are located and executed. The environment variable `OMP_PLACES` allows the user to specify abstract names such as

“threads”, “cores” or “socket“ to set the granularity of places. It can be combined with a number in parentheses to specify the number of places. A number inferior to the total number of available core will then let unused resources usable for dedication. However, this solution does not let the user explicitly set the exact places to put the threads on. With a n cores processor, `OMP_PLACES=cores (n-1)` will create the good number of places, but the actual core not used is not explicitly defined. The solution which can be used is then the comma-separated list of places. For example, `OMP_PLACES = {1:n-1}` will ensure that the first core will not be used by OpenMP as it specify that places available are on core 1 to n.

It is also important to adapt the number of threads to create with one less core available for computation. Creating more threads than the number of places would let two threads on the same places. This will slow the application down as each thread will alternatively be scheduled with a non-negligible overhead. Applications usually split the work to do equitably between all threads. Thus, with two threads running on one core, the actual work to be done by this core may be twice the other in function of OpenMP parameters. This unbalance has an impact on the performance as all threads often wait in a barrier at then end of a parallel region. However, removing one thread will slow the application down in a more moderate way as the work will be split efficiently on each core. We can then set the number of thread using `OMP_NUM_THREADS= n-1`.

Finally, a good practice to have better performance in this configuration is to set the environment variable `OMP_PROC_BIND = true` to prevent threads to migrate between cores. Indeed, with a thread number equal to the number of places, most of the applications will benefit from sparing the migration cost and keeping each thread on one resource.

5.2 Implementing NBC in NewMadeleine

NewMadeleine, through its MPI interface MadMPI, proposes blocking and nonblocking implementation of MPI point-to-point operations. In this section, we present collective algorithms based on the NewMadeleine point-to-point operations. We will give the general ideas of the implementation of these algorithms and illustrate it with some examples.

5.2.1 NewMadeleine algorithms for nonblocking collectives

The NewMadeleine communication library relies on event handlers to manage the progression work. These handlers are triggered following driver notifications on the communication status (e.g. send or receive complete). This allows the mechanism in charge of the progression to react wherever it is on the system. The Pioman event manager can be used to manage these event handlers. Tasklets are then used to execute drivers code. Driver events are then sent to NewMadeleine using upcalls.

Pioman tasklet-based worker for event handling

The Pioman engine is conceived to schedule tasklets to opportunistically execute the handlers on the available resources. When paired with NewMadeleine, it can execute

the progression handlers to bring the reactivity and computation resources needed. As a consequence, it allows an efficient resource management and limit the progression impact on the application. Its event-based structure give a better handle with multiple threads in comparison to active waiting for each thread. The main purpose is thus to simplify the resource management by making this progress work doable by any of the available CPU core.

The aggressive polling policy used by this dedicated worker can however have an impact on the application by flooding the memory bus. This could have for effect to generate contention for the task submission. The Pioman list used for this purpose are actually split in two, one using spinlock for the polling, and the other using lock free mechanisms used for the submission. The submitted tasks are then transferred from one queue to the other when the lock is held. This ensures to minimise the potential contention between them.

Event-based collective algorithms

The collective progression work consist of multiple pre-determined point-to-point operations. This set of operations achieve the expected collective purpose with optimisation. Following this idea, the collective algorithm need to schedule the multiple point-to-point operations following specific events. This must take into account the state of the collective between these operations (e.g, A rank must wait to receive a value before sending it to the next one in the algorithm). The communication scheme is known since the initialisation of the collective, meaning that every point-to-point operation at every step is defined from the beginning of the collective. We can then define the algorithm as a sequence of states transformed through events. For example, in a tree-based broadcast, to respect the required state defined by the algorithm, let us say that the rank y will have to send the value to rank z, but also receive the value from the rank x. The collective algorithm must ensure that the operation $xsendtoy$ is done before $ysendtoz$.

To respect these constraints, the collective algorithm is divided in multiple steps. Each step include the operations ready to be performed. The events corresponding to the end of receptions or sending are triggered to assess that all these operations are done. When all the events needed for one step are triggered, they make new operations available. These newly available operations are included in the next step to be executed.

To be able to run the available handlers, it is necessary to keep track of the context of the collective. To be executed anywhere on the system, handlers include a context of the current state of the collective, the rank it is associated with, and other information required. MPI use request for nonblocking communications which is wrapped in a bigger NewMadeleine object including the current state of the collective.

Similar to the functioning of other nonblocking implementation, the MPI calls have the purpose of initialising the necessary structures for the collective. The previous cited request is initialised in this function. After this initialisation the MPI call can start the collective algorithm by scheduling the first step. The progression engine then have to execute newly available handlers and run the next step in function of the collective state.

Name	*Description*
MPI_MAX	*maximum*
MPI_MIN	*minimum*
MPI_SUM	*sum*
MPI_PROD	*product*
MPI_LAND	*logical and*
MPI_BAND	$bit-wise\ and$
MPI_LOR	*logical or*
MPI_BOR	$bit-wise\ or$
MPI_LXOR	*logical xor*
MPI_BXOR	$bit-wise\ xor$
MPI_MAXLOC	*max value and location*
MPI_MINLOC	*min value and location*

Table 5.1: MPI operators with their descriptions

To summarise, the collective algorithm have the role of splitting the necessary communications in handlers. It ensures that all these handlers are executed in a good order to perform the purpose of the collective. These handlers can be executed by an engine composed by one or multiple execution units using tasklets. The split of the total workload among handlers then allows all the resources involved to execute them wherever they are in the system.

5.2.2 Implementation of collectives in NewMadeleine

Collectives feature different communication schemes that can be classified as the following: Some collectives called '$Nto1$ will communicate from the N ranks involved in the collective to the root. This is the case for `MPI_Ireduce` for example. On the contrary, Some collective such as `MPI_Ibcast` will communicate data from the root to the n ranks ($1toN$). Finally, the last kind of collective move data from the N nodes to the N nodes. This is the case for the `MPI_Ialltoall`.

Collective algorithms greatly differ to achieve these 3 type of collectives. $1toN$ or$Nto1$ collectives will usually use tree-based algorithms to efficiently spread or gather data from the root. N to N collectives are typically trickier to optimise as sending from N to N implies that all ranks will have work to do during the entire duration of the collective. Some of these, such as `MPI_Allgather` and `MPI_Allreduce` use step-based algorithm(reduce followed by broadcast or recursive halving). The `MPI_Ialltoall` can introduce a lot of simultaneous communications. We give the implementation for two examples of algorithms respectively in the next sections.

`MPI_Ireduce` Implementation

To illustrate the general process of these N to 1 algorithms, we will detail in this section the algorithm of `MPI_Ireduce`. The reduce collective take values from all ranks involved

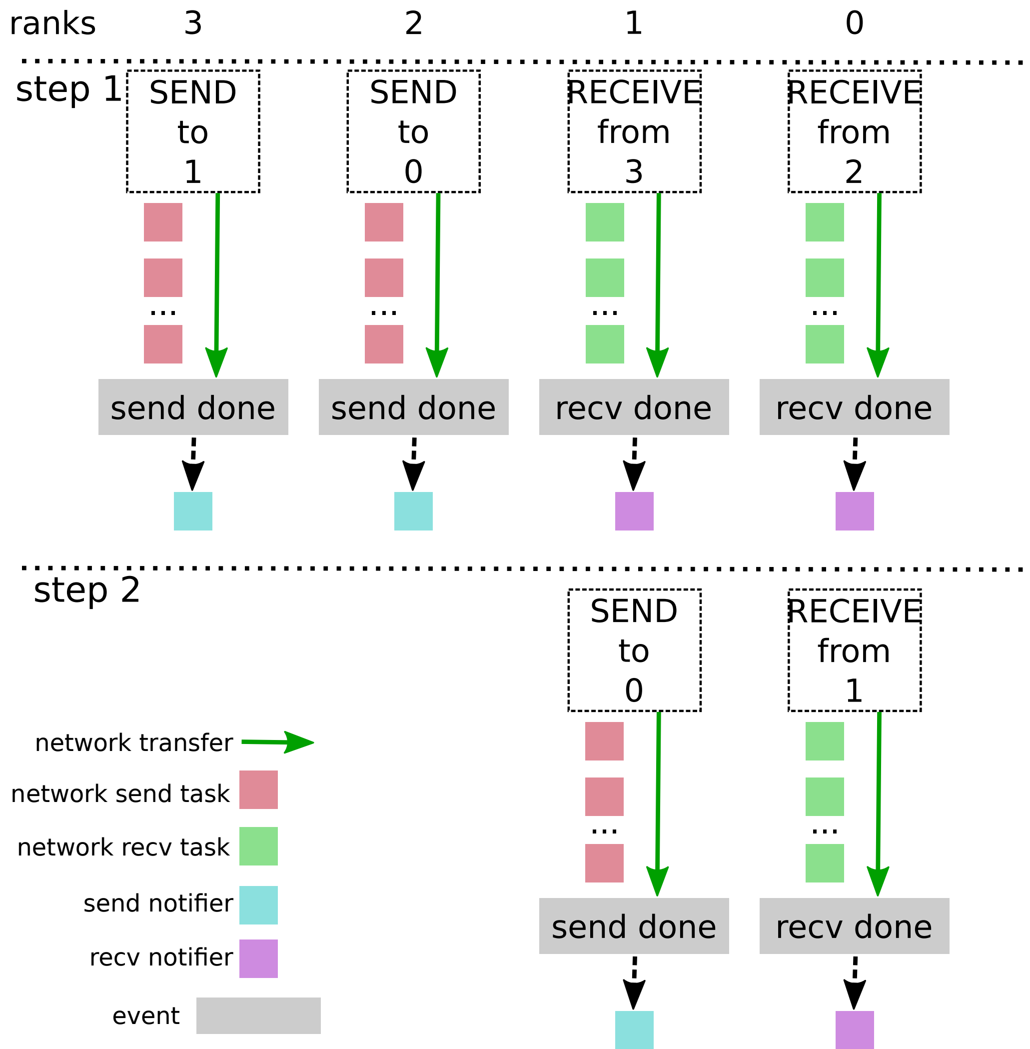


Figure 5.2: Task creation and order of an `MPI_Ireduce`

in the collective, and merge them on the root rank using a predefined operation chosen by user such as sum, product... (table 5.1)

This ireduce implementation is developed from a tree-based algorithm similar to the one used for broadcast. In a reduce however, the tree will be browsed backward i.e, from the leaf to the root, as data is gathered on the root instead of being spread.

The figure 5.2 depicts the two steps required to perform the ireduce using tasks. The initialisation done in `MPI_Ireduce` allocate the structures needed for the collective such as the request containing the global context of the collective. The first step can then be set up by initiating the tree-based corresponding point-to-point communications. The communication is then handle by Pioman which split the total data in tasks. Depending on the amount of data to send, multiple tasks will be created to send (red square) and receive (green square) the packets.
The completion of the communication then generates an event for both receive and send side. This event schedules a send (light blue square) or receive (purple square) notifier task to handle the communication termination. For the reduce the receive notifier is in charge of performing the local reduce operation between the received and the current process value. The last notifier to be executed also has the charge to set up the next step.

Network intensive collective management

Some collectives are not naturally step-based, This type of collectives such as Allgather, Allreduce and Alltoall are designed to communicate data from every nodes to every nodes (N to N collectives). This type of communication are usually much more network intensive than 1 to N as every node can be considered as the root. The nonblocking versions of these collective can initiate all the ready communication at the initialisation of the collective and try to perform then at the same time. Moreover, some of these such as IAlltoall have no data dependencies to regulate the readiness of the needed point-to-point communications: from the initialisation every node is able to communicate the value to every other node, in contrast to a tree-based communication, such as reduce, where values must be received and reduced before being sent to the tree child. This amount of ready communication can then cause a communication burst, especially with a great number of process. This burst can then greatly degrade the network performance and slow the collective execution.
The implementation of `MPI_Ialltoall` in MadMPI restricts the number of submission of send for each process at the same time. At the opposite all receive are already pre posted at the initialisation of the collective. In this way, the total point-to-point communications needed are split between multiple step. When one send is completed, the send notifier can take a pending communication to continue the progress of the collective.

