On the Interactions between HPC Task-based Runtime Systems and Communication Libraries

Philippe Swartvagher

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE
DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Philippe Swartvagher**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

On the Interactions between HPC Task-based Runtime Systems and Communication Libraries

---

Sous la direction de Alexandre Denis et Emmanuel Jeannot

Soutenue le 29 novembre 2022

Devant la commission d'examen composée de :

M. Denis Barthou	Professeur, Bordeaux INP	Président du jury
M. George Bosilca	Assistant Professor, University of Tennessee	Examinateur
M. Alexandre Denis	Chargé de Recherche, Inria, Université de Bordeaux	Examinateur
M. Emmanuel Jeannot	Directeur de Recherche, Inria, Université de Bordeaux	Directeur de thèse
M. Arnaud Legrand	Directeur de Recherche, CNRS, Université Grenoble Alpes	Rapporteur
Mme Didem Unat	Associate Professor, Koç University	Rapportrice

Titre De l'interaction entre les supports d'exécution à tâches HPC et les bibliothèques de communications

Résumé Les supercalculateurs sont utilisés pour résoudre des problèmes numériques complexes demandant beaucoup de ressources de calcul (simulations, prévisions météorologiques, modélisations, *etc*), impossibles à exécuter sur des ordinateurs classiques. Ces supercalculateurs sont composés de nombreux puissants ordinateurs, connectés par un réseau. Bien que la puissance de ces supercalculateurs ne cesse d'augmenter, le développement d'applications exploitant toute leur puissance de calcul est de plus en plus complexe. En effet, de nombreux aspects doivent être considérés : des unités de calculs hétérogènes qui se programment différemment, la hiérarchie mémoire et les transferts de données, les communications réseau, l'ordonnancement, *etc*. Pour parer à ces difficultés, les *supports d'exécutions à tâches* ont émergé : ils représentent les applications par des graphes de tâches. Les différentes opérations exécutées par l'application et les dépendances entre elles forment un graphe. Il suffit alors de donner une implémentation de chaque tâche pour les unités de calcul ciblées, les dépendances entre les tâches et le support d'exécution se charge d'exécuter l'application : ordonnancer les tâches sur les différentes unités de calcul, réaliser les transferts mémoire et les communications réseau nécessaires, *etc*.

Dans cette thèse, nous explorons les différentes interactions possibles entre les supports d'exécution à tâches et la bibliothèque de communication utilisée pour réaliser les transferts réseau. L'objectif est de faire plus collaborer ces deux couches logicielles, pour améliorer la performances des applications exécutées. Pour analyser et comprendre les interactions entre les supports d'exécution et les communications, tracer les applications est une technique pertinente, malgré certaines limitations. C'est pourquoi nous commençons par évaluer les surcoûts possibles en termes de performances induits par un système de traces. Nous proposons des techniques pour réduire ces surcoûts et avons également évalué l'impact de la précision de la synchronisation d'horloge pour les traces distribuées. Ensuite, comme interaction positive entre un support d'exécution à tâches et la bibliothèque de communications, nous proposons une solution pour efficacement envoyer une même donnée à plusieurs destinataires, tout en respectant les contraintes du support d'exécution. D'autre part, nous considérons également les éventuelles interactions négatives, en évaluant les différentes sources d'interférences entre les calculs et les communications exécutés en parallèle, dégradant leurs performances respectives. Ayant observé que la contention mémoire entre les calculs et les communications a le plus d'impact, nous proposons finalement un modèle prédisant la répartition de la bande-passante mémoire entre les calculs et les communications. Ce modèle permet de mieux comprendre le comportement du composant mémoire en cas de contention et de prendre en compte ce phénomène dans les décisions du support d'exécution. Les contributions présentées montrent qu'améliorer les interactions entre les supports d'exécutions à tâches et les bibliothèques de communications a du potentiel pour améliorer les performances des applications HPC.

Mots-clés Calcul haute performance, programmation par tâches, calcul distribué, supports d'exécution, bibliothèques de communication, MPI, traces, contention mémoire

Laboratoire d'accueil Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33400 Talence

Title On the Interactions between HPC Task-based Runtime Systems and Communication Libraries

Abstract Supercomputers are used to solve complex and demanding computational problems (simulations, climate and weather forecasting, modelling, *etc*), impossible to run on regular computers. These supercomputers are composed of many powerful computers, interconnected through a network. While the power of these supercomputers increases over time, it becomes more and more challenging to develop applications taking benefit from all offered computing power. Indeed, many aspects have to be considered by the developer: heterogeneous computing units programmed in different manners, memory hierarchy and transfers, network communications, scheduling, *etc*. To overcome these challenges, *task-based runtime systems* have emerged. They model applications by graph of tasks: sub-computations and dependencies between them form a graph. The programmer has to provide the implementations of the tasks for each targeted computing unit, express dependencies between the tasks and then the runtime system is in charge of the application execution: scheduling tasks on different computing units, performing required memory movements between memories and network transfers, *etc*.

In this thesis, we explore the possible interactions between a task-based runtime system and the communication library it relies on to perform network transfers. The goal is to make these two software layers more collaborate, to improve performance of executed applications. To understand and analyze the interactions between the runtime systems and communications, tracing applications is a powerful technique. However, it can have some limitations. Thus, we first evaluate sources of performance overhead when tracing applications, propose solution to alleviate them and evaluate the impact of clock synchronization accuracy for distributed application tracing. Then, as a positive interaction between the task-based runtime system and the communication library, we propose a solution to efficiently send the same piece of data to several nodes, coping with the constraints of the considered runtime system. On the other hand, we also consider possible negative interactions, by evaluating the different sources of interferences between computations and communications being executed in parallel, more or less degrading their respective performance. Since we observe memory contention between computations and communications have the most impact, we finally propose a model predicting the memory bandwidth share between computations and communications, to better understand the behaviour of the memory system in case of contention and be able to take into account this phenomenon into decisions of the runtime system. Contributions presented in this manuscript show that improving interactions and cooperations between task-based runtime systems and communication libraries has potential to increase performance of HPC applications.

Keywords High performance computing, task-based programming, distributed computing, runtime systems, communication libraries, MPI, traces, memory contention

Hosting Laboratory Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33400 Talence

Contents

List of Figures

Remerciements

TROIS années de thèse sont passées, le manuscrit a été rédigé et la soutenance s'est clôturée par le fameux « *nous vous décernons le titre de Docteur en Informatique de l'Université de Bordeaux* ». Il semblerait bien qu'il ne reste plus qu'à rédiger les remerciements pour vraiment achever cette thèse.

Tout d'abord, merci au jury. Merci à Arnaud et Didem pour avoir relu ce (long et dense) manuscrit. J'espère que vous avez pris autant de plaisir à le lire que j'en ai eu lors de l'écriture. Merci pour vos commentaires constructifs qui ont permis de l'améliorer. Merci à George pour avoir accepté d'être examinateur, avoir un développeur de PaRSEC dans le jury d'une thèse qui traite principalement de StarPU me semblait à la fois intéressant et audacieux ! Merci à Denis pour avoir présidé le jury.

Les remerciements suivants s'adressent aux deux personnes qui ont permis à cette thèse d'être ce qu'elle est aujourd'hui : Alexandre et Emmanuel. Votre encadrement et votre soutien sans faille pendant ces trois années (voire plus pour Alexandre si on compte sa casquette de maître de stage) ont été très importants pour moi. Merci pour m'avoir transmis (une partie de) votre savoir, pour m'avoir fait découvrir le monde de la recherche ainsi que le raisonnement et la méthode scientifiques en informatique. Je pense qu'on a aussi les mêmes exigences en terme de travail bien fait et pour aller au fond des choses ; je trouve toujours agréable de travailler avec des personnes qui ont les mêmes attentes que moi, j'espère que c'était pareil pour vous. Avoir deux encadrants peut parfois être déstabilisant quand l'un dit *noir* et l'autre dit *blanc*, mais ces divergences d'opinions (finalement peu nombreuses) ont su être masquées par votre complémentarité. Je me demande parfois quelle est la part des encadrants dans le travail et les résultats d'un doctorant, à quel point *ma* thèse est aussi la *vôtre* : je vous laisse répondre à cette question, et peut-être que le futur me donnera aussi la réponse !

Un merci plus collectif à toute l'équipe TADaaM, la meilleure équipe de l'Inria Bordeaux ! Merci à Brice, François, Guillaume, Guillaume et Francieli pour hwloc, le droit des logiciels, la théorie qui m'est imperméable, le standard MPI, les I/Os, le Brésil, les Ardennes, le baby-foot, le militantisme syndical, les râleries sur les étudiants, la médiation scientifique, les fous-rires mais aussi le sérieux quand il s'agit de travailler (un petit jeu : saurez-vous relier chaque terme au permanent correspondant ?). Ensuite, merci à l'open-space, tout d'abord pour m'avoir supporté, merci à ceux que j'appelle mes grands-frères de thèse : Valentin, Andrès, Nicolas et Florian. Votre parcours durant la thèse m'a aidé d'une

manière ou d'une autre dans ma thèse à moi. Après les grands-frères, les petits-frères de thèse : Alexis, Clément, Robin, Julien et Richard. J'espère (humblement) aussi avoir réussi à vous transmettre ce que mes grands-frères m'ont transmis. Bon courage pour la suite de vos thèses et profitez bien, la fin et l'après-thèse ne sont pas vraiment les périodes les plus agréables... Merci aux autres qui sont passés par l'open-space, quelque soit leur statut : Luan, Clément, Clément, Adrien, Valentin, Corentin et Amaury. Merci aux intrus de l'open-space, Luc et Romain, qui n'étaient pas de TADaaM et qui couvraient les tableaux de signes bizarres ! Un merci particulier à Alexis et Luan, comme fidèles compagnons pour la découverte par deux fois des États-Unis. Pour finir avec les membres de l'équipe, un grand merci à Catherine, qui, avec une grande gentillesse et une patience à toute épreuve, a toujours su m'assister dans toutes mes démarches administratives.

Merci aux autres équipes HPC du centre, Storm et HiePACS. J'ai tellement travaillé et discuté avec les membres de Storm, que j'avais l'habitude de dire que j'étais à moitié de membre aussi de Storm ! Merci en particulier à Samuel et Nathalie pour les discussions techniques (ou pas) et à Mathieu, Emmanuel et Pierre pour, entre autres, leurs explications des algorithmes d'algèbre linéaire.

Merci ensuite à tout ce qui gravite autour de la recherche : le SED pour les discussions techniques (ou pas), le SCM pour la médiation scientifique (mais pas que), les enseignements à l'Enseirb-Matmeca (merci à, entre autres, David et Mathieu), Ludovic pour Guix et les notions de recherche reproductible, le monde du libre en général et tous ceux qui y contribuent. Et parce que c'est presque le seul outil dont on a besoin pour faire de la recherche en HPC, merci aux plateformes de calculs, en particulier les Dalton et PlaFRIM. Merci aux personnes qui les administrent, ça fait toujours plaisir de pouvoir discuter de vive voix de ce qui fonctionne et ce qui ne fonctionne pas !

Pas merci au Covid-19, qui nous aura forcé au télétravail une bonne partie de la thèse et nous aura privé de ~~séjours touristiques~~ conférences à Varsovie, Lisbonne, Bonn, Lyon, Seattle, ou encore à Chicago. Difficile de dire quelle voie aurait pris cette thèse sans cette pandémie...

Merci à ma famille pour son soutien indéfectible, sans jamais poser de questions, alors même qu'elle ne comprend pas un mot de mes sujets de recherche (et ce n'est pas faute d'avoir essayé de l'expliquer) ! Et merci à Hélène pour s'être gentiment proposée pour faire le dessert de mon pot de thèse.

Pour finir, merci à toutes celles et ceux que j'ai pu oublier, mais qui ont été près ou loin de moi lors de cette thèse.

Résumé étendu en français

Les problèmes numériques complexes (comme les simulations, les prévisions météorologiques, la climatologie, la cosmologie, la biologie, la chimie, les phénomènes physiques, *etc*) font parties des applications ciblées par le calcul haute-performance (en anglais *High Performance Computing* – HPC). Ces applications demandent généralement une puissance de calcul importante ainsi qu'une grande quantité de mémoire pour être exécutées, les rendant hors de portée des ordinateurs classiques. À la place, ces applications sont exécutées sur des *supercalculateurs* qui sont un ensemble d'ordinateurs (appelés *nœuds*) individuellement très puissants, inter-connectés ensemble par un réseau rapide. Les programmes peuvent alors faire des calculs sur plusieurs nœuds simultanément, permettant d'agréger la puissance de calcul de ces nœuds. Dans ce cas, une bibliothèque logicielle de communication réseau s'occupe de déplacer les données entre les différents nœuds, par exemple lorsque le résultat d'un calcul intermédiaire effectué sur un certain nœud est nécessaire sur un autre nœud pour lancer un autre calcul. Concernant les nœuds de calcul, leur conception ainsi que leur utilisation se sont complexifiées au fil du temps, en même temps que leur puissance de calcul augmentait. De nos jours, les nœuds de calcul équipant les supercalculateurs ont chacun des unités de calcul de différents types (CPU, GPU, FPGA, *etc*) qui ne se programment pas de la même manière et qui sont plus ou moins efficaces selon le type d'instructions qu'ils doivent exécuter. Cela rend la conception d'applications plus complexe, puisqu'il faut minutieusement choisir quelle unité de calcul va exécuter quelle opération. De plus, cette décision impacte également les transferts de données nécessaires entre les différentes mémoires disponibles au sein d'un nœud. Mal gérés, ces transferts mémoires peuvent être très lents et devenir un facteur limitant de l'application. Pour résumer, l'hétérogénéité présente dans les nœuds de calcul actuels rend difficile l'utilisation efficace de *toute* la puissance de calcul qu'un nœud est (théoriquement) capable de fournir.

Face à ces difficultés, les *support d'exécutions à tâches* (en anglais *task-based runtime systems*) connaissent un bel essor. Leur modèle de programmation repose sur la représentation des applications par un graphe de tâches : chaque opération de l'application est représentée par une tâche qui est un sommet du graphe. Chaque opération produit des données, qui peuvent être utilisées en entrée d'autres opérations. Ces dépendances entre tâches sont les arêtes du graphe de tâches. Pour chaque tâche, les instructions à effectuer sur chaque unité de calcul ciblée doivent être fournies, ainsi que les dépendances entre les tâches. Le support d'exécution se charge ensuite du reste : ordonnancer les tâches

sur les unités de calcul, faire les transferts de données nécessaires, exécuter les tâches, ... L'écriture d'application parallèle est facilitée avec le modèle de programmation à tâches, puisque c'est le support d'exécution qui infère le parallélisme de l'application, à partir des dépendances entre tâches qui forment le graphe de tâches. Dans le cas d'applications distribuées (utilisant plusieurs nœuds), le support d'exécution à tâches peut également découvrir et s'occuper des communications réseau nécessaires, mais en déléguant généralement la réalisation de ces communications à une bibliothèque tierce.

Habituellement, les supports d'exécution et les bibliothèques de communications sont deux briques logicielles bien distinctes. Le support d'exécution utilise l'interface de la bibliothèque de communication, et cette dernière se contente de traiter les requêtes qui lui sont adressées : elle n'a pas d'informations particulières sur l'application exécutée, ni sur l'état courant du support d'exécution. Cependant, le support d'exécution peut avoir des informations qui permettraient d'aider la bibliothèque de communication dans ses prises de décisions (les futures communications à exécuter, les priorités des tâches, le chemin critique de l'application, *etc*). De la même façon, la bibliothèque de communications possède une vision sur le réseau et les communications en cours, et certaines informations pourraient être utiles au support d'exécution (par exemple : les messages reçus incessamment sous peu, une estimation de la durée des communications).

Le but de cette thèse est d'**explorer les interactions possibles entre les supports d'exécutions à tâches et les bibliothèques de communications**, en échangeant plus d'informations entre ces deux couches logicielles, afin d'améliorer les décisions de l'un et l'autre et finalement augmenter les performances des applications.

Ce manuscrit présente les contributions réalisées en ce sens pendant trois années de thèse. Tout d'abord, une évaluation et amélioration des outils de traces pour analyser l'exécution des applications à base de tâches (et ainsi mieux comprendre les interactions entre le support d'exécution et les communications) ont été réalisées. Une première interaction positive entre supports d'exécution à tâches et bibliothèque de communications a été proposée en implémentant une solution pour être capable d'envoyer efficacement une même donnée à plusieurs nœuds différents. Nous avons également étudié les interactions négatives, en évaluant les différentes sources d'interférences possibles entre calculs et communications, lorsqu'ils sont exécutés en parallèle, comme c'est le cas dans de nombreux supports d'exécutions à tâches. Puisqu'il s'est avéré que la contention mémoire entre les accès mémoires pour les calculs et les communications est la plus grande source d'interférences pénalisant leurs performances respectives, nous avons proposé un modèle pour prédire la bande-passante mémoire accordée à chaque type de flux (calculs ou communications). Ce modèle nous a permis de mieux comprendre le fonctionnement de la mémoire en cas de contention et de pouvoir prédire les performances des calculs et des communications. La suite de ce résumé détaille ces différentes contributions.

Évaluation et amélioration du système de traces

Pour comprendre les performances et le comportement des applications, tracer leurs exécutions et analyser les détails du déroulement de l'exécution peut être une méthode très efficace. Pour être suffisamment robustes, les systèmes de traces doivent avoir une préci-

sion satisfaisante et interférer au minimum avec l'exécution de l'application tracée. Avoir un impact sur l'application tracée peut changer son comportement, ce qui signifie que l'exécution décrite dans les traces est différente de l'exécution standard (non tracée). Ce phénomène peut être gênant, puisque les traces servent généralement à comprendre ce qui se passe lorsqu'une application n'est pas tracée ! Un système de traces pas assez précis, avec des horloges mal synchronisées, produira des traces incohérentes, notamment concernant les exécutions distribuées (par exemple : un message peut apparaître comme reçu avant d'avoir été envoyé).

Nous avons évalué les différentes sources possibles de surcoût en terme de performances de l'application, pouvant changer le comportement d'une application, lorsqu'une application utilisant un support d'exécution à tâches est tracée. Nous avons également proposé des solutions pour éviter (ou au moins réduire) ces surcoûts. De plus, nous avons implémenté dans le système de traces du support d'exécution des techniques de synchronisation d'horloges correspondant à l'état de l'art, et nous avons évalué l'amélioration de la précision des traces ainsi obtenues.

En plus de proposer des solutions aux problèmes causés par les systèmes de traces, notre travail avait aussi pour but de faire prendre conscience aux personnes utilisant ces systèmes de traces les potentiels problèmes qui peuvent survenir lors de la trace de programmes, potentiellement déformant la réalité.

Ce travail peut être perçu comme un pré-requis pour analyser sereinement les exécutions d'applications, tout en ayant en tête les possibles problèmes.

Broadcasts dynamiques

Les communications sont l'un des facteurs limitant pour faire passer les applications à l'échelle sur de nombreux nœuds. Un motif de communications qui peut facilement s'optimiser à l'aide d'algorithmes de routages déjà présents dans la littérature, et qui se retrouve dans les graphes de tâches de certaines applications, est l'envoi de la même donnée à plusieurs nœuds distincts. Ce motif s'appelle un *broadcast*.

Les bibliothèques de communications habituelles en HPC proposent des routines pour exécuter des broadcasts d'une façon optimale. Cependant, plusieurs critères doivent être remplis pour pouvoir utiliser ces fonctions : par exemple, tous les nœuds impliqués dans le broadcast doivent faire appel à la même fonction avec les mêmes paramètres, en particulier la liste de tous les destinataires de la donnée. De plus, ce genre de fonction introduit une sorte de synchronisation, qui, pour des raisons de performances, doit être évitée autant que possible dans les supports d'exécution à tâches. Malheureusement, ces contraintes ne sont pas satisfaites dans le contexte du support d'exécution considéré : les broadcasts n'y sont pas explicites (ils doivent être inférés à partir du graphe de tâches) et seulement l'émetteur du broadcast connaît tous les destinataires. Les destinataires, quant à eux, ne savent même pas s'ils vont recevoir le message par un broadcast ou par une communication point-à-point classique (et donc ne savent pas quelle fonction appeler).

Pour pouvoir profiter d'algorithmes de routages efficaces, tout en respectant les contraintes du support d'exécution à tâches, nous avons proposé des *broadcasts dyna-*

miques. Le mécanisme pour détecter les broadcasts a une fiabilité satisfaisante, et une fois qu'une donnée à envoyer par un broadcast est disponible, l'interface que nous avons développée dans la bibliothèque de communication le prend en charge de façon transparente pour le support d'exécution : la donnée venant d'un broadcast est reçue comme si elle était reçue par une communication point-à-point.

Des microbenchmarks ont montré l'efficacité de notre implémentation. Les gains de performances pour les applications avec des broadcasts dans leurs graphes de tâches dépendent de plusieurs facteurs. Les broadcasts dynamiques ont permis d'améliorer les performances de 30 % sur des factorisations de CHOLESKY et de multiplier par 6 les performances de factorisations QR sur des matrices avec une forme et une distribution spécifiques.

La pertinence des broadcasts dynamiques montre le potentiel des interactions entre les supports d'exécution à tâches et les bibliothèques de communications : à l'aide d'une interface simple et générique exposée par la bibliothèque de communications, le support d'exécution peut exécuter des broadcasts plus efficacement.

Interférences entre calculs et communications

La plupart des supports d'exécutions à tâches permettent d'exécuter en parallèle calculs et communications. Puisque cela signifie exécuter simultanément des calculs et des communications qui partagent ressources matérielles communes, nous avons évalué les possibilités d'interférences entre calculs et communications, impactant leurs performances respectives.

Les variations de fréquences causées par les calculs n'ont pas d'impact majeur sur les performances des communications. Les communications lancées par le support d'exécution peuvent être pénalisées par un surcoût en latence important, à cause de la pile d'appel de fonctions à traverser avant d'atteindre la bibliothèque de communications. Mais la dégradation de performances la plus importante, lorsque calculs et communications sont exécutés en parallèle, provient de la contention mémoire entre les mouvements de données pour les calculs et pour les communications.

La contention mémoire peut être influencée par plusieurs facteurs : le placement des données et des threads, la taille des messages et l'intensité arithmétique des calculs. Lorsque de la contention mémoire se produit, les performances des calculs peuvent être impactées, mais ce sont les communications qui sont le plus pénalisées.

Pour mieux comprendre la contention mémoire se produisant entre calculs et communications et être capable de la prédire, nous avons proposé un modèle de ce phénomène donnant le débit mémoire accordé aux calculs et aux communications, selon le nombre de cœurs exécutant des calculs et le placement des données. La difficulté de conception du modèle venait du fait que la gestion de la contention par les composants mémoire des processeurs est un secret bien gardé des fabriquants. L'évaluation de notre modèle sur une large gamme de machines aux caractéristiques différentes ont confirmé nos hypothèses initiales : tant qu'il n'y a pas de contention, calculs et communications obtiennent la bande-passante mémoire qu'ils requièrent ; lorsque la capacité du bus mémoire est atteinte, la bande-passante pour les communications est d'abord réduite, pour préserver

celle des cœurs qui calculent ; une bande-passante minimale est tout de même toujours assurée pour les communications, pour éviter les famines ; finalement, si la demande en débit mémoire continue d'augmenter, alors les cœurs de calcul sont également pénalisés.

Bien que ces interactions négatives entre calculs et communications apparaissent dans tout programme qui exécute simultanément des calculs et des communications, quel que soit le support d'exécution utilisé, les possibilités et l'abstraction offertes par les supports d'exécution à tâches devraient permettre de prendre en compte ces phénomènes et essayer de les éviter. Cette prise en compte de la contention entre calcul et communications par le support d'exécution serait un autre cas d'interaction positive.

Conclusion

Les supports d'exécution à tâches sont une solution pour tirer plus facilement profit de la puissance des supercalculateurs, en fournissant un haut niveau d'abstraction, allant jusqu'à inférer le parallélisme des applications et les communications réseaux nécessaires.

Cette thèse aura montré que les interactions entre les supports d'exécutions à tâches et les bibliothèques de communications, qu'elles soient positives ou négatives, ne doivent pas être négligées. Dans un cas, elles ouvrent la voie à de considérables améliorations des performances, mais dans l'autre, elles peuvent aussi pénaliser la vitesse d'exécution des applications. Il n'est donc pas possible d'ignorer les communications au niveau des supports d'exécutions à tâches.

Introduction

High Performance Computing

BETWEEN theoretical and experimental work, simulation can be considered as the third pillar of current scientific research. Indeed, simulations allow to verify theories and avoid real experiments, which can be costly and dangerous. Simulation can be used in many research areas: weather forecasting, climate prediction, fluid dynamics, earthquakes, crash-tests for vehicles, rocket takeoff, biology, genomics, epidemiology, nuclear phenomena, cosmology, *etc.* Many cars can be saved while perfecting the airbag, as well as many rockets while the engine is not finely tuned; phenomena hard to mimic at the human scale, from the atom to the universe scale; dangerous experiments about nuclear reactions or health can be performed; all thanks to simulation.

Simulations are usually performed with computer programs, solving numerous complex equations, tracking the progression of a phenomenon over time steps, handling huge amount of data. These specific programs often require lots of computing power and memory to achieve precise simulations in a reasonable amount of time. More powerful computers allow to increase simulation precision (more precise weather forecasting, for instance) and to treat bigger problems. Using powerful computers to execute simulation programs belong to the High Performance Computing (HPC) domain.

The computing power required for simulations is so important that specific computers are built for this purpose. The most powerful of them are listed twice a year by the TOP500 ranking. In June 2022, the FRONTIER machine holds the first position[1], with a performance of 1.102 Eflops: about 10^{18} floating operations per second, while a regular laptop treats about 200 Gflops (*i.e.* FRONTIER is equivalent to 5 million laptops!). Such powerful computers aimed at HPC are called *supercomputers* and are in fact composed of many inter-connected *nodes*: individual computers which can be compared to more regular servers. This connection of many nodes gives to the overall supercomputer an important computing power: FRONTIER is composed of 9 472 nodes. Moreover, each node has a processor with 64 cores and 4 GPUs. In the end, FRONTIER totals 8 730 112 computing units executing instructions in parallel.

These supercomputers can feature cutting-edge technology and the programming of

[1] https://top500.org/news/ornls-frontier-first-to-break-the-exaflop-ceiling/

applications dedicated to HPC machines can be quite challenging: applications have to be parallel (*i.e.* divided in smaller independent parts executed simultaneously) to reduce execution time and fully exploit the computing power of the machine, they have to support different computing units (CPUs and accelerators), correctly manage different memories, *etc*. With the increasing power and complexity of supercomputers, fully exploiting all the computing power becomes more and more difficult. Moreover, each supercomputer is differently designed, which requires abstractions to have performance portability of applications.

To address these difficulties, *runtime systems* are developed, to abstract part of the machine complexity: machine topology, scheduling, accelerator management, *etc*. Different kinds of runtime systems exist, with different features and abstraction levels, like abstracting the machine but letting the developer explicitly express the application parallelism. Runtime systems following the task-based programming model with data dependencies abstract even this level and ease the writing of parallel applications, by requiring only an implicit expression of the parallelism. With task-based runtime systems, the application is divided in small parts represented by tasks, and tasks are connected according to the dependencies between those tasks (*i.e.* order constraints based on data manipulated by each task). Tasks and dependencies form a graph, which, once given to the runtime system, can be efficiently scheduled on parallel computing units.

Specific software libraries (sometimes part of runtime systems) are also developed to communicate between the nodes, by using the high performance network interconnecting them. Exchanged data are mainly results produced by a node, required by another node to perform a computation. Combining several runtime systems and a third-party communication library is common for HPC applications.

Goals and contributions of this thesis

Usually runtime systems and communication libraries are two distinct and independent software. The runtime system uses the interface provided by the communication library to exchange messages on the network. The communication library is aware only of information related to network communications, and knows nothing about the application nor the status of the runtime system. However, a task-based runtime system may have information that could help the communication library in its decisions (future communications, priorities, application critical path, *etc*). Conversely, the communication library is aware of the status of communications and could also share its knowledge (incoming communications, estimated time of reception, *etc*) with the task-based runtime system.

The goal of this thesis is to explore the possible interactions between task-based runtime systems and communication libraries, in order for these two software layers to better exchange their respective information about the application execution and increase each other knowledge to take better decisions.

The main two software libraries used in this work to implement prototypes and make experiments are StarPU and NewMadeleine, introduced in Chapter 1.

This manuscript presents the following contributions, made during these last three

years:

Tracing systems. Recording application executions permits to later analyze the execution and understand the behaviour of the application, the decision taken by the runtime system, *etc*. However, tracing executions can add a performance overhead, potentially changing the behaviour of the application, making the behaviour represented in traces different than the behaviour in normal conditions, without traces. We evaluated the impact of different sources of overhead and proposed solutions to reduce them. Moreover, when tracing distributed applications, clocks used to timestamp events on each node have to be accurately synchronized to have consistent information recorded in traces, especially regarding communications. We evaluated the impact of clock synchronization on accuracy of communication duration.

Findings in this work are submitted to the journal *Concurrency and Computation: Practice and Experience* [134].

Dynamic broadcasts. Some task-based algorithms require to send the same data to different nodes. While common communication libraries provide routines for such use-case, with optimized communication scheme, these routines cannot be used within StarPU because of (among other things) a lack of information by all nodes receiving the data. We proposed a solution to use optimal routing algorithms for such communication patterns that fits with StarPU's constraints.

The idea, its implementation and its evaluation were presented in an article published at the 26^{th} *Euro-Par* conference [130].

Interferences between computations and communications. Many runtime systems allow to execute simultaneously computations and network communications. Since these two different operations share common resources, interferences between them can occur, impacting their respective performance. We evaluated several possible sources of interferences and measured the impact on performance of both computations and communications.

Results of this study were published in an article at the *ICPP 2021* conference [128].

Model of memory contention. The major source of interferences between computations and communications is the possible memory contention generated by data movements to perform computations and communications. We proposed a model to better understand the contention and to be able to predict the share of memory bandwidth between computations and communications.

Our model and its evaluation got the best paper award of the *APDCM 2022* workshop, in conjunction with the 36^{th} *IPDPS* conference [129]. A research report contains results on more machines [135].

Organization of the document

Chapters 1 and 2 present the context of this work by respectively introducing distributed task-based runtime systems and reviewing related work. Chapter 3 digs in the tracing

system of StarPU, evaluates possible performance overheads when tracing applications and proposes solutions to reduce these overheads and to improve accuracy of distributed traces. Chapter 4 explains how we developed a broadcast system to fit with StarPU's constraints while using an optimized communication pattern. Chapter 5 studies the possible interferences between computations and network communications, when they are executed in parallel. Chapter 6 focuses on the impact of memory contention on computations and communications, by proposing a model for memory bandwidth sharing. Finally, last chapter summarizes our work and discusses possible perspectives.

Appendix A explains in detail differences between the communication backends of StarPU. Appendix B presents the followed methodology regarding reproducible experiments. Appendix C describes the characteristics of the clusters used for the experiments presented in this thesis. Appendices D and E provide respectively algorithmic versions of equations described in Chapter 6 and the parameter values obtained during the evaluation of the model.

Chapter 1

Distributed Task-based Runtime Systems

This first chapter presents the context and the problematic of this thesis. First, it describes the increasing complexity of HPC supercomputers, which makes them harder to program in the most efficient way. Then, we introduce task-based runtime systems, a solution to address these programming difficulties. We explain how they work on a single node but also for distributed applications on several nodes. Finally, we announce the general problematic covered by this thesis and its main contributions.

1.1 The growing complexity of HPC machines

Although we reached some physical limits in the conception of processors, preventing from increasing always the same characteristics (*e.g.* processor frequency and Instructions Per Cycle (IPC)), performance of supercomputers did not stop to increase over time. Machine vendors keep innovating, in several directions, to provide always more and more computing power to their users. The diversity of solutions for real high performance computing has a cost: efficiently exploiting the whole computing power offered by the machines is much more difficult, and, moreover, it is a challenge left to application developers.

1.1.1 More powerful machines...

At the core of computers, processors execute all the instructions. To offer better performances and better exploit their possibilities, they present many features, more or less complex to take into account by the application:

Multi-core processors. All available transistors in a processor can be used to split the processor into several processing units (called *cores*), independently executing different instructions at the same time. Nowadays, HPC processors feature several tens of such cores, each executing its own flow of instructions. From the operating system point of view, the different cores appear like different processors.

Simultaneous multithreading. Since some processor instructions require several clock cycles to complete (floatting-point instructions, memory accesses, *etc*), the progress of the instructions in the pipeline is not assured for each clock cycle. When such long instruction is executed, the core has to wait for the dedicated subsystem to handle the instruction, and thus waste clock cycles. These holes in the instruction pipeline are called *stalls* (or *bubbles*). With *simultaneous multithreading*, physical cores are split in several (usually two) logical cores: when a stall occurs in the instruction flow of a logical core, the physical core executes instructions from another logical core. INTEL implements this technique under the name of *Hyper-Threading*, which became a common term to designate this feature, as well as *hyper-thread* to talk about a logical core. From the operating system point of view, these logical cores appear as regular cores.

Many-core processors. Another way of exploiting the transistors of a processor is to split the processor into *many cores*. Like multi-core processors, each core manages its own instruction flow, but many-core processors feature much more cores: several hundreds of hyperthreads. This important number of cores is possible at the cost of simpler cores, with a reduced instruction set and less features. The main representative of this class of processors is the INTEL XEON PHI, launched in 2010 and installed for example in the TIANHE-2 supercomputer, first in the TOP500 ranking in 2013.

Heterogeneous processors. More recently, *heterogeneous* processors have been introduced, especially to reduce the power consumption: a set of cores requiring very few energy is used for generic purpose and another set of cores more powerful, yet consuming more power, is enabled only in case of more demanding computations. The main processor architectures in this family are the ARM's BIG.LITTLE and the INTEL's ALDER LAKE. However, these models have not landed yet on supercomputers.

Instruction set. The important number of available transistors can be used to create specific processor instructions, for instance implementing arithmetic operation directly as one processor instruction, instead of several instructions called by the software. This kind of optimization can save clock cycles and thus increase the computing power of the processor. Vectorized instructions are also improved over the years: they consist in executing the same operation on multiple registers at the same time (Single Instruction on Multiple Data (SIMD)). The *vectorization* of the code can be done implicitly by the compiler, or explicitly by the programmer by using the dedicated instructions. The most famous SIMD instruction set is the Advanced Vector Extensions (AVX) from INTEL. The last version, AVX-512, released in 2017 on the SKYLAKE processors, handles 512-bit registers (which can be used as arrays of 16 `float`s, for instance).

GPUs. While CPUs (*Central Processing Unit*) are designed for a generic purpose (running the operating system, softwares) and have to support a wide range of instructions, including branchings and loops; GPUs (*Graphics Processing Unit*) implement mainly the SIMD architecture, thus they can exhibit a high level of parallelism, much higher than regular CPUs, but have very low performance in case of branching instructions. This specific kind of processor was originally designed for graphic processing such as image rendering, and then other applications with similar requirements (*e.g.* linear algebra and training of machine learning models) adopted it.

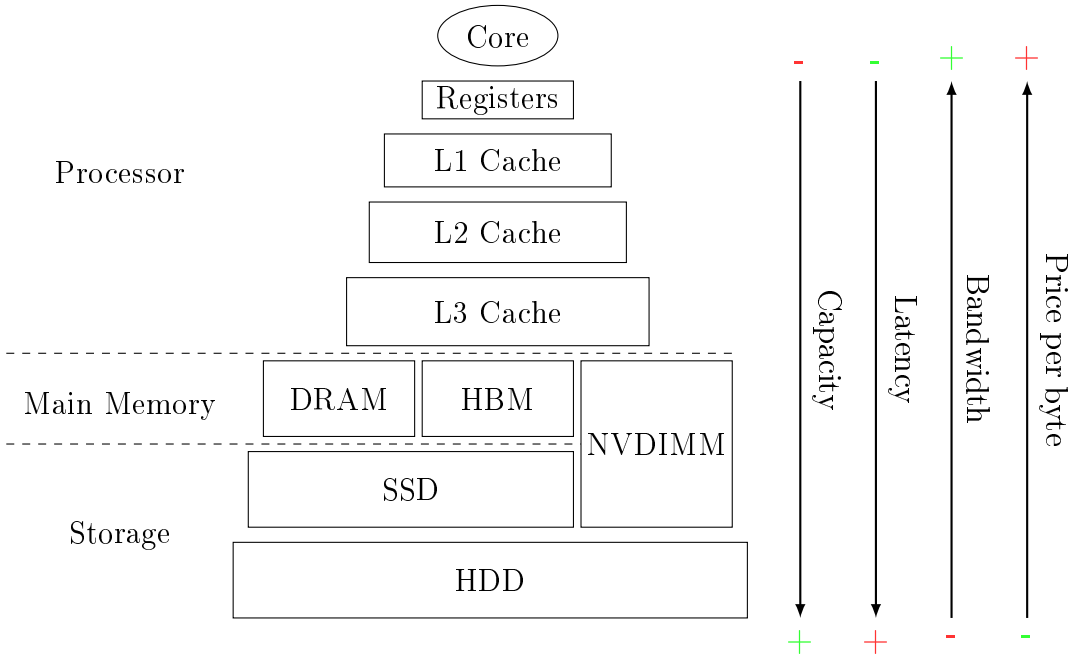


Figure 1.1: Typical memory hierarchy.

FPGAs. *Field-Programmable Gate Arrays* are processors that can be entirely reprogrammed to execute a specific computation. This way, computations can be faster on an FGPA than on a regular CPU, since the circuit of the FGPA is reprogrammed specifically to execute only the desired computation, while the genericity of the CPU can add important overhead. The main advantage of the FPGA is also its main drawback: the programming of the FPGA is made in a specific (low-level) language and the reconfiguration of the FPGA to execute a new program can be very long (several hundreds of milliseconds). Such processors also tend to cost more than CPUs and have a lower frequency.

All these different uses of transistors rely on the *parallel* paradigm: applications are divided in smaller independent instructions flows, each being executed simultaneously by different sets of transistors (*e.g.* several cores or hyper-threads). Parallel applications are the most common in HPC area, since this programming model is required to exploit current powerful computers, featuring these parallel computing units.

Many-core processors, GPUs and FPGAs are examples of *accelerators*: they are installed in addition to a CPU, to accelerate only the kind of computations they are specialized in. While CPUs have direct access to the main memory (RAM) of the computer, accelerators have their own memory. Some accelerators can access the CPU RAM, but with a potential performance overhead if used improperly[1]; a technique to counterbalance this overhead is to explicitly copy data to the accelerator memory, perform the maximum computations on the accelerator manipulating the copied data (now located in the accelerator memory, with quick access) and then copy back the data on the CPU memory.

[1] https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

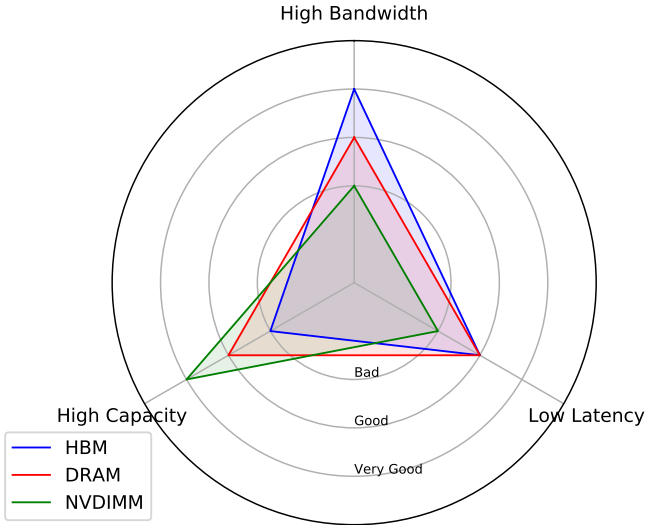


Figure 1.2: Characteristics of different memory kinds. From the PhD thesis of Andrès Rubio Proaño [98].

Speaking of memory, memories accessible by a CPU bring also their set of complexity. Computers feature a so-called *memory hierarchy*, as illustrated by Figure 1.1. Indeed, processors can access several kinds of memory, more or less close to each core. The further the memory is located (from an architectural point of view), the longer it will take to access this memory (to read or write), however, the higher the memory capacity. This hierarchy comes from the complexity (or even impossibility!) to design the perfect memory: fast, with high capacity and cheap: thus faster memories are smaller and closer to computing units, to take benefit from an increased speed.

We can distinguish memory inside and outside the processor. Memory inside the processor are caches, organized in several levels: first levels are private to each core and last levels are shared between cores. The most common memory outside processors is Dynamic Random Access Memory (DRAM). Processors fetch memory from the DRAM in caches and cache lines are evicted and stored back in the DRAM. DRAM memories provide good trade-off in terms of capacity, bandwidth and latency. Again, since there is no perfect memory with outstanding values in all performance metrics, other types of memories exist, each type being better regarding one characteristic, as illustrated by Figure 1.2: Non-Volatile Dual Inline Memory Module (NVDIMM) has a higher capacity and High Bandwidth Memory (HBM) a higher bandwidth.

With the current important number of cores per processor, and given the fact that HPC nodes can have several processors (usually two), the memory system cannot serve the memory requests of all cores and exhibit correct performance. To tackle this issue, a Non-Uniform Memory Access (NUMA) architecture is used: the whole memory is segmented and each segment is dedicated to a set of cores, as illustrated by Figure 1.3. A *NUMA node* is the name of the set composed of several cores and their dedicated segment of memory. Cores can normally access memory located on a NUMA node different from their NUMA node, but this kind of memory request will take longer to achieve.

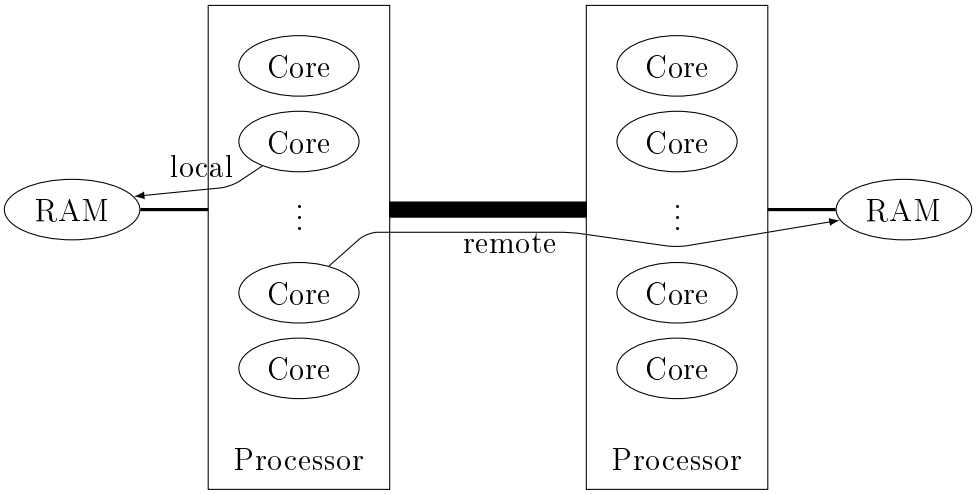


Figure 1.3: A NUMA machine: each processor has its own memory (*local*), but can also access memory of other processors (*remote*).

Current supercomputers feature several multi-core processors, NUMA architecture and GPUs. Since they have different kinds of processing units, they are qualified as *heterogeneous*. Many-core processors such as the Xeon Phi are less present in current supercomputers, while FPGAs and heterogeneous processors will probably be part of the future of HPC. Increasing performance of supercomputers will continue, by increasing also the complexity of their components. Unfortunately, this evolution is not transparent anymore for application developer...

1.1.2 ... but more complex to program!

The application developer has now access to powerful machines, with different kinds of computing units. The drawback of such wide panel of technologies is the difficulty of efficiently programming these platforms. Indeed, the challenges are numerous.

Computing unit programming. A CPU, a GPU or an FPGA are not programmed with the same source code, and the programming paradigms can be quite different.

Programs can be parallelized on CPU cores with several methods: processes, threads, a runtime system, *etc.* Programming application for CPUs can look simpler than for accelerators, since it does not require a specific language and the use of the standard library provided by the language can be enough. However, some points require particular attention, for instance: one core should not execute several threads, the load balancing between the cores as well as memory access patterns should be optimized (to efficiently use the cache, for instance), *etc.* In HPC applications, a popular method to easily parallelize code is the use of OpenMP pragmas: by annotating the source code, the developer can explain how the application should be parallelized.

Programming GPUs (and other accelerators) can be more complicated. Interacting with GPUs is made with specific APIs (Cuda for Nvidia, Hip for AMD) or even

specific languages: CUDA programs are written in a C-like language and are compiled with a specific compiler. Moreover, since the GPU has its own memory, the application developer has to manage the GPU access to application data.

One problem here is the portability: the application written to be executed on a CPU has to be adapted to use a GPU (and the changes will be different according to which GPU vendor is targeted).

To ease the writing of applications targeting heterogeneous systems, several frameworks implement a *write once, run everywhere* model: the program is written in one language (usually close to C) and the framework translates it to instructions for all required computing units. OPENCL [6], OPENACC [5] or SYCL [7] are examples of such frameworks.

Memory management. When using accelerators, memory transfers between the host and the device are explicit. These data transfers can be long compared to computations (limited by the throughput of the PCIe bus where they are plugged: 12 GB/s *vs* 20 GB/s for throughput between CPUs and RAM memory) and since the memory capacity is limited, data moved on accelerators have to be carefully chosen to then be able to execute as much computations as possible on this data without requiring additional data movements.

NUMA architectures has also to be taken into account for high performance. Since each core can access a close memory faster than a remote memory, cores should rather only access their local memory and minimize access to remote NUMA nodes. Accelerators are also affected by NUMA configurations: in the memory hierarchy of a machine, they are plugged into a specific NUMA node, hence the same performance consideration for CPU cores apply.

The organization of heterogeneous machines forces application developers to be careful about the *data locality*: on which memory node (NUMA or accelerator) data has to be allocated and then potentially moved. The decision is taken according to which computing unit will work with this data, but also according to memory performance and application requirements, when less standard memories (NVDIMM or HBM) are available.

Data management involving an accelerator with its own memory is explicit and has to be programmed by the application developer. Memory management on NUMA architectures can also be made explicitly by the developer, but it is not mandatory since all cores can access all the NUMA memory. Still, optimizing it can improve application performance. Some runtime systems can relieve developer's work by abstracting the memory management and handling all data transfers implicitly.

Exploiting accelerator affinities. Accelerators are usually efficient to execute a limited types of computations, not all possible kinds of computations. For instance, GPUs are very efficient for matrix multiplications (30 times faster than on a CPU core), but have more difficulties for matrix factorizations (*only* 3 times faster than on a CPU core). Thus, the choice of which computing unit (a CPU core or an accelerator) has to execute a computation is based also on the performance of computing units on the kind of the computation.

To help taking this kind of decision, performance calibrations and models can be used: computation kernels are executed on all possible computing unit to build a performance model and be able to decide where to execute which computation.

Scheduling. Executing applications ends by raising the question: on which computing unit execute which computations? In other words: how to schedule the computation tasks? All the difficulty of scheduling is to optimize the use of all computing units, to reduce the application execution time. The elements of the answer are, among other, the performance of computing units and the data locality. Many trade-offs appear, for instance: is it worth to spend time moving data to the GPU, even if computations will take less time on the GPU? Can CPU cores execute computations at the same time as the GPU? In which order execute computations to expand as much parallelism as possible, to always feed computing units with work?

This problem is actually NP-hard [121] and consists in a whole research field. Heuristics and algorithms are developed to minimize the application makespan.

In the end, the application developer has to consider many factors to efficiently exploit the whole computing power theoretically available in a machine. All these constraints make it difficult to manually optimize the code of an application and require often expert knowledge in different areas (programming, architecture, scheduling, the application domain, *etc*). Even if applications are successfully optimized for a supercomputer, the next question is the portability: manually optimized applications are usually tuned for a specific computer, with a specific configuration. What is the necessary effort to optimally execute the same application on a different configuration (*e.g.* change of GPU; instead of one GPU, now two GPUs are available)? Ideally, it should be free, to let the scientist focus on their research field instead of software development.

One solution to tackle these problems is to add an abstraction layer to represent the memory and the computing units, thus getting rid for the application developer of the hardware specificities. These abstractions are usually offered by *runtime systems*: the application developer interacts with the runtime system, and the runtime system takes all the burden of interacting with the hardware.

Unfortunately, the complexity of HPC systems is not restricted inside one HPC node: these nodes are linked together to be used together for *distributed* applications.

1.2 Distributed systems

Current supercomputers are composed of thousands of HPC nodes, linked together with a high-performance network. The individual nodes are not really different from regular servers dedicated to other purposes than HPC: they are optimized for computing performance and scientific applications, which are programmed to exploit the computing power: numerous CPU cores, accelerators, high memory capacity, accelerators, *etc*.

1.2.1 Motivation for distributed computing

HPC clusters are composed of a set of HPC nodes connected through a high-performance network. Applications are executed on several of these nodes and they exchange data (inputs and outputs of computations, results of intermediary computations, *etc*) between the cluster computers with the network. In addition to using a *shared memory* programming model (several cores share the same memory), applications use a *distributed memory* model: the total memory available for the whole application is distributed among several computers.

Moreover, distributed computing allows to speed overall application duration up, since more computing cores are available. The distributed memory provides more memory to the application, to work simultaneously on more data, thus making applications able to work on bigger problems, whose limiting factor is the memory capacity.

However, distributed computing comes with few drawbacks. First, network communications are much longer than local memory accesses ($\sim 1\,\mu$s *vs* ~ 100 ns). Hence the gain of having more computing cores can be offset by the communication overhead. Nowadays, communications are considered as one of the major performance bottleneck, especially when it comes to application scalability on many nodes. Then, applications (and even algorithms) have to be redesigned to deal with the distributed memory: how to distribute data and computations among the nodes, how and when to communicate between nodes, *etc*.

These difficulties can prevent perfect scaling: doubling the number of nodes used by an application may not exactly double its performance, it depends on the application behaviour.

1.2.2 Environment for distributed systems

From the hardware side, high performance network systems are composed of special network adapters, plugged as a PCIe extension card.

The major manufacturers of such network interfaces are Mellanox, bought by Nvidia in 2020, implementing the InifiniBand standard; Cornelis Networks, a spin-off company from Intel, providing Omni-Path networks; Atos with the BXI interconnect; and Cray, bought in 2019 by HPE, with the Slingshot network. All these network devices exhibit low latency ($\sim 1\,\mu$s *vs* $\mathcal{O}(10)\,\mu$s for Ethernet networks) and high bandwidth ($\mathcal{O}(100)$ Gb/s *vs* usually 1 Gb/s or 10 Gb/s for Ethernet). Some HPC clusters are also equipped with RDMA over Converged Ethernet (RoCE) networks.

Even if it is usually possible to use this kind of network interfaces with the IP protocol, these interfaces are programmed from the user-space, *i.e.* the user can directly send instructions to the device, without involving the OS kernel. This allows to save the cost of user/kernel mode switch, which leads to a better latency. For bigger message sizes, transfers are made with zero-copy: network interfaces directly access the memory location where communication payload is stored (for send operations) or will be stored (for reception operations), without involving a processor to actually make the data transfer between the network interface and the RAM memory. This allows higher network bandwidth.

The programming of these network interfaces can be very low-level and specific to each vendor. To simplify the writing of distributed HPC applications, the *de facto* standard that emerged is Message Passing Interface (MPI) [51]. As its title states, this standard defines a set of functions to exchange messages between processes. The first version was released in 1994 and the current version 4 was released in 2021. In an MPI program, each process has a *rank* and each process decides which instructions to execute based on its rank[2]. There is at least one MPI process per node. More processes per node can be used to exploit the machine architecture: for instance, one MPI process per NUMA node. Then simple functions are available to send and receive messages between processes, by providing the memory buffer to send / to receive in, the size of this buffer and the ranks of sender and receiver processes. The following code snippet executes a *ring*: a value is passed from a process to another, and each process increments the value before sending it to the MPI process with immediate higher rank value.

```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nb_nodes); // let's assume nb_nodes > 1

if (rank == 0)
{
  dest = 1; // to right node
  source = nb_nodes-1; // from last node
  buffer = 0;

  MPI_Send(&buffer, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
  MPI_Recv(&buffer, 1, MPI_INT, source, 0, MPI_COMM_WORLD, NULL);

  printf("Ring completed [%d]\n", buffer);
}
else
{
  source = (rank - 1) % nb_nodes; // from left node
  dest = (rank + 1) % nb_nodes; // to right node

  MPI_Recv(&buffer, 1, MPI_INT, source, 0, MPI_COMM_WORLD, NULL);
  buffer++; // work
  MPI_Send(&buffer, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);

}

MPI_Finalize();
```

The interface proposed by the MPI standard is implemented in several available libraries. OpenMPI [52] is the most popular one. Other implementations exist, such as MPICH [3], MVAPICH [4], Intel MPI [2] or MPC [92].

[2]More precisely, a process has a *rank* inside an MPI *communicator* (a group of processes), but the special communicator `MPI_COMM_WORLD` contains all the processes. Thus, each process can be identified by its rank in this global communicator. To ease the reading, *rank of an MPI process* will refer to its rank in the `MPI_COMM_WORLD` communicator.

1.2.3 An HPC communication library: NewMadeleine

NewMadeleine [17] is a small HPC communication library, used as a research project to implement prototypes and experiments. NewMadeleine has its own end-user interface, but to support MPI applications, NewMadeleine exhibits also an interface called MadMPI, implementing the MPI standard.

In the interfaces presented to the end-user, primitives to perform communications manipulate *messages*. While mainstream MPI libraries watch and manage the network activity when the API is called from the user code, NewMadeleine decouples the network activity from the calls to the API by the user. This separation allows to add a layer applying an optimizing strategy before forming *packets* ready to be sent to the network. A *packet* may contain multiple *messages* (aggregation), a *message* may be split across multiple *packets* (multi-rail), and *messages* may be actually sent on the wire out-of-order depending on packet scheduler decision and priorities.

Moreover, the separation between network and application activities permits to make network communications *progress*, without any specific action form the application (such as loops over `MPI_Test` functions). This efficient background progression is achieved thanks to the PIOMan sub-project [44]. PIOMan allows also NewMadeleine to provide a native and efficient support for multi-threaded applications (the `MPI_THREAD_MULTIPLE` threading support level defined in the MPI standard; see section V.C of [44]), to be able to make calls to the library from different threads at the same time.

NewMadeleine is designed with an event-driven paradigm, especially, its core activity is triggered by the network. When the network is busy, *messages* to be sent are simply enqueued; when the network becomes ready, the optimization strategy is called to form a new *packet* from the pending *messages*. A receive is always posted to the driver, and all the activity is made of up-calls (event notifiers) triggered from the lowest layer when different events occur: first byte received, message fully received, completed reception, *etc.* These events can be hooked to execute a callback function defined by the application. This programming model makes NewMadeleine a library well-suited to program with the Remote Procedure Call (RPC) model (used later in this thesis): a node can execute a function on another node with parameters from the caller node.

1.3 Task-based runtime systems

Runtime systems are software layers abstracting the complexity of machines, developed to ease the writing of applications executed on these machines. There are many runtime systems, with different features, different levels of abstraction and based on different programming paradigms. In this thesis, we focus on the programming based on *tasks*. Other paradigms are mentioned in the next chapter.

