Compile-time Validation and Optimization of MPI Nonblocking Communications

van Man Nguyen

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR
DE L'UNIVERSITÉ DE BORDEAUX

ECOLE DOCTORALE DE MATHÉMATIQUES ET

D'INFORMATIQUE

Par **Van Man NGUYEN**

Compile-time Validation and Optimization of MPI Nonblocking Communications

Sous la direction de : **Denis BARTHOU**
Co-directeur : **Patrick CARRIBAULT**

Soutenue le 16 décembre 2022

Membres du jury :

Mme. Alexandra JIMBOREAN	*Professor*	*Universidad de Murcia*	Rapporteure
M. Purushotham BANGALORE	*Professor*	*University of Alabama*	Rapporteur
M. Vincent LOECHNER	Maître de conférences	Université de Strasbourg	Examinateur
M. Emmanuel JEANNOT	Directeur de recherches	INRIA	Examinateur
M. Denis BARTHOU	Professeur	Bordeaux INP	Directeur
M. Patrick CARRIBAULT	Ingénieur de Recherche	CEA	Co-Directeur
M. Julien JAEGER	Ingénieur de Recherche	CEA	Examinateur
Mme. Emmanuelle SAILLARD	Chargée de Recherche	INRIA	Examinatrice

Vérification et optimisation à la compilation des communications MPI non-bloquantes

Résumé : Les clusters de Calcul Haute Performance (HPC) sont composés de multiples unités de calcul ou de stockage mémoire, aussi appelés « nœuds », interconnectés par un réseau haut débit et basse latence. Une telle architecture est qualifiée de « distribuée ». Les calculs sont ainsi distribués sur ces nœuds qui vont chacun travailler sur une section d'une simulation, ce qui permet de réduire le temps d'exécution de simulations grâce au parallélisme. Les nœuds doivent toutefois se communiquer leurs résultats afin d'avancer dans les calculs, ce qui cause des latences. *Message Passing Interface* (MPI) est la solution la plus utilisée en HPC pour définir ces échanges entre nœuds de calcul. Elle définit notamment des communications point-à-point et des communications collectives. Ces communication existent en trois versions : bloquante, non-bloquante, et persistante. Les communications non-bloquantes permettent une meilleure utilisation des ressources de calcul en recouvrant les communications par des calculs, ce qui permet de réduire le temps d'obtention des résultats. Cependant, ces communications non-bloquantes sont plus complexes à l'usage, et offrent moins de mécanismes de sécurité. Les développeurs sont davantage susceptibles de commettre des erreurs de programmation pouvant conduire à des blocages du programme ou corrompre les résultats. Cela conduit à une moindre popularité de ces communications pour dégager du recouvrement, en particulier de la forme collective introduite lors de la troisième révision majeure de l'interface en 2012. L'objectif de cette étude est de proposer des méthodes visant à aider les développeurs à utiliser ces communications. Premièrement, nous proposons une méthode pour associer les appels non-bloquants lors de la compilation d'un programme MPI en utilisant des informations sur le flot de contrôle et de le flot de données. Dans un deuxième temps, nous proposons une méthode pour transformer automatiquement les appels bloquants dans leur version non-bloquante. Cette méthode va ensuite réorganiser le code d'une fonction en déplaçant les dépendances des communications dans le but de maximiser la taille des intervalles de recouvrement. Cette méthode est également appliquée sur les appels non-bloquants existants en exploitant les associations trouvées par la méthode de validation. Enfin, nous exploitons les limitations de la transformation automatique afin de proposer une méthode permettant d'améliorer le potentiel de recouvrement des programmes MPI en identifiant les bornes de ces intervalles et en suggérant des modifications de code à appliquer. Les trois méthodes que nous proposons ont été implémentées via des passes LLVM, et testées sur plusieurs benchmarks, dont des miniapps et des codes CORAL.
Mots-clés : Message Passing Interface, communications non-bloquantes, vérification de code, optimisation de code, compilation

---

Compile-time Validation and Optimization of MPI Nonblocking Communications

Abstract: High-Performance Computation (HPC) clusters are made of multiple computing and memory storage units (or nodes) interconnected with a high performance network. Such architecture is called "distributed". Computations are spread over these nodes which each work on a subset of the whole simulation, leading to increased performance thanks to parallelism. The results then need to be shared between the nodes to carry out the computations, which is source of latencies. The Message Passing Interface (MPI) is the most widely used solution in HPC to implement these exchanges. It defines multiple flavors of communications, including point-to-point and collective communications. These communications exist in three versions: blocking, nonblocking, and persistent. Nonblocking communications allow the overlapping of communications by computations, thus leading to a better use of computing resources and lower time to results. Yet this version of the communications, whose collective forms were added to the interface by its third major version in 2012, are harder to use because of their composition and offer less security mechanisms. Developers are more prone to commit programming errors which can lead to deadlocks or data corruption. Consequently, nonblocking communications, and more specifically the collective forms, are still not widely used to create overlapping opportunities. The goal of this study is the development of methods to help developers in using these communications. First, we propose a method to match nonblocking calls at compile-time and to detect programming errors involving those using information on the control flow and the data flow. Secondly, we propose a method to automatically transform existing blocking calls into their nonblocking versions. This method then reorganizes the code of a function by moving the dependencies of communications in order to maximize the length of the overlapping intervals. It is also applied on existing nonblocking calls using the matching information found by the verification method. Finally, we build upon the limitations of the automatic approach to propose a method to improve the overlapping potential of MPI programs by identifying the boundaries of overlapping intervals and suggesting code modification to developers. These three methods we proposed have been implemented as LLVM passes, and tested on several benchmarks, including miniapps and CORAL codes.

Keywords: Message Passing Interface, nonblocking communications, code validation, code optimization, compilation

---

Unités de recherche
STORM, Inria Bordeaux Sud-Ouest, 33000 Bordeaux, France.
CEA, DAM, DIF, F-91297 Arpajon, France.

Vérification et optimisation à la compilation des communications MPI non-bloquantes

Contexte

Alors que nous approchons des limites physiques des microprocesseurs, il devient de plus en plus difficile de gagner en puissance de calcul en augmentant leur fréquence de fonctionnement. Depuis le milieu des années 90, les ingénieurs en calcul haute performance (HPC) ont recours au parallélisme de masse afin d'augmenter la capacité d'un système à faire des opérations. Ces systèmes, aussi appelés clusters, sont de nos jours dotés de plusieurs centaines de milliers de nœuds interconnectés par un réseau haut débit et basse latence, sur lesquels les calculs seront distribués. Afin que le système puisse converger vers le résultat final, ces noeuds doivent se communiquer leurs résultats. Or, ces communications induisent des latences nous empêchant d'exploiter pleinement la puissance des supercalculateurs.

MPI (*Message Passing Interface*) est l'interface de programmation la plus utilisée en HPC pour implémenter ces communications entre noeuds de calcul. Cette interface définit plusieurs formes de communications, dont notamment les communications non-bloquantes. Sous cette forme, les communications sont scindées en deux appels distincts : un appel d'initiation, et un appel de complétion, tous deux reliés par un objet dénommé "requête". Cette forme nous permet de recouvrir les temps de communication par du calcul. En effet, l'appel d'initiation n'est pas bloquant, et rend le contrôle à la fonction appelante, ce qui nous permet d'exécuter des instructions pendant que la communication est menée, idéalement de façon concurrentielle. L'appel de complétion se charge d'attendre que les données de la communication soient à nouveau accessibles. Cette forme de communication nous permet d'optimiser l'usage des ressources de calcul. Toutefois, elle est bien plus complexe que la forme bloquante, et est davantage sujette aux erreurs de program-

mation. Une mauvaise utilisation des données de la communication ou une erreur d'association des appels non-bloquants peut fausser les résultats numériques, ou encore causer un blocage du programme. L'objectif de nos travaux est alors de proposer des solutions afin d'encourager les développeurs à utiliser cette forme de communication. À ce titre, nous avons identifié deux pistes : l'une consiste à les aider au cours du cycle de développement en relevant les erreurs de programmation qu'ils pourraient commettre, et l'autre consiste à les aider à exprimer le potentiel de recouvrement dans leurs programmes.

La grande majorité des programmes utilisant MPI sont écrits dans un langage compilé. Dès lors, il est intéressant d'analyser et de transformer les codes à la compilation. En effet, cela s'inscrit naturellement dans le cycle de développement, puisque la compilation du programme en est une étape indispensable, et de nombreuses analyses et transformations du code y sont appliquées. Nous pouvons alors nous baser sur ces "passes" existantes afin de définir la nôtre. De plus, l'analyse des codes à cette étape est peu coûteuse en ressources matérielles et temporelle : aucune exécution du programme cible n'est nécessaire, et ces solutions peuvent passer à l'échelle sur des codes de plusieurs centaines de milliers de lignes. Certains compilateurs, tels que LLVM, permettent l'extension de leur capacités via la définition de plugins. Nous nous basons sur ce mécanisme afin de permettre à LLVM de reconnaître les appels MPI et d'y appliquer un traitement spécifique.

C'est ainsi dans ce contexte que nous proposons les contributions suivantes afin d'encourager l'usage des communications non-bloquantes. Premièrement, nous allons définir une méthode de vérification à la compilation des communications non-bloquantes, qui puisse détecter erreurs d'association et d'accès concurrents aux données échangées. Deuxièmement, nous proposons une méthode permettant de créer et d'étendre automatiquement le potentiel de recouvrement d'un programme. Cette méthode va alors transformer tout appel de communication bloquant en son équivalent non-bloquant, et va déplacer les appels non-bloquants ainsi que leurs dépendances de façon à maximiser le potentiel de recouvrement tout en préservant sa validité. Enfin, nous proposons une méthode d'aide à l'optimisation par l'émission de suggestions de modifications afin de maximiser le potentiel de recouvrement d'un programme, et de passer outre les limitations trouvées par l'approche automatique.

Contributions

Cette Section résume les différentes contributions que nous avons pu faire afin d'encourager l'utilisation des communications non-bloquantes.

Vérification des communications non-bloquantes

Les communications non-bloquantes sont plus complexes à l'usage, et sont davantage propices aux erreurs de programmation. L'objectif de cette contribution est d'aider les développeurs à identifier les erreurs liées à une mauvaise association des appels d'une communication non-bloquante, ainsi qu'à une mauvaise utilisation des données de la communication, et à les corriger avant qu'elles n'aient un impact sur l'exécution du programme. La disponibilité d'un tel outil les rendrait plus enclins à utiliser la forme non-bloquante, résultant à de meilleures performances. La détection de ces erreurs s'effectue en deux temps. Dans un premier temps, pour chaque appel d'initiation, nous déterminons les appels de complétion formant l'intervalle de recouvrement de la communication. Pour cela, nous nous basons sur une étude du flot de contrôle à l'aide de la notion de post-domination, et sur une étude du flot de données afin de déterminer les chemins d'exécution possibles. Dans un second temps, pour les communications dont l'intervalle de recouvrement a pu être défini à l'étape précédente, nous déterminons s'il existe des instructions pouvant mettre à mal les données de la communication, y compris de la requête. Pour cela, nous construisons la liste des instructions définissant ou utilisant ces données, et vérifions si l'intervalle contient une telle instruction.

Cette méthode a été implémentée sous la forme d'une passe de compilation pour LLVM, et nous avons pu l'évaluer sur cinq benchmarks. Nous constatons que notre approche est capable d'associer correcter les appels non-bloquants, mais qu'elle se révèle imprécise quant à la détection des accès concurrents. Cela est causé par le manque d'une analyse interprocédurale, qui nous aurait permis de déterminer si une méthode peut modifier ou accéder à des variables non locales telles que des variables globales ou des attributs de classe. Enfin, nous proposons une adaptation de la méthode pour valider l'association des appels de communications persistantes.

Création et extension automatique du potentiel de recouvrement

L'intérêt majeur des communications non-bloquantes est la possibilité de recouvrir les temps de communication par du calcul. Or, l'expression du recouvrement est complexe en raison des dépendances de la communication, de l'algorithmique du code, et des contraintes des communications non-bloquantes. Ces raisons nous poussent à développer une méthode permettant de créer automatiquement du potentiel de recouvrement. L'originalité de notre approche repose dans le déplacement des dépendances. Cela nous permet de maximiser le potentiel de recouvrement des communications en réorganisant la structure des dépendances du code, tout

en préservant l'ordre des dépendances. Pour cela, nous transformons tout appel bloquant en sa forme non-bloquante. Nous analysons ensuite les communications non-bloquantes (y compris celles qui ont été converties) afin de déterminer leurs dépendances. Enfin, nous déplaçons appels et dépendances jusqu'à atteindre une frontière. Celle-ci peut être un autre appel de fonction, appel MPI, les bornes de la fonction appelante ou de la structure de contrôle dans laquelle la communication se trouve, ou encore une structure de contrôle possédant une dépendance qui ne peut être déplacée.

Nous avons implémenté cette approche sous la forme d'une passe LLVM, et nous l'avons appliquée sur cinq benchmarks. Les résultats prouvent la capacité de la méthode à créer des intervalles de recouvrement, dont certains ont une taille équivalente ou supérieure aux intervalles préexistants. Toutefois, la majorité des intervalles crées sont vides ou très étroits, ce qui ne leur permet pas de recouvrir la communication. En cause, de nombreuses communications et leurs dépendances se trouvent dans des branches conditionnelles, ce qui nous empêche d'exposer davantage de recouvrement. De futurs travaux pourront se concentrer sur ces motifs de code pour créer plus de recouvrement.

Maximisation du potentiel de recouvrement par une approche semi-automatique

L'approche automatique pour la création du potentiel de recouvrement est limitée par certains motifs de code. Afin d'outrepasser ces frontières, nous proposons ici une méthode basée sur la génération de suggestions de modifications du code à partir des analyses du compilateurs. Cette approche s'inscrit dans la continuité de la passe de transformation automatique. La génération des suggestions se base sur les résultats de la transformation automatique. Pour chaque type de frontière (un autre appel MPI, les limites d'une structure de contrôle, ou bien la présence d'une dépendance dans une structure de contrôle), une suggestion est donnée, accompagnée de la localisation de la frontière, ainsi que des dépendances. Ces éléments devraient faciliter la prise de décision et l'application de la transformation pour les développeurs.

S'agissant d'une méthode de transformation manuelle, bien qu'assistée par la présence des indices, il est impossible pour un développeur de traiter entièrement des codes qui peuvent avoir plusieurs milliers d'appels MPI. Dès lors, nous proposons également une méthode permettant de réduire le nombre d'appels à considérer. Celle-ci se base sur un profilage préalable du programme afin de déterminer le temps de communication de chaque opération MPI. En ne retenant que les

échanges dont le rapport du temps de communication sur le temps d'exécution du programme dépasse 1%, nous pouvons réduire le nombre d'appels à analyser à une dizaine pour chaque benchmark.

Enfin, nous avons appliqué la génération des suggestions sur certains de ces appels. Bien que complexe à analyser pour une passe de compilation, un regard humain, aidé par les bonnes indications, peut rapidement déterminer si, par exemple, une communication peut être sortie d'une boucle et des remplacements à effectuer pour préserver la validité du code. La taille des intervalles de recouvrement qui en découlent est supérieure à celle que nous obtenions par la transformation automatique. Toutefois, cette approche n'est plus complètement automatique, et passe difficilement à l'échelle sur un code entier. C'est la raison pour laquelle nous devons cibler nos transformations sur celles qui seraient les plus rentables. La détermination de cette rentabilité nécessite une exécution préalable du programme.

Conclusion

Les communications MPI non-bloquantes sont indispensables pour pleinement exploiter les capacités de calcul d'un cluster HPC. Toutefois, elles souffrent d'une popularité moindre, surtout concernant les communications collectives, du fait notamment de leur complexité. Afin d'encourager les développeurs des codes de simulation à en faire usage, nous travaillons sur deux axes. Le premier consiste à les aider à détecter les erreurs de programmation liées à l'utilisation des communications non-bloquante. Le second axe consiste à aider les développeurs à introduire du recouvrement dans leur code.