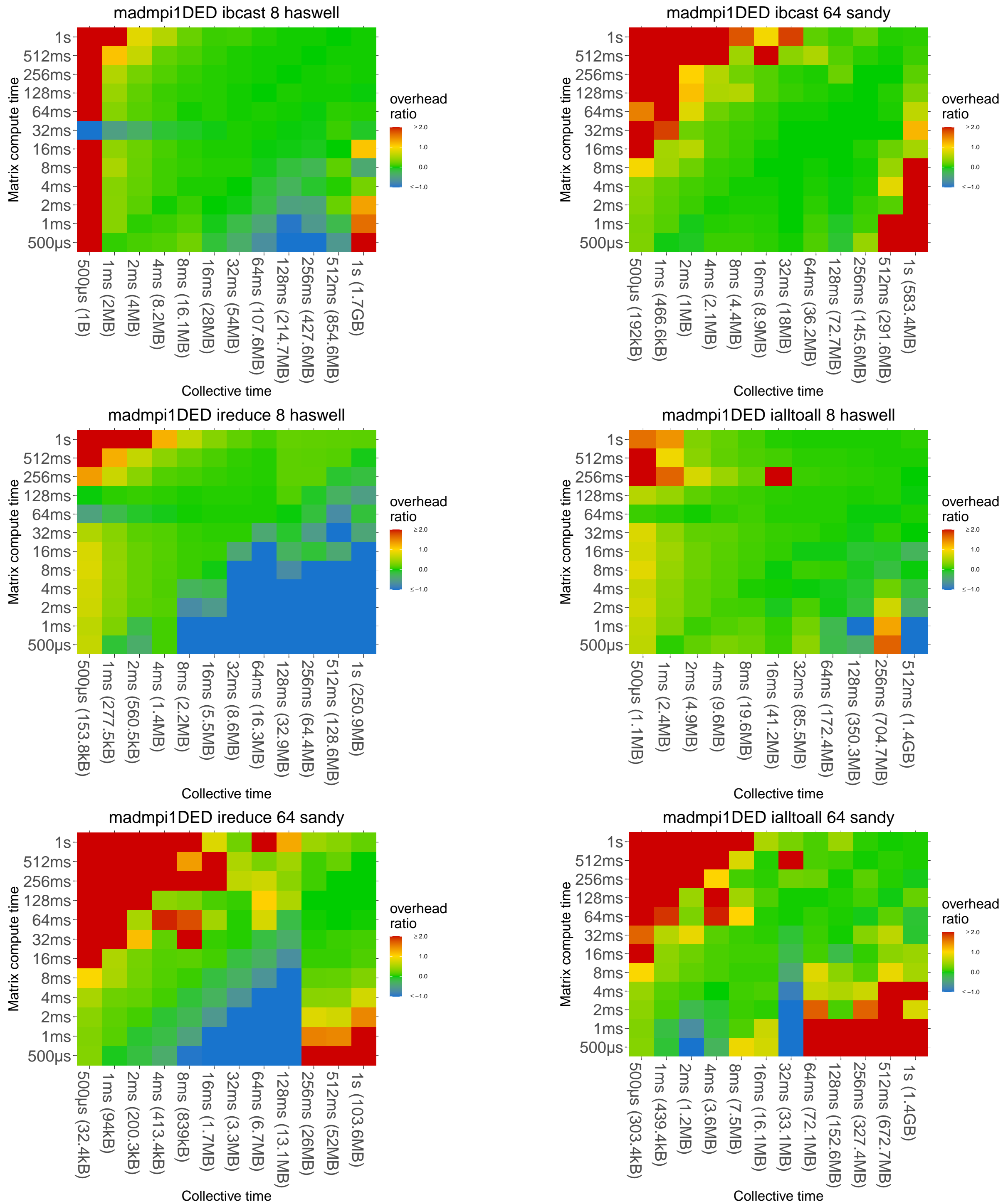


Figure 5.3: MadMPI overlap with dedicated core

5.3 Pioman dedicated core benchmarking

5.3.1 MadMPI dedicated core performance using BenchNBC

We enabled the dedicated core configuration in MadMPI and bound the worker on a free core. Overhead ratio of this configuration is shown in Figure 5.3. We observe that MadMPI with a dedicated core manages to overlap perfectly communications with computation, for any computation/communication sizes with every collective on 8 Haswell nodes. The efficient overlap also scales up to the tested 64 nodes, though the computation and communication times need to be closer than on 8 nodes. This can be observed as upper left and bottom right corners of each graph with 64 nodes tends to have an overhead ratio superior to 2. This can be explained as order of magnitude between computation and communication make the measure imprecise. This will eventually slow the overall collective down and penalise where the overlap conditions are not tasks. Figure 4.10 shows $r_{\text{MPI_impact}}$, the impact of MadMPI on computation (left 2 bars), with the dedicated core and without the dedicated core. The default configuration presented in section 4.5.3 has little impact on computation but the configuration with dedicated core reduces this impact, which is not surprising considering that the default uses the same cores as computation, whereas having a dedicated core should avoid any interaction.

5.3.2 Analysis of collective behaviour using concurrency and all ranks metrics

The global performance of a dedicated core progressing collectives are satisfying, but the metrics such as communication and computation concurrency ratios and the all rank results can give a clue about the remaining obstacles for a perfect overlap.

Ibcast

The different metrics for the broadcast represent an archetype of this perfect overlap. The figures 5.4 and 5.5 which represent respectively the execution on 8 Haswell cores and on 64 Sandy Bridge cores shows very little differences. This shows that the progression mechanism is able to scale on numerous nodes.
The communication concurrency ratio is split in two parts, the upper left part shows that with enough computation, the collective time is totally hidden. The lower right part tends to have a sequential communication. This can be easily explained as this part represent when the experiment does not have enough computation to overlap all the communication part. Then the less computation is available the more the progression is done by the `MPI_Wait`.
Finally, the all rank view, show a perfectly flat shape, except for the 500 microseconds experiment.
The communication concurrency ratio for the entire 500 microseconds shows that the `MPI_Wait` is poorly reduced with overlap. The combination of very small communication and computation time is highly impacted by variations in measure. With this kind of value, the dedicated core can sometimes lack of reactivity making it hard to perfectly overlap for very small executions All these slowed down values can be then be explained

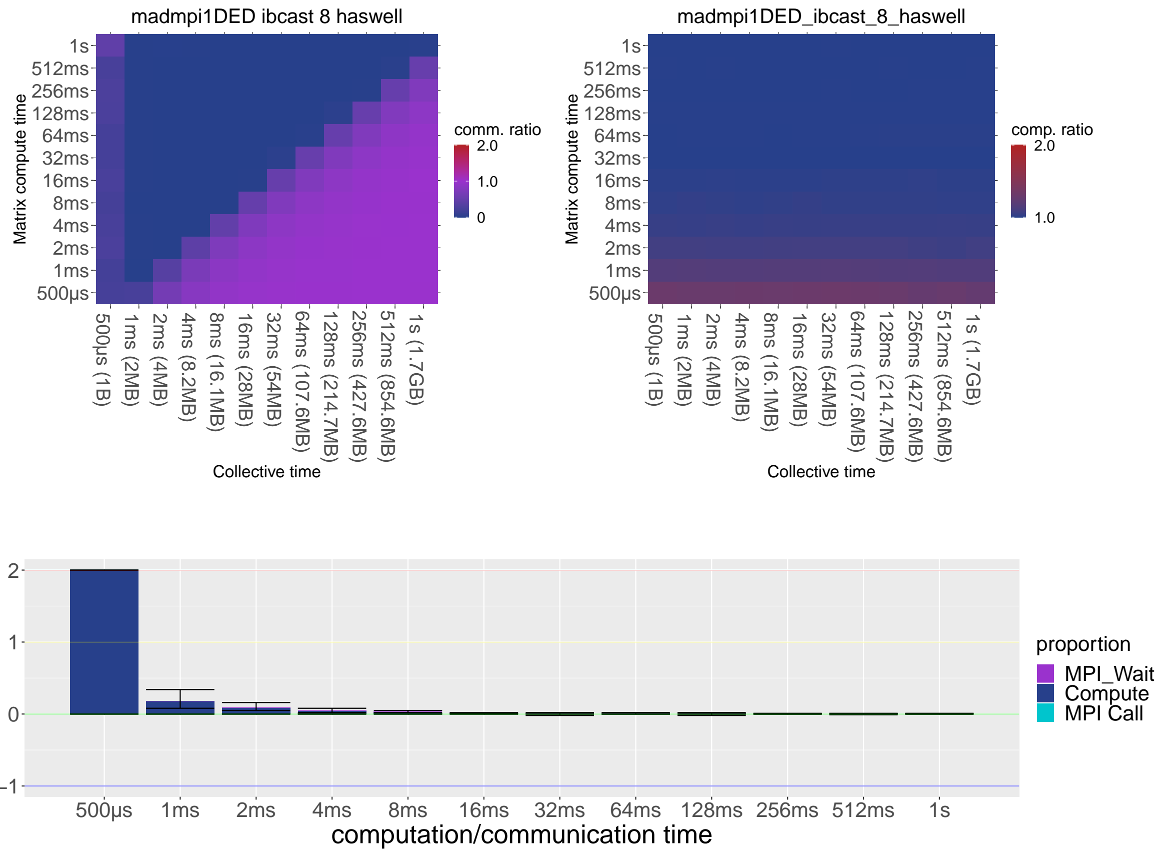


Figure 5.4: Concurrency ratios for `MPI_Ibcast` on 8 Haswell nodes

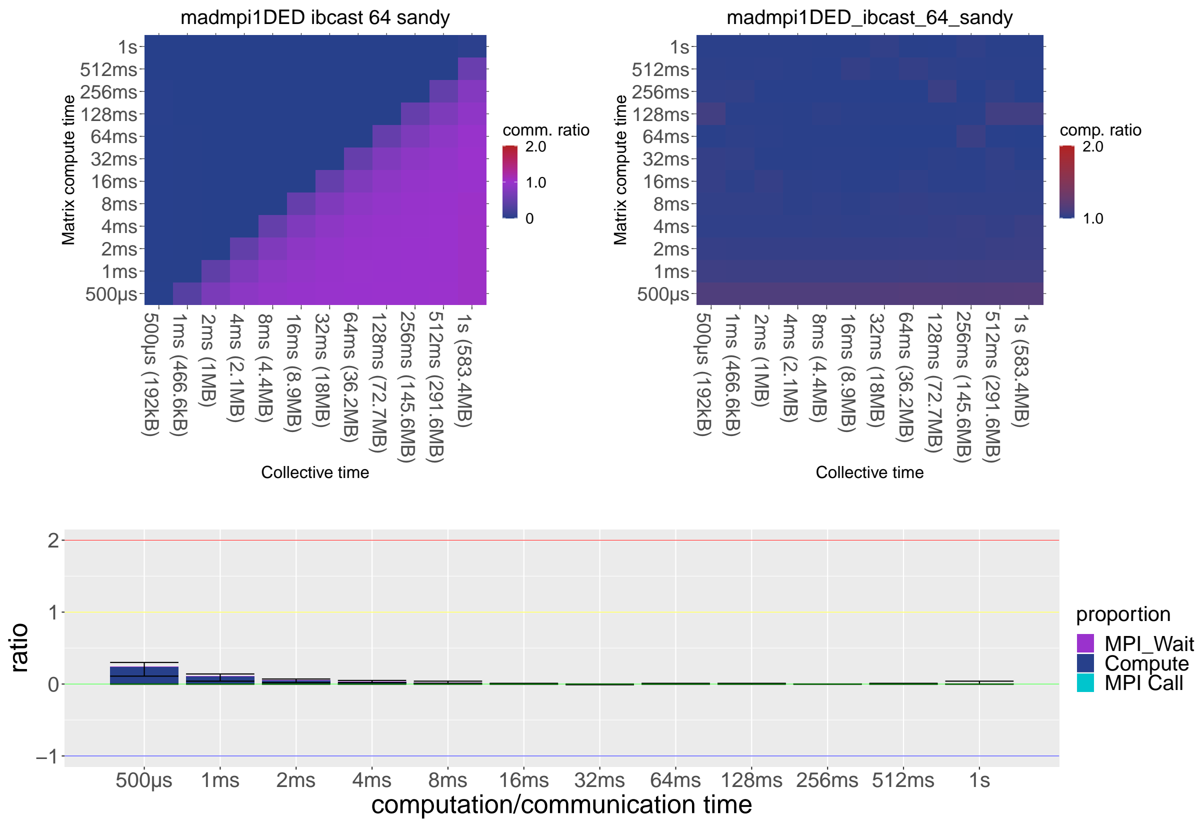


Figure 5.5: Concurrency ratios for `MPI_Ibcast` on 64 Sandy Bridge nodes

by the combination of variation and the measure of very small cases. The flat boxes show that every rank have a perfect overlap as minimum, maximum and median rank overhead ratio are merged at value 0.

Ireduce

The ireduce collective show a global equivalent for both set of figures 5.6 and 5.7. The communication concurrency ratio for both figures shows that the ratio is split diagonally for the same reasons as for the broadcast. However, we can also see a frontier from 64 microseconds to 128 microseconds for the 8 nodes execution and from 128 microseconds to 256 microseconds for the 64 nodes execution. This can be explained as the MPI reduce include a reduce operation. It is likely due to the operation execution hindering the computation on these cases.
The all rank view show a very good but not perfect overlap, there is difference between the two executions as the maximum overhead ratio is significantly higher for multiple values. This shows that the reduce is more affected by the number of ranks involved in the collective.

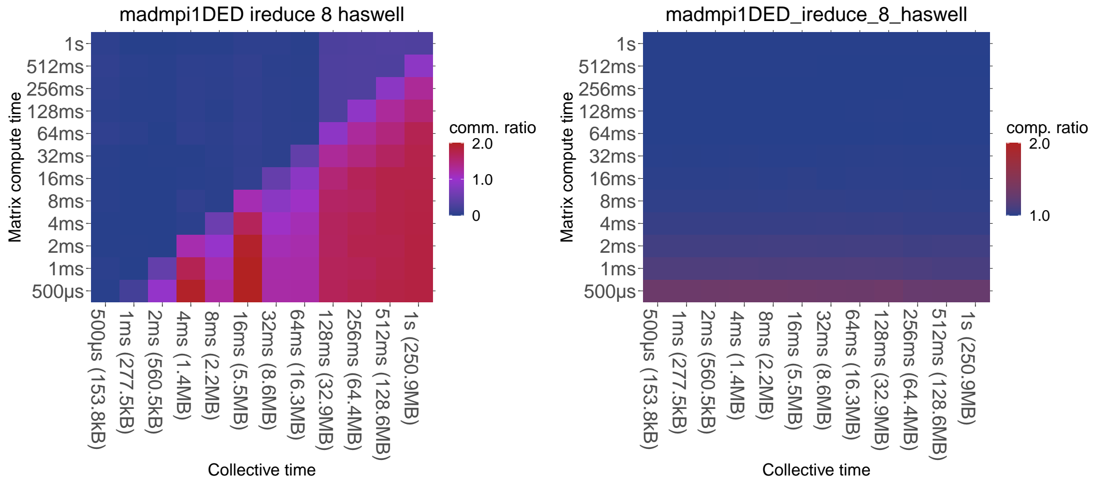
Ialltoall

The section 5.2.2 showed that collective such as Ialltoall are based on a "NtoN" communication pattern. These collectives are then more likely to be impacted by scalability problems. The figures 5.8 and 5.9 show some differences especially for the all ranks graphs. For the 8 nodes results, the global behaviour present near zero overhead ratios and is very similar to the 8 node execution of the broadcast. The 64 nodes results introduce close to 0 but negative ratios. With a great number of nodes involved in the collective, the `MPI_Ialltoall` require achieving a lot of communication. This make it more sensible to variations between all executions. At the end, the time get from each execution of the collective can be different. If the variation imply a shorter time than the reference time, a good overlapped experiment must be faster than the reference and give these negative ratios.