1.3.1 General concepts

The task-based programming model consists in decomposing applications into small tasks and describing the dependencies between those tasks. Tasks are pure functions that are sub-parts of the whole applications. Dependencies between tasks represent the constraints in the execution order of tasks, and are determined with the data manipulated by the tasks, *e.g.* a task B which works on a result generated by a previous task A will impose a dependency from the task A to the task B. The set of tasks and their dependencies form a Directed Acyclic Graph (DAG): tasks are the graph nodes and dependencies between tasks are the edges. From this task graph, the runtime system can infer which tasks can be executed as soon as their dependencies have been satisfied (*i.e.* parent tasks in the task graph have already been executed), as well as which tasks can be executed in parallel, since they write in different data. If the runtime system knows the whole task graph, it is aware of the future tasks to execute and can optimize its scheduling decisions.

This programming paradigm based on a DAG allows to schedule computations (*i.e.* task executions) in a asynchronous manner: since the dependencies are known from the task-graph, the runtime system knows when tasks can be executed and does not require synchronizations (or *barriers*) to wait for the termination of tasks, potentially source of idle time of computing units, increasing the application duration. The knowledge of the DAG of the application also permits to apply scheduling algorithms to optimize the execution time.

To use a task-based runtime system, the application developer has to provide the task graph to the runtime system: the task themselves and the dependencies between them. A task is a structure describing the small piece of computation it executes. It contains a function actually executing the computations for each targeted architecture, with the specific instructions for each architecture: a function to be executed by a CPU, a function to be executed by a GPU, *etc*. Computations being made on data (at least, on memory buffers), a task describes also which memory buffers it manipulates, with which accesses: read-only, write-only or read-and-write. This information will later be useful to build the task graph: read buffers will create in-going edges and written buffers will create out-going edges.

Several programming models exist to instantiate the task graph of an application, to tell the runtime system which tasks have to be executed on which data. StarPU relies on the Sequential Task Flow (STF): by sequentially describing which tasks have to be executed with which data, the runtime system can infer the DAG. Indeed, for each *submitted* task, the runtime system knows which data will be manipulated by the task and by comparing with the data used (and their access mode) by the previously submitted tasks, it can detect if a data-dependency exists between the task and the previous tasks.

Once the application developer provided to the runtime system the task descriptions and the dependencies between them, the runtime systems is in charge of all the remaining work to actually execute the application. The main jobs (yet the more complex) are scheduling and executing tasks on computing units, and managing data transfers between memories. Basically, such a task-based runtime system does all the complicated work the application developer has to previously do manually to efficiently use their whole machine. The programmer can focus on the application side, and ignore the hardware problematic

managed by the runtime system[3].

Task-based runtime systems are widely used for linear algebra applications. One of the most common operation to illustrate working and performance of task-based runtime systems is the CHOLESKY factorization (or decomposition). For a given symmetric definite positive matrix A, the CHOLESKY algorithm computes a lower triangular matrix L, such that $A = LL^T$. Its main purpose is to solve linear systems. To easily parallelize this algorithm, the tiled flavour is used: the matrix is decomposed in $N \times N$ (usually square) *tiles* (sub-matrices, also called *blocks*) where $A[i][j]$ is the tile of row i and column j. The data manipulated by the tasks of the CHOLESKY factorization are the matrix tiles. Algorithm 1 depicts the tiled version of the CHOLESKY algorithm: at each step k, it performs a CHOLESKY factorization of the tile on the diagonal of panel k (`POTRF` kernel) then it updates the remaining of the tiles of the panel using triangular solve (`TRSM` kernel). The trailing sub-matrix is updated using the `SYRK` kernel for tiles on the diagonal and matrix multiply (`GEMM` kernel) for the remaining tiles.

Algorithm 1 Tiled version of the CHOLESKY factorization.

```
1:  for k = 0 to N − 1 do
2:      A[k][k] ← POTRF(A[k][k])
3:      for m = k + 1 to N − 1 do
4:          A[m][k] ← TRSM(A[m][k], A[k][k])
5:      end for
6:      for n = k + 1 to N − 1 do
7:          A[n][n] ← SYRK(A[n][n], A[n][k])
8:          for m = n + 1 to N − 1 do
9:              A[m][n] ← GEMM(A[m][n], A[m][k], A[n][k])
10:         end for
11:     end for
12: end for
```

Each task is *inserted* in the DAG during the execution of the nested loops. Each task indicates which tiles it requires. This presentation with a pseudo-language is not so far from what needs to be written in C when using STARPU, as will be seen later. For this algorithm, the application developer needs to provide the description of four tasks: `POTRF`, `TRSM`, `SYRK` and `GEMM`.

The task graph corresponding to the CHOLESKY factorization is depicted on Figure 1.4.

1.3.2 STARPU

STARPU is the task-based runtime system used in this thesis. Other task-based runtime systems exist, they are covered in the next chapter.

STARPU [16] is a task-based runtime system developed in Inria Bordeaux, by the STORM team. This C library was born as a proof-of-concept for the PhD thesis of Cédric

[3]Application developer still needs to give task functions for each targeted architecture. However, piloting the device to execute the function is left to the runtime system.

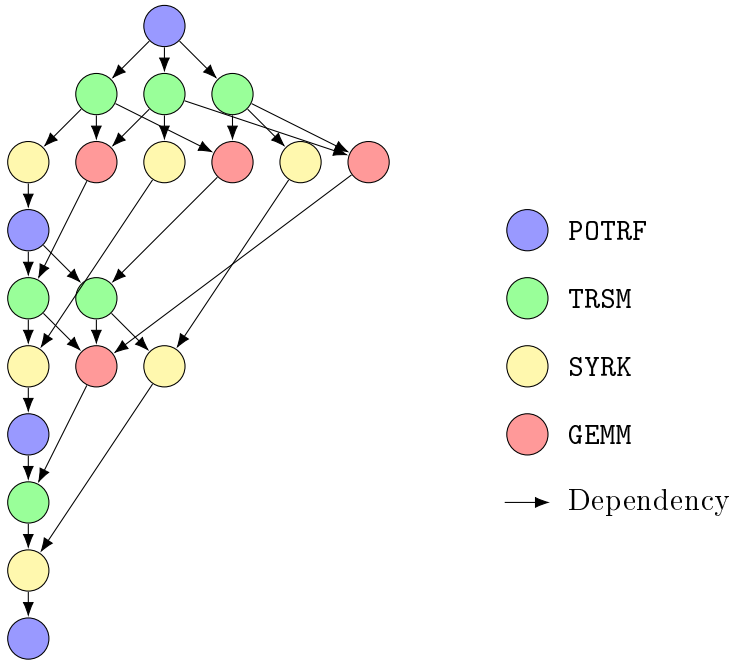


Figure 1.4: Task-graph of the CHOLESKY factorization for a matrix divided in 4×4 tiles.

AUGONNET about dynamic task scheduling on heterogeneous machines. The original goal was to execute tasks simultaneously on CPU cores and GPUs, in a portable manner, with the minimum of pain for the application developer. From the beginning of the development in September 2008, the project gained maturity over time to be used by several scientific applications[4] but also as a playground for research ideas and experimentation about scheduling policies, performance model and prediction, programming models, *etc*.

STARPU supports CPUs, GPUs (NVIDIA and AMD), INTEL XEON PHI and FPGA[5]. Supporting additional architectures consists of implementing a defined interface, with mainly instructions to launch computations on the device and make memory transfers. In the STARPU's jargon, each device being able to execute tasks (a CPU core, a GPU, *etc*) is called a *worker*.

The default scheduling policy of STARPU is *local work-stealing*, based on a list-scheduling policy: each worker has its own queue of tasks ready to be executed. When a worker finishes the execution of a task, it executes the next task in its queue. If its queue is empty, it tries to *steal* a task from another worker, but by respecting the machine hierarchy (*local* work-stealing): to efficiently take benefit from the caches, it first looks for tasks in the queues of other workers in its NUMA node, and only if it is unsuccessful it looks in the task queues of further workers. Other scheduling policies, used especially with accelerators, are based on performance models and implement a Minimum Completion Time (MCT) policy: STARPU can predict the duration of tasks and data transfers and takes this information into account for scheduling decisions to minimize the application

[4] See for instance https://starpu.gitlabpages.inria.fr/#software or https://starpu.gitlabpages.inria.fr/publications.html#PublicationsOnApplications.

[5] STARPU: *PU – STARPU aims at supporting any kind of PU!

duration.

StarPU abstracts data pointers by using *data handles*: each piece of data manipulated by a task is first registered by the user as a data handle: the user precises the size of the data, its structure (a simple variable, an array, a matrix, *etc*), its type (`float`, `double`, *etc*), and optionally an already allocated buffer (otherwise StarPU can allocate the buffer on-the-fly when required). Then, StarPU is free to manage the location(s) of the object referred to by this data handle: track its locality, duplicate the buffer on several NUMA node (thus the data is closer to more workers), move the data to accelerator memory, synchronize all replicates when one buffer is modified by a task, *etc*.

StarPU's tasks are instantiations of *codelets*. A codelet is a structure factorizing all properties common to all tasks executing the same computation: the functions to execute on workers (several functions can be provided for a same type of worker: in this case, the scheduler can chose the best function based on performance models), the number of data handles manipulated by the functions and their access modes, specific options, attributes to ease debugging (name, color, ...), *etc*. A task is an instantiation of a codelet: it references a codelet and stores information specific to this task, which will be executed only once: data handles which will provide data for the codelet functions, priority, callback functions to be executed before or after the task execution, *etc*.

The Cholesky algorithm implemented with StarPU can look like the following code snippet:

```
/* Data registration and codelet definitions not shown. */

for (k = 0; k < N; k++)
{
  starpu_task_insert(&potrf_cl, RW, A_handles[k][k], 0);
  for (m = k+1; m < N; m++)
  {
    starpu_task_insert(&trsm_cl,
                       R, A_handles[k][k], RW, A_handles[m][k], 0);
  }
  for (n = k+1; n < N; n++)
  {
    starpu_task_insert(&syrk_cl,
                       R, A_handles[n][k], RW, A_handles[n][n], 0);
    for (m = n+1; m < N; m++)
    {
      starpu_task_insert(&gemm_cl,
                         R, A_handles[m][k], R, A_handles[n][k],
                         RW, A_handles[m][n], 0);
    }
  }
}
starpu_task_wait_for_all();
```

The variables `portrf_cl`, `trsm_cl`, `syrk_cl` and `gemm_cl` are the codelets corresponding to the computation kernels. The data handles used for the tasks are in this example blocks of the matrix A.

Finally, StarPU users and developers can rely on a large set of tools to analyze application performance, which is important to understand the behaviour of the runtime

system, scheduling decisions, *etc.*

1.3.3 Distributed StarPU

General concepts

The StarPU runtime system has been extended to support distributed applications [12]. When distributed applications are executed, each process executes the same program, the same instructions, but parts of the program are conditioned by the rank of the process.

The starting point to write distributed applications with StarPU is the data distribution: how the data manipulated by tasks are spread over the available nodes. The data handles are created by specifying on which node they are actually allocated.

StarPU by default executes tasks where the data is located: the node which owns data to execute a task will execute this task. Since all StarPU processes execute the same program, all processes handle the same instructions to create tasks, and thus unroll the same DAG. By analyzing the data required to execute each task, and their access mode, StarPU knows which node has to execute which task: each node will execute the tasks which write in the buffers it owns. If this node does not own buffers it needs to read to execute this task, the node which owns the data will send it. If a task needs to write several data handles that are not all owned by a common node, the default policy is to select the node which will require the smallest total of transferred amount of data.

With such working, data exchanges between nodes are inferred by the runtime system from the task graph and are not explicitly written by the application developer, which simplifies a lot the development of distributed applications.

The DAG of the distributed version of the Cholesky factorization depicted in Algorithm 1 is the same, but mapped on the set of available nodes, as illustrated by Figure 1.5. Internal data dependencies in a node are represented by gray arrows and required network communications between nodes to exchange data handles are represented by black arrows.

Communication engine

StarPU-MPI, the name of the StarPU extension for distributed executions, provides an API to the user to manage distributed executions. Internally, it uses by default the MPI interface to make network communications, thus it can work on top of any MPI library. A StarPU thread is dedicated to manage the communications: this thread makes communications progress and watches their termination. Indeed, all MPI communications posted by StarPU are *non-blocking* (*i.e.* the function call does not block the thread until the completion of the communication) and the status of these communications has to be watched to know when a communication is over, which can unblock tasks waiting for a data handle from another node or waiting to write in a data handle being sent to another node. Only this thread makes MPI calls, to avoid the need for `MPI_THREAD_MULTIPLE`, required to make MPI calls simultaneously from different threads.

The use of non-blocking functions and a dedicated communication thread allows to

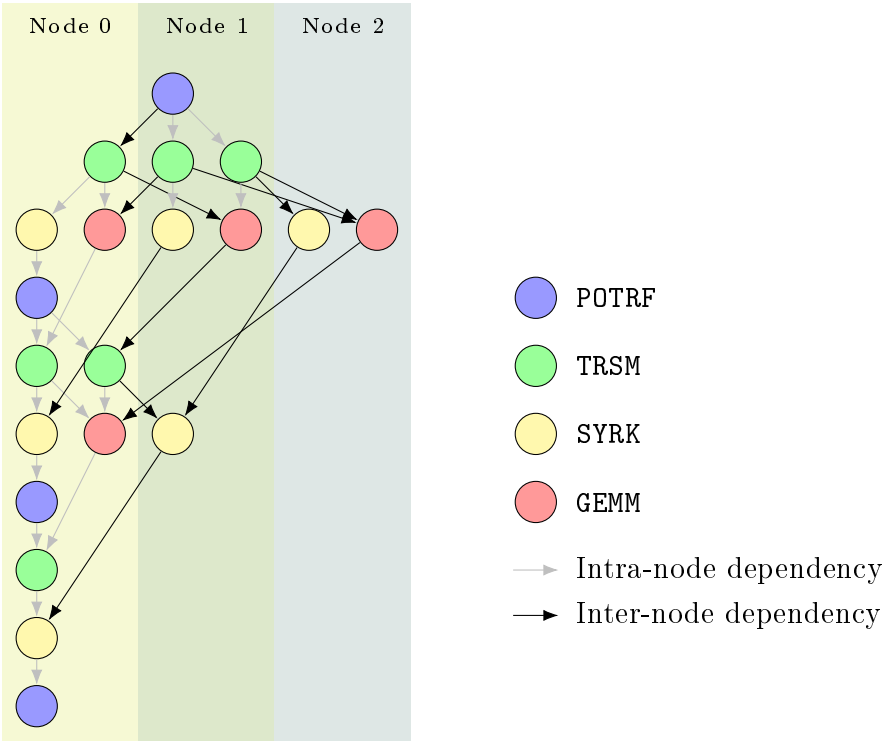


Figure 1.5: Distribution of the Cholesky DAG on 3 nodes. Inter-node dependencies trigger network communications.

overlap communications with computations, *i.e.* executing simultaneously communications by the MPI thread and tasks on other workers. While this can be a tedious work to code such feature manually, StarPU-MPI allows to do it seamlessly, without real additional work from the developer.

Actually, the MPI interface is not well-suited for distributed task-based runtime systems. The MPI API fits with (and was designed for) Bulk Synchronous Parallel (BSP) applications, which alternates phases of computations and communications: all workers execute computations, then only one worker is in charge of communications and then circling back to computations (this is called the *fork-join* model). Task-based applications are more irregular: each task is scheduled and executed independently from each other (except the respect for dependencies), and any type of synchronization is avoided. Thus communications can be triggered as soon as a task ends, and conversely, tasks can be executed as soon as a data is received, without any synchronization at node or application level.

In this context, the main missing feature from MPI for task-based runtime systems is event notifications: being able to register a callback function to execute when an event occurs: for instance, telling StarPU expected data handles have just been received, which fulfils the dependency requirements of a task and allows to schedule this task for execution. As well as communications could be issued directly from the task prolog, executed by the same worker which just executed this task, to avoid the extra cost of passing through the communication thread. This means the multi-threading support of

the communication library has to exist and be efficient.

Moreover, task-based runtime systems can issue bursts of communications requests (at the beginning of an application, when the result of a task is required by many nodes, *etc*), which have to be efficiently handled by the communication library. First of all, the communication library has to scale well regarding the number of issued requests. Then, hints can be given to the communication library about the communication priorities, to help the communication scheduling: the developer of a StarPU application can define priorities to submitted tasks, and these task priorities can be used by StarPU to determine communication priority. Unfortunately, the MPI standard does not support communication priorities.

All these wished features are actually provided by NewMadeleine. This is why StarPU-MPI has been ported to use the native interface of NewMadeleine [21], and takes benefit from the multi-threading support, event notifications, priority support, *etc*. When this back-end is used (enabled at compile time), much part of the communication engine logic is delegated to NewMadeleine: communication priorities, monitoring of running requests, communication progress, *etc*. Send operations are issued directly from workers and a thread managed by NewMadeleine executes callback functions.

Differences between the NewMadeleine and the MPI backends of StarPU are further discussed in Appendix A.

1.4 Goals and contributions of this thesis

Optimizing HPC applications pursues usually one general objective: saving resources. These resources can be temporal: reduce the duration of an application execution; and/or spatial: require less memory capacity or less hardware resources (*e.g.* less nodes) to run the same application.

To achieve this goal, optimizations can be made in several layers of our software stack: improving the application algorithm in the user application (better initial data distribution, communication-avoiding algorithms, *etc*), improving the runtime system behaviour to better exploit the resources (efficient schedulers, correct performance prediction models, low overhead, *etc*), and improving performance of communication libraries (good scalability of the number of request, low matching duration, *etc*).

Even if these software layers are dependent on each other (*i.e.* an application needs a runtime system, but runtime systems are developed for applications) and performance issues are found when using the whole stack, applications, runtime systems and communication libraries can be optimized independently. Indeed, changing the application algorithm does not require interaction with runtime system developers and improving scheduling policies can stay opaque to the end-user of runtime systems, for instance.

The proposal of this thesis is to do the opposite: **optimize task-based runtime systems and communication libraries together, by improving their interactions**. Both the runtime system and the communication library have information about their activities: the runtime system can know future communications to be issued, the estimated end of a task execution, the location of data handles, task priorities, *etc*; while the com-

munication library knows the state of the network, which communications are incoming, performance of the network and its possible optimizations, *etc*. These two software layers should be able to exchange information about their respective state in order to provide more hints for their decision-makings.

Although, in this thesis, the ideas to increase interactions between task-based runtime systems and communication libraries are implemented in StarPU and NewMadeleine, the coupling between the runtime system and the communication library should remain loose. Rather, we propose interfaces to exploit the ideas. These interfaces should be portable enough to be implemented in any task-based runtime system and communication libraries with similar constraints as the ones we are dealing with. Implementations made in StarPU and NewMadeleine can be seen as prototypes: they aim at proving the feasibility of the interface and evaluating the performance gain.

Three kinds of opportunities for better collaboration between task-based runtime systems and communication libraries have been explored during this thesis:

Tracing systems. Being able to precisely and efficiently trace application executions is paramount in order to understand the behaviour and performance of applications. Recording execution events can add an execution overhead, reduce application performance, and more important: change the behaviour of the applications. For distributed applications, the traces gathered on each node have to be synchronized to keep coherency in the timeline of events. However, this requires distributed synchronized clocks, which is not straightforward to provide.

In Chapter 3, we first describe the tracing system used with StarPU. Then, we present few sources of performance overhead when using tracing systems and give hints to reduce them. We evaluate different methods of clock synchronization and evaluate their accuracy for analysis of distributed traces. Finally we discuss requirements all tracing system should fulfill to be competitive enough.

Dynamic broadcasts. Some task-based algorithms need to send the same data to multiple nodes, *i.e.* broadcast patterns appear in their task graph. Plain MPI applications can take advantage of routines dedicated to such situation, which will optimize the broadcast by using the most suitable routing algorithm. However, broadcasting routines provided by the MPI standard does not fit with the constraints of StarPU.

We propose in Chapter 4 a new approach to overcome these constraints, while using optimized broadcasting trees and improve application performance.

Interferences between computations and communications. Runtime systems such as StarPU naturally overlap communications by computations. This means computations and communications are executed simultaneously. Since these two activities share the same resources (processor, memory system, *etc*), interferences between computations and communications can happen, impacting their respective performance.

We present in Chapter 5 the impact of computations and communications on the performance of each other, when they are executed side-by-side. We study especially the effects of processor frequency variations and memory contention.

Chapter 6 focuses on the impact of memory contention between computations and communications. We propose a model to predict memory bandwidth for computations and communications, when they are executed simultaneously, according to the number of computing cores and the data locality. This model helps to better understand how the system deals with memory contention and where are the bottlenecks in the memory system.

Chapter 2

Related Work

This chapter presents the work related to topics covered in this thesis. First sections describe programming models used in HPC applications and present existing task-based runtime systems. Then, we explain how distributed models are integrated in other task-based runtime systems, and how communications can be optimized by the runtime system. The rest of the chapter browses the literature related to the contributions of this thesis.

2.1 Programming models

The general problematic about programming HPC machines is how to express parallelism when writing application code. Several methods exist, providing more or less abstraction of the machine and requiring instructions from the programmer more or less implicit or explicit. Task-based model (presented in section 1.3.1) makes programmer implicitly express the parallelism of its application and provides a high level of abstraction of the machine to ease the writing and portability of applications.

Fork-join model

The most common parallel programming model is the *fork-join* model (or BSP) where the program is a sequence of parallel regions interleaved by sequential sections, as illustrated by Figure 2.1. Many applications use this model to make computations in parallel sections: each thread works on its dedicated set of data; then in the sequential section, only one thread does a specific task, for instance communication instructions.

Fork-join applications can be written in several manners, the most common being with OpenMP. OpenMP [91] is a runtime system, driven by annotations in the application code (called *pragmas*) to explain how the code should be parallelized. An example is given in the following code snippet, which parallelizes the sum cell-by-cell of two arrays in a third array:

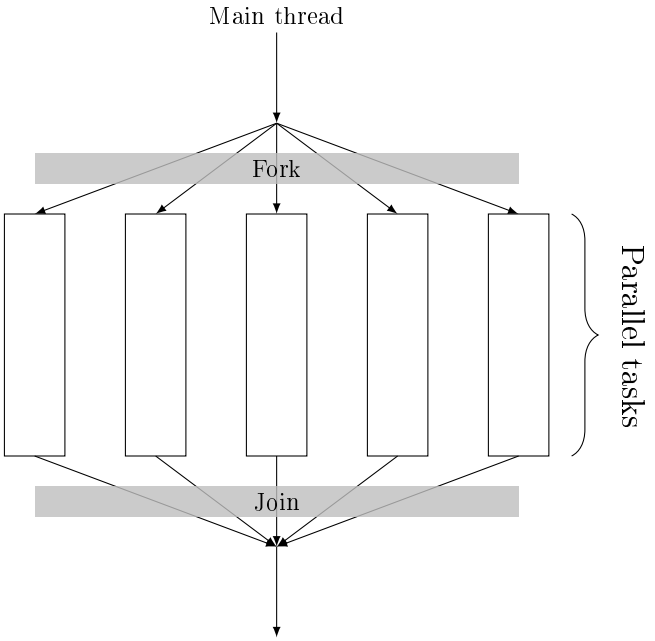


Figure 2.1: The *fork-join* model.

```
int i;

#pragma omp parallel for
for (i = 0; i < n; i++)
{
  c[i] = a[i] + b[i];
}
```

Compared to task-based programming paradigm, this model is simpler, yet more limited: since there is no mechanism to finely manage dependencies in the whole application (at least in the first versions of the OpenMP standard), the parallel sections (*fork*) are almost limited to *embarrassingly parallel* sections of the program. To start the next part of the program, if the previous part has to be finished to keep data coherency (*e.g.* not reading data that is not completely computed (written) yet), a synchronization is required: wait (*join*) for the end of all threads in the previous parallel section. In case of load-imbalance, this can lead to idle time (some threads do nothing, waiting for the other threads), which represents wasted resources. In the same vein, it can be difficult to overlap several steps of the program when data dependencies have to considered or overlap communications with computations [105, 76].

With task-based programs, the knowledge of data dependencies allows an implicit parallelism handled by the runtime system. As long as dependencies are satisfied, tasks can be executed without having to wait for other threads, which are maybe working on independent data (thus waiting for them is useless). Overlapping program phases or communications and computations is implicit and thus straightforward with task-based runtime systems.

Plain MPI

Distributed systems with multicore processors can be programmed with only MPI instructions to parallelize the application. In this case, one MPI process can be launched per processor core. The parallelization is driven by selecting the computations to execute and the data to use according to the rank of the MPI process. Synchronization points consist in exchanging messages between MPI processes: it can be between processes running on the same node (*intra*-node communication) or on different nodes (*inter*-node communication).

While this solution is simple to use, it has several drawbacks. With one MPI process per core, it can be more difficult to take into account the machine topology to optimize applications. The point-to-point communications are more numerous and collective communications involve more MPI processes, whereas several MPI processes in a collective can be on the same node. The memory consumption can be much higher with plain MPI programs: data being replicated on several MPI processes can in fact be replicated in the memory of a single node: if processes knew the topology, maybe one copy of the data on the node would be enough. More communications implies also more memory allocated for reception buffers.

On heterogeneous systems, MPI processes running on a same node have to find a consensus on which process will use the accelerators, and then efficiently balance the workload between processes computing with an accelerator and other processes.

MPI+X

MPI+X is a programming model with MPI associated with another runtime system handling the program parallelization. In such case, usually MPI manages distributed memory and X shared memory. A common mix is MPI and OpenMP: there is one MPI process per node, MPI handles the distributed memory by exchanging messages between nodes; and each MPI process uses OpenMP to express the possible parallelism on each node, using shared memory. This way each MPI process can exploit the topology of nodes, especially regarding the memory organization.

Other than OpenMP, the companion of MPI can be any programming model running on a single node, exploiting shared memory. It can be, for instance, simple threads, or even... MPI [64]!

A trade-off between one MPI process per core and one MPI process per node (with another runtime system for parallelization inside a node) is to use one MPI process per memory group in the memory topology, especially per processor or per NUMA node. This can be a good approach if the runtime system X ignores the memory topology.

StarPU can be used in a MPI+StarPU model: in this case, StarPU would not see the distributed side of the program, would only be used for parallelism inside nodes and would let the application developer manually manage the distributed aspect of their application. Such use-cases can appear, for instance, when progressively transforming an MPI+X application into a StarPU application.

The distributed extension of StarPU can be considered as an implementation encompassing an MPI+threads model.

PGAS

The MPI standard is based on explicitly exchanging messages between MPI processes. Another communication model is the Partitioned Global Address Space (PGAS) model: a global memory address space is split across all the nodes, and each process can access memory regions exposed by other processes, without any specific instruction from the target process for this specific access. In other words, this model allows to access remote memory regions without requiring synchronization between the sender and the receiver process.

Several libraries and runtime systems implement this communication model: UPC [47], UPC++ [127], Chapel [32], *etc.* PGAS is now part of the Fortran 2008 standard with *coarrays*. The MPI standard supports a PGAS model (also called *one-sided* operations) with its set of Remote Memory Access (RMA) routines.

The PGAS programming model does not fit with StarPU's design. One of the principles in the StarPU's working is a process can work by ignoring the current state of other processes. When communications are necessary, the sender sends a message when it is ready to send the data and the receiver explicitly (internally by StarPU) posts a receive request when it is ready to receive the message (*e.g.* it does not use the buffer anymore for other purposes). By requiring explicit actions from nodes involved in a communication, the data coherency is guaranteed. With PGAS systems, only one of the process takes the initiative to access remote memory of another node. However, without any additional synchronisation mechanism, the active node cannot know if the data located in the remote memory of the passive node is already in the state the active node wants to get the data. In the same fashion, it is important for StarPU to know when a data is completely received or sent, since it can unlock tasks to execute, which requires a notification mechanism.

To summarize, the PGAS model would need synchronization and notification features to properly work with StarPU. Implementing them would mimic the working of MPI backend of StarPU...

Conclusion

Presented parallel programming models require to explicitly express parallelism in the application and to use synchronization points to ensure application coherency. Such work can be tedious for the application developer and the resulting program may not be portable on other clusters. Moreover, the developer has to know several runtime systems: usually at least two, for both inter- and intra-node parallelization. The task-based model allows to implicitly express parallelism (at a node or cluster level) of applications and the role of the runtime system ensures a good portability across different machines.

2.2 Task-based runtime systems

StarPU (already presented in section 1.3.2) is not the only distributed task-based runtime system. Others exist, and they all have their own specificities.

PaRSEC

PaRSEC [25], the successor of DAGuE [26], is a runtime system similar to StarPU, except it supports more input formats to describe programs. It is originally focused on the PTG representation, but it can also be used with the STF model.

Parametrized Task Graph (PTG) is a algebraic representation of a task-based program [37]: each task is described by its code (usually calls to C functions) and dependencies between tasks are represented with algebraic conditions on input and output data of each task. The following code snippet is the `TRSM` part of the PTG version of the tiled Cholesky factorization:

```
1  TRSM(k, m)
2
3  // Execution space
4  k = 0    .. NT-1
5  m = k+1  .. NT-1
6
7  // Task Mapping
8  : A[m][k]
9
10 // Flows & their dependencies
11 READ A <- A POTRF(k)
12 RW   C <- (k == 0) ? A[m][k]
13        <- (k != 0) ? C GEMM(k-1, m, k)
14        -> A SYRK(k, m)
15        -> A GEMM(k, m, k+1..m-1)
16        -> B GEMM(k, m+1..NT-1, m)
17        -> A[m][k]
18 BODY
19   trsm(A /* A[k][k] */, C /* A[m][k] */);
20 END
```

The main advantage of such model is the lightweight representation of the DAG for the runtime system: each task composing the whole task graph of an application does not need to be instantiated ahead of time, instead the algebraic representation of the task graph indicates to the runtime system, from the current application state, which kind of task has to be executed after the end of a task. While the memory used for the DAG representation of StarPU applications is linear with the number of tasks; with the PTG representation of PaRSEC applications, it is linear with the number of types of tasks.

Unlike StarPU, each PaRSEC's thread can change its purpose during program execution: during a phase without communications, all threads can execute tasks and no thread will be in charge of communications. When a phase with communication will start later, any thread can take the bullet to deal with communications. The same goes for threads driving GPUs. This mechanism allows to better adapt the runtime system

behaviour to application phases.

By default PaRSEC relies on the MPI interface for network communications. Nevertheless PaRSEC supports other communication systems, such as directly UCX (*Unified Communication X* [106]), a low-level communication library, abstracting the access to different types of network interfaces; or LCI [41], which is a lightweight communication library designed originally for graph analytic, avoiding the pitfalls of MPI mismatching with requirements of graph applications and supports multithreading (in a word: a library with features similar to NewMadeleine).

OmpSs

The OmpSs programming model [46] is similar to OpenMP (annotations, fork-join model, *etc*) and is used as research platform to experiment and implement research ideas, which could end in the OpenMP standard. Its distributed implementations [28, 112] rely on the PGAS communication model and a master/slave design: the master process analyzes the task graph and sends to slave processes tasks to execute.

The master/slave model is known to have scalability issues: when the number of slave processes increases, the bottleneck is the single master process which has to supervise the activity of more slave processes. To mitigate this issue, several levels of hierarchy can be introduced in the master/slave model: the main master process distributes coarse-grain work to submasters and these submasters manage a set of slave processes.

With StarPU, each node unrolls the same task graph of the application and keep only the task it will execute. This method avoids the need for a centralized approach and monitoring the state of each node or exchange status messages between nodes. However, it cannot dynamically handle load unbalance if the original data (or task) distribution is not optimized.

OpenMP

Version 3 of the OpenMP standard introduces the `task` keyword and the version 4 adds the support of dependencies between tasks. The conception of task-based applications with OpenMP is different from just using a fork-join model: the application has to be *taskified* and dependencies have to be expressed. The `pragma` instructions can become quite long and complex...

Regarding distributed applications, OpenMP can be used along with MPI with the fork-join model. Calling MPI routines directly from OpenMP threads allows to avoid the need to wait the end of a parallel section to launch communications, but it adds a complexity level to the program and can be tedious to master. Moreover, it requires a correct support of `MPI_THREAD_MULTIPLE` from the MPI implementation. To ease this usecase, the version 4 of the MPI standard introduced *partitioned communications*: each thread can bring its contribution to a single message, without requiring to initialize the MPI library with the `MPI_THREAD_MULTIPLE` threading support.

Actually StarPU provides an implementation of the OpenMP interface, to be used

as the runtime system executing OpenMP instructions [11].

And many others

Quark (QUeing And Runtime for Kernels) [126] is a task-based runtime system for multi-core systems with shared memory, especially targeting linear algebra applications. It uses the STF model and inspired a lot the design of StarPU, especially the `task_insert` semantic. However, the project does not seem active anymore. YarKhan showed in his PhD thesis [15] some concerns about the distributed extension of Quark, QuarkD: unrolling the whole DAG on all nodes might become a bottleneck when the number of task increases, and can be partially unnecessary for specific applications, for instance, coupling applications described by task graphs with several components connected with only few dependency edges.

Legion [19], introduced in Bauer's PhD thesis [20], is a task-based runtime system focused on data locality.

Other task-based runtime systems exist, among them: oneAPI (now encompassing Intel's Threading Building Block), SuperGlue [115], Charm++ [9], HPX [75], *etc.* More detailed comparisons between task-based runtime systems about their different working, features and performance can be found in the literature [114, 108, 66].

2.3 Communications with task-based runtime systems

As discussed in the previous chapter (in page 28), MPI is not the most well-suited to be integrated with task-based runtime systems. MPI was not designed for irregular applications requiring asynchronous mechanisms and high reactivity.

Integrating communications in task-based runtime systems

Different techniques can be applied to use a communication library in a task-based programming context. Each task-based runtime system proposed its solution to integrate a communication library and support distributed executions.

HCMPI [34] is an extension of the Habanero-C task-based programming model to support distributed executions. One major goal of HCMPI is to avoid the synchronizations imposed by the *fork-join* model, where communications are made by a single worker after waiting for all workers to finish their computations. The design of the distributed extension is positioned between the MPI+OpenMP model and a PGAS model: to handle the asynchronicity of the task-based model and be able to easily overlap communications by computations, communications are non-blocking, can be launched from tasks and are handled by specific workers, different from those executing computation tasks.

In the OmpSs programming model, MPI communications can also be made inside tasks. To prevent blocking communication operations from wasting worker time to wait for a communication inside a task, Sala *et al.* proposed [100] mechanisms to pause tasks

and exclude them from the ones being ready to be executed, when they start a blocking MPI communication, and release them when the MPI communication is finished. This way, the worker which was executing the task can execute other tasks while the blocking MPI operations are not completed. Another thread is dedicated to poll the completion status of the MPI operations.

One-sided communications can also be used in OmpSs [99]. Both the runtime system and the PGAS library needed proper changes, especially to be able to notify receivers when data arrived and to wait for specific communication operations, and not a set of communications.

Some attempts were made to try to extend the MPI interface to include a notification system. For instance, Schuchart *et al.* proposed [102] an extension to register callback functions to be executed when an MPI request is finished. Their solution still requires a progress thread to make MPI check the status of current requests, but their evaluations on several applications (including PaRSEC ones) show performance gains. Similar proposal was made by Protze *et al.* [97].

To the best of author knowledge, a combination of a task-based runtime system with a natively event-driven HPC communication library has not been proposed nor evaluated. This thesis might in a sense fill this lack, by exploring the possibilities of the couple StarPU and NewMadeleine.

Considering communications from a runtime system point of view

After integrating communications into the working of a task-based runtime system, the runtime system can also optimize their use and take them into account to make smart decisions.

Castillo *et al.* expose [30] internal events of MVAPICH and implement mechanisms in OmpSs to listen to them and react accordingly. The knowledge of the communication library internal states now available to the runtime system allows them a better reactivity to network events and especially an optimization of collective operations which collect data from several peers (*e.g.* `MPI_Gather` or `MPI_Alltoall`): when data from a collective operation is only partially received (*i.e.* only a subset of involved nodes already sent their contribution), tasks that require only the received portion of data can immediately be launched, without waiting the end of the whole collective operation. This permits a better overlap of collective communications with computations.

Pereira *et al.* propose [93] to prioritize OpenMP tasks containing MPI send instructions (and their parent tasks), to reduce the idle time waiting to receive data. They study different techniques: manually setting the priority of OpenMP tasks, with a higher value for those containing send instructions; a semi-automatic approach annotating OpenMP tasks which contains send instructions to inform the runtime system and let it manage the priorities; and an automatic approach based on building profiles of tasks which contained MPI operations, to be able to set suitable task priority to future tasks matching the same profile. The main goal in this work was to reduce the time a task blocks a thread because of a blocking MPI operation inside.

This situation cannot appear in StarPU, since communications are not executed inside tasks and are non-blocking; a lengthy MPI communication will only delay the execution of tasks waiting to receive this data; during the time, workers can execute other ready tasks.

Again, presented work is close to the problematic of this thesis (better collaborating between communications and task-based runtime systems), but does not consider a runtime system with an STF model and event-based communication library.

In StarPU

In the second chapter of his PhD thesis [104], Marc Sergent improves the scalability of StarPU-MPI in three ways, detailed in the remaining of this section.

In StarPU-MPI applications, by default all nodes unroll the same task graph. This is required by the STF model to avoid synchronizations and/or control messages between StarPU processes. This means each node analyzes all tasks of the program, and keeps only the tasks it is involved in. Sergent showed the time required to analyze the whole task graph (the *submission* time) can become a bottleneck when the number of tasks and nodes increase. To address this issue, the task graph can be *pruned* from the application level: each node will submit only tasks, which the runtime system needs to be aware of on this node: none of the submitted task will be useless for the local StarPU process. This reduces the number of task each node has to discover and improves scalability.

Another factor limiting the scalability is the memory consumption by both the runtime system itself and the application. A cache mechanism was introduced in the memory management of StarPU to avoid heap fragmentation and memory waste. This is mainly dedicated to memory allocations for MPI receptions.

Unrolling the whole DAG at the beginning of application execution allows for the runtime system to have visibility about all the data dependencies and future tasks, which allows to optimize scheduling decisions. However, it requires to allocate more memory at the beginning of the execution for the internal task structures but also for buffers receiving data from MPI communications. These early allocations can lead to consume too much memory. To tackle this issue, Sergent introduces a mechanism to control the task submission flow by blocking the submission of tasks, based on two possible criteria: the number of submitted tasks not executed yet, or the amount of memory allocated for the submitted tasks.

To prevent duplicated communications, *e.g.* when the same data has to be sent to another node for several different tasks, the data is sent only once, and not independently for each task that requires it [12]. This reduces the number of communications and save memory consumption used by reception buffers.

This work improves performance of the distributed extension of StarPU, but at the StarPU level: the communication library is not aware of the optimizations made in the runtime system.

2.4 Work related to our contributions

This section presents work related more specifically to our different contributions.

2.4.1 Broadcasts in task-based runtime systems

The main question addressed in Chapter 4 is how to use optimized routing algorithms for broadcast communication patterns between nodes, while these broadcasts are not explicit and no StarPU process has a whole vision of the application DAG. The two challenges are the detection of the broadcast and the participation to the broadcast of nodes that do not know ahead of time the data they will receive is actually part of a broadcast.

Efficient broadcasting routing schemes have already been discussed a lot [123, 95, 118, 101]. The presented work relies on these existing algorithms and can use any tree-based broadcasting algorithm.

With the PTG representation used by PaRSEC, all nodes know the full task graph, they can easily know all nodes involved in a broadcast and the entire graph being known at the beginning of the execution, explicit call to broadcast routines can be made. In practice, PaRSEC uses binomial or chained trees, on the top of MPI point-to-point requests. Broadcasts are identified directly from the algebraic representation of the task graph, which the application programmer thus has to provide, while our approach can be introduced in most task-based runtime systems, which use a dynamic task submission API.

With the master/slave model of OmpSs, only the master node knows the whole task graph and distributes tasks to slave nodes. Thus, the master node can easily detect broadcasts and tells slave nodes how to handle them. However, no information is published about the optimization of broadcasts.

Charm++ comes with the TRAM subsystem for collective communications, but it is supposed to be used explicitly by the application, which makes its constraints different from our use case.

HPX executes task on remote nodes *via* active messages. Its API contains routines to explicitly invoke a broadcast involving several nodes.

All in all, other task-based runtime systems either do not optimize broadcasts, or have an API or a DAG representation that allows for explicit use of broadcasts, which are different constraints than dynamic task submission.

2.4.2 Interferences between computations and communications

Chapters 5 and 6 explore the possible negative interferences between computations and communications when they are executed simultaneously, since task-based runtime systems usually provide this feature. We mainly study the impact of processor frequency and the impact of memory contention, of which we propose a memory bandwidth sharing model. Unlike many studies which tend to neglect performance of communications

in favour of computations, our work consider equally performance of computations and communications.

Impact of frequencies

A lot of research is done about the impact of CPU frequency scaling, mostly to save energy. However, most of these works consider communication phases as a good opportunity to reduce CPU frequency, because communications would be less CPU-intensive. In our work, we want to reach maximal performance of both communications and computations.

Liu *et al.* studied [84] power consumption of Remote Direct Memory Access (RDMA) communications. They noticed that RDMA consumes less CPU cycles and memory bandwidth than TCP/IP. Moreover, CPU frequency has almost no effect on RDMA performance, unlike TCP/IP. Their work focuses on power consumption and only on communications.

In order to save energy, Lim *et al.* proposed [82] to decrease CPU frequency in communication phases of executed programs. They observed that reducing frequency does almost not degrade communications. However they only use Ethernet-100, which does not behave the same as high-performance networks.

Sundriyal *et al.* applied [111] Dynamic Voltage and Frequency Scaling (DVFS) and CPU throttling techniques during collective communications to reduce energy consumption. They accepted a communication performance loss of 10 % and only changed the behavior of the communication core, not of the whole machine.

Memory contention between computations and communications

Regarding memory contention, previous works focus mainly on impact of memory contention on computation and tend to neglect performance of communications.

Memory contention caused by communications and computations is observed by Chai *et al.* [31]. They did not evaluate the impact of this contention.

Balaji *et al.* studied [18] CPU load and memory traffic caused by communications with TCP/IP over 10 Gbps Ethernet and with RDMA over 10 Gbps InfiniBand. They did not discuss the interaction with simultaneous memory-bound computations.

NiMC (*Network-induced Memory Contention*) is introduced by Groves *et al.* [58]: they studied the memory contention generated by network communications on a set of applications with and without RDMA. However, they only considered the performance of computation, not the performance of the network communications. The solutions they proposed are already implemented in our software stack (using a dedicated core, offloading RDMA transfers) or penalize communications (reducing network bandwidth to reduce memory bandwidth for communications and save it for computations).

Modeling memory contention

Gropp *et al.* proposed [57] to improve the *postal* model, commonly used to model the performance of ping-pong exchanges, by taking into account the number of MPI processes accessing the Network Interface Card (NIC) at the same time. It is not applicable to our work since in our context only one thread handles all communications done by a host.

A theoretical model of the memory bandwidth sharing between computing and communicating threads was made by Langguth *et al.* [79]. Although they considered communications and computations are executed simultaneously, in their model, when communications end before computation, computation gets again all the available bandwidth and *vice-versa* when computation ends before communications. We rather focus on the steady state when there are always computations and communications in parallel (as in many StarPU applications), by considering bandwidths instead of durations. Moreover our model is more low-level, by considering the data placement on the machine topology and the number of computing cores.

Work presented in the rest of this section did not consider communications, but was helpful to better understand the memory system, and the possibilities to model its behaviour, especially under contention.

Queuing theory is often used [35, 119] to model memory contention. Each queue can represent one contention point, and assembling them can describe the general behaviour of the whole memory system. Model parameters are derived from hardware counters, read while executing applications. This kind of model fits well with homogeneous queue consumers (computing cores, caches, memory controllers), but is more difficult to use in our context, because of the heterogeneity of data streams to consider.

Wang *et al.* presented [122] the possible bottlenecks in the memory system to model them with Integer Programming, to find the optimal number of cores to execute memory-bound applications, especially on NUMA systems.

Majo and Gross studied [86] the behaviour of memory controllers in charge of serving local and remote memory accesses. They distinguished the local memory bandwidth (of the local memory controller) and the remote memory bandwidth (of the QPI bus) and modeled the maximum available bandwidth as a pondered sum of the two bandwidths, by introducing a *sharing-factor*. The evolution of this factor depending on the number of computing cores helps to understand how the memory controller manages its queuing fairness between different types of memory requests.

Goodman *et al.* presented [55] Pandia, a framework to predict performance of other configurations (number of threads and their placement) of parallel applications. From a machine description and 6 well-chosen application runs, they have all required information to make accurate predictions, by knowing the bandwidth capacity of the different memory buses. They take into account parallel fraction, memory accesses, load balancing and computing resource demands of applications, and rely on hardware counters to get these information.

2.4.3 Tracing systems

To better understand interactions between runtime systems and communication libraries, tracing application executions can be very helpful. Tracing applications consists in recording the behaviour of an application execution to analyze it latter in depth and try to understand the performance. Many articles start from observations of trace executions to explain their findings and improvements, for instance [53, 29, 90]. Two main steps form the tracing process: telling the application what to record and then analyze the execution traces.

Tracing solutions

A large set of tools dedicated to the tracing workflow is available. Some tools focus only on a subset of the steps composing the whole tracing process, while others take care of the whole workflow. As examples, FxT [42] and LiTL [70] are libraries handling event probes and their storage; EZTrace [116] is a library to easily wrap function calls: it stores events when the functions are entered and left. More complex tools such as Tau [107] and OpenSpeedShop [103] execute the application to trace, process all the collected data and give information back to the user about the execution.

Trace file formats resulting from a traced execution are usually a raw format, understandable only by the library which generated the file. However, once converted, the files depicting the execution can be in more common file formats, like Pajé [43] or OTF2 [49], to be read by tools to view and analyze the trace, like Vampir [89], Paraver [94], Scalasca [54] or ViTE [38]. Usually, tools focusing on a particular step of the tracing workflow are linked to specific tools focused on other steps. Some collaborations tend to reinforce these affinities, like Score-P [78], a joint performance measurement environment gathering, among others, Tau, Scalasca and Vampir. All these tools can be used to trace any kind of application, even if they tend to focus on parallel applications.

An overview of the characteristics of traces representing executions of task-based applications is given in [62], along with propositions to ease the trace analysis. The beginning of Chapter 3 explains in detail the specificities of task-based applications when it comes to tracing their executions and how tracing process works within StarPU.

Regarding other task-based runtime systems, OmpSs [46] relies on the Extrae [1] library to generate traces, browsable by the Paraver trace explorer, and PaRSEC [25] uses an internal system to record events [29] and provides a set of tools to convert resulting trace files in more convenient file formats, such as Pajé.

Distributed clock synchronization

When benchmarking or tracing distributed applications, getting the current time to timestamp events, and thus be able to locate them in the time, is usually done by relying on the local clock of each node. However, each node can have a different time origin and, even worse, clocks can have different drifts. Therefore, clocks have to be synchronized between nodes to be accurate enough.

The problem of distributed synchronized clocks, applied to tracing systems or other, has already been covered in the literature, to explain the origin of clock differences in distributed systems, to propose algorithmic solutions, and to present solutions used by applications. Among many work, Becker *et al.* explain [24] how non-constant clock drifts, caused for instance by processor frequency variations, can have a severe impact on distributed clock synchronization. They show that *post-mortem* linear interpolation of clock drift based on clock synchronizations before and after application execution is not enough to compensate for these clock variations, especially on long runs (after several minutes). Jones *et al.* statistically evaluate [74] the accuracy of time synchronization on several leadership class supercomputers in 2016, and report their clock synchronization is not as good as expected, for such top-world supercomputers.

The problem of clock synchronization is not only present in the HPC area. For commodity computers, the Network Time Protocol (NTP) [8] is used by operating systems to have a correct clock, but with a coarse-grain precision: only around 200 μs on local networks. In research about distributed systems, for instance, Clément and Dagenais synchronize [36] event timestamps to trace events in OS kernel during distributed executions.

In the HPC area, there are two main features requiring accurate synchronized distributed clocks: correctly timestamping events to trace distributed executions and benchmarking inter-node communications. Both suffer from the same problems and can usually be solved by similar solutions, but there are still some differences.

In tracing systems, one of the most important requirements is to keep sequential consistency in the traces, for instance to avoid communications appearing as received before they are sent (this kind of artifacts are sometimes called *tachyons*). In addition to correctly synchronizing clocks in order to accurately compute the clock offset during post-processing, some tools also rely on logical clocks: they look for timing inconsistencies and try to correct them by changing their timestamp to preserve the correct chronological event order [48]. With Vampir, two barriers are used before and after the application execution, and then the event timestamps are corrected by interpolating the clock offset [77]; Scalasca use additional logical clocks to fix remaining inaccuracies [23, 22].

For communication benchmarks, especially *collective* communications (involving several processes), the accuracy of a synchronized clock is of paramount importance to have precise measurements and to be able to correctly analyze the results. The problem lies more in being able to start an MPI operation at the exact same time on all nodes, rather than measuring the duration of an action taking place over several nodes. If all processes are able to start at the exact same time, we can use local clocks to measure the duration of the local action, and then aggregate the duration of all local events to get an overview of the global duration. Many articles [60, 61, 65, 80, 69, 68] explore what different methods are used in MPI benchmark sets to synchronize clocks. Lots of tools just use MPI barriers in each loop iteration to start the MPI function at the same time on all processes, despite the inaccuracy MPI barriers can suffer from, as pointed out in several papers [69, 65]. A common practice is also to use the same process (and thus the same clock, not requiring distributed synchronization) to collect the start and end time of the routine execution to be benchmarked. SKaMPI was the first MPI benchmark [125] to implement the most efficient technique to start a function on several distributed processes at the exact same

time, with a so-called *window-based synchronization*.

In both cases, tracing or benchmarking, synchronizing clocks requires efficient algorithms, to compute clock offset as fast as possible. The literature contains some work about the communication patterns to use to synchronize clocks [45, 69, 67, 68, 65], implementation techniques [73] or statistical approaches [85, 39].

In the MPI standard, the function `MPI_Wtime` returns the time in seconds since an arbitrary time in the past. The origin of the clock used by `MPI_Wtime` is guaranteed not to change during the life of the process. However, the used clock does not have to be necessarily synchronized with other processes in the MPI job. In other words, having a global synchronized clock is left to the appreciation of MPI library developers, which is currently not the case in OpenMPI, MVAPICH or Intel MPI.

We explain in Chapter 3 how clocks are synchronized to trace distributed executions of StarPU applications, and we empirically evaluate the accuracy of our implementation.

2.5 Conclusion

Task-based paradigm is an emerging programming model for HPC applications, to easily abstract the complexity of supercomputers. Nonetheless, there is a lot of room for improvements regarding the distributed mechanisms of current task-based runtime systems: the MPI standard does not fit well with the desynchronized aspect of task-based runtime systems, which leads to more or less complex solutions, presented above. Moreover, network communications could be more taken into account by runtime systems to make smarter decisions about scheduling, data placement, *etc.*

Two steps are required to achieve correct performance with distributed task-based runtime systems: the use of a communication library in the runtime system to execute distributed applications and then, optimizing communications from the runtime system, by improving interactions between task-based runtime systems and communication libraries. This thesis focuses on the second step, by exploring which information have to be shared between the task-based runtime system and the communication library.

Most of the reviewed related work does not consider the use of an event-based communication library, more suitable to requirements to task-based runtime systems. Thus, we try to improve the possible interactions between them and more suitable communication libraries, like NewMadeleine.

Chapter 3

Tracing Task-based Runtime Systems

Working on the interactions between runtime systems and communication libraries requires to first understand the existing interactions and then observe and analyze the implemented improvements. Tracing systems can be a handy tool here: they are usually used to record details of an application execution, to then be able to precisely analyze and understand the execution. However, these tracing systems have a cost: some of them require code instrumentation (*i.e.* modification in the application code) and they can add an important performance overhead, sometimes changing the application behaviour when tracing is enabled, which can be dramatic if tracing the execution makes the developer actually observes different behaviours from the ones he wanted to understand originally. Challenges for tracing systems are thus to be as light as possible, as well as being able to bring insightful information to the application user or developer.

Since with task-based runtime systems, application performance can depend as well on the runtime system behaviour as the application behaviour itself, it is important to be able to understand how each component of the runtime system (scheduling, memory management, communications, *etc*) works, and thus have a well-integrated tracing system in the chosen runtime system. Some of them use their own tracing systems, while others rely on existing ones.

In this chapter, we present the challenges we faced to efficiently use and improve the tracing layer in the StarPU task-based runtime system. We explain how the tracing system works within StarPU and focus on two aspects: the different sources of performance overhead coming from the tracing system, and the clock synchronization issue for precise traces of distributed executions.

3.1 Background: tracing task-based runtime systems

This section presents tracing systems, how they work, the problems they face, and finally what the special requirements of task-based runtime systems are regarding tracing systems.

3.1.1 Generic tracing systems

Development of runtime systems and applications includes being able to trace their executions, to fix bugs, improve performance, *etc*, by having an overview of what is precisely happening during executions. Here, we focus on *offline* (or *post-mortem*) analysis: execute the application by recording a set of events describing the application behaviour. When the application terminates, files containing the execution *trace* are saved and can be exploited by tools dedicated to trace analysis.

The tracing workflow can be decomposed in several steps, each coming with their set of problematic and solutions:

1. Collecting information from application executions. This can be achieved mainly by manually putting probes into the code of the component to be traced (method called *instrumentation*) or by wrapping function calls to let the trace system catch them;

2. Storing collected information. Trace systems have to timestamp all events and save all collected data in a persistent format to make the trace data available to the user for the *post-mortem* analysis. Data has to be stored in a coherent format (keeping the chronological order of events, storing all possible kinds of additional data for each event, *etc*), preferably minimizing the size of the trace files;

3. Converting raw trace files to more practical file formats. Because of the constraints on the previous step, the raw trace files are usually not directly exploitable, and need some processing to be read by other tools;

4. Analyzing the trace files. The converted files during the previous step can be read by tools to visualize the execution timeline and to highlight the performance bottlenecks and hotspots, for instance.

3.1.2 Tracing distributed applications: synchronizing clocks

With distributed executions, usually each process is traced locally, generating one trace file per process. A subsequent conversion step is then in charge of merging the trace files to generate a single exploitable trace file describing the behaviour of the whole application execution.

The main concern with distributed traces is the clock synchronization between nodes: each node usually has a different clock origin. Since each process uses its local clock to timestamp events stored in the trace file, clocks have to be synchronized between the different nodes. Even worse, clocks of different nodes can have different drifts, causing a single clock synchronization not to be accurate enough after some elapsed time.

In practice, badly synchronized clocks can break temporal order of events (the best example is a communication between two nodes appearing in the trace as being received before it was sent) and/or distort the durations of actions involving several nodes (a communication can for instance look faster than in the reality).

There is no straightforward solution to synchronize distributed clocks, which satisfies all requirements: accurate, fast to initialize and to access, scalable with the number of processes, precise enough for a defined amount of time and without overhead for the application using this kind of clock. Moreover, many factors may influence the accuracy of synchronized clocks, from hardware characteristics (processor frequencies, network performance, computing load, *etc*) to software features (algorithmic complexity, for instance).

The distributed clock synchronization problem is discussed in detail in section 3.4.