Cela a abouti sur trois contributions majeures. La première est le développement d'une méthode de détection à la compilation des erreurs causées par une mauvaise association des appels d'une communication non-bloquante et des erreurs causées par une mauvaise utilisation des ressources d'une communication. Cette méthode montre de bons résultats en ce qui concerne l'association des appels, mais se révèle imprécise sur la détection des accès concurrents. Ce point pourrait être amélioré par l'introduction d'analyses interprocédurales permettant de mieux discerner les effets d'un appel de fonction sur la mémoire. La seconde contribution est le développement d'une passe de transformation automatique capable de maximiser le potentiel de recouvrement d'une application en déplaçant appels et dépendances. Son application sur plusieurs benchmarks montre qu'elle est capable de créer du recouvrement dans des programmes qui en étaient dépourvus, mais qu'elle est limitée par certains motifs de code dont la résolution à la compilation est complexe. Enfin, la troisième contribution est une méthode permettant d'aider

les développeurs en générant des suggestions de modifications à partir d'une analyse et d'une transformation du code à la compilation. Couplé à un profilage du programme cible, cette méthode permet la création d'intervalles de recouvrement plus larges et de façon plus ciblée, maximisant l'impact des transformations tout en limitant l'investissement nécessaire de la part des développeurs.

À l'issue de nos travaux, plusieurs axes d'amélioration sont possibles. En premier lieu, l'ajout d'une analyse interprocédurale permettra d'affiner les résultats de la vérification et de la transformation. Les méthodes d'analyse et de transformation que nous avons proposés dans cette étude peuvent également être étendues à d'autres formes de communication MPI, notamment aux communications persistantes et partitionnées, mais aussi à d'autres modèles de programmation. Par exemple, les promesses, introduites dans le standard 17 du C++, ou encore les opérations asynchrones sur les streams CUDA, adoptent une forme similaire aux communications non-bloquantes. Enfin, à l'instar des optimisations *back-end* spécifiques à certaines architectures de microprocesseurs, permettre aux compilateurs d'effectuer des analyses et des optimisations spécifiques à un type de réseau d'interconnexion ou à une architecture de cluster pourrait être bénéfique pour la qualité des programmes voués à s'exécuter sur de telles machines. Cela pourrait par exemple nous permettre d'estimer directement lors de la compilation les appels qui seraient intéressants à optimiser en priorité.

Acknowledgments

What an adventure it had been! The last few months of this thesis had been really intense. One really couldn't find a more fitting end to it than getting the influenza days before the defense and a public transportation strike that week. Some say that we have to be passionate to engage in a scientific career. The three kilometers walk from my rent to the place where my defense was scheduled, carrying my *pot de thèse* on my arms, during an unusually warm winter, reflecting on my quest, is certainly one Passionate moment that I will fondly remember for the rest of my life.

In all seriousness, I would like to first mention Pr. Jimborean and Pr. Bangalore, who have accepted to review this manuscript despite the short reviewing period, and to whom I would like to express my gratitude. I would like to thank Pr. Jeannot for presiding my defense, but especially Pr. Loechner, who agreed to cross the whole expanse of France from Strasbourg to Bordeaux to attend to my defense, defying the train cancellations and the strike. It was really unfortunate that many of you could not attend to it in Bordeaux, but I am sure that we will have the opportunity to cooperate on further projects very soon!

Speaking of attending to my defense, I believe we had quite an audience in remote, thanks to colleagues who, from what I could gather, organized a literal watch party at Teratec. So many people attended, so to all of you, thank you, my friends, for staying even though it was the afternoon right before the winter break! Not only did old acquaintances showed up in the call, but also wonderful people I've met along the way. Anton, Simon, Florian, Sofiane, Romain, Xavier, Alexiane, and so many others I wouldn't be able to fit all of you guys in here, thank you for making these three years so enjoyable despite the pandemic. I also had the chance to meet incredible people who welcomed me in Bordeaux, within the STORM Inria team. Gwenolé, Maxime, Baptiste, and Célia, I am ever so grateful to you for accepting me in the open-space whenever I visited. We had many memorable rants about the paperwork needed for our theses! I would also like to dedicate a message for my family who had been very supportive and to the

many friends that I met in both the real world and in virtual worlds, who helped me to find the strength to continue and to find joy even amid deepest despair.

Finally, I would like to have some words for my supervisors. Five years ago, I barely knew what HPC was all about. Under the guidance of so many teachers, masters of their arts, and in particular from Julien and Patrick, I discovered to joy and the woes of analyzing MPI codes. Through you, I had the chance to meet the two wonderful persons that are Denis and Emmanuelle. Together, we spent three amazing years, trying to figure out data dependencies in IR and trying to squeeze out every single bit of overlapping out of a function. Through our weekly meetings, you found the right words of encouragement, and steered me back on tracks when needed. I can't thank you enough for your presence during this thesis, and I hope we will work together in the future.

Before we move on to the main matter, I would like to have some words to whoever might be reading this work. Maybe you are just taking your first steps in the world of academic research, maybe you are nearing the end of your thesis, still battling with your own manuscript. Whatever you are full of ambitions and projects, or struggling to find the correct words to express your ideas and feelings, know this: you are not without allies. Reach out to your peers, your comrades, your friends and relatives, and share your emotions. Do not let them overwhelm you. We all wish to hear your voice, feel your happiness, and together, think about solutions that might help in relieving your burdens and sorrows.

Contents

Introduction

As we approach the physical limitations of the material, it becomes more and more difficult to gain computing performance by ever increasing the frequency of microprocessors. Instead, engineers turned to massive parallelism by increasing the number of computing units a system can hold, especially in high performance clusters. Communications between each unit become necessary, but they can hinder the performance of the whole system. This problematic is even more valid on these clusters which have a distributed memory architecture: distant computing units have to exchange data through an interconnect. No matter its bandwidth, the interconnect can be source of performance degradation. While a computing unit is waiting for the information to be transmitted, it is not harnessing its computing resources. As a consequence, the system is not exploited to its full potential.

The Message Passing Interface (MPI), one of the most used programming library in HPC to handle communications between distant computing nodes, proposes a solution to this problem by defining nonblocking operations. They are composed of two calls. One initiates the communication and gives back control to its caller, and one blocks the caller until the completion of the communication. This form allows the insertion of statements between the calls which will hide the communication time. Indeed, on appropriate systems, these statements can be executed concurrently on the computing node while another entity independently progresses the communication. Ultimately, it leads to a better usage of computation resources, and reduces the execution time of MPI codes. Yet, these calls are not as popular as the blocking form of MPI communications. More specifically, the collective form of the nonblocking calls are still not widely in use to allow overlapping in codes, even ten years after their introduction to the interface. This is mainly caused by the complexity of the form, which requires the developers to ensure the correct matching of the nonblocking calls. Furthermore, the safety of the communication buffers is not guaranteed during the exchange, and it also falls to the developers to make sure that there are no illegal accesses to them. As a result, nonblocking communications are more prone to programming errors leading

to deadlocks or race conditions, which are faults that can be difficult to track.

The goal of this work is to help developers in using MPI nonblocking communications. To this end, we identified two approaches: detecting and reporting misuses of nonblocking calls, and assist in the creation of overlapping opportunities in codes. We decided to tackle these approaches at compile-time. The vast majority of MPI programs are written in compiled languages such as C or Fortran. It is very convenient to analyse and transform the code at compile-time, and scales fairly well as the size of the targeted program increases, as opposed to dynamic approaches. The tools acting at compile-time can accompany the developers through the whole life cycle of a program.

In our work, we propose three major contributions. The first contribution is the definition of a method to check, at compile-time, the matching of MPI nonblocking initiation and completion calls and to detect race conditions caused by an improper use of the communication arguments. Its implementation successfully detected several dangerous situations in large scale benchmarking applications. The second major contribution is the definition of a compiler optimization pass that automatically creates and extends the overlapping potential of a program. It transforms all existing blocking communications into their nonblocking form. All nonblocking calls are matched with the help of the verification method, and their overlapping interval is extended by reorganizing the structure of the code without harming the semantics of the program. The pass has been successfully applied on several benchmarks, and it was able to introduce overlapping potential in existing code. The third and last major contribution is the definition of a tool-assisted method to guide the developers of an MPI application to adapt their code in favor of the length of overlapping intervals. It leverages the results of the optimization pass to provide suggestions of code modifications to further lengthen the intervals, leading to greater overlapping potential. Coupled to a profiling of the program, we were able to determine the most impacting communications to focus our efforts in maximizing their overlapping potential.

This dissertation is composed of seven Chapters. The first two provide the key concepts to grasp our work on the High Performance Computing discipline, the Message Passing Interface, and on optimizing compilers. Chapter 3 presents the State of the Art on the verification of MPI programs and on the optimization of their overlapping potential. Chapter 4 discusses the compile-time analysis techniques we make use of, and their adequacy for the verification and the optimization of MPI programs. The verification method is the highlight of Chapter 5, while Chapter 6 focuses on the automatic optimization pass. Before the concluding part of this thesis, Chapter 7 exposes the compiler-assisted approach to guide

the developer in the maximization of the overlapping potential of a code.

Part I

Context and Literature Review

Chapter 1

High Performance Computing

A scientific simulation is defined as the imitation of a natural phenomenon by a numerical model on a computer[1]. Since the half of the twentieth century, scientists have been collaborating with models on computers to study natural phenomenons. The ENIAC machine, built in the late 40s, is often recognized as the first general purpose and programmable electronic computer. With its help in 1950, meteorologists were able to provide numerical weather forecasts[2], among other types of works. Beside enabling tests of a model before setting an experiment up, simulations that can imitate a real world event with enough precision can help in avoiding waste of resources (e.g. simulating material deformations in a crash test), or unethical experiments on live subjects. Nowadays, scientific computing resources are several orders of magnitude faster and offer much more precision than ENIAC. In this Chapter, we show how humankind was able to exponentially increase its computing powers.

1.1 Computing clusters

1.1.1 Evolution of supercomputers

Supercomputers are large machines designed to reach the highest computing performance possible, in execution speed and precision. They are made to run large and intensive simulation codes that might last for weeks[3]. Engineers and researchers have been striving to increase the performance of supercomputers, thus reducing the execution time of these simulations.

There are two approaches for a computer to lower its time to result. First, have its central processing unit (CPU) performing more computations in the same time

unit. Tremendous progress have been made in that regard since the first processors made by Intel in the 70s. Released in 1971, the Intel 4004 had a clock frequency of 750kHz, compared to the AMD Ryzen 7900X released in 2022, capable of going up to 5.6GHz. However, the growth in clock speed had not been a constant increase. Figure 1.1 shows the highest clock frequency found in a CPU released per year, from 1971 to 2014. While gains have been exponential between the 70s and the beginning of the millennium, it has been stagnating since 2005.

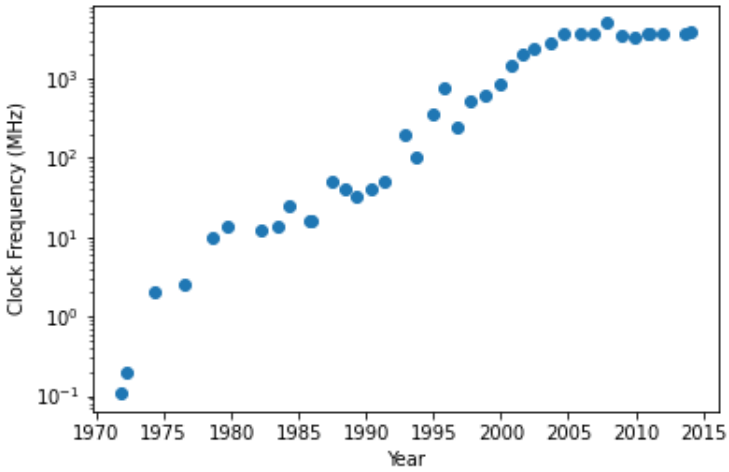


Figure 1.1 – Highest clock frequency CPU released per year [1]

Dennard et al. suggested that, the smaller a transistor is, the better its performances are[4]. As the size of a transistors shrinks, the required power to achieve a given performance level also shrinks. In other words, keeping the same voltage and current on a smaller transistor should lead to better performance. From several micrometers in the 70s, nowadays the dimensions of transistors is measured in nanometers. Yet, increasing the clock frequency of computing units seems harder and harder. It indicates that we might be close to electrical and material limitations, before entering the quantum domain[5].

The other approach is parallelization, or having a computing unit performing more operations in a clock cycle. Hailed as the first supercomputer, the ILLIAC IV was put in operation in 1975[6]. It was quickly followed by the Cray-1 the next year[7]. They are among the first parallel computers, and share the similarity of sporting a vector processor. It allowed them to operate on an array or vector of

1. Data from `http://cpudb.stanford.edu`

data, thus multiplying their computing power. Subsequent machines would add more of those processors, such as the Cray X-MP with four vector processors. This architecture allowed the handling of multiple instructions, and multiple data (MIMD)[8].

Parallel to the development of vector processors, microprocessors for personal home computer saw rapid growth. From the release of the Intel 4004 in 1971 to the end of the 90s, performance of microprocessors have been almost doubling each year[9]. Those small, cheap, and power efficient chips became more and more popular through the years[10]. In the beginning of the millennium, thanks to the miniaturization of transistors and of components, fitting multiple processing units within the same chip became possible, leading to the creation of multicore microprocessors. Widely in use in our computers today, they allow a chip to perform several tasks at the same time. To achieve massive parallelism, several microprocessors are linked together, first on the same board forming multiprocessor computers, then across multiple boards with the help of a dedicated network, thus forming clusters of computers[11].

During the previous decade, graphics processing units (GPU) became more and more prominent in high-end computers to satisfy the demands of the general public for fast video processing and graphics rendering, especially in video games. GPUs, similarly to vector processors, are specialized cards that excel in handling arrays of data through massive parallel operations and high speed memory. Their ability to provide a better performance/power on this kind of workload compared to regular CPUs make them interesting additions to computing clusters[12][13]. Clusters equipped with GPUs, also labeled "accelerators", are said to have an heterogeneous architecture.

Today's most powerful supercomputers are heterogeneous clusters. In their efforts to reach the symbolic barrier of the exaflop (10^{18} floating-point operations per second), engineers and researchers made clusters of thousands of interconnected CPUs and GPUs, consuming altogether several megawatts. Their efforts are recorded in a ranking which is updated every six months[14]. As of June 2022, the fastest supercomputer is the american-built Frontier supercomputer, installed at the Oak Ridge National Laboratory. It is the first machine to enter the exascale era, with a maximum performance of 1.1 exaflop. This feat is made possible with the help of more than 9 400 CPUs and 37 000 GPUs interconnected with the Slingshot interconnect[15].

1.1.2 Hardware architecture of a computing cluster

Today's scientific computing resources are concentrated on cluster computers. In this context, a cluster is a set of smaller scale computers, also called computing nodes, interconnected by a local network[11]. Mathematical models are discretized, and each computing node independently handles a small subset of the whole simulation, eventually resulting in shorter execution times thanks to parallelism.

The performance of each computing node is close to a high end home workstation. They are composed of one or multiple multicore CPUs forming a multiprocessor structure, accelerated with multiple GPUs. The computing units within a node share the same memory pool. Such construction is also called a symmetric shared memory multiprocessor (SMP). However, as the number of computing cores increases inside a node, it reduces the amount of memory per core, and challenges the bandwidth of the memory bus. In an attempt to tackle these issues, another hierarchy layer is added to restrict the memory space a processor can interact with, resulting in smaller SMPs, and thus restoring bandwidth for accesses to local data[16]. Should a processor access data from the memory space of another, it must perform a remote query through a bus linking the two memory areas. This remote access is more time consuming than the local access, hence the name of this structure: Non-uniform Memory Access (NUMA).