Global overview of the impact of node count on progression

A good thing to note is that most of the NBC have actually a modified behaviour as node and MPI process count implied in the collective grow. The more nodes are implied in the collective, the more complex the execution will be. This has a double effect on the interpretation of these results. First, the multiplication of nodes have an effect on the measure itself. As node number increases, the global communication count needed grows up. The impact on the measure can be seen on the global overview of figure 5.9. The benchmark in this situation more likely get an error delta on the metric calculation and can give these negative ratios.

Another effect is that collective tends to be harder to overlap when the node count increases. Indeed, the benchmark is conceived to calculate the global time of the collective based on the maximum value of all MPI processes. Following this, the chances of having scheduling events or more generally system noise increase. This may eventually delay the



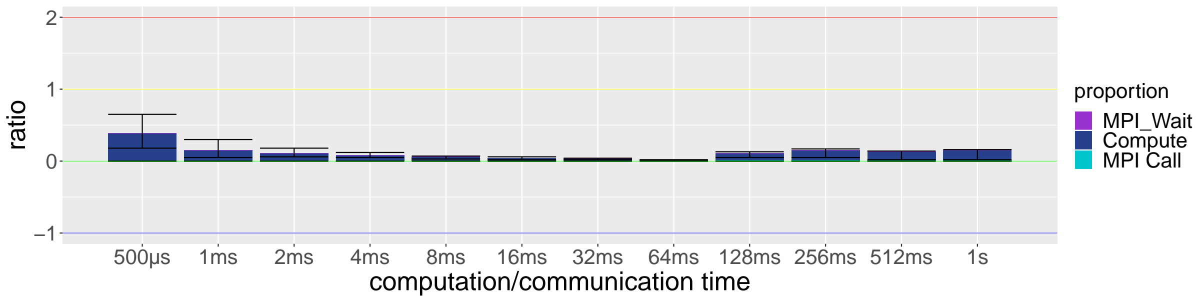
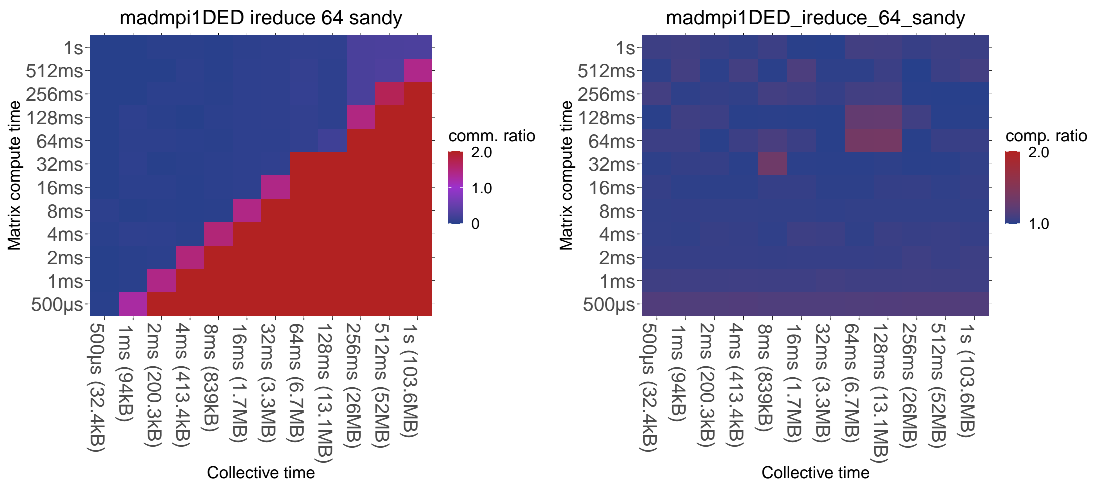


Figure 5.6: Concurrency ratios for `MPI_Ireduce` on 8 Haswell nodes



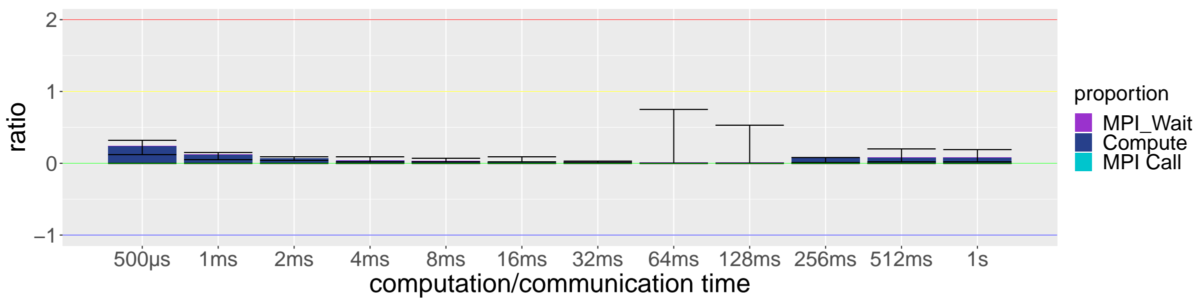


Figure 5.7: Concurrency ratios for `MPI_Ireduce` on 64 Sandy Bridge nodes

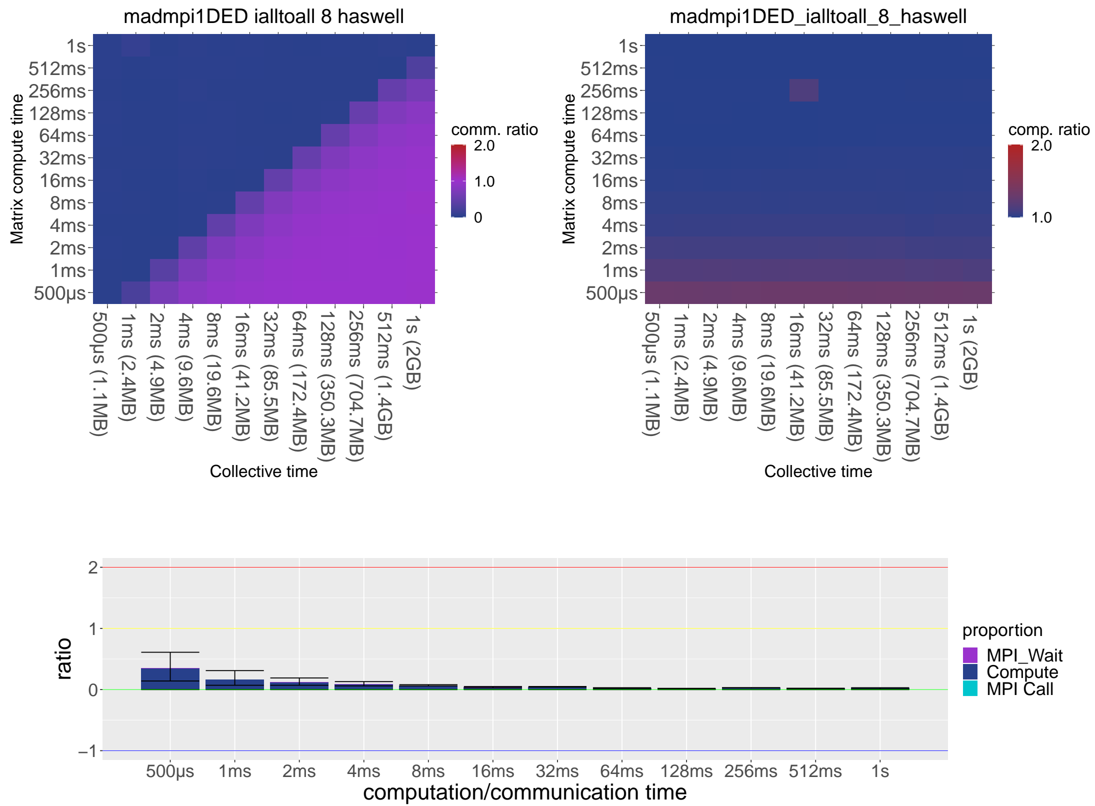


Figure 5.8: Concurrency ratios for `MPI_Ialltoall` on 8 Haswell nodes

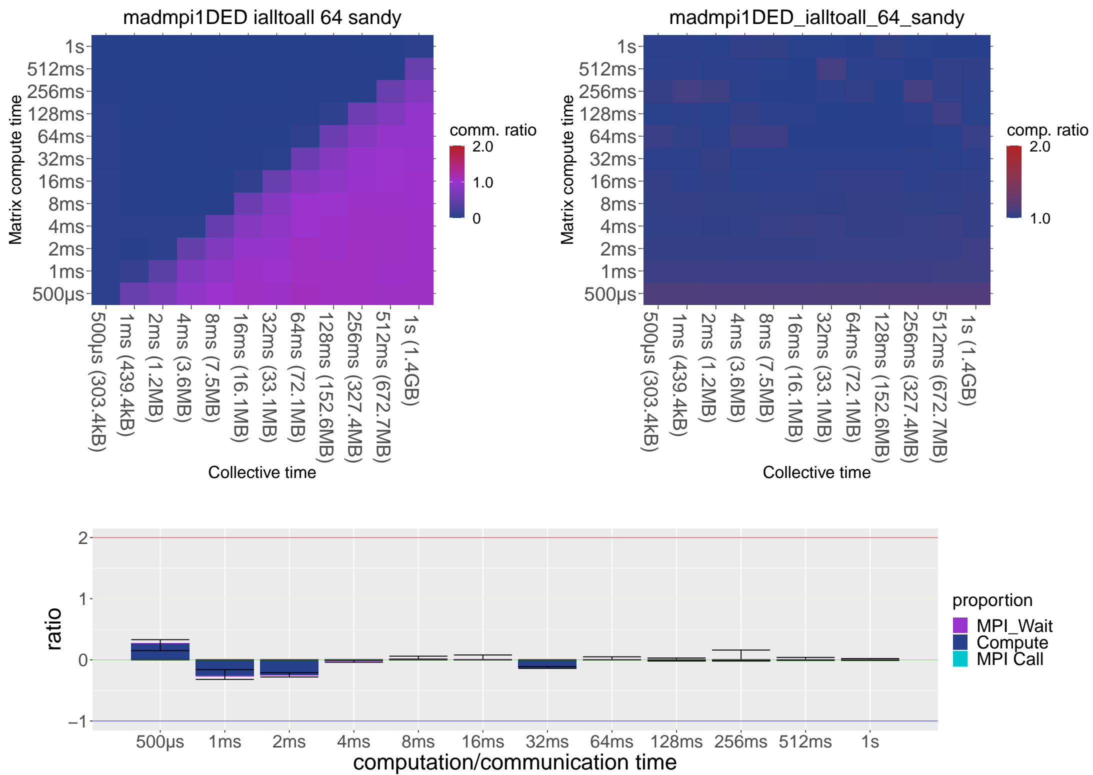


Figure 5.9: Concurrency ratios for `MPI_Ialltoall` on 64 Sandy Bridge nodes

progression of one node. Thus, one node being late will give higher maximum and poorer results, as observed on the global overview on figure 5.7.

Effect of unbalanced computation and communication on the measure

This benchmark implementation experiment cases where computation and communication are unbalanced. Looking at all the heatmaps presented in this document, the upper left and lower right corner of each heatmap introduce the most unbalanced cases. One thing to note is that these cases are the most difficult to interpret. As the order of size become more different, the variation in measure become more impacting. For example, trying to overlap 1 second of computation with 500 microseconds of communication will have an incertitude on both time. The variation on the second of computation can however be around 500 microseconds or even more. In this situation the variation can totally override the communication time making the metrics go on very high or very low ratio.

If the precise interpretation of this case is difficult, keeping this results and related ones with similar sizes remains useful. The global look at these extreme values gives a clue on the capacity of the runtime to remains stable with extreme cases.

The dedicated core is a reliable way to progress communication in the purpose of getting overlap. The evaluation of its performance was based on the benchmark introduced in chapter 4. Even when trying to be as close as possible from real application, trying to get overlap remains difficult. Using an efficient progression mechanism is just the half of the whole problem, as application may not be conceived to permit overlap.
Using a dedicated core in those situations where no overlap is possible is not free. As it has to steal resources from applications, knowing when to enable it is very important to ensure the best performances in all cases.

Chapter 6

A model for application performance with a dedicated core for communication progression

Contents

6.1 Modelling dedicated core impact on hybrid applications . . . 96

6.1.1 Overview of the performance model 96

6.1.2 Impact of a dedicated core on computation 97

6.1.3 Modeling MPI performance with dedicated core 100

6.1.4 Global model with MPI and computation 103

6.2 Gathering applications information to use the model 104

6.3 Evaluation of the model 104

6.4 Using the model 109

6.4.1 Understanding the computation-slowdown/communication-speedup ratio 109

6.4.2 Evaluating the effectiveness of nonblocking communication . . . 110

6.4.3 Potential gain of transforming blocking call to nonblocking . . . 111

Modelling is a powerful tool to predict the performance of applications. The previous chapter showed the efficiency of stealing a core from the application to progress the communications. However, the missing core from the application reduce the computation performances. To be valuable, the overlap of computation and communication must be greater than the computation performance loss. Based on this, we can design a model, evaluating the behaviour of both the communication and computation parts, to predict if it is worth it to use a dedicated core for the application communication progression. Hybrid MPI + OpenMP applications are composed of OpenMP parallel computation parts and MPI communication parts that need to be merged to execute independent computation during the communications. The applications used in HPC thus propose multiple profiles considering the duration of both communication and computation part in these potential overlap situations, the number of these overlap situations and their proportion in relation

to the total time of the application. These different profile of application can have very different reactions to a dedicated core, from winning time to degrading the performances. In this this chapter, we will design such model. Then, we will describe how to gather the information needed to use the model, and for which purposes the model can be useful.

6.1 Modelling dedicated core impact on hybrid applications

In this section, we will design our model. First, we will give a general overview of the model, then we will detail how the model is built for both computation and communication parts.

6.1.1 Overview of the performance model

To sort out cases where overlap with dedicated core is beneficial from cases where it is detrimental to performance, we propose a model to predict performance. This model takes into account two phenomena:

- the impact on computation performance of having one less core because the core dedicated to communications is not available for computation anymore. We will study this aspect in section 6.1.2;
- the impact on communication performance and on overlap of communication and computation of having a dedicated core. It will be described in section 6.1.3.