3.1.3 Tracing systems and task-based runtime systems

Even if task-based runtime systems can be traced and analyzed with existing tools, these tools are widely used for more classic applications, which are usually more regular (or even based on the BSP model), and can miss some important information related to the working of task-based runtime systems. For instance, one of the key components of such runtime systems is the scheduler, orchestrating the DAG execution onto the computing units: tracing its behaviour to control and understand its decisions requires to collect and visualize particular information, such as the number and types of tasks ready to be executed, which memory node the data buffers these tasks will use are on, what the current status of each computing unit is, *etc*. Moreover, since all the program execution relies on a DAG, saving this graph, *e.g.* all information about the tasks and the dependencies between them, is also important to understand the application structure and how the runtime system deals with it.

3.1.4 Contributions

Within the context of StarPU, this chapter presents some challenges and solutions while integrating and using a tracing systems. It makes the following contributions:

1. A presentation of three sources of performance overhead caused by tracing systems. For each source of overhead, we measure the performance penalty and propose solutions to reduce it;
2. An empirical evaluation of different distributed clock synchronization methods, along with implementation details to compute clock offsets between nodes;
3. A discussion from the different elements learned in the following sections about the method to efficiently trace applications and which requirements has to fit a generic competitive tracing system.

3.2 Tracing StarPU's behaviour

The large number of concepts specific to task-based runtime systems (tasks, dependencies, memory transfer, scheduling, *etc*) shows how complex the work of such runtime systems

can be. Thus, it is important to be able to precisely analyze the runtime system behaviour, to check if it works as expected, to detect and investigate performance issues, *etc*. Each previously enumerated concept can give valuable and ample information about application execution. A method to retrieve all this information is to record them during the application execution and exploit them later, in a *post-mortem* analysis.

To better understand following sections, this one explains more in depth how the trace gathering works within StarPU and how the collected data can then be exploited.

3.2.1 Trace gathering

The big picture to explain StarPU's tracing mechanism is that the internal code of StarPU is riddled with probes to describe what is happening. These probes are instructions to save an event with a timestamp and additional provided information. These events are then stored in a file, to be analyzed later.

Let us consider the example of pushing a task to workers: all data dependencies of this task have been fulfilled, the task is ready to be executed by a worker. The function in charge of this action begins as follows:

```
1 int _starpu_push_task_to_workers(struct starpu_task *task)
2 {
3     _STARPU_TRACE_JOB_PUSH(task, task->priority);
4
5     // ... actually push the task to computing units
```

`_STARPU_TRACE_JOB_PUSH` will generate an event representing the push of the task given as a parameter, with the given priority. In fact, it is a preprocessor macro that checks whether tracing is enabled and then calls the tracing library to store the event.

StarPU relies on a third-party library, FxT [42], to record and store events. FxT is in charge of collecting events, possibly filtering them, timestamping them and saving them in a raw file. Then, FxT is also used to read the trace file and get all event information: timestamp, event type, thread ID and additional given information (in the previous example: the task and its priority).

Internally, FxT allocates a buffer to temporarily store events, before flushing this buffer to a disk, in the trace file. The buffer is flushed when it is full or when the application terminates. During a flush, other threads can continue to record events, thanks to a double-buffering system.

For distributed executions, a raw trace file per StarPU process is created.

In StarPU, all possible events belong to a category, for instance:

- `TASK`: task information: name, color, submission time, dependencies, number, throttling, *etc*;
- `WORKER`: computing unit activity: start and end of task execution, sleep, memory transfer to execute tasks, *etc*;

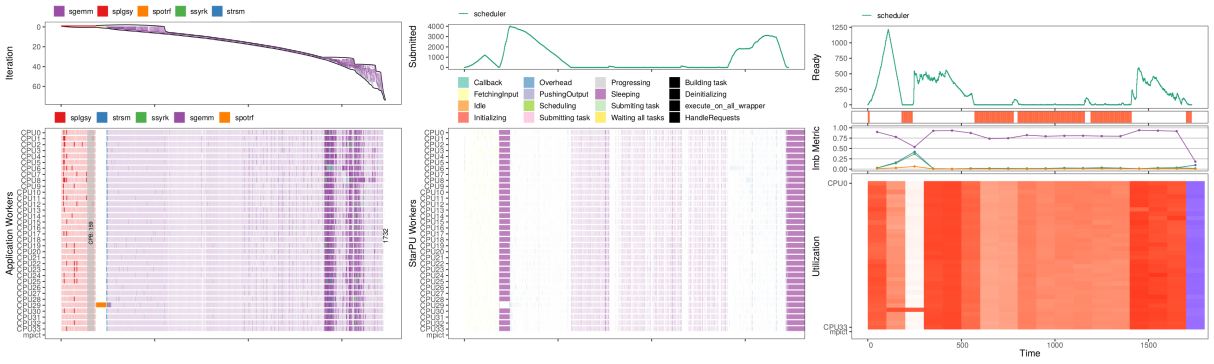


Figure 3.1: Example of visualization of an execution of the Cholesky algorithm with the StarVZ framework.

- `DSM`: all memory management made by StarPU: allocation, release, transfers between memory node, *etc*;
- `SCHED`: scheduler activity: new task to schedule, scheduled task, work stealing, *etc*;

At execution time, users can select which event categories they want to be recorded.

3.2.2 Trace exploitation

Once the application execution has been traced, a raw trace file per StarPU process is left to the user for *post-mortem* analysis. Since these files are understandable only for FxT, StarPU provides the tool `starpu_fxt_tool` which reads the trace files, and transforms them into files with a more convenient format, for instance:

- `paje.trace`: the Pajé format stores timestamped events to describe application behaviour;
- Several `rec` files (a format similar to *Comma-Separated Values*) listing all communications, tasks, data buffers, *etc* and their characteristics,
- `dag.dot`: a `DOT` file representing the task graph of the application, executed by StarPU.

These different processed files can be exploited in different ways:

- The ViTE [38] software can be used to display the Gantt diagram described in the `paje.trace` file: it will statically represent along a timeline the activity driven by the runtime system: task executions by workers, memory and network data transfers, *etc*. An example of the representation of a StarPU application by ViTE is given by Figure 3.3 (page 55).
- StarVZ [53] is an R framework, useful to manipulate data from the trace and to easily make all sorts of plots about information stored in the trace file. Figure 3.1

is an example of basic visualization rendered with StarVZ, where the different plots represent: the parallelization of Cholesky iterations, the task executions by workers, the number of submitted tasks, the worker status, the number of ready tasks, a metric representing work imbalance and worker utilization.

- Users can manually parse files generated by `starpu_fxt_tool` to produce their own analysis and plots to represent metrics they are interested in.

3.3 Reducing impact on performance

Tracing applications implies executing instructions for the original application processing, but also additional instructions to record the events. These additional instructions can add a performance overhead and thus reduce the application performance or, even worse, change the application behaviour. This section presents three sources of overhead caused by the trace recording and proposes solutions to reduce these overheads.

3.3.1 Avoid writing traces on the disk during execution

As mentioned earlier, FxT flushes its event buffer on the disk when it is full. FxT will notice the buffer is full when it will try to record a new event: if the buffer is full, writing the buffer in the file will increase the duration of the probe routine, as much as the necessary time to flush the buffer. This can affect application performance, if it happens during application execution, on a critical path.

We made an experiment to evaluate this possible source of tracing overhead on performance of applications. To generate a lot of events and observe how trace buffer flushes reflect in application performance, we execute several times the same Cholesky decomposition of a matrix of size $24\,000 \times 24\,000$ and plot the performance of each run. At the end, the trace file size is 7.1 GB while the trace buffer size is 1024 MB (*i.e.* flushes occurred during application execution). The trace file was recorded on a BeeGFS parallel filesystem. Results of this experiment are depicted on Figure 3.2: blue dots represent application performance in Gflops and runs during which a flush of the trace buffer occurred are highlighted with a vertical red line. When runs are not disturbed by a flush, application performance is around 3 Tflops (small variations may be caused by processor frequency variations, to avoid overheating). When a flush occurs during application execution, performance can be severely reduced (1.1 Tflops for runs 6 and 12, 1.8 Tflops for runs 18, 30 and 36) or not (3 Tflops for run 25).

Indeed, the impact of a trace buffer flush on the disk depends on when (and where in the StarPU's code) it happens. Figure 3.3 represents the Gantt chart of several executions of a Cholesky decomposition (here the size of the trace buffer was 512 MB and the resulting trace file weights 1.7 GB). The different executions are separated by the vertical white dashed lines and red areas represent idle computing units. Trace buffer flushes occurring at different times lead to different situations, highlighted in Figure 3.3:

[1] Machines used for experiments are described in Appendix C.

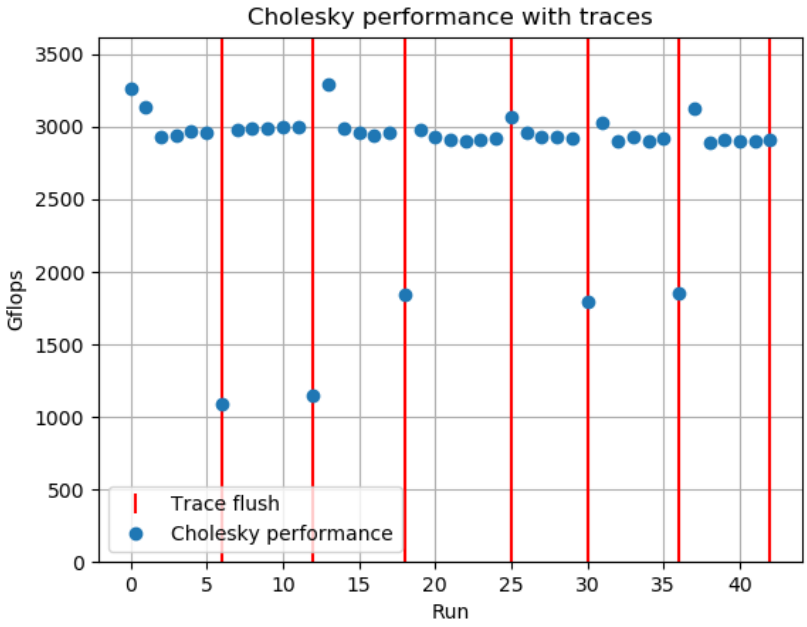


Figure 3.2: Impact of buffer flush on application performance, on a `bora` node[1].

Figure 3.3: Trace of several runs with highlighted impacts of event buffer flushes.

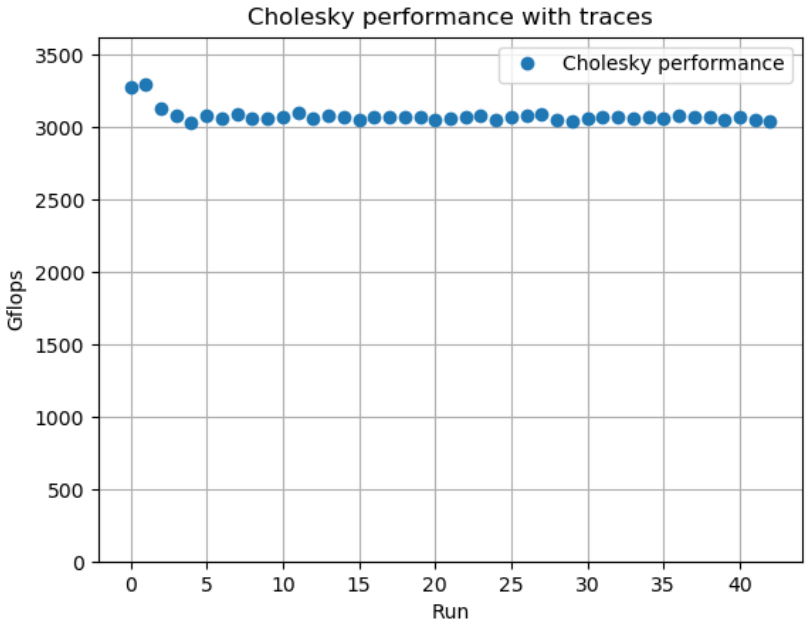


Figure 3.4: Performance of several runs without interrupting buffer flushes, on a `bora` node.

- **A**: flush occurred during *overhead* (somewhere in StarPU's code, but not in a specific section): it did not disturb the application too much, because other workers were able to execute tasks and make the application progress;
- **B**: flush occurred during *progressing* (a memory transfer): in this situation, no computing unit was able to execute other tasks, because a lock was taken, preventing StarPU from launching tasks on other workers;
- **C**: the flush occurred during a task, other workers were able to work as long as the result of the blocked task was not necessary to process remaining tasks.

One way to avoid troubles caused by trace buffer flushes during critical moments is to be able to set the size of the trace buffer. When users execute the application a first time, they are warned for each flush occurring during the execution; at the end of execution, the user can look at the size of the trace file to have a rough idea of the required size of the trace buffer to avoid flushes during execution. Then, users execute the application again, but with specifying the size of the trace buffer. Figure 3.4 presents performance with a trace buffer of 8192 MB (as said previously, the trace file for this experiment has a size of 7.1 GB). There is no outliers and the remaining small variations are probably caused by processor frequency variations.

Another (not implemented) idea to avoid disturbing buffer flushes is to dedicate a non-bound thread to flush the buffer. Since FxT has a double-buffering system, enabling to record events in a second buffer while the first one is being flushed, the thread could write the buffer on the disk without disturbing other important threads. Moreover, this

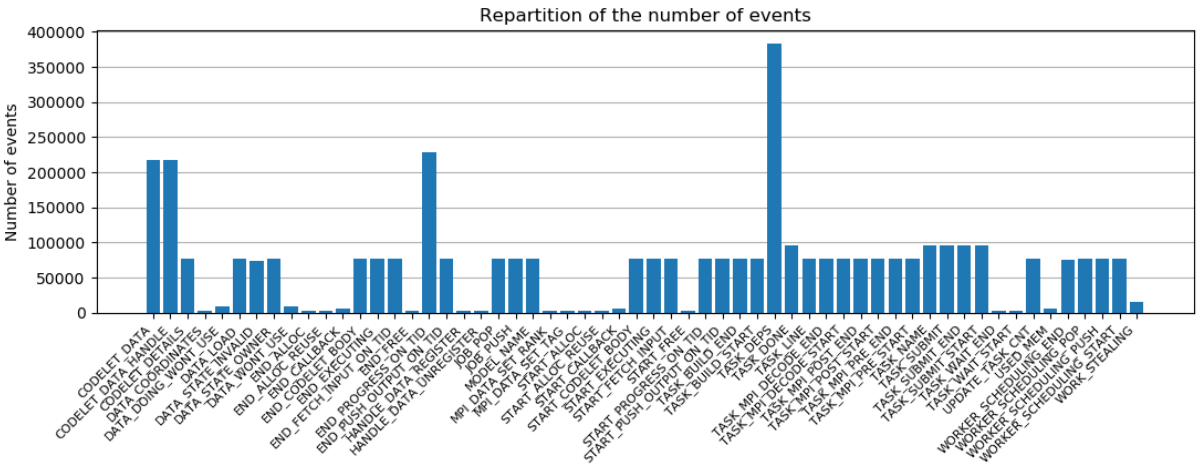


Figure 3.5: Number of events according to their type.

thread would be performing only I/O activities, requiring few CPU resources. This could avoid having to manually specify a buffer size to avoid the problem.

3.3.2 Number of recorded events

The more events are recorded, the more time is spent in the tracing library and we can presume the overhead will be more important. By default, all available event types in StarPU are recorded. The resulting number of recorded events in the trace file can be considerable.

Figure 3.5 depicts the number of events according to their type, for one run of the Cholesky decomposition of a matrix of size of $24\,000 \times 24\,000$. The trace file weights 170 MB and contains 3 887 676 events. On the histogram, only events with more than 2000 occurrences are considered. We can notice some event types are more represented than others: for instance, `TASK_DEPS` (records dependencies between tasks), `END_PROGRESS_ON_TID` (records end of memory transfers), `CODELET_DATA` and `CODELET_DATA_HANDLE` (both record information about the data buffers used by tasks) make the majority.

Recorded events also depict the potential different phases of the analyzed application. Thus, the number and type of recorded events can change during the application execution. Figure 3.6 represents the number of events generated during the application execution. Even without knowing in detail which events are recorded, we can notice four phases: (A) data and problem initialization, (B) task executions, (C) task graph submission, and (D) data release. There are more events during the phase C, because the task graph submission is overlapped by the task executions, which are two different StarPU's activities, each generating their own events. If we look at the same plot, but with details about which types of events are recorded (Figure 3.7), our hypothesis is confirmed: events corresponding to task graph submission occur only in phase C, while events about task execution occur during the whole phase B.

From figures 3.6 and 3.7, we can determine which event types are dominant in the

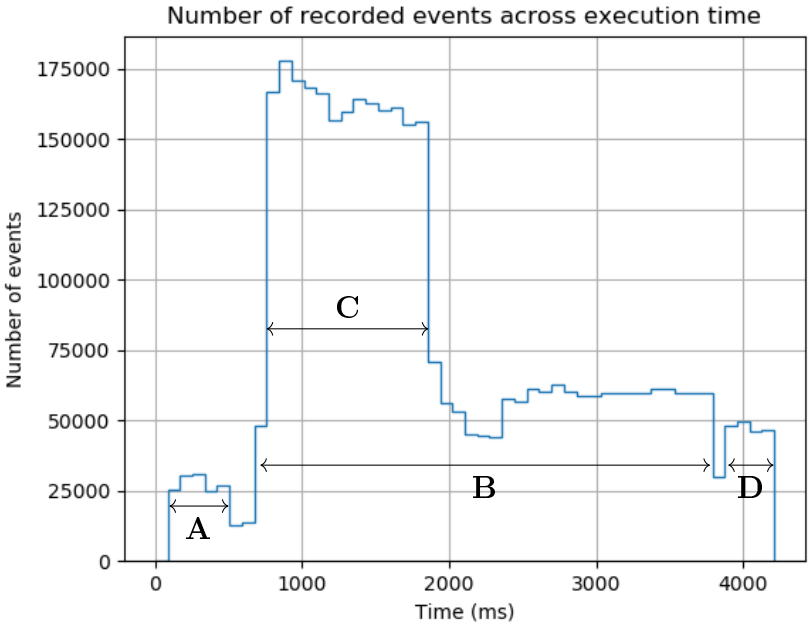


Figure 3.6: Number of events across time (see Figure 3.7 for details about the number of events).

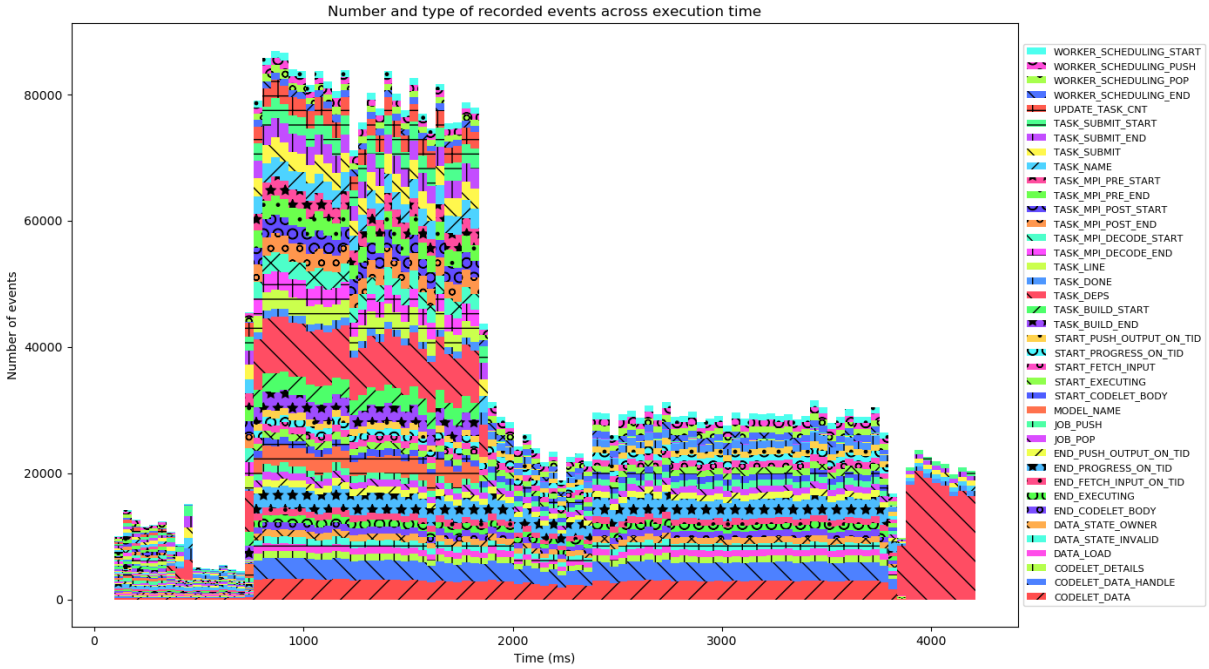


Figure 3.7: Number of events across time, for each event type (detailed version of Figure 3.6).

trace file and thus which ones have to be filtered out in priority, if we want to lighten the tracing activity. The difficulty in selecting which event types to drop during trace recording is finding the good trade-off between acceptable trace overhead (caused by an important number of events to collect) and enough events in the trace file to be able to do insightful *post-mortem* analysis.

If the user knows on which events to focus to analyze the trace of an execution, the set of recorded events can be reduced to keep only the interesting events, and thus reduce the tracing overhead. There are several possible approaches:

- Using the environment variable `STARPU_FXT_EVENTS` to specify which event categories have to be recorded:

```
export STARPU_FXT_EVENTS="TASK|DATA|WORKER"
```

- Manually changing in the source code of StarPU which events will be recorded (by removing some tracing probes, for instance). This can be much more complicated than the previous solution, but it allows a more fine-grain selection of events than just filtering out whole categories.

Figure 3.8 depicts the tracing overhead according to which events are recorded, which changes the number of recorded events (as reported by the red dots to be read on the right Y-axis). One can notice that building StarPU with the trace support, without enabling trace recording at the runtime, does not add an overhead. Then, as expected, the more there are recorded events, the more the impact on application performance is important. It should be noted that this seems to be relative to the runtime system behaviour: in this case, the task graph submission is longer than task execution, thus workers were actively waiting for new tasks to execute. In an execution with another configuration, where the task graph submission is shorter than task execution, the overhead of traces is almost negligible (see Figure 3.9). We can conclude that the trace overhead is mainly caused by events to record on the runtime system critical path, especially when this critical path is under pressure.

3.3.3 Scalability of the number of recording cores

The number of workers (threads bound on CPU cores, for instance) used by StarPU to execute tasks can be set by the user at execution time. The default configuration is to put one thread per processor core. All these threads produce events to be recorded.

By observing the performance of the strong scaling of the Cholesky decomposition, we can notice that the more there are threads recording events, the higher the impact seems to be on performance: Figure 3.10 shows the results on `peabody`, an Intel machine.

On AMD `zonda` nodes, when the MKL library (providing routines called by tasks to actually make the linear algebra computations) is used with its default settings, the maximal reached performance is 1 Tflops, and there is no impact on performance when traces are enabled, regardless of the number of computing cores (see Figure 3.11). However, when the MKL is correctly set up to use all features of the AMD processor (Figure 3.12),

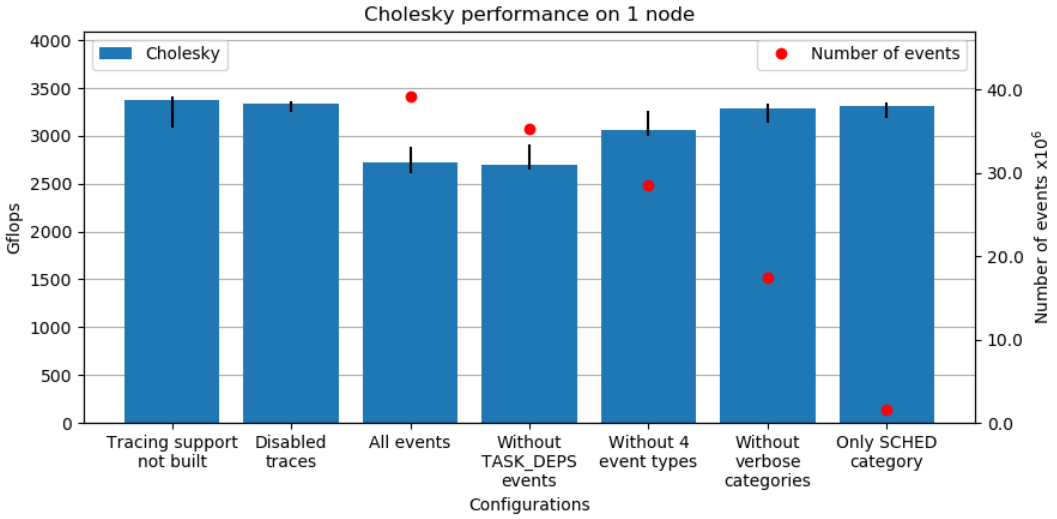


Figure 3.8: Impact of the number of recorded events on the trace overhead. The *4 event types* are the ones previously mentioned being the most numerous in the trace: `TASK_DEPS`, `CODELET_DATA`, `CODELET_DATA_HANDLE` and `END_PROGRESS_ON_TID`.

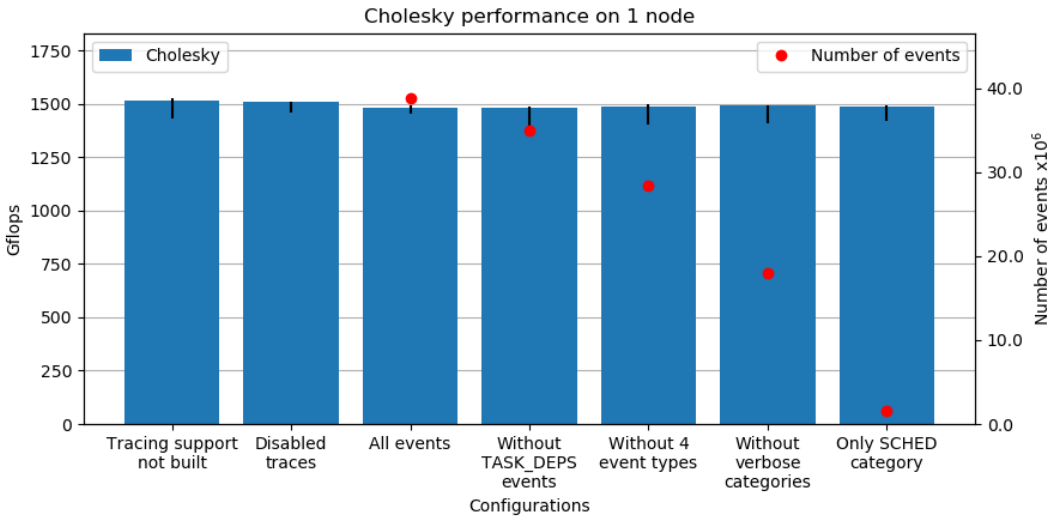


Figure 3.9: Similar to Figure 3.8, but in this case the tasks took more time to complete, reducing the pressure on the runtime system, thus tracing had less impact on performance.

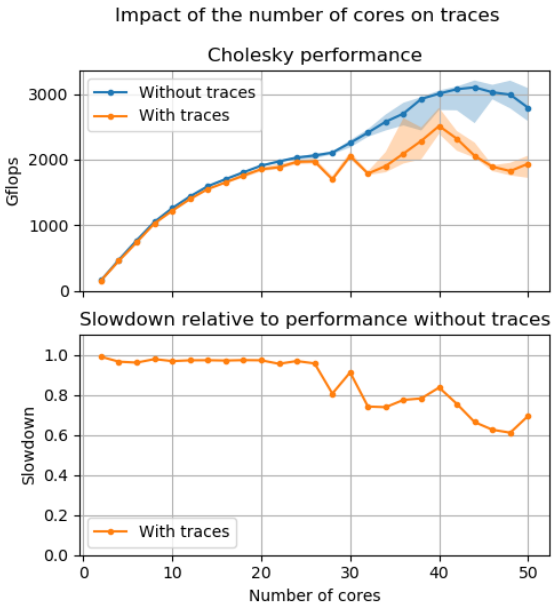


Figure 3.10: Impact of the number of cores on performance with traces on peabody, with Intel processor.

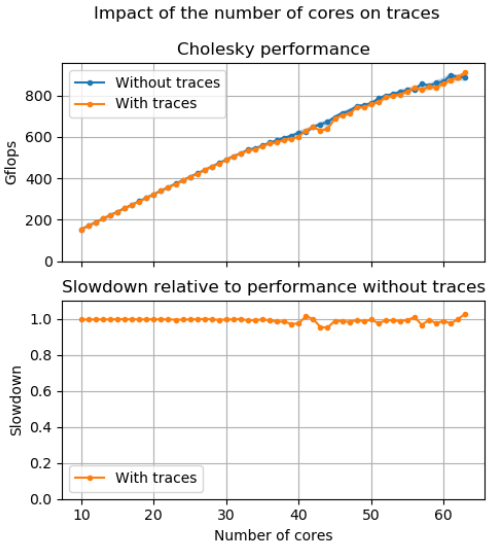


Figure 3.11: Impact of the number of cores on performance with traces on zonda, with AMD processor and badly configured MKL library.

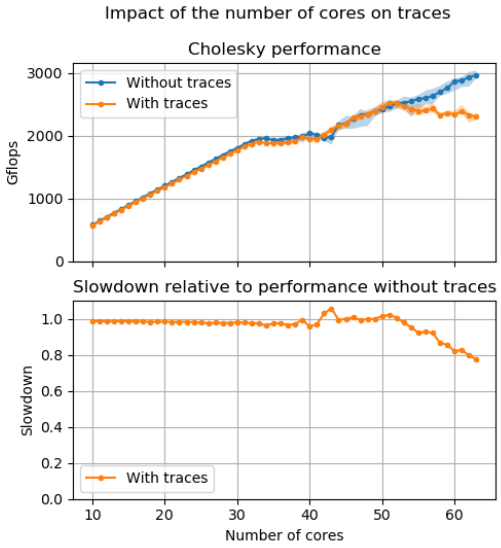


Figure 3.12: Impact of the number of cores on performance with traces on zonda, with AMD processor and correctly configured MKL library.

the maximal reached performance is 3 Tflops, and there is an impact on performance with enabled traces, starting from 53 computing cores (out of 64). This makes sense since faster execution of tasks means a higher throughput of events to record. We can notice here the tracing overhead also depends on performance of the analyzed application, and not only from the implementation of the tracing library or the runtime system.

The observed phenomenon comes from a lock that protects the single list of recorded events in the FxT library. This single list allows to easily keep the temporal order of recorded events in the trace file. With a list of events per core, there would be no lock (and thus no contention on waiting for this lock), but the events would then need to be correctly reordered before any possible exploitation: during the writing of the trace file on the disk or during the conversion of the trace file with `starpu_fxt_tool`. The reordering could be based on the timestamps of the events, which need proper synchronization even when running on a single node, as detailed in section 3.4.2.

3.3.4 Summary about the tracing impact on performance

The various experiments and results presented in this section explored three sources of disturbances caused by the tracing system, impacting application performance and behaviour. Writing on the disk the buffer containing all recorded events can have a severe impact on the application performance, depending on where the flush occurs. Recording many events means an important intrusion in the runtime system behaviour, and an increased activity of the tracing library, which can increase the performance overhead caused by the tracing system. Similarly, many cores recording events can generate contention on getting access to the tracing library.

All these experiments were made on a single node. In distributed executions, the distributed aspect does not bring additional specific overhead coming from the tracing system. However, it has also its set of difficulties, as explained in the next section.

3.4 Precise distributed traces

When distributed applications are traced, there is a need for precise synchronized clocks between nodes to keep a temporal coherency between events. If clocks are not synchronized, the event order can be wrong and, for instance, network data transfers can appear as being received before they are sent! A software adjustment is necessary to avoid these artifacts.

This section presents the problems requiring synchronized clocks, our implementation of precise distributed clocks to trace StarPU applications and its empirical evaluation.

3.4.1 Motivation for synchronized clocks

Usually, when tracing distributed applications, each process uses the local clock to timestamp events recorded in the trace file. To keep a correct temporal coherency between

events (an event on a node occurring after another event on another node is presented as such in the post-processed trace files), clocks of all nodes have to be synchronized: all clocks need to have the same origin and the same speed.

Unfortunately, such perfectly synchronized clocks are usually not present on computing clusters. The local clocks have as origin the start of the node, which is hardly the same on all nodes (nodes are sometimes rebooted independently from each other, for maintenance tasks, for instance). They can also have different speeds, depending on differences in crystal manufacturing, temperatures, and voltage variations.

Synchronization methods exist (such as NTP) to have the current time available on all nodes, but they are too much coarse-grain for the tracing requirements. If we need duration of communications reported in traces, the allowed error on the clock synchronization has to be lower than the minimum network latency (approximately 1 μs on InfiniBand networks).

All in all, if we want precise and coherent distributed traces, the problem of distributed clocks to be precisely synchronized has to be considered in tracing systems. The rest of this section presents how we addressed this issue in StarPU.

3.4.2 Synchronized clocks in StarPU

This section presents in detail how we implemented state-of-the-art techniques to have distributed traces with events precisely timestamped within StarPU. We also report how it improved the accuracy of event timestamps, by comparing the different solutions.

To evaluate the accuracy of several synchronized methods, we will execute several times a ring communication pattern between all nodes, with 4-byte-long messages.

Implementation overview

The general idea is to record events timestamped with the local clock; record an event at the exact same time on all nodes; and during trace file post-processing, use this event to compute clock offsets and adjust the timestamps of all events.

A naive approach to execute an event at the exact same time on all nodes is to perform an MPI barrier, and record the event just after the exit of the barrier. An MPI barrier is a function, provided by the MPI standard, that blocks as long as not all processes called the function. However, an MPI barrier is not precise enough to synchronize clocks, because all nodes do not leave the barrier at the same time, as illustrated by Figure 3.13: the event recorded just after processes left the barrier will not happen at the exact same time ($t_0 \neq t_1 \neq t_2 \neq t_3$), while it will be considered as such ($t_0 = t_1 = t_2 = t_3$) to compute clock offsets. In practice, this can actually make communications being received after they are sent, as shown by Figure 3.14: `starpu_fxt_tool` considered that all MPI processes have left the MPI barrier at the same time (as indicated by the four vertically aligned white circles). However, this was not true: the MPI process 2 left the barrier a moment after the other processes.

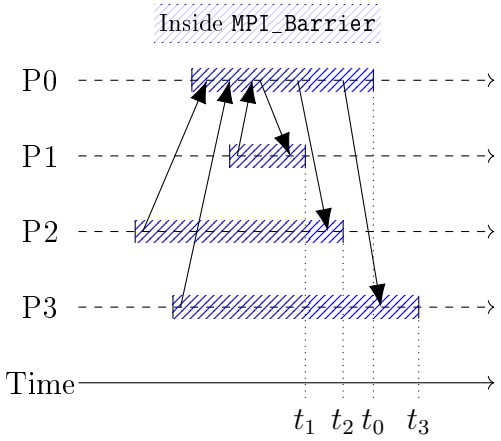


Figure 3.13: `MPI_Barrier`: Not all processes leaves the barrier at the same time.

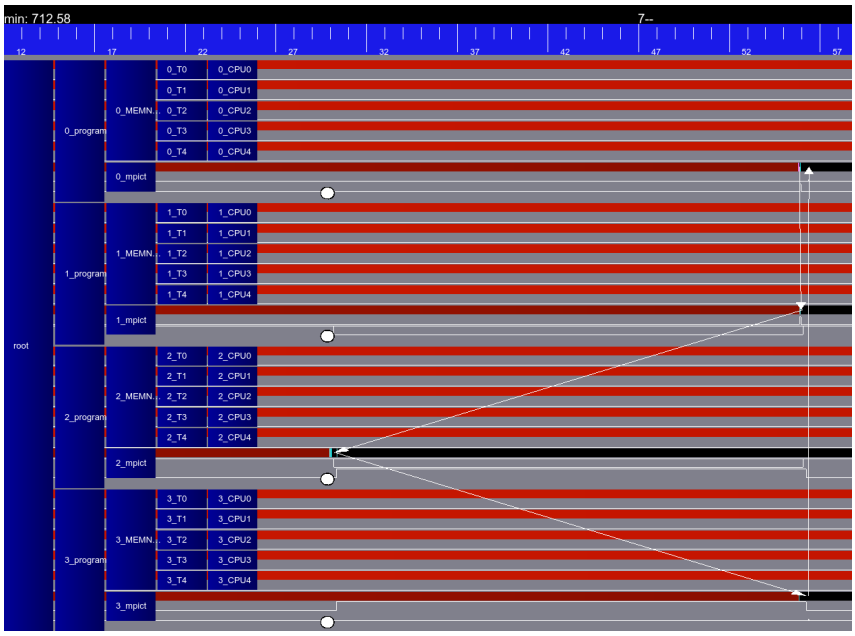


Figure 3.14: The communication from node 1 to node 2 is received before it is sent!

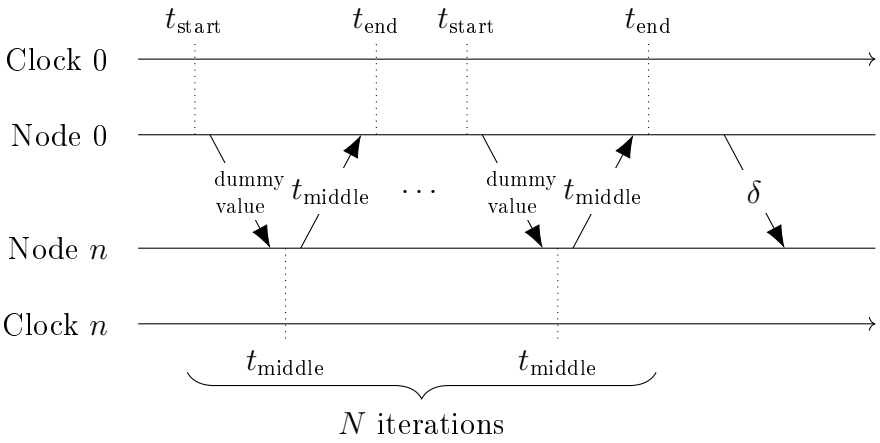


Figure 3.15: How clock offset δ is computed between nodes 0 and n.

Using a precise barrier To make all nodes leaving the barrier at the same time, we use a *synchronized barrier*, based on a window-based synchronization [125]. During application execution, after computing clock offsets (see below), the node 0 (arbitrary choice) decides at which time all nodes will leave the barrier. This time is broadcasted to all nodes, each node applies the previously computed clock offset to get the decided time in its local clock and then waits in the barrier until the deadline. Thus, all nodes leave the barrier at the exact same time. The next instruction after the barrier is to record the event which will be used to compute clock offsets during the trace post-processing.

Computing clock offset To compute the clock offset between nodes, for each node to convert the time received from the node 0 to its local clock, clocks of the two nodes are compared, with the following protocol, illustrated by Figure 3.15:

1. the node 0 saves the current time t_{start};
2. the node 0 sends a message to the node n;
3. just after the node n receives the message from the node 0, it saves the current time t_{middle};
4. the node n sends the time t_{middle} to the node 0;
5. just after the node 0 receives the message from the node n, it saves the current time t_{end};
6. the node 0 can now compute the clock offset δ between the nodes 0 and n: $\delta = \frac{t_{\text{start}}+t_{\text{end}}}{2} - t_{\text{middle}}$;
7. the previous steps are repeated N times and we select the clock offset obtained with the minimal difference $t_{\text{end}} - t_{\text{start}}$;
8. the node 0 sends to the node n the selected offset δ.

Let's explain some details of the protocol. To compute the clock offset between the nodes 0 and n, we need to send the time given by the clock n to the node 0, so it can compare with its own local clock (step 4). However, the time required to send t_{middle} to the node 0 has to be taken in account when comparing the two clocks. To do so, we need to know the duration of the communication transferring t_{middle}. Since the only accurate way to achieve this is to use the same clock to read off the times before and after the communication, we measure on node 0 the necessary duration to send a dummy value with the same size of the transferred message from node 0 to n (step 2) and receive back t_{middle}. Then, the average of t_{start} and t_{end} should be aligned with t_{middle}, and the difference between this average and t_{middle} gives the clock offset between the nodes 0 and n. This is based on the assumption that the communications from the node 0 to the node n and inversely are exactly symmetric: this is why we send a dummy message from the node 0 to the node n (instead of an empty message), and we take the clock offset obtained with the minimum duration of the exchanges between the nodes 0 and n (the smallest duration is when the fluctuation of network latency is minimal, thus a better symmetry of the two communications).

Taking into account clock drift With this synchronized barrier, we can have one reference point in the trace files to compute clock offsets between nodes. However, due to clock drift being different on each node, the computed clock offset is valid only to adjust timestamps of events recorded a short time after the synchronized barrier. Then, clocks follow different drifts and the synchronization is not valid anymore.

A solution to take into account clock drifts is to perform two synchronized barriers, delimiting the period requiring precise well-synchronized timestamps and then, in the trace file conversion step, interpolate the clock offset to apply to each timestamp, with the events recorded after the two synchronized barriers as reference points.

Figure 3.16 shows the difference between using one (at the beginning of application execution) or two (at the beginning and at the end) synchronized barriers. This figure plots the duration of communications from the rings mentioned earlier: the send and receive times are taken from two different nodes (the sender and the receiver), thus having a precise synchronized clock is crucial here for a good estimation of communications duration. Since each communication has the same message size, communications duration should be constant. With only one synchronized barrier at the beginning of the execution (blue dots), measured durations are constant, but for less than one second. Then, the durations follow lines with non-zero slopes, coming from clock drifts. The different slopes come from the clock drift differences between pairs of nodes (clock drift difference between nodes 0 and 1 is different from the clock drift difference between nodes 1 and 2). With two synchronized barriers (before and after the region of interest, orange dots), measured durations are, as in the reality, constant. Durations measured with a simple MPI barrier are not represented, because they are much higher, and the chart scale prevents from seeing the interesting differences between one or two synchronized barriers.

However, linearly interpolating clock drift assumes the clock drift is linear, which in practice is not the case after several minutes. Thus, this method can be used to trace only short executions.

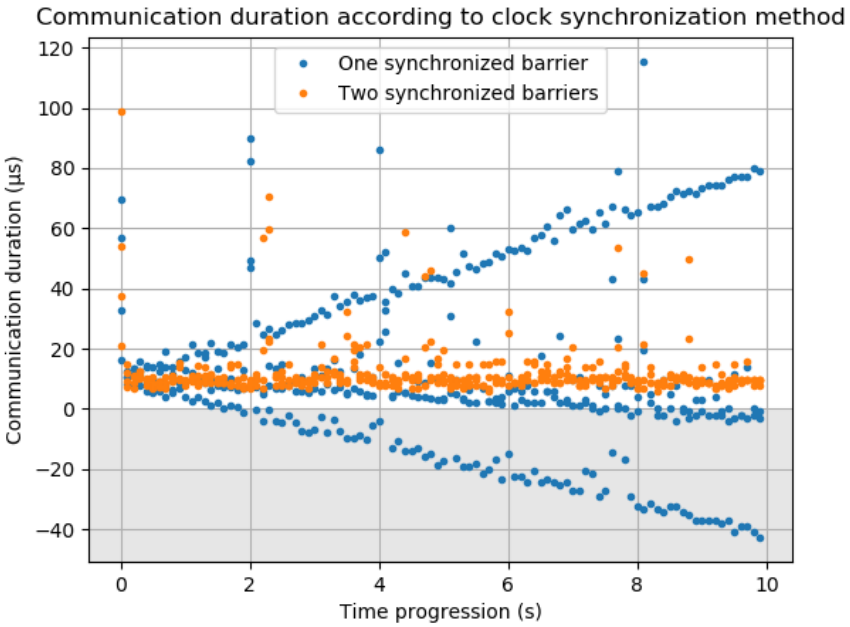


Figure 3.16: Communications duration over time: two synchronized barriers are required to take into account clock drifts.

Using the right clock

To get a sufficient resolution for the clock and reduce drift, we must use the right clock source. There are multiple sources for clock in a regular computer:

- Real Time Clock (RTC): this is the basic clock available in every computer/AT-compatible machine since 1984. It has been considered legacy for a long time. Its resolution is very poor.
- ACPI_PM: this is the clock device from the ACPI Power Management specification [120]. The frequency is hardwired to 3.579545 MHz which makes it a better resolution than RTC but still poor to timestamp events in the range of the gigahertz. It is considered legacy as a clock source.
- High Precision Event Timer (HPET) [72]: this is a clock that was introduced especially to get a precise and steady clock source. It guarantees a resolution higher than 10 MHz. Its resolution is usually average and it costs a system call to be read.
- Time Stamp Counter (TSC): this high resolution timer was introduced [71] in the Intel Pentium Pro family. It is synchronized with the instruction counter and thus has a resolution sufficient for tracing. Moreover, it is very cheap to consult, using a single unprivileged instruction. However, it is not guaranteed to be synchronized between cores, and its frequency varies with the processor frequency (Turbo Boost, energy saving, *etc*), which makes it an unreliable source for timing.
- Invariant TSC: more recent CPUs feature an *invariant* TSC, which is based on the Always Running Timer (ART), running at the crystal clock frequency. This flavor

of TSC is synchronized between cores and uses a constant frequency, even when the CPU changes its frequency for energy saving.

Hence, the most relevant clock to use for traces is invariant TSC, when available. It has a good resolution, is steady, and cheap to consult. When it is not available, the second-best choice is HPET, though the resolution is usually much lower.

To get access to the clocks, multiple interfaces are available. The old `gettimeofday` interface cannot express high resolution, and the clock it provides access to is affected by NTP adjustments, which makes it not steady. It is not suitable for our use case.

Then, there is the direct access to a hardware clock. However, direct reading from `/dev/hpet` is slow (involves a system call) and usually reserved to root. The direct read of TSC (with the `rdtsc` instruction) is fast but is non-portable and requires non-trivial code to check its properties (mostly whether it is invariant TSC or not).

The best solution is to use the POSIX.1-2001 function `clock_gettime` that gives access to various clocks of the system: `CLOCK_REALTIME` is actually the same clock as `gettimeofday` and should not be used for tracing; `CLOCK_MONOTONIC` is fast and monotonic, derived from the default system clock (usually TSC, sometimes HPET if TSC is not invariant) but affected by NTP (no jumps, but not steady); `CLOCK_MONOTONIC_RAW` is a direct portable way (although Linux-only) to get direct access to the default clock.

The main goal of NTP is to compensate for the drift between the local clock and the real time. It adjusts the rate of local clock to reach sync with the root clock, thus locally it causes small fluctuations in the speed of the local clock. Therefore, to avoid those fluctuations, it is relevant to use a raw clock when we compensate for the drift ourselves, since we do not need features from NTP and it would only add noise in the clock. However, in cases where we use a single barrier with offset, we need to use a regular clock (with NTP roughly compensating for the drift) since we do not compensate for the drift ourselves.

As a consequence, we use `clock_gettime(CLOCK_MONOTONIC)` when using a single offset, and `clock_gettime(CLOCK_MONOTONIC_RAW)` when interpolating between two synchronized barriers.

Protection from preemption between synchronized barrier and event probe

As mentioned earlier, one of the most efficient ways to have an event recorded at the exact same time on all nodes is to perform a synchronized barrier and then execute the probe corresponding to the synchronizing event. In the code, it can look like this:

```
1 mpi_sync_barrier();
2 _STARPU_MPI_TRACE_BARRIER(rank);
```

Since the barrier is synchronized, we are sure each process will leave the barrier function call at the same time and should execute the next instruction (here: the trace probe to later compute clock offsets) also at the same time. However, a thread preemption can happen between the end of the barrier call and the trace probe: this could prevent the event from being recorded at the exact same time on each distributed process. To avoid

such problem, the solution is to get the local time the barrier is waiting to unblock, use this time as an additional data for the trace probe, and then use this time as the event timestamp, instead of the timestamp set by the tracing library to compute the clock offset:

```
mpi_sync_barrier(&local_sync_time);
_STARPU_MPI_TRACE_BARRIER(rank, local_sync_time);
```

In fact, the interesting value to correctly synchronize distributed clocks is not the timestamp of the event indicating this synchronization, but the local time the barrier was waiting for.

Even if it can seem obvious, it is important to mention that the clock used for timestamping events and the one used for the time as additional data of the synchronized event have to come from the same clock and have the same origin. Since timestamping events is done by the tracing library and the synchronized barrier is done by another library, this might not be trivial, and requires proper support: either being able to specify the same clock to be used by the both libraries or to convert the time given by the synchronized barrier to the one used by the tracing library.

Compute clock offsets to adjust event timestamps

A synchronized barrier followed by a recorded event is done in the StarPU initialization and release phases, thus delimiting the application execution requiring events with precise timestamps. In order to merge the distributed traces, `starpu_fxt_tool`, the tool in charge of converting raw trace files to exploitable ones, now has two tasks to achieve: computing the clock offsets between the nodes from the events of the synchronized barriers, and then interpolating the clock offsets to apply the timestamp adjustment on each event. Figure 3.17 describes how the clock offsets corresponding to one synchronized barrier are computed:

- Clock origins can be different on each node, because the clocks we are using have as origin the start of the node. Then, the application does not start at the exact same time on all nodes (the instruction to start the application is not synchronized to be executed at the exact same time on all nodes). In the runtime system initialization, once all distributed processes are launched, we execute a first synchronized barrier to record the event used as reference point by all processes.

- **First step**: we consider the event following the synchronized barrier as the local time origin. Thus, the synchronized time $\widetilde{t_i}$ of an event on process i with a timestamp t_i is: $\widetilde{t_i} = t_i - t_i^{\text{Barrier}}$. This also allows having small timestamp values, easier to analyze, since the clock origin will now be the beginning of the application execution.

- **Second step**: the previous step makes the events occurring before the barrier having a negative adjusted timestamp. To avoid this, we shift the time origin, to the first application start, which is the maximum of the distances between application starts and synchronized barriers. Now the transformation to apply is: $\widetilde{t_i} = t_i - t_i^{\text{Barrier}} + \max_j\{t_j^{\text{Barrier}} - t_j^{\text{Start}}\}$.

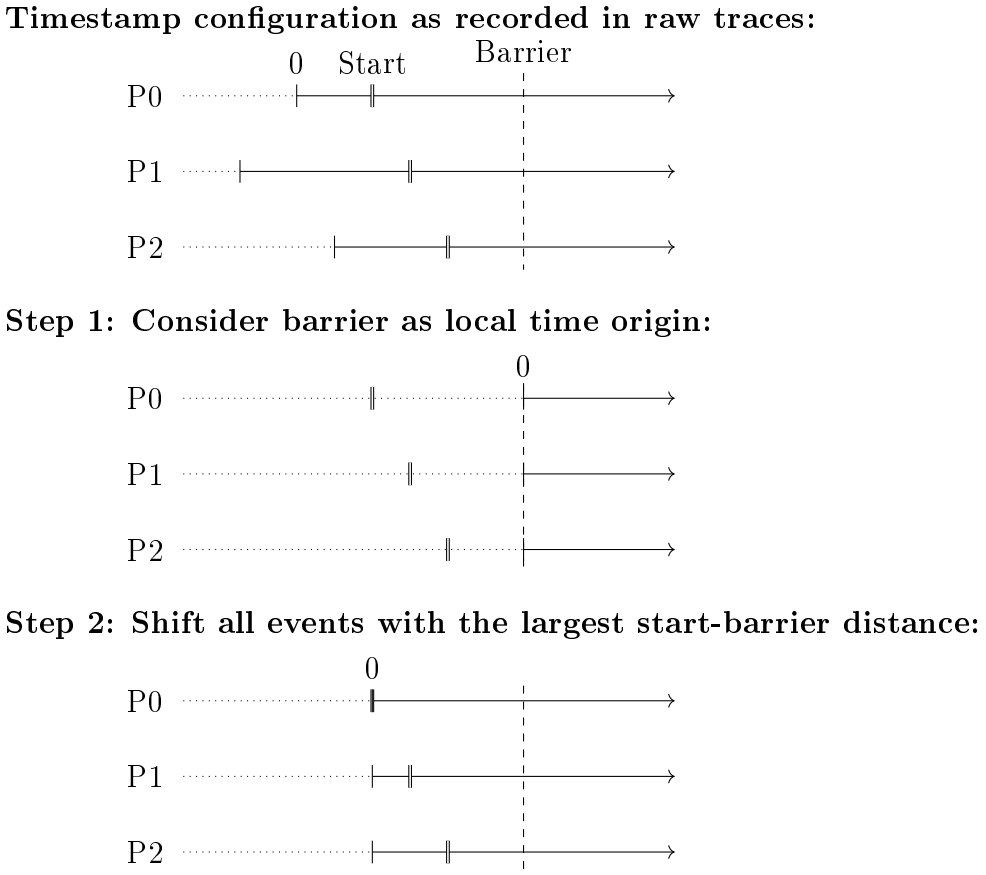


Figure 3.17: Clock offset computation.

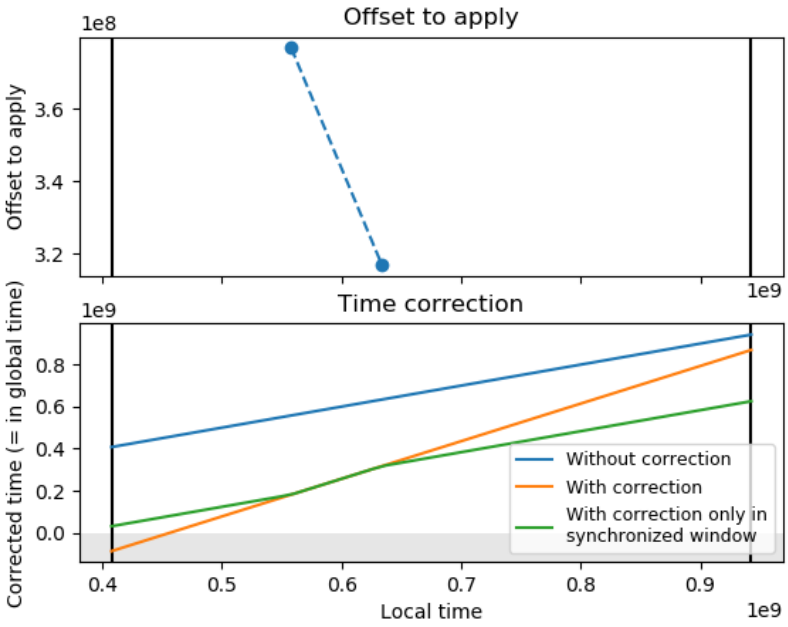


Figure 3.18: Interpolating offsets outside of the synchronized window can lead to negative timestamps.

As one may notice, there is no one node designated as having the reference clock, before beginning computing clock offsets.

When two synchronized barriers are used, we follow the same steps: the two barriers give two sets of clock offsets between nodes, and then the offset to apply to an event timestamp is linearly interpolated. Outside of the interval delimited by the two barriers, we cannot extrapolate the clock offset, because there is no guarantee a linear extrapolation will fit the real clock drift. Thus, in some cases, applying the extrapolated offset can lead to negative timestamps. Figure 3.18 illustrates this phenomenon. The black vertical lines represent the first and last events recorded in the trace. The synchronized barriers and the offsets computed for each of them are represented by the blue dots on the upper plot. The time range between these two dots is the range where interpolating clock offsets gives valid results, while extrapolating clock offsets outside of this window can lead to negative timestamps, as illustrated by the orange curve on the lower plot, before local time of 0.45. If we only interpolate offsets in the synchronized window and use the clock offset of the first synchronized barrier for events before it and the clock offset of the second synchronized barrier for events after it, we get acceptable timestamps even outside the synchronized window, as illustrated by the green curve. The resulting inaccuracy is acceptable outside the synchronized window, because timestamp of events outside of the window do not have to be very precise, they are usually more descriptive than indicative of a state change.