A computing cluster is composed of several computing nodes upon which computations are parallelized, with each node only processing a portion of the whole picture. These nodes will inevitably have to communicate with each other to carry on the simulation to its end state. For example, a node might need the temperature calculated by the neighboring nodes to determine the temperature at the next time step. Yet, a NUMA node only has access to its local memory spaces. Data transfers are needed to retrieve other nodes' results. In a computing cluster, a local network interconnects these nodes, allowing these data exchanges. This network must be able to achieve high communication speed and throughput to handle large quantities of data. In 2022, the most used interconnect technologies are InfiniBand, 10 and 25G Ethernet, Intel's Omni-Path, Cray's Aries and Slingshot, IBM's BlueGene, or Atos' BXI[17]. Such architecture is dubbed as "distributed memory".

1.1.3 Software architecture of a computing cluster

Coding an application for a high performance computing cluster is nothing like coding for a home desktop station. Data handling is key to an HPC program. Developers of such programs must take notice of sheer amount of computing units

to exploit, as well as to the dichotomy between the shared and distributed memory scopes. This results in specific programming models and interfaces for each scope.

In the shared memory area, interfaces such as OpenMP and the POSIX threads are widely used. They allow the definition of parallelizable sections in the code, and will create threads which are executed, ideally in parallel, on the cores of one node. Data protection is enforced through the expression of the dependencies and the visibility of variables in OpenMP, or by mutexes in POSIX threads.

In the distributed memory area, message passing is the dominant paradigm. It consists in defining messages, their content and format, the recipients and the senders, and conveying the messages on a network. There is one major library currently in use that implements this parallel programming paradigm : the Message Passing Interface (MPI), which will be the star of our study and of the next Section. Many HPC programs are written using a combination of OpenMP, for handling data inside a NUMA node, and MPI, for handling data between NUMA nodes. Such kind of programming model is called "hybrid", or "MPI+X", where X can be, for example, OpenMP.

Finally, with the advent of heterogeneous clusters, specific programming models had to be defined to leverage their ability to efficiently handle large arrays of data. To that end, manufacturers of GPUs have designed coding interfaces helping developers to use their power. We can mention Nvidia's CUDA, or the target directives of OpenMP. The latter allows to define regions of code to execute on a specific device, such as an accelerator.

1.2 The Message Passing Interface

First released in 1994, the Message Passing Interface is a programming interface defining multiple flavors of communications between MPI processes, which we will detail in this Section, ways to organize and manipulate data and to structure the MPI processes into specific topologies. In this regard, it has become the most widely used solution in HPC programs to implement communications across distant memory locations.

The initial release defined point-to-point communications in their blocking, nonblocking, and persistent versions, blocking collectives, as well as other utility functions[18]. In 1997, the second major revision of the interface most notably saw the addition of one-sided communications as "remote memory accesses" (RMA). The third major revision, released in 2012, added nonblocking collective communications. Finally, in 2022 for its fourth major release, persistent collectives were added to the interface, along with partitioned point-to-point communications[19].

The interface is currently being maintained by the MPI Forum which regroups volunteers, mostly HPC researchers and users of the interface, to decide on the amendments and additions.

The forum only defines the interface, which includes the functions with C and Fortran bindings, and their expected behavior. They do not provide an implementation of these procedures. It is left to the open-source community, hardware vendors, and private initiatives. The two most used implementations are MPICH[20] and OpenMPI[21], which are open-source implementations. We can also note other open-source implementations such as MPC from the CEA[22], madMPI which is a component of the NewMadeleine library from the Inria[23], or ExaMPI from the UTC and the LLNL[24]. Supercomputer vendors also provide their own MPI implementation, which are often derived from one of the two major open-source implementation: for example, Intel MPI is derived from MPICH[25].

1.2.1 MPI communications

MPI communications are the core component of the interface. In this Section, we describe the various kind of communications. We focus on the nonblocking communications, their benefits and impact on the code and the development of the program.

Initializing the MPI execution environment

MPI communications involve multiple independent MPI processes, which are the senders or the recipients (or both at the same time). In an MPI+X context, one MPI process usually matches with one SMP scope, while the "X" library, for example OpenMP, handles the data exchanges inside the SMP scope. It is also possible to have multiple MPI processes within one SMP scope. There are two ways to initialize these MPI processes: the *world model*, and the *sessions model*.

In the world model, the MPI processes are initialized by a call either to the `MPI_Init` function, or to the `MPI_Init_thread` function. These calls allow the execution of most MPI operations, such as the communications. The latter function specifies the rules and the expected behavior to which the MPI implementation must abide when used with threads, and how these threads are allowed to call MPI operations. Eventually, each MPI process initialized by this model must call `MPI_Finalize` to clean up the initialized objects. All pending operations in which the MPI process is involved, such as nonblocking communications, must be completed beforehand. These initializing and finalizing calls define the boundaries of the MPI environment in a code, and can only be called once during the lifetime of

a program. Once initialized, the MPI processes can be grouped in communicators, with the one containing all initialized MPI processes called `MPI_COMM_WORLD`.

The sessions model allows to bypass the limitation of the world model. During the lifetime of a program, multiple "sessions" can be initialized, with a call to `MPI_Session_init`, and then finalized, with a call to `MPI_Session_finalize`. It is then possible to create a group of MPI processes that are initialized in the session, and generate a communicator from the group.

Beside the initialization and finalization methods, and the definition of the "global" communicator, MPI communications behave the same for both models. From now on, we focus on the world model.

Overview of MPI communications

A communication involves multiple MPI processes. Communications can be categorized in three classes: point-to-point communications which only involve a sender and a recipient, collectives which involve one to multiple MPI processes, and one-sided communications which allow an MPI process to directly read or write to the memory of another MPI process.

A communication can be split in four distinct steps. The first step is the *initialization*. At this stage, only the addresses of buffers and other meta arguments are given to the communication. Then comes the second step, which is the *starting* step. Now, the communication is in possession of the contents of the buffers. After that is the *completion* stage, which releases the content of those buffers. At this point, the inbound buffers are correctly filled with the data the communication was expecting. They can safely be reused. Finally, the *freeing* step is the release of the remaining arguments and of the memory locations of the buffers.

Point-to-points are the most basic kind of communication. There is a receiver, and a sender. The receiver calls the `MPI_Recv` function, and waits until the expected data from the designated source arrives and is locally copied, before carrying on. On the emitting end, four modes of sending primitives exist.

— The **buffered** mode is possible when the data to send can fit into a dedicated communication buffer. To use this mode, the sender calls `MPI_Bsend`, which will copy the content of the message into the buffer. It returns the control back to the caller function once the copy is done, independently of the reception status.

— The **synchronous** mode is generally preferred if the message cannot fit into the buffer. Upon calling the `MPI_Ssend` primitive, the sender will wait until its partnered MPI process is ready to receive the message.

— The **ready** mode only allows the sender to call `MPI_Rsend` if the receiver is ready to catch the message. The operation results in an error if the receiver is not ready.

— The behavior of the fourth mode, the **standard** mode which is used by `MPI_Send`, depends on the implementation and on the options set by the users. It decides to perform either a buffered send, or a synchronous send.

Collective communications, as their name suggests, involve all MPI processes from a communicator. There are multiple kinds of collective communications. Some collective operations have a "variable" variant named `MPI_*v`, e.g. `MPI_Gatherv`. This allows the specification of the number of elements to send or receive for each MPI process involved in the communicator.

— The synchronization **Barrier** blocks all involved MPI processes until they all reach the function.

— The **Broadcast** function defines one sender, also called the "root", which propagates data to all MPI processes in the targeted communicator.

— The **Gather** function defines a receiver with an array of data to fill. All MPI processes in the communicator send a piece of information into this array. The **Gatherv** variant also exists.

— With an **Allgather**, each MPI process has a piece of data to send, and an array of data to fill. It can be summarized as performing the gathering call on all MPI processes. The **Allgatherv** variant also exists.

— The **Scatter** function defines a sender with an array filled with data. Each MPI process in the communicator receives a distinct piece of information from this array. The **Scatterv** variant also exists.

— The **Reduce** function defines a recipient and an operation. All other processes from the communicator send an information to the receiver, which will perform the operation on this data, and store its result. The possible operations include arithmetic operations such as the sum or the multiplication, and Boolean operations such as the logical `and` or `or`.

— The **Allreduce** function is similar to the regular reduction, with all MPI processes in the communicator receiving the result.

— The **Reduce-scatter** function is a reduction followed by an `MPI_Scatterv` operation. The **Reduce-scatter-block** variant also exists, which is a reduction followed by an `MPI_Scatter` operation.

— The **Scan** operation performs a reduction into the reception buffer of an MPI process with the rank i from the MPI processes with a rank in $[0; i]$.

The exclusive variant **Exscan** also exists, which performs the reduction for the MPI processes with a rank in $[0; i-1]$.

— Finally, in an **Alltoall** operation, each MPI process has an array of data to send, and an array of data to receive. An MPI process receives a segment of data from all other MPI processes, and sends one distinct piece of its data array to each other MPI process. The **Alltoallv** variant exists, but also an **Alltoallw** variant, which allows the specification of not only the size of the message for each MPI process, but also of the type of the contained data.

Developers using collective communications in their code must take extra care to their order to avoid deadlocking their program. Indeed, all processes of a communicator must call the collective primitives in the same order. For example, there is a risk of deadlock in a communicator if the MPI process with the rank 0 performs a barrier followed by a broadcast, but the MPI process with the rank 1 does the broadcast before the barrier.

All of the previously described communications exist in three versions: blocking, nonblocking, and persistent. In the following, we will detail each version, and introduce the notion of overlapping.

Finally, the interface also defines one-sided communications through RMA. In this mode, the MPI processes define a window during which RMA operations are permitted. An MPI process is then able to access the memory of another, without notifying and without having the remote host being aware of this access. However, this class of communications does not fall into the scope of the present work.

Blocking communications

This is the most basic form of communication. All the communications we described in the previous Section are blocking communications. The four steps of an MPI operation are merged into a single call. Consequently, upon reaching a blocking call, the contents of the communication buffers are provided to the communication. The call will not return until the involvement of the MPI process in the communication has not finished. For example, a synchronous `MPI_Send` (i.e. behaving like an `MPI_Ssend`) will not return until the receiving end is ready to acquire the data, whereas a buffered `MPI_Send` (i.e. behaving like an `MPI_Bsend`) is completed and is free to return as soon as the data has been copied to a buffer.

This form of communication protects against data races. It is impossible to modify the communication buffer while the exchange is ongoing, since the communication calls will only return once the buffers are correctly filled, or not needed

anymore.

Nonblocking communications

Nonblocking communications are split into two procedure calls. One regroups the initialization and starting phases, and is called the initiation call. The other regroups the completion and freeing stages, and is commonly called the completion call. The procedure calls of a nonblocking communication are linked together by an object called the request (`MPI_Request`). While the details of its form is left opaque in the standard, a request must carry the communication metadata such as the rank of the sender and of the recipients, the type of request, and, more importantly, the status of the communication.

The initiation call is of the form `MPI_I*` followed by the desired type of communication primitive (`send`, `bcast`, `alltoall`, ...). The prototype of most nonblocking initiation calls is strictly similar to their blocking counterpart, with the addition of the request at the tail of the arguments list. Regrouping the initialization and the starting phases together, its role is to take ownership of the request, and prepare for the communication by identifying the address space of the communication buffers and their contents. It returns as soon as these preparations are done, regardless of the readiness of the recipients or of copying buffers.

Regrouping the completion and freeing phases, the role of the completion calls is to ensure the communication buffers and the other arguments can be safely reused. Past them, and supposing the code does not contain any error, buffers can be accessed and modified again, and inbound buffers are guaranteed to contain the expected contents. In other words, it allows the reuse of these arguments. The completion calls can be grouped in two families. The first, the simplest, is the waiting calls. It waits until the request attached to the communication is marked as complete, marking the end of the MPI process' involvement in the process, and the possibility of using the buffers. The second, the testing calls, only checks if the request has been marked as "completed". If the request has been marked as "completed", `MPI_Test` sets a flag reflecting the completeness of the request. Consequently, accesses to the communication buffers is only safe if that flag indicates the completeness of the exchange. Both form exist in four variants.

— The base variant, `MPI_Wait` and `MPI_Test`, handles only one request.

— The "all" variant, `MPI_Waitall` and `MPI_Testall`, takes an array of requests as argument, and handles all of them.

— The "any" variant, `MPI_Waitany` and `MPI_Testany` takes an array of requests, and handles one of them. The processed request cannot be known in

advance, it is only determined at the execution of the program.

— The "some" variant, `MPI_Waitsome` and `MPI_Testsome`, takes an array of requests, and handles a variable number of them. Similarly to the any form, the processed requests are unknown before execution.

The nonblocking form allows the creation of overlapping intervals. It consists in covering the communication times by computations. This leads to a better use of computing resources. Indeed with blocking communications, the MPI process has to wait until its involvement in the exchange is finished. In this configuration, the MPI process is not performing anything "useful" during the exchange. In nonblocking communications, since the initiation call returns control to the caller function as soon as the preparations to start the communication are finished, the MPI process is free to perform the independent computations that come after. It is only blocked once it arrives at the completion call. In order to effectively hide the communication time, the communication must be carried, or progressed, independently. For example, in a NUMA node, it is possible to dedicate a computing unit to progress incoming and outgoing communications. Another solution can come from specific network interface hardware. Some networking cards, such as those in use for Atos' BXI interconnect, are equipped with microprocessors to handle communications.

However, this form is more prone to coding errors. Each initiated communication must be completed, and the safety of the communication buffers, and the request, must be preserved inside the overlapping interval.

Persistent and partitioned communications

While the following Chapters of this work mostly focus on the nonblocking communications, we should also note the persistent and partitioned communications, which also allow overlapping. They are split in four calls, each one corresponding to one of the communication phases. Persistent communications are designed to reduce the overhead caused by the initialization step. In this mode, the argument list of the communication and the request is initialized once. The request is then set to active when starting an exchange, and to inactive when completing one. This is particularly suited for loops of communications and computations. More details on this mode are provided in Section 5.4.

The partitioned mode, which has been added to the interface in 2022, only exists for point-to-point communications at the time of writing. They serve a similar purpose as persistent communications in avoiding the initialization overhead. In this mode, the communication data is cut into smaller partitions. It is then

possible to notify the MPI process about the readiness of a partition so that it can be sent as soon as possible. In a similar fashion on the receiving end, it is possible to wait for a partition of the communication buffer to handle it as soon as it has been received. It allows developers to finely interleave communications and computations, thus creating overlapping opportunities.

Chapter 2

Compile-Time Code Analysis and Transformation

Our computers are designed to manipulate bits: successions of 0s and 1s forming comprehensible data and instructions for the targeted processor. Any order or information we want to give to a computer must first be put under this form, also called the machine language. While the pioneers of computer science wrote their first programs directly in this language, it is extremely hard to decipher for humans. The role of a higher level language is then to make it easier for developers to express their intents to the computer. Code files written in high level languages are easier to read and write by humans, but computers cannot decipher them. A compiler is one of the solutions to translate a file written in a high level language to a file containing instructions in machine language that can be processed by a computer, hence the name "executable". The first human readable programming languages and their compilers appeared in the 1950s in the form of, most notably, FORTRAN and Algol, respectively in 1957 and 1958[26].

Compilers can also be regarded as tools to aid developers. They can help in two ways: detection of programming errors in the source code, and optimization of the program. The FORTRAN compiler developed in the 50s was able to find some errors in source code files, such as a missing comma, and perform some optimizations on the allocation of registers[26]. In 1973, Wulf et al. described the design of an optimizing compiler to produce machine code adapted that will efficiently use the capabilities of a computer architecture[27]. It consists in a succession of phases whose goal is to translate a source code file into a binary object. Each phase would perform an action on a given representation of the code in some language, resulting in another representation, potentially in another

language, which is then processed by another phase. While the Bliss/11 language, for which this compiler was conceived, has been superseded by the C programming language since long, today's compilers are still following this design, such as GCC and LLVM.