The first part about computation is expected to be a performance degradation. The second part, about overlap, is expected to be a performance improvement; it may be negligible if communication may not be overlapable, or if communications constitute a small part of the total application execution time. Then, the computation slowdown may overcome the communication speedup. For this reason, the dedicated core cannot just be a “enabled and forget” feature. The actual challenge is to know whether the degradation is compensated by the gain. To be used in the relevant situations, we must be able to predict its behaviour.

Unfortunately, the behaviour of HPC applications diverges greatly with regard to computation and communication. Even on the same application, different inputs (parameters, data...) may lead to different performance behaviour. Worse, two set of parameters can seem to be similar, showing very close computation time when run without dedicated core. However, with the use of dedicated core, only one set of parameters will give a significant time gain. This is due to the input parameters, which may drastically change communications/computation scheme and proportions.

To cover these cases and be oblivious to input parameters, we will design our model to be *independent of the application*. To do so, our model decomposes an application as a sum of specialized parts, each with a different behaviour when a dedicated core for communication progression is involved. These parts will be presented in section 6.1.2 and section 6.1.3.

Thanks to our model, we aim at predicting if the use of a dedicated core for progression is beneficial for an application. To apply our model, we will first run the application on all cores without a progression thread to get a reference time $t_{\text{noprogress}}$. Details on how the exact timings are measured on the different MPI ranks is given in Sec. 6.2. This $t_{\text{noprogress}}$ is composed of:

- t_{comp} the time spent by the application in computation;
- t_{MPI} the time spent in the MPI library. This time is itself composed of $t_{\text{MPIoverlapable}}$ for communications that may benefit from overlap (nonblocking point-to-point and collective primitives) and progress in background, and $t_{\text{MPInotoverlapable}}$ for the remaining communications and all MPI management time.

This distinction will help us model and analyse precisely the behaviour of applications with and without dedicated core. It is clear that the expected speedup from the use of a dedicated core will be limited to the time spent in the MPI library.

Let $t_{\text{dedicated}}$ the execution time of the application with a dedicated core for communications. It will diverge from $t_{\text{noprogress}}$ as follows:

- on the t_{comp} part, computation will be slowed down by an overhead t_{overhead};
- on the t_{MPI} part, we expect the nonblocking communication primitives to be fully overlapped and thus the gain should be $t_{\text{MPIoverlapable}}$.

Thus, the general formulation for our model is:

$$t_{\text{dedicated}} = t_{\text{noprogress}} + t_{\text{overhead}} - t_{\text{MPIoverlapable}}$$

This formulation helps to get an intuitive idea of the model. However, instead of computing t_{overhead} and $t_{\text{MPIoverlapable}}$, in the later sections we will decompose t_{comp} and t_{MPI}.

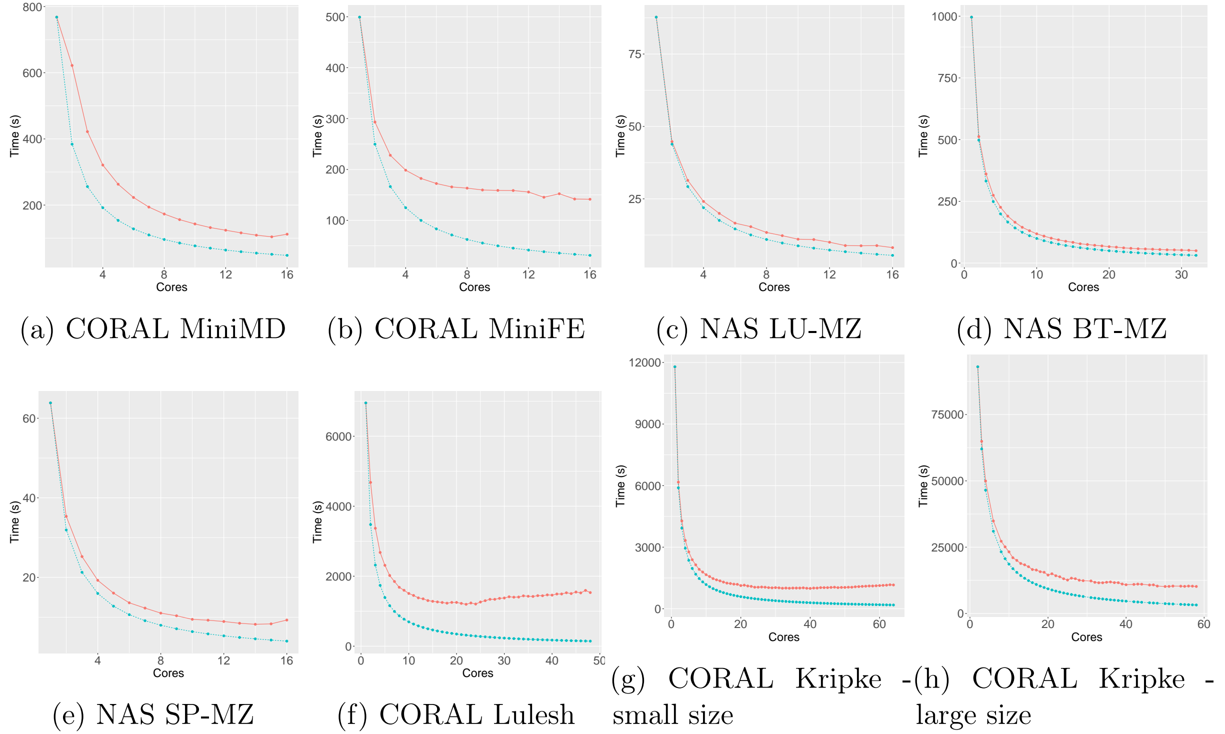
6.1.2 Impact of a dedicated core on computation

The evolution of modern processors exhibits an increase in the number of cores. With more and more cores, it becomes more difficult to optimize applications and fully exploit those CPUs. Thus stealing a core to an application becomes less impacting for computation performances.

Study of OpenMP applications scalability

To quantify the loss caused by stealing a core to the application, we ran a study on the scalability of OpenMP threading for several well-known benchmarks :

- from the NAS Parallel benchmarks [64]:
  - BT-MZ: a Block Tri-diagonal solver app;
  - LU-MZ: a Lower-Upper Gauss-Seidel solver app;
  - SP-MZ: a Scalar Penta-diagonal solver app;
- and from the CORAL Benchmarks:



(a) CORAL MiniMD (b) CORAL MiniFE (c) NAS LU-MZ (d) NAS BT-MZ

(e) NAS SP-MZ (f) CORAL Lulesh (g) CORAL Kripke - small size (h) CORAL Kripke - large size

Figure 6.1: Study of OpenMP scalability on several well-known benchmarks (red plain line) against theoretical linear scaling (blue dotted line)

- MiniMD: a simple parallel molecular dynamics (MD) code [65];
- MiniFE: a proxy application for unstructured implicit finite element codes [65];
- Lulesh: the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics [66];
- Kripke: a simple scalable 3D Sn deterministic particle transport code [67].

We tested those benchmarks on several test platforms:

- *inti/sandy-bridge*: 234 nodes, with dual-socket Intel Xeon E5-2680, each with 8 cores at 2.7 GHz, equipped with Mellanox QDR Infiniband (used for the LU-MZ, SP-MZ, MiniFE and MiniMD benchmarks),

- *inti/haswell*: 8 nodes, with dual-socket Intel Xeon E5-2698, each with 16 cores at 2.3 GHz, equipped with Mellanox MT27600 (Connect-IB) InfiniBand boards (used for BT-MZ benchmark),

- *inti/skylake*: 32 nodes, with dual-socket Intel Xeon Platinum 8168, each with 24 cores at 2.7 GHz, equipped with Mellanox MT27700 (Connect-IB) InfiniBand boards (used for the Lulesh benchmark),

- *inti/KNL*: 24 nodes, with Intel Xeon Phi 7250, each with 68 cores at 1.4 GHz, equipped with Mellanox EDR InfiniBand boards (used for the Kripke benchmark).

OpenMP scalability results for all benchmarks are displayed in figure 6.1. From these graphs, we observe two types of behaviour. On the one hand, benchmarks on the top

row show the expected behaviour : the execution time follows a $1/(\delta \times N_{\text{core}})$ slope (with δ often very close to 1). On the other hand, the first three benchmarks of the bottom row show bad OpenMP scalability, as they all lose performance when the number of threads is greater than a given threshold (specific to each application). We can see on some benchmarks (LU-MZ and Lulesh) that the last point of the curve shows a greater speedup than the previous points. However, since those points are outliers, we consider they do not impact the general slope of the curve.

One can observe that the Kripke benchmark displays the behaviour we described in section 6.1.1: two set of input parameters cause different OpenMP scalability. When executed with small sizes (i.e. x=16, y=8 and z=8, in figure 6.1g), the bad OpenMP scalability causes a slowdown instead of a speedup when large number of threads are used. However, when executed with large sizes (i.e. x=400, y=200, z=200, in figure 6.1h), we observe speedup even with a large number of threads, and the scalability follows the $1/N_{\text{core}}$ slope.

We consider that taking a core from the parallel computation will cause slowdown induced by the loss of computational power. As we have seen, if the OpenMP scalability is not good, taking a core from the computation will not cause a slowdown, but a speedup! Hence, since the impact on computation time is supposed to hinder the speedup gained thanks to the communication progression, we will consider the worst-case scenario for our model : good OpenMP scalability. This case is the one actually producing a slowdown which may prevent a dedicated core for progression to be effective.

As we have seen, most applications with a good OpenMP scalability exposes a $1/(\delta \times N_{\text{core}})$ slope (with δ very close to 1). Hence, in our model, we will approximate the computation slowdown induced by the removal of one computational core with the linear equation $1/N_{\text{core}}$.

Modeling manycore cpu impact on thread scalability

The overhead expectation t_{overhead} due to the core stealing impact can be modelled using the computation time (t_{comp}) from no progression execution. With our approximation, we estimate that the new computation time when removing N_{dedcore} cores among N_{core} is $t_{\text{newcomp}} = t_{\text{comp}} \times \frac{N_{\text{core}}}{N_{\text{core}} - N_{\text{dedcore}}}$. Specifically, it means that when taking one dedicated core for progression, the expected computation time is $t_{\text{newcomp}} = t_{\text{comp}} \times \frac{N_{\text{core}}}{N_{\text{core}} - 1}$. The induced slowdown is then computed with $t_{\text{overhead}} = t_{\text{newcomp}} - t_{\text{comp}}$. This equation can be simplified to $t_{\text{overhead}} = \frac{t_{\text{comp}}}{N_{\text{core}} - 1}$. For a 16-core node, taking a computational core will cause a $\frac{t_{\text{comp}}}{15}$ slowdown, and on a 64-core node, the slowdown will be $\frac{t_{\text{comp}}}{63}$. Thus, the more core a processor has, the more negligible the slowdown will be.

Link to the measures and metrics in chapter 4

The model for computation is thus constructed using the t_{comp} get from the execution. The chapter 4 introduced a timing for computation called t_{comp} that also represent computation. However, theses two timings do not represent the same computation part: t_{comp} represent the computation part to be overlapped and placed between MPI call and MPI Wait. Thus, t_{comp} is a sub part of t_{comp} and is interesting for modelling MPI part as the

amount of computation between initialisation and wait impact the overlap capacity. This timing is however hard to measure as it have sense only from the MPI runtime point of view: i.e, this is a computation part defined by its both MPI calls bounds, but is also code executed outside the runtime. An envisaged solution would be to instrument the runtime to calculate this time from its two bounds.

The section 4.1.2 presented a concurrency ratio designed to measure the direct impact from the progression mechanism on the benchmark parallel computation. For dedicated core, this impact is minimal as resources are not shared and each core is not disturbed. However, the real slowdown caused by the dedicated core is the removal of one core to the application. This slowdown has the advantage of being much easier to model and predict than a slowdown due to threads interactions, as these interaction prediction would require to simulate the scheduling of each thread, the effects on caches and more.

6.1.3 Modeling MPI performance with dedicated core

The second part of the model estimates the decrease of the time spent in MPI functions thanks to a dedicated core.

An MPI distributed application usually includes different MPI calls. These MPI calls will not display the same behaviour when a core dedicated for background progression is used. To build an accurate model for the communication part when using a dedicated core, we need to classify the MPI functions regarding their response to the use of said dedicated core.

Classification of MPI calls

For our model we distinguish four types of MPI calls:

- blocking communication calls
- nonblocking communication initialisation calls
- nonblocking communication completion calls (e.g. `MPI_Test`, `MPI_Wait`)
- other MPI calls (e.g. runtime initialisation, communicators management, datatypes management, etc.)

With this classification, we decompose the total time spent in MPI in the following categories:

$$t_{\text{MPI}} = t_{\text{MPIblocking}} + t_{\text{MPInonblocking}} + t_{\text{MPItest}} + t_{\text{MPIwait}} + t_{\text{MPIother}}$$

with the following definitions:

- $t_{\text{MPIblocking}}$ is the time spent in blocking communication calls;
- $t_{\text{MPInonblocking}}$ is the time spent in nonblocking initialisation calls such as `MPI_Isend`, `MPI_Irecv` or `MPI_Iallgather`;

- $t_{\mathrm{MPItest}} + t_{\mathrm{MPIwait}}$, the time spent in active progression functions such as `MPI_Test` and `MPI_Wait`;
- t_{MPIother} is the remaining time spent in the MPI runtime (e.g. initialisation, communicators management, datatypes management, etc.).

In the following sections, we will describe how each category of MPI call will behave when a dedicated core is used for background progression.

Impact of dedicated core on each MPI call category

Impact on nonblocking initialisation and completion calls As we presented in section 5.1, when the MPI library implements no progression mechanisms, then nonblocking communications only progress inside MPI calls. These calls can be nonblocking initialisation calls (e.g. `MPI_Isend`, `MPI_Irecv`, `MPI_Iallgather`), completion calls (e.g. `MPI_Test`, `MPI_Wait`), or any MPI calls involved in communications (even blocking primitives).