3.4.3 Conclusion on synchronizing distributed traces

In this section, we presented why we need clock synchronization to trace distributed executions and how we ended up with a working solution within StarPU.

The first motivation for clock synchronization is the use of clocks having as origin the start of the node to timestamp events. A coarse-grain synchronization can be done with a classic MPI barrier. With a real use-case, the presented experiences show, and confirm what is reported in the literature, that a simple MPI barrier is not precise enough to synchronize clocks, and the clock drift has to be taken into account by interpolating the clock offsets between two reference points. The given implementation details specify which clock we use, how we avoid thread preemption between synchronized barrier and event probe, and how we compute clock offsets during post-processing to merge distributed traces.

The solution we implemented in StarPU is valid while the clock drift is linear, *i.e.* only for short executions. This is not limiting, since we usually only trace short executions, to ease their analysis. We do not use logical clocks, because it would require lots of development, and the correction of chronological order of events would not reach the precision we are looking for in distributed executions.

3.5 Lessons learned

This section summarizes the insights learned by exploring the issues previously described to deal with when tracing applications: it proposes a methodology to apply to efficiently trace applications and a list of requirements for a satisfying tracing system.

3.5.1 Methodology to apply when tracing applications

As observed previously, the overhead caused by tracing systems can impact application performance, and even modify the behaviour we want to observe in the *post-mortem* analysis. Thus, the user has to be sure that the potential tracing overhead is acceptable regarding the purpose of the traces. To do so, the best way is to compare the application performance with and without tracing it. If the impact on performance is too important, the user can try to reduce it with several methods:

- Set a correct trace buffer size to avoid applications being disturbed by the flushes on the disk;
- Reduce the number of recorded events, by focusing only on important ones;
- If it appears to be a source of overhead, reduce the number of workers. However, this can also change the application behaviour.

Reducing the number of events to record is also useful to reduce the size of trace files, making them much more convenient to manipulate (to transfer between clusters, to convert and exploit, for instance).

If the user is interested in precise timestamps of events occurring on several nodes, using a synchronized barrier to synchronize trace files is a good solution to trace fast executions ($\mathcal{O}$(several seconds)). An easy way to verify the clock synchronization is correct is to compare the observed communication duration to the theoretical duration (for instance, on InfiniBand networks, a transfer of small data lasting less than 1 μs might be suspect).

3.5.2 Requirements for an efficient tracing system

All these highlighted possible problems appearing when tracing applications allow us to suggest a list of requirements for an efficient tracing system. Of course, such systems have to feature the lowest performance overhead and be precise enough, which can be possible with:

- A good scalability of the number of cores recording events: this excludes all general locks protecting a resource accessed by all cores at the same time;
- Accurate-enough timestamps of events, especially on distributed executions, but this is also true if each core records events in its own buffer: at the end, events have to be presented ordered to exploit the trace;
- A good control of trace buffer flushes on the disk, to avoid them occurring in the traced region of interest;
- A user-friendly system to easily filter which events have to be recorded, to reduce the tracing overhead and the trace file size;
- A completely transparent tracing mechanism from the point of view of the user application: only the runtime system has to be aware whether tracing is enabled or not;
- Regarding the integration of a tracing system in the runtime system, the runtime developer has to be careful about where to put the probes in the code, to avoid overloading critical paths. Also, if the tracing system allows it, it might be interesting to distinguish events requiring a timestamp (state changes, action which duration is an important information, *etc*) from others, usually more descriptive (for instance the dependencies of a task, which scheduling policy is selected, *etc*), since the latter do not require to be correctly ordered to keep a temporal coherency. Thus, it can reduce the contention of resources in charge of respecting the temporal order of events.

3.5.3 Extension to other runtime systems

We discussed previously the specificities of task-based runtime systems when it comes to tracing their executions: especially information about the application DAG and its scheduling has to be recorded in the trace. However, the majority of phenomenons covered in this chapter is also valid in the context of other runtime systems than task-based ones.

We took the occasion of improving the tracing system used by StarPU to evaluate its capabilities, but in the end, the reported issues are not specific to task-based runtime systems: performance overhead caused by buffer flushes or the number of events to record has to be considered with every tracing system, the number of cores recording events has to be considered for parallel applications, while distributed synchronized clocks is a general problem when tracing distributed applications.

3.6 Conclusion

Tracing application executions helps to understand their behaviour and improve them, it is a valuable technique given the current complexity of supercomputers. We presented how we integrated a tracing layer in the StarPU task-based runtime system.

We evaluated the different sources of performance overhead coming from the tracing system: writing the event buffer on the disk during the execution, the number of recorded events, and of recording cores, are all responsible for penalizing the application performance. The tracing overhead can depend of the application behaviour, but can also change it; thus the user should always try to minimize the overhead caused by the tracing system, or at least be aware of the performance difference when traces are recorded.

We improved the clock synchronization mechanism in StarPU by implementing state-of-the-art techniques, and observed the different timestamp accuracy when different techniques are used to synchronize clocks. This work also confirms that using accurate synchronized clocks is necessary when dealing with distributed applications.

From the observations and conclusion we made, we proposed a methodology to follow when an application is traced, to minimize the phenomena we described; and we suggested a list of features a tracing system should implement to ease the application integration and user manipulation.

Finally, even if our feedback comes from experiments with StarPU and FxT, most of our observations and conclusions are valid also to trace applications that do not rely on a task-based runtime system.

Chapter 4

Dynamic Broadcasts

We have seen in Chapter 2 that most task-based runtime systems rely on the MPI standard for network communications, while MPI libraries and the MPI interface do not fit with working of task-based runtime systems. Sending the same data to several recipients in a optimized manner is not as straightforward as it can be in BSP applications where calling the right MPI function at the right place is enough. Indeed, the requirements to be able to call the appropriate MPI function are not met in the StarPU programming model. Nonetheless, optimizing the broadcast of a data to several nodes is important for the scalability of the application when the number of nodes (thus the number of recipients) increases.

In this chapter, we propose an algorithm for a *dynamic broadcasts*, coping with StarPU's constraints: only the root knows the list of all recipients, and recipients do not have to know in advance whether data will arrive through a broadcast or a point-to-point operation, while still being able to leverage optimized tree-based broadcast algorithms.

This chapter presents in detail the incompatibilities between task-based runtime systems featuring the STF model and the MPI interface, explains the working of dynamic broadcasts, and evaluates the performance improvements they bring.

4.1 Broadcasts in dynamic task-based runtime systems

In task-based applications, a given piece of data may be a dependency for multiple tasks (a vertex with several outgoing edges in the DAG). If the receiving tasks are located on different nodes, the same data will have to be sent to multiple nodes. This communication pattern is generally known as a *multicast*, or a *broadcast* in MPI speaking, which is a kind of *collective* communication.

The naive way to perform a broadcast is to send data from the root to each node using independent point-to-point transfers. With such an implementation, the duration of a broadcast is *linear* with the number of nodes. MPI libraries usually implement much better algorithms [95, 118, 101] for `MPI_Bcast`, such as binary trees, binomial trees, rings, pipelined trees, or 2-trees, which exhibit a *logarithmic* complexity with the number of

nodes. It is thus strongly advised to use `MPI_Bcast` to broadcast data when possible.

However, for task-based runtime systems that dynamically build the DAG, such as StarPU, nodes do not have a global view of data location and do not synchronize their scheduling. This makes the use of `MPI_Bcast` or `MPI_Ibcast` difficult and inefficient, for the following reasons:

Detection. All the information the runtime system knows about data transfers is the DAG. A broadcast appears as a task whose result is needed by multiple other tasks. However, in general the whole DAG is not known statically but generated while the application is running. Therefore, the runtime system cannot know whether the list of recipient is complete or if another recipient will be added later.

Explicit. The `MPI_Bcast` function has to be called explicitly by the sender and all the receivers. Therefore, each receiver node has to know in advance whether a given piece of data will arrive through an `MPI_Bcast` or a point-to-point `MPI_Recv` communication. Application programmer cannot give any hint, since communications are driven by the DAG, and thus depends on where tasks are mapped during the execution.

Communicator. The `MPI_Bcast` function works on a *communicator*, a structure containing all nodes taking part in the broadcast (sender node and recipients). The construction of a communicator is also a collective operation: to build it, each node participating in a communicator must know the list of all nodes in the communicator. Thus, if we build a communicator containing a specific list of nodes for a given broadcast operation, all nodes have to know the list of all nodes participating in the broadcast.

Yet, the runtime system on a node only has a local view of the task graph: receiver nodes know which node will send them the data, but they do not know all other nodes which will also receive the same data. Hence, building an MPI communicator is impossible without first sending the list of nodes to all nodes, but that would mean we need a broadcast before being able to do a broadcast! In our case, communicators cannot be built before broadcast executions (for instance, during application initialization), because the broadcasts, and thus the nodes taking part in those broadcasts, are discovered dynamically, during the execution of the DAG. Moreover, each node can be involved in different broadcasts composed of different sets of recipients: this would require a communicator for each broadcast with different recipients.

Synchronization. Even if we use a non-blocking `MPI_Ibcast` instead of a blocking `MPI_Bcast`, it works on a communicator. The creation of a communicator with the precise set of nodes has to be performed by all nodes at the same time. Moreover, a single communicator creation may take place at the same time. This means broadcasts, and their associated communicator creation, must nonetheless be executed in the same order by all nodes, which implies some kind of synchronization to agree on broadcast scheduling, thus hindering one of the most important feature of distributed task-based runtime system: its ability to scale by avoiding unnecessary synchronization.

As a consequence, the mechanisms needed to actually use the `MPI_Bcast` function to broadcast data in a task-based runtime system are likely to cost more than the benefit brought by the use of an optimized broadcast.

The general problem is being able to use desynchronized and optimized broadcasting algorithms, without all nodes of the broadcast know each other, and even the root node cannot know when the list of recipients is complete. Next, we present the solution we developed to achieve this goal.

4.2 General concepts of dynamic broadcasts

Our algorithm for dynamic broadcasts is comprised of two parts: the detection of broadcasts by the task-based runtime system, and the broadcast itself in the communication library.

4.2.1 Broadcast detection

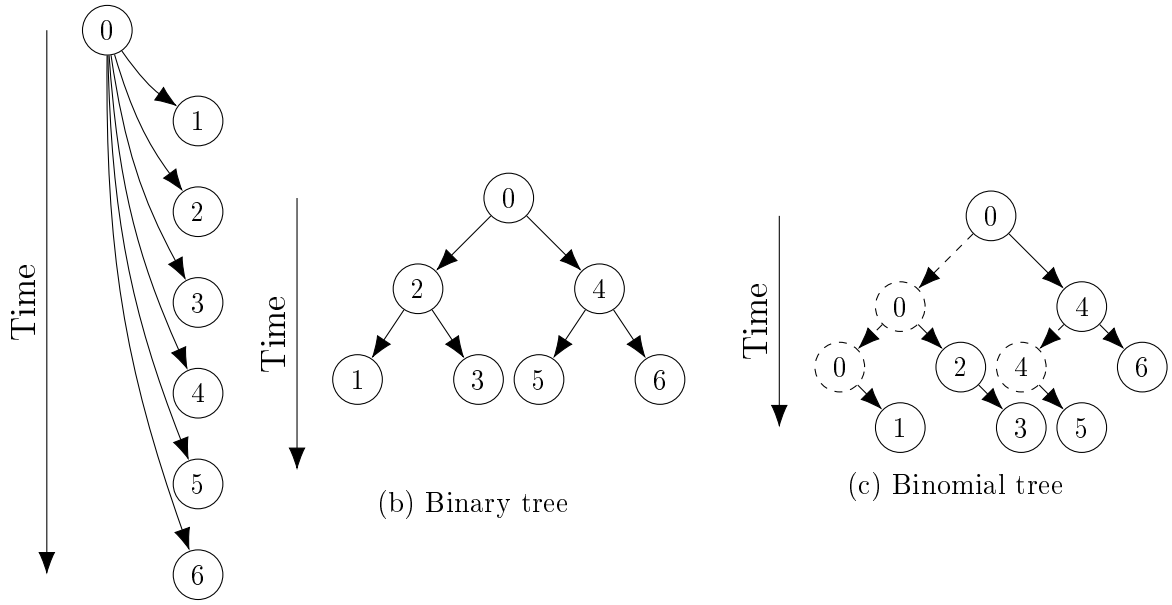
As explained previously, the detection of broadcast patterns is not straightforward since the DAG is dynamic.

From the dependency graph view, a broadcast is a set of outgoing edges from the same vertex and going to tasks executed on different nodes. During task graph submission, the runtime system creates a send request for each of these edges, even before the data to send is available. When the data becomes available, the requests are actually submitted to the communication library to actually execute the communication.

The detection of broadcast consists in noticing on creating a send request that one already exists for the same data, and aggregating them into a single request with a list of recipients. When the data becomes available (*e.g.* the task that generated this data is finished), if the list contains more than one recipient, a broadcast is submitted to the communication library.

This method may miss some send requests if they are posted after the data became ready, *i.e.* if a task is submitted after the completion of the task that produces the data it depends on. This happens if the task graph submission takes longer than the task graph execution (which is not supposed to happen in general), or if the application delays submission of parts of the task graph for its own reasons, in which case the runtime system did not need to send this data sooner anyway. Code instrumentation showed that 98 % of broadcasts were detected with the correct number of recipients for the Cholesky decomposition (described latter). The 2 % missed broadcasts correspond only to communications performed during the very beginning of the application execution, when the application has only started submitting the task graph, and thus task execution has indeed caught up quickly and made some data available before the application could submit all inter-node edges for them. Quickly enough, tasks submission proceeds largely ahead of tasks execution, and all broadcasts are entirely detected.

To avoid redundant transfers of the same data between two nodes, a cache mechanism is used, as explained in Chapter 2. If two tasks scheduled on the same node need a piece of data from another node, only one communication will be executed. Hence, the recipient list does not contain duplicates.



(a) Flat tree

(b) Binary tree

(c) Binomial tree

Figure 4.1: Examples of routing trees for 6 recipients. Levels in the tree are steps in the algorithm.

4.2.2 Dynamic broadcast algorithm

We propose here a broadcast algorithm, that we call *dynamic broadcast*, that fulfills the requirements to be used by task-based runtime systems, namely: use optimized broadcast algorithms; all recipients of the broadcast do not have to know each other; have a seamless integration for the receiver who is expecting a point-to-point communication.

Optimized broadcast algorithm

Several optimized algorithms for broadcast exist [123]. The main idea of all these optimized algorithms is that after a node received data, it sends this data to other recipients, so that the root node has less communications to execute, which shortens the global execution time. For most algorithms, routing is organized as a tree: the source node sends to a set of nodes, each of these nodes then sends to a set of other nodes, and then recursively until all recipients get the data. Tree-based algorithms have usually a logarithmic complexity in the number of nodes. The choice of a broadcast algorithm depends mainly on the number of recipients and the size of data to transmit.

Figure 4.1 illustrates three routing algorithms for a broadcast to 6 recipients, initiated by the node 0. With a flat tree (Figure 4.1a), the naive broadcast algorithm, the node 0 sends sequentially the message to one of the recipients. The complexity of this algorithm is linear with the number of recipients and involves the node 0 for the whole collective duration.

With a binary tree (Figure 4.1b), each node sends the message to at most two another

nodes in a row: node 0 sends to nodes 2 and 4, and the job of 0 is done.

In the binomial tree algorithm, each node receiving data contributes to the diffusion by sending data to next nodes, and keep sending data to other recipients until all nodes received the data. In the example depicted by Figure 4.1c, node 0 starts by sending to node 4, then 0 sends to 2 and at the same time 4 sends to 6 and finally while 0 is sending to 1, 2 is sending to 3 and 4 is sending to 5.

The binary tree has a worse complexity in the number of recipients (yet still logarithmic: in the example with 6 recipients, the binary tree as only one additional step compared to the binomial tree), but has the advantage to require less handling time from each forwarding node, and thus can make the received data available to the application faster. In the other hand, binomial trees are a good trade-off for a single all-purpose algorithm to get good performance on a wide range of data sizes and numbers of nodes. Other optimized algorithms [118, 101, 95] could be used in our dynamic broadcast, following the same approach.

Self-contained messages

Since nodes do not know in advance whether they will be participating in a broadcast, our algorithm is based on self-contained messages. They are processed upon reception, outside of the application flow, without requiring the application to call specific primitives in the communication library. The message contains all the information needed to unroll the collective algorithm.

Only the root of the broadcast knows the complete list of recipients. Recipients themselves only need to know to which nodes they will need to forward the data, *i.e.* the sub-tree below them. We send this list of nodes together with data, in the header of the message. When a node forwards data to other nodes, it trims the list of nodes so as to include only the nodes contained in the relevant sub-tree.

In the case depicted in Figure 4.1c, the list of nodes sent by node 0 to 4 is $\{5, 6\}$, the list sent to 2 is $\{3\}$ and the list sent to 1 is empty.

The general idea behind this mechanism is that routing information are transmitted with the data itself, and are not assumed to be prior knowledge, as `MPI_Bcast` would otherwise require.

Transparent receive

When a request which is part of a broadcast is received, the data is first forwarded to nodes contained in the list (following the chosen routing algorithm), and then delivered locally to the runtime system. Since nodes cannot predict whether data will arrive through point-to-point communication or through a broadcast, on the receiver side our algorithm injects data received by a broadcast in the path of point-to-point receive. The runtime system posts a regular point-to-point receive request, and when data arrives through a dynamic broadcast, it is actually received by this point-to-point request for a seamless integration.

We called our algorithm *dynamic broadcast* because nodes realize they take part in a broadcast in a dynamic fashion, on the fly at the same time as data arrives.

4.3 Implementation

Dynamic broadcasts were implemented as a new interface of NewMadeleine, and the NewMadeleine backend of StarPU was adapted to exploit this new interface.

4.3.1 Broadcast detection

The detection of broadcasts is implemented in StarPU. When the application submits a task B which depends on data produced by a task A mapped on a different node, an inter-node communication request is issued. If a previous request or collective was already detected for this data, the new request is merged in to get a bigger collective.

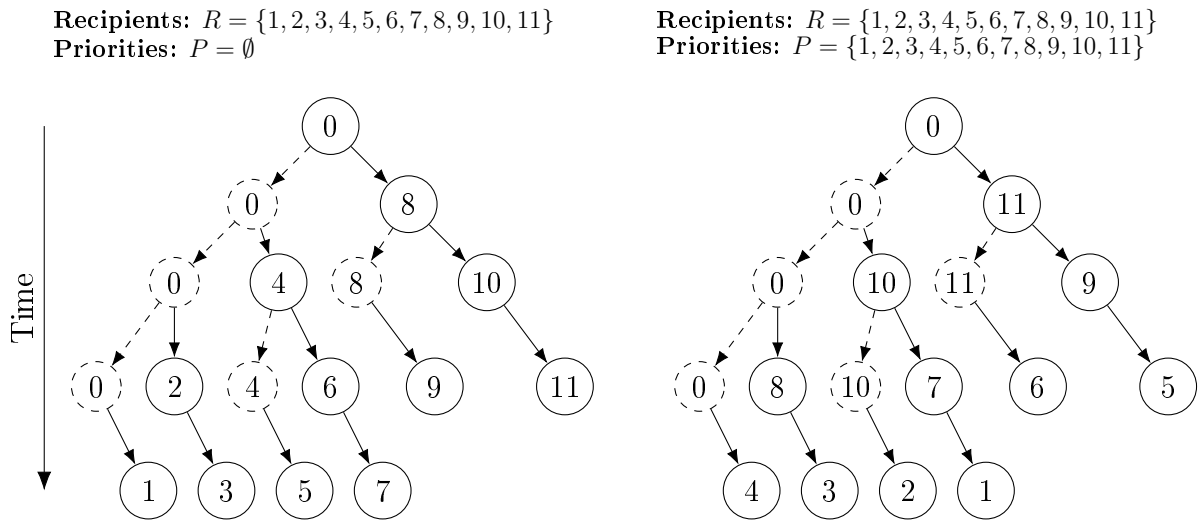
Most often, task submission proceeds quickly, and thus the submission front is largely ahead of the execution front. As a consequence, when task B is submitted, task A will probably not have been executed yet, and similarly for all tasks which depend on A. This is why our approach catches most potential for broadcasts. Once task A is completed and thus the data available for sending, the whole collective request is handed to NewMadeleine.

Once the broadcast has been triggered, if additional tasks which require the data produced by the task A are submitted and need data transfers, these communications will be executed as point-to-point communications. Our mechanism catches only pending transfers inferred while the data is not available.

4.3.2 Dynamic broadcast interface

The dynamic broadcast itself is implemented in NewMadeleine with a specific interface called `mcast`. It is using its non-blocking RPC interface to call functions on remote nodes, with arguments to express the remaining of the broadcast. They use a dedicated communication channel that is separate from the channel used for point-to-point communications. Thus, the library distinguishes broadcasts, which needs special processing, from regular point-to-point messages. The library is always listening for RPC requests and is thus able to always process dynamic broadcasts for all tags and from all nodes.

When a broadcast is received, the matching point-to-point receive is searched and data is received in-place in the buffer of the point-to-point request, forwarded to nodes in sub-trees, and the completion of the point-to-point request is notified. If the matching point-to-point receive was not posted yet, the broadcast request is locally postponed until the matching point-to-point receive is posted. To be able to match a message arriving through a broadcast with a point-to-point request, the original source node (root of the broadcast) is also part of the broadcast metadata.



(a) Without priorities

(b) With priorities: nodes with higher priority have to receive data first

Figure 4.2: Tree reordering to take into account communication priorities. Message to recipient $R[k]$ has a priority $P[k]$.

4.3.3 Enforcing communication priorities

StarPU sets a *priority* level for each communication request, depending on task priorities, defined by the application: during the submission of tasks, the user can define the priority of each task, by specifying an integer. This information may be used by a communication library that is able to schedule packets by priority like NewMadeleine.

The different communication requests in a broadcast can have different priorities. To respect the communication priorities, we reorder the list of nodes of broadcasts so that higher-priority requests are closer to the root of the tree, for them to get data earlier. Indeed, the initial list of recipients may not respect the priority order, since the order of insertion in the list is the same of the task insertion order.

Moreover, in addition to the list of nodes, we transmit the list of priorities within the broadcast metadata. This way, when inner nodes of the tree have to forward messages, they get inserted in their local packet flow, managed by NewMadeleine, with the right priority.

Figure 4.2 illustrates the difference in reception order when priorities are set or not. Without priorities, there is no guarantee in the reception order: in Figure 4.2a the node 8 receives data first and nodes 1, 3, 5 and 7 receive it in the last step. When priorities are provided and respected, (in Figure 4.2b, recipient nodes with higher ranks have higher priority), the node 11 receives data first, and nodes 4, 3, 2 and 1 receive it in the last step, as indicated by the priorities.

4.3.4 Using just received data but still being forwarded

With point-to-point communications, as soon as data is received from the network by NewMadeleine, StarPU is notified the data is ready to be used, which can unlock tasks waiting for this data.

When a node receives a data which is part of a dynamic broadcast, this node may have to forward the data to other nodes, as stated by the routing tree. In such case, received data is available on the node, but the application cannot use it while NewMadeleine forwards it to other nodes, because immediately launching tasks unlocked by the data reception may modify the data being forwarded. Waiting for all forwards being finished adds a delay to make the data available to StarPU and unlock tasks, compared to a point-to-point communication.

A trade-off to reduce the waiting time while preserving the integrity of forwarded data is the following mechanism:

1. Unlock tasks which only need the data in a read-only mode as soon as the data is received on the node;
2. Let NewMadeleine forward the data to other nodes if required;
3. Notify StarPU NewMadeleine does not need the data anymore, it can also unlock tasks which need to write in this buffer.

From the implementation point of view, these two states (*data received but still forwarding* and *finished forwards*) corresponds to two distinct events triggered by the communication library, on which are bound the action of unlocking tasks, which only read the data or also write it.

4.4 Evaluation

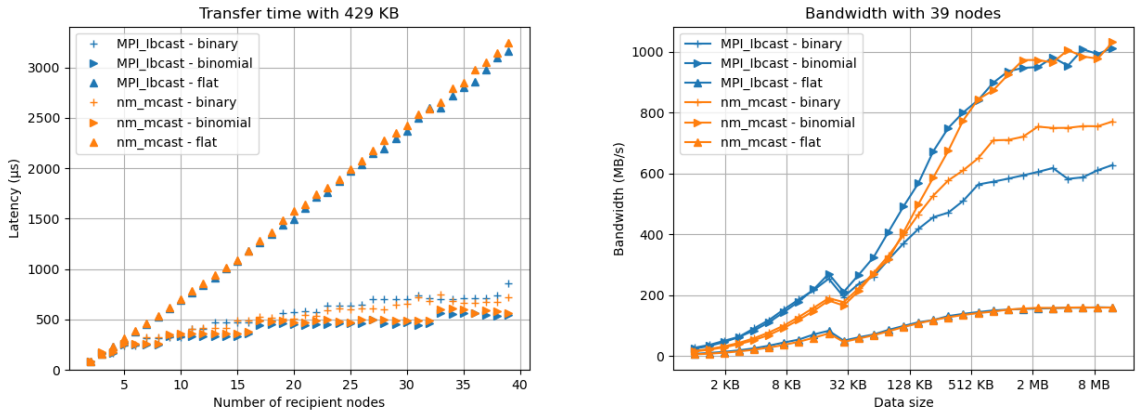
In this section, we present the performance results we obtain for mechanisms presented earlier. We executed microbenchmarks to ensure the broadcast performance is as expected and then we evaluated the impact on two real computing kernels, the Cholesky and QR factorizations. Other applications could take benefit from dynamic broadcasts, such as matrix multiplications [13].

The majority of results depicts executions on the `occigen` machine, described in Appendix C.

4.4.1 Microbenchmarks

General performance

To be sure our algorithm and its implementation have the expected behaviour and performance, we conducted microbenchmarks of the dynamic broadcast and compared its performance to the one of `MPI_Ibcast` from MadMPI (the MPI interface of NewMadeleine),



(a) Latency (b) Bandwidth

Figure 4.3: Microbenchmarks of the dynamic broadcasts with NewMadeleine, on `occigen`.

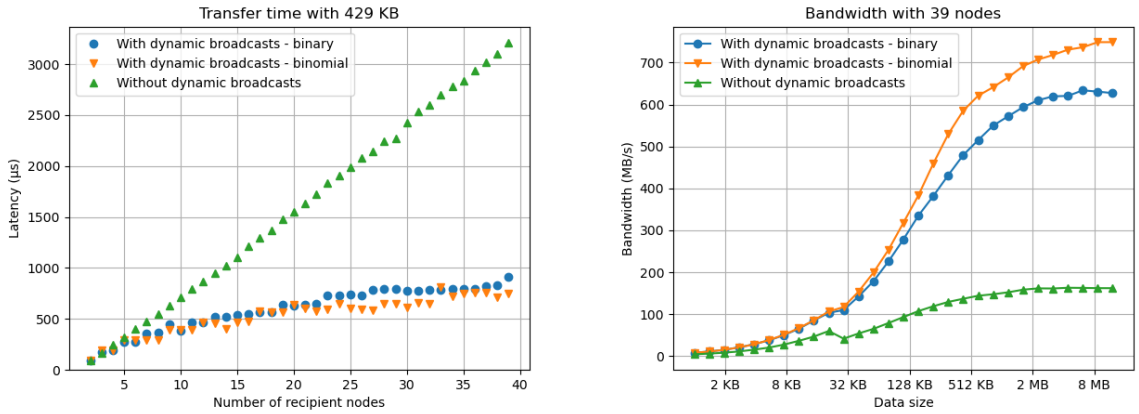
and a naive broadcast (a loop of point-to-point requests to the recipient nodes; *flat* in the legends).

We considered the same metrics as NetPIPE [109]: the latency and the bandwidth of broadcasts. The latency of the broadcast is defined as the time difference between the start on the root node and the last received data on the last node. With a constant message size (429 KB in the following experiments), we increase the number of recipient nodes in the broadcast. The bandwidth reached with a broadcast is the data size divided by its latency. With a constant number of recipient nodes (38 in the following experiments), we increase the size of the broadcasted data.

Figure 4.3 depicts the performance results with only NewMadeleine, to first evaluate its raw performance. The reported latencies on Figure 4.3a clearly show the different complexities of the used routing algorithms. The naive algorithm (flat) is linear with the number of recipients, and both binary and binomial trees have a logarithmic complexity with the number of nodes (we can even notice the different plateau corresponding to the different heights of the routing trees). Broadcasts are faster with binomial or binary trees than with linear sequential point-to-point communications (less than 1000 μs *vs* more than 3000 μs). The same ranking of routing algorithms applies to the communication bandwidth (Figure 4.3b): flat algorithm cannot reach 200 MB/s while binomial trees reach 1000 MB/s when broadcasting data to 38 recipients.

The performance difference between dynamic broadcasts and regular MPI broadcasts is insignificant, which shows that the additional routing data and the treatment when receiving data is negligible.

The same microbenchmarks are conducted with StarPU and NewMadeleine, to evaluate performance when StarPU triggers broadcasts. Results are depicted on Figure 4.4: *Without dynamic broadcasts* means that the detection of broadcasts is disabled in StarPU and point-to-point communications are submitted to NewMadeleine as soon as a communication is detected. Overall results are similar to NewMadeleine alone,



(a) Latency (b) Bandwidth

Figure 4.4: Microbenchmarks of the dynamic broadcasts with StarPU and NewMadeleine, on `occigen`.

except for a small overhead coming from the StarPU management of communications.

Performance with priorities and reading data as soon as possible

To verify that the priorities attached to communication are correctly taken into account, we consider the time of data reception on each node, and not only the maximum used in the previous benchmarks to get the overall duration of the broadcast. Then, we plot this time according to the node rank. First nodes which have to receive data (with a higher priority) should have a smaller reception time.

To correctly represent the impact of priorities, reported times have to distinguish the time when data arrived on the node (influenced by the priority of the incoming communication) and the time when all forwards are finished and data is not needed by the NewMadeleine anymore. Without this distinction, the forwarding duration would be included in the reported time, which could fake the representation of priority impact.

The described benchmark was first conducted with only NewMadeleine executing a program where node 0 sends a broadcast to many nodes. The priorities of communications are equal to the rank of the recipient node (*i.e.* nodes with higher rank should receive data first). Each recipient node records the time when data arrived and when it is released by NewMadeleine. This set of values can be plotted as an area: the X-axis corresponds to the node rank and reported times are read on the Y-axis. Times when data is received are the lower boundary of the area and times when data are completely released from NewMadeleine is the upper boundary. The lower boundary can be used to see impact of priorities and the height of the area is the duration NewMadeleine took to forward the data to other nodes.

Figure 4.5 depicts such benchmark with 38 recipients in the broadcast and a data with a size of 409 KB. Three routing algorithms were evaluated. For the flat tree (node 0 sends sequentially the data to each recipient), nodes receive the data in a reversed order of their

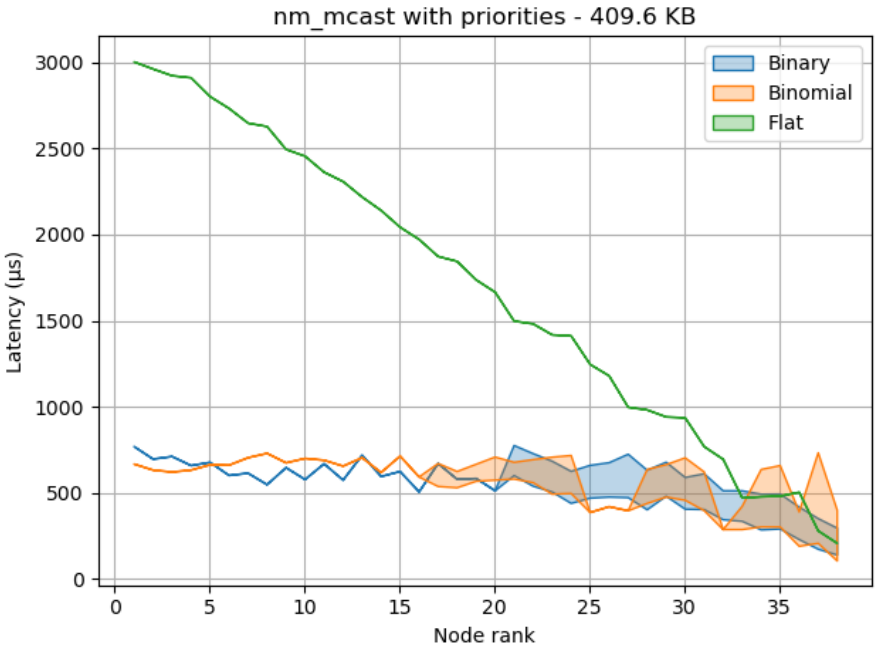


Figure 4.5: Microbenchmark to check the respect of priorities in dynamic broadcasts with NewMadeleine, on occigen.

rank, and since receiving nodes do not forward the data to other nodes in this algorithm, the data is immediately available to the application after it is received: the lower and upper lines of the area are superimposed. When using a binary tree, nodes receiving the data have to forward it to 2 other nodes. We easily see the difference between forwarding nodes and the other leafs of the tree: the former spend time to forward the data, thus there is a delay between receiving the data and delivering it to the application; the latter do not have to forward to other nodes and thus can directly notify the application. Similar behaviour can be noticed for the binomial tree, except forwarding nodes are involved until the end of the collective operation. This is why the height of area is inversely proportional to the node rank (and thus the priority): the deeper a node appears for a first time in the binomial tree, the less time it will spent forwarding to other nodes, since it will forward to fewer nodes, corresponding to fewer steps to reach the end of the collective. The observed small variations in latency of nodes belonging to the same levels in routing trees comes from the system noise and the fact that nodes of a same level are not synchronized: in reality, trees are a little bit distorted (*e.g.* a node can start communications from the next level quicker than a node which just received the data and has to initiate its participation in the broadcast).

In all cases, priority are respected: nodes with higher priority receive data before those with lower priority.

Similar benchmark was executed with StarPU (see Figure 4.6), but the difference between just receiving the data and finishing forwarding is plugged to the task model: to trigger the broadcast tasks which only need to read the data are submitted on all recipient nodes and then tasks which also need to write the data are also submitted on all nodes.

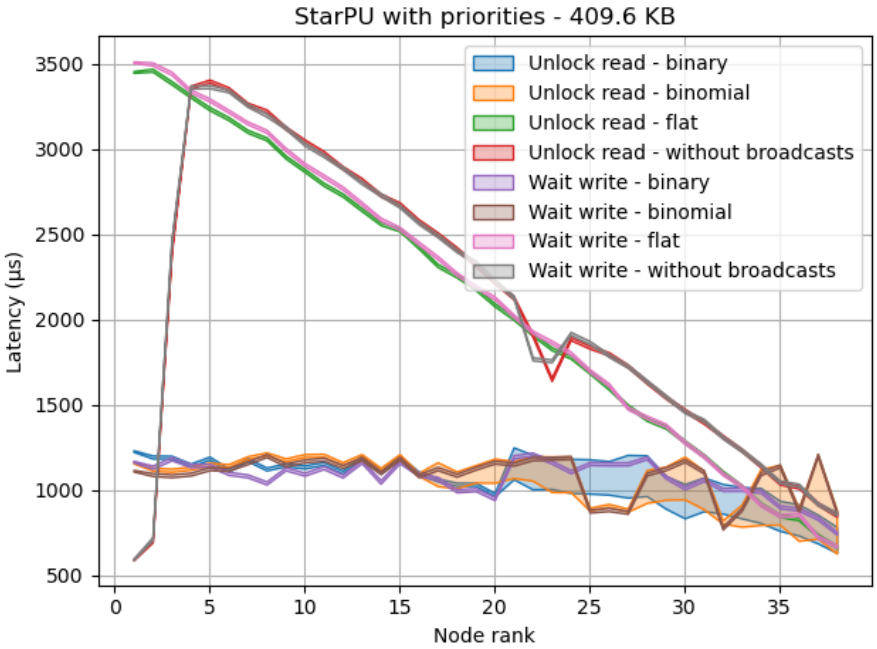


Figure 4.6: Microbenchmark to check the respect of priorities in dynamic broadcasts and impact of unlocking tasks as soon as data is received, with StarPU and NewMadeleine, on occigen.

Actually these tasks only get the current time to know when they started to be executed. The interpretation of the areas can now be different: the lower border represents the start of the reading-only task and the upper border the start of the writing task.

We evaluated two cases: when StarPU is allowed to launch read-only tasks just when data is received (*Unlock read*) and when StarPU has to wait for NewMadeleine to completely release the received data (*i.e.* finish all the forwards) before launching any task accessing to the received data (*Wait write*). Three trees were tested: binary, binomial and flat. An execution with disabled dynamic broadcasts in StarPU are also been performed (*without broadcasts*). With disabled broadcasts, the broadcasts are not detected and StarPU submits each communication request to NewMadeleine as soon as they are detected, NewMadeleine then performs the independent point-to-point communications. With the flat tree, the broadcast is detected by StarPU and submitted as such to NewMadeleine, but NewMadeleine follows a flat tree to execute the broadcast.

The results of this performance evaluation are depicted on Figure 4.6. The general shape of the curves are similar to those with plain NewMadeleine. The *wait write* areas are very thin: they represent the elapsed time between the end of a task (the read-only one) and the beginning of another task (the write one). Since both tasks on each node are unlocked when forwards are finished, these areas match with the top of their *unlock read* counterparts, where StarPU launches the read-only tasks as soon as the data is received and waits for the end of forwards to launch the write task.
The sudden increase in the first nodes when dynamic broadcasts are not enabled can

be explained with the priority strategy followed by NewMadeleine. When dynamic broadcasts are disabled, StarPU submits send requests to NewMadeleine sequentially one-by-one. The order the send requests are submitted correspond to the order of task submission: here, in the order of node ranks, thus inversely proportional to the attached priorities. When NewMadeleine handles the first submitted request (with the lowest priority), there is no other communication with higher priority waiting to be handled, thus NewMadeleine executes this first low-priority communication. This works for the first send requests, but after a short time, all send requests are submitted to NewMadeleine while only the first send requests are finished. When NewMadeleine has to choose again which communication to handle, they are now all waiting, and NewMadeleine can pick the one with the higher priority. When dynamic broadcasts are enabled, NewMadeleine receives directly the whole list of communication requests and thus is aware of the communication with the highest priority.

To summarize, our implementation works as expected: routing algorithms with logarithmic complexities exhibit corresponding performance, priorities are correctly taken into account and unlocking tasks that only need to read the received data allows to launch them sooner.

4.4.2 Cholesky factorization

Description

The Cholesky factorization, described previously in Algorithm 1 (page 24), is a good use-case for the dynamic broadcast problem. Indeed, as shown in Figure 4.7, each $A[k][k]$ tile computed by a `POTRF` kernel is broadcasted to the $N-k-1$ `TRSM` kernels of the same panel (blue arrows). Moreover, each $A[m][k]$ $(m > k)$ tile generated by the `TRSM` kernels, is used by one `SYRK` kernel (to update the tile $A[m][m]$, red arrows) and $N-k-2$ `GEMM`

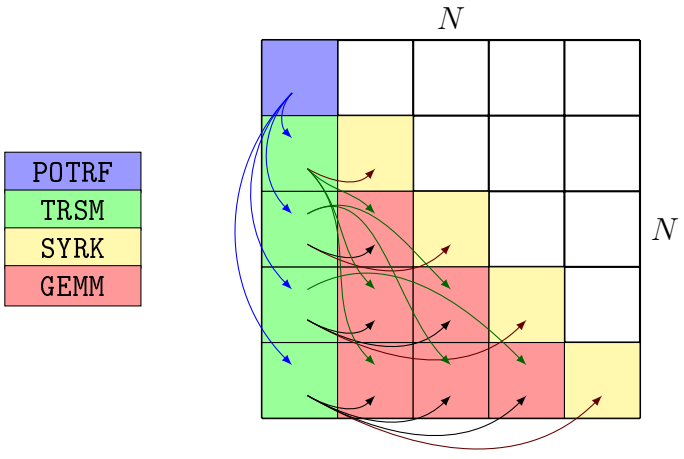


Figure 4.7: The two different types of broadcasts for the Cholesky factorization for $N = 5$ and $k = 0$. Blue arrows: from 1 `POTRF` to $N-k-1 = 4$ `TRSM`. Green and black arrows from 1 `TRSM` to $N-k-2 = 3$ `GEMM` and red arrows to 1 `SYRK`.

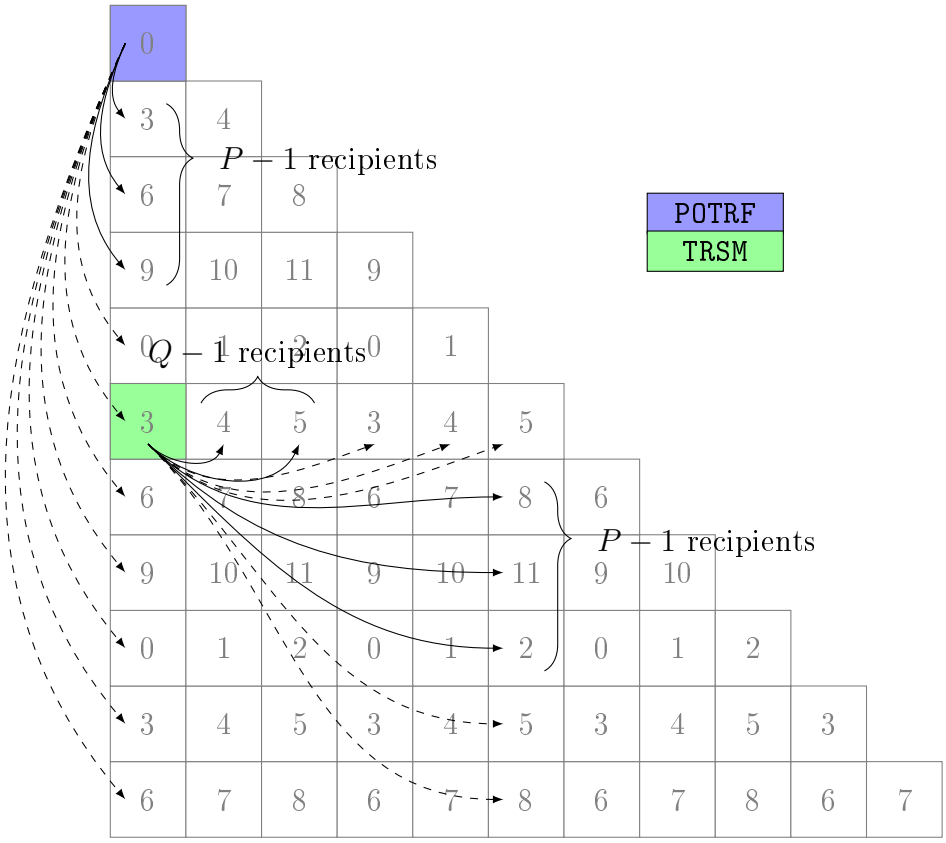


Figure 4.8: Recipients in the two types of broadcasts in the CHOLESKY algorithm on a 2D-block-cyclic distribution $(P, Q) = (4, 3)$. Gray numbers indicate on which MPI process is the tile and dashed curves represent duplicated communications, not done in reality.

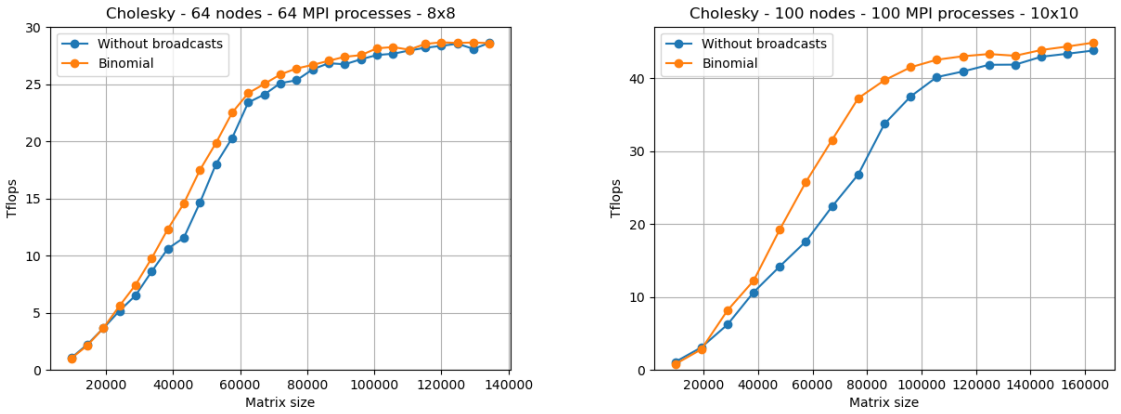
kernels (to update the tiles $A[m][n]$ $(k < n < m)$, black arrows; and the tiles $A[m][n]$ $(m < n < T)$, green arrows).

In practice, as the matrix distribution on available nodes is 2D-block-cyclic with P rows and Q columns (with $P \times Q =$ the number of nodes), the maximum number of recipients of a broadcast in the CHOLESKY algorithm is $P + Q - 2$. Figure 4.8 illustrates this formula: the result of a `POTRF` is broadcasted to all nodes of the same column: with a 2D-block-distribution, this means $P - 1$ recipients in the broadcast. Results of `TRSM` kernels are broadcasted to nodes belonging to the same line ($Q - 1$ recipients) and the column of the most-right recipient of the line ($P-1$ recipients since one recipient is already counted as a recipient in the line): $Q - 1 + P - 1 = P + Q - 2$.

We used the CHOLESKY factorization from the CHAMELEON library [10], which can use STARPU as task-based runtime system.

Results

Two sets of results will be presented. First results use a proof-of-concept implementation of the dynamic broadcasts and were collected on the `inti` machine from the CEA. `inti`



(a) 64 nodes

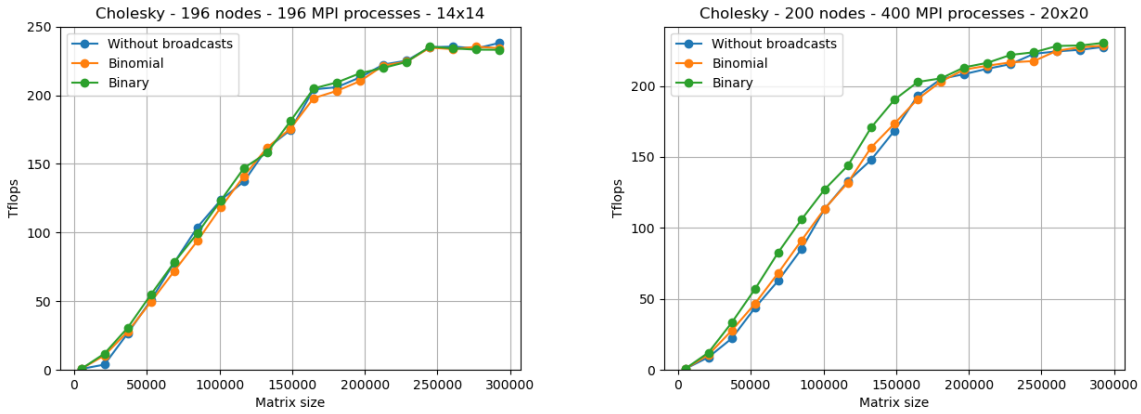
(b) 100 nodes

Figure 4.9: Performance of Cholesky factorization, on `inti` (old implementation).

nodes were dual Intel Xeon E5-2680 at 2.7 GHz, with 16 cores and 64 GB RAM, and equipped with InfiniBand Connect-IB QDR. `inti` was replaced by another machine before a more robust implementation was rewritten from scratch, with additional features (different routing trees, support of priorities and read-only use of received data). The second set of results uses this more recent implementation and comes from executions on `occigen`. Old results on `inti` are presented because they show performance improvements with dynamic broadcasts, more significant than on `occigen`.

Results for machine `inti` on 64 and 100 nodes are shown in Figure 4.9. There is one MPI process per node and each point on graphs is the average of two runs. We compare the *baseline* NewMadeleine version without dynamic broadcast against NewMadeleine with dynamic broadcasts following binomial tree routing. At that time, recipients were not reordered according to priorities and NewMadeleine released the received data to StarPU only once all forwards were finished. For both number of nodes, using dynamic broadcasts improves performance, especially for medium-sized matrices: on 64 nodes the improvement is up to 20 % and on 100 nodes up to 30 %. Since the number of nodes in broadcasts increases with the total number of nodes, the more nodes are used, the more the broadcast takes time, thus dynamic broadcasts improve overall scalability with the number of nodes. For larger matrices, the hypothesis is that communications have less impact since there are always enough ready tasks to execute before having to wait for data coming from the network, hence it is not surprising to observe the best performance improvement for smaller matrices.

Figure 4.10 shows the performance of the current implementation of dynamic broadcasts on the `occigen` machine, which has more recent processors than `inti` with more cores (28 *vs* 14). With 1 MPI process per node on 196 nodes (Figure 4.10a), the use of dynamic broadcasts with binomial or binary trees does not improve performance. However, with 2 MPI processes per node (*i.e.* one per NUMA node) on 200 nodes[1], dynamic broadcasts with binary trees improve performance up to 15 % again for medium-sized matrices. Using 2 MPI processes per node increases the total number of MPI processes and thus creates broadcasts with more recipients: up to 38 recipients[2] *vs* 26 for the case



(a) 196 nodes with 1 MPI process / node

(b) 200 nodes with 2 MPI processes / node

Figure 4.10: Performance of CHOLESKY factorization, on occigen.

with one MPI process per node.

Priorities in the CHOLESKY factorization permits to prioritize tasks and communications on the critical path of the execution (especially `POTRF` kernels) which will unlock many tasks and exhibits wide parallelism. The importance of communication priorities is illustrated by the Figure 4.11. First when dynamic broadcasts are disabled, if NEWMADELEINE does not have priorities to order the communications, the general application performance is very poor: only 151 Tflops compared to the plateau of 228 Tflops reached when priorities are attached to communications. When dynamic broadcasts are used, reordering recipients according to their priorities improves the binary routing but not the binomial one. Probably because with a same number of recipients, binary trees are deeper than binomial trees, *i.e.* recipients may wait longer to receive the data, thus a correct recipient ordering is more important for the binary routing.

Figure 4.12 depicts the impact of unlocking tasks that only read the received data as soon as it is received (*Unlock read*) or waiting the end of the node participation in the collective to unlock any task requiring the data (*Wait write*). Curve when dynamic broadcasts are disabled is not showed, since this feature affects only dynamic broadcasts. Unlocking tasks as soon as possible improves performance with the binomial but not for the binary tree. It was expected: with binary trees, the recipient node forwards the received data to only two other nodes: its participation in the collective is very quick and the delay between the reception of the data and the end of forwards to other nodes is small. On the contrary, when a node starts to contribute to a binomial tree, it is involved for all the next steps of the tree: in this case the delay can be much longer and unlocking tasks which only needs to read the data can be beneficial.

[1] We slightly change the number of nodes to be able to always have a square 2D-block-cyclic distribution (*i.e.* the square root of number of MPI processes has to be an integer).

[2] This is why the microbenchmarks were made to have broadcasts with up to 38 recipients.

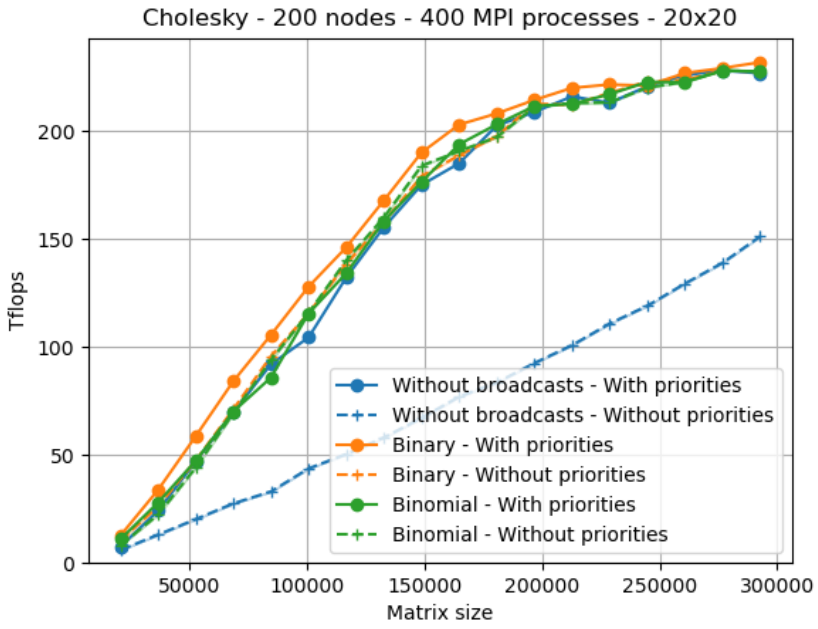


Figure 4.11: Impact of priorities on CHOLESKY factorization, on occigen.

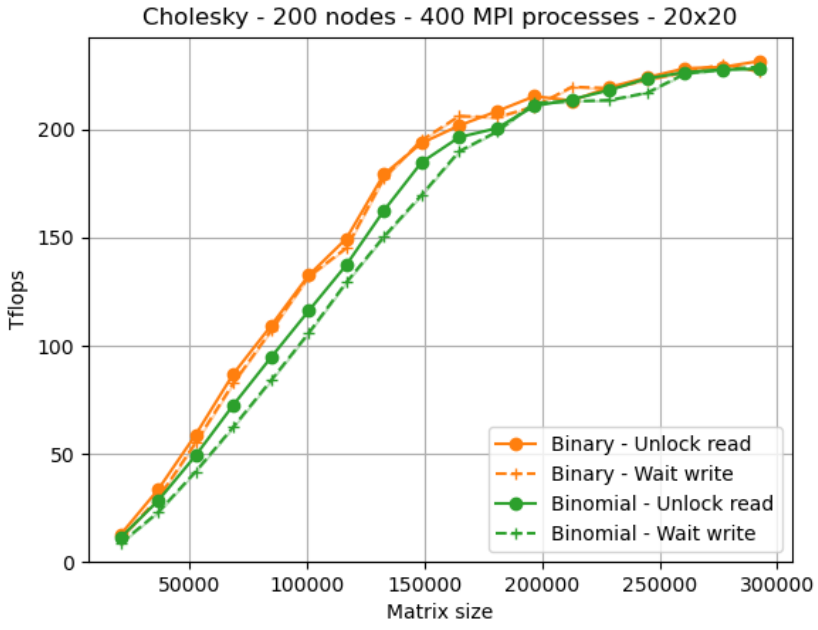


Figure 4.12: Impact of reading data as soon as possible on CHOLESKY factorization, on occigen.