The vast majority of MPI programs are written using one, or a combination of compiled languages: C, C++, or FORTRAN. Thus, it is possible to analyze the behavior of such programs, and perform MPI specific optimizations at compile-time. However, the compilers are not aware of the semantics of MPI operations, and thus they are unable to, for example, automatically detect MPI specific errors or to extend the overlapping potential. In this Chapter, we give an overview of the structure of a modern compiler, and of some analyses and transformation processes that are useful in handling codes with MPI operations. We illustrate these notions on LLVM, the compilation framework we used in the present work, and explain how it can be expanded into an MPI capable optimizing compiler.

2.1 Overview of a modern compiler framework

This Section describes the structure of an optimizing compiler, and several key concepts useful to the analysis and transformation of code. More specifically, it describes the middle-end of a compiler, which allows source language independent analyses and transformations.

2.1.1 Architecture of a modern compiler

The primary goal of a compiler is to act as a translator. It *compiles* a *source* file into a *target* file. There are compilers that create human readable target files. They are called "source to source" compilers, or transpilers.

The whole compilation of a source file into a binary file can be split in four major phases. The first phase is the **preprocessing** step. It takes the source file, performs several transformations, and outputs another source file, which is still human readable. For example, a C preprocessor will expand macros and headers. The second step is the **compilation** step itself. From a source file, it creates an optimized and architecture dependent assembly file. The result can still be understood by a human, but it is one step closer to machine language. This machine language is generated during the third step, the **assembler**. The products of this step are also called "objects". They are eventually given to the **linker**, which is in charge of *linking* machine code files together to form an executable or

a library. A library is a binary file which can be *linked* to other objects, possibly in different projects, to provide some functionality[28].

The compilation phase itself can be split into three major steps: the frontend, the middle-end, and the backend. The frontend is in charge of checking the legality of the source code against the grammar of the language, and eventually applies corrections when possible and permitted (e.g. casting an operand into the correct type for an operation). It generates an intermediate representation (IR) of the program. This IR is independent of the source and target languages, and is passed to the middle-end. This allows the middle-end to define and apply source language agnostic analyzes and optimizations as passes. Section 2.1.2 provides an in-depth look on the IR and the middle-end. This IR is then given to the backend, whose goal is to apply architecture dependent and machine dependent optimizations, resulting in the assembly file. From nom on, we will focus on the middle-end.

2.1.2 The intermediate representation

The middle-end is one of the three major steps of the compilation phase. Before entering this phase, the code must be put under an intermediate representation (IR) which is generated at the end of the frontend step. This IR is generally independent of the source code. A compiler framework is constituted of multiple frontends, usually one for each source code language. For example, Clang is the most famous frontend of LLVM, and supports C, C++, and Ojective-C. It eventually generates an intermediary file in LLVM IR, the unique intermediate representation of LLVM.

The IR is the representation upon which the compiler applies multiple analyses and transformation routines, called *analysis passes* or *transformation passes*. Passes are similar to the phases defined by Wulf et al.[27]. Each pass takes as input an IR file in a given state, and outputs another IR file, potentially with some modifications compared to the input. A single pass (or routine) can be applied once or multiple times during the whole middle-end step. A pass can also generate results or information that can be processed by other passes. Consequently, a pass can be dependent on such information or transformations, and thus on other passes.

Three addresses code

IRs usually adopt the three addresses form for its instructions. As its name indicates, there are at most three addresses in a statement. Binary and unary operators abide to this form: on the right-hand side, there are either two (binary) or one (unary) addresses, and an operator, with one remaining address being

the left-hand side if needed. Conditional jumps can also be put under the three addresses form: two addresses being the operands of a relational operator, and the third being the location of the jump if the condition is fulfilled. Function calls can also be put under the three addresses form. While they might have many arguments, it is possible to translate a call site to this form by first pushing the arguments into a stack beforehand. The call site statement can now be translated as a three address statement: one address for the function, a second for the stack, and potentially a third to retrieve the returning value[28].

The major benefit of this form is the simplification of the code, which makes writing analyses and optimizations much easier.

Control Flow Graph

Statements in IR are grouped in blocks, called *basic blocks*. A basic block can have one or multiple successors, and one or multiple predecessors. Using this structure, any program can be represented as a flow graph, called the *control flow graph* (CFG).

Definition 1 (Basic Block)**.** A basic block is a succession of statements with only one entry point and one exit point. Once the first statement of the block has been executed, all other statements from that block must be executed in succession.

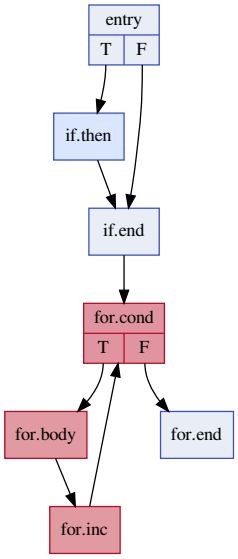
Definition 2 (Control Flow Graph)**.** A control flow graph is a flow graph $G = (V, E)$, where V is the set of basic blocks that make up the program. The edges E connect basic blocks together, and represent the possible execution paths.

Figure 2.1b shows the CFG of the function `main` in Listing 2.1a as generated by LLVM [1]. Each basic block corresponds to a sequence of uninterrupted statements. For example, the `entry` block contains the allocation of the variables, setting of `a`, the comparison and jump statements. This basic block stops at the jump statement because it creates a divergence in the execution path. The statement writing 4 to `a` will be executed depending on the result of the comparison. Therefore, there is a discontinuity in the succession of statements, and that statement has to be put in another basic block. There are two outgoing edges from `entry`, indicating that the control flow (or execution path) can either go through the `if.then` block, or directly to the `if.end` block. In this context, `entry` is also called a fork node, since the control flow splits after it, while `if.end` is the join node.

1. The source code has been first compiled to LLVM IR by clang with the `-disable-llvm-passes` option, then forwarded to opt which built the CFG with the `-dot-cfg-only` pass.

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv)
4  {
5      int a,b;
6      a = 1;
7      if (argc == 2)
8          a = 4;
9      printf("Hello world!\n");
10     for (b = 0; b < a; ++b) {
11         int c = b + a;
12         printf("c=%d\n", c);
13     }
14     return 0;
15 }
```

(a) main function



(b) CFG of the function main

Figure 2.1 – Representation of a function in its CFG form

A similar observation can be made about the blocks `for.cond`, `for.body`, and `for.inc`, whose subgraph constitutes a strongly connected component, or loop, in the CFG. It represents the `for` loop from lines 10 to 13.

The CFG and the three addresses code help in building analyzes and transformation on the IR. In this Section, we will provide an overview of some techniques and notions we will be using in this work.

2.1.3 Flow control analysis

The representation of the code of a function as a graph enables the analyzing of the its control flow. Techniques of graph theory can be applied, such as the detection of strongly connected component to find loops in the function. The notion of domination and post-domination is one of the key concepts.

Domination and Post-domination

Definition 3 (Domination)**.** Let $G = (V, E)$ be a control flow graph, with a single entry node $e \in V$ and a single exit node $o \in V$. A node $n \in V$ is said to be dominated by another node $x \in V$ if every path from e to n must contain x. The relationship is strict when $x \neq n$.

Definition 4 (Post-domination)**.** Let $G = (V, E)$ be a control flow graph, with a single entry node $e \in V$ and a single exit node $o \in V$. A node $n \in V$ is said to be post-dominated by another node $x \in V$ if every path from n to o must contain x. The relationship is strict when $x \neq n$.

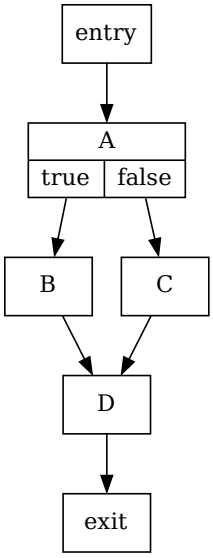
These two concepts are particularly useful in determining which basic blocks are bottleneck points for another basic block. In other words, for a given basic block, a dominating basic block is a block that will, in any circumstance, be executed before. These relationship can be represented in a tree. There is an edge from vertex x to vertex y in a dominator or post-dominator tree if x is dominates or post-dominates y. Figure 2.2b is the post-domination tree for the CFG shown in Figure 2.2a. For example, the edge $D \rightarrow A$ in Figure 2.2b illustrates the post-domination of A by D.

The definition of the post-domination requires a single exit CFG. However, any multiple exit CFG can be made into a single exit one by the addition of a global virtual sink which becomes the successor of all existing exit nodes. Figure 2.3a is the CFG of a function with branching paths, one of which containing a returning instruction, thus leading to an early exiting point. This virtual node does not match with any concrete source code instruction, and only exists to help us carry out the analysis. By linking the concrete exiting nodes to this newly inserted one, we obtain a CFG with only one exit node. The definition 4 can then be applied on this modified CFG. The resulting post-domination relations are still valid on the original CFG.

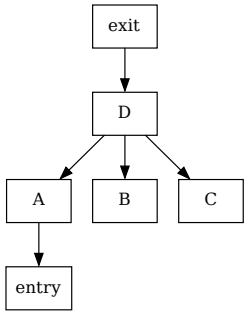
The notions of domination and post-domination can be generalized to cover dominating and post-dominating sets of some basic block[29]. The definition of a post-dominating set is as follows:

Definition 5 (Generalized post-domination)**.** Let $G = (V, E)$ be a control flow graph, with a single entry node $e \in V$ and a single exit node $o \in V$. A set of nodes $S \subset V$ post-dominates a node $n \in V$ if the following conditions are met:

1. All paths from n to the sink o must contain a node from S.
2. For each node $x \in S$, there is at least one path from n to the sink that contains x, and that does not contain any other node from S.



(a) CFG with an if/else branching



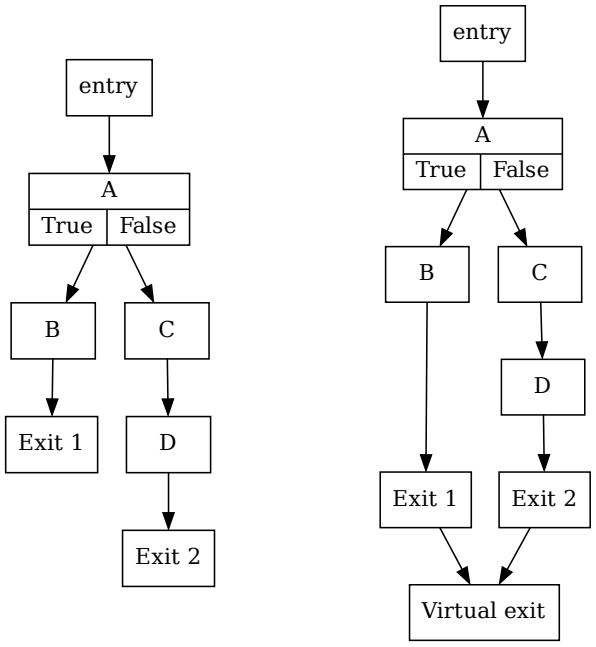
(b) Associated post-domination tree

Figure 2.2 – Example of post-domination in a simple CFG

The first criteria from definition 5 is a direct transposition of the single node post-domination to sets of nodes. Instead of having to flow through one node specific node, the generalized post-domination states that the flow can be intercepted by any node from a set. The second criteria is novel. It states that a post-dominating set must be minimal.

The notion of immediate dominance or post-dominance indicates the "closest" dominator or post-dominator to the considered vertex of a CFG. More formally, a node x is the immediate dominator of a node n if x is dominated by every other dominator of n. Every other dominator must be "between" x and the entry of the CFG. A similar definition can be given for the immediate post-domination. The notion can be extended to cover the dominating and post-dominating sets. The multiple-node immediate post-domination is defined as follows:

Definition 6 (Multiple-node immediate post-domination (mipdom)[29])**.** Let $G = (V, E)$ be a control flow graph, with a single entry node $e \in V$ and a single sink node $o \in V$. A multiple-node immediate post-dominator set of $n \in V$ is defined



(a) CFG with a returning instruction in a branch

(b) Addition of a virtual sink node

Figure 2.3 – Transforming a multiple exit CFG into a single exit

to be a subset of Succ(n) which post-dominates n.

In a similar fashion to the post-dominator tree, the multiple-node immediate post-dominators can be represented as a graph, called a post-dominator directed acyclic graph (PDDAG). It would contain all vertices and edges from the post-dominator tree, to which we add the sets of *mipdom*. If we look again at Figure 2.2b, we would add the node $\{B, C\}$, and an edge from it to A, informing us that this set is the immediate post-dominator of said vertex A. The determination of the mipdom of a vertex can be made with the help of the DJ-graph[30].

Definition 7 (DJ-graph[30] *adapted to post-dominators*)**.** Let $G = (V, E_g)$ be a control flow graph. Its DJ-graph $H = (V, E_h)$ is constructed using the same set of vertices V, and whose edges are composed of D-edges and J-edges. Let $e = (x \to y) \in E_g$, and $e^{-1} = (y \to x)$ the corresponding edge in the reverse CFG.

1. e^{-1} is a D-edge if y is an immediate post-dominator of x.
2. e^{-1} is a J-edge if y does not immediately post-dominate x.

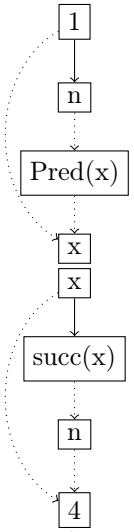
From the Definition 7, the DJ-graph for the post-domination relation can be built from the post-dominator tree. An edge $(y \to x)$ in the post-dominator tree represents the immediate post-domination of x by y, which is, by definition, a D-edge. The J-edge are added from the CFG. For each edge $e = (x \to y)$ of the CFG, we determine if y immediately post-dominates x. If this condition is not verified, then its corresponding edge e^{-1} in the reverse CFG is a J-edge. The algorithm for the determination of the mipdom of a vertex, and ultimately of all of its generalized post-dominators, is discussed in Section 4.2.

Dominance and Post-dominance frontier

From the domination and post-domination, the notions of domination and post-domination frontiers can be built[31]. The DF and PDF can be considered as the limits of the dominating or post-dominating effect of a node, hence the "frontier". The nodes beyond the PDF of a node n cannot be post-dominated by n.

Definition 8 (Dominance frontier, DF)**.** Let $G = (V, E)$ be a CFG with single entry and exit. The dominance frontier of $n \in V$ is the set $\mathrm{DF}(n)$ such as $\forall x \in \mathrm{DF}(n)$, n dominates a predecessor of x, but does not strictly dominate x.

Definition 9 (Post-dominance frontier, PDF)**.** Let $G = (V, E)$ be a CFG with single entry and exit. The post-dominance frontier of $n \in V$ is the set $\mathrm{PDF}(n)$ such as $\forall x \in \mathrm{PDF}(n)$, n post-dominates a successor of x, but does not strictly post-dominate x.



The concept of iterated frontiers extends the two previous definitions. It can be seen as the "frontier of the frontier", and its determination consists in recursively determine the DF or PDF until stability. The definition of the iterated PDF is as follows:

Definition 10 (Iterated post-dominance frontier, PDF^+)**.** Let $G = (V, E)$ be a CFG with single entry and exit. The iterated post-dominance frontier of $n \in V$,

noted $\mathrm{PDF}^+(n)$ is the limit of the sequence:

$$\begin{cases} \mathrm{PDF}_1(n) & = \mathrm{PDF}(n); \\ \mathrm{PDF}_{i+1}(n) & = \mathrm{PDF}(n \cup \mathrm{PDF}_i(n)) \end{cases}$$

Note that the PDF of a set is the union of the PDF of all vertices in the set.

2.1.4 Single Static Assignment

A code is said to be in the Single Static Assignment form (SSA) if a variable is only defined once, or only has a single assignment[32]. The translation of a code, usually from the three addresses form, to the SSA is done by versionning any variable that might be defined more than once in a program. Figure 2.4 is