With a dedicated core, nonblocking communications are expected to progress on said dedicated core, thus not consuming computational power from the other cores. However, even if the communication itself takes place on the dedicated core, the time taken by the actual involved MPI calls is not completely nullified: the requests still have to be initialized and the associated operation registered in the MPI runtime for initialisation calls, ; the status of the request has to be fetched, with the relevant synchronisation, for completion calls.

We define $t_{\mathrm{minMPInonblockinit}}$, $t_{\mathrm{minMPItest}}$ and $t_{\mathrm{minMPIwait}}$ to be the minimum time required to execute a nonblocking initialisation call, a test call and a wait call respectively, without performing any progression.

The minimum time for nonblocking initialisation calls and `MPI_Test` calls is easy to meet. When an asynchronous progression mechanism is involved, the initialisation call job is to just fill in the request argument, then let the progression happen in background. So no extra time is taken for progression. For a `MPI_Test` call, its job is just to test if the associated operation is finished. If not, it will let the background mechanism to progress the operation, and will not take extra time to realize such progression.

However, a call to `MPI_Wait` actually has to *wait* until the operation is done. If the operation is not finished when the call is performed, then it will block until the operation completes. Even if the progression happens in background, it cannot be overlapped by computations and it will not be hidden. Our model aims at predicting if an application will benefit from the use of nonblocking communications with a dedicated core for asynchronous progress. Because of the semantics of the `MPI_Wait` procedure, nonblocking communications by themselves are useful only if there is enough computation to hide the communications. Hence, in our model, we consider the best-case scenario for using nonblocking communications. We assume that, when transformed to use said nonblocking communications, the application exhibits enough computations between an initialisation call and its corresponding completion call to completely overlap the communication time. In this case, when using a dedicated core for progression, the execution time for the `MPI_Wait` calls will always be the minimum time.

Thus, if we consider that the application embeds $N_{\mathrm{nonblocking}}$ nonblocking initialisation calls, N_{test} `MPI_Test` calls and N_{wait} `MPI_Wait` calls, we model the time for all calls

involved in nonblocking communications to be:

$$t_{\text{MPInonblockdedicated}} = N_{\text{nonblocking}} \times t_{\text{minMPInonblock}} + N_{\text{test}} \times t_{\text{minMPItest}} + N_{\text{wait}} \times t_{\text{minMPIwait}}$$

Impact on blocking calls We have seen so far how the nonblocking calls already in the application will behave with a dedicated core. We will now see what would happen if blocking calls would be changed into nonblocking calls to benefit from the use of a dedicated core.

Due to algorithmic constraints, it is expected that even with a large refactoring of the application, not all blocking calls may be changed to nonblocking with overlap. Thus blocking communications in the original application will either remain blocking communications, or be changed in their nonblocking counterparts (hence adding the necessary initialisation and completion calls). Both cases will behave differently when a dedicated core is used.

For the first case, one can think that it is trivial, as dedicated core is used to progress nonblocking calls and not blocking calls. This is wrong. As we said in the previous part, blocking calls may help progress nonblocking communications. We decided to decompose the time of a blocking call $t_{\text{MPIblocking}}$ in the time of the actual blocking communication $t_{\text{MPIblockingcom}}$ and the time spend to progress pending nonblocking communications $t_{\text{MPIblockprogress}}$:

$$t_{\text{MPIblocking}} = t_{\text{MPIblockingcom}} + t_{\text{MPIblockprogress}}$$

If a dedicated core is used for progress, then blocking calls will not progress nonblocking communications anymore. Thus, $t_{\text{MPIblockprogress}}$ becomes null, and we have :

$$t_{\text{MPIblockingdedicated}} = t_{\text{MPIblockingcom}}$$

For the second case, as the blocking communication is transformed in a nonblocking communication, its time becomes similar to nonblocking communications. The most direct way to change a blocking communication to its nonblocking counterpart is to call the corresponding initialisation call, then `MPI_Wait` as its completion call. Thus, the new time when using a dedicated core for progression is : $t_{\text{MPIblockingtransformed}} = t_{\text{minMPInonblock}} + t_{\text{minMPIwait}}$.

Let us consider an application with N_{blocking} MPI blocking communications, and that a ratio α of these blocking communications are transformed in their nonblocking counterparts, the new time for these communications are:

$$t_{\text{MPIoldblocking}} = \alpha \times N_{\text{blocking}} \times t_{\text{MPIblockingtransformed}} + (1-\alpha) \times N_{\text{blocking}} \times t_{\text{MPIblockingdedicated}}$$

which can be developed in:

$$t_{\text{MPIoldblocking}} = \alpha \times N_{\text{blocking}} \times (t_{\text{minMPInonblock}} + t_{\text{minMPIwait}}) + (1-\alpha) \times N_{\text{blocking}} \times t_{\text{MPIblockingcom}}$$

Impact on the other MPI calls and full MPI model The last type of MPI calls in our classification is *other*, which are the MPI calls that handle the library and internal structures management, but perform no communications and no progression. These calls are not impacted by the use of a dedicated core for progress, and the time t_{MPIother} remains the same.

Thus, when putting together all parts of the MPI model and considering that a ratio α of blocking calls will be changed between the initial version of the application and the version we want to model, the MPI time when using a dedicated core is:

$$t_{\text{MPIdedicated}} = t_{\text{MPInonblockdedicated}} + t_{\text{MPIoldblocking}} + t_{\text{MPIother}}$$

which can be developed in:

$$\begin{aligned} t_{\text{MPIdedicated}} = {} & N_{\text{nonblocking}} \times t_{\text{minMPInonblock}} + N_{\text{test}} \times t_{\text{minMPItest}} + \\ & N_{\text{wait}} \times t_{\text{minMPIwait}} + \alpha \times N_{\text{blocking}} \times (t_{\text{minMPInonblock}} + t_{\text{minMPIwait}}) \\ & + (1-\alpha) \times N_{\text{blocking}} \times t_{\text{MPIblockingcom}} + t_{\text{MPIother}} \end{aligned}$$

Link to the measures in chapter 4

The chapter 4 defined the MPI time in two parts: the t_{call} is the time spent in the MPI initialisation call and the t_{wait} time spent in the MPI Wait. Theses two parts defined to compute metrics represent the duration on a single *place* of overlap. In real applications, overlap situations can be multiple. The definition of $t_{\text{MPInonblocking}}$ can then be defined as the sum of the t_{call} from each MPI initialisation call in the application. Similarly, the t_{MPIwait} can be defined as the sum of all t_{wait}.

6.1.4 Global model with MPI and computation

The final model is the combination of the estimated computation slowdown modelled in section 6.1.2, and the communication evolution modelled in section 6.1.3:

$$t_{\text{dedicated}} = t_{\text{newcomp}} + t_{\text{MPIdedicated}}$$

which can be developed in:

$$\begin{aligned} t_{\text{dedicated}} = {} & t_{\text{comp}} \times \frac{N_{\text{core}}}{N_{\text{core}} - 1} + \\ & N_{\text{nonblocking}} \times t_{\text{minMPInonblock}} + N_{\text{test}} \times t_{\text{minMPItest}} + \\ & N_{\text{wait}} \times t_{\text{minMPIwait}} + \alpha \times N_{\text{blocking}} \times (t_{\text{minMPInonblock}} + t_{\text{minMPIwait}}) \\ & + (1-\alpha) \times N_{\text{blocking}} \times t_{\text{MPIblockingcom}} + t_{\text{MPIother}} \end{aligned}$$

The good cases where the dedicated core brings some improvement in the execution time should have a gain on the MPI part (coloured part in the model) higher than the overhead estimated in the computation part (black part in the model).

6.2 Gathering applications information to use the model

The whole model is based on timings for each part of the original application. These timings need to be measured on an execution of the original application. Several methods can be used to gather these timings, such as using a tool wrapping MPI calls, or having probes inside the MPI runtime.

In our experiments, we used the former method with the *mpiP* [68] tool. *mpiP* is a light-weight profiling library for MPI applications.

It generates logs gathering time spent in each MPI functions, number of calls, message sizes with minimum, maximum and average value. Information are collected for each MPI rank. This tool is generic enough to gather information on any tested application. However, because we only have application-level timings and not runtime-level timings, we will approximate some terms of our model.

First, we have to approximate the minimum time for nonblocking initialisation calls, `MPI_Test` and `MPI_Wait` calls. To do so, we consider the minimum measured time of each type of call when executing the application. Even when no dedicated core is used for progress, completion calls may not perform progress. When a completion call is used on a request, the associated operation may already be done, because it has been progressed by other previous MPI calls (blocking calls, or completion calls for other operations). If there is no other pending operation, then the completion call will just check the status of the request, and its timing will be the minimum. We assume this case happens at least once in an MPI run for nonblocking initialisation, `MPI_Test` and `MPI_Wait` calls. Thus, we take the minimum measured time for each of these as the respective timings for $t_{\text{minMPInonblock}}$, $t_{\text{minMPItest}}$ and $t_{\text{minMPIwait}}$.

Second, without having probes in the MPI runtime, it is not possible to know which part of the time spend in a blocking call actually relates to the execution of the associated blocking operation, or if some of the measured times correspond to progression of pending nonblocking operations. For this reason, we consider that the time measured for each blocking call is fully dedicated to the associated operation, and that $t_{\text{MPIblockprogress}}$ is always null.

These approximations may cause some discrepancies between our model forecasts and measured runs, but we will show that it remains accurate enough for our purpose.

6.3 Evaluation of the model

In this section, we evaluate how close to reality the model we propose is. We run the applications presented in section 6.1.2 on various data sets and number of nodes; it is especially important to use various data sets since the behaviour of some applications depends on inputs. Each run is performed in hybrid MPI+OpenMP mode, with one MPI process per node, and each compute resources of a node used by the OpenMP threads for computation. We performed our tests with the MPI implementation *MadMPI*, as it allows activating and deactivating asynchronous progression and use of a dedicated core [30]. The machines used are *inti/sandy-bridge* presented in section 6.1.2.

To evaluate the model, we compare its output to the real results obtained when running the applications with a dedicated core. This evaluation is limited to applications that

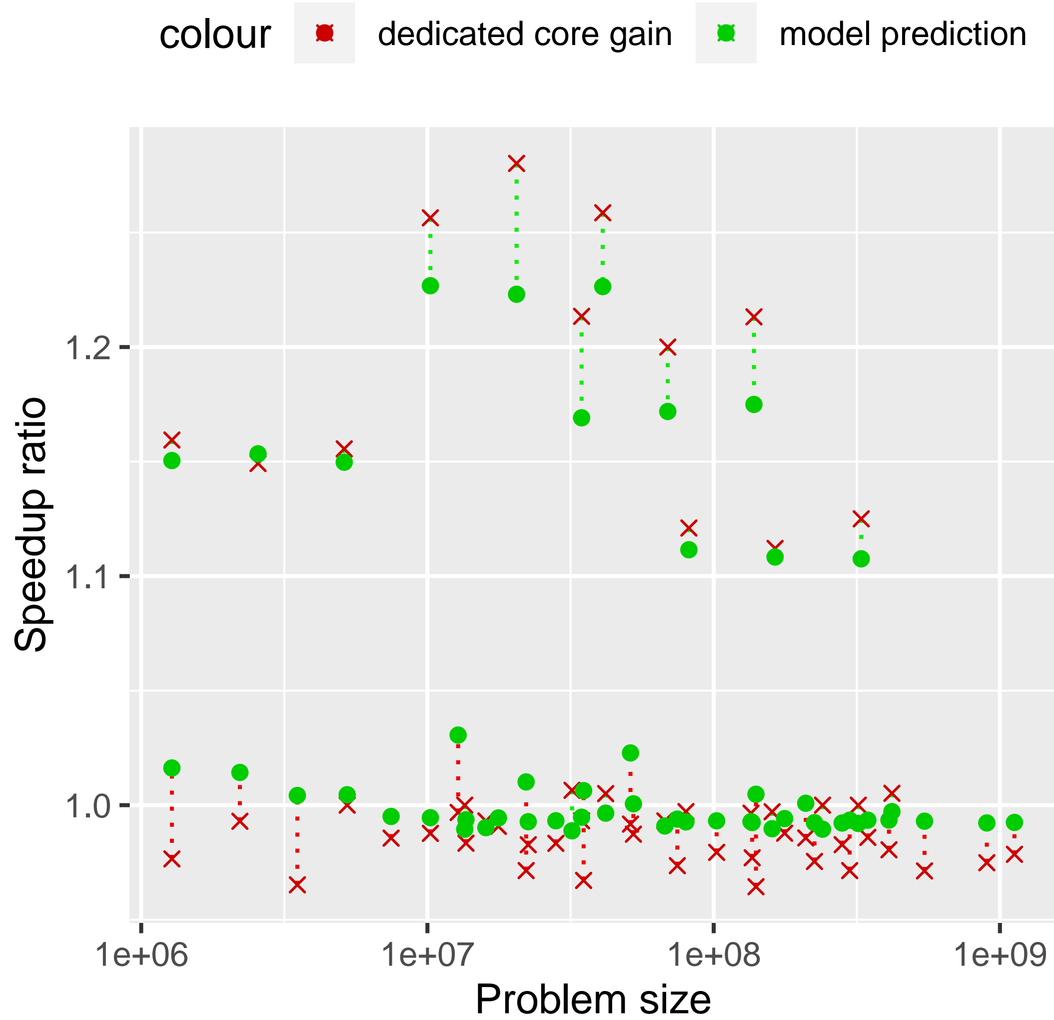


Figure 6.2: Comparison of the model prediction and the dedicated core case (Speedup to the time with no progression) for the Kripke application for different problem sizes on *inti/sandy-bridge*.

Figure 6.3: Comparison of the model prediction and the dedicated core case (Speedup to the time with no progression) for NAS BT-MZ for different configurations on *inti/sandy-bridge*.