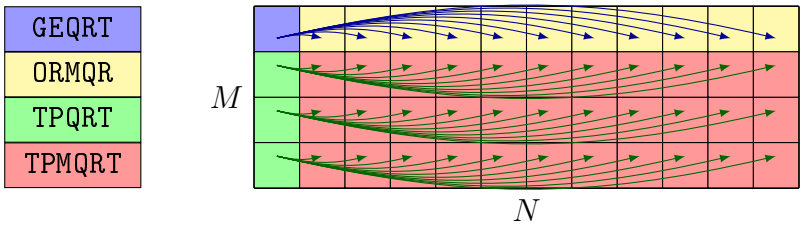


Figure 4.13: The two different types of broadcasts for the QR factorization for $M = 4$, $N = 12$ and $k = 0$. Blue arrows: from 1 `GEQRT` to $N - k - 1 = 11$ `ORMQR`; green arrows: from each of $M - k - 1 = 3$ `TPQRT` to $N - k - 1 = 11$ `TPMQRT`.

4.4.3 QR factorization

Description

The second application used to benchmark the dynamic broadcasts is the QR factorization. From a matrix A, the QR factorization computes an orthogonal matrix Q and an upper triangular matrix R, such as $A = QR$. One difference compared to the CHOLESKY factorization is that the matrix A does not need to be square.

Algorithm 2 Tiled version of the QR factorization.

1: **for** $k = 0$ **to** $\min(M, N) - 1$ **do**
2: `GEQRT`(`RW`, $A[k][k]$, `W`, $T[k][k]$)
3: **for** $j = k + 1$ **to** $N - 1$ **do**
4: `ORMQR`(`R`, $A[k][k]$, `R`, $T[k][k]$, `RW`, $A[k][j]$)
5: **end for**
6: **for** $i = k + 1$ **to** $M - 1$ **do**
7: `TPQRT`(`RW`, $A[k][k]$, `RW`, $A[i][k]$, `W`, $T[i][k]$) ▷ Executed on node (i, k)
8: **for** $j = k + 1$ **to** $N - 1$ **do**
9: `TPMQRT`(`R`, $A[i][k]$, `R`, $T[i][k]$, `RW`, $A[k][j]$, `RW`, $A[i][j]$) ▷ Executed on node (i, j)
10: **end for**
11: **end for**
12: **end for**

We used the version described in the Algorithm 1 of [59], implemented in CHAMELEON, and summarized in Algorithm 2. Like the CHOLESKY factorization, at each step k, the k^{th} tile of the diagonal is factorized (`GEQRT` kernel) and then the right panel is updated accordingly. As illustrated by the Figure 4.13, the two outputs of the `GEQRT` kernels are broadcasted to all `ORMQR` kernels of the same line, as well as two outputs of each `TPQRT` kernel to all `TPMQRT` kernels of the same line.

To push the dynamic broadcasts to their limits, we used on purpose large matrices (three times larger than their height), with a specific data distribution: all tiles of a column are on the same node (*i.e.* $P = 1$ and Q equals the number of nodes: a 1D-block-cyclic distribution). This way the number of recipients in broadcasts are always the number of nodes minus one and all broadcasts have the same number of recipients.

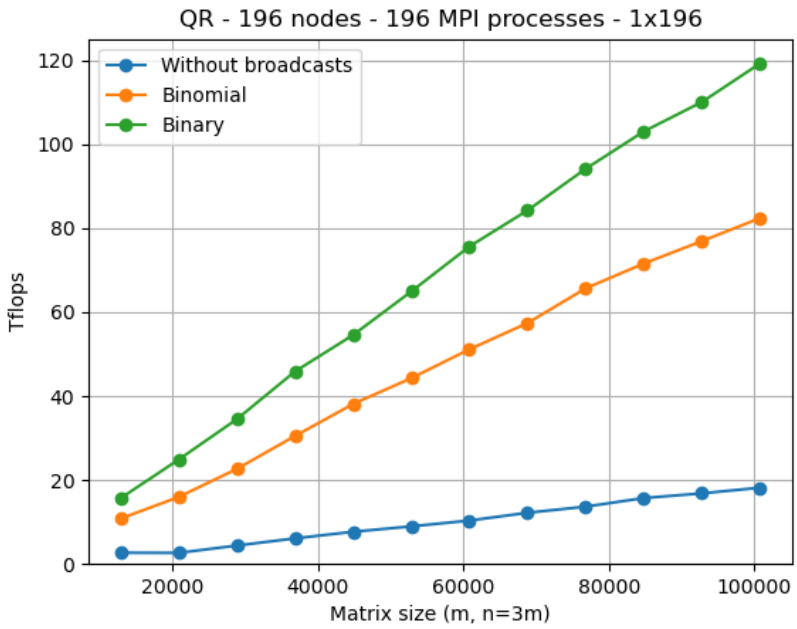


Figure 4.14: Performance of QR factorization, on occigen.

Results

The impact of dynamic broadcasts for the QR factorization on 196 nodes (with one MPI process per node) is plotted on Figure 4.14. For this application with such distribution, dynamic broadcasts considerably increase the performance: with binary trees it is 7 times higher than without broadcasts and 5 times with binomial trees.

The first comment on these results is that the baseline (without broadcasts) is very low (20 Tflops with 196 nodes is *low*, this performance should be reached with only some twenty nodes!), because of the suboptimal data distribution which requires lot of communications. Since communications are a bottleneck and broadcasts appear in the algorithm, improving the broadcasts execution increase performance, as confirmed by the curves for the two different trees. Even if the performance was not analyzed in detail, a hypothesis to explain the better performance of binary trees over binomial ones could be the following. With binary trees a node is involved in a broadcast for a short time (only to send to two other nodes): thus, network bandwidth is quickly available for other communications, from other broadcasts. The contrary applies to binomial trees. If this hypothesis is verified, it would mean for the QR algorithm that all recipients of a broadcast do not need to receive the data very fast, it is enough if a few of them receive it quickly, because binary trees are deeper than binomial ones.

Finally, performance is the same if tasks are unlocked as soon as possible or if the end of forwards to other recipients is awaited. Chameleon does not provide priorities for tasks of the QR algorithm, thus taking into account priorities in dynamic broadcasts is useless.

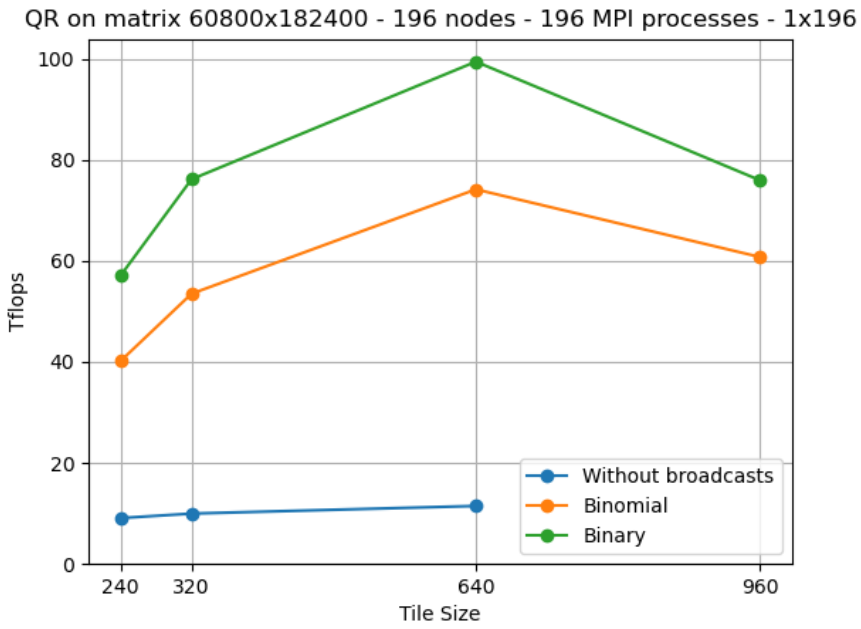


Figure 4.15: Impact of the tile size on QR factorization, on occigen. The run without broadcasts with a block size of 960 × 960 failed due to a deadlock.

The chosen size for the matrix tiles can influence several aspects of the program execution: the number of tasks is different, therefore there is more or less parallelism, tiles may or not fit in the different caches, causing potentially lot of cache evictions, *etc.* From a communication point-of-view, the number of messages transmitted through the network will be different and their size as well: 225 KB for 240 × 240 tiles, 409.6 KB for 320 × 320 tiles[1], or 3.6 MB for 960 × 960 tiles. As observed previously with the microbenchmarks (especially on Figure 4.3b), with bigger messages, it is even more attractive to use dynamic broadcasts, regarding the network bandwidth. Figure 4.15 shows the same curve order and differences compared to Figure 4.14, but according to the tile size. The optimal tile size seems to be around 640 × 640: below this value there is too many communications and above the communication latency is too costly.

In the two previous experiments, the data distribution was on purpose suboptimal, to create broadcasts with many recipients. Figure 4.16 evaluates the performance of other 2D-block-cyclic data distributions, on a small matrix (28 800 × 86 400). The least intuitive data distributions for QR factorization (1D-block-cyclic or close to: 1 × 196, 2 × 98) appear to be those with best performance with dynamic broadcasts. With the 1 × 196 data distribution, binary trees allow the application to reach 30.7 Tflops, while the version with disabled broadcasts does not even reach 5 Tflops. Performance with dynamic broadcasts is never worse than without broadcasts. In all cases, the performance is very low, with regards to the number of nodes. This comes probably from the rectangular shape of the matrix and its small size which does not allow to reach the maximum of possible parallelism.

[1] 320 × 320 is the default tile size in all other experiments.

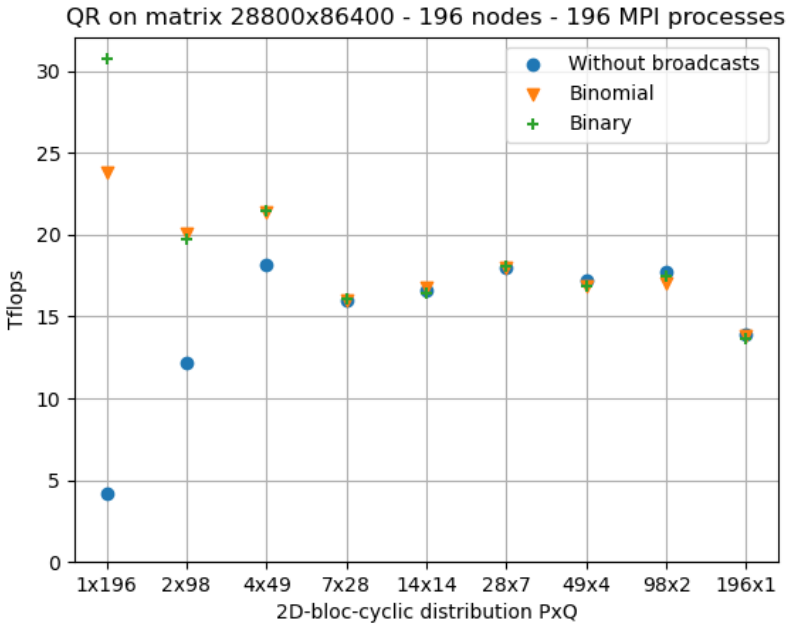


Figure 4.16: Impact of the 2D-block-cyclic distribution parameters on QR factorization, with tile size of 320×320, on occigen.

4.5 Discussion

4.5.1 Performance analyses

As mentioned when commenting the performance results, an in-depth analysis of the performance with dynamic broadcasts has not been performed. Indeed, it is not straightforward for several reasons: it requires a deep analysis of the runtime system and the communication library *together*, to understand how dynamic broadcasts influence the application execution.

Since we use trees with a logarithmic number of steps with the number of recipients, the performance gain provided by dynamic broadcasts will be visible only for broadcasts with many recipients: to add one level in a tree, we need to double the number of recipients; according to the data distribution, this can require to at least double the number of nodes. Executing applications at a large scale (several hundreds of nodes) can be very costly in terms of resources and time. The suboptimal data distribution used for the QR algorithm allowed to see how our implementation behaves with larger broadcasts (a binomial tree with 195 recipients has 8 steps, a binary tree has 7 steps).

Introducing a theoretical model to predict the performance gain thanks to dynamic broadcasts is not easy, because of the natural overlap of communication by computations by the runtime system and the asynchronicity of the programming model: there are no synchronizations, no well-identified phases where only communications occur and whose duration would be easy to measure and compute.

Despite these difficulties, precisely understanding the performance of dynamic broadcasts would allow to explain why dynamic broadcasts improve performance on some machines (*e.g.* CHOLESKY on `inti`, Figure 4.9) but not on others (*e.g.* CHOLESKY on `occigen`, Figure 4.10a). First hypothesis suggest it has to do with the ratio of computation / communication performance and/or number of tasks and network communications.

4.5.2 Generalization and extensions of the concept

Other types of collective operations

This work was focused on broadcasts. One may ask about similar techniques to improve other collective operations.

There was opportunity for performance improvement with the broadcast operation, since smart routing algorithms can be used to increase its performance. Main difficulties to set up an efficient broadcast in the STARPU context was the lack of synchronization in the programming model and the difficult detection.

For operations which cannot benefit in general from clever routing techniques, especially because each of their internal point-to-point communications transmits a different data, there is no trivial improvement to bring. This is the case for the *gather* (`MPI_Gather`, collect different portions of an array from different processes to a single process), or *scatter* (`MPI_Scatter`, distribute different portions of an array from a single process to several processes) operations. Moreover, using these operations within STARPU does not really make sense: each portion of the array should be represented by a different data handle and should be independently communicated, without waiting for all data portions to start the collective operation (a sort of synchronization!).

The reduction operation is more interesting because it may leverage optimized routing schemes and needs to execute computation at each step with received data. The reduction tree could be managed by the communication library and the computations could be tasks managed by STARPU. Actually, reductions can already be executed with STARPU, with specific flags attached to tasks to indicate the reduction. STARPU manages itself the reduction tree, submitting point-to-point communications following a tree scheme and corresponding tasks. In a word, the reduction is decomposed internally by STARPU in point-to-point communications and computation tasks, without any optimization directly from the communication library.

On top of problems mentioned for `MPI_Gather` and `MPI_Scatter`, the main difficulty also for other collective operations such as `MPI_AllGather` (all processes get the final whole array); `MPI_AllReduce` (all processes get the result of the reduction) `MPI_AllToAll` (a combination of `MPI_Scatter` and `MPI_Gather`), is the detection from the DAG. For a broadcast, the pattern is easy to recognize in the DAG: a node with several outgoing edges. Patterns corresponding to other collective communications are much harder to recognize in the tangle of dependencies between tasks. Even if the patterns are found in the task graph, it can be difficult to propose solutions using optimized collective operations, without synchronization, and which at the end improve performance.

Tree types

When a broadcast is initiated, the routing algorithm is executed to know to which nodes the direct recipients of the root node will need to forward the data. This list of nodes forming a subtree is included in the message header. When a node receives such a message, it executes the routing algorithm on the list of nodes included in the message header, to know to which nodes it has to forward. The process continues recursively while the leaf of the tree are not reached.

All kind of trees can be used with this technique: we showed binomial and binary trees, but it can be extended to chains, k-ary, k-nomial trees, *etc.*

We made experiments with what we called *bitrees*: the list of recipients of a broadcasts is split in two groups (one group can contains more items), the root node starts a first broadcast with the first group of recipients and when this broadcast is finished, the root node starts a second broadcast with the second group. The two broadcasts can follow different routing schemes. This pattern can be useful if a set of recipients needs the data very quickly but not the other set (*e.g.* communication priorities form two distant clusters): the first broadcast can use a binomial tree and the second a binary one. First results show that mixing this way two different routing trees gives performance similar to performance obtained with only one tree with all recipients, following the routing algorithm of the tree with the highest number of recipients in the bitree version.

Another option is to *delegate* the broadcast execution to the first recipient: the root node sends data to only the first recipient, and the latter has to execute the remaining of the broadcast, as if it was the root node, with the selected tree type. This can be useful if the original root node has to send many broadcasts with independent recipient sets very quickly: therefore the different broadcasts can start to be executed simultaneously on different nodes.

An improvement (not implemented yet) is to take into account the network topology in the recipient reordering, in addition to communication priorities. This could load-balance the communications and reduce the general occupancy of the network links. However, gains should only be expected with broadcasts delivering big messages to many recipients.

4.6 Conclusion

In DAG of task-based applications, a situation may appear where a given piece of data needs to be sent to multiple nodes. The use of an optimized broadcast algorithm is desirable for scalability. However, the constraints of relaxed synchronization and asynchronous schedulers on nodes make the use of `MPI_Bcast` inappropriate: the cost of required mechanisms to be able to use this function would cost more than the gain of performance it could provide.

We have introduced a *dynamic broadcast* mechanism which makes it possible to use an optimized tree-based broadcast algorithm without needing all the participating nodes to know all the other nodes, and without even needing them to know at all they are involved in a broadcast. The integration is seamless and nodes receive data with a regular

point-to-point receive function. We evaluated its performance on CHOLESKY and QR factorizations. Results show that our dynamic broadcast may improve overall performance and scalability.

Many applications are developed and optimized to reduce costly communications. Our system allows to be less concerned by communications which can be detected as broadcasts, since they will be automatically detected and optimized. This mechanism is also a good example of interaction between task-based runtime systems and communication libraries: the runtime system shares with the communication the recipient list of a broadcast, and the communication library is able to notify the runtime system as soon as data arrived from a broadcasts but forwards are not finished. Moreover, we extended the interface of the communication library to cope with STARPU's constraints.

Chapter 5

Interferences between Communications and Computations

StarPU naturally overlap communications with computations by executing ready tasks in parallel of communications. This means communications and computations are executed simultaneously. Using this technique with task-based runtime systems or just with non-blocking MPI calls is supposed to improve application performance, by masking the duration of communications. We have observed with execution traces that, in some applications, sometimes, when computations and communications are executed side by side, communications are slower than nominal performance and computations can also be degraded, which is consistent with the literature reviewed in Chapter 2.

Since possible interactions between communications and computations, and especially the impact on communication performance, are not well detailed in other studies (but only mentioned), we assess the possible causes of these interferences and measure their impact on performance of *both* communication and computing. We investigate the impact of processor frequencies, memory contention and the use of a task-based runtime system.

This chapter presents the following study. We measure the impact of frequency scaling on communications. We also study the impact, in the case of CPU-bound applications and memory-bound ones, on communication bandwidth and latency. Moreover, we study the effect of data locality and thread mapping on the interference between computation and communication. Further, we introduce a benchmark with tunable arithmetic intensity to observe how the application memory pressure penalizes the performance of communications. We also study the communication performance degradation caused by the use of StarPU. For all possible presented interferences, we measure their impact on both communication and computing performance. Finally, we realize the same study on two important HPC kernels: conjugate gradient (`CG`) and matrix multiplication (`GEMM`).

5.1 Methodology

Our goal is to measure performance of communications and computations when they are run side by side and evaluate the potential impact of interferences on performance of both communications and computations. To achieve this, we have designed a multithreaded and parallel benchmark using MPI+OpenMP[1]. One thread is dedicated to communications (it submits communication instructions and ensures MPI progression) and other threads do computations. This communication benchmark performs ping-pongs to measure network latency and bandwidth.

We need to compare performance of communications and computations when they are executed alone and when they are executed together. Therefore, we decomposed our benchmark into the following steps:

1. Computation without communication;
2. Communication without computation;
3. Computation with side by side communication.

Computations and communications use different data and hence are completely independent. The majority of plots in this chapter compares performance of communications and computations when they are executed separately or simultaneously (see Figure 5.3 for instance). The former are represented by plain curves and the later by dashed curves. Curves represent median value of the results obtained with several runs and background areas are delimited by the first and the last decile of these results.

Communications and computations are done in dedicated threads, all belonging to the same program. Each thread (computing ones and communicating one) is bound to a different core to stabilize performance and ensure reproducibility. In the remaining of this chapter, we will call computation cores the cores that execute the computation threads and communication core the core that executes the communication thread.

We use the same metrics as in the previous chapter: *latency* represents the duration of a communication (time elapsed between the beginning of `MPI_Send` and the end of `MPI_Recv`, *i.e.* half ping-pong) and *bandwidth* is the obtained network throughput by dividing the size of the transmitted data by this *latency*. When we do not mention data size, *latency* is measured on 4 bytes of data (one `float`), and *bandwidth* is the asymptotic value, evaluated for 64 MB message size. Buffers used for ping-pongs are recycled to mimic standard applications that update internal data step by step and to take benefit of *registration cache* mechanism [113]. Since our first observations about possible interferences between computations and communications were made with StarPU-MPI applications, our benchmark mimics on purpose its working. We chose to measure communication performance with ping-pongs for their simplicity, they require only few parameters to be analyzed and StarPU uses mainly point-to-point communications; analyzing also collective communications would be beyond the scope of this work. In the same way, when many threads make MPI communications, it brings many other considerations we do not want to explore in this study.

[1] Available on https://gitlab.inria.fr/pswartva/memory-contention

We ran our own benchmark suite on several clusters with different characteristics: from small experimental clusters to large production ones. Since results are generally similar on all tested clusters, we present only results obtained on `henri` nodes (described in Appendix C) and mention noteworthy differences on other clusters.

We show results obtained with MadMPI, the MPI interface of NewMadeleine; we observed similar results with other MPI implementations, such as OpenMPI 4.0.

5.2 Impact of frequencies

In this section, we study the impact of frequencies on communication performance. To avoid overheating and minimize energy consumption, processors change their frequencies depending on the processor load. This dynamic frequency scaling feature of modern processors has a direct impact on computing performance. Since computation may cause changes in processor frequencies, we assess, in this section, whether these frequency variations also have an impact on communications.

We consider two kinds of frequencies: core and uncore. The core frequency impacts computation units and L1 and L2 caches[2]. The uncore frequency [63] concerns last level cache and the memory controller[3]. We measure the impact of these two frequencies independently by setting them to a constant value for all cores and sockets. We evaluate network performance for the two extremes of the permitted ranges of frequencies: 1000-2300 MHz for core frequency and 1200-2400 MHz for uncore frequency.

5.2.1 Impact of frequencies on only communications

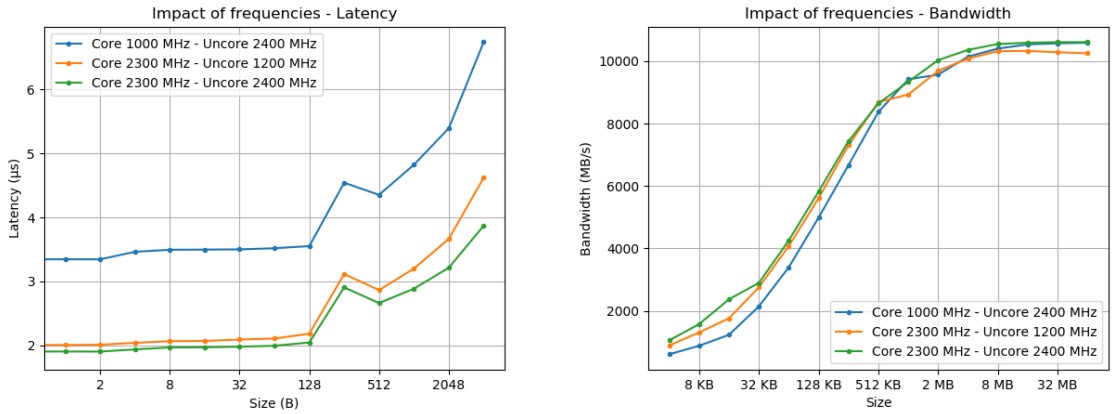
We performed ping-pong benchmarks to measure network latency and bandwidth in function of core and uncore frequencies. Since we study only the impact of frequencies, the ping-pong benchmark relies only on an MPI library and no other runtime. No computation is done at the same time.

Concerning the core frequency, as seen on Figure 5.1a, the network latency is lower when the frequency is higher: 1.8 μs with 2300 MHz *vs* 3.1 μs with 1000 MHz. We explain this as follows. The network latency is comprised of hardware latency (time to move the data over the wire) and software overhead (time for software to process the communication operation, the o of the *LogP* model [40]). Hence, with a lower core frequency, the software overhead is higher. Variations of the CPU frequency do not affect the network bandwidth (Figure 5.1b), except for the fixed overhead of latency that impacts slightly the bandwidth for small messages. It is explained by the fact that large messages are transferred through DMA, without going through the CPU, thus their speed is unaffected by CPU frequency.

Conversely, the uncore frequency has no impact on the latency (the difference when changing only the uncore is negligible regarding the difference when changing only the

[2]We use the `userspace` governor and the `cpupower` tool to set the constant core frequency we wish, otherwise the `performance` governor is used.

[3]We use Likwid [117] to get and set the uncore frequency.



(a) Impact on latency

(b) Impact on bandwidth

Figure 5.1: Impact of constant frequencies on network performance, on `henri` nodes.

core frequency: +5 % *vs* +72 %) but has a small impact on the bandwidth (10.5 GB/s with 2400 MHz *vs* 10.1 GB/s with 1200 MHz).

5.2.2 Impact of frequency variations caused by computations

We now observe network performance when one core executes the communications (using the ping-pong benchmark) and other cores are executing CPU-bound computations: a computing benchmark counts in a very naive way the number of prime numbers in an interval. This forces the CPU to execute instructions which do not require any memory access (the algorithm uses only few integer variables).

In Figure 5.2, we plot the frequencies of the different cores when (A) only communication is performed, (B) all cores are idle and (C) communications are performed while 20 cores are executing the compute-bound benchmark. We see that all cores have a higher frequency when computations and communications are done at the same time than when communications are executed alone. We have also measured the bandwidth and the latency when communications and computations are done side by side: the network bandwidth is very slightly improved when computation is done at the same time (9097 MB/s *vs* 9063 MB/s), as well as the network latency (1.52 μs *vs* 1.7 μs). As the CPU frequency of communication core is the same in case (A) and (C) we conclude that the communication latency is not impacted only by the frequency of the core doing these communications: when other cores increase their frequency, it improves the communication latency.

However, these results seem to be hardware-dependent: on `bora`, the network bandwidth has a wide deviation[4] and computations are highly impacted by the communications when there are more than 15 computing cores. In Figure 5.3, we see the computing duration jumps from 183 ms to 236 ms: the computation is slowed down when it starts using the socket performing communication. Network latency is constant and duration of computations done along the latency benchmark is also constant regardless of the number of

[4]We observed this behaviour on other clusters equipped with Intel Omni-Path networks.

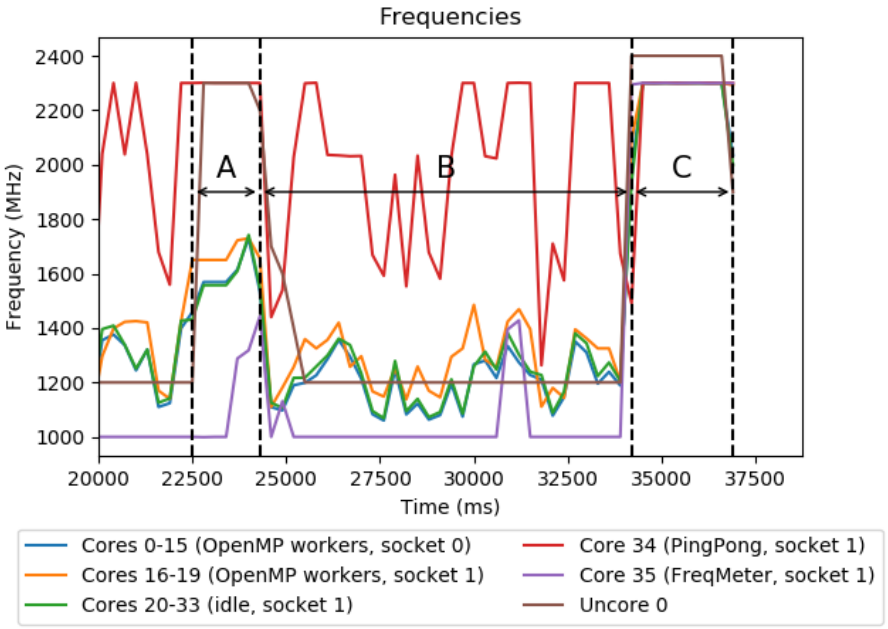


Figure 5.2: Frequency variations during (A) only communications, (B) sleep and (C) communications and computations, with 20 computing cores, on `henri` nodes.

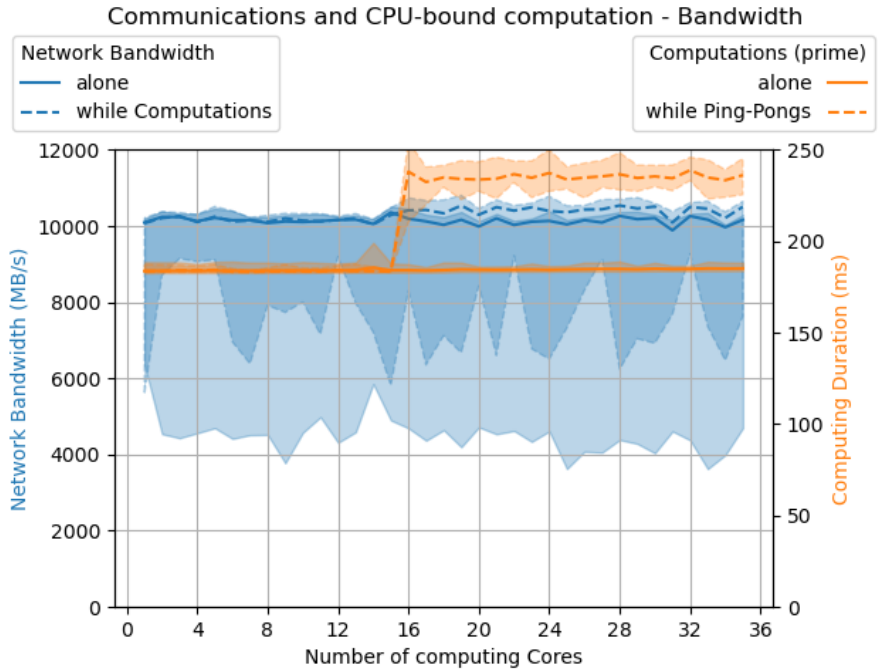


Figure 5.3: CPU-bound computations and network bandwidth performance, on `bora` nodes.

computing cores, exactly like on henri nodes (not displayed on Figure 5.3).

5.2.3 Impact of AVX instructions on frequencies

Computing cores can use wide vector instructions such as those from the AVX family [81]. Although these instructions allow reaching better computing performance, they force the cores executing them to reduce their frequency because these instructions consume more power [56]. The core frequency is further reduced when there are more cores executing AVX instructions at the same time. On the other hand, with *turbo-boost*, if only few cores execute AVX instructions, these cores can increase their frequency.

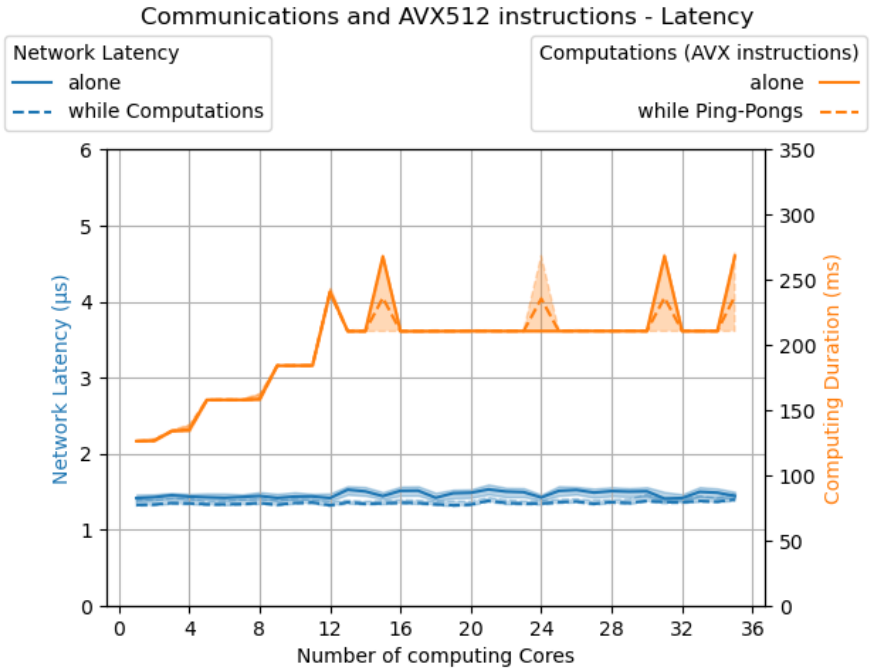
We study here if computing cores doing AVX instructions can have an impact on the frequency of the core executing the communication thread and thus change the communication latency. In our experiment, each computing core does the same amount of computation: a set of multiple AVX512 floating instructions (weak scalability). Since the range of frequency variation is higher when turbo-boost is enabled, we show only this case here (results are similar when turbo-boost is disabled).

As expected, computations are faster with only few computing cores (Figure 5.4a). With only 4 computing cores (Figure 5.4b), computing cores work at 3 GHz and computations last 135 ms and with 20 computing cores (Figure 5.4c), their frequency is 2.3 GHz and computations last 210 ms (lowered core frequency increases computing duration). In both cases, the frequency of the communication core is stable at 2.5 GHz and is not negatively impacted by cores executing AVX instructions: no matter the number of computing cores, the network latency is always slightly better when computations are done at the same time (1.33 μs *vs* 1.49 μs) of computations. This is consistent with what we have observed with previous experiments (in section 5.2.2) on the same machine: the uncore frequency (constant regardless the number of computing cores) has no effect on network latency, while a higher core frequency can improves latency. On bora nodes, computation and communication performance with AVX instructions are the same as those observed without AVX instructions.

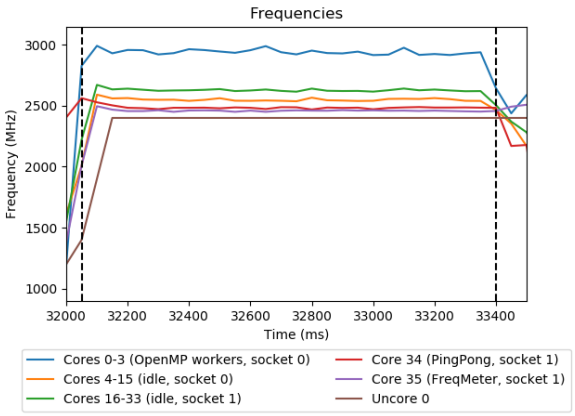
To summarize, cores executing AVX instructions do not impact the frequency of the core executing the communications and thus communication performance is not affected.

5.2.4 Conclusion on the impact of frequency variations

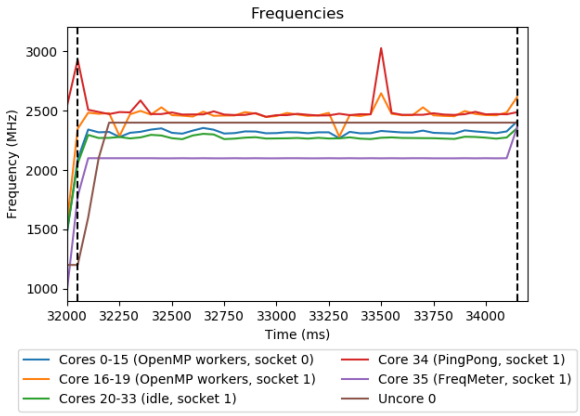
In conclusion, we have observed that the core CPU frequency impacts communication latency: the higher the frequency, the lower the latency. To a lesser extent, uncore frequency slightly impacts the communication bandwidth. Computations can change the frequency of the cores executing them, but do not change the frequency of the core executing communications, even with AVX instructions, and hence do not impact the communication performance. On the contrary, latency is slightly better when CPU-bound computations are made at the same time.



(a) Network latency



(b) Frequency variation with 4 computing cores and communications



(c) Frequency variation with 20 computing cores and communications

Figure 5.4: Impact of AVX512 computations on network latency, on `henri` nodes with turbo-boost.

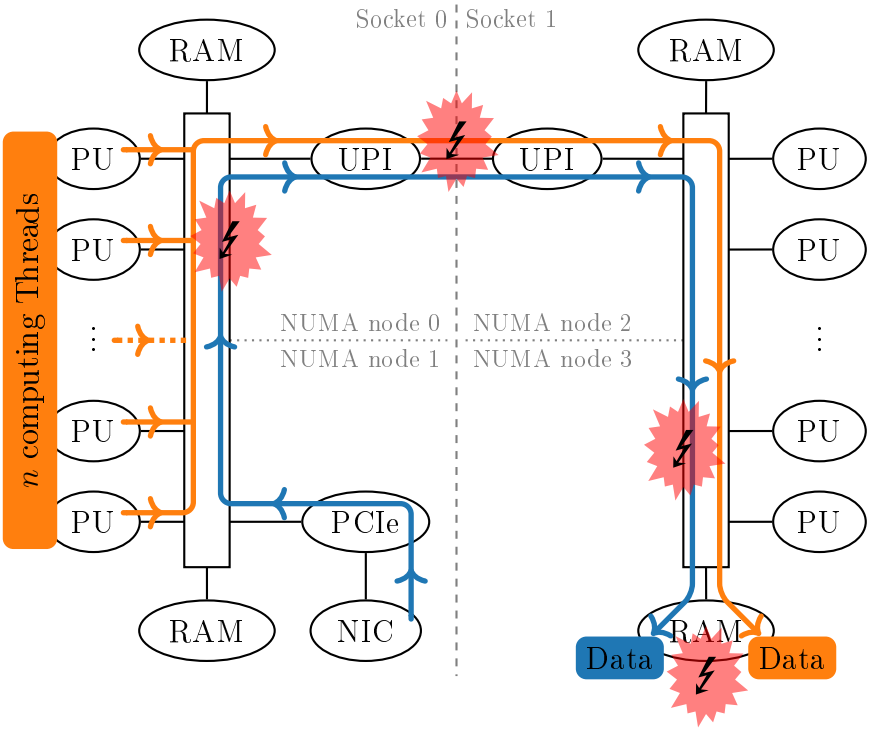


Figure 5.5: Diagram of different data streams in an HPC node: contention between computations and communications can occur at different locations. The *inter-socket bus* is called *Infinity Fabric* (IF) on AMD processors, and *Ultra Path Interconnect* (UPI) on INTEL processors.

5.3 Memory contention

Data moving from memory to the CPU and data moving from memory to the NIC are actually using the same memory bus. Therefore, in this section, we study the interaction between memory accesses used for computations and communications over the network, to check whether there may be contention between the data streams for computations and for communications, as illustrated by Figure 5.5.

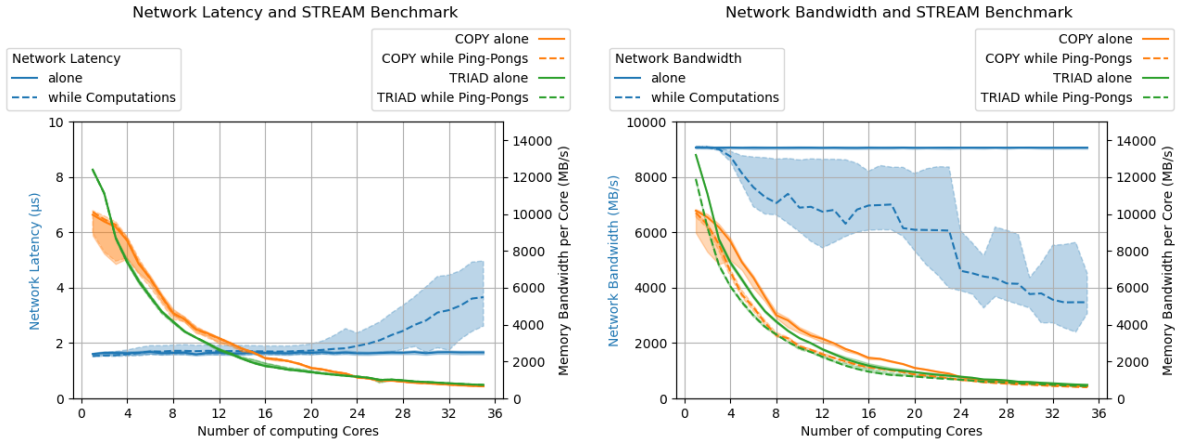
5.3.1 Benchmarking memory contention

To see what happens when memory contention occurs, we produce memory contention with the STREAM benchmark suite [87], especially the following two STREAM kernels that perform simple arithmetic on large arrays:

`COPY` copy each element of an array to another one: $b[i] \leftarrow a[i]$

`TRIAD` multiply each element of an array by a constant, add it to the element of another array and store the result in another one: $c[i] \leftarrow a[i] + C \times b[i]$

These kernels are memory-bound, causing high pressure on the memory bus. Moreover, to really produce memory contention, we allocate memory on a single NUMA node, to



(a) Network latency

(b) Network bandwidth

Figure 5.6: Memory-bound computations and network performance, on `henri` nodes.

increase the traffic on the memory bus between cores belonging to different NUMA nodes. The loop iterating over these arrays is parallelized on available computing cores with OpenMP. The performance of the computing benchmark is measured using the memory bandwidth per core (hence higher is better). For communications, we execute a ping-pong benchmark (see section 5.1) in its own thread. Such communication benchmark is run alongside STREAM in a separate thread to measure the impact of interferences.

5.3.2 Impact of memory contention

In the execution reported in Figure 5.6, memory is allocated on the NUMA node where the NIC is also connected, communication thread is bound to the last core of the other NUMA node, and computing threads are bound to cores respecting the order of the logical core numbering. Figure 5.6a shows that network latency is impacted by the STREAM operations when there are at least 22 computing cores and this impact can double the regular latency when all available cores are computing. However, STREAM operations are not impacted by the ping-pong benchmark. The network bandwidth is impacted sooner, from 3 computing cores (Figure 5.6b). When all available cores are computing, the network bandwidth is reduced by almost two third from the regular network bandwidth. Memory bandwidth measured by the STREAM benchmark is lower when network bandwidth is measured at the same time as when network latency is measured, which is expected because one bandwidth ping-pong transfers more data than a latency ping-pong (64 MB *vs* 4 B). On `bora` nodes (Figure 5.7), the network bandwidth is impacted, but later: from 20 computing cores; latency gives similar results. Results on `billy` and `pyxis` nodes are similar to those observed on `henri` nodes.

5.3.3 Impact of thread and data placement

Here, we study the impact on the performance of the data locality and the communication thread mapping. To do so, we bind the communication thread to a core either on the same

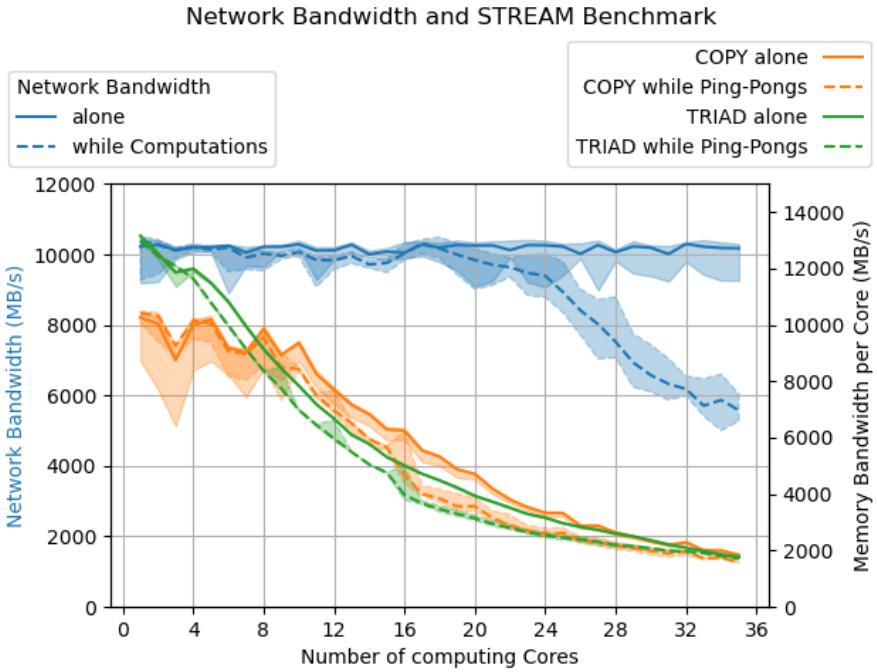


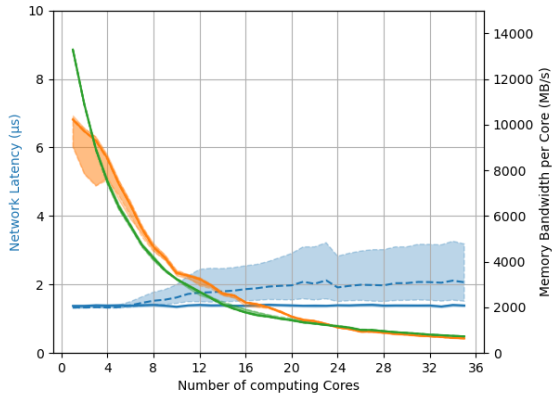
Figure 5.7: Memory-bound computations and network bandwidth performance, on `bora` nodes.

NUMA node where the NIC is plugged (*near the NIC*) or on the other NUMA node (*far from the NIC*). Similarly, we explicitly allocate the memory (used for computations and communication) either on one or the other NUMA node. It is known [88] that placement may have an impact on communication performance; we check whether contention worsens this phenomenon.

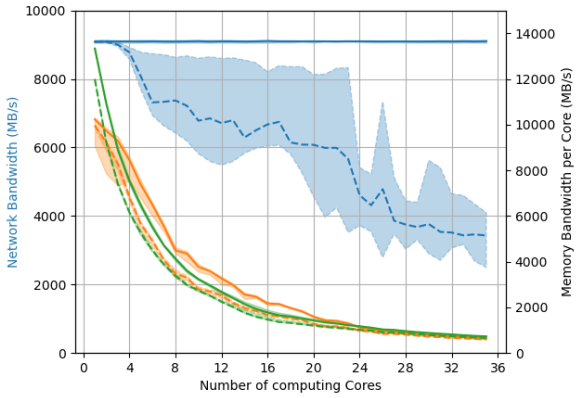
On `henri` nodes, Figure 5.6 shows results for data near the NIC and communication thread far from the NIC and Figure 5.8 shows results for other placement schemes. Results on other clusters were similar. When the communication thread is bound near the NIC, the latency increases as soon as we use more than 6 computing cores, but then stays around 2 μs (even if the error range is higher). When the communication thread is far from the NIC, the latency increases considerably from 25 computing cores by doubling its nominal value and reaching more than 4 μs. Without computations at the same time, latency is slightly better when communication thread is bound near the NIC (1.39 μs *vs* 1.67 μs). As expected, because small messages are sent from the CPU to the NIC (using programmed I/O), thus if the communication thread is closer to the NIC, the latency is better.

Bandwidth curves have generally the same shape wherever the communication thread is bound. When data is located near the NIC, bandwidth decreases steadily when the number of cores increases. When data is located far from the NIC, bandwidth drops much more abruptly. Since large messages are sent through Direct Memory Access (DMA), the crucial factor is data placement; when data is closer to the NIC, we observe better overall bandwidth and less impact of memory contention than when data is far from the NIC.

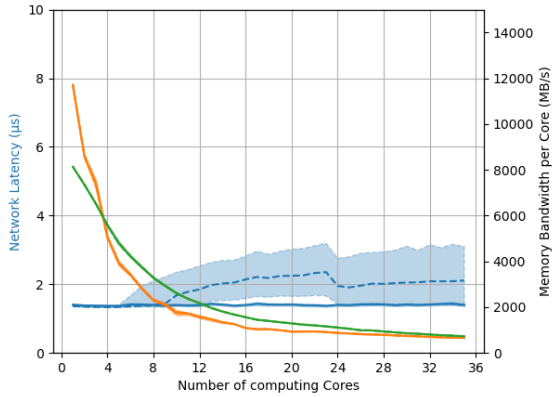
In all configurations, latency benchmarks do not impact the STREAM benchmark per-



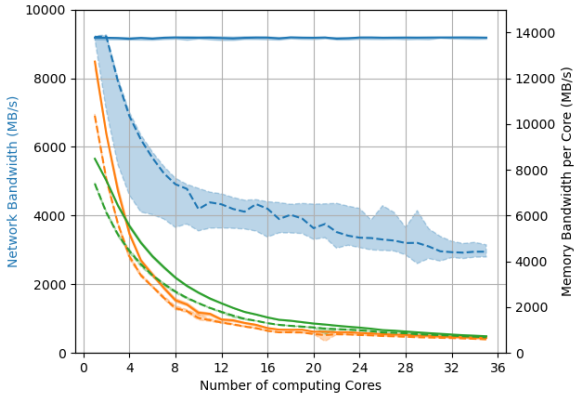
(a) Network latency, data: near the NIC, MPI thread: near the NIC



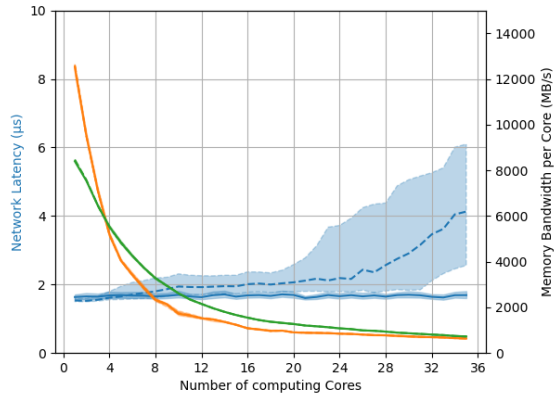
(b) Network bandwidth, data: near the NIC, MPI thread: near the NIC



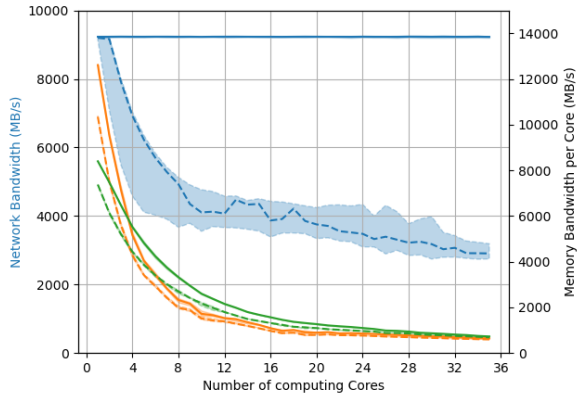
(c) Network latency, data: far from the NIC, MPI thread: near the NIC



(d) Network bandwidth, data: far from the NIC, MPI thread: near the NIC

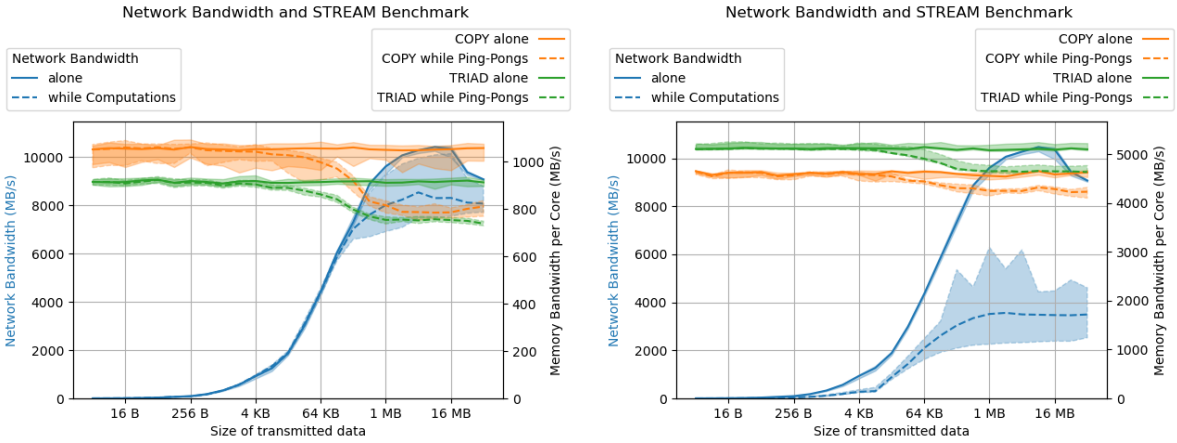


(e) Network latency, data: far from the NIC, MPI thread: far from the NIC



(f) Network bandwidth, data: far from the NIC, MPI thread: far from the NIC

Figure 5.8: Impact of communication thread placement and data locality, on `henri` nodes. Legend is the same as in Figure 5.7.



(a) 5 computing cores

(b) 35 computing cores

Figure 5.9: Impact of size of communicated data, on `henri` nodes.

formance, but bandwidth benchmarks do: STREAM loses up to 25 % (with 5 computing cores) of its performance when executed side by side of communications.

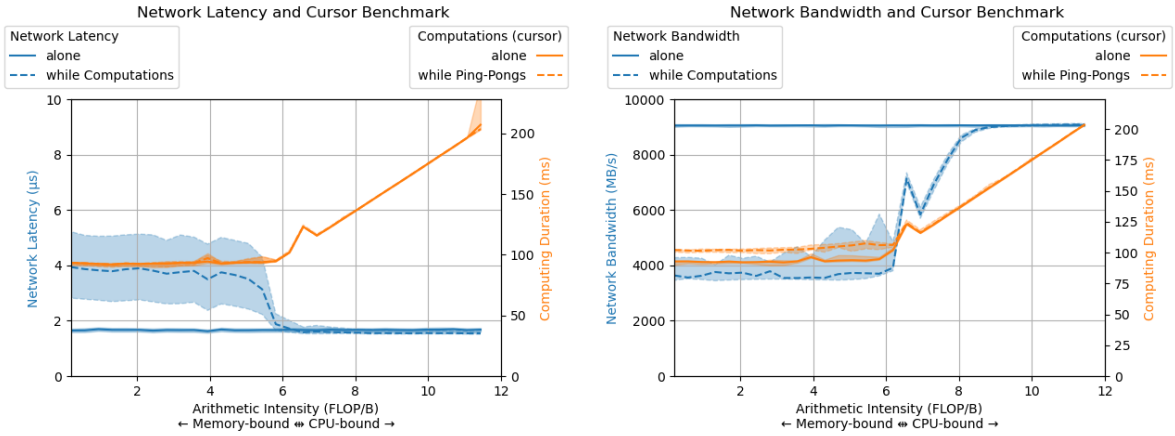
To sum up, placement of communicating thread and locality of memory has an impact on network performance and on the memory contention. When the communication thread is far from the NIC, latency suffers more from contention on the memory bus. When data is far from the NIC, network bandwidth suffers more from contention. Moreover, in all configurations, transmitting large messages on the network increases the impact of contention on the computations.

5.3.4 Impact of transmitted data size on memory contention

In this section, we study the impact of message size on the contention on the memory bus. We have observed in the previous section that bandwidth benchmarks are more impacted by contention than latency benchmarks. Moreover, communication libraries can exhibit different behaviours according to the size of data to transmit (*e.g.* switch from *eager* to *rendez-vous* protocol). We measure STREAM and network performance with varying message size transmitted through the network, so as to know which message sizes cause a performance drop. We performed this benchmark with two different numbers of computation cores: 5 cores, which is the point where STREAM is the most impacted by communications; and 35 cores, where the network bandwidth is the most impacted by STREAM, as we saw on Figure 5.6b.

With 5 computing cores (Figure 5.9a), communications begin to be degraded with a message size of 64 KB, but STREAM begins to be impacted sooner, from 4 KB transferred over the network. With 35 computing cores (Figure 5.9b), communications begin to be degraded sooner than with 5 computing cores: from 128 B and STREAM is impacted from 4 KB as well. Results were similar on other clusters, when there is a small computing cores or when all available cores are computing.