```
1 a = b + c
2 d = a + c
3 a = d + b
4 b = a + d
```

(a) Three address form

```
1 a1 = b1 + c
2 d = a1 + c
3 a2 = d + b1
4 b2 = a2 + d
```

(b) Versionned SSA

Figure 2.4 – Translation into SSA

an example of the transformation of a three addresses code to its SSA form. The variables a and b are defined twice in this example. As a consequence, there are two versions for each of these variables, $\mathtt{a}_1$ and $\mathtt{a}_2$, and $\mathtt{b}_1$ and $\mathtt{b}_2$. Any use of a variable after its reassignment uses its newer version, as shown in line 4. This form helps in avoiding any doubt on the state of the variables at any point in the code.

It is possible that two versions of a variable conflict due to different execution paths. The code shown in Figure 2.1a is an example, with the variable `a` getting different values depending on the result of the comparison at line 7. In the SSA form, this behavior is modeled by the introduction of a special instruction called the ϕ function. It is inserted at join nodes where these conflicts can arise. Subsequent uses of this variable will refer to the result of the ϕ function instead. Cytron et al. introduced in 1989 a technique to quickly detect join nodes and thus to place ϕ functions using the dominance frontier[31].

As with the three addresses form, the SSA form is a step further in simplifying the code for analyses and transformations. With values that are only defined once in the lifespan of a program, it is much easier to analyze the dependencies. For example, from an instruction that uses some value as an operand, it is immediately

possible to determine the instruction defining the said operand: that is the use-define (use-def) chain. Inversely, from an instruction defining a value, it is also possible to immediately know the instructions using the said value as an operand: that is the definition-use (def-use) chain.

2.2 LLVM

LLVM is a compiler framework in development since 2003[33]. It is designed to "provide analyses and transformations for arbitrary programs at compile, link, and run times"[34]. At the time of writing, the fifteenth major version has just been released.

2.2.1 The compilation framework

LLVM adopts the structure of a modern compiler we described through this Chapter. It is composed of multiple frontends, with Clang for the C and C++ programming languages being the most notorious. Other third-party frontends also exist and are compatible with the LLVM framework, such as the Rust and Julia frontends.

From now on, we will focus on the compilation of a C program. To compile such code, we will resort to the Clang frontend. The source code is preprocessed and translated into an abstract syntax tree (AST). It analyzes, and detects any errors related to an incorrect use of the programming language, such as unrecognized keywords, mismatching types, and missing semicolons. Correct codes are translated to framework's unique intermediate representation, used by the middle-end: LLVM IR. Its human readable form (as opposed to its bytecode form) is a typed language in SSA form, and partially in three addresses form[35]. It is also able to attach metadata to statements. They usually contain debugging information such as the location of the equivalent instruction in the original source code file. They are available when the `-g` option is passed to Clang.

The output of the frontend, either in readable text or in bytecode, can be given to Opt. Opt handles the application of analysis and transformation passes on the IR. Users can select and reorder the passes to apply on an LLVM IR input, either through the predefined optimization levels `-O0, -O1, -O2, -O3`, or explicitly [2].

The LLVM IR files are eventually sent to `llc` which optimizes and translates them to an architecture dependent assembly file or binary object. Finally, the

2. List of available passes: `https://www.llvm.org/docs/Passes.html`

linking can be done by any linker, whether the GNU `ld` or `gold`, or LLVM's own linker `lld`.

Clang can act as a driver for the whole compilation pipeline. It is able to detect the input file type, and it determines the compilation phase at which it should pick up. For example, when given a source code file, it applies the whole compilation pipeline from the preprocessing to the generation of the object file, or until the desired phase that was given in the command line. If the input is an LLVM IR file, then it starts at the assembly code generation instead.

2.2.2 Analysis passes in LLVM IR

In this Section, we present some language independent analysis and transformation passes that are implemented in LLVM. They are typically applied on LLVM IR files with Opt. Besides the passes computing the dominator and post-dominator trees of a function, the recognition of loops, and the determination of the use-def and def-use chains, there are many analysis passes helping in determining instruction dependencies and transformations helping these analyses.

Pointers and alias analysis

A pointer is a special kind of variable storing the address of a memory location. Rather than the value itself, the value of a pointer is the location of the value. When an instruction accesses the "pointed" memory location to fetch the stored value, it is said that the pointer is being dereferenced.

Two pointers are *alias* if they are pointing to the same memory location. If an instruction dereferences one of the pointers and modifies the stored value, since the pointers are aliased, or point to the same place, subsequent dereferencing of the aliases will return the updated value. Yet, strictly speaking, no modification were made to these pointers, except the content of the referenced value. Because of this, pointers are often obstacles to analyses and safe optimizations.

To remedy to this, LLVM has implemented several techniques to detect aliasing pointers. Each of them gives one of these three possible answers: no-alias, must-alias, partial-alias, and may-alias. The first indicates that two memory locations are not aliased, while the second indicates that the two memory locations are aliased. The third indicates that they have a non empty intersection. The last answer is returned when the analysis is unsure whether the pointers alias.

However, the documentation on LLVM's available alias analysis passes is sur-

prisingly scarce [3]. Among the available passes, `basic-aa` is an alias analysis pass that is able to distinguish accesses to different fields of a structured type, distinct global variables and local allocations, or array accesses through statically distinct subscripts. We can also mention `globals-aa` and `tbaa`. The first analysis is fully dedicated to the disambiguation of global variables and is able to determine if a function accesses these global variables. The latter analysis stands for "type-based alias analysis". As its name suggests, it makes use of the type of the dereferenced pointers to determine if they can alias. For example, it is able to disambiguate to memory accesses if they are of distinct types[36].

The alias analysis passes implemented in LLVM can be chained, each one refining the results returned by the previous.

Memory SSA

It is possible to track the redefinitions of a variable in the LLVM IR thanks to the SSA form. However, it does not protect the accesses and modifications of the memory locations. In LLVM IR, only the value of variables are protected by the SSA, the actual location of the pointers is not: it is only applied to the value of the pointer, which is the address it holds. To track modifications of the memory locations, LLVM provides a virtual IR with an SSA form for these accesses: MemorySSA (MSSA)[37]. The MSSA form is built using the regular SSA form and the results of the alias analysis passes. In this form, each memory accessing instruction is mapped to one of the these three possible operations: `MemoryUse`, `MemoryDef`, and `MemoryPhi`. `MemoryUse` will be associated to an instruction which simply reads into the memory location without modifying it. Typically, they will be mapped onto `load` instructions. `MemoryDef` represents instructions that will potentially modify memory, such as `store`. A defining operation will create a new value in the MSSA. These values represent a state of the memory, and obey to the same rules as regular SSA values. Finally, the `MemoryPhi` behave similarly to ϕ-functions. They are inserted at join nodes where multiple defining operations can conflict. Using the MSSA form, it is possible to compare if two memory accesses will refer to the same version of the memory.

Scalar Evolution

The Scalar Evolution (SCEV) is a symbolic analysis centered around the recognition of induction variables and how they evolve[38]. It relies on the computation

3. LLVM's Alias Analysis documentation page: `https://www.llvm.org/docs/AliasAnalysis.html`

of chains of recurrence[39] of induction variables. This mathematical object allows more efficient computations of the values of a closed form mathematical expression.

More specifically, loop counters and induction variables are closed form expressions. An induction variable is being updated by a fixed amount at each iteration. The update is an arithmetical operator, but it is generally an addition or a difference. Consequently, for an induction variable evolving in a linear fashion, the SCEV analysis is able to compute its boundaries, the operator used for computing its value at each iteration, and the step value. In a loop, its counter is an induction variable. Thus, it is available for analysis by the SCEV, and it is able to determine its properties, such as an upper bound for the number of iteration. Giving a more precise estimate is difficult since loops can contain early exit points, such as `break` instructions.

2.2.3 Transformation passes in LLVM IR

Some of the analyses we presented in the previous Section require the code to be formatted in a specific shape. SCEV in particular can only be applied to natural loops, and requires them to be modified in a specific fashion, thus it depends on the following transformations: `sroa`, `loop-rotate`, and `loop-simplify`.

`sroa` stands for Scalar Replacement Of Aggregates. It breaks the local memory allocation instructions of a structured type variable into smaller allocation instructions of each element of the structure. `loop-simplify` homogenizes the form of loops. After application of that transformation pass, all loops have one preheader, a single backedge (an edge from inside the loop to the header), and every exit block (a basic block that does not belong to the loop, but has a predecessor in) must be dominated by the loop header. It simplifies the analysis of the loop, since it will only have one predecessor (the preheader) and one "looping back" point. Finally, `loop-rotate` depends on the previous transformation. As its name suggests, it "rotates" `while` and `for` loops into the `do-while` form. It also adds a guard block, protecting entry into the loop if, for example, the condition in the `while` is not satisfied, ensuring the preservation of the program's behavior.

Among the other notable transformation passes which we will use in our work, `instcombine` simplifies the IR by, for example, removing redundant instructions. `simplifycfg` performs several modifications of the CFG, including the merging of successive basic blocks that are neither a fork or a join (i.e. the first has only one successor, and the next has only one predecessor), or the removal on unreachable basic blocks. `lowerinvoke` removes any kind of exception handling.

2.2.4 LLVM plugins

The features of LLVM can be expanded thanks to plugins. They allow the definition of, for example, a new analysis or transformation pass to be applied on the IR. These new passes can be inserted at specific points in the pass manager, which handles the order of the middle-end passes. Plugins allow us to write new code for the compiler from an independent directory, rather than directly into the LLVM project directory. It is easier to maintain the project and to share it, for example as an open-source project.

Since version 14, LLVM has completely switched to a newer pass manager, and what follows here is based on the legacy pass manager which was deprecated in that release. Plugins are dynamic libraries that can be loaded by Clang or Opt. Upon opening the dynamic library by either of these entry points, the passes it defines are made known to the pass manager. Their name, dependencies, and impact on the IR and other analysis passes are registered. The new passes can be inserted in the optimization pipeline at predefined moments, for example when the middle-end begins the analysis of the module, thus allowing interprocedural analysis in the scope of a translation unit, or when it analyzes a function. The passes are then applied as part of an optimization level. When loading the plugin library and calling the pass with Opt, it is possible to insert the passes with a finer grain. Opt allows the selection and the ordering of the passes to apply on an IR file. For example, the command `opt -load plugin.so -loop-rotate -my-pass` will call the plugin inserted pass `my-pass` after the loop rotation pass.

While most of the key concepts described in this Chapter are already part of LLVM, several notions, such as the DJ-graphs and the post-domination frontier, are not implemented, or have been deprecated and removed from the framework. Thanks to the plugin mechanism, it is possible to expand the capabilities of LLVM. It is possible to introduce new objects, analyses and transformation methods, such as passes that verifies and optimizes MPI codes during the middle-end.

Chapter 3

Literature Review and Problem Statement

In pursuit of better performance and parallelism, the number of NUMA nodes within a cluster, have been increasing. As a consequence, more and more communications are needed between these nodes. While the evolution of interconnect solutions leads to better throughput, communications can still have a negative impact on the time to obtain results from a simulation. Causes for the degradation of performance can include the synchronization of processes before an exchange can be performed, congestion on the network, for example a bottleneck on some specific nodes due to an inadequate communication algorithm, or because of the network distance between nodes. The time a computing unit is spending in waiting for a data to arrive or depart before it can safely access it, is time a computing unit is not investing in performing computations.

Resorting to asynchronous communications, such as the nonblocking primitives defined in MPI, helps computing units to fully utilize their computing potential. These calls enable the overlapping of communication times by computations. By being "nonblocking", the initiation calls allows the MPI process to prepare for the communication, and delay the blocking, or waiting, part until later in its execution flow. Meanwhile, the MPI process is free to work on other computations, and the communication has the possibility of being progressed by another entity, for example a dedicated execution thread or on a capable network interface card.

In 2017, a large survey aimed at developers of MPI applications for exascale class supercomputers was carried to understand their views on the interface and its critical aspects for performance. It outlined that, while 80% of the surveyed developers use point-to-point nonblocking primitives to allow overlapping in their

applications, "only" 59% use nonblocking collectives for that purpose[40]. Another study conducted in 2019 on the code itself of 110 MPI programs also noticed that, while point-to-point nonblocking calls are more prominently used than their blocking and persistent counterparts, only a few applications use nonblocking collectives[41].

A decade after their introduction to the interface, nonblocking collective primitives are still not as popular as their point-to-point cousins, despite their benefits on increasing the performance of MPI applications. The reasons for this unpopularity are multiple.

First, they are more complex to handle because of their form. The initiation call not being a blocking call, it leaves the communication buffers at the mercy of the overlapping computations, which can cause data races. Splitting the communication in halves also implies the necessity of a matching between the calls, and the preservation of the linking element (the request). Contrary to blocking communications, it is of the developers' responsibility to ensure the absence of race conditions and of deadlocks, which are the consequences of an improper use of communications buffers and of a mismatching of nonblocking initiation and completion calls.

Moreover, the mere existence of an overlapping interval does not guarantee the actual overlapping of the communication time. There must be a proper progression mechanism allowing the concurrent handling of the communication, eventually leading to a better use of the computing resources. A recent study from Denis et al. highlighted the struggles of multiple implementations of the interface, including OpenMPI, MPICH, or MVAPICH, to correctly progress nonblocking communications[42]. Compared to blocking communications, the use of nonblocking operations, and more specifically of nonblocking collective calls, with the default configuration of these libraries shows no performance gains in most cases, and even losses in some cases. Only libraries supporting an explicit progression mechanism are able to expose performance gains by overlapping communications. These mechanisms are dedicated computing cores[43], or the offloading of the progression to the networking hardware[44][45].

Combined with their complexity, the lack of guarantees on the performance gains, and even the risk of degrading the execution times when using an ill-configured environment, might deter developers from using nonblocking communications. Yet, this form of communication enables the possibility of a better use of the computing resources. By not resorting to nonblocking communications, programs miss an opportunity to fully exploit the capabilities of a cluster, which is necessary to reach the exascale scope in an environment where the Moore's Law is

starting to buckle. As a consequence, the main objective of this study is to develop solutions to help and encourage developers of MPI programs in using nonblocking communications. To that end, we have identified two possible leads.

The first would be to check MPI programs to detect programming errors. Having a debugging solution helps in producing codes of better quality, and encourages the adoption of some language feature. It directly addresses the complexity of nonblocking calls, and relieves the burden put upon the developers. Such a solution would detect errors and help them in correcting their code. To that end, we propose in Chapter 5 a compile-time verification tool that focuses on the detection of misuses of nonblocking communications. The tool we propose is applied at the level of the IR, and aims at finding and alerting the developers about deadlocks caused by mismatching initiation and completion calls or mismanagement of the requests, and race conditions caused mishandling of the communication buffers.

The second lead is the creation and the expansion of the overlapping of nonblocking communications in MPI programs. In this study, we focus on the "overlapping potential". The "potential" part derives from the lack of guarantees for an "actual overlapping" of a communication, which would only be allowed by the presence of a concurrent progression mechanism. Rather than operating at the level of the implementation to allow concurrent progression of communications, we operate at the level of the application to express opportunities for the overlapping. In fact, these two approaches are complementary, and equally necessary to achieve actual overlapping. With existing studies on the lower level approach[43][46], our goal is the maximization of the overlapping potential of an application. This aim contributes in helping the developers in using nonblocking communications and in adapting their code to allow overlapping. In order to tackle this approach, we propose in Chapter 6 a scalable solution to automatically create and expand overlapping intervals at compile-time. Chapter 7 proposes a feedback-based approach to further expand overlapping opportunities by guiding the developers in adapting their code. The goal of these two propositions is to reorganize the code, while preserving its correctness, to maximize the overlapping potential.

The following Sections of this Chapter describes how existing studies tackled both of these approaches. First, Section 3.1 focuses on the verification tools for MPI nonblocking communications, and Section 3.2 presents solutions to create overlapping potential. Finally, Section 3.3 concludes this Chapter with a discussion on the State of the Art of the analysis and of the transformation of nonblocking primitives, and presents how we address their limitations in our contributions.

3.1 Verification of MPI communications

Verifying the correctness of a code helps the developers during the development cycle of a program. With a correct and informative feedback, an error detected by a verification tool allows the developers to fix their code so that it produces the expected result without interrupting or damaging its environment which can be caused by, for example, a memory leak. The methods and tools described in this Section aim at helping the developers of MPI programs by detecting errors. This encourages the developers to make use of most of the features of the interface such as the nonblocking collectives.

3.1.1 Dynamic approaches

One popular solution to check the correctness of codes is to perform a dynamic analysis. These solutions analyzes the program during or after its execution.

Umpire[47] is an example of a dynamic verification tool. It makes use of the profiling interface of MPI[19], which allows developers to peek at each MPI call to fetch information on the communication or perform some action, before executing the desired MPI operation. It is able to detect deadlocks caused by mismatching collective communications, but also of modifications to an outbound buffer inside the overlapping interval of a Send operation, misuses of the derived types, and the overwriting of an active `MPI_Request`. However, Umpire has a significant impact on the performance of the instrumented program, with a slowdown up to 49% in one of the benchmarks tested by the authors, which appears to be increasing with the number of MPI calls.

Marmot[48] is another similar tool to Umpire, and exploits the profiling interface of MPI. It detects mishandling of the communication resources: validity of a communicator and of a group, of derived types and operators, the range of a desired MPI rank, and the overwriting of requests. Marmot also detects deadlocks by implementing a time-out mechanism. If an MPI process waits for an operation for a duration that exceeds the time-out duration, the verification tool emits a warning message and useful debugging information to the developers. Marmot faces similar challenges as it introduces overhead to the targeted program.

MUST[49], which stands for "Marmot Umpire Scalable Tool", is built upon the two previous solutions. It performs deadlock detection by building a wait-for graph which depicts the scheduling dependencies between MPI processes. This graph allows MUST to reduce the overhead of instrumenting the MPI calls at execution time. It is able to detect deadlocks caused by the mismatching of point-to-point, collective, and nonblocking calls. The authors succeed in decreasing the overhead