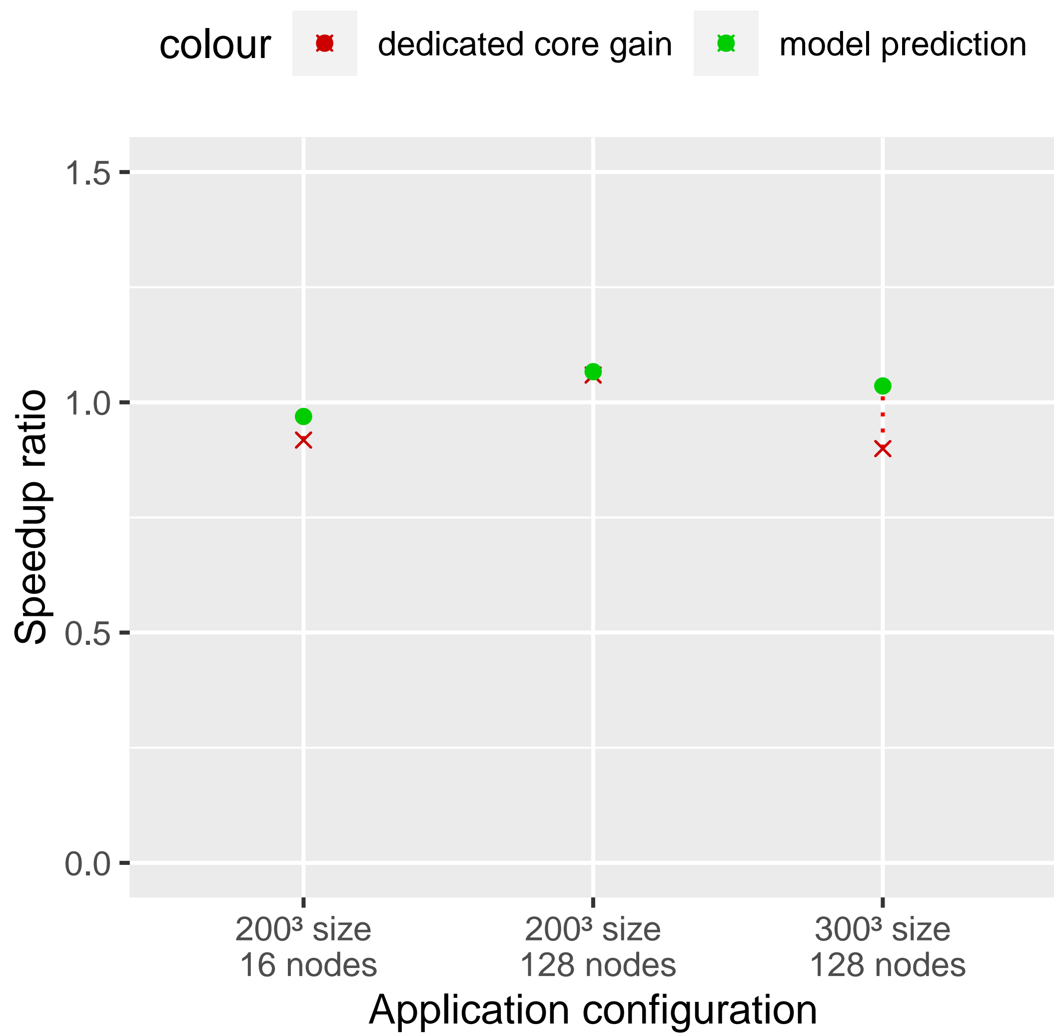


Figure 6.4: Comparison of the model prediction and the dedicated core case (Speedup to the time with no progression) MiniMD for different configurations on *inti/sandy-bridge*.

already leverage nonblocking communications. Since we only run unmodified applications, in this section we have $\alpha = 0$.

Each combination of applications/parameters were run twice:

- First, we collect data to calibrate our model. We run the application without a dedicated core, and without any progression mechanisms for communications. This run is similar to what a user obtains with a regular MPI library without progression. We measure $t_{\text{noprogress}}$ and we gather all the input parameters needed by the MPI model as explained in section 6.1.3, with the help of *mpiP*. We used bash scripts to get the data necessary to compute our predicted t_{model} with an *R* script[1] building our model presented in section 6.1.4.

- The second run uses one dedicated core mapped on the first logical core (e.g. core #0). We configure the OpenMP runtime to use $N_{\text{core}} - 1$ threads and they are bound on all other available cores (1 to $N_{\text{core}} - 1$). All progression mechanisms are enabled. This execution gives us the real value for $t_{\text{dedicated}}$.

The closer t_{model} is to $t_{\text{dedicated}}$, the better the model is.

We run the model on three applications: BT-MZ from the NAS Parallel Benchmark, and Kripke and MiniMD from the CORAL benchmarks. The results are shown in Figure 6.2 for Kripke, Figure 6.3 for NAS BT-MZ, and Figure 6.4 for MiniMD. On these figures, the x-axis corresponds to executions on various data sets, and various number of nodes for BT-MZ and MiniMD. The y-axis shows the performance represented as a speedup compared to the basic execution without dedicated core. For each configuration, we display two values. The red cross corresponds to the real execution with a dedicated core, defined as $s_{\text{dedicated}} = \frac{t_{\text{noprogress}}}{t_{\text{dedicated}}}$; the green bullet represents the performance predicted by the model, defined as $s_{\text{model}} = \frac{t_{\text{noprogress}}}{t_{\text{model}}}$. Thus, the closer the red cross is from the green bullet, the more accurate our model is.

Kripke was run with 52 different data sets, depicted in Figure 6.2. We distinguish two types of behaviour: cases where the dedicated core brings a significant speedup (> 1.1), and cases where the dedicated core does not bring any benefit (speedup below or very close to 1). Note that beneficial cases are not grouped together, so it may be hard for a user to know specifically which case is a good candidate for running with a dedicated core. We observe that our model accurately predicts the behaviour of the application for each data set. For all cases where the model predicts a significant speedup, it is confirmed by the experimental execution. And, for all cases where the predicted speedup is close to 1, we see this exact behaviour with the experimental run.

For NAS BT-MZ (figure 6.3) and MiniMD (figure 6.4), the results are less identical. On NAS BT-MZ, the results for class C and D on 16 nodes, the predictions from the model are correct — the dedicated core brings no gain. However, for class D on 128 nodes, the model predicts a speedup higher than 3 where the reality is a mere 0.96. This is due to the communication scheme. Our model makes the assumption that nonblocking operations are always overlapped by computation. Unfortunately, in this application the nonblocking operations are not designed to overlap communications and computation;

[1] The source code is available at <suppress for double-blind evaluation>

they are used to overlapping multiple communications. Indeed, the code calls a series of `MPI_Isend` and `MPI_Irecv` with a final `MPI_Waitall`, and no computation in between.

For MiniMD we observe that the first two predictions are correct and the third is much too optimistic. For the first two cases, the execution with a dedicated core is slower than without (see Fig 6.1a) and the model is accurate. For the last cases, when looking at the detailed values for all parameters of the model and compare them to the real execution, we observe an increased time spent in `MPI_Wait` with dedicated core (4,19 s with no progression against 8,69 s with dedicated core). This behaviour is sometimes observed when there is an interaction between the dedicated core and the `MPI_Wait`: they try to progress simultaneously. In those cases, the progression is not just done by the dedicated core and hence the gain is less than expected by the model.

We have shown here that the proposed model successfully predicts performance of a hybrid MPI application using a core dedicated to communication when the hypotheses are fulfilled. It exhibits a very good precision in this case, as seen with the Kripke application.

However, the user should be wary of using it blindly. It may give wrong results when hypotheses are not fulfilled: when no computation is executed at the same time as nonblocking communications, like in NAS BT-MZ; when the progression is still performed in `MPI_Wait` like in MiniMD, and when the OpenMP scalability is too far from linear.

In conclusion, the model is strong enough so as to be used to predict performance of dedicated core, but the user should always check all terms of the model and not only rely blindly on the total. These differences due to the approximation made in the model can help application developers find suboptimal behaviour related to the use of nonblocking communications in their applications.

6.4 Using the model

In this section, we will detail how to use the model to understand the compromise between computation slowdown and the communication speedup, to evaluate the effectiveness of nonblocking communication usage, and to predict the maximum performance we would get if blocking communications were converted to nonblocking.

6.4.1 Understanding the computation-slowdown/communication-speedup ratio

As we have seen for Kripke in figure 6.2, the application exhibits two types of behaviour with a dedicated core: either a significant speedup, or no gain at all. The beneficial cases are spread along the x-axis, showing that it does not depend on the problem sizes but instead on the communication scheme.

To understand this specific behaviour, we analyse all the different timings gathered in the basic run displayed in table 6.1, and the real value for $t_{\text{dedicated}}$ in addition. The case #1 is an example of successful use of dedicated core with 20% speedup; case #2 is typical of situations where the dedicated core does not bring performance gain. In both cases, the prediction of the model is correct with an error less than 2%.

Kripke features some calls to `MPI_Isend` and `MPI_Irecv`; they are progressed using `MPI_Testany` between computation phases. The communication is ended by a final call

	Case #1	Case #2
$t_{\text{noprogress}}$	147	155
t_{MPI}	73.4	8.3
t_{model}	119.82	156.75
$t_{\text{dedicated}}$	117	154
N_{core}	16	16
t_{comp}	73.6	145.7
$N_{\text{nonblocking}}$	480000	96
$t_{\text{minMPInonblock}}$	$2.14e-05$	$2.16e-05$
N_{test}	1119810	109679
$t_{\text{minMPItest}}$	$2.05e-05$	$2.2e-05$
N_{wait}	10000	2
$t_{\text{minMPIwait}}$	$3.2e-05$	$7.19e-05$
$t_{\text{MPIblocking}}$	11.7	4.98
t_{MPIother}	0.0	0.34

Table 6.1: Values of model parameters for two sets of parameters of Kripke (times in seconds).

to `MPI_Waitall`. We observe in the table the main differences between the two runs is $N_{\text{nonblocking}}$: the first case uses a lot of nonblocking operations while the second has very few of them. It should be noticed that both cases execute approximately in the same duration, even though once decomposed, the timing details are very different.

In the first case, the application uses enough nonblocking operations for the communication speedup to overcome the computation slowdown. On the contrary, in the second case, the small number of nonblocking operations does not successfully counterbalance the computation slowdown. With our model, an application developer can better understand the dynamic behaviour of its code, and know the cases where the nonblocking operations are used. The real impacting factor is the communication scheme and the amount of communications compared to computation.

6.4.2 Evaluating the effectiveness of nonblocking communication

An application may have everything theoretically to gain time in terms of MPI overlappable communication and computation, and structurally never put computation and nonblocking communication in parallel. This is the case for NAS BT-MZ. As we have seen in Figure 6.3, on 128 nodes the model is too optimistic. This is due to the nonblocking operations being used not to overlap with computation, but to overlap multiple communications. Hence, even if globally there would be enough computation to overlap all nonblocking communications, it is not placed at the same time as the nonblocking communication.

If an application already uses nonblocking communications, the model may be used to diagnose pathological cases: by comparing an actual run and the prediction of the model, we can check whether overlap performs as expected or not, whether the computation inserted between the nonblocking operation and the corresponding wait is long enough or not. If the actual run is far from the prediction, there may be room for improvement in

	Case #1	Case #2	Case #3
number of nodes	16	128	128
problem size	200^3	200^3	300^3
$t_{\text{noprogress}}$	112	17.9	57.3
t_{MPI}	23.4	8.17	22.4
t_{model}	98	12.61	54.9
$t_{\text{dedicated}}$	122	16.9	63.7
N_{core}	16	16	16
t_{comp}	94.55	9.73	34.9
N_{blocking}	5863	5863	263
$t_{\text{minMPInonblock}}$	$2.50e-05$	$7.49e-05$	$1.03e-05$
N_{wait}	5862	5862	5862
$t_{\text{minMPIwait}}$	$5.33e-06$	$5.47e-06$	$5.59e-06$
$t_{\text{MPIblocking}}$	19.16	4.99	0.41
t_{MPIother}	2.20	1.42	17.8

Table 6.2: Values of model parameters for three sets of parameters of MiniMD (times in seconds).

the organization of communications and computations.

We have observed the symmetrical case, where the application performs actually better with a dedicated core than predicted by the model. It is especially the case with Kripke, on the points of figure 6.2 with a speedup higher than 1.1. When looking at the detailed parameters we measured, we observe that with the dedicated core, N_{test} drops compared to the reference run. The model assumes that `MPI_Test` will be shorter with dedicated core, because it will not have to make communication progress in the call itself. But, in addition, N_{test} is decreased, because the communication finished earlier. A more progression-compliant version of Kripke should remove the calls to `MPI_Test` and let the dedicated core do all the progress work in the background. However, this requires a refactoring of the application. The lesson learned is that we may gain more than the sole cost of communication progression. In addition, we may gain the cost of all tests scattered throughout the code.

6.4.3 Potential gain of transforming blocking call to nonblocking

Finally, the model is able to predict the potential gain of transforming blocking operations to nonblocking ones. As described in section 6.1.3, the model can take into account the behaviour of an application if a ratio α of blocking operations are changed to their nonblocking counterparts.