On the whole, a large number of computation cores causes a high memory pressure



(a) Network latency

(b) Network bandwidth

Figure 5.10: Impact of memory pressure on network performance, on `henri` nodes.

which impacts communication for a wide range of message sizes on the network. Conversely, large messages exchanged through the network cause enough traffic on the memory bus to impact computation even with only 5 cores.

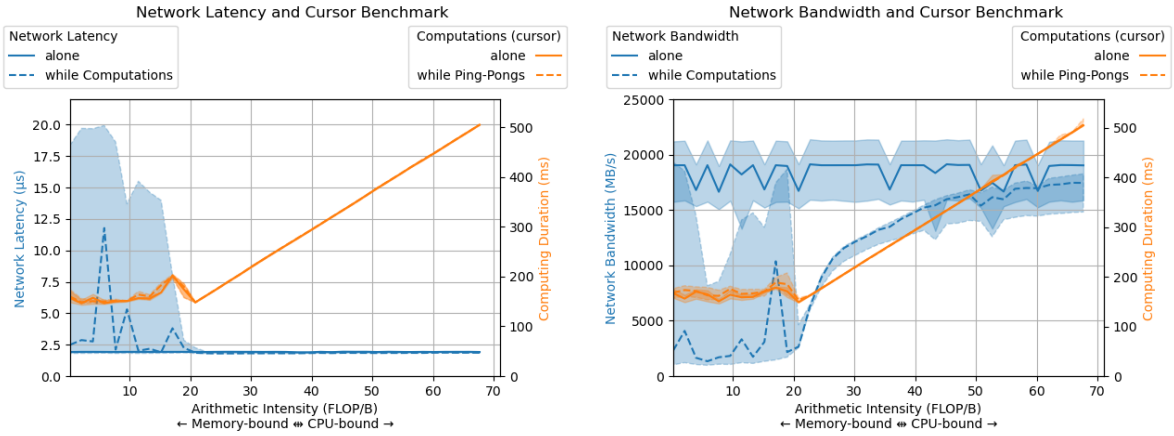
5.3.5 From CPU- to memory-bound applications

Real-world applications are usually not fully CPU-bound or completely memory-bound, but somewhere between these two extremes. Previous sections showed that CPU-bound applications have almost no impact on communications, but memory-bound ones do. Therefore, we modified the `TRIAD` algorithm in the STREAM benchmark to be able to tune the memory pressure, so as to see how the variation of this pressure degrades the network performance.

The memory pressure caused by computation is expressed as the *arithmetic intensity*, as used by the *roofline model* [124]: it is defined as the number of flops per byte of moved data.

In practice, to make the arithmetic intensity of our benchmark tunable, we added a simple loop in STREAM, repeating the operation on each item of the array, before moving to the next item. By changing the number of repetition on each item, the arithmetic intensity varies: with few repetitions the program moves rapidly from one item of the array to the next one (memory-bound) and with many repetitions, it spends more time before accessing the next item (CPU-bound). We call this number of iterations per item in the array a *cursor*: changing the cursor value thus makes the benchmark progressively moving from being memory-bound to CPU-bound.

Results of this benchmark on `henri` nodes are depicted in Figure 5.10. We observe that high levels of memory pressure cause a huge performance drop for the network. When the arithmetic intensity is lower than 6 flop/B, the memory pressure has an impact: in the latency benchmark (Figure 5.10a), the network latency doubles and the computing duration is constant, which is the confirmation that it is actually memory-bound, and



(a) Network latency

(b) Network bandwidth

Figure 5.11: Impact of memory pressure on network performance, on `billy` nodes.

unaffected by the small messages on the network. In the same interval in the bandwidth benchmark (Figure 5.10b), the network bandwidth drops by 60 % and the computation is slowed down by 10 % because of interference with network operations which use large packets in this benchmark.

For arithmetic intensity higher than 6 flop/B, the program becomes CPU-bound, the memory pressure decreases: communication performance gets back to its nominal value and computation time is unaffected by network interference. On `billy` (Figure 5.11), the boundary between memory- and compute-bound programs is at 20 flop/B: it is also visible on both computation and communication performance, but the network bandwidth becomes not impacted by computations only when the arithmetic intensity is higher than 70 flop/B.

5.3.6 Conclusion on memory contention

Contention on the memory bus, caused by data movement between main memory and cores or NIC, can have a strong impact on performance. We have shown that the impact depends on several factors: (1) the placement of the communication thread and the data locality, since contention amplifies the impact of NUMA effects on the network; (2) the size of the messages transferred over the network, since larger messages cause a higher impact on computation performance; and (3) the arithmetic intensity of the code executed by computing cores, since code with low arithmetic intensity (thus high memory pressure) has a higher impact on network performance.

5.4 Runtime system impacts on communications

In this section, we study the impact of StarPU on communication performance, by executing a ping-pong benchmark, written with the StarPU API instead of plain MPI.

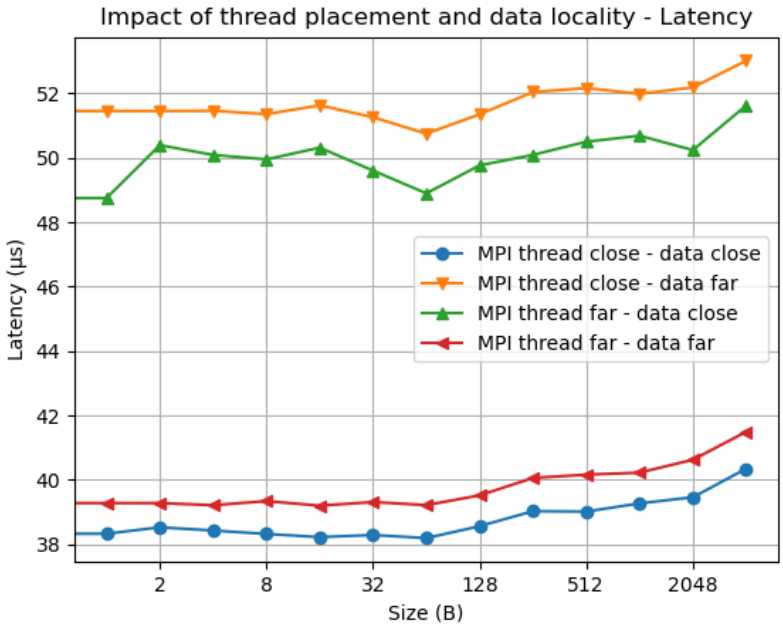


Figure 5.12: Impact of data locality and thread placement on network latency with StarPU, on `henri` nodes. *close* means on the same NUMA node as the NIC and *far* means on the other NUMA node.

5.4.1 Runtime system overhead

Using the StarPU API for the communications adds extra software layers on the path of messages that have to go through message lists, be processed by a worker and then by the communication thread. These mechanisms add an overhead to communication performance: in StarPU, the latency is increased by 38 µs on `henri` nodes, by 23 µs on `billy` nodes and by 45 µs on `pyxis` nodes. This latency difference is also noticeable on network bandwidth benchmark for messages smaller than 64 MB.

5.4.2 MPI thread and data placement

Within StarPU, a thread is dedicated to communications and makes them progress. This thread is usually bound to a dedicated core (similarly to what we did in section 5.3.3). The issue of memory locality and communication thread placement is still present when memory is directly allocated by workers, namely by different cores. If there are workers on all available cores and memory allocation uses the *first-touch* strategy, memory will be allocated on different NUMA nodes. Therefore, the performance of the transfer for those messages should depend on where the memory was allocated regarding the NIC, as observed previously.

Figure 5.12 shows the network latency overhead explained previously, but mainly that the most important for the network latency is that data to transfer and the communication thread are on the same NUMA node. That is expected, because for small messages, if

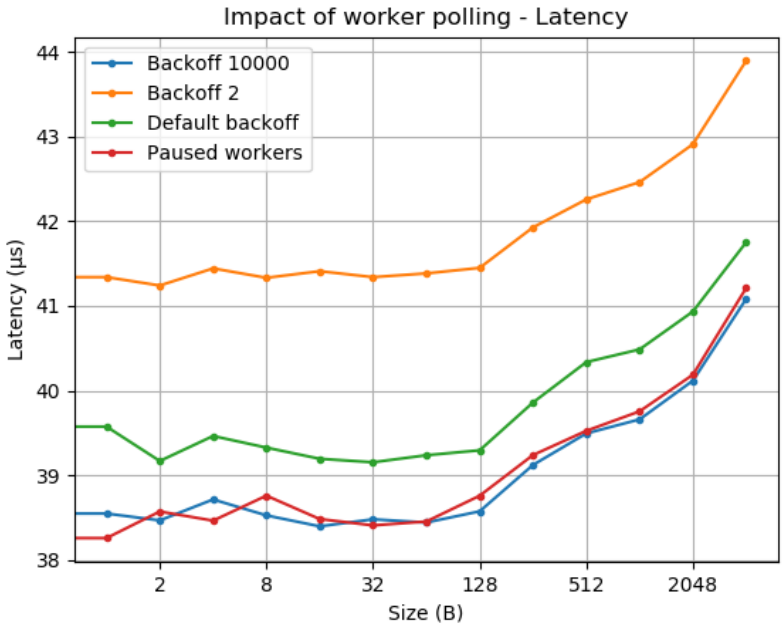


Figure 5.13: Impact of polling workers on network latency, on `henri` nodes.

the communication thread needs a remote NUMA access to get the data to send, it adds some delay to the latency. StarPU does not impact more the network bandwidth than previously observed.

5.4.3 Worker polling

StarPU's scheduler stores tasks submitted to the runtime system in a list. When a worker finishes a task execution, it consults this list to get the next task to execute. To be reactive enough, workers wait actively for tasks, this mechanism is called *busy waiting* or *polling*: if the list is empty, the worker waits a moment by executing a number of `nop` instructions, and then tries again to get a task. The number of `nop` is defined by an *exponential backoff* algorithm: it is doubled after each unsuccessful poll until a maximum is reached. The number is reset to its minimum when the worker finally gets a task. The maximum number of `nop` instructions can be defined by the user. With a small number, workers will be very reactive when a new task is pushed to the list (the task will start sooner). However, it produces traffic on the memory bus, because this list of tasks is shared among all workers. Worker polling can be interrupted by pausing workers.

To study the impact of polling done by workers on communications, we implement a benchmark with a ping-pong on network running without any task to execute, hence workers are constantly polling for new tasks. We executed the benchmark with default configuration (the default maximum number of `nop` instructions is 32), with a huge backoff (10000: workers poll rarely), with a small backoff (2: workers poll very frequently) and with paused workers (they are not polling at all). Figure 5.13 shows that polling workers have an impact on communication latency: the latency is higher when workers poll more

often. A long period between two polls is equivalent to paused workers and does not impact the latency. We explain this result by the increased traffic induced by workers accessing the list of tasks and locking mechanisms. However, polling workers have no impact on communication performance on `billy` and `pyxis` nodes.

5.4.4 Conclusion on runtime system impact

Our experiments show that task-based runtime system (here StarPU) can negatively impact the communication performance (especially latency) because of the longer software stack messages have to go through and due to aggressive worker polling.

5.5 Use-cases: computational kernels

To measure interferences between communications and computations in real computation codes, and especially the impact of computations on communications, we executed a dense conjugate gradient (`CG`) and a dense general matrix-matrix multiplication (`GEMM`), both built with StarPU, using the Intel MKL BLAS library and distributed on two `henri` nodes (2 MPI processes are enough to see the interferences and simplify the analysis). Using the profiling utility provided by NewMadeleine, we measured the time spent to send data over the network. This gives a sending network bandwidth: the network bandwidth as perceived by the sending node, not taking into account the time to receive data on the receiving node. We also used `pmu-tools`[5], a tool built on the top of the Linux `perf` program to read CPU performance counters, to evaluate the memory pressure caused by the computations. Regardless of the number of computing cores, the execution parameters are the same: matrix sizes and/or number of iterations (strong scaling), hence the amount of network communications is also the same.

Figure 5.14 depicts measured values according to the number of computing cores. All curves are the average of values obtained on the two MPI processes. The top plot represents the normalized bandwidth for network sends and the bottom one plots the proportion of execution time when the CPU was stalled on accesses to memory. This figure shows that, the more there are computing cores, the more cores are spending time to access the memory and hence affects negatively the sending bandwidth, as observed previously with our microbenchmarks. `CG` is more affected by this effect than `GEMM`. This is because `CG` is more memory bound than `GEMM`: with the maximum number of workers, 70 % of stalls are caused by memory accesses, while it is only 20 % with `GEMM`. As seen previously, these different memory pressures affect differently the network send bandwidth: with `GEMM`, communications lose at most 20 % of performance, while with `CG`, the loss is up to 90 %.

To sum up, common computational kernels, such as `CG` and `GEMM`, can have a significant impact on communications executed at the same time as computations. This impact depends on the arithmetic intensity of the executed kernels.

[5] https://github.com/andikleen/pmu-tools

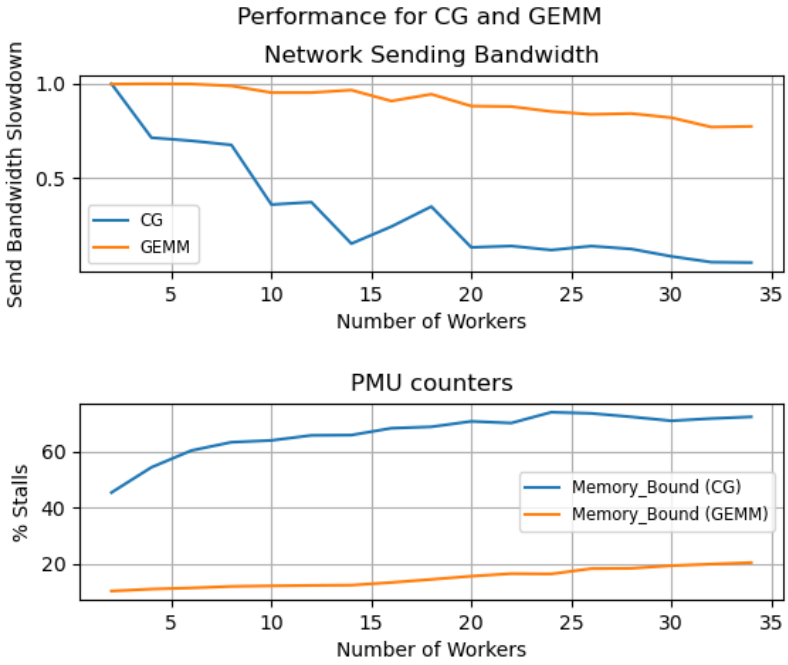


Figure 5.14: Network performance and hardware counter values of `CG` and `GEMM` executions, on `henri` nodes.

5.6 Conclusion

Doing parallel computation side by side with communications is one of the key features of task-based runtime systems and MPI libraries to achieve high-performance. However, such feature can have side effects that actually degrade performance. We studied in this chapter the possible effects and evaluated their impact on performance of both computations and communications. Frequency variations caused by computing cores have little impact on communications. However, memory contention caused by memory-bound computing programs and network transfers of big chunks of data has a strong impact on both computation and communication performance. This impact depends on the placement, the arithmetic intensity of the program executed by computing cores and the amount of data transferred across the network. Moreover, using StarPU can also penalize communications, just with the runtime system overhead, but also with internal mechanisms like polling on task queues. Communication thread placement, data locality and node topology (to which NUMA node the NIC is the closest) also impact performance. We observed the penalty on communications also in the execution of common HPC kernels such as conjugate gradient and matrix multiplication programs. These preliminary detailed results are necessary to be aware of these behaviours, and before being able to present solutions.

These phenomena are examples of negative interactions between runtime systems and communication libraries. The interferences do not appear only with task-based runtime systems, but because of their natural communications/computations overlap mechanisms, these kinds of runtime systems can also be victims of these interferences. Moreover, with all the features and abstractions provided by task-based runtime systems, it should be

easier to avoid these negative interactions. Taking the interferences between computations and communications would then become a positive interaction between the two software layers. To do so, we present in the next chapter a model to predict bandwidth sharing between computations and communications.

Chapter 6

Modeling Memory Contention between Communications and Computations

We observed in the previous chapter that memory contention between computations and communications executed in parallel (illustrated by Figure 5.5 on page 106) is the most important interference impacting performance of both computations and communications, because when overlapping communications and computation, data movement for the computation and for the network may share parts of the path in the machine memory system. Several factors can influence the contention: data placement, message size and arithmetic intensity of computing kernels. Performance is the most reduced when computing kernels are memory-intensive (putting important pressure on memory buses), big messages are exchanged (thus moving large amount of data through memory buses) and data to send to the network is located on a NUMA node different than the one where the network interface is plugged to.

In this chapter, we propose a model of this contention between computations and communications. Given a number of computing cores, the model can predict memory bandwidth available for computations and communications, when they are executed simultaneously, while taking into account the locality of data they manipulate. More than just predicting performance, the proposed model allows us to test our hypotheses about the internal working of processors' memory system, how they deal with contention between different kinds of streams.

6.1 Context and hypotheses

Since different kinds of data streams share the same memory bus, it is possible to sum the measured bandwidths of each data stream, to get the overall occupancy of the bus capacity, from a bandwidth point of view, like Majo and Gross did [86]. Indeed, this assumption is the cornerstone of our model; once the bandwidth capacity of the bus is known, one has to distribute the available bandwidth between computations and communications.

However, it is important to note that behaviours of processors and memory controllers

regarding contention are not publicly documented by processor manufacturers. Moreover, the values they use to characterize hardware features (the memory controller bandwidth or the SMP interconnect rate, for instance) can hardly be linked to experimental observations, nor directly used as parameters of the model.

Thus, we propose a model whose parameters are determined through experiments rather than theoretical capacity of hardware. We make our own set of hypotheses explaining memory bandwidth in case of contention, as well as our own set of benchmarks to get model parameters.

6.1.1 Contention behaviour

Memory buses have a finite bandwidth. When this capacity (or threshold) is reached, the bus capacity is shared between all components accessing it, reducing memory bandwidth available for each accessor. Memory requests issued by CPU cores may have a different (often higher) priority than requests coming from Peripheral Component Interconnect Express (PCIe) devices, *e.g.* from a network interface. However, a minimal memory bandwidth will always be available for these devices, to prevent starvations. We can also assume in some cases computing cores can generate contention with each other, even without communications in parallel.

If we put together these hypotheses: when communications and computations executed in parallel reach together the memory bus bandwidth threshold, communication bandwidth starts to decrease to avoid impacting computing cores. When the assured minimum bandwidth for communications is reached, the performance of computations decreases uniformly between computing cores to fit the memory bus capacity; but the contention between the computing cores can already create contention penalizing computation performance too.

6.1.2 NUMA systems

Within NUMA systems, the performance of a memory access varies whether a core is accessing its own memory or the memory from another memory bank. Hence, we will use the terms *local* and *remote* to qualify whether computing cores use memory respectively close or far to them.

The main consequence of such NUMA systems is that memory bandwidth will vary whether cores or network interface are accessing local or remote memory. Moreover, depending on where is located memory used for computations or communications, the path taken by the data between the NUMA node and the computing core or the network interface will be different, changing the locations of contention. Thus, our model has to take into account on which NUMA node data manipulated by computations or communications are located.

To focus on the interferences between computations and communications, **we will not mix local and remote accesses from computing cores**. This means we will model performance of computations and communications when cores of only one socket

are computing. Considering computing cores of all sockets accessing the same NUMA node (thus, some of them are doing local accesses and other ones remote accesses) is another problematic that is left for future work.

6.1.3 Last level caches

The last-level cache (L3 cache on most machines), between cores and RAM memory, tends to alleviate the number of memory transfers done by computations. Thus, we would overestimate the number of memory movements if we assumed that every memory access instruction would lead to an actual transfer through the whole memory system. It would lead to inaccurate results about contention since our model only takes as input actual memory transfers.

If the data of the streams we are predicting the bandwidth go through the last-level cache, our model has to describe two phenomena: the contention on memory bus and the behaviour of the cache. However, the behaviour of the cache is complex to model [33, 14], implements undocumented strategies different for each processor manufacturer, and changes for each kind of application. All in all, modeling the cache is another topic, different from the one we are currently dealing with.

For all these reasons, **we chose to ignore the last-level cache and make the data stream bypass it**.

6.1.4 Modeling methods

A widespread model for contention is queuing theory [35, 119]: cores or network interfaces are customers; when they make a memory request, they enter in the queue: they leave the queue when the request is processed. Closed-form expressions exist for properties of such queues, especially the mean time spent in a queue. Unfortunately this kind of model is not relevant for our usecase. Since NUMA machines have a hierarchical organization of their memory, bottlenecks can appear on several places in the memory system (see Figure 5.5 on page 106). Each place where contention can occur has to be represented by a dedicated queue, and the different queues of all memory components have to be combined to model the behaviour of the whole memory system. Correctly assembling the queues requires to have a sharp understanding of how the memory system works (knowledge usually not available publicly and specific to each processor generation and manufacturer). Even if we succeed in proposing an assembly of queues, getting all parameters of all queues would require lot of execution samples to be precise enough. Moreover, obtained parameters characterizing the queue can lack physical meaning, making the parameter interpretation harder. Most queuing models are built with the assumption that all customers have the same request rate; it is not necessarily true in our case: one network interface can issue memory requests at a higher rate than one computing core (a single computing core can reach a memory bandwidth of 5 GB/s, while network bandwidth can be around 10 GB/s). In such situation, we lose the closed-form expressions, which were the main advantage of queuing theory.

We chose a simpler model, easier to manipulate, but accurate enough for our needs:

a basic threshold. While the bandwidth required by all issuers of memory requests stays under the memory bus capacity, there is no contention, no impact on performance. When it does not fit the memory bus anymore, only the bus capacity is available, and is split among computing cores and network interface. This model, described in detail below, has the advantages of requiring few application runs to calibrate the parameters and has understandable parameters with a physical meaning, well-known units, and coherent values regarding performed benchmarks and hardware features.

6.2 A model for memory bandwidth sharing

The model aims at giving the memory bandwidths available for computations and communications, and thus predicts the impact of the contention on their performance, using as parameters the number of computing cores, the memory location of data used by computations and communications and the topology of the machine.

Since memory bandwidth depends on which NUMA node is accessed, we need to instantiate our model once for *local* accesses, noted $\mathcal{M}_{local}$ (*e.g.* memory for computations and communications located on the first NUMA node of the first socket), and once for *remote* accesses, noted $\mathcal{M}_{remote}$ (*e.g.* memory for computations and communications located on the first NUMA node of the second socket). Parameters of each model are defined with metrics collected on executions where data used for computations and communications are on the same NUMA node (case with the largest contention). Once parameters are collected, performance can be predicted according to these parameters. Performance with memory placement configurations other than the ones used to instantiate the model is predicted by combining the local and remote models. This section explains how model parameters are defined, then how the model predicts performance and finally how models for local and remote accesses are combined to predict performance of all possible data placements.

6.2.1 Model parameters

The model requires several parameters describing the behaviour of the machine when communications and computations are executed independently, to know nominal performance and predict them correctly when there is no contention, and in parallel, to know what can be the impact of contention.

A convenient way to understand how bandwidths which will be predicted by the model evolve is to sum memory bandwidths for computations and communications and visualize them by stacking them. Since both streams share parts of the same memory system, it allows to easily represent the share of the bus capacity among the two different streams. Figure 6.1 is an example of such representation: according to the number of computing cores, the orange area depicts memory bandwidth for all computing cores and blue area depicts memory bandwidth for communications, when they are both executed in parallel. We also show the graph of the memory bandwidth for computation executed alone, in green.

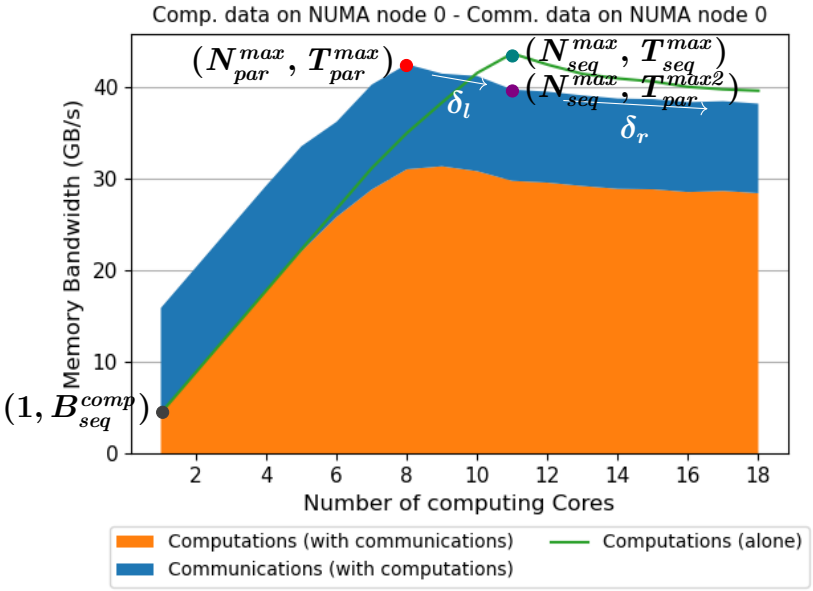


Figure 6.1: Stacked memory bandwidth for computations and communications, with coordinates of the interesting points to instantiate the model: memory bandwidth for one computing core, the maximum memory bandwidths reached with lonely computations, with computations and communications in parallel, and the loss of total memory bandwidth when additional cores are computing. The values are obtained with the benchmark described in section 6.3.1.

One can notice memory bandwidth for computations alone scales perfectly from B_{seq}^{comp} (● on the plot, with one computing core) until T_{seq}^{max} with N_{seq}^{max} computing cores (● on the plot). With more computing cores, the memory bandwidth slightly decreases almost linearly.

When computations and communications are executed in parallel, the maximum bandwidth is different (T_{par}^{max}, ● on the plot) than when computations are executed alone, as well as the number of computing cores (N_{par}^{max}) necessary to reach this maximum. With more computing cores, the total bandwidth decreases linearly too, but with a slope discontinuity when there are N_{seq}^{max} computing cores. Between N_{par}^{max} and N_{seq}^{max} computing cores, each additional computing core reduces the total bandwidth T_{par}^{max} by δ_l. With more than N_{seq}^{max} computing cores, each additional computing core reduces the total bandwidth for computations and communications with N_{seq}^{max} computing cores (T_{par}^{max2}, ● on the plot) by δ_r.

Other important parameters of the model are related to network performance. Network communications require a bandwidth B_{seq}^{comm} when they are executed alone (the nominal performance, not appearing in Figure 6.1). Although it is difficult to perceive on this figure, the memory bandwidth available for communications is reduced in the worst case by a factor α, computed as follow: $\alpha = \min_i(\frac{B_{par}^{comm}(i)}{B_{seq}^{comm}})$, where i represents the number of computing cores. On most machines we evaluated the model, the factor α has a value around 0.9 (*i.e.* the network bandwidth has 10 % of its nominal value) for a number of

computing cores higher than N_{seq}^{max} (precise values can be found in Appendix E).

Introduced notations characterize behaviour of the memory system and compose the model parameters. To sum up, the model requires the following parameters to predict memory bandwidth for computations and communications:

- N_{par}^{max}, T_{par}^{max}: the maximum total memory bandwidth T_{par}^{max} reached when computations and communications are executed simultaneously, and with how many computing cores N_{par}^{max} it is reached;
- N_{seq}^{max}, T_{seq}^{max}: the maximum memory bandwidth T_{seq}^{max} reached when computations are executed alone, and with how many computing cores N_{seq}^{max} it is reached;
- T_{par}^{max2}: the total memory bandwidth when communications are performed and N_{seq}^{max} cores are computing in parallel;
- δ_l, δ_r: the memory bandwidths lost per additional computing core when there are respectively between N_{par}^{max} and N_{seq}^{max} computing cores and when there are more than N_{seq}^{max} computing cores;
- B_{seq}^{comp}: the memory bandwidth used by one single computing core;
- B_{seq}^{comm}: the communication bandwidth when communications are executed alone;
- α: the ratio of the available bandwidth for communications in the worst case.

6.2.2 Modeling memory bandwidth

The goal of our model is to predict the performance of computations *and* communications when they are executed in parallel, for every possible number n of computing cores on one socket. It also predicts performance when computations and communications are executed independently, to be able to predict performance of memory binding configurations without contention.

Memory bandwidths are predicted in two steps: first, the total bandwidth $\mathcal{T}$ the memory system can support according to the number of computing cores is estimated, then this total bandwidth is split between communications and computations.

The maximum available bandwidth $\mathcal{T}(n)$ for computations and communication when n cores are computing is given by the following equation:

$$\mathcal{T}(n) = \begin{cases} T_{par}^{max} & \text{if } n \leq N_{par}^{max} \\ T_{par}^{max} - \delta_l \times (n - N_{par}^{max}) & \text{else if } N_{par}^{max} < n \leq N_{seq}^{max} \\ T_{par}^{max2} - \delta_r \times (n - N_{seq}^{max}) & \text{otherwise} \end{cases} \tag{6.1}$$

The different cases linearly approximate the maximum bandwidth: while there are less than N_{par}^{max} computing cores, the bandwidth is at its higher level T_{par}^{max}; when there are more computing cores, contention starts to impact total bandwidth and for each additional core, δ_l or δ_r is subtracted, whether there are less or more than N_{seq}^{max} computing cores (corresponding to the left or the right of the inflexion point ● on Figure 6.1).

Bandwidth allocated to computations and communications follows different equations which depend if satisfying computing core requirements ($n \times B_{seq}^{comp}$) and assuring minimum bandwidth to communications ($\alpha \times B_{seq}^{comm}$) is lower than the bus capacity or exceeds it. The bandwidth required to fit into the bus, noted $R(n)$, is given by the following formula:

$$R(n) = n \times B_{seq}^{comp} + \alpha \times B_{seq}^{comm} \tag{6.2}$$

The share of the total bandwidth allocated to computing cores is then described by the following equation:

$$\mathcal{B}_{par}^{comp}(n) = \begin{cases} n \times B_{seq}^{comp} & \text{if } R(n) < \mathcal{T}(n) \\ \mathcal{T}(n) - \mathcal{B}_{par}^{comm}(n) & \text{otherwise} \end{cases} \tag{6.3}$$

While all memory bandwidth requested by computing cores and minimal bandwidth assured for communications fit in the total available bandwidth, computations on n computing cores will get the memory bandwidth $\mathcal{B}_{par}^{comp}(n)$, corresponding to their request (perfect scaling). When the threshold is reached, computations get the remaining bandwidth after communications got their share of the bandwidth.

The bandwidth for communications is allocated as stated by the following equation:

$$\mathcal{B}_{par}^{comm}(n) = \begin{cases} \min(\mathcal{T}(n) - \mathcal{B}_{par}^{comp}(n), B_{seq}^{comm}) & \text{if } R(n) < \mathcal{T}(n) \\ \alpha(n) \times B_{seq}^{comm} & \text{otherwise} \end{cases} \tag{6.4}$$

While $R(n)$ is lower than the bus capacity $\mathcal{T}(n)$, communications get the share of the total bandwidth unused by computing cores, but they cannot use more than the nominal performance of the network B_{seq}^{comm} (hence the min). When $R(n)$ exceeds the bus capacity, the bandwidth for communications is impacted by a factor $\alpha(n)$:

$$\alpha(n) = \begin{cases} \frac{\mathcal{B}_{par}^{comm}(i)}{B_{seq}^{comm}} - \frac{\frac{\mathcal{B}_{par}^{comm}(i)}{B_{seq}^{comm}} - \alpha}{N_{seq}^{max} - i} \times (n - i) & \text{if } N_{seq}^{max} - N_{par}^{max} > 1 \text{ and } n < N_{seq}^{max}, \\ & \text{where } i = \max_j(\{j | R(j) < \mathcal{T}(j)\}) \\ \alpha & \text{otherwise} \end{cases} \tag{6.5}$$

With N_{seq}^{max} or more computing cores, communications get their minimal assured bandwidth, thus $\alpha(n) = \alpha$. When there are less computing cores, more than one core between N_{par}^{max} and N_{seq}^{max}, and $R(n) \geq \mathcal{T}(n)$ (*i.e.* the case when $\alpha(n)$ has to be computed), bandwidth for communication does not abruptly drop to $\alpha \times B_{seq}^{comm}$. Therefore, we linearly interpolate the factor, with a line passing by the points where the factor of impact on communications with the maximum number of computing cores where $R(n) < \mathcal{T}(n)$ is still valid (noted i in the equation), and α with N_{seq}^{max} computing cores.

To predict performance on all memory placement configurations, the model needs in some configurations to predict performance of computations and communications executed alone, when there is no contention. The bandwidth for communications executed alone is simply the model parameter B_{seq}^{comm}. The bandwidth for computations executed alone is given by the following formula:

$$\mathcal{B}_{seq}^{comp}(n) = \min(n \times B_{seq}^{comp}, \mathcal{T}(n), T_{seq}^{max}) \tag{6.6}$$

The formula considers a perfect scaling of memory bandwidth allocated to computing cores, limited by the memory bus capacity $\mathcal{T}(n)$ and cannot neither exceed the maximum bandwidth T_{seq}^{max} when computations are executed alone.

6.2.3 Model NUMA effect

NUMA systems present different memory bandwidths depending if accesses are made to a local or a remote NUMA node. Therefore, we need two model instantiations, each with its own set of parameter values. The set of parameters describing local accesses, when both computations and communications make memory accesses to the same local NUMA node (regarding to computing cores), is noted $\mathcal{M}_{local}$, and conversely, the set of parameters describing remote accesses, when they make memory accesses to the same NUMA node of a another socket, is noted $\mathcal{M}_{remote}$.

Using equations 6.1 to 6.5, we can model performance for the two memory binding configurations we used to calibrate the two models (data for computations and communications both on the same NUMA node than the NIC and on the other NUMA node) , by directly using the corresponding model. However, we need to combine these two models to predict performance on all other memory binding configurations. Predicting bandwidths of computations and communications requires now two additional parameters, to take into account data location: the index of the NUMA node where is located data used by computations (m_{comp}) and by communications (m_{comm}). These parameters, in addition to the number of NUMA nodes per socket (noted $\#m$), allow to select the corresponding bandwidth according to placement.

In the rest of the section, the notation $\mathcal{B}(\mathcal{M})$ means the bandwidth $\mathcal{B}$ is given by using the model instantiation $\mathcal{M}$.

Regarding communications, the model to apply is selected with the following equation:

$$\mathcal{B}_{par}^{comm}(n, m_{comp}, m_{comm}) = \begin{cases} \mathcal{B}_{par}^{comm}(\mathcal{M}_{remote}, n) & \text{if } m_{comp} \geq \#m \text{ and } m_{comp} = m_{comm} \\ \mathcal{B}_{par}^{comm}(\mathcal{M}_{local} \setminus B_{seq}^{comm}(\mathcal{M}_{remote}), n) & \text{else if } m_{comm} \geq \#m \\ \mathcal{B}_{par}^{comm}(\mathcal{M}_{local}, n) & \text{otherwise} \end{cases} \tag{6.7}$$

If both computations and communications access to the same remote NUMA node, communication bandwidth is given by the remote model $\mathcal{M}_{remote}$. In all other cases, communications are less subject to contention and follow the local model $\mathcal{M}_{local}$. However, on some machines, the network performance is very sensible to the locality of exchanged data. Since $\mathcal{M}_{local}$ is instantiated with communication bandwidth with data located on the local NUMA node, it may not fit the network performance when data for communications are located on the remote NUMA node. Therefore, in this case, we use the local model, but with the nominal network performance when data is located on remote memory, *i.e.* the B_{seq}^{comm} of $\mathcal{M}_{remote}$.

The model for computation bandwidth is selected with the following equation:

$$\mathcal{B}_{par}^{comp}(n, m_{comp}, m_{comm}) = \begin{cases} \mathcal{B}_{par}^{comp}(\mathcal{M}_{local}, n) & \text{if } m_{comp} < \#m \text{ and } m_{comp} = m_{comm} \\ \mathcal{B}_{seq}^{comp}(\mathcal{M}_{local}, n) & \text{if } m_{comp} < \#m \text{ and } m_{comp} \neq m_{comm} \\ \mathcal{B}_{par}^{comp}(\mathcal{M}_{remote}, n) & \text{if } m_{comp} \geq \#m \text{ and } m_{comp} = m_{comm} \\ \mathcal{B}_{seq}^{comp}(\mathcal{M}_{remote}, n) & \text{if } m_{comp} \geq \#m \text{ and } m_{comp} \neq m_{comm} \end{cases} \tag{6.8}$$

Computations are impacted by contention only when data used for communications is

on the same NUMA node as data for computations. In such case, bandwidth for computations is the one with communications in parallel $\mathcal{B}_{par}^{comp}$, from the model corresponding to computation data location, local or remote. In the same fashion, when computations and communications do not use the same NUMA node for their data, computations get their nominal memory bandwidth $\mathcal{B}_{seq}^{comp}$.

Appendix D presents algorithms to predict memory bandwidth for computations and communications, implemented using equations described above.

6.3 Evaluation of the model

We want to measure the impact of memory contention on computations and communications, when they are executed in parallel, and to compare it with the predictions of our model. To know the impact, we need the performance of computations and communications executed alone and in parallel.

6.3.1 Experimental setup

Benchmarking program

We used the same benchmarking program we presented in Chapter 5 to get performance of computations and communications, executed either alone or in parallel.

Performance is measured on a single node, but we still need two machines for network exchanges. We study the performance penalty caused by memory contention, therefore, to control and understand memory movements, all computing cores perform *non-temporal* `memset` instructions to move data from cores to memory, and communication performance is measured with the bandwidth observed to receive messages of 64 MB from the other machine. We use *non-temporal* instructions to bypass the last level cache, as explained in section 6.1.3: they tell the processor to store data directly in memory, bypassing the cache.

Data used for communications and computations are explicitly bound on selected NUMA nodes, to know the data location and consider it in the model. Unlike what we did in the previous chapter, buffers for computations and communications are not necessarily allocated on the same NUMA node.

Memory bandwidth for computations is computed from the duration of the `memset` instructions, each computing core always work on the same amount of data (weak scaling). Memory bandwidth for communications is considered to be the same as the network bandwidth, *i.e.* the message size over the necessary time to receive data from the other machine, since this stream has also to go through the memory bus after arriving on the network interface.

Only samples collected during steady state are considered: all cores execute computation iterations for a defined amount of time, then we skip performance of first and last iterations of each core, to get rid of the performance when not exactly all cores are

computing.

To bind memory on a specific NUMA node, bind threads to cores and gather topology information, we use HWLOC [27].

Modeling all placements

With NUMA machines, we need two model instantiations: one for local memory accesses and another one for remote accesses. On a machine with two sockets (processors) and two NUMA nodes per socket, we would execute our benchmarking program to get model parameters using memory for computations and communications both located on the first NUMA node of the first socket for the local model and using memory located on the first NUMA node of the second socket for the remote model. Thus, we need to measure memory bandwidths of two placement configurations, to latter be able to predict performance of all other configurations (16 in this example, since there are 4 possibilities where to put data for computations and the same 4 possibilities for communication data), as explained in section 6.2.3.

The program to measure memory bandwidth of one placement configuration needs to be executed for all possible numbers of computing cores, in the range of the number of cores on the first socket, as explained in section 6.2. Once the performance metrics (memory bandwidth for computations alone and in parallel of communications, network bandwidth for communications alone and in parallel of computations) are extracted from benchmark outputs, the evolution of the bandwidths over the number of computing cores is analyzed (it mostly looks for minimums and maximums) and the parameters of the model, listed in section 6.2.1, are computed.

Note that this process can be optimized: once the peaks of bandwidth T_{par}^{max} and T_{seq}^{max} are found, one can skip executions with number of computing cores greater than N_{seq}^{max}, except the execution with all cores of the first socket, required to compute δ_r. Here we still need to execute the program with all possible numbers of computing cores, to evaluate the accuracy of our model.

Testbed platforms

We evaluated our model for the bandwidth metrics obtained on several platforms with different characteristics: from small experimental platforms to large production ones, presented in Appendix C. Hyperthreading is only enabled on platforms `dahu`, `grvingt`, `pyxis` and `occigen`, however, on all platforms, threads are bound to physical cores (*i.e.* hyperthreads are not used).

Values of model parameters for each platform are presented in Appendix E.

6.3.2 Results

Figures 6.2 to 6.7 depict performance of computations and communications as well as the model predictions. Each figure is composed of several subplots, one per possible

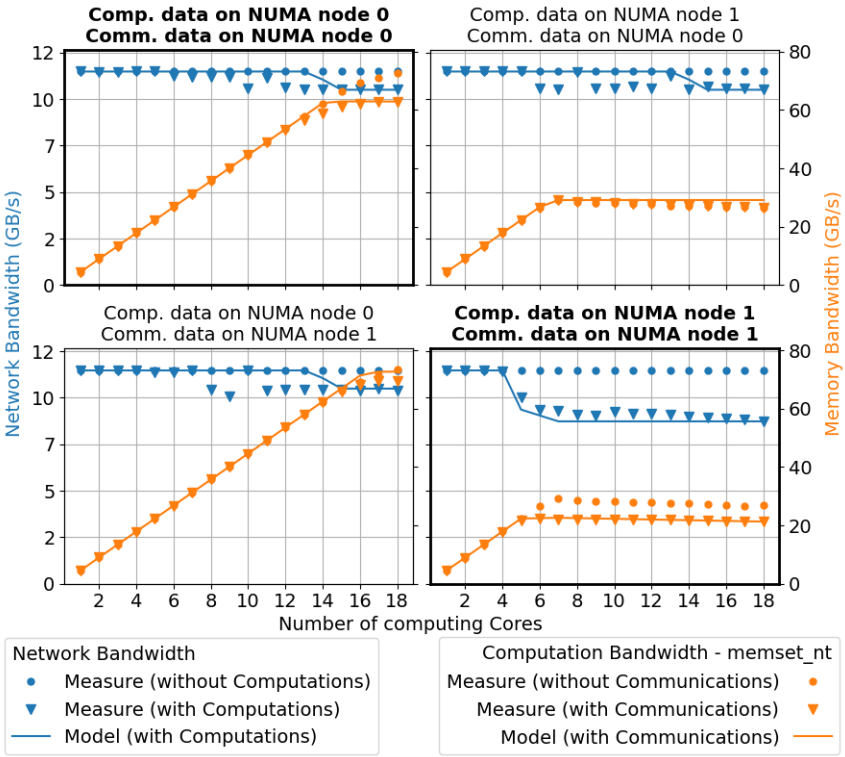


Figure 6.2: Performance of computations and communications along with our model prediction, on `henri` nodes (Intel, InfiniBand).

placement combination of data for computations and data for communications on available NUMA nodes. For instance, Figure 6.2 represents results on the `henri` platform, with 2 NUMA nodes. Data for communications can be located on 2 NUMA nodes, as well as data for computations, which leads to 4 placement combinations. Each line of plots represents one placement for communication data, while columns represent placements for computation data. Titles above each plot precise the placement of data. The two placement combinations used to instantiate the local and remote models are highlighted with a bold title and a thicker frame. Each subplot presents, according to the number of computing cores, network bandwidth (in blue, to be read on the left Y-axis) and memory bandwidth for computations (in orange, to be read on the right Y-axis), when they are executed alone (● markers) and in parallel (▼ markers). The blue and orange lines indicate our model predictions of the bandwidth for respectively communications and computations. Error bars are not shown to ease reading, but the run-to-run variability is very low.

`henri` Figure 6.2 shows there is contention between computations and communications, impacting them both, more or less severely according to data placement. Our model is accurate when computations and communications perform both remote memory accesses. When computations and communications perform both local accesses, our model reflects the correct impact on communications too late (the model predicts a decrease starting with 14 computing cores, while it is 10 in reality), because communications start to be impacted before the total bandwidth threshold $\mathcal{T}$ is reached. Other memory placement

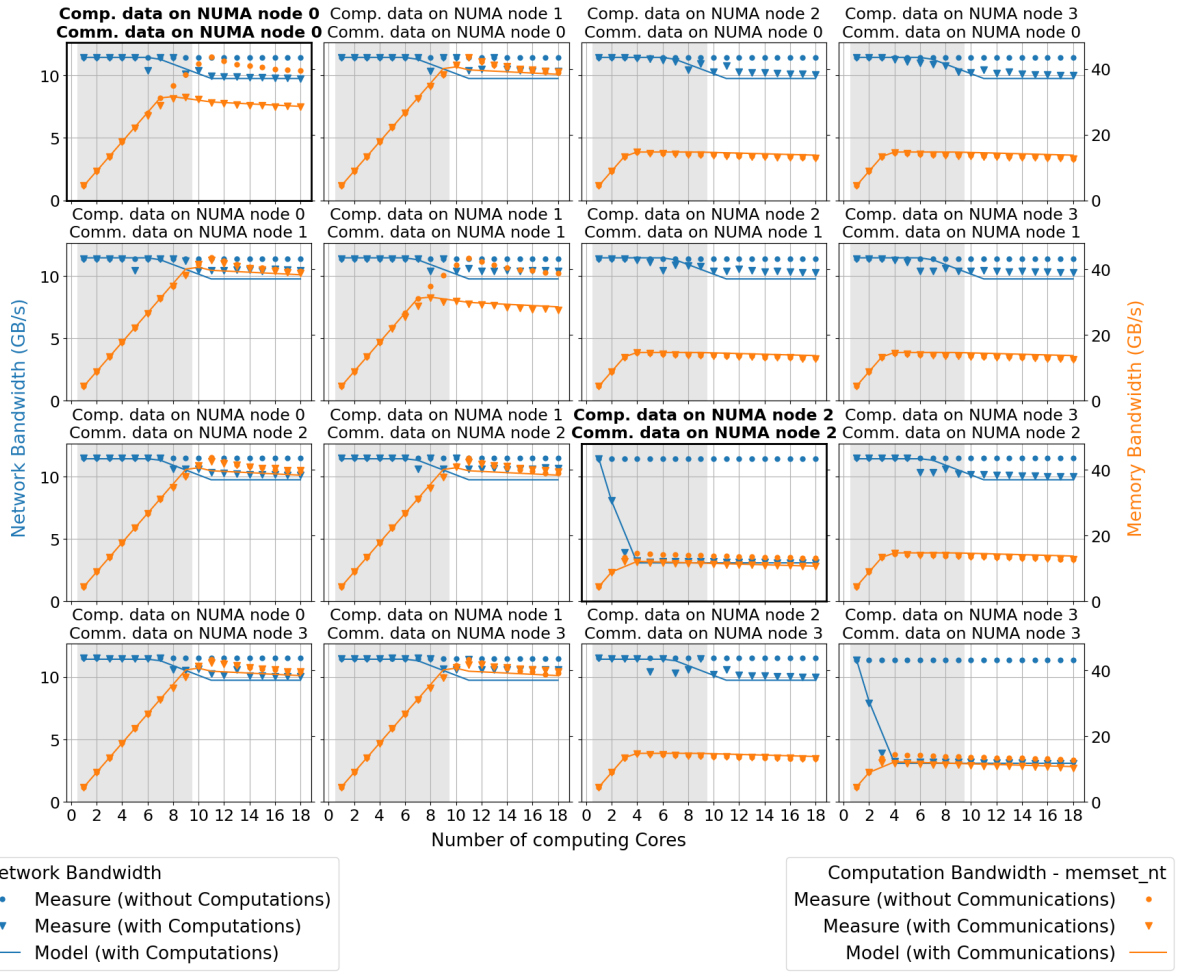


Figure 6.3: Performance of computations and communications along with our model prediction, on `henrisubnuma` nodes (Intel, InfiniBand).

configurations, not used to instantiate the model, show the same flaws.

`henrisubnuma` The `henri` platform configured with 4 NUMA nodes allows 16 data placement combinations, described by Figure 6.3. The grey and white areas are used to distinguish the two NUMA nodes of the first socket. With such numerous configurations, symmetries in performance appear, mimicking symmetries of the machine topology: for instance, when data for computations and communications are on different NUMA nodes of the second socket (right half of set of plots), performance is always the same, regardless of which NUMA nodes are used. These symmetries allow the predictions done by the model to be correct, with only two configurations used to predict all 16 combinations. All these configurations also show that the placement configurations the most disturbed by memory contention are the ones where data for computations and communications are on the same NUMA node (*i.e.* subplots on the diagonal of the figure), while computations are almost not impacted in other cases. Therefore, we can guess memory contention occurs the most on memory controllers (responsible of accesses to one dedicated NUMA node), rather that on the inter-socket connection bus.

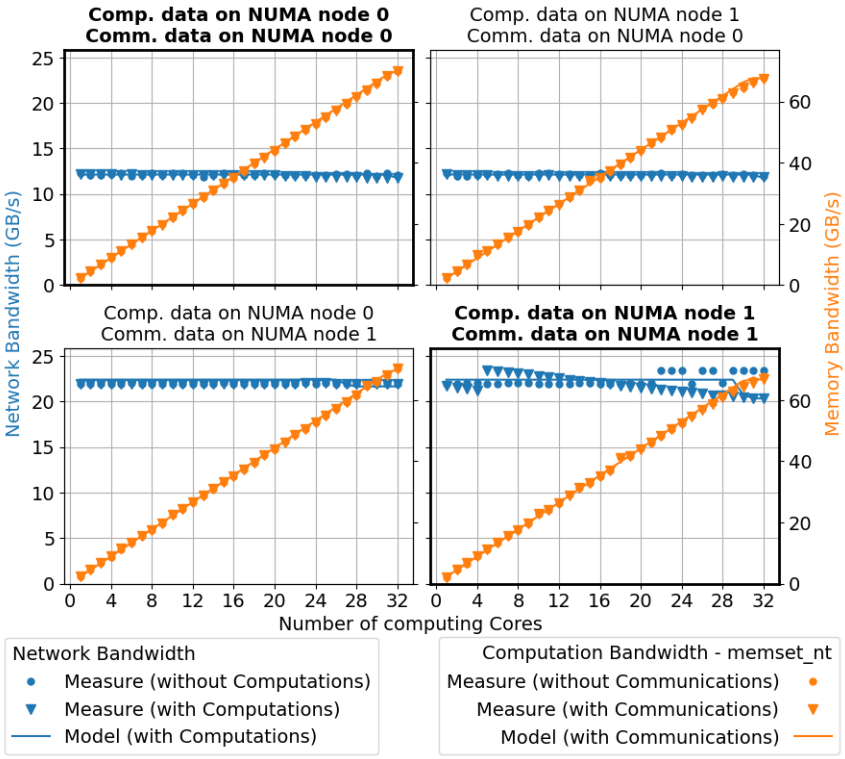


Figure 6.4: Performance of computations and communications along with our model prediction, on `diablo` nodes (AMD, InfiniBand).

Figure 6.1 used previously to explain the model is the stacked version of the top left subplot of the Figure 6.3.

`diablo` Figure 6.4 shows results on the `diablo` platform, and illustrates especially the case when network performance is highly sensible to data locality: when data for communications is on the first NUMA node, network bandwidth reaches only 12.1 GB/s whereas when data is on the second NUMA node (which the NIC is actually plugged to), network bandwidth can raise up to 22.4 GB/s. Our model succeeds in predicting performance, even if there is almost no contention on this platform.

`billy` Figure 6.5 depicts results on the `billy` platform, similar to `diablo`. The network performance is still sensible to placement (we get stable 14 GB/s when data for communications is on the first NUMA, up to 20 GB/s otherwise), but is more chaotic (when data for communications is on the second NUMA node, network bandwidth oscillates between 12 GB/s and 24 GB/s); the model fails to capture the variability, but follows the general trend of observations.

`occigen` Figure 6.6 shows results on the only production platform of our testbed. On this ancient platform (2014-2022), only computations are impacted when computations and communications do both remote memory accesses. For instance, with 7 computing cores, memory bandwidth for computations decreases from 21.1 GB/s to 18.5 GB/s when

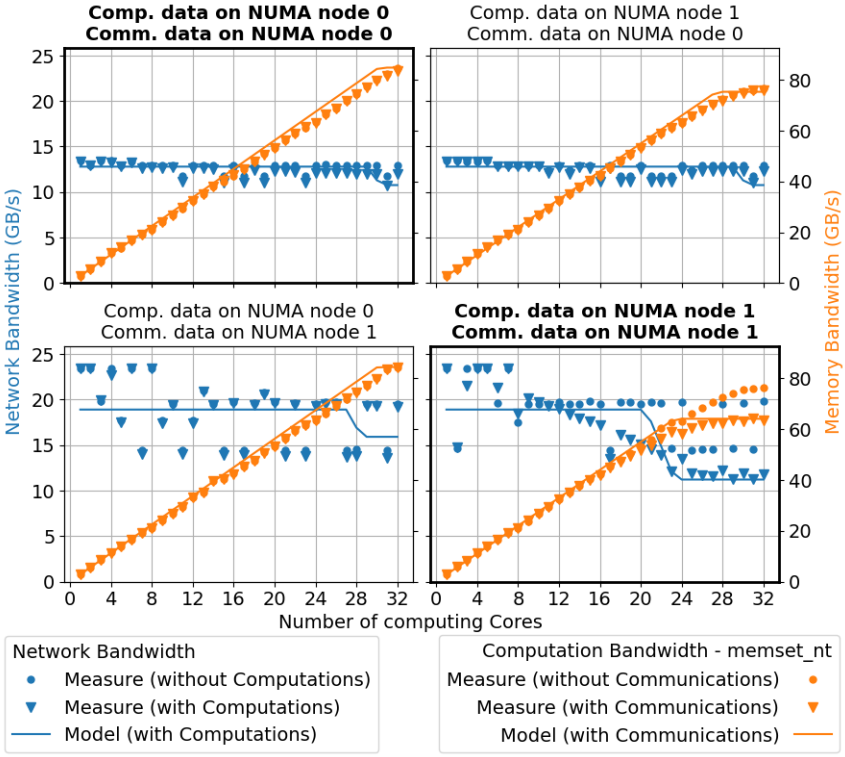


Figure 6.5: Performance of computations and communications along with our model prediction, on `billy` nodes (AMD, InfiniBand).

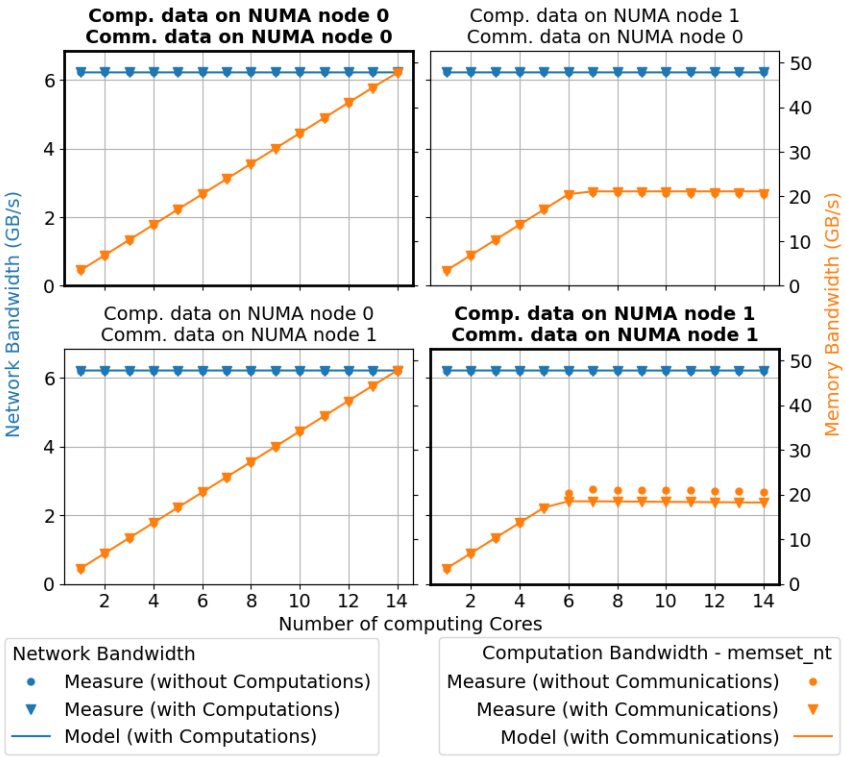


Figure 6.6: Performance of computations and communications along with our model prediction, on `occigen` nodes (Intel, InfiniBand).