introduced by dynamic tools, the errors are only detected after an execution of the program.

The Intel Trace Analyzer and Collector (ITAC)[50] is a tool to profile and analyze MPI programs. As it name suggests, it traces the behavior of the program, thus it requires its execution. While designed to analyze and display performance information of MPI communications, it is also able to detect deadlocks, race conditions, and misuses of communication resources.

Overall, dynamic verification solution have the best accuracy and coverage of misuses of MPI nonblocking communications. However, by definition, errors are only reported once they occurred. Thus, they require a preliminary execution of the targeted program. Moreover, dynamic tools rely on the injection of code on MPI calls to analyze their behavior, which can lengthen the execution times of these programs. Finally, the detection of errors depends on the inputs used during the verification run. It would potentially require multiple runs with different inputs to analyze the entire code base. All of these characteristics are undesirable in an HPC environment, where there might be quota on the usage of computing resources for each developer, and where programs might require several hours or days to run.

3.1.2 Static and static-dynamic hybrid approaches

This Section discusses verification tools that does not only rely on the execution of the actual program. Some solutions are composed of a dynamic analysis component, making them static-dynamic hybrid approaches.

Model checking

In the model checking approach, the analyzed programs are modeled as a state transition graph. This representation is given to a verification tool, along with the properties to check, expressed using temporal logic. The verification tool automatically checks if the model satisfies the property[51]. In our situation, an example of a property could be "An MPI nonblocking initiation call is always followed by a completion call", and the model would be based on the code of an MPI program.

MPI-Spin[52] is a verification tool based on the Spin model checking engine. It is able to detect deadlocks caused by a misuse of nonblocking communications such as mismatched calls or mishandling of the requests. Nonetheless, their model has to make several assumptions which limit its applicability to many MPI programs. As part of these limitations, it only considers the `MPI_COMM_WORLD` communicator.

Moreover, the developers need to create a model of their program so that MPI-Spin can analyze it.

SimGrid MC [53] is another model checking tool, based on SimGrid, It relies on dynamic partial order reduction techniques to reduce the number of states to explore. It checks for safety, and liveness properties, and it is able to handle undeterministic communications such as an `MPI_Waitany`. Examples of detected errors are mishandling of requests, or race conditions on the communication buffers[54]. The authors confronted their tool to the MPICH3 test suite, and they observed a large memory consumption, from several megabytes to more than a dozen of gigabytes. The analysis time also ranges from a few minutes to several hours.

The model checking approach struggles with large programs, whose models are complex and large in size, because of the explosion of possible states. Despite attempts at reducing the number of states to explore with techniques such as the dynamic partial order reduction, such tools still face long analysis time and resource consumption.

Symbolic execution

One of the popular solutions to test programs is through symbolic execution. The program is not actually executed for a specific set of inputs. Instead, it consists in evaluating a program with symbolic values (for example, a mathematical variable) to explore the possible control flow paths, and determine the validity of the program after each path[55].

CIVL [56] and MPI-SV [57, 58] both combine symbolic execution with model checking. CIVL translates the source code to an intermediary language which is then sent to a symbolic execution and model checking backend. The translation phase supports the major parallel programming libraries and codes using a combination of them: MPI, OpenMP, CUDA, or Pthreads. As far as MPI is concerned, CIVL is able to check the compatibility of message metadata, the absence of deadlocks, data races, or the matching of undeterministic communications. However, CIVL cannot handle nonblocking communications.

MPI-SV, on the other hand, can handle nonblocking calls, and focuses on the detection of deadlocks and undeterministic communication patterns. The use of model checking allows them to reduce the number of paths to explore by the symbolic execution process. The detected misuses of the MPI interface include the overwriting of requests, and unmatched completion calls, but it does not detect race conditions on communication buffers.

Ye et al.[59] developed a tool that uses partial symbolic execution to detect MPI usage anomalies, based on the Klee symbolic execution engine. The source

code is first compiled to LLVM IR, where several analyses are performed. This representation is provided to Klee, along with the results from the static analyses, to help with the symbolic execution of the program. Their solution checks for the correct matching of message types, race conditions on the message contents, the overwriting of an active request, an uncompleted nonblocking communication, and the mismatching of point-to-point communications. Despite the large coverage they offer, their solution has a high memory footprint and a significant analysis time on several benchmarks.

The approaches based on symbolic execution suffer from similar limitations to the model checking approaches, as the number of symbolic values might significantly increase with the size and number of paths in a program. This results in time and memory consuming verification tools.

Compile-time verification

Compile-time solutions are applied during the compilation of the program to analyze. They do not require any preliminary execution, neither of the actual code or a modeled version of it. Multiple entry points are possible during the compilation of a program: during the frontend to directly analyze the source code, as an IR pass to analyze code regardless of the source language, or at link-time to have a more complete vision of the program.

MPI-Checker[60] is based on the Clang Static Analyzer. It operates at the frontend level, and is composed of two analyses. The path-sensitive analysis is based on symbolic execution techniques to detect the possible paths in a function, and to determine the conditions for their execution. This analyses allows the detection of request overwriting, and mismatched nonblocking communications. The other analysis relies on the AST, the representation under which the code is analyzed in Clang, to detect mismatched point-to-point communications, and several misuses of message metadata. MPI-Checker is, however, unable to detect race conditions. Furthermore, it operates at the front-end. As a consequence, it cannot be easily interfaced with other analyses passes to benefit from the cached results, and cannot analyze code written in languages not supported by Clang, such as Fortran which is still being used in HPC applications.

Hybrid approaches combine a static analysis with a dynamic one. This allows such solutions to benefit from the low overhead of static analyses with the accuracy of dynamic tools. PARCOACH [61, 62] is one of these solutions. It focuses on the detection of deadlocks caused by mismatching collective operations. PARCOACH raises warnings for potential errors with precise debugging information, such as the location of the fork node leading to different sequences of collective

communications. The static phase, operating as an IR pass, is completed by an instrumentation of the potentially faulty communications. The dynamic phase properly terminates the program and provides useful feedback if the error flagged at compile-time is actually a true positive. PARCOACH is mainly focused on the ordering and matching of collectives. In [63], an extension was proposed to incorporate a light analysis that checks if each nonblocking initiation can be matched with a completion call. This check was done by counting the number of initiation and completion calls on each path of the program and did not consider requests, rendering it inaccurate.

Compile-time verification methods are the most scalable analysis approaches. They generally introduce low to moderate overhead to the compilation time, and they do not usually impact the execution performance of the program, since it is not modified, and they do not have to rely on a specific set of input data. Compile-time tools can be directly integrated into the compilation of the program. However, they lack information that are only available at the execution of the program, such as the number and the distribution of the MPI processes, or the nature or range of values of the input data. These weaknesses are source of inaccuracies in the analysis[64][54].

3.2 Optimizing the overlapping potential of MPI codes

Besides the detection of misuses of nonblocking communications, transforming and adapting the code to favorise the overlapping of communications is another approach to encourage developers to use MPI nonblocking operations. The existing solutions described in this Section aim at helping the developers to enhance the performance of their MPI programs by overlapping communication times with computations. They also aim at minimizing the required effort to adapt the codes.

Enforcing progression inside the overlapping interval

One solution to ensure the progression of a nonblocking communication is to periodically call MPI functions, for example, `MPI_Test`. Depending on the implementations, it might force the MPI runtime to progress the communication, thus enabling overlapping.

Hoefler et al. proposed an optimization of a conjugate gradient solver using nonblocking collective calls[65] with LibNBC[66], a library that adds nonblocking collectives on top of the MPI-1 interface. They made periodic calls to `NBC_TEST`,

which is equivalent to an `MPI_Test` inside the overlapping interval they manually created to progress the `NBC_IALLTOALLV` nonblocking communication initiation. The authors observed an increase in the performance of their solver when using their custom implementation of nonblocking collectives with the addition of progression calls. However, they did note a degradation when there are less than eight MPI processes, which might be caused by the introduction of these progression calls.

Song et al. proposed a 3D Fast Fourier Transform algorithm using MPI nonblocking communications. Their algorithm is further improved with the help of an auto-tuning solution which determines the optimal values for several parameters, including the number of calls to `MPI_Test` to force the progression of nonblocking communications. Having too many progression calls might lead to a strong overhead, while too few would limit the progression.

Enforcing the progression of an MPI communication by calling MPI functions is a form of *weak* progress. It is opposed to *strong* progress where these calls are not necessary, and the progression of the communication is, for example, concurrently performed on a dedicated thread. Weak progression is not desirable. It introduces function calls that does not have any significance in the algorithms, and that instead add avoidable overheads[67]. Going forward, strong progression might become more and more prominent as the research on progression cores and offloading advances.

Creation and extension of the overlapping potential

Another solution to improve the overlapping of communications by computations in a code consists in exposing as much potential, as in computations to hide the communication, as possible, regardless of the presence of a progression mechanism.

Danalis *et al.* an MPI-aware compiler that is able to create communication-computation overlapping possibilities[68]. They propose the transformation of blocking calls into their nonblocking counterparts, the decomposition of collective calls into point-to-point ones, the application of code motion, variable cloning, and loop tiling and fission to increase the overlapping window. However, their approach has not been implemented, and their method had only been manually applied to several examples.

ASPhALT implements a subset of those optimizations as a compilation pass for the open64 source-to-source compiler [69]. It aims at optimizing producer-consumer loops by performing prepush transformations, meaning that it will try to send the data as soon as it is generated so that consumer computation can

be performed while the next chunk of data is being produced. The producer-consumer loop is partitioned with an arbitrary size to control the amount of data that is generated, shared and computed. Their solution is only able to handle this sole communication pattern.

Guo *et al.* showed how to improve this approach by adding a performance analytical model of the application [70]. With the help of user-added annotations and information such as the input data and the number of MPI processes, they build a model of the program to predict performance. Based on the results of this model, their tool decides when the transformation of blocking calls into nonblocking ones becomes worthy depending on the amount of independent computations surrounding it. The transformation itself and the code motion are still manually done.

Das *et al.* proposed an approach based on a Wait Graph to sink the completion call of nonblocking communications [71]. Their goal is to move the completion call at a later point in the execution. This graph contains information about the control and data flow, enabling them to sink the wait call to the nearest statement that uses a communication buffer whilst remaining in the original control flow scope. They have also proposed an algorithm to go beyond the limits of the encompassing scope, but it had not been implemented. Only the completion calls are subjected to the code motion, therefore this approach would most likely miss opportunities that might exist above the initiation call.

Petal [72] is an interprocedural compiler pass implemented within the ROSE[73] compiler moving completion calls to the nearest dependency point. Ahmed *et al.* used an alias analysis to detect whether a statement uses a communication buffer. Their method transforms nonblocking communications into persistent communications when they are nested inside a loop. Lacking the ability of matching nonblocking communication calls together, only the existing blocking communications are considered and transformed. Eventually, the expansion of the overlapping interval is only performed by moving the completion call.

The existing and implemented solutions for the creation and expansion of the overlapping potential of MPI programs consist in identifying the dependencies, related to both the control and data flow, then in displacing the nonblocking communications. However, they all face different limitations, and they either focus on a specific communication pattern or only consider one of the nonblocking calls. Furthermore, they only have the vision on the closest dependency, which they consider as the insertion point of the nonblocking call. This prevents these approaches from fully exploiting the overlapping potential that might be available in a program, and that might exist beyond these dependencies.

3.3 Problem statement

On the verification side, a compile-time solution is most suited to analyze the large codes that can be found in an HPC context, despite their inaccuracies. They require little to no intervention from the developers, and can be easily integrated into the development cycle of an MPI application. The existing works in this regard are either unable to catch race conditions in the overlapping interval, or do not have a reliable solution to match nonblocking communications. We address these points in Chapter 4 and Chapter 5, by proposing a verification tool that operates as an IR pass to detect mismatching nonblocking calls as well as race conditions.

Among the existing compile-time verification tools, MPI-Checker[60] is the closest solution to ours. They handle many errors caused by nonblocking communications, but their analysis is performed during the front-end. This makes it impossible to analyze MPI programs written in other languages, such as Fortran, and to interface with other source independent analyses or optimization passes. Furthermore, they do not check for race conditions. All of these limitations justify the proposition of a compile-time verification method on the IR to detect misuses of nonblocking communications and the race conditions they might cause.

On the optimization side, some of the existing solutions focus on specific communication patterns, and all of the implemented works are limited in their abilities to transform the code. Their efforts focus on the completion call, and stop at the closest dependency. Our contributions allow the bypassing of the data dependencies, allowing us to reach the overlapping potential that might reside beyond the limitations of the existing solutions. Operating as an IR optimization pass, it is also able to benefit from the results of the verification pass to improve the overlapping potential of existing nonblocking communications. Chapter 6 introduces this automatic optimization pass, and Chapter 7 describes a feedback-based approach to further improve the results obtained by the automatic pass.

Petal[72] is the implemented optimization method that shares the most similarities with ours. However, their transformation method faces the aforementioned limitations. In contrast, our contribution allows the optimization of existing nonblocking calls, and aims for a complete reorganization of the code to favorise its overlapping potential. MPI-Aware[68] described several transformations that are similar to those we implement, especially in identifying the independent statements in a function to move them into the overlapping interval. Yet, these transformations were manually performed, and to our knowledge, no other work attempted to implement those.

To summarize, our contributions, a compile-time verification pass, a compile-time code transformation pass, and the provision of suggestions of code modifications, constitute a cohesive system. The verification pass validates the code and provides matching information for the transformation pass. In turn, this transformation pass analyzes the overlapping potential of a function, and automatically adapts the code. From the results of these analyses, of the performed modifications, and from the boundaries of the overlapping intervals, suggestions of code modifications can be generated to let the developers decide whether they should further adapt their code, which can be then checked by the verification pass. This ensemble aims at accompanying the developers of MPI programs during the whole development cycle to help in using nonblocking communications.

Part II

Contributions

Chapter 4

Static Analysis of Nonblocking Communications

Chapter 3 presented the various approaches to analyze MPI codes. Dynamic solutions require the execution of the code. While accurate, it is not ideal for HPC codes, which are meant for long execution times that ranges in days or weeks. Moreover, these solutions tend to introduce overhead in the code, further slowing their execution time, and will most likely not analyze all the code in one run. Model checking and symbolic execution approaches are more suitable than dynamic solutions thanks to their ability to cover the entirety of the code. However, they can be complex and costly to apply.

Analyzing and transforming the code during its compilation presents many advantages in terms of scalability and ease of use. Such tool naturally fits into the development cycle of a program that uses a compiled language. Furthermore, almost all HPC applications are written in a compiled language: C, C++, or Fortran. Therefore, the verification and optimization methods for MPI nonblocking communications we propose in the coming Chapters are all compile-time solutions which can be applied to the majority of existing and future programs.

In this Chapter, we discuss the adequacy and the adaptation of the notions presented in Chapter 2 for the analysis of MPI programs. Its goal is not to (re)demonstrate the veracity of the properties, theorems, and algorithms we base our work upon, but rather to illustrate how they apply to the analysis of nonblocking communications. Section 4.1 describes how an analysis of the data flow can be used to detect completion calls and data dependencies of a nonblocking communication. Sections 4.2 and 4.3 discuss the static matching of nonblocking calls with an analysis of the CFG, and of the possible execution paths. Finally,

Section 4.4 synthesizes these two approaches to the matching of nonblocking calls by defining the mandatory set, and concludes this Chapter.

4.1 Data flow and nonblocking communications

Analyzing the data-flow of an MPI communication is crucial for its verification and for the optimization of its overlapping potential. It allows the determination of the dependencies of the communications: all statements that require a result from the communication, and all statements that define a value used in the communication. These dependencies not only allows the detection of race conditions, but also include the request object, which serves as a link between the initiation and completion calls.

4.1.1 Identification of potential completion points

The calls defining an MPI nonblocking communication are tied together with an object called the "request". This object is given to nonblocking calls through their argument list. Knowing their position in this argument list allows us to identify the request being used by each call. For example, initiation calls always expect to have a request at the tail of the list. By accessing to this position, we fetch the request an initiation call will set. Completion calls have a slightly different behavior, and we have to study two distinct situations.

Single pointer request completion calls

Single request completion calls are easier to handle, since they have a similar behavior to initiation calls, and will involve only one pointer to a request object. Single request completions, `MPI_Wait` and `MPI_Test` have their request in the first position of their argument list, thus accessing it is immediate.

If the accessed request is from an array of requests, indexes will have to be compared in order to confirm that the same object is being used both in the completion and in the initiation calls. To do so, the dependency graph of the request is built, using its use-def chains. This graph contains every instruction defining and using the request.

In LLVM-IR, accesses to an element in an array or structure is done with the help of a `getelementptr` instruction (GEP). It performs addresses calculations, and the result is given to a `load` instruction, which effectively accesses the memory zone. Starting from the users of the request, either an initiation or a completion