We apply the model on MiniMD, on the three configurations used in section 6.3. MiniMD uses mostly blocking calls, with very few nonblocking operations. The figures for the three configurations are gathered in table 6.2. The predicted speedups for varying α for each configuration are displayed in figure 6.5. A ratio $\alpha = 0$ is equivalent to the original unmodified application. A value $\alpha = 1$ means that all blocking communications are transformed to their nonblocking counterparts. However, the ability to transform

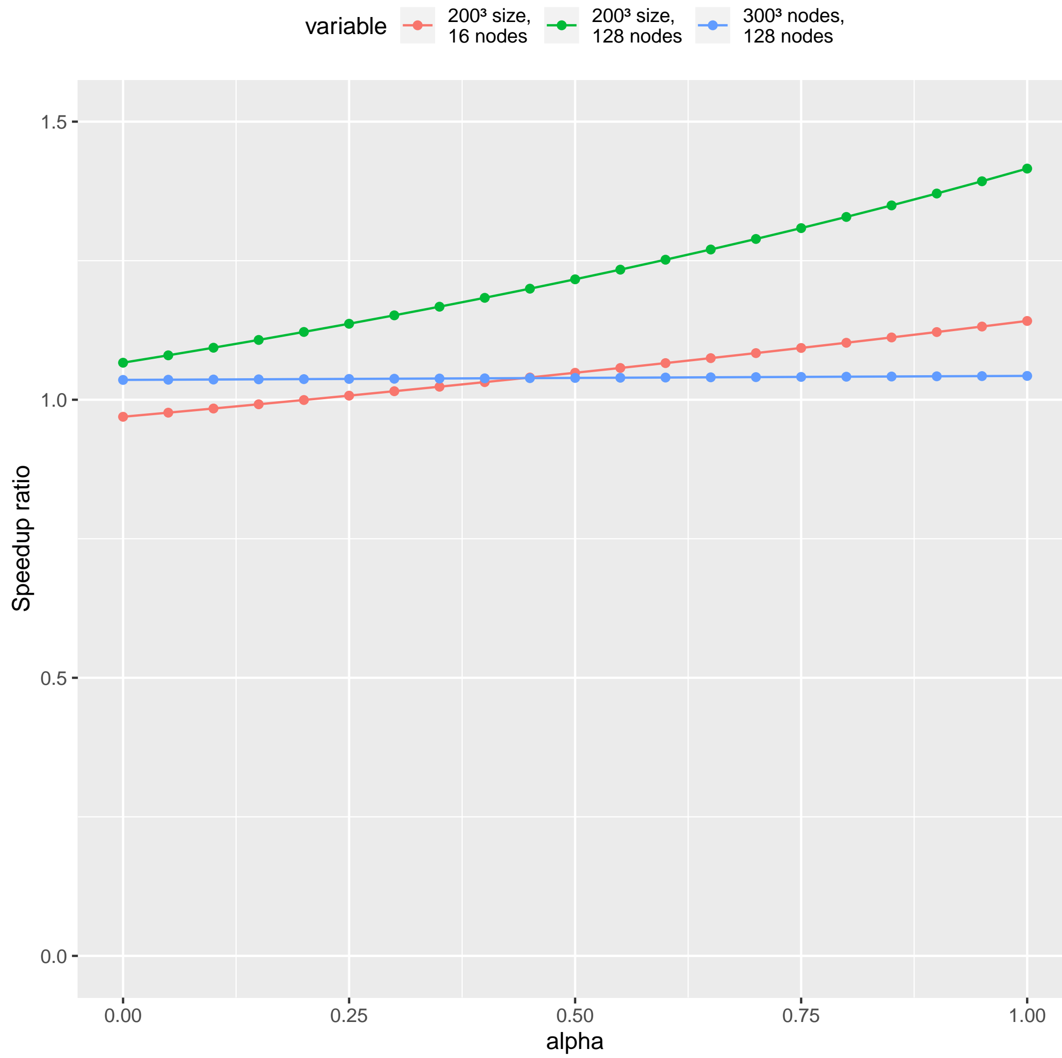


Figure 6.5: Evolution of predicted speedup relative to the proportion of blocking call converted to nonblocking for MiniMD

blocking to nonblocking calls depends on having available data-independent computation between the call and the wait. For most applications, $\alpha = 1$ is not reachable; we need to study the full range of values for α.

The results for the first and third sets of parameters (red and blue line) show that, even in the unlikely event of being able to convert all communications to nonblocking, the expected gain is poor. Values shown in table 6.2 reveal that the number of blocking calls N_{blocking} is low and the time spent in blocking communication $t_{\text{MPIblocking}}$ is low compared to the $1/16$th overhead on computation. In the first case, for realistic values of α, no gain may be expected. In the third case (blue line), the model predicts a very small gain for any value of α. However, we know that there would be actually no gain at all, considering that we have already seen in section 6.3 that our model is optimistic and overestimate by 12% the potential gain for this precise application.

The second set of parameters (green line) is the same as the first, except for the higher number of nodes. The predictions of the model, however, are very different, with a predicted significant performance improvement, even for moderate values of α. With the higher number of nodes compared to the first case, the computation time is lower, and thus the communication time gets a higher proportion of the total time, which is enough to compensate for the slowdown cause by one less core for computation with a dedicated core.

The slope of each case plot is the most interesting metric as it represents the efficiency of blocking to nonblocking transformation. The first and the second cases have a positive slope since the time spent in blocking communications $t_{\text{MPIblocking}}$ is much larger than what is lost in computation speed with a dedicated core; in contrast, the third case has a quasi-neutral slope. Even with t_{MPI} representing a large part of the total time, the proportion of $t_{\text{MPIblocking}}$ is negligible. As a consequence, α has a very low impact on the potential gain and the transformation is not efficient. Thus, for every case, we have to evaluate the potential gain by running the model with the parameters from the application.

This model is thus able to discriminate the cases where a transformed application will be effective or not. Taking into account the base point of the α plot, it can show the current state of the dedicated core effect on the application. At last, looking at the slope for used case of the application will show the impact of the transformation on the performances using a dedicated core.

Chapter 7

Conclusion

MPI is one of the most used programming model to exploit distributed memory systems. It allows the nodes composing the cluster to communicate using messages. However, performing communication has a cost and impact the overall performance. The overlap of computation and communication is a method used to reduce the impact of the communications MPI include notably nonblocking operations designed to execute independent computation during the communication on the network and enable overlap.

We showed that to efficiently have communication and computation operate simultaneously, the MPI runtime needs computation power to progress the communications. Moreover, MPI in its 3.0 version implement nonblocking collectives. The need for progression of these collectives is greater as these algorithms can't be only launched and let running on their own: they need to be run regularly all along the collective duration.

The necessity of progression to efficiently overlap nonblocking collectives is at the origin of new mechanisms developed for this purpose in MPI runtimes. Indeed, the application often rely on shared memory programming model such as OpenMP to benefit from multicore nodes. Thus associate a good overlap and a minimal impact on the global performance remains an issue. This thesis focused on the idea of progressing MPI communications using dedicated resources. The use of a dedicated core to remedy this problem come with multiple aspects that need to be taken into account such as:

- Creating a mechanism able to efficiently progress communications without degrading the performance
- Managing the resources required by both MPI runtime for the progression and the application computation.
- Understanding the causes and find solutions to diagnose bad overlap.
- Give a clue on the effect of a dedicated core in function of the application and system configuration.

We first presented the bases of high performance computing including hardware architectures and their evolution to the current problem we face up to. We saw the different

software technologies such as programming models used to operate these complex hardware systems. We saw that benchmarking methods used to evaluate the MPI runtimes may miss important behaviour to assess the performance of progression mechanisms.

In this thesis, we propose a global solution to efficiently progress communications and especially nonblocking collective in order to overlap communication and computation. The final purpose is then to benefit from this overlap to improve the performance of hybrid applications. This solution is composed of multiple axis: we propose a new benchmarking method and metrics focusing on the evaluation of nonblocking collective progression. This benchmarking method have for purpose to be use as a base to measure the overlap capability and monitor the effects of progression mechanism on overall performances.
We thus propose a new mechanism to progress nonblocking collective, relying on task-based algorithm and dedicated core. The use of dedicated core have shown an efficient capability of gaining performances through overlap, but requires to be used in the good conditions concerning the system and application structure.
Thus, we finally propose a model to predict the effects of using this dedicated core mechanism on real hybrid MPI + OpenMP applications. This model can be used to simulate the potential gain when converting blocking communication application to use nonblocking operations.

7.1 Contributions

The contributions made in this thesis have followed three main axes: the benchmarking method for nonblocking collective, the conception of a progression mechanism based on a dedicated core and the modelling of this mechanism on the performance of hybrid MPI + OpenMP applications.

7.1.1 Nonblocking collective benchmarking

We first proposed a full methodology to not only measure the overlap capability of mechanisms but also their impact on the overall system. This methodology relies first on new metrics to accurately measure the performance of nonblocking collective with the definition of an overhead ratio. They also assess the global behaviour of the runtime with the computation and communication concurrency ratio.
We then proposed an implementation addressing multiple common issues encountered when benchmarking distributed operations. Such problems include the time variation between MPI processes, the management of time synchronisation between nodes clocks, the requirement of a simultaneous start, and finally the necessity of controlling the variable amount of computation and communication done in the overlapped situation compared to the sequential one.
Finally we used this benchmark to run a survey on the state-of-the-art most common MPI libraries, and the different progression mechanisms they use. This showed the impact of these mechanisms on the system and how they fail to manage resources in order to gain performances: the progression mechanism usually consume a non-negligible amount of computation power and hinder the application running on the whole system. This brings the necessity to create a new mechanism focusing on these points.

7.1.2 Dedicated core for progression

The chapter 5 presented an overview of software necessities required to progress communication using dedicated cores. We then illustrated this overview with the conception needed for adapting the NewMadeleine + Pioman runtime to use dedicated core for the progression of nonblocking collective. We first analysed the benefits of using a dedicated core for this purpose and how this mechanism fits for this work.
We then proposed an implementation of a specific worker in Pioman tuned and adapted to be executed on a dedicated core. We proposed an implementation of nonblocking collective algorithms in NewMadeleine based on event handlers to be efficiently executed by the Pioman dedicated core worker.
Finally, we evaluated the progression capacity of this global configuration using the BenchNBC benchmark on the madMPI MPI interface on top of NewMadeleine.
The dedicated core evaluation showed a good capability of progressing communications to gain performances. However, to be used in a real context, applications must repsect constraints that need to be precisely defined to benefit from this solution.

7.1.3 Hybrid application with dedicated core modelling

The final chapter proposed a solution to classify applications following their relevance to be progressed with dedicated core in order to gain performances. We focused on the modelling of hybrid MPI + OpenMP applications performance using the dedicated core progression mechanism.
We defined a predicting model focusing on analysing the performance evolution of application using a dedicated core. The model is split in two parts One computing the loss due to the removal of one core from the application, and the other the gain of overlapping the application computation and communication using the dedicated core.
We introduced the methodology used to gather the necessary input from application to feed the model. This model thus rely on universal input which can be found in every MPI + OpenMP applications such as MPI operation or computation duration.
We evaluated this model using multiple common and representative HPC applications, such as Kripke, MiniMD and Lulesh. Finally, we ran our model to predict the benefit of transforming blocking calls to nonblocking calls on these applications based on the understanding of computation slowdown and communication speed-up ratio.

The evaluation of the model demonstrated its capacity to distinguish cases to use dedicated core and cases to not use it. It also showed its usage to hint developers about the efficiency to re-design blocking applications to nonblocking ones.

7.1.4 Answers about the dedicated core usage

The set of contributions presented in this thesis bring answers to the problem of the effective progression of nonblocking collectives. The combination of theses contributions not only helps the effective progression of nonblocking collective to gain performances, but also gives clues about the requirements to do so. It also defines an environment to use these primitives.

Globally, these contributions gives an answer to the questions of the nonblocking collective usage such as: How to gain performance using a dedicated core, what are the runtime requirements to gain performance with nonblocking collectives and when to use the dedicated core.

7.2 Perspectives

The contributions presented in this thesis open multiple opportunities and perspectives concerning MPI benchmarking and modelling. There are also works to continue concerning the dedicated core implementation.

7.2.1 Use of MPI nonblocking collectives benchmark for MPI library improvement

The benchmark method and implementation presented in the chapter 4 propose to diagnose the behaviour of MPI library. It could however be enhanced to fully cover the need of a complete benchmark.

Implementation of all MPI nonblocking collectives benchmark

The benchmark currently covers multiple collectives, however some remains not supported for now. Especially the support of collective with various data size (e.g, `MPI_Igatherv`, `MPI_Ialltoallv`) is an interesting point as these operations introduce the exchange of different data length. The main purpose of benchmarking these functions is to measure the impact of the communication unbalance on the progression, overlap, and overall behaviour of the system. To do so, the benchmark must introduce a new feature to assess the variation of different deltas between lengths used.

Work on a microbenchmark feature

The current implementation is conceived to test an overall set of computation and communication time, in order to give a general overview of the performance.
Another perspective is to be able to test specific cases of overlap. These test cases could be an association of a computation time, a communication time and a collective. Such microbenchmark would be useful to stick to the numerous real cases of actual applications.

Improve MPI libraries

The use of this complete methodology has for purpose to be used to expose weak points of MPI libraries concerning their progress mechanisms, in order to improve these points. This benchmark has shown performance issues concerning these progress mechanisms and identified the causes of these issues.
The combinations of existing metrics and future ones presented in this section could then be used as a tool to correct future implementation of multiple MPI libraries such as MPICH or OpenMPI.

7.2.2 Evolution of the dedicated core impact model

The current model presented in chapter 6 uses the mpiP framework to gather the information needed as input for the model. mpiP is a generic tool and lacks some timings that could greatly improve the model.

Get more timing on the computation part

One point to be refined in the model is the management of computation time. Indeed, with the current instrumentation, we have no idea of the actual amount of the computation respecting the constraint to be overlapped with communication The first mandatory one being the part of the computation located between the MPI initialisation call and the MPI Wait. Among this computation part, another constraint is to be sure that the data computed is actually independent of the data transferred over the network. However, this must already be the case from the conception of the code.

The section 4.4 showed that the computation have a direct impact on the overlap capability of a runtime. Especially in function of the unbalance between computation and communication time, the progression mechanism can be much less effective than with a balanced case.
This independent computation timing would thus be a precious addition to the model as it would allow it to simulate this efficiency loss calculated from the unbalance ratio.

Instrumenting the runtime

One solution envisaged to get these timings is to rely directly on a specific mode of an MPI runtime to gather the information we need. The modification of MPI initialisation calls and `MPI_Wait` would allow to measure the time between those two calls.
Coupled with the use of a dedicated core, we ensure that the time between these two bounds is not hindered by the MPI runtime.

Study on application transformation to overlap

The model in its current state is able to help to decide on the efficiency of transforming blocking application to nonblocking application. It could then be interesting to run a study on common and most used applications to evaluate the possibility of transforming these applications in order to gain performances.
Based on the proposed improvements, the enhanced model could gather quantitative information on the performance of these applications. Thus, it could be possible to measure the potential gain of this transformation to give a clearer hint on the cost/gain ratio of this operation.