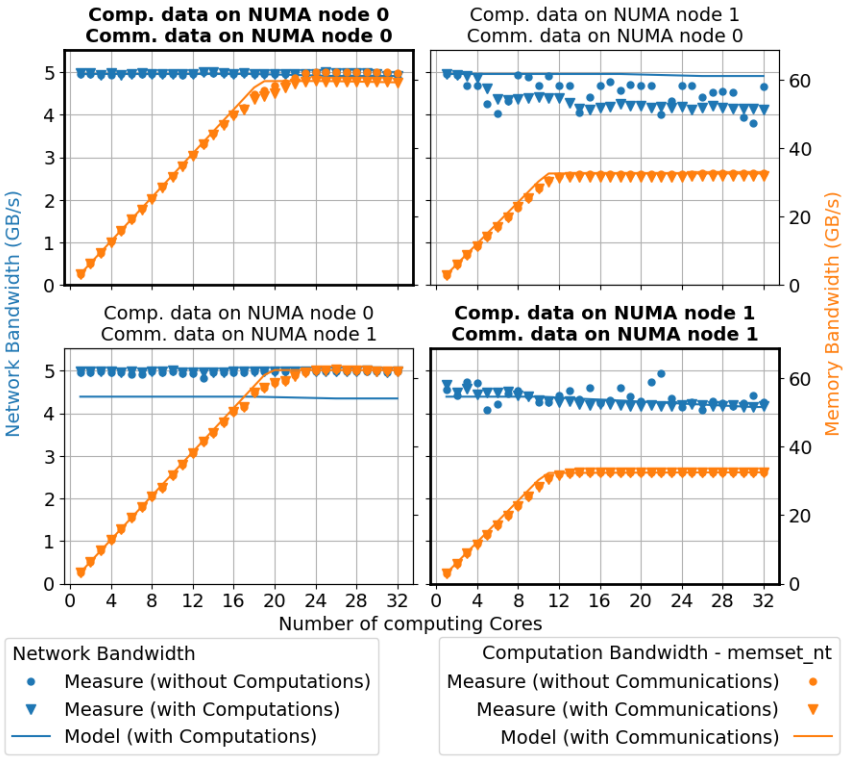


Figure 6.7: Performance of computations and communications along with our model prediction, on pyxis nodes (ARM, InfiniBand).

communications are executed in parallel. Network bandwidth stays always constant at 6.2 GB/s. This platform is where our model is the most accurate, with the lowest prediction error (see further).

pyxis Figure 6.7 shows results on a platform with ARM processors. Our model predicts correctly performance of computations, although it does not catch that memory bandwidth for computations does not scale well when it gets closer to the threshold. For instance, with data for computations on the first NUMA node and data for communications on the second node (bottom left plot), our model predicts a memory bandwidth of 62.5 GB/s for 19 computing cores, while in reality is 58.7 GB/s. Network performance is not correctly predicted for placement configurations which were not used to instantiate the model. On this architecture, the network performance seem to be harder to predict by just relying on the locality of the data.

bora, dahu, grvingt Model predictions for platforms equipped with similar hardware (Intel processor and Omni-Path network) give as expected similar results, as can be seen on figures 6.8, 6.9 and 6.10.

Table 6.1 reports the prediction errors on all platforms. The error is estimated with the mean absolute percentage error ($\frac{100\,\%}{n}\sum_{k=1}^{n}\left|\frac{a_k-p_k}{a_k}\right|$), for predictions of computations and communications separately, by distinguishing also predictions made by the model on a placement configuration used to instantiate the model (*samples*) or not (*non-samples*).

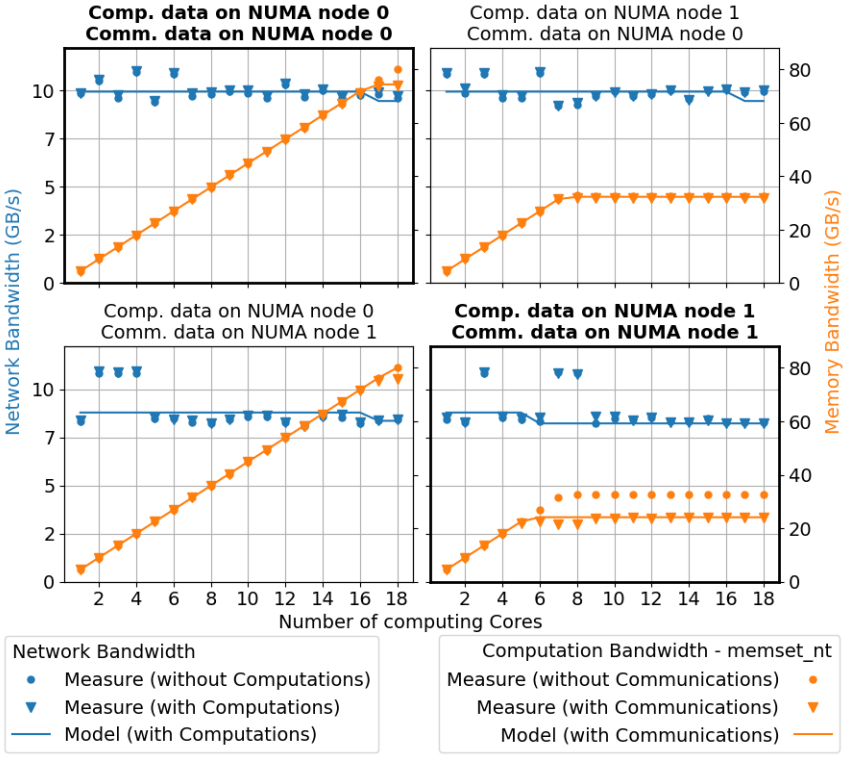


Figure 6.8: Performance of computations and communications along with our model prediction, on `bora` nodes (Intel, Omni-Path).

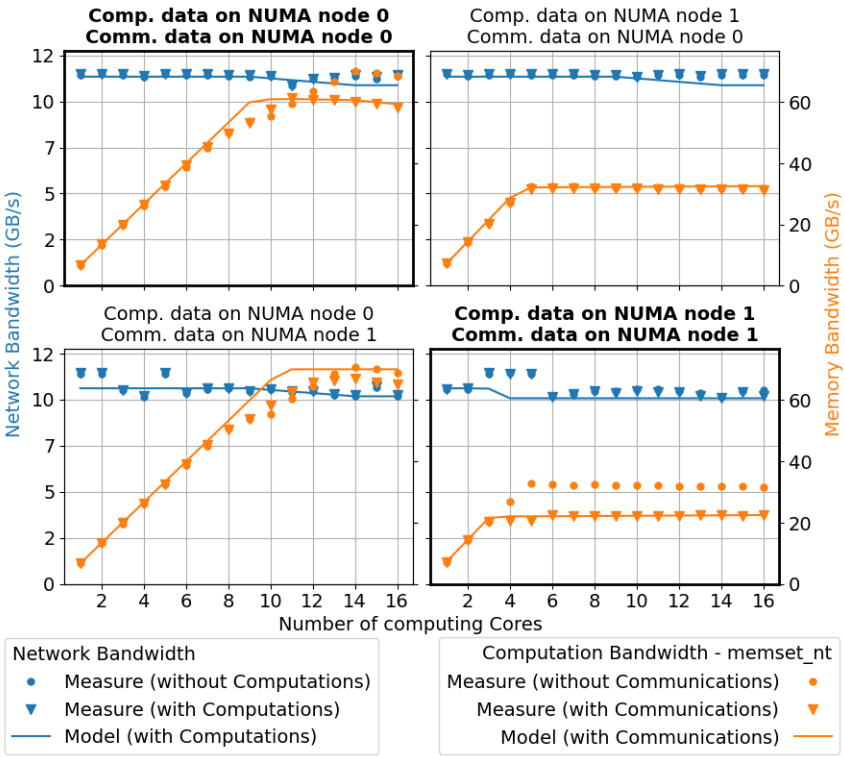


Figure 6.9: Performance of computations and communications along with our model prediction, on `dahu` nodes (Intel, Omni-Path).

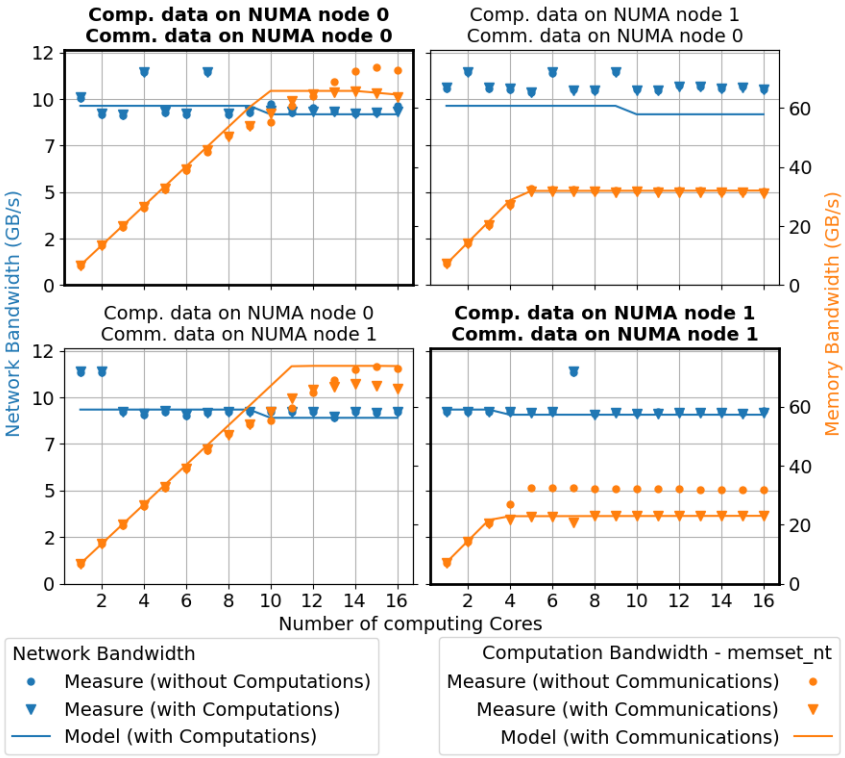


Figure 6.10: Performance of computations and communications along with our model prediction, on grvingt nodes (Intel, Omni-Path).

Platform	Communications			Computations			Average
	on Samples	*on non-Samples*	*all*	*on Samples*	*on non-Samples*	*all*	
henri	2.62 %	3.53 %	3.08 %	0.80 %	2.34 %	1.57 %	2.32 %
henrisubnuma	2.90 %	3.80 %	3.69 %	1.89 %	3.66 %	3.44 %	3.56 %
billy	8.22 %	10.84 %	9.53 %	3.98 %	3.40 %	3.69 %	6.61 %
bora	4.39 %	5.14 %	4.77 %	1.34 %	0.78 %	1.06 %	2.91 %
dahu	2.76 %	2.38 %	2.57 %	2.00 %	3.85 %	2.92 %	2.74 %
diablo	2.32 %	1.54 %	1.93 %	0.92 %	0.99 %	0.95 %	1.44 %
grvingt	3.41 %	8.06 %	5.74 %	2.44 %	4.48 %	3.46 %	4.60 %
pyxis	1.15 %	13.32 %	7.24 %	1.95 %	2.79 %	2.37 %	4.80 %
occigen	0.01 %	0.01 %	0.01 %	0.22 %	0.58 %	0.40 %	0.20 %
Average	3.09 %	5.40 %	4.28 %	1.73 %	2.54 %	2.21 %	**3.24 %**

Table 6.1: Model errors on testbed platforms.

Regarding communications, the highest prediction error on all configurations is on billy (8.22 % on sample configurations, 10.84 % on others), explainable by the high variability of network performance, even when communications are executed alone, not taken into account by our model. The prediction error is also high on pyxis, especially on non-sample configurations (13.32 %), caused by the wrong appreciation of locality impact on this architecture, as discussed above. On other platforms, the average of prediction error of network bandwidth on all placement configurations is below 6 %. Performance of computations is better predicted, with an overall error lower than 4 %. Worst cases are on billy (3.69 %) and grvingt (3.46 %), where the model tends to over-estimate the bandwidth for computations because it assumes a perfect scaling when the number of computing cores increases, but in reality computing cores start to create contention before reaching the bandwidth threshold.

6.3.3 Discussion

Results presented above show our model is valid to predict memory bandwidth allocated to communications and to computations: from sample executions on two different placement configurations to instantiate the whole model, we are able to predict bandwidths with all possible placement configurations, with an overall prediction error lower than 4 %. Higher prediction errors come most often from unstable input data, nonetheless the model formulation allows us to better understand in which circumstances memory contention happens and how the hardware deals with it.

Model limits

Even though our model makes overall good predictions, there are some corner cases where it show its limits. It has difficulties to accurately predict network bandwidth if network performance is not stable even without contention (see for instance results on billy, pyxis, bora or grvingt). On systems where data locality can highly influence network performance (such as diablo, billy, pyxis or bora), the model can be wrong more often, especially on placement configurations not used to instantiate the model. These weaknesses are not related to modeling of contention, since the odd network performance is also hard to predict with communications executed alone. Being able to model network performance in all placement configurations, when communications are executed alone, would help improving our model, to predict network bandwidth in case of contention.

On machines with many NUMA nodes (more than 4; for instance, billy can be configured with 8 NUMA nodes – 4 per processor), network performance under memory contention depends on data locality and the heuristic given by formula 6.7 is not sufficiently accurate anymore. Moreover, when communications and computations use the same NUMA node for their data (*i.e.* when contention has the most impact), the distribution of memory bandwidth between computations and communications before the threshold is reached (first cases of equations 6.3 and 6.4) is, in our model, more in favour of computations as in reality. Thus, these more complex system topologies would require more precise hypotheses about memory routing between NUMA nodes to model them accurately. Moreover, evaluating the model on all placement configurations (64 with 8

NUMA nodes!) is more difficult due to necessary time to execute benchmarks on all configurations (about one hour per configuration).

The model predictions are only valid for the parameters of the benchmarks used to instantiate the model: the computation kernels executed by computing cores and the message size used by communications. For different computation kernels and message sizes, memory contention can be different and thus model parameters as well. However, since the computation kernels and message size were chosen here to maximize the contention, other kernels or message size should produce less contention, but the insights provided by our model in the worst case should still be valid.

Lessons learned

Distinguishing location of data used for computations and for communications allows to change paths of the two different data streams in the memory system and thus better locate bottlenecks, where memory contention occurs. First hypotheses assume contention happens in memory controller (controlling the memory of a NUMA node) or in inter-processor link. Results on machines with 2 NUMA nodes show contention occurs when data for communications and computations are located on the same NUMA node, especially on the same *remote* NUMA node (*i.e.* data streams have to go through inter-processor link and memory controller). When communications and computations use each their own NUMA node for their data, memory contention is very low (when not null). Results on machines with 4 NUMA nodes (2 *local* and 2 *remote* nodes, for instance on `henrisubnuma`), refine the location of the bottleneck: when computations and communications do both remote accesses (data streams have to go through the inter-socket link), performance is the most impacted due to contention when they use the same remote NUMA node. Thus, **the place where the most contention occurs is memory controller, and not the inter-socket link**.

The hypotheses made to design the model, and validated with experiments, teach us **memory bandwidth for network communications is the first reduced** in case of memory contention, to preserve memory bandwidth dedicated to computations. However, a **minimum bandwidth is always assured for network**, to prevent starvations. When this minimum bandwidth is reached, bandwidth for computations starts to decrease to fit memory system capacity.

6.4 Conclusion

Computations and communications in parallel distributed HPC applications can be executed in parallel to save execution time. With memory-bound computations and network exchanges with large messages, contention can occur in the memory system, reducing performance of both computations and communications.

In this chapter, we proposed a model to predict memory bandwidth allocated for computations and communications when they are executed in parallel. Predictions are made from parameters describing behaviour of the memory system with two data place-

ment configurations. From these parameters, the topology description of the machine and information about data locality, our model is able to predict memory bandwidth for computations and for communications, regardless on which NUMA node data are located, with an average prediction error lower than 4 %.

Building this model allows to better understand that memory contention is the most severe when computations and communications use data located on the same NUMA node, bottleneck causing this contention is mainly located in the NUMA node controller, rather than in the inter-socket connection bus. In case of contention, the system first degrades memory bandwidth allocated to communications, but ensures a minimum, and then reduces computation bandwidth if necessary.

This knowledge about the system behaviour in case of memory contention could help the runtime system to optimize data placement and the scheduling of memory-bound computations and communications, in order to avoid (or at least minimize) possible memory contention impacting performance of the application.

Conclusion and Perspectives

To address the increasing complexity of HPC machines, task-based runtime systems have emerged. The task-based paradigm decomposes applications into smaller parts with dependencies between the tasks, forming a DAG, scheduled and executed by the task-based runtime system. Task-based runtime systems ease the exploitation of the whole computing power offered by HPC machines and permit better application portability on different machines. Distributed applications can be handled by such runtime systems when they provide a component managing network communications. Network communications are usually delegated to a third-party library.

Task-based runtime systems have specific requirements in terms of communications, because of their asynchronous and dynamic behaviour. In this thesis, we explored the possible interactions between task-based runtime systems and communication libraries, especially between StarPU and NewMadeleine, in order that these two different and independent software layers exchange their respective knowledge to take more clever decisions.

Summary of contributions

We first evaluated and improved the tracing system of StarPU to have precise execution traces describing behaviour of applications, to analyze interactions between StarPU and the communication library. We developed dynamic broadcasts, a positive interaction between StarPU and NewMadeleine to improve application performance. On the other hand, we also evaluated negative interactions between communications and computations when they are executed simultaneously.

Tracing systems

To understand performance and behaviour of application executions, tracing them and then analyzing the execution details is a first-class citizen technique. To be robust enough, tracing systems have to disturb as less as possible the program execution and provide satisfactory precision. Impacting the application execution can change the application performance, but mainly the behaviour of the application: thus the execution described

in the trace files is different than an execution not traced. This can be inconvenient, since we usually trace application to understand what is happening when the application is not traced! A bad precision can lead to inconsistent traces, especially for distributed executions, when the event causality can be broken (a message can appear as received before it was sent) if clocks are not accurately enough synchronized.

We evaluated possible sources of performance overhead when StarPU applications are traced and provided possible solutions when the overheads are too important. We improved the clock synchronization mechanism in StarPU to use state-of-the-art techniques and empirically evaluated the accuracy improvement.

More than providing robust and efficient solutions to some problems caused by tracing systems, our work aimed at making users of such systems aware of these possible flaws, and the execution traces they are analyzing might reflect a distorted regular execution.

This work can be seen as a prerequisite to serenely analyze application executions with tracing systems, with all the possible problems in mind.

Dynamic broadcasts

Communications are one of the bottlenecks for application scalability over many computing nodes. One communication pattern which can be easily optimized is the broadcast, thanks to a tree routing of the required exchanges for all recipients nodes to get the data.

Regular MPI applications can use dedicated routines to perform optimized broadcasts. Unfortunately for task-based runtime systems such as StarPU, several criteria have to be met to use these routines: all nodes in the broadcast have to call the function with the same parameters, especially the list of recipients. Moreover, such call is a sort of synchronization we want to avoid. With the STF model of StarPU and the implicit communications, broadcasts are not explicit and only the sender node knows all the recipients. Recipient nodes do not even know that the data will come from a broadcast and not from a regular point-to-point communication.

We proposed what we called *dynamic broadcasts* to perform optimized broadcasts following tree routing, yet fitting StarPU's constraints. The mechanism to detect broadcasts may miss only few of them, at the beginning of the application execution. Once the data to broadcast is ready to be sent, the interface we developed in NewMadeleine handles it and the StarPU recipient processes receive the data transparently as it came from a point-to-point communication.

Microbenchmarks showed the efficiency of our implementation. Performance gain on applications exhibiting broadcasts in their DAG depends on several factors. Dynamic broadcasts bring up to 30 % performance improvement for the Cholesky factorization and can multiply by 6 the performance of QR factorizations with specific data distribution and matrix shape.

The relevance of dynamic broadcasts illustrates the potential for positive interactions between task-based runtime systems and the communication libraries: with a simple and generic interface, StarPU is able to better take profit from NewMadeleine and NewMadeleine can notify StarPU of different statuses of communications, in order

for StarPU to use the data as soon as possible.

Interferences between computations and communications

While the original goal of the thesis was to develop *positive* interactions between task-based runtime systems and communication libraries, this contribution evaluated the possible drawbacks of executing computations and communications in parallel, as done in many runtime systems.

We studied different source of negative interferences between runtime systems and communications. Frequency variations caused by computations do not have a major impact on communications. The communications launched by runtime abstractions can suffer from an important overhead caused by the call stack to cross before reaching the communication library. But the most important performance degradation come from the memory contention between data moving for computations and data being moved for communications.

Memory contention can be influenced by several factors: data and thread placements, message size and arithmetic intensity of computational tasks. When memory contention occurs, it can impact computations, but mostly performance of communications.

Despite the lack of information regarding contention management in processors and memory systems, we proposed a model to predict the memory bandwidth share between computations and communications, taking into account contention and data locality. The evaluation of our model on a wide range of machines with different architectures confirmed our initial hypotheses: as long as there is no contention, computations and communications get the memory bandwidth they require; when the memory bus capacity is reached, bandwidth of communications is first decreased to preserve the computations as long as possible; a minimal bandwidth for communications is still assured, to avoid starvations; then if the memory bandwidth requirement from computations and communications keeps increasing, the memory bandwidth allocated to computations decreases uniformly across the cores.

Although these negative interactions between computations and communications appear in all programs overlapping communications by computations, the features and abstractions offered by task-based runtime systems would ease the implementations of solutions to avoid (or at least minimize) their impact.

Perspectives

We showed in this thesis interactions between task-based runtime systems and communication libraries can be strengthened to improve the collaboration of these two software layers and increase application performance. These first results pave the way to other possible interactions.

Improve the interaction with other StarPU's features

Some specific features in StarPU are good candidates to carefully consider their interplay with communications.

The distributed extension of StarPU features fault tolerance mechanisms [83] with checkpoint instructions inserted in the task graph. These instructions produce communications to save data and state of a StarPU process to another process, in case the former had a failure. These additional communications for check-pointing generate more traffic on the network and thus can disturb initial communications required for the DAG execution. Priorities of the checkpointing communications can be reduced to favor initial application communications. However, the checkpointing communications cannot be performed too late, to avoid loosing too much data in case of failure. More than just priorities, mechanisms could be included in communication libraries to specifically handle checkpoint messages: splitting (not to occupy the network for a long time), piggybacking, *etc.* Of course, it assumes a robust priority mechanism in the communication library, from both sender and receiver sides.

To tackle the problems of task graph submission time and task granularity (*e.g.* smaller tasks on CPUs and bigger tasks on GPUs), research on *hierarchical tasks* is conducted [50]: a coarse-grain DAG is submitted and then according to which worker will execute a task, the task can issue a sub-DAG with smaller tasks leading to the same result as the bigger original task. Creating on-the-fly smaller tasks requires to *partition* data buffers: split them into smaller ones to feed the tasks of the sub-DAG. Although a distributed version of such mechanism is not implemented yet, it presents several challenges, especially to know the granularity of the received data: is the received data partitioned and has to be used with small tasks? Should we wait to receive all sub-data buffers to reassemble a larger buffer to execute a bigger task? Can we aggregate only available sub-buffers to launch a medium-sized buffer? The main possible problem involving the interaction between the runtime system and the communication library is that receiver process may not know beforehand in which state the data will arrive: partitioned or not.

What about GPUs?

GPUs can execute some types of tasks much faster than CPUs (*e.g.* `GEMM` kernels). Tasks executed faster can mean more communication requests to handle at the same time, very fast. Thus, it requires high reactivity and efficiency from the communication library.

Executing tasks on GPUs implies explicit memory transfers from the RAM memory to the GPU memory. Like network communications, these movements take place on the memory bus, shared by other data streams as well. Our study on memory contention between computations on CPUs and network communications could be extended to consider also the impact of memory transfers with GPUs on CPU computations and communications. This would add a third dimension to the contention problem (memory bandwidth for computations, communications, and now GPU transfers), but on the other hand transfers between GPU memory and the host memory are explicit: the runtime system can decide when to perform these transfers and then computations on GPU will

not disturb CPU computations and communications, as opposed to CPU computations, which generate implicit memory transfers by directly accessing to the RAM memory.

Moreover, GPUs improve their integration to distributed environments. Technologies such as GPUDirect [96] allow to move the data directly between the GPU memory and the NIC, without requiring intermediate copies through the RAM memory. Besides, most recent supercomputers, such as Fugaku, Frontier or Lumi, have GPUs with embedded network interfaces. With such configuration, it is possible to transfer through the network data on the GPU memory, with good performance thanks to the optimum memory affinity of the embedded NIC. These specific configurations increase the complexity of the machine: appropriate functions have to be called to drive the NIC of the GPU and the scheduler has to be clever enough to know data can be directly sent from the GPU, whether it can be received directly by the NIC of a GPU of the remote node or not, *etc*. Task-based runtime systems have been designed to hide such kind of difficulties, hence efficiently integrating mechanisms of 'network-aware' GPUs in task-based runtime systems should be the next move.

GPUs have not been considered in this thesis, nevertheless, considering GPUs and their impact on communications within task-based runtime systems is definitely a relevant future work.

Performance model with communications

Most of the work done around performance models of whole applications is done by neglecting the performance of network communications. This assumption is usually justified by the overlap of communications by computations. When communications are considered, it is in *fork-join* models, where identified communications occur in specific application phases, without computations in parallel. In this case, it is much easier to model the cost of communications in the application execution.

With task-based runtime systems, the dynamic and asynchronous behaviour of the DAG execution makes the model of communication cost more complicated: one cannot know precisely when a communication will start, which communications are on the application critical path (and thus their performance is critical) and which communications are further from the critical path and/or actually overlapped by computations.

The majority of results presented in this thesis shows that changing only the behaviour of communications can have an impact of the whole application performance: communications have to be considered in performance models!

Having such model would satisfy several needs. First, it could tell which performance improvement is expected if only communications are improved. For instance, it could explain the different performance improvements observed with dynamic broadcasts on different machines and predict the performance on other machines. Moreover, we only empirically observed performance improvement with dynamic broadcasts, but maybe the performance are far below the performance that would have predicted the model: the model could guide the improvement of communications and the location of bottlenecks. The second goal of the model would be to understand what are the important network metrics for distributed task-based programs: whether it is the latency or the bandwidth, or

maybe it is not important at all? Such information would help predicting performance of task-based applications on machines with known characteristics and could answer whether the network performance is important or not for a specific application. The last possible use of such model is to provide information to simulators (such as SimGrid, already used by StarPU [110]) for more accurate simulations, especially to simulate configurations hard to obtain in reality (many nodes to study scalability, many GPUs, specific accelerators, *etc*).

Towards a better integration to the scheduler

Currently, network communications are totally ignored by StarPU's scheduler, while transfers between memories are considered in some scheduler policies.

The scheduler could take into account the number of communications and their durations to optimize the application makespan. Avoiding the memory contention between computations and communications observed in this thesis could be a motivation: the scheduler could avoid scheduling memory-bound tasks and communications at the same time.

The fact that intra-node memory transfers are already taken into account by scheduling policies could lead to try to consider all nodes of a StarPU execution as a single big node with inter-node communications becoming sort of intra-big-node memory transfers. However, this idea brings back the master-slave model to have one scheduler instance with a global overview of the task graph and required memory transfer. Unfortunately, this model has scalability issues one cannot ignore these days.

Costly inter- and intra-node movements of data involving communications could be minimized with heuristics in the runtime system. From the local point of view of a StarPU process, buffers which will receive data from the network or buffers containing data which will be sent to the network could be allocated directly (or opportunistically copied) on the NUMA node where the NIC is plugged. The risk is that this NUMA node becomes overloaded, hence a clever heuristic is required. Inter-node communications could be reduced with several mechanisms, for instance redistributing data on nodes for a better load-balancing or work-stealing between nodes. However, the integration of these techniques with the STF model and dynamic task-based runtime systems is not straightforward.

The scheduler might also help predicting the number of occurring communications in the near future. With this information, the polling thread can be suspended (or at least have a reduced activity) when no communications are planned and tasks can be executed on its core; conversely the number of computing cores could be reduced to dedicate more memory bandwidth to communications, when numerous important communications will occur. With the help of model performance of tasks, the scheduler is able to estimate the end time of a running task. If the end of this task releases a communication with high priority, other ready communications could be postponed if they could not be finished before the end of the task, to immediately be able to send the important communication.

Consider other types of applications

We studied mainly communications in task-based runtime systems with dense linear algebra applications. With only this kind of applications, we noticed different behaviours regarding communications which depend on applications: applications with or without broadcasts, memory-bound application causing contentions, *etc*.

This means other types of applications could bring their specific challenges for communications. Possible other applications, among other, are sparse linear algebra, Fast Fourier Transform (FFT), stencils, graph algorithms, neural networks for machine-learning, *etc*. The main difficulty is that these applications have to be compatible with the task model.

Final words

Task-based runtime systems are a solution proposed to tackle the increasing complexity of supercomputers, by providing an abstraction layer between the application and the machine. Different types of task-based runtime systems exist, with different features and different input formats. These runtime systems are usually dynamic and asynchronous, and they implicitly overlap communications by computations.

Most of distributed extensions of task-based runtime systems rely on the *de facto* communication standard MPI, which is not adapted to such usage. Communication libraries better suited to task-based runtime systems are event-based, to easily react on communication events; propose to attach priorities to communications; and support multithreading to be able to submit communication request from different threads. Active messages can be convenient to execute instructions according to incoming message.

There are many possible interactions between task-based runtime systems: positive, by improving their cooperation; as well as negative, coming from the simultaneous execution of both computations and communications. Regardless the type of interaction, the separation between the runtime system and the communication library has to be clearly defined, and the interface of communication library allowing these interactions has to be generic enough, to be usable outside of the scope of task-based runtime systems. In the end, this prevents from integrating the communication library in the runtime system, and ease benchmarking and debugging of the communication library alone.

This thesis showed such software design is possible to make task-based runtime systems and communication libraries better work together, thanks to the abstraction offered by the runtime system and the specific interface of the communication library. It paves the way to many ideas of possible positive interactions!

Appendix A

Differences between MPI and NewMadeleine backends in StarPU

In its distributed extension, StarPU can use two different interfaces (*backends*): one using the MPI interface and another one using the NewMadeleine interface. The choice between these two backends is done before the compilation[1]. The MPI backend was implemented first, before the NewMadeleine one. The MPI backend can work with any library implementing the MPI standard: OpenMPI, Intel MPI, *etc*, but also NewMadeleine trough its MPI implementation called MadMPI (in this case, the MPI backend of StarPU can use only functions from the MPI standard, and not the interfaces specific to NewMadeleine). The NewMadeleine backend uses the native interface of NewMadeleine.

Since the working and the features offered by MPI libraries and NewMadeleine are different, this appendix explains the differences between the two StarPU backends for distributed computing.

MPI backend

With the MPI backend, each StarPU process (corresponding to an MPI process) launches a thread to manage communications, bound on a dedicated core. This thread executes an infinite loop which performs all required MPI calls and makes communication progress.

When, at the end of a task, a buffer written by the task has to be sent through the network (as defined in the application DAG), a communication request is issued by the core which executed the task. This request is inserted in a list of *ready requests*. This list is sorted by communication priorities.

When a reception from the network is detected in the DAG, the thread detecting it submits a reception request by inserting it in a list of *reception requests*.

[1] In the `./configure` step: by default, the MPI backend is selected if an MPI library is found; the NewMadeleine backend has to be explicitly enabled with `./configure --enable-nmad`.

The loop in the MPI thread of StarPU takes requests from lists of requests and calls the corresponding MPI function (*e.g.* `MPI_Isend` to send a data, `MPI_Irecv` for receptions). Each send operation is actually divided in two MPI communications: the first contains the *envelope* to tell the receiver process the properties of the incoming message (*e.g.* its tag and its size). When the receiver node acknowledged this envelope, the real data is sent with a non-blocking call. To avoid over-loading the MPI library, only a finite number of send operations are issued simultaneously (10, by default). The rest of the loop handles reception instructions: it posts a `MPI_Irecv` to receive envelope messages and then tests the status of posted communications with `MPI_Test`. Once a communication is terminated, StarPU releases the used data handle, which can unlock tasks. Finally, if an envelope message has been received, it is handled differently whether a matching receive request has already been submitted or not. In the first case, the data handle provided by the application to receive the data is used, otherwise a buffer is allocated to receive the data while the matching reception is not posted (*early request*).

The use of a unique and dedicated thread to perform MPI operations is to make sure communication progress (thanks to the `MPI_Test` calls in each loop iteration) and not having to use the `MPI_THREAD_MULTIPLE` threading support level, causing performance troubles to many MPI implementations. Moreover, two main missing features of the MPI standard are emulated in this communication backend: communication priorities by using sorted list of requests and communication completion notifications by polling regularly on communications to get their status.

NewMadeleine backend

Since NewMadeleine supports multithreaded applications, communication priorities and is designed with an event-driven paradigm, the NewMadeleine backend is much simpler than the MPI one (1525 lines of C code *vs* 3195).

The thread to make communications progress is managed by NewMadeleine (more precisely PIOMan); StarPU only indicates on which core this thread can be bounded. When a send communication has to be issued at the end of a task, the core which executed the task calls in the end directly NewMadeleine functions. The same goes for reception instructions: they can be issued from any StarPU's thread.

When NewMadeleine is notified of the end of a communication, a function containing StarPU code is executed by the progression thread to release the data handle used for the finished communication.

Communication priorities can be directly passed as parameters to NewMadeleine functions to submit the communication request, thus there is no need for priority management made by StarPU. The information sent in envelopes of the MPI backend can be sent as message headers with NewMadeleine: the receiver can read the headers of a message before actually receiving the message content, which can be convenient to take decisions according to the header content (*e.g.* the size of the incoming data and where to store it). There is no early requests in the NewMadeleine backend: network transfers are performed only when both send and receive instructions are posted.

Appendix B

Reproducible Experiments

Experiments presented in Chapters 3 and 6 were performed following a reproducible methodology.

The reproducibility of the software stack used for the experiments is assured thanks to the Guix software deployment tool. Guix is a functional package manager, with packages defined in the Guile Scheme language. Each package is installed in a dedicated directory, named after a cryptographic hash based on the definition of the package and its dependencies. This allows to install different versions (release version, but also other dependencies, for instance) of the same software without breaking the system.

With a software stack like the one used in this thesis, such feature is very welcome: it is very easy to create an isolated environment with customized packages. Guix proposes several so-called *package transformations* to change on-the-fly properties of a package: the commit or the branch of the source code of the software, replace a dependency by another, *etc.* Used with the isolated environments of Guix, these package transformations remove the burden of compiling the correct version of each software before each experiment.

To ensure reproducibility of experiments made with Guix, software versions have to be pinned and saved along with scripts to launch the experiments. The file describing pinned software versions can then be provided to Guix commands to use the specific version of Guix and have exactly the same packages as those available when the software versions were exported.

Guix provides many mainstream packages. The Guix-HPC effort[1]packages applications specific to the HPC environment and aims at easing the use of Guix in a HPC context. All packages directly used in this thesis (NewMadeleine, StarPU, Chameleon, FxT) are defined in the Guix-HPC *channel*.

The following command is an example of Guix usage:

[1] https://hpc.guix.info/

```
guix time-machine --channels=guix-channels.scm -- \
    shell --pure --preserve=^SLURM \
    chameleon slurm@19 \
    --with-input=openmpi=nmad \
    --with-input=openblas=mkl \
    --with-branch=starpu=nmad-coop-mcast -- \
    mpirun chameleon_stesting -o potrf --n 4800:40000:3200
```

It uses the versions described in the `guix-channels.scm` file, creates an isolated environment (`--pure`), but preserves environment variables starting with `SLURM` and populates the environment with the packages of CHAMELEON and the version 19 of SLURM. The dependency graph is modified to replace OPENMPI by NEWMADELEINE (OPENMPI is a dependency defined in the STARPU package, dependency of CHAMELEON), replace OPENBLAS by the INTEL MKL and use the branch `nmad-coop-mcast` of the GIT repository of STARPU used to get the sources. Once the environment is setup, `chameleon_stesting` is launched with `mpirun`, with software available in the GUIX environment only.

With the `guix-channels.scm` and this command line backed-up, one is sure to reproduce the same environment at any time.

Making the experimental scripts publicly available is another step to achieve a reproducible experiments. It requires to clearly organize experiments, describe their goals and workings, and ensure the maximum independence from cluster specificities (or document which changes are necessary to launch the experiments on another cluster). When the repository describing the experiments is completed, archiving it on SOFTWARE HERITAGE and providing the obtained ID to easily retrieve the scripts is effortless.

Scripts and instructions to reproduce experiments presented in Chapters 3 and 6 are available online and are archived on SOFTWARE HERITAGE[2].

[2]Chapter 3: https://gitlab.inria.fr/pswartva/paper-starpu-traces-r13y
Archived with the ID `swh:1:snp:e098c2012fa26baf67056e82f8d822dfcecc08cb`
Chapter 6: https://gitlab.inria.fr/pswartva/paper-model-memory-contention-r13y
Archived with the ID `swh:1:snp:306f7c10cf69a5860587e5aad62b76070b798ecd`

Appendix C

Machine Descriptions

Experiments made for this thesis were carried out on several machines, from different cluster centers, with different features. This appendix presents the characteristics of each type of machine.

The resource of the following computing clusters were used:

- the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr);
- the HPC resources of CINES under the allocations 2019-A0060601567, 2020-A0080601567, 2021-A0100601567 attributed by GENCI (*Grand Equipement National de Calcul Intensif*);
- the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr);
- the Dalton local experimental platform, supported by Inria.

billy

Platform : Dalton

Processor : dual AMD EPYC 7502 at 2.5 GHz with 64 cores, enabled hyperthreading

Memory : 128 GB of RAM, with 2 NUMA nodes

Network : InfiniBand ConnectX-6 HDR

Number of nodes : 2

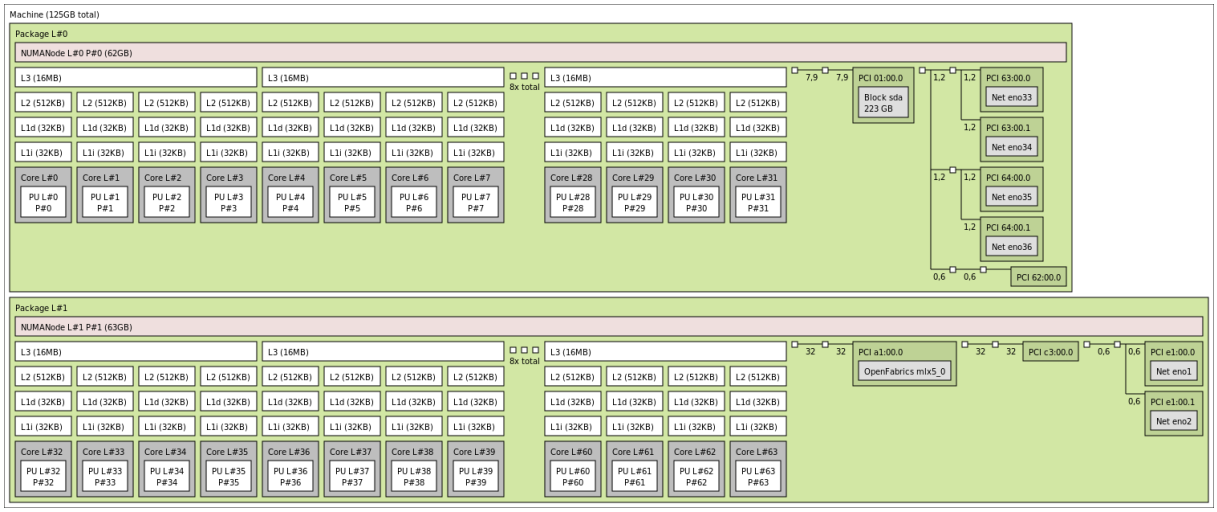


Figure C.1: `billy`'s topology.

`bora`

Platform : PlaFRIM

Processor : dual Intel Xeon Gold 6240 at 2.6 GHz with 36 cores, disabled hyperthreading

Memory : 192 GB RAM, with 2 NUMA nodes

Network : Omni-Path HFI Silicon 100 series

Number of nodes : 44

`dahu`

Platform : Grid'5000

Processor : dual Intel Xeon Gold 6130 at 2.1 GHz with 32 cores, enabled hyperthreading

Memory : 192 GB of RAM, with 2 NUMA nodes

Network : Omni-Path HFI Silicon 100 Series

Number of nodes : 32

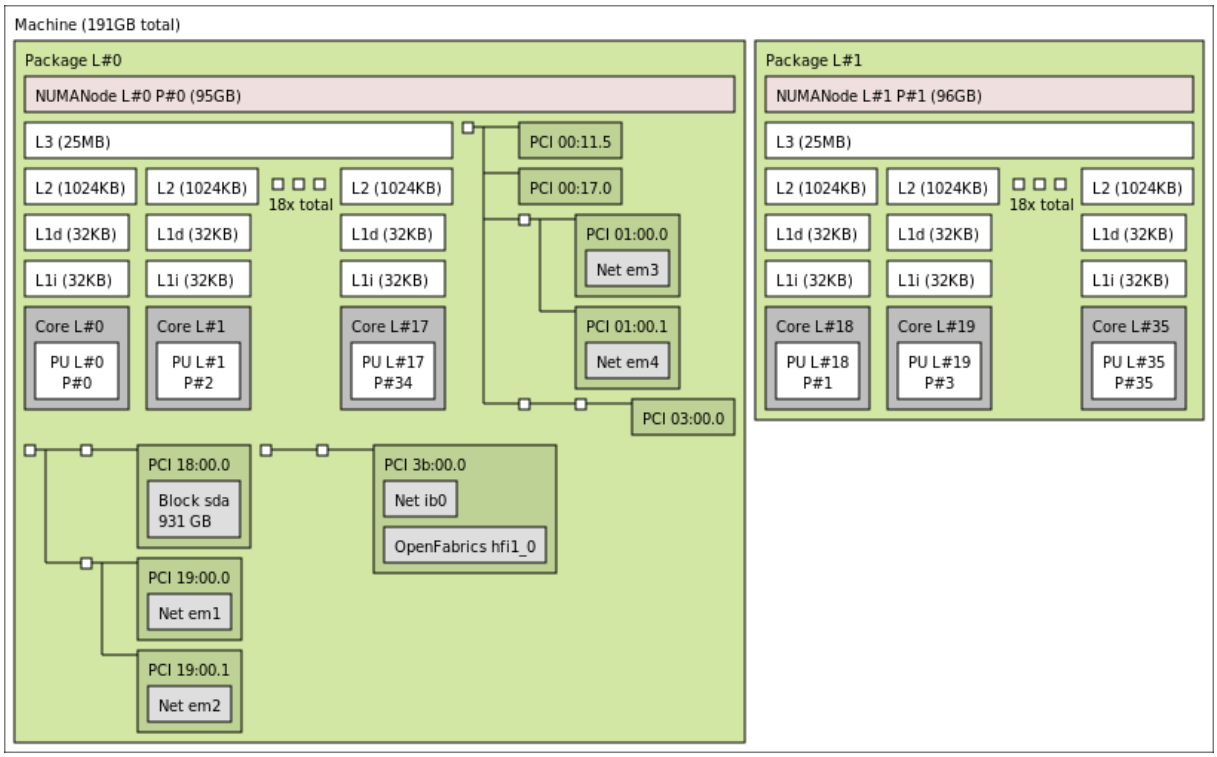


Figure C.2: `bora`'s topology.

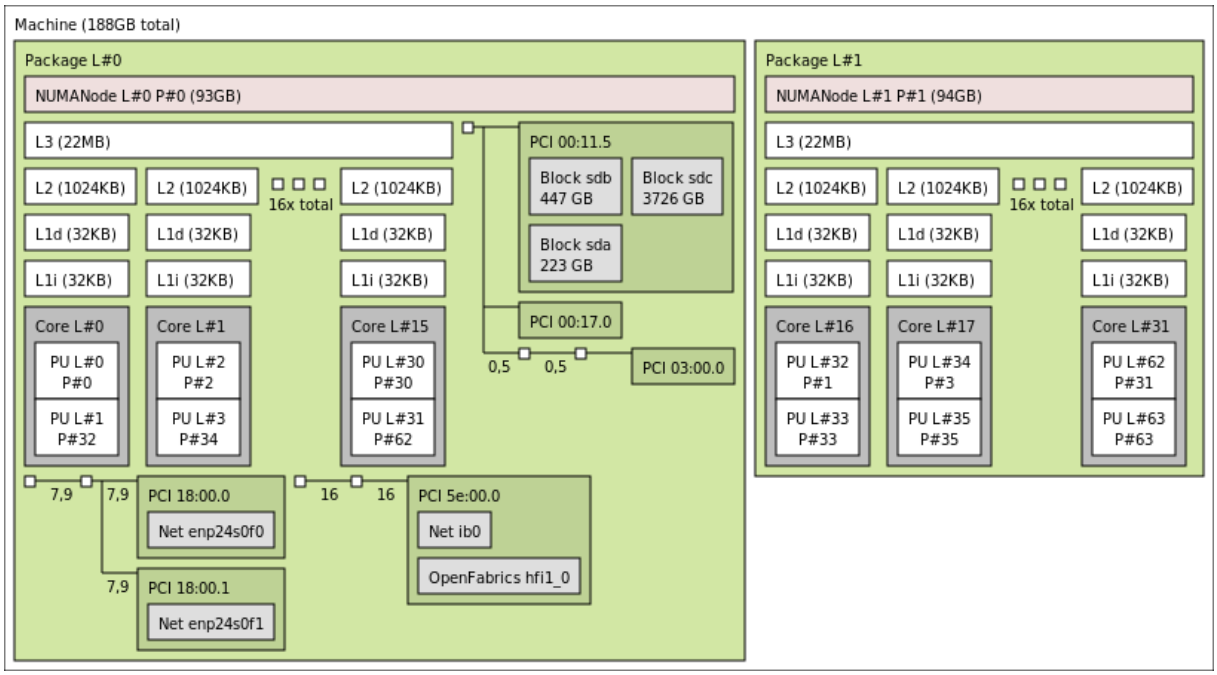


Figure C.3: `dahu`'s topology.

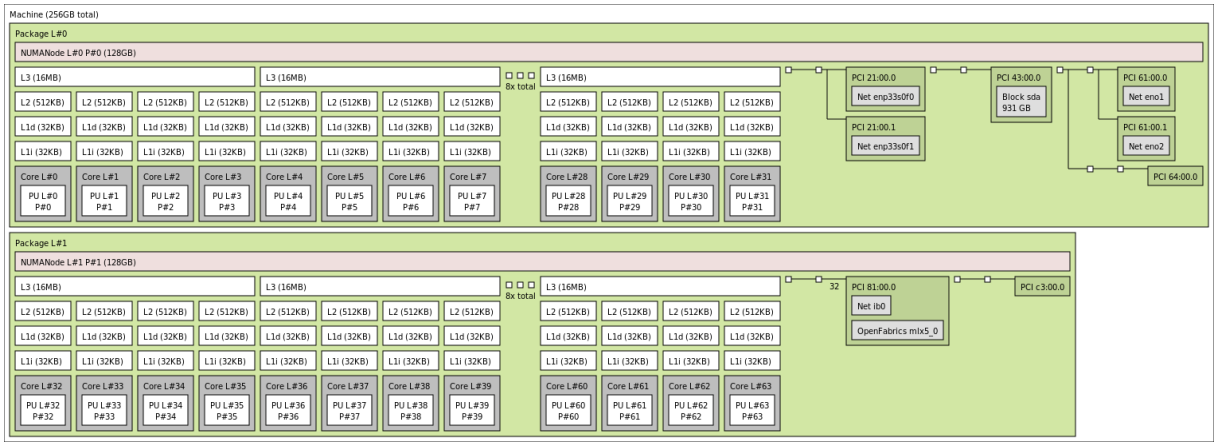


Figure C.4: `diablo`'s topology.

`diablo`

Platform : PlaFRIM

Processor : dual AMD EPYC 7452 at 2.35 GHz with 64 cores, disabled hyperthreading

Memory : 256 GB of RAM, with 2 NUMA nodes

Network : InfiniBand ConnectX-6 HDR

Number of nodes : 4

`grvingt`

Platform : Grid'5000

Processor : dual Intel Xeon Gold 6130 at 2.1 GHz with 32 cores, enabled hyperthreading

Memory : 192 GB of RAM, with 2 NUMA nodes

Network : Omni-Path HFI Silicon 100 Series

Number of nodes : 64

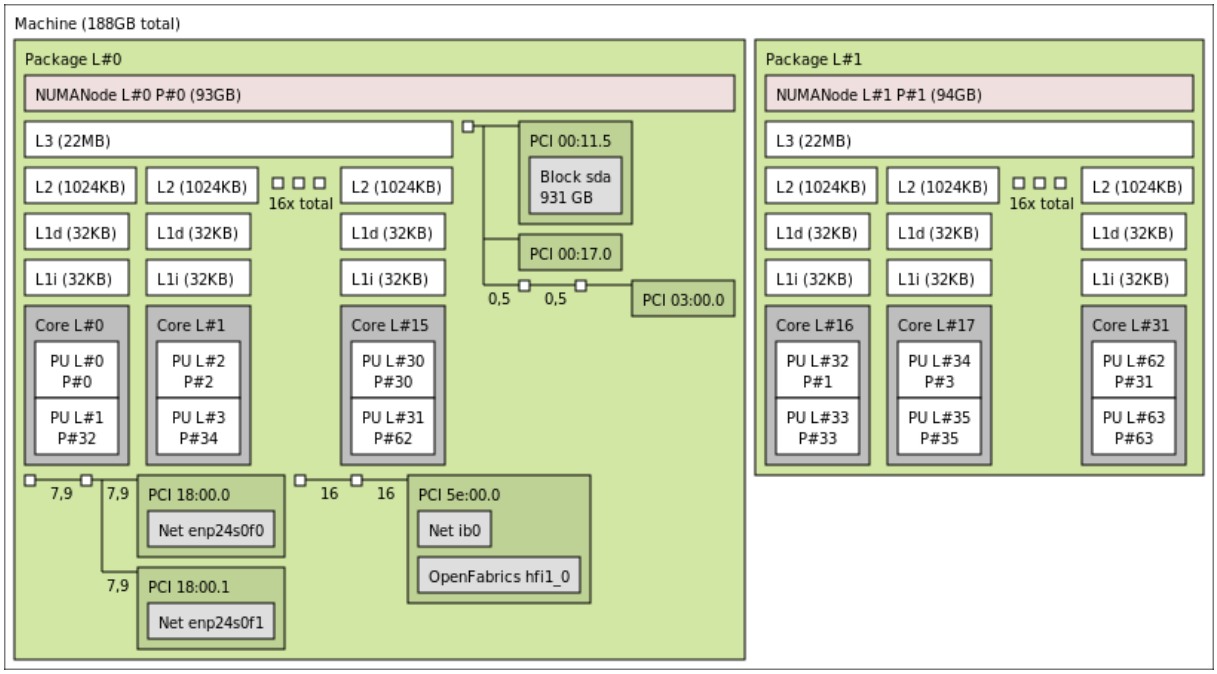


Figure C.5: `grvingt`'s topology.

`henri`

Platform : Dalton

Processor : dual Intel Xeon Gold 6140 at 2.3 GHz with 36 cores, disabled hyperthreading

Memory : 96 GB of RAM, with 2 NUMA nodes

Network : InfiniBand ConnectX-4 EDR

Number of nodes : 2

`henrisubnuma`

`henri` nodes reconfigured with sub-NUMA clustering: they feature 4 NUMA nodes instead of 2.

`occigen`

Platform : Dalton

Processor : dual Intel Xeon E5 2690v4 at 2.6 GHz with 28 cores, enabled hyperthreading

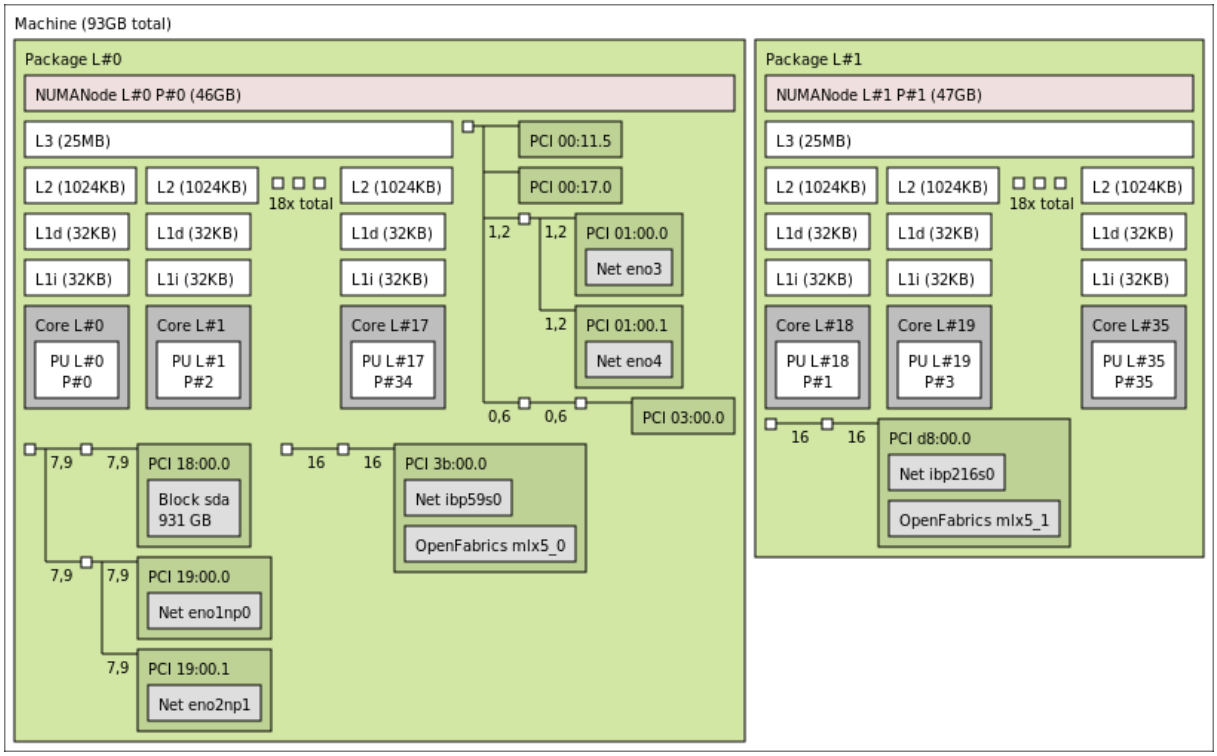


Figure C.6: `henri`'s topology.

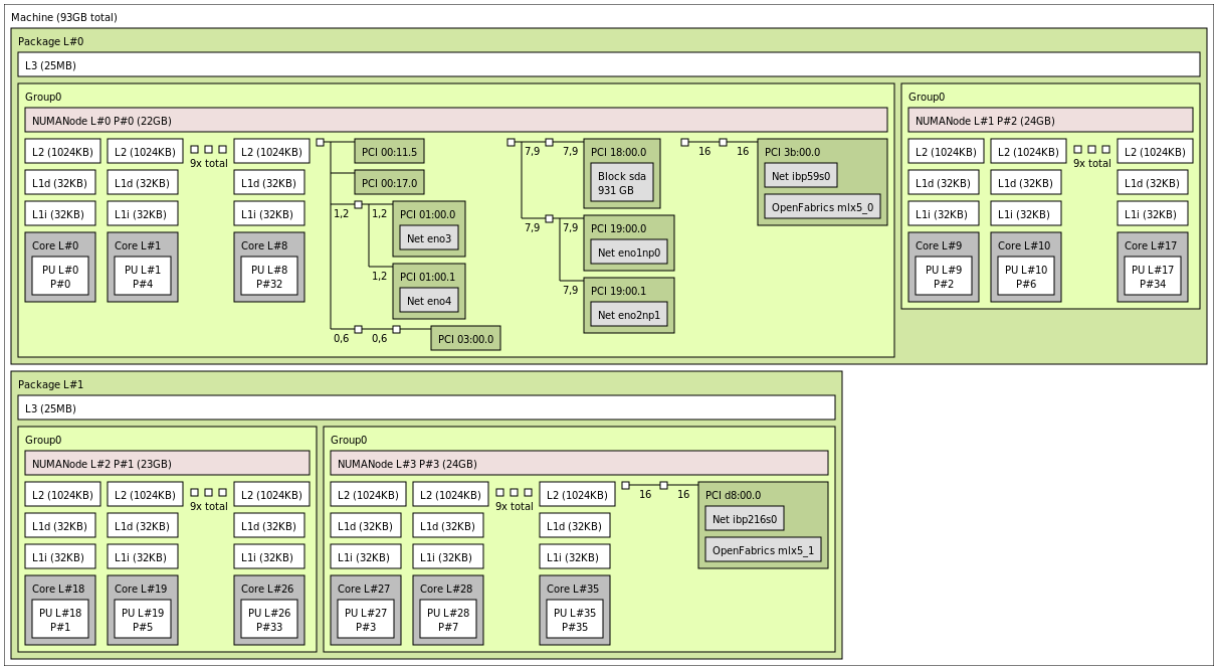


Figure C.7: `henrisubnuma`'s topology.

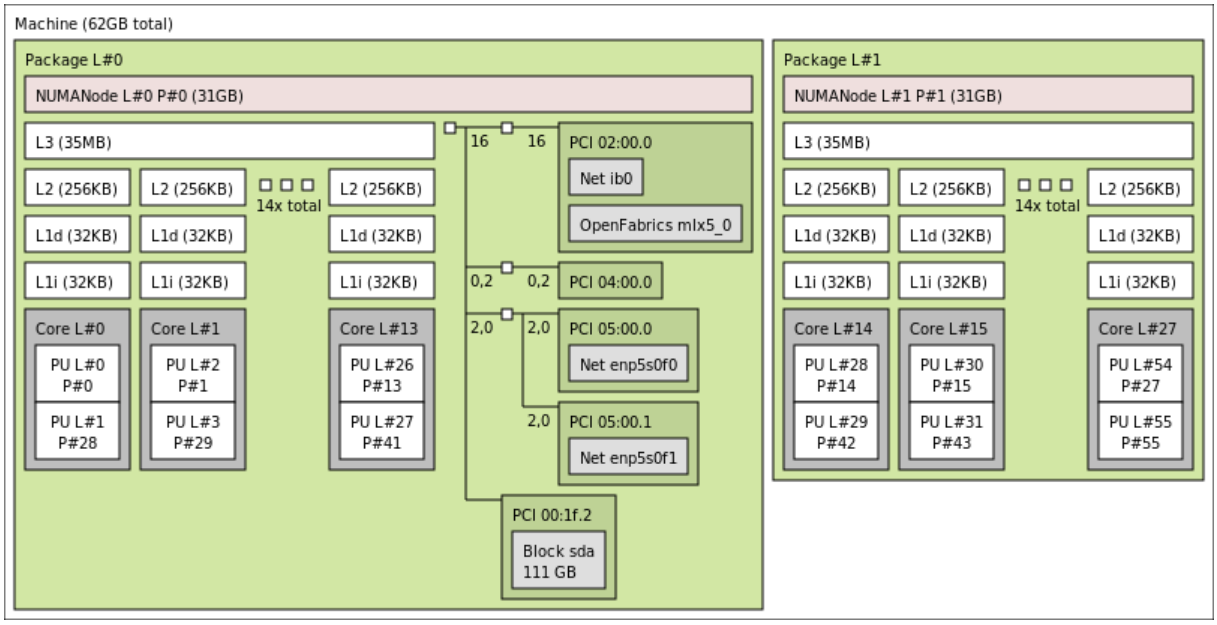


Figure C.8: `occigen`'s topology.

Memory : 64 GB of RAM, with 2 NUMA nodes

Network : InfiniBand Connect-IB FDR

Number of nodes : 1260

`peabody`

Platform : Dalton

Processor : dual Intel Xeon Gold 6230R at 2.1 GHz with 52 cores, enabled hyperthreading

Memory : 32 GB RAM, with 2 NUMA nodes

Number of nodes : 1

`pyxis`

Platform : Grid'5000

Processor : dual Cavium-ARM ThunderX2 99xx at 2.5 GHz with 32 cores, enabled hyperthreading

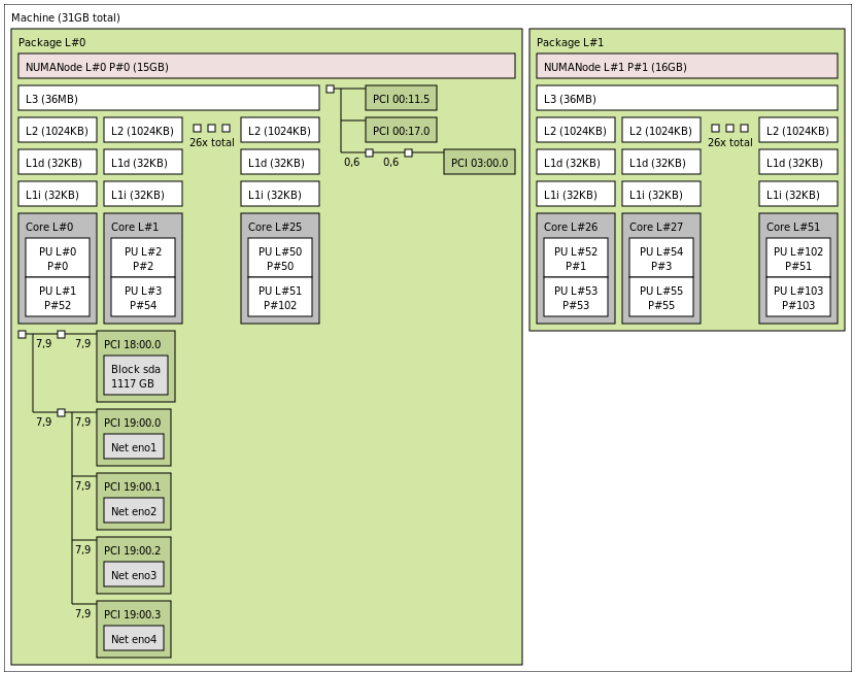


Figure C.9: `peabody`'s topology.

Memory : 256 GB of RAM, with 2 NUMA nodes

Network : InfiniBand ConnectX-6 EDR

Number of nodes : 4

`zonda`

Platform : PlaFRIM

Processor : dual AMD EPYC 7452 at 2.35 GHz with 64 cores, disabled hyperthreading

Memory : 256 GB RAM, with 2 NUMA nodes

Network : 10 Gb/s Ethernet

Number of nodes : 21

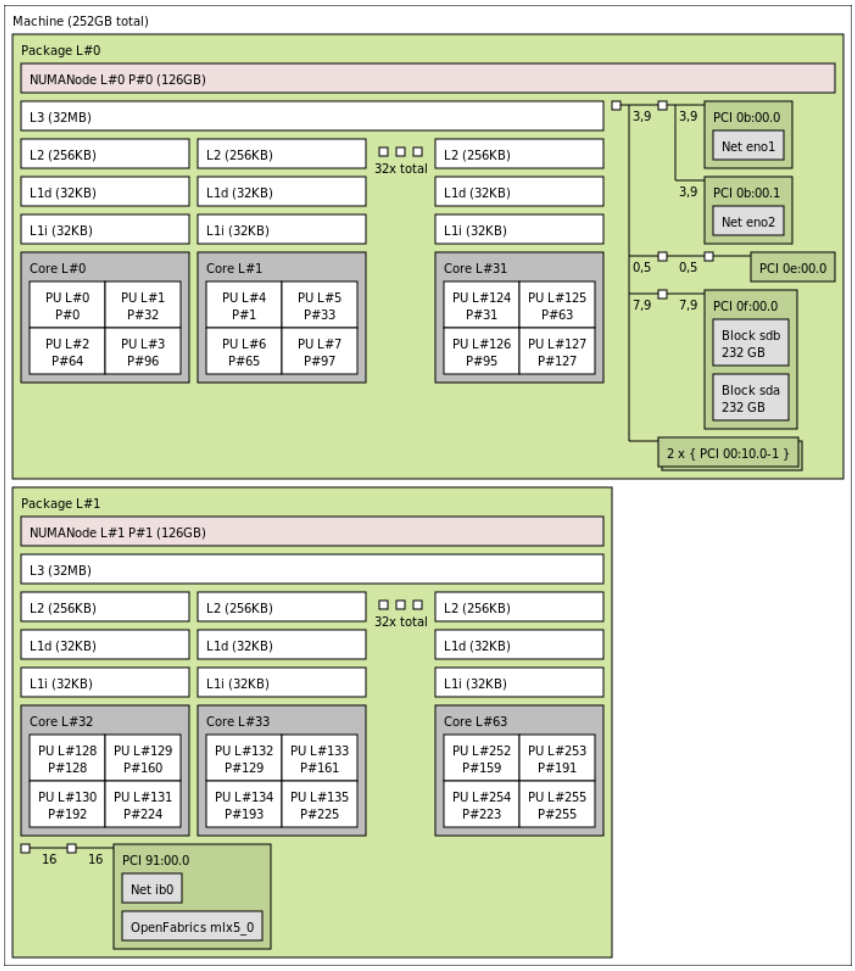


Figure C.10: `pyxis`'s topology.

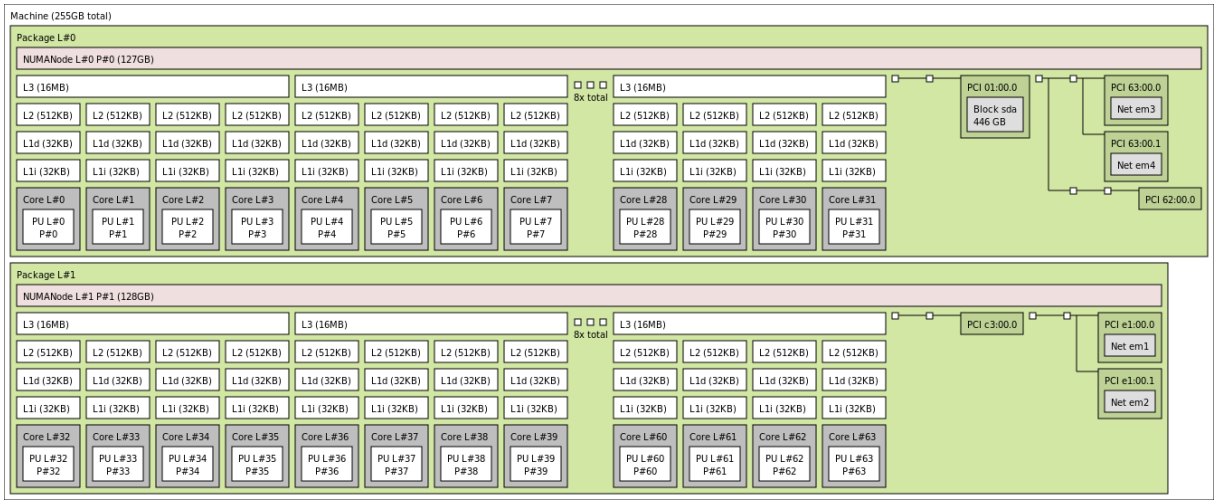


Figure C.11: `zonda`'s topology.

Appendix D

Algorithms to Model Memory Contention between Computations and Communications

The following algorithms are the implementation of equations presented in Chapter 6, to model the memory bandwidth share between computations and communications.

Algorithm 3 Predict total memory bandwidth (implementation of equation 6.1)

Inputs: N_{par}^{max}, T_{par}^{max}, N_{seq}^{max}, N_{par}^{max}, T_{par}^{max2}, δ_l, δ_r
Output: $\mathcal{T}$

1: **function** PREDICTTOTAL
2: $\mathcal{T} \leftarrow []$
3: **for** $i = 1$ **to** $number\_cores$ **do**
4: **if** $i \leq N_{par}^{max}$ **then**
5: $\mathcal{T}[i] \leftarrow T_{par}^{max}$
6: **else if** $i \leq N_{seq}^{max}$ **then**
7: $\mathcal{T}[i] \leftarrow T_{par}^{max} - \delta_l \times (i - N_{par}^{max})$
8: **else**
9: $\mathcal{T}[i] \leftarrow T_{par}^{max2} - \delta_r \times (i - N_{seq}^{max})$
10: **end if**
11: **end for**
12: **end function**

Algorithm 4 Compute memory bandwidths available for computations and communications (implementation of equations 6.2, 6.3, 6.4, 6.5 and 6.6)

Inputs: $\mathcal{T}$, B_{seq}^{comp}, T_{seq}^{max}, N_{seq}^{max}, N_{par}^{max}, B_{seq}^{comm}, α
Outputs: $\mathcal{B}_{par}^{comp}$, $\mathcal{B}_{par}^{comm}$, $\mathcal{B}_{seq}^{comp}$

function PREDICTBANDWIDTHS
  $\mathcal{B}_{par}^{comm} \leftarrow []$
  $\mathcal{B}_{par}^{comp} \leftarrow []$
  $\mathcal{B}_{seq}^{comp} \leftarrow []$
  $\beta_v \leftarrow None$
  $\beta_i \leftarrow None$
  for $i = 1$ **to** $number\_cores$ **do**
    $\mathcal{B}_{seq}^{comp}[i] \leftarrow \min(B_{seq}^{comp} \times i, \mathcal{T}[i], T_{seq}^{max})$ ▷ Equation 6.6
    if $i \times B_{seq}^{comp} + \alpha \times B_{seq}^{comm} < \mathcal{T}[i]$ **then** ▷ Equation 6.2
      $\mathcal{B}_{par}^{comp}[i] \leftarrow i \times B_{seq}^{comp}$ ▷ Equation 6.3
      $\mathcal{B}_{par}^{comm}[i] \leftarrow \min(\mathcal{T}[i] - \mathcal{B}_{par}^{comp}[i], B_{seq}^{comm})$ ▷ Equation 6.4
      $\beta_v \leftarrow \mathcal{B}_{par}^{comm}[i]/B_{seq}^{comm}$
      $\beta_i \leftarrow i$
    else
      $\alpha_i \leftarrow \alpha$ ▷ Equation 6.5
      if $(N_{seq}^{max} - N_{par}^{max}) > 1$ **and** $i < N_{seq}^{max}$ **then**
        $\delta_c \leftarrow (\beta_v - \alpha)/(N_{seq}^{max} - \beta_i)$
        $\alpha_i \leftarrow \beta_v - \delta_c \times (i - \beta_i)$
      end if
      $\mathcal{B}_{par}^{comm}[i] \leftarrow \alpha_i \times B_{seq}^{comm}$ ▷ Equation 6.4
      $\mathcal{B}_{par}^{comp}[i] \leftarrow \mathcal{T}[i] - \mathcal{B}_{par}^{comm}[i]$ ▷ Equation 6.3
    end if
  end for
end function

Algorithm 5 Predict communication performance according to memory placements (implementation of equation 6.7)

Inputs: m_{comp}, m_{comm}, $\mathcal{M}_{local}$, $\mathcal{M}_{remote}$
Outputs: $\mathcal{B}_{par}^{comm}$

function GetCommBandwidths
  if $m_{comp} == m_{comm}$ **and** $m_{comp} \geq \#m$ **then**
    $\mathcal{B}_{par}^{comp}, \mathcal{B}_{par}^{comm}, \mathcal{B}_{seq}^{comp} \leftarrow$PredictBandwidths($\mathcal{M}_{remote}$)
    return $\mathcal{B}_{par}^{comm}$
  else
    if $m_{comm} \geq \#m$ **then**
      $\mathcal{B}_{par}^{comp}, \mathcal{B}_{par}^{comm}, \mathcal{B}_{seq}^{comp} \leftarrow$PredictBandwidths(
        $\mathcal{M}_{local}$,
        $B_{seq}^{comm} \leftarrow B_{seq}^{comm}(\mathcal{M}_{remote})$ ▷ Use B_{seq}^{comm} from $\mathcal{M}_{remote}$ in the function
)
      return $\mathcal{B}_{par}^{comm}$
    else
      $\mathcal{B}_{par}^{comp}, \mathcal{B}_{par}^{comm}, \mathcal{B}_{seq}^{comp} \leftarrow$PredictBandwidths($\mathcal{M}_{local}$)
      return $\mathcal{B}_{par}^{comm}$
    end if
  end if
end function

Algorithm 6 Predict computation performance according to memory placements (implementation of equation 6.8)

Inputs: m_{comp}, m_{comm}, $\mathcal{M}_{local}$, $\mathcal{M}_{remote}$
Outputs: $\mathcal{B}_{par}^{comp}$

function GetCompBandwidths
  if $m_{comp} < \#m$ **then**
    $\mathcal{B}_{par}^{comp}, \mathcal{B}_{par}^{comm}, \mathcal{B}_{seq}^{comp} \leftarrow$PredictBandwidths($\mathcal{M}_{local}$)
    if $m_{comp} == m_{comm}$ **then**
      return $\mathcal{B}_{par}^{comp}$
    else
      return $\mathcal{B}_{seq}^{comp}$
    end if
  else
    $\mathcal{B}_{par}^{comp}, \mathcal{B}_{par}^{comm}, \mathcal{B}_{seq}^{comp} \leftarrow$PredictBandwidths($\mathcal{M}_{remote}$)
    if $m_{comp} == m_{comm}$ **then**
      return $\mathcal{B}_{par}^{comp}$
    else
      return $\mathcal{B}_{seq}^{comp}$
    end if
  end if
end function

Appendix E

Parameter Values of Contention Model

The following tables contain the parameter values of the models instantiated in Chapter 6.

Parameter		$\mathcal{M}_{local}$	$\mathcal{M}_{remote}$
N_{par}^{max}	(number of computing cores)	32	32
T_{par}^{max}	(MB/s)	95555.5	75276.9
N_{seq}^{max}	(number of computing cores)	32	32
T_{seq}^{max}	(MB/s)	84420.4	76466.4
T_{par}^{max2}	(MB/s)	95555.5	75276.9
α		0.842	0.593
δ_l	(MB/s/core)	0.0	0.0
δ_r	(MB/s/core)	0.0	0.0
B_{seq}^{comp}	(MB/s)	2808.8	2746.5
B_{seq}^{comm}	(MB/s)	12793.0	18898.9

Table E.1: Parameter values for executions on `billy` nodes (Figure 6.5).

Parameter		$\mathcal{M}_{local}$	$\mathcal{M}_{remote}$
N_{par}^{max}	(number of computing cores)	18	8
T_{par}^{max}	(MB/s)	83849.6	32315.3
N_{seq}^{max}	(number of computing cores)	18	8
T_{seq}^{max}	(MB/s)	80107.7	32733.3
T_{par}^{max2}	(MB/s)	83849.6	32315.3
α		0.951	0.936
δ_l	(MB/s/core)	0.0	0.0
δ_r	(MB/s/core)	0.0	1.6
B_{seq}^{comp}	(MB/s)	4489.2	4487.6
B_{seq}^{comm}	(MB/s)	9948.4	8784.7

Table E.2: Parameter values for executions on `bora` nodes (Figure 6.8).

Parameter		$\mathcal{M}_{local}$	$\mathcal{M}_{remote}$
N_{par}^{max}	(number of computing cores)	11	5
T_{par}^{max}	(MB/s)	72147.6	32102.5
N_{seq}^{max}	(number of computing cores)	14	5
T_{seq}^{max}	(MB/s)	70072.9	32677.4
T_{par}^{max2}	(MB/s)	71509.2	32102.5
α		0.959	0.949
δ_l	(MB/s/core)	212.8	0.0
δ_r	(MB/s/core)	656.8	-38.4
B_{seq}^{comp}	(MB/s)	6656.5	7171.3
B_{seq}^{comm}	(MB/s)	11341.2	10607.0

Table E.3: Parameter values for executions on `dahu` nodes (Figure 6.9).

Parameter		$\mathcal{M}_{local}$	$\mathcal{M}_{remote}$
N_{par}^{max}	(number of computing cores)	32	32
T_{par}^{max}	(MB/s)	81846.8	87081.5
N_{seq}^{max}	(number of computing cores)	32	32
T_{seq}^{max}	(MB/s)	70092.1	67694.6
T_{par}^{max2}	(MB/s)	81846.8	87081.5
α		0.967	0.908
δ_l	(MB/s/core)	0.0	0.0
δ_r	(MB/s/core)	0.0	0.0
B_{seq}^{comp}	(MB/s)	2221.1	2210.2
B_{seq}^{comm}	(MB/s)	12139.7	22394.3

Table E.4: Parameter values for executions on `diablo` nodes (Figure 6.4)

Parameter		$\mathcal{M}_{local}$	$\mathcal{M}_{remote}$
N_{par}^{max}	(number of computing cores)	14	5
T_{par}^{max}	(MB/s)	75015.2	31928.2
N_{seq}^{max}	(number of computing cores)	15	5
T_{seq}^{max}	(MB/s)	73818.4	32576.8
T_{par}^{max2}	(MB/s)	74398.4	31928.2
α		0.953	0.971
δ_l	(MB/s/core)	616.8	0.0
δ_r	(MB/s/core)	649.2	-10.6
B_{seq}^{comp}	(MB/s)	6698.7	7178.5
B_{seq}^{comm}	(MB/s)	9628.8	9345.7

Table E.5: Parameter values for executions on `grvingt` nodes (Figure 6.10).

Parameter		$\mathcal{M}_{local}$	$\mathcal{M}_{remote}$
N_{par}^{max}	(number of computing cores)	17	5
T_{par}^{max}	(MB/s)	73423.0	31629.7
N_{seq}^{max}	(number of computing cores)	18	7
T_{seq}^{max}	(MB/s)	72589.9	29130.7
T_{par}^{max2}	(MB/s)	73387.7	31278.1
α		0.915	0.761
δ_l	(MB/s/core)	35.3	175.8
δ_r	(MB/s/core)	0.0	119.8
B_{seq}^{comp}	(MB/s)	4455.4	4455.2
B_{seq}^{comm}	(MB/s)	11481.1	11459.6

Table E.6: Parameter values for executions on `henri` nodes (Figure 6.2)

Parameter		$\mathcal{M}_{local}$	$\mathcal{M}_{remote}$
N_{par}^{max}	(number of computing cores)	8	11
T_{par}^{max}	(MB/s)	42487.7	16936.1
N_{seq}^{max}	(number of computing cores)	11	4
T_{seq}^{max}	(MB/s)	43655.6	14726.2
T_{par}^{max2}	(MB/s)	39718.9	15217.8
α		0.853	0.270
δ_l	(MB/s/core)	922.9	859.1
δ_r	(MB/s/core)	191.7	103.3
B_{seq}^{comp}	(MB/s)	4456.4	4455.4
B_{seq}^{comm}	(MB/s)	11450.4	11410.0

Table E.7: Parameter values for executions on `henrisubnuma` nodes (Figure 6.3)

Parameter		$\mathcal{M}_{local}$	$\mathcal{M}_{remote}$
N_{par}^{max}	(number of computing cores)	14	7
T_{par}^{max}	(MB/s)	53948.2	24706.4
N_{seq}^{max}	(number of computing cores)	14	7
T_{seq}^{max}	(MB/s)	47817.2	21137.3
T_{par}^{max2}	(MB/s)	53948.2	24706.4
α		1.000	1.000
δ_l	(MB/s/core)	0.0	0.0
δ_r	(MB/s/core)	0.0	39.9
B_{seq}^{comp}	(MB/s)	3417.6	3417.8
B_{seq}^{comm}	(MB/s)	6220.0	6219.5

Table E.8: Parameter values for executions on `occigen` nodes (Figure 6.6).

Parameter		$\mathcal{M}_{local}$	$\mathcal{M}_{remote}$
N_{par}^{max}	(number of computing cores)	24	25
T_{par}^{max}	(MB/s)	64626.9	36737.2
N_{seq}^{max}	(number of computing cores)	26	31
T_{seq}^{max}	(MB/s)	62462.7	32649.8
T_{par}^{max2}	(MB/s)	64566.2	36696.7
α		0.990	0.945
δ_l	(MB/s/core)	30.3	6.8
δ_r	(MB/s/core)	56.9	-8.4
B_{seq}^{comp}	(MB/s)	3211.5	3030.8
B_{seq}^{comm}	(MB/s)	4958.6	4390.1

Table E.9: Parameter values for executions on `pyxis` nodes (Figure 6.7).

Acronyms

ACPI Advanced Configuration and Power Interface. 67

ART Always Running Timer. 67

AVX Advanced Vector Extensions. 14, 104

BSP Bulk Synchronous Parallel. 28, 33, 51, 75

DAG Directed Acyclic Graph. 23, 24, 27, 28, 37, 39, 41, 42, 51, 73, 75–77, 96, 97, 139, 140, 142, 143, 147

DMA Direct Memory Access. 101, 108

DRAM Dynamic Random Access Memory. 16

DVFS Dynamic Voltage and Frequency Scaling. 43

FFT Fast Fourier Transform. 145

HBM High Bandwidth Memory. 16, 18

HPC High Performance Computing. i, ii, 5, 9, 10, 13, 15–17, 19–22, 29, 33, 40, 46, 47, 99, 106, 116, 137, 139, 149, 151

HPET High Precision Event Timer. 67, 68

I/O Inputs/Outputs. 57, 108

IPC Instructions Per Cycle. 13

MCT Minimum Completion Time. 25

MPI Message Passing Interface. 21, 22, 27–30, 35, 36, 38–42, 44, 46, 47, 63, 66, 72, 75, 76, 82, 83, 88–90, 93, 99–101, 109, 112, 115, 116, 140, 145, 147, 148

NIC Network Interface Card. 44, 106–110, 112, 113, 116, 126, 131, 143, 144

NTP Network Time Protocol. 46, 63, 68

NUMA Non-Uniform Memory Access. 16–18, 21, 25, 26, 35, 44, 89, 106–108, 112–114, 116, 119–122, 126–131, 133, 136–138, 144, 151, 152, 154, 155, 157, 158, 185

NVDIMM Non-Volatile Dual Inline Memory Module. 16, 18

OS Operating System. 20, 46

PCIe Peripheral Component Interconnect Express. 18, 20, 120

PGAS Partitioned Global Address Space. 36, 38–40

PTG Parametrized Task Graph. 37, 42

RAM Random Access Memory. 15, 18, 20, 121, 142, 143

RDMA Remote Direct Memory Access. 43

RMA Remote Memory Access. 36

RoCE RDMA over Converged Ethernet. 20

RPC Remote Procedure Call. 22, 80

RTC Real Time Clock. 67

SIMD Single Instruction on Multiple Data. 14

SMP Symmetric Multiprocessing. 120

STF Sequential Task Flow. 23, 37, 39, 41, 75, 140, 144

TSC Time Stamp Counter. 67, 68

References

[1] Extrae. https://tools.bsc.es/extrae. Accessed: 2021-10-01.

[2] Intel MPI. https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html. Accessed: 2022-06-22.

[3] MPICH: High-Performance Portable MPI. https://www.mpich.org/. Accessed: 2022-06-22.

[4] MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. http://mvapich.cse.ohio-state.edu/. Accessed: 2022-06-22.

[5] OpenACC. https://www.openacc.org. Accessed: 2022-07-13.

[6] OpenCL Overview. https://www.khronos.org/opencl/. Accessed: 2022-07-13.

[7] SYCL Overview. https://www.khronos.org/sycl/. Accessed: 2022-07-13.

[8] Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305, March 1992.

[9] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, et al. Parallel programming with migratable objects: Charm++ in practice. In SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 647–658. IEEE, 2014.

[10] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, GPU Computing Gems, volume 2. Morgan Kaufmann, September 2010.

[11] Emmanuel Agullo, Olivier Aumage, Bérenger Bramas, Olivier Coulaud, and Samuel Pitoiset. Bridging the gap between OpenMP and task-based runtime systems for the fast multipole method. IEEE Transactions on Parallel and Distributed Systems, page 14, April 2017.

[12] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. IEEE Transactions on Parallel and Distributed Systems, 2017.

[13] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Julien Herrmann, and Antoine Jego. Task-Based Parallel Programming for Scalable Algorithms: application to Matrix Multiplication. Research Report RR-9461, Inria Bordeaux - Sud-Ouest, February 2022.

[14] Diego Andrade, Basilio B. Fraguela, and Ramón Doallo. Accurate prediction of the behavior of multithreaded applications in shared caches. Parallel Comput., 39(1):36–57, jan 2013.

[15] Asim YarKhan. Dynamic Task Execution on Shared and Distributed Memory Architectures. PhD thesis, 2012.

[16] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009, 23:187–198, February 2011.

[17] Olivier Aumage, Elisabeth Brunet, Nathalie Furmento, and Raymond Namyst. NewMadeleine: a Fast Communication Scheduling Engine for High Performance Networks. In Workshop on Communication Architecture for Clusters (CAC 2007), Long Beach, California, United States, March 2007.

[18] Pavan Balaji, Hemal Shah, and D.K. Panda. Sockets vs RDMA Interface over 10Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck. 01 2004.

[19] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pages 1–11, Nov 2012.

[20] Michael Bauer. Legion: Programming Distributed Heterogeneous Architectures with Logical Reg PhD thesis, 2014.

[21] Guillaume Beauchamp. Portage de StarPU sur la bibliothèque de communication NewMadeleine. Master's thesis, Universite Bordeaux, September 2017.

[22] Daniel Becker, John C. Linford, Rolf Rabenseifner, and Felix Wolf. Replay-based synchronization of timestamps in event traces of massively parallel applications. In Wu-chi Feng, editor, 2008 International Conference on Parallel Processing - Workshops, pages 212–219, 2008.

[23] Daniel Becker, Rolf Rabenseifner, and Felix Wolf. Timestamp synchronization for event traces of large-scale message-passing applications. In Franck Cappello, Thomas Herault, and Jack Dongarra, editors, Recent Advances in Parallel Virtual

Machine and Message Passing Interface, pages 315–325, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[24] Daniel Becker, Rolf Rabenseifner, and Felix Wolf. Implications of non-constant clock drifts for the timestamps of concurrent events. In Yutaka Ishikawa, editor, 2008 IEEE International Conference on Cluster Computing, pages 59–68, Tsukuba, Japan, 2008.

[25] George Bosilca, Aurélien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack Dongarra. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. Computing in Science and Engineering, 15(6):36–45, November 2013.

[26] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing. Parallel Computing, 38(1):37–51, 2012. Extensions for Next-Generation Parallel Programming Models.

[27] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Pisa, Italy, February 2010.

[28] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M. Badia, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Productive Programming of GPU Clusters with OmpSs. In 2012 IEEE 26th International Parallel and Distributed Processing Symposium, pages 557–568, 2012.

[29] Qinglei Cao, Yu Pei, Thomas Herault, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Performance Analysis of Tile Low-Rank Cholesky Factorization Using PaRSEC Instrumentation Tools. In 2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools), pages 25–32, 2019.

[30] Emilio Castillo, Nikhil Jain, Marc Casas, Miquel Moreto, Martin Schulz, Ramon Beivide, Mateo Valero, and Abhinav Bhatele. Optimizing computation-communication overlap in asynchronous task-based programs. In Proceedings of the ACM International Conference on Supercomputing, ICS '19, page 380–391, New York, NY, USA, 2019. Association for Computing Machinery.

[31] L. Chai, Q. Gao, and D. K. Panda. Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System. In Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07), pages 471–478, 2007.

[32] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. The International Journal of High Performance Computing Applications, 21(3):291–312, 2007.

[33] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In 11th International Symposium on High-Performance Computer Architecture, pages 340–351, 2005.

[34] Sanjay Chatterjee, Sagnak Tasırlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. Integrating Asynchronous Task Parallelism with MPI. In 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pages 712–725, 2013.

[35] Younghyun Cho, Surim Oh, and Bernhard Egger. Performance modeling of parallel loops on multi-socket platforms using queueing systems. IEEE Transactions on Parallel and Distributed Systems, 31(2):318–331, 2020.

[36] Eric Clément and Michel Dagenais. Traces synchronization in distributed networks. Journal of Computer Systems, Networks, and Communications, 2009, 2009.

[37] M. Cosnard and M. Loi. Automatic task graph generation techniques. In Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences, volume 2, pages 113–122, 1995.

[38] Kevin Coulomb, Augustin Degomme, Mathieu Faverge, and François Trahay. An open source tool chain for performance analysis. In Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch, editors, 5th Parallel Tools Workshop, pages 37–48, Dresden, Germany, September 2011. Springer.

[39] Flaviu Cristian. Probabilistic clock synchronization. Distributed computing, 3(3):146–158, 1989.

[40] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. SIGPLAN Not., 28(7):1–12, July 1993.

[41] Hoang-Vu Dang, Roshan Dathathri, Gurbinder Gill, Alex Brooks, Nikoli Dryden, Andrew Lenharth, Loc Hoang, Keshav Pingali, and Marc Snir. A lightweight communication runtime for distributed graph analytics. In 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 980–989, 2018.

[42] Vincent Danjean, Raymond Namyst, and Pierre-André Wacrenier. An efficient multi-level trace toolkit for multi-threaded applications. In José C. Cunha and Pedro D. Medeiros, editors, Euro-Par 2005 Parallel Processing, pages 166–175, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[43] Benhur de Oliveira Stein and Jacques Chassin de Kergommeaux. Pajé trace file format. online, March 2003.

[44] Alexandre Denis. pioman: a pthread-based Multithreaded Communication Engine. In Euromicro International Conference on Parallel, Distributed and Network-based Processing, Turku, Finland, March 2015.

[45] Jens Doleschal, Andreas Knüpfer, Matthias S. Müller, and Wolfgang E. Nagel. Internal timer synchronization for parallel event tracing. In Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 202–209, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[46] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters, 21(02):173–193, 2011.

[47] Tarek El-Ghazawi and François Cantonnet. UPC performance and potential: A NPB experimental study. In SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, pages 17–17. IEEE, 2002.

[48] Loretta Ellwood and Michael Heath. A Tracing Environment for MPI. 07 1995.

[49] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In Koen De Bosschere, Erik H. D'Hollander, Gerhard R. Joubert, David A. Padua, Frans J. Peters, and Mark Sawyer, editors, Applications, Tools and Techniques on the Road to Exascale Computing, Proceedings of the conference ParCo 2011, 31 August - 3 September 2011, Ghent, Belgium, volume 22 of Advances in Parallel Computing, pages 481–490. IOS Press, 2011.

[50] Mathieu Faverge, Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Raymond Namyst, Samuel Thibault, and Pierre-André Wacrenier. Programming Heterogeneous Architectures Using Hierarchical Tasks. Research Report RR-9466, Inria Bordeaux Sud-Ouest, March 2022.

[51] MPI Forum. MPI: A Message-Passing Interface Standard. Technical report, USA, 1994.

[52] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In Proceedings, 11th European PVM/MPI Users' Group Meeting, pages 97–104, Budapest, Hungary, September 2004.

[53] Vinícius Garcia Pinto, Lucas Mello Schnorr, Luka Stanisic, Arnaud Legrand, Samuel Thibault, and Vincent Danjean. A visual performance analysis framework for task-based parallel applications running on hybrid clusters. Concurrency and Computation: Practice and Experience, 30(18):e4472, 2018.

[54] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca Performance Toolset Architecture. Concurrency and Computation: Practice and Experience, 22(6):702–719, April 2010.

[55] Daniel Goodman, Georgios Varisteas, and Tim Harris. Pandia: Comprehensive contention-sensitive thread placement. In Proceedings of the Twelfth European Conference on Computer Systems, EuroSys '17, page 254–269, New York, NY, USA, 2017. Association for Computing Machinery.

[56] Mathias Gottschlag and Frank Bellosa. Reducing AVX-Induced Frequency Variation With Core Specialization. In The 9th Workshop on Systems for Multi-core and Heterogeneous Architectures. Dresden, 2019.

[57] William Gropp, Luke N. Olson, and Philipp Samfass. Modeling MPI Communication Performance on SMP Nodes: Is It Time to Retire the Ping Pong Test. In Proceedings of the 23rd European MPI Users' Group Meeting, EuroMPI 2016, page 41–50, New York, NY, USA, 2016. Association for Computing Machinery.

[58] T. Groves, R. E. Grant, and D. Arnold. NiMC: Characterizing and Eliminating Network-Induced Memory Contention. In 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 253–262, 2016.

[59] Bilel Hadri, Hatem Ltaief, Emmanuel Agullo, and Jack Dongarra. Tile QR Factorization with Parallel Panel Processing for Multicore Architectures. In 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), Atlanta, United States, April 2010.

[60] Nor Asilah Wati Abdul Hamid and Paul Coddington. Comparison of MPI Benchmark Programs on Shared Memory and Distributed Memory Machines (Point-to-Point Communication). The International Journal of High Performance Computing Applications, 24(4):469–483, 2010.

[61] Nor Asilah Wati Abdul Hamid, Paul Coddington, and Francis Vaughan. Comparison of MPI benchmark programs on an SGI Altix ccNUMA shared memory machine. In Paul Spirakis and H.J. Siegel, editors, Proceedings 20th IEEE International Parallel Distributed Processing Symposium, 2006.

[62] Blake Haugen, Stephen Richmond, Jakub Kurzak, Chad A. Steed, and Jack Dongarra. Visualizing execution traces with task dependencies. In Proceedings of the 2nd Workshop on Visual Performance Analysis, VPA '15, New York, NY, USA, 2015. Association for Computing Machinery.

[63] David L Hill, Derek Bachand, Selim Bilgin, Robert Greiner, Per Hammarlund, Thomas Huff, Steve Kulick, and Robert Safranek. The Uncore: A Modular Approach To Feeding The High-Performance Cores. Intel Technology Journal, 14(3):30–49, 2010.

[64] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. MPI + MPI: A new hybrid approach to parallel programming with MPI plus shared memory. Computing, 95, 12 2013.

[65] Torsten Hoefler, Timo Schneider, and Andrew 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, 25(4):241–258, 2010.

[66] Reazul Hoque and Pavel Shamis. Distributed task-based runtime systems - current state and micro-benchmark performance. In 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pages 934–941, 2018.

[67] S. Hunold and A. Carpen-Amarie. Hierarchical Clock Synchronization in MPI. In Dimitrios Nikolopoulos and Bronis R. de Supinksi, editors, 2018 IEEE International Conference on Cluster Computing (CLUSTER), pages 325–336, Los Alamitos, CA, USA, sep 2018. IEEE Computer Society.

[68] Sascha Hunold and Alexandra Carpen-Amarie. MPI benchmarking revisited: Experimental design and reproducibility. CoRR, abs/1505.07734, 2015.

[69] Sascha Hunold and Alexandra Carpen-Amarie. On the Impact of Synchronizing Clocks and Processes on Benchmarking MPI Collectives. In Proceedings of the 22nd European MPI Users' Group Meeting, EuroMPI '15, New York, NY, USA, 2015. Association for Computing Machinery.

[70] Roman Iakymchuk and François Trahay. LiTL: Lightweight Trace Library. Technical report, INF - Département Informatique, Telecom SudParis, July 2013.

[71] Intel Corporation. Pentium® Pro Family Developer's Manual, December 1995.

[72] Intel Corporation. IA-PC HPET (High Precision Event Timers) Specification, revision 1.0a, October 2004.

[73] Terry Jones and Gregory A. Koenig. Clock synchronization in high-end computing environments: a strategy for minimizing clock variance at runtime. Concurrency and Computation: Practice and Experience, 25(6):881–897, 2013.

[74] Terry Jones, George Ostrouchov, Gregory A. Koenig, Oscar H. Mondragon, and Patrick G. Bridges. An evaluation of the state of time synchronization on leadership class supercomputers. Concurrency and Computation: Practice and Experience, 30(4):e4341, 2018.

[75] H. Kaiser, M. Brodowicz, and T. Sterling. ParalleX An Advanced Parallel Execution Model for Scaling-Impaired Applications. In 2009 International Conference on Parallel Processing Workshops, pages 394–401, Sep. 2009.

[76] Timothy H Kaiser and Scott B Baden. Overlapping communication and computation with OpenMP and MPI. Scientific Programming, 9(2-3):73–81, 2001.

[77] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, Tools for High Performance Computing, pages 139–155. Springer Berlin Heidelberg, 2008.

[78] Andreas Knüpfer, Christian Feld, Dieter Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TA pages 79–91. Springer Berlin Heidelberg, Berlin, Heidelberg, 01 2012.

[79] J. Langguth, X. Cai, and M. Sourouri. Memory Bandwidth Contention: Communication vs Computation Tradeoffs in Supercomputers with Multicore Architectures. In 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS), pages 497–506, 2018.

[80] Alexey Lastovetsky, Vladimir Rychkov, and Maureen O'Flynn. MPIBlib: Benchmarking MPI Communications for Parallel Computing on Homogeneous and Heterogeneous Clusters. In Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 227–238, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[81] Gregory Lento. Optimizing performance with Intel advanced vector extensions. online, September, 2014.

[82] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. Adaptive, Transparent Frequency and Voltage Scaling of Communication Phases in MPI Programs. In SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, pages 14–14, 2006.

[83] Romain Lion and Samuel Thibault. From tasks graphs to asynchronous distributed checkpointing with local restart. In FTXS 2020 - IEEE/ACM 10th Workshop on Fault Tolerance for HPC at eXtreme Scale, Atlanta / Virtual, United States, November 2020.

[84] Jiuxing Liu, Dan Poff, and Bülent Abali. Evaluating High Performance Communication: a Power Perspective. pages 326–337, 01 2009.

[85] E. Maillet and C. Tron. On efficiently implementing global time for performance evaluation on multiprocessor systems. Journal of Parallel and Distributed Computing, 28(1):84–93, 1995.

[86] Zoltan Majo and Thomas R. Gross. Memory System Performance in a NUMA Multicore Multiprocessor. In Proceedings of the 4th Annual International Conference on Systems and Storage, SYSTOR '11, New York, NY, USA, 2011. Association for Computing Machinery.

[87] John McCalpin. Memory bandwidth and machine balance in high performance computers. IEEE Technical Committee on Computer Architecture Newsletter, pages 19–25, 12 1995.

[88] Stéphanie Moreaud and Brice Goglin. Impact of NUMA Effects on High-Speed Networking with Multi-Opteron Machines. In PDCS, Cambridge, United States, November 2007.

[89] Wolfgang E. Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer, 63, Vol. XII(1):69–80, 1996.

[90] Lucas Leandro Nesi and Lucas Schnorr. Detection, Evaluation and Mitigation of Resource Affinity and Communication Contention Problems in a Task-Based Runtime over Heterogeneous Clusters. In Anais do XXI Simpósio em Sistemas Computacionais de Alto Desempenho, pages 275–286, Porto Alegre, RS, Brasil, 2020. SBC.

[91] OpenMP Architecture Review Board. OpenMP Application Programming Interface. https://www.openmp.org/.

[92] Marc Pérache, Hervé Jourdren, and Raymond Namyst. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In Emilio Luque, Tomàs Margalef, and Domingo Benítez, editors, Euro-Par 2008 – Parallel Processing, pages 78–88, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[93] Romain Pereira, Adrien Roussel, Patrick Carribault, and Thierry Gautier. Communication-Aware Task Scheduling Strategy in Hybrid MPI+OpenMP Applications. In IWOMP 2021 - 17th International Workshop on OpenMP, OpenMP : Enabling Massive Node-Level Parallelism (IWOMP 2021), pages 1–15, Bristol, United Kingdom, September 2021.

[94] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. WoTUG '96: Proceedings of the 19th World Occam and Transputer User Group Technical Meeting on Parallel Processing Developments, 1996.

[95] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E Fagg, Edgar Gabriel, and Jack J Dongarra. Performance analysis of MPI collective operations. Cluster Computing, 10(2):127–143, 2007.

[96] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda. Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs. In 2013 42nd International Conference on Parallel Processing (ICPP), pages 80–89, Los Alamitos, CA, USA, oct 2013. IEEE Computer Society.

[97] Joachim Protze, Marc-André Hermanns, Ali Demiralp, Matthias S. Müller, and Torsten Kuhlen. MPI Detach - Asynchronous Local Completion. In 27th European MPI Users' Group Meeting, EuroMPI/USA '20, page 71–80, New York, NY, USA, 2020. Association for Computing Machinery.

[98] Andrès Rubio Proaño. Data Placement Strategies for Heterogeneous and Non-Volatile Memories Theses, Université de Bordeaux, October 2021.

[99] Kevin Sala, Sandra Macià, and Vicenç Beltran. Combining One-Sided Communications with Task-Based Programming Models. In 2021 IEEE International Conference on Cluster Computing (CLUSTER), pages 528–541, 2021.

[100] Kevin Sala, Xavier Teruel, Josep M. Perez, Antonio J. Peña, Vicenç Beltran, and Jesus Labarta. Integrating blocking and non-blocking MPI primitives with task-based programming models. Parallel Computing, 85:153–166, 2019.

[101] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. Parallel Computing, 35(12):581 – 594, 2009. Selected papers from the 14th European PVM/MPI Users Group Meeting.

[102] Joseph Schuchart, Philipp Samfass, Christoph Niethammer, José Gracia, and George Bosilca. Callback-based completion notification using MPI Continuations. Parallel Computing, 106:102793, 2021.

[103] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. OpenSpeedShop: An Open Source Infrastructure for Parallel Performance Analysis. Sci. Program., 16(2–3):105–121, April 2008.

[104] Marc Sergent. Passage à l'echelle d'un support d'exécution à base de tâches pour l'algèbre linéair Theses, Université de Bordeaux, December 2016.

[105] Marc Sergent, Mario Dagrada, Patrick Carribault, Julien Jaeger, Marc Pérache, and Guillaume Papauré. Efficient Communication/Computation Overlap with MPI+OpenMP Runtimes Collaboration. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, Euro-Par 2018: Parallel Processing, pages 560–572, Cham, 2018. Springer International Publishing.

[106] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. UCX: an open source framework for HPC network APIs and beyond. In 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, pages 40–43. IEEE, 2015.

[107] Sameer Shende. The Tau Parallel Performance System. International Journal of High Performance Computing Applications, 20:287–311, 05 2006.

[108] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S. Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, Wonchan Lee, Sean Treichler, Patrick McCormick, and Alex Aiken. Task bench: A parameterized benchmark for evaluating parallel runtime performance. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '20. IEEE Press, 2020.

[109] Quinn Snell, Armin Mikler, and John Gustafson. NetPIPE: A Network Protocol Independent Performance Evaluator. IASTED International Conference on Intelligent Information Management and Systems, 1, 06 1996.

[110] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. Concurrency and Computation: Practice and Experience, page 16, May 2015.

[111] Vaibhav Sundriyal, Masha Sosonkina, and Zhao Zhang. Achieving Energy Efficiency during Collective Communications. Concurrency and Computation: Practice and Experience, 25, 10 2013.

[112] Enric Tejedor, Montse Farreras, David Grove, Rosa M Badia, Gheorghe Almasi, and Jesus Labarta. A high-productivity task-based programming model for clusters. Concurrency and Computation: Practice and Experience, 24(18):2421–2448, 2012.

[113] Hiroshi Tezuka, Francis O'Carroll, Atsushi Hori, and Yutaka Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. pages 308 – 314, 01 1998.

[114] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. A taxonomy of task-based parallel programming technologies for high-performance computing. J. Supercomput., 74(4):1422–1434, apr 2018.

[115] Martin Tillenius. SuperGlue: A Shared Memory Framework Using Data Versioning for Dependency-Aware Task-Based. SIAM J. Sci. Comput., 37(6):C617–C642, jan 2015.

[116] François Trahay, François Rue, Mathieu Faverge, Yutaka Ishikawa, Raymond Namyst, and Jack Dongarra. EZTrace: a generic framework for performance analysis. In Nalini Venkatasubramanian and Craig Lee, editors, IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pages 618–619, Newport Beach, CA, USA, 05 2011.

[117] J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego CA, 2010.

[118] Jesper Larsson Träff and Andreas Ripke. Optimal broadcast for fully connected processor-node networks. Journal of Parallel and Distributed Computing, 68(7):887 – 901, 2008.

[119] Bogdan Marius Tudor, Yong Meng Teo, and Simon See. Understanding off-chip memory contention of parallel programs in multicore systems. In 2011 International Conference on Parallel Processing, pages 602–611, 2011.

[120] UEFI Forum, Inc. Advanced Configuration and Power Interface Specification, version 6.4, January 2021.

[121] J.D. Ullman. NP-complete scheduling problems. Journal of Computer and System Sciences, 10(3):384–393, 1975.

[122] Wei Wang, Jack W. Davidson, and Mary Lou Soffa. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale NUMA machines. In 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 419–431, 2016.

[123] Udayanga Wickramasinghe and Andrew Lumsdaine. A survey of methods for collective communication optimization and tuning. CoRR, abs/1611.06334, 2016.

[124] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM, 52(4):65–76, 2009.

[125] Thomas Worsch, Ralf Reussner, and Werner Augustin. On Benchmarking Collective MPI Operations. In Dieter Kranzlmüller, Jens Volkert, Peter Kacsuk, and Jack Dongarra, editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 271–279, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[126] Asim YarKhan, Jakub Kurzak, and Jack Dongarra. QUARK users' guide: Queuing and runtime for kernels, version 1.0. technical report UT-ICL-11-02, University of Tennessee Innovative Computing Laboratory, Knoxville, Tennessee 37996, April 2011.

[127] Yili Zheng, Amir Kamil, Michael B. Driscoll, Hongzhang Shan, and Katherine Yelick. UPC++: A PGAS Extension for C++. In 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pages 1105–1114, 2014.

Publications

Articles in Peer-reviewed Conferences

[128] Alexandre Denis, Emmanuel Jeannot, and Philippe Swartvagher. Interferences between Communications and Computations in Distributed HPC Systems. In ICPP 2021 - 50th International Conference on Parallel Processing, page 11, Chicago / Virtual, United States, August 2021.

[129] Alexandre Denis, Emmanuel Jeannot, and Philippe Swartvagher. Modeling Memory Contention between Communications and Computations in Distributed HPC Systems. In APDCM - IPDPS - 2022 - IEEE International Parallel and Distributed Processing Symposium W page 10, Lyon / Virtual, France, May 2022.

[130] Alexandre Denis, Emmanuel Jeannot, Philippe Swartvagher, and Samuel Thibault. Using Dynamic Broadcasts to improve Task-Based Runtime Performances. In Euro-Par - 26th International European Conference on Parallel and Distributed Computing, Euro-Par 2020, Warsaw, Poland, August 2020. Rzadca and Malawski, Springer.

[131] Philippe Swartvagher. Amélioration des performances de supports d'exécution à tâches à l'aide de broadcasts dynamiques. In COMPAS 2020 - Conférence francophone d'informatique en Parallélisme, Architecture et Systèm Lyon, France, June 2020.

[132] Philippe Swartvagher. Interactions entre calculs et communications au sein des systèmes HPC distribués. In COMPAS 2021 - Conférence francophone d'informatique en Parallélisme, Architecture et Systèm Lyon, France, July 2021.

[133] Philippe Swartvagher. Interactions entre calculs et communications au sein des systèmes HPC distribués : évaluation et modélisation. In COMPAS 2022 - Conférence francophone d'informatique en Parallélisme, Architecture et Systèm Amiens, France, July 2022.

Articles in Peer-reviewed Journals

[134] Alexandre Denis, Emmanuel Jeannot, Swartvagher Philippe, and Samuel Thibault. Tracing task-based runtime systems: feedbacks from the StarPU case. Concurrency and Computation: Practice and Experience. Submitted.

Research Reports

[135] Alexandre Denis, Emmanuel Jeannot, and Philippe Swartvagher. Modeling Memory Contention between Communications and Computations in Distributed HPC Systems (Extended Version). Research Report RR-9451, INRIA Bordeaux, équipe TADAAM, February 2022.

Technical Reports

[136] Pierre-Antoine Bouttier, Ludovic Courtès, Yann Dupont, Marek Felšöci, Felix Gruber, Konrad Hinsen, Arun Isaac, Pjotr Prins, Philippe Swartvagher, Simon Tournier, and Ricardo Wurmus. Guix-HPC Activity Report 2020-2021. Technical report, Inria Bordeaux - Sud-Ouest ; Université Grenoble - Alpes ; Université Paris, February 2022.

Posters

[137] Philippe Swartvagher. Interferences between Communications and Computations in Distributed HPC Systems. Euro-Par - 27th International European Conference on Parallel and Distributed Computing, August 2021. Poster.

[138] Philippe Swartvagher. Interferences between Communications and Computations in Distributed HPC Systems. Journée de l'École Doctorale Mathématiques et Informatique, May 2021. Poster.

Software Contributions

This section presents the contributions made to softwares for the purposes of the experiments done in this thesis.

NewMadeleine

- Implementation of the `mcast` interface for dynamic broadcasts, and related microbenchmarks.

FxT

- Add support to record executions generating trace files bigger than 2 GB.
- Add possibility to register a callback function to call when the event buffer is flushed on the disk.

StarPU

- Maintenance of the NewMadeleine backend.
- Integration of dynamic broadcasts in the NewMadeleine backend to use the `mcast` interface.
- Improvement of the tracing system:
  - Warn the user by printing a message when the event buffer of FxT is flushed during application execution.
  - Add option to filter the event to record in traces.
  - Add support of the `mpisyncclocks` library to synchronize distributed traces.
  - Save in traces the NUMA node on which are located data buffers manipulated by StarPU (for tasks or communications).

Chameleon

- Miscellaneous bug fixes.

Memory contention

- Development of a benchmark to evaluate interferences between computations and communications.
- Development of scripts to analyze, plot and model the results of the benchmark.

Guix

- Maintenance of the packages of the PM2 project (NewMadeleine, PIOMan, `mpisyncclocks`, and other sub-projects).
- Miscellaneous bug reports.

ViTE

- Miscellaneous bug fixes.