```
1 MPI_Waitall(int count, MPI_Request req_array[], MPI_Status status_array[]);
```

Figure 4.1 – `MPI_Waitall` prototype

```
1 /* ... */
2 MPI_Irecv(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &req_array[2]);
3 /* ... */
4 MPI_Waitall(count, req_array, status_array);
```

Figure 4.2 – Undecidable matching with an `MPI_Waitall`

call, the GEP can be retrieved by iterating over the use-def chains. Consequently, it is possible to fetch the indexes used to access the request, and compare the used memory zones to know if a completion is able to catch an initiation.

Completion calls taking an array of requests

The completion calls that accept an array of requests in their arguments list are more complex to handle. The `all`, `any`, and `some` versions of both the waiting and testing completions fall in this situation. The length of the array is arbitrary, and in most cases it is unknown at compile-time.

Listing 4.1 shows the prototype of the Waitall completion call. The `count` argument gives the number of requests expected by the completion call. The second argument, `req_array`, is the starting address of a continuous array of requests. This means the Waitall will expect `count` requests starting at the address given in `req_array`. Unless the `count` value can be statically determined, we cannot know for sure if a request is going to be completed by these calls. Figure 4.2 showcases such situation. The nonblocking receive sets the 3^{rd} request, assuming the request array is of size greater than 3. If the value of `count` cannot be determined at compile-time, then there is no information on whether this Waitall call is able to complete the request that was set by this receive operation. As a consequence this completion call cannot be associated with the nonblocking receive.

In this study, we focus on single request completion calls, and on some situations where the number of initiated and expected requests is known, or comparable. In such situations, it is possible to know, for a given initiation call, if a potential completion call is able to catch the initiated request.

4.1.2 Identification of the dependency slice of a communication

Contrary to blocking communications, nonblocking calls do not offer protection of the communication buffers, and these are left exposed for the duration of the exchange. Depending on the nature of the communication, illegal access to those can result in faults in the results. This justifies the need of the identification of the communication buffers and their dependencies to verify if no illegal access have been committed inside the overlapping interval. Furthermore, analyzing the dependencies of the communication will help in characterizing the overlapping potential of a function.

The identification of communication dependencies relies on the same principles as the detection of the requests described in Section 4.1.1. Requests are, in fact, a specific kind of dependency which allows the association of nonblocking calls.

A dependency graph is constructed too, starting from the arguments of the communication and iterating over the use-def and def-use chains. This graph contains every instruction that recursively use or define a communication argument.

4.2 Control flow and nonblocking communications

Analyzing the dependency graph of the request of a nonblocking communication will provide us with a list of all instructions using the said request, including the potential completion points. In order to define an overlapping interval for this nonblocking communication, it is necessary to precisely identify which completion call, or subset of completion calls, are effectively completing the communication. In a simple linear code without any loop or branching, an initiation call is completed by the first completion call which comes after it in the execution path, assuming both refer to the same request.

Real world codes are, unfortunately, not so simple, and contain branches, loops, and sometimes multiple exit points. By analyzing the CFG, it is possible to determine which completions is able to complete an initiation. We qualify such analysis as "flow sensitive", since we are analyzing each potential flow, or path, of execution in a CFG.

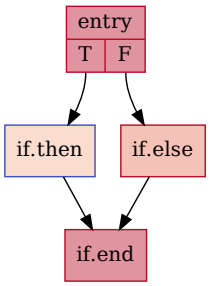
Post-domination by a single node

The post-domination (see Definition 4) helps in determining if a node is on the way to the exit. Any nonblocking communication must be terminated before

the call to `MPI_Finalize`. Assuming the `MPI_Finalize` is at the exit node of a program, the concept of post-domination allows us to determine if a nonblocking communication is safely completed.

```
1  int f(int arg)
2  {
3      int a = 10;
4      MPI_Request req;
5      MPI_Ibcast(&a, 1, MPI_INT, 0,
           MPI_COMM_WORLD, &req);
6      if (arg == 0){
7          printf("Hello!\n");
8      } else {
9          printf("World\n");
10     }
11     MPI_Wait(&req, MPI_STATUS_IGNORE);
12     return a;
13 }
```

(a) MPI code with branches



CFG for 'f' function

(b) CFG of function in Figure 4.3a

Figure 4.3 – Example of post-domination in a simple CFG

Figure 4.3b shows the CFG of the function `f` in Figure 4.3a. The `MPI_Ibcast` at line 5 in the source code file, and in the basic block "`entry`" in the CFG, can be completed by the `MPI_Wait` at line 11 and in the block `if.end`. To determine if it is safely completed, we must determine if there is a possibility for the initiation call to avoid the completion call. In the example, the `entry` block is post-dominated by `if.end`, since all paths from it to the exit of the CFG must contain `if.end`. As a consequence, there is no possibility for the `Ibcast` to avoid the completion call, thus ensuring the matching of the nonblocking communication. However, if the completion call were to be in the `if.then` block, then there is a path, through `if.else`, bypassing the completion call, thus there is no post-domination and the communication is unsafe.

To sum up, we can directly apply the definition 4 to nonblocking initiations and completions. Let us consider x a basic block containing a completion call, and n the initiation call, and assume these calls use the same request. The post-domination of n by x ensures the completion of the communication. Every path from the initiation call (n) must contain the completion call (x) before reaching the sink. Thus the communication is properly terminated. The absence of post-domination of n by x indicates the existence of a path bypassing the completion

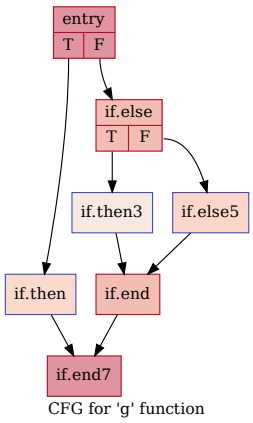
call, which might prevent a communication from being correctly completed.

Post-domination by multiple nodes

It is possible for a nonblocking initiation to be caught by multiple completion calls. For example in the Figure 4.4a, the nonblocking broadcast communication is correctly completed. Regardless of the value of `arg`, each path contains a completion call. Figure 4.4b is the CFG of function `g`. The initiation call is in node `entry`, while the completion calls are in `if.then` and `if.end`. If the communication is properly completed, then it must be post-dominated by a completion call. Yet, none of `if.then` and `if.end` is able to post-dominate `entry`. The set composed of these two nodes must be considered instead, and we use the concept of generalized post-domination (Defintion 5).

```
1  int g(int arg)
2  {
3      int a = 10;
4      MPI_Request req;
5      if (arg == 0){
6              printf("Hello␣World!\n");
7      } else {
8          MPI_Ibcast(&a, 1, MPI_INT, 0,
               MPI_COMM_WORLD, &req);
9          if (arg < 0) {
10             MPI_Wait(&req,
                   MPI_STATUS_IGNORE);
11         } else {
12             MPI_Wait(&req,
                   MPI_STATUS_IGNORE);
13         }
14     }
15     return a;
16 }
```

(a) MPI and generalized post-domination



(b) CFG of function in Figure 4.3a

Figure 4.4 – Example of generalized post-domination

Using the notations from Definition 5, n would be the initiation call, and S the set of completion calls. The minimal property also ensures there is only one "layer" of completion calls in the set, or in other words, that there are no successive completions.

For example in Figure 4.4b the second criteria excludes the set composed of nodes `if.then`, `if.end`, and `if.else` from post-dominating `entry`. In the case of node `if.end`, every path from `entry` to the sink which contains `if.end` also have to contain `if.else`, thus contradicting the definition. However, the sets {`if.then`, `if.end`}, {`if.then`, `if.else`}, and {`if.then`, `if.then3`, `if.else5`} are all post-dominating sets of `entry`.

Implementing the generalized post-domination relation

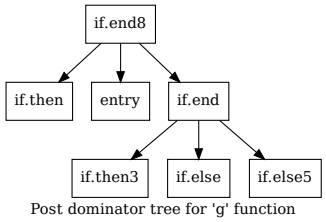
The verification and transformation methods we propose in the coming Chapters have been implemented as LLVM passes. The detail of their implementation are given in their respective Chapter. LLVM, which is the focus of Section 2.2, is a production ready compiler framework. It implements many analysis techniques, including the single node post-domination relation. However, the generalized post-domination is not implemented in LLVM. This Section discusses the implementation of this notion in our analysis and transformation passes.

Sreedhar et al. proposed an algorithm to compute the generalized domination in polynomial complexity[30]. We have adapted the algorithm for the generalized post-domination, considering that the post-domination can be seen as the domination applied to the inverted CFG, which is obtained by inverting the direction of each edge in the original CFG. The computation of the sets is based on the determination of the multiple node immediate post-dominator sets (see Definition 6. The determination of these sets is done on the **post-dominator DJ-graph** of the CFG (see Definition 7).

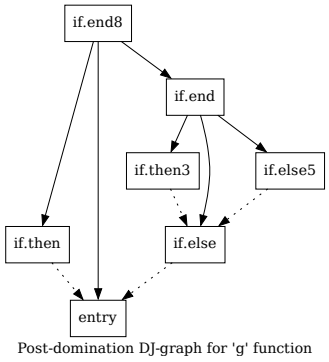
Figure 4.5b is the DJ-graph for the CFG shown in Figure 4.4b. It is built upon its post-dominator tree in Figure 4.5a. The solid edges are the edges from the post-dominator tree, and thus are the D-edges, while the J-edges are dotted. As an illustration, let us consider the vertex `if.else`. It is immediately post-dominated by `if.end`, hence the D-edge. Its successors in the CFG are `if.then3` and `if.else5`, but none of those are post-dominating it. As a consequence, the reverse edges (`if.then3` $\rightarrow$ `if.else`) and (`if.else5` $\rightarrow$ `if.else`) are J-edges.

Their algorithm [1] to build the multiple-node immediate post-domination relation of a vertex is based upon the following observation. Given two nodes x and y in a CFG, if y belongs to the *mipdom* of x, then there must be at least one path from y to the single immediate post-dominator of x (noted $sipdom(x)$) that does not contain any successor of x, excluding y. The algorithm consists in checking the successors of each vertex x in a CFG. For each successor y of x, if there is a

1. Algorithm 3.1 in [30]



(a) Post-dominator tree of CFG in Figure 4.4b



(b) DJ-graph of CFG in Figure 4.4b

Figure 4.5 – Post-dominator tree and its associated DJ-graph of function `g` from Figure 4.4a

path from y to $sipdom(x)$ to it that does not contain any other node from $\text{Succ}(x)$, then y is added to $mipdom(x)$. The DJ-graph helps in finding such a path.

As an illustration of this property, let us take a look at vertex `if.else` in Figure 4.4b, and see how this property applies to it. The *sipdom* of `if.else` is `if.end`. The only path from $y =$ `if.then3` to $sipdom(x) =$ `if.end` is `if.then3` $\rightarrow$ `if.end`. It does indeed not contain any node from $\text{Succ}(x =$ `if.else`$) \setminus \{$`if.then3`$\} = \{$`if.else5`$\}$, thus checking the property in this example. The same can be applied to the vertex `if.else5`, which the other successor of `if.else`. By definition, we know the set $\{$`if.then3`, `if.else5`$\}$ is post-dominating `if.else`,

Since the completion calls might not be in the immediate post-dominating set of a nonblocking call, it is necessary to fetch all post-dominating sets of a node. To do so, we rely on an algorithm [2] developed by Gupta [29]. As with the concept of *mipdom*, what follows is an adaptation for the post-domination relation.

Let $\mathcal{D}_x$ be the set of generalized post-dominators of x. In order to build $\mathcal{D}_x$, it starts by adding all immediate post-dominators of x, both single nodes and sets. In this algorithm, single nodes post-dominators are viewed as sets of cardinality 1. From this point, the goal of the algorithm is to complete the set of generalized post-dominators thanks to the transitivity of the relation, and by combining the post-dominators to create other sets.

For each element w from a given immediate post-dominating set V, it looks at

2. Figure 5 in [29]

its immediate post-dominator (both single and multiple nodes). For each immediate post-dominator d_w, it checks if d_w does not post-dominate any other node of V. Should this condition be satisfied, it builds a set V' which is made of $d_w \setminus \{w\}$ (i.e. the strict post-dominators of w). If V' is already in $\mathcal{D}_x$, nothing is done and it goes over to the next immediate post-dominator of w. Otherwise it is added to $\mathcal{D}_x$, and the method is recursively repeated with V' until it converges.

Both of those algorithms have been implemented in our LLVM plugin passes. Applying this algorithm to the code in Figure 4.4a, it allows us to match the nonblocking broadcast initiation call with the two completion calls in the branches. In this example, the initiation call is in the `if.else` basic block, while the completion calls are in `if.then3` and `if.else5`.

4.3 Analysis and comparison of paths in a CFG

While the flow sensitive approach is useful for matching nonblocking communications functions when their execution is not constrained by a condition, it fails at matching nonblocking calls residing in different branches or loops.

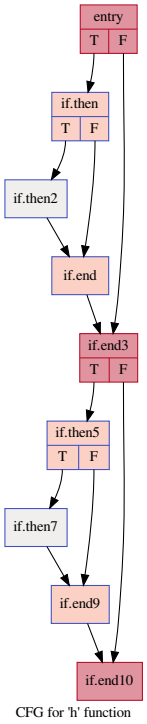
Figure 4.6 is an example of situation where the post-domination returns a false positive. The `MPI_Ibcast` at line 7 in Listing 4.6a is in the basic block `if.then2` in Figure 4.6b, while its completion call is in basic block `if.then7`. When considering the code, the nonblocking communication is safe: the execution of the initiation call implies that `x` and `y` are equal to 0, and the execution of the completion call requires both values to be 0. Furthermore, `x` and `y` are not modified between the two conditions. The conditions allowing the execution of the initiation and the completion calls are the same, ensuring the safe completion of the communication. Yet, there is no post-domination relation between `if.then2` and `if.then7`.

Figure 4.7 exposes a similar situation with an initiation loop and a completion loop. Both loops have the same number of iterations, and the access to the request by the initiation and the completion calls is performed under the same conditions, without overwriting or without offset in the indexes. This communication is sound. Yet again, there is no post-domination relation between `loop.body` where the initiation call is, and `loop.body4` which holds the completion call.

The example above showed that the post-domination method is unable to decide if, once the flow has gone through a sequence of blocks, it is likely to follow another sequence of blocks. To remedy to this shortcoming, we complement that analysis with a path sensitive matching, which is the focus of this Section.

```
1  int h(int x, int y)
2  {
3      int a = 10;
4      MPI_Request req;
5      if (x == 0)
6          if (y == 0)
7              MPI_Ibcast(&a, 1, MPI_INT, 0,
                   MPI_COMM_WORLD, &req);
8      if (x == 0)
9          if (y == 0)
10             MPI_Wait(&req, MPI_STATUS_IGNORE);
11     return a;
12 }
```

(a) Safe communication in nested branches



(b) CFG with nested branches

Figure 4.6 – Matching of nonblocking communications in branches

4.3.1 Condition dependent communications

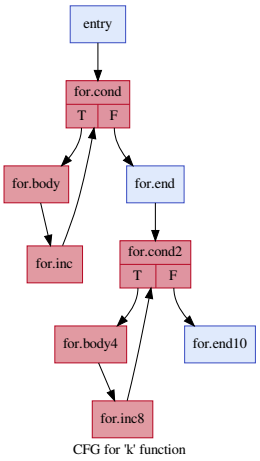
In this section, we focus on determining the conditions guarding the nonblocking communication calls. Once these conditions have been identified, it is then possible to determine if the completion call is likely to be executed after the initiation.

Identification of fork nodes

In order to retrieve the conditions guarding a path, it is necessary to first identify the basic blocks from which the path originates. For example in Figure 4.6b, the node A is the forking point which leads to either B or E. Thus, it contains the instruction that determines which path should be taken depending on some

```
1  int k(int x, int y)
2  {
3      int a = 10;
4      MPI_Request req[x];
5      for (int i = 0; i < x; ++i) {
6          MPI_Ibcast(&a, 1, MPI_INT, 0,
               MPI_COMM_WORLD, &req[i]);
7      }
8      for (int i = 0; i < x; ++i) {
9          MPI_Wait(&req[i],
               MPI_STATUS_IGNORE);
10     }
11     return a;
12 }
```

(a) Safe communication in loops



(b) CFG with multiple loops

Figure 4.7 – Matching of nonblocking communications in loops

condition, which is of interest for our analysis.

The creation of the SSA form of a program involve the placement of ϕ-functions. Those functions are placed at join nodes with the help of the domination relation. The same principle can be applied to detect fork nodes in a CFG. Knowing that the post-domination relation is equivalent to the domination relation on the reverse CFG, and that the fork nodes become join nodes in the reverse CFG, it is possible to use the post-domination to determine these fork nodes. Instead of resorting to the dominance frontier, we will use the post-dominance frontier (see Definition 9) to find the forks in a CFG.

The PDF can be seen as the limits of the post-domination "power" of a vertex. The nodes beyond this line cannot be post-dominated by that vertex. To reformulate the formal definition, if a node x belongs to the PDF of a node n, then:

1. x is not post-dominated by n,
2. One of the successors of x is post-dominated by n.

This means there are multiple paths originating at x. At least one of them is guaranteed to contain n before reaching the sink of the CFG. On the other hand, at least one of these paths allows the bypassing of n. Thus, the node x decides whether n should be visited.

For example, let us find the PDF of node `if.then2` in Figure 4.6b. `if.then2` does not post-dominate `if.then`, but does post-dominate one of the successors of `if.then`: `if.then2` itself. As a result, PDF(`if.then2`) = {`if.then`}. It contains the instruction deciding the path to take.

However, the decision in `if.then` is not the only condition allowing the execution of the initiation call in Figure 4.6b. It is also guarded by the `entry` basic block. This situation translates a nested `if`, and can be resolved with the iterated post-dominance frontier (see Definition 10). The PDF^+ allows us to identify all forks leading to the node of interest. In Figure 4.6b, the PDF^+ of vertex `if.then2` would be the set {`if.then`, `entry`}. `if.then` is computed at the first step (simple PDF of `if.then2`). `entry` is the result of PDF(`if.then`). At this point, the determination of the iterated post-dominance frontier has converged, since A does not have any PDF.

Implementing the PDF

While LLVM provides an implementation of the dominance frontier, none is provided for the post-dominance frontier. Sreedhar et al proposed an algorithm [3] to compute the dominance frontier based on the DJ-graph and in linear complexity relative to the number of edges in a CFG [30]. In their study, they showed that a vertex z belongs to the dominance frontier of x if and only if there is, among the nodes that are dominated by x, a node y which satisfies the following conditions:

1. $y \rightarrow z$ is a J-edge,
2. x is at equal depth, or deeper than z.

Their algorithm is a direct application of this property, by iterating over all the outgoing edges of each node dominated by x. The nature of each edge is checked against the DJ-graph to determine whether it is a J-edge, and the depth of its destination is compared to the depth of x. We adapted the algorithm for the post-domination relation. Finally, computing the iterated post-dominance frontier consists in recursively finding the PDF.

Comparing conditions and execution paths

By identifying the forking nodes, it is possible to know the required conditions to execute some nodes in a CFG, and in our situation, to know the conditions leading to the execution of nonblocking MPI calls. At compile-time unfortunately, it is difficult to predict the execution path because of the lack of information on

3. Algorithm 5.1 in [30].

the inputs, the effective value of variables, or the behavior of functions. However, it is possible to compare values and predicates: to what do they refer, do they represent the same memory zone, have they been modified in the meantime, are they being used in the same way?

In order to know if an initiation call can be matched with a completion call, because the execution of the first ensures the execution of the second, we compare the conditions leading to their execution, and the resulting path. If the conditions are similar, and the selected path contains both the initiation and completion calls, then we can assume that, if the initiation is executed, the completion call is likely to be executed as well.

For example in Listing 4.6a and its corresponding CFG in Figure 4.6b, the broadcast initiation is only executed if both the conditions in `entry` and then in `if.then` are valued to "true". The completion call, in the `if.then7` basic block, follows a similar pattern, and is only executed if the branching condition in both `if.end3` and then `if.then5` are "true". To determine if the execution of the initiation call leads to the execution of the completion call, and thus the safety of the communication, we must first determine if the conditions in (`entry`, `if.then`) are equivalent to the conditions in (`if.end3`, `if.then5`).

Two Boolean expressions are equivalent if the following criteria are met:

1. The operands of the expression must refer to the same object, which must not be altered between the two expressions,
2. The operators must be the same.
3. The order of the operands must be the same in both expressions, if the operator is not commutative,

Using the same operators in both expressions ensures the same decision is being taken depending on the operands. The order of the operands must also remain the same. Two exceptions exist to the third and fourth rules. First, if the operator is commutative, then the order of the operands may not be considered, as for any commutative operand $\otimes$, $A \otimes B$ is equivalent to $B \otimes A$. Secondly, if the other operand is different but happens to be the inverse, then it is possible for the operands to have their position swapped. For example, $A > B$ and $B < A$ should be treated as equivalent Boolean expressions. Note that if the expressions deciding which paths are taken are results of a function call, then both calls must refer to the same functions and the operands (or arguments) must be used in the same order.

The first condition ensures the values used in the expressions are the same. The SSA form, defined in Section 2.1.4 guarantees that once a variable has been

defined, every further uses of that variable will use the same value. Should the value be modified, a copy of the variable must be made, and the new value written into the copy, the previous variable keeping its original value. Therefore, if two expressions use the same SSA variable, they use the same value. In LLVM-IR however, it is possible for SSA variables to contain addresses whose memory space is not protected by this property, typically when their represent pointers. The SSA form only protects the value of the variable, which is, in the case of pointers, the address. To correctly handle pointers, we rely on the MemorySSA, described in Section 2.2.2. It is a virtual SSA built on top of the IR to track the evolution of the memory state. Thus, we are able to determine if two operands have the same value, using both the regular SSA and the MemorySSA.

Consequently, the three criteria for asserting the equivalence of two Boolean expressions are met. Their equivalence ensures that, once a branching decision has been made in the first expression, the same branching decision will be made in another equivalent expression. Yet, the equivalence of two expressions does not guarantee the matching of the nonblocking calls. We must also consider the result, that is the subsequent path from the decision. For example in Listing 4.6a, the `MPI_Ibcast` is only executed if the result from `"x == 0"` is true, and if the result from `"y == 0"` is true. The initiation call is on the path which have both expressions evaluated to "true". The Boolean expression protecting the completion calls are equivalent. Yet, if the completion call were to be in the "else" branch of one of those conditions, the matching would have to fail.

To keep track of the required results to reach some basic block, we defined a structure composed of: 1. the Boolean expression, and 2. the required result. As we determine the forking nodes through the PDF^+, we build a list composed of these structures. Each forking node is attributed such a structure, with the Boolean expression used in the branching instruction and the expected result of the expression to reach the basic block of interest. This list allows us to compare the conditions of an execution path. Two execution paths, each represented by a list of these structures, are equivalent if the two lists have the same elements.

Applied to the Listing and CFG we used thorough this Section, both the path leading to the initiation call and the one leading to the completion call would be represented as a list with two elements. The first element being `"y == 0"` and `"true"`, while the second element `"x == 0"` and `"true"`. These two lists have the same elements, since the Boolean expressions are equivalent, and the subsequent result of each expression matches, thus ensuring the matching of the nonblocking communication.

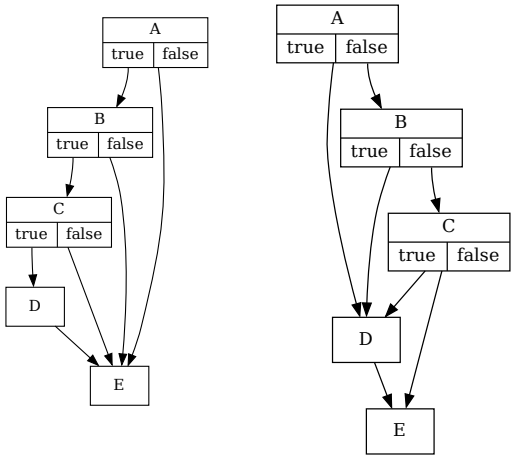
The method we have developed here allows us to compare execution paths and

determine at compile-time the consequences of the execution of a section of the CFG. It is primarily based on the post-dominance frontier and an analysis of the data flow.

Composed Boolean expressions

Composed Boolean expressions are made of several single operand Boolean expressions linked together by Boolean operators (e.g. `and`, `or`). In LLVM, these expressions will result in different kinds of subgraphs in the CFG.

A succession of Boolean expression linked with `and` operators, for example `if(A and B and C)`, will be translated as a succession of forking nodes, where the scope protected by the `if` can only be accessed if all conditions are *true*. The resulting CFG is illustrated in Figure 4.8a. The node D, containing the scope of the `if` instruction, can only be accessed if the branching conditions in both A, B, and C are true. Otherwise, the flow is being directly sent to E. Identifying the access conditions in this situation is similar to how nested `if` are handled, by building $\mathrm{PDF}^{+}(D)$. The handling of this situation is similar to the example shown in Figure 4.6.



(a) Succession of expressions chained with `and`

(b) Succession of expressions chained with `or`

Figure 4.8 – Representation of composed Boolean expressions in CFG

A succession of `or` operators, for example `if(A or B or C)`, will result in

multiple forking nodes, each of those potentially leading into the protected scope. Figure 4.8b illustrates this situation. The first occasion to go into D occurs when the condition in A is verified. If not, a second chance occurs in B. The last chance occurs in C, and eventually goes to E if none of the conditions were met. It is possible to detect all the possibilities to execute D by calculating its PDF. Indeed, in this pattern PDF(D) is the set $\{A, B, C\}$. D do not strictly post-dominate any of these vertices, but it does post-dominates one of their successors, which is itself. In this situation, only one of these three conditions needs to be equivalent.

4.3.2 Communication calls in loops

A looped nonblocking communication will initialize multiple requests, which must all be caught by a completion call. We assume that, given an array of requests, each iteration of an initiation call will write its request to a distinct memory location, without overwriting other requests. Similarly to the comparison of branching conditions, we need to verify that the completion calls catch a sufficient number of requests in the correct range.

To that end, it is necessary to compare the number of calls. Since we are analyzing the code at compile-time, it is difficult to know this value with precision: it can depend on factors only decidable at execution, such as the input parameters. Therefore, as we compared the branching conditions, we can determine if the number of initiated requests matches with the number of completed requests. Given two loops, the initiation call inside the body of the first and the completion inside the second, for example nodes (A, B) and (D, E) in Figure 4.7b. We determine if B and E are executed the same amount of time.

The Scalar Evolution, described in Section 2.2.2, is analysis with the ability to track the evolution of induction variables in loops. More specifically, it is able to track the evolution of the loop counter: its initial and final values, its step, and its incrementing or decrementing operator.

The request graph we defined allows us to fetch the base address that is being used as a base for accessing the array. Using this graph, it is possible to determine which variable is being used as index to access its elements through the recorded GEP instructions. If the same array is being used by both the initiation and completion calls, the indexes are given to the SCEV, which will give insights on their behavior in their respective loops. First we identify the upper bound of the number of initiated communications, then the upper bound for the number of completion calls or the number of expected requests in the case of the `Waitall` version. The second value must be comparable to the first: it must be possible to

determine if the two values are the same, either statically, or by using the same methods to match branching conditions. The lower bound, operator, and step value are also considered using the same methodology, and their values must be equal. This ensures there are enough requests being caught. This also ensures that both calls are using the same section of the array.

Consequently, coupled to the request graph we defined, it is possible to determine if two loops will have a similar number of iterations by using the SCEV, and thus checking if loops of initiation calls can match with loops of completions. It also allows the matching of the `Waitall` version. Instead of checking against the number of iterations in the completion loop, we compare to the number of expected requests in the completion call.

In the end, matching the properties of two loops helps us in matching nonblocking calls. Thanks to it, it is possible to determine that the initiation