7.2.3 Multiple runtime centralisation of resources management

The seek of performance in high performance computing imply to rely on clusters composed of numerous processors sharing memory and distributed nodes. To make the best of these complex hardware architectures, simulations and most widely applications may use and

associate multiple programming models. This combination of different runtimes on system bring issues for the management of computation resources provided by the cluster.

A tool for global resource allocation

To distribute the resources between the runtime we used along this thesis, we relied on manually setting the placement of threads for each used runtime. Despite being a fully working solution, this has the disadvantage of requiring to resolve the resource problem with every different cases. The potential case of having more than two runtimes could make this resource issue even more complicated. To allow the good cooperation between those runtimes, an idea could be to develop a tool with the global overview of the available resources. From this global view, it could then take decision on how to allocate resources for the given runtimes used by applications.

A task based global runtime

Different runtimes may sometimes propose features with a similar workload management. The use of tasks has shown a great capacity for the progression of communications. Moreover, we presented hybrid MPI + OpenMP applications which represent a very common association of runtimes. OpenMP, on its side, also features tasks to be executed by its runtime. Thus, it is possible to have both OpenMP running a task engine to execute parallel tasks, and an MPI runtime relying on tasks to achieve the communication progression work.
The use of a global runtime implementing both OpenMP and MPI standard could thus unify the task engine to be used by all runtimes when needed. The conception of that solution would allow runtime to thus share resources for similar work. However, conceiving a global task engine requires combining the requirements for each runtime. For instance, OpenMP tasks usually rely on data dependencies to organise and execute the "taskified" workload. On its side, the MPI implementation presented in this thesis use events to trigger the available tasks. Thus, the design differences of each task engine is an issue in order to implement such global engine.
The notion of unifying runtimes into a global implementation have already been covered. MPC is a framework including implementations for both MPI and OpenMP. It could be used as a base to develop a task-based progression environment for MPI using a global task engine for both the OpenMP and MPI runtimes.

Bibliography

[1] "Pvm website." `https://www.csm.ornl.gov/pvm/`.

[2] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawak, and C. V. Packer, "Beowulf: A parallel workstation for scientific computation," in *Proceedings, International Conference on Parallel Processing*, vol. 95, pp. 11–14, 1995.

[3] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W.-K. Su, "Myrinet: a gigabit-per-second local area network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, 1995.

[4] M. Pérache, H. Jourdren, and R. Namyst, "Mpc: A unified parallel runtime for clusters of numa machines," in *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, (Berlin, Heidelberg), p. 78–88, Springer-Verlag, 2008.

[5] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing* (IEEE, ed.), (Pisa, Italy), Feb. 2010.

[6] "Gomp website."

[7] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Euro-Par 2009*, LNCS, (Delft, Netherlands), Aug. 2009.

[8] O. Aumage, E. Brunet, N. Furmento, and 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.

[9] S. Derradji, T. Palfer-Sollier, J.-P. Panziera, A. Poudes, and F. W. Atos, "The bxi interconnect architecture," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 18–25, 2015.

[10] "Mpich website." `https://www.mpich.org/`.

[11] O. Aumage, E. Brunet, N. Furmento, and R. Namyst, "New madeleine: a fast communication scheduling engine for high performance networks," in *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–8, March 2007.

[12] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The mvapich project: Transforming research into high-performance mpi library for hpc community," *Journal of Computational Science*, vol. 52, p. 101208, 2021. Case Studies in Translational Computer Science.

[13] "Intelmpi website." https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html#gs.m7ly1i.

[14] Q. Snell, A. R. Mikler, and J. L. Gustafson, "Net-pipe: Network protocol independent performance evaluator," 1997.

[15] Intel Corporation, "IMB-NBC benchmarks." `https://github.com/intel/mpi-benchmarks`.

[16] T. O. S. University, "Omb (osu microbenchmarks)." `http://mvapich.cse.ohio-state.edu/benchmarks/`.

[17] R. Reussner, P. Sanders, and J. L. Traff, "SKaMPI: A comprehensive benchmark for public benchmarking of MPI," *Sci. Program.*, vol. 10, p. 55–65, Jan. 2002.

[18] S. Hunold, A. Carpen-Amarie, and J. L. Träff, "Reproducible MPI micro-benchmarking isn't as easy as you think," in *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, (New York, NY, USA), p. 69–76, Association for Computing Machinery, 2014.

[19] D. Becker, R. Rabenseifner, and F. Wolf, "Implications of non-constant clock drifts for the timestamps of concurrent events," pp. 59–68, 09 2008.

[20] S. Hunold and A. Carpen-Amarie, "On the impact of synchronizing clocks and processes on benchmarking MPI collectives," in *EuroMPI*, pp. 8:1–8:10, ACM, 2015.

[21] A. Nigay, L. Mosimann, T. Schneider, and T. Hoefler, "Communication and timing issues with mpi virtualization," in *27th European MPI Users' Group Meeting*, EuroMPI/USA '20, (New York, NY, USA), p. 11–20, Association for Computing Machinery, 2020.

[22] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for mpi," p. 52, 01 2007.

[23] T. Hoefler, T. Schneider, and A. Lumsdaine, "Accurately Measuring Overhead, Communication Time and Progression of Blocking and Nonblocking Collective Operations at Massive Scale," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 25, pp. 241–258, Jul. 2010.

[24] A. Denis and F. Trahay, "MPI Overlap: Benchmark and Analysis," in *International Conference on Parallel Processing*, 45th International Conference on Parallel Processing, (Philadelphia, United States), Aug. 2016.

[25] MPI Forum, *MPI: a Message Passing Interface Standard V1.0.* 1993.

[26] D. R. Martinez, J. C. Cabaleiro, T. F. Pena, F. F. Rivera, and V. B. Perez, "Performance modeling of mpi applications using model selection techniques," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pp. 95–102, 2010.

[27] S. Sur, H.-W. Jin, L. Chai, and D. Panda, "R545250odma read based rendezvous protocol for mpi over infiniband," pp. 32–39, 01 2006.

[28] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing - to thread or not to thread?," in *2008 IEEE International Conference on Cluster Computing*, pp. 213–222, Sep. 2008.

[29] M. Si, A. Peña, P. Balaji, M. Takagi, and Y. Ishikawa, "Mt-mpi: multithreaded mpi for many-core environments," 06 2014.

[30] A. Denis, "pioman: a pthread-based Multithreaded Communication Engine," in *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, (Turku, Finland), Mar. 2015.

[31] M. Miwa and K. Nakashima, "Progression of mpi non-blocking collective operations using hyper-threading," pp. 163–171, 03 2015.

[32] M. Jiayin, S. Bo, Y. Wu, and Y. Guangwen, "Overlapping communication and computation in mpi by multithreading.," pp. 52–57, 01 2006.

[33] MPI Forum, *MPI - a Message Passing Interface Standard V3.0. Ean 1114444410030*. English Book Service, 2012.

[34] D. Buettner, J.-T. Acquaviva, and J. Weidendorfer, "Real asynchronous mpi communication in hybrid codes through openmp communication tasks," pp. 208–215, 12 2013.

[35] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein, "Asynchronous mpi for the masses," 02 2013.

[36] G. Hager, G. Schubert, T. Schoenemeyer, and G. Wellein, "Prospects for truly asynchronous communication with pure mpi and hybrid mpi/openmp on current supercomputing platforms," 01 2011.

[37] S. Paul, M. Araya, J. Mellor-Crummey, and D. Hohl, "Performance analysis and optimization of a hybrid seismic imaging application," *Procedia Computer Science*, vol. 80, pp. 8–18, 06 2016.

[38] A. Ruhela, H. Subramoni, S. Chakraborty, M. Bayatpour, P. Kousha, and D. Panda, "Efficient asynchronous communication progress for mpi without dedicated resources," pp. 1–11, 09 2018.

[39] H.-V. Dang, M. Snir, and W. Gropp, "Towards millions of communicating threads," in *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, (New York, NY, USA), p. 1–14, Association for Computing Machinery, 2016.

[40] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine, "Hybrid MPI: efficient message passing for multi-core systems," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE, 2013.

[41] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa, "MT-MPI: Multithreaded MPI for many-core environments," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, (New York, NY, USA), p. 125–134, Association for Computing Machinery, 2014.

[42] P. Carribault, M. Pérache, and H. Jourdren, "Enabling low-overhead hybrid MPI/OpenMP parallelism with MPC," in *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, Proceedings of the 6th International Workshop on OpenMP (IWOMP 2010)* (M. Sato, T. Hanawa, M. Müller, B. Chapman, and B. de Supinski, eds.), vol. 6132 of *Lecture Notes in Computer Science*, pp. 1–14, Springer Berlin Heidelberg, 2010.

[43] S. White and L. V. Kale, "Optimizing point-to-point communication between adaptive mpi endpoints in shared memory," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 3, p. e4467, 2020. e4467 cpe.4467.

[44] N. Hjelm, M. G. F. Dosanjh, R. E. Grant, T. Groves, P. Bridges, and D. Arnold, "Improving MPI multi-threaded RMA communication performance," in *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, (New York, NY, USA), Association for Computing Machinery, 2018.

[45] A. Ruhela, H. Subramoni, S. Chakraborty, M. Bayatpour, P. Kousha, and D. K. (DK) Panda, "Efficient design for MPI asynchronous progress without dedicated resources," *Parallel Computing*, vol. 85, pp. 13 – 26, 2019.

[46] R. E. Grant, M. G. F. Dosanjh, M. J. Levenhagen, R. Brightwell, and A. Skjellum, "Finepoints: Partitioned multithreaded mpi communication," in *High Performance Computing* (M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, eds.), (Cham), pp. 330–350, Springer International Publishing, 2019.

[47] J. H. Göbbert, H. Iliev, C. Ansorge, and H. Pitsch, "Overlapping of communication and computation in nb3dfft for 3d fast fourier transformations," pp. 151–159, 02 2017.

[48] T. Straatsma and D. Chavarría-Miranda, "On eliminating synchronous communication in molecular simulations to improve scalability," *Computer Physics Communications*, vol. 184, pp. 2634–2640, 12 2013.

[49] K. Al-Tawil and C. A. Moritz, "Performance modeling and evaluation of mpi," *Journal of Parallel and Distributed Computing*, vol. 61, no. 2, pp. 202–223, 2001.

[50] L. Adhianto and B. Chapman, "Performance modeling of communication and computation in hybrid mpi and openmp applications," *Simulation Modelling Practice and Theory*, vol. 15, no. 4, pp. 481–491, 2007. Performance Modelling and Analysis of Communication Systems.

[51] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis, “Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications,” in *SC’06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pp. 17–17, IEEE, 2006.

[52] R. Brightwell, R. Riesen, and K. Underwood, “Analyzing the impact of overlap, offload, and independent progress for message passing interface applications.,” *IJHPCA*, vol. 19, pp. 103–117, 01 2005.

[53] V. Gupta, H. Kim, and K. Schwan, “Evaluating scalability of multi-threaded applications on a many-core platform,” tech. rep., Center for Experimental Research in Computer Systems, Georgia Tech, 05 2012.

[54] A. Daumen, P. Carribault, F. Trahay, and G. Thomas, “Scalomp: Analyzing the scalability of openmp applications,” in *OpenMP: Conquering the Full Hardware Spectrum* (X. Fan, B. R. de Supinski, O. Sinnen, and N. Giacaman, eds.), (Cham), pp. 36–49, Springer International Publishing, 2019.

[55] C. Iwainsky, S. Shudler, A. Calotoiu, A. Strube, M. Knobloch, C. Bischof, and F. Wolf, “How many threads will be too many? on the scalability of openmp implementations,” in *Euro-Par 2015: Parallel Processing* (J. L. Träff, S. Hunold, and F. Versaci, eds.), (Berlin, Heidelberg), pp. 451–463, Springer Berlin Heidelberg, 2015.

[56] M. Woodyard, “An experimental model to analyze openmp applications for system utilization,” in *OpenMP in the Petascale Era* (B. M. Chapman, W. D. Gropp, K. Kumaran, and M. S. Müller, eds.), (Berlin, Heidelberg), pp. 22–36, Springer Berlin Heidelberg, 2011.

[57] N. Drosinos and N. Koziris, “Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters,” in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pp. 15–, 2004.

[58] S. Derradji, T. Palfer-Sollier, J. Panziera, A. Poudes, and F. W. Atos, “The BXI interconnect architecture,” in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 18–25, 2015.

[59] D. L. Mills, “Internet time synchronization: the network time protocol,” *IEEE Transactions on Communications*, vol. 39, no. 10, pp. 1482–1493, 1991.

[60] S. Hunold and A. Carpen-Amarie, “Hierarchical clock synchronization in mpi,” in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 325–336, 2018.

[61] “Bench NBC website.” `http://pm2.gitlabpages.inria.fr/bench_nbc/`.

[62] A. Denis, J. Jaeger, E. Jeannot, M. Pérache, and H. Taboada, “Study on progress threads placement and dedicated cores for overlapping mpi nonblocking collectives on manycore processor,” *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1240–1254, 2019.

[63] M. P. I. Forum, “MPI: A message-passing interface standard – version 4.0,” June 2021.

[64] H. J. R.F. Van Der Wijngaart, “Nas parallel benchmarks, multi-zone versions,” tech. rep., NASA Ames Research Center, Moffett Field, CA, 2003.

[65] P. Crozier, H. Thornquist, R. Numrich, A. Williams, H. Edwards, E. Keiter, M. Rajan, J. Willenbring, D. Doerfler, and M. Heroux, “Improving performance via mini-applications,” 01 2009.

[66] I. Karlin, J. Keasler, and R. Neely, “Lulesh 2.0 updates and changes,” Tech. Rep. LLNL-TR-641973, August 2013.

[67] A. J. Kunen, T. S. Bailey, and P. N. Brown, “Kripke - a massively parallel transport mini-app,” 6 2015.

[68] J. Vetter and C. Chambreau, “mpip: Lightweight, scalable mpi profiling,” *URL: http://www. llnl. gov/CASC/mpiP*, 01 2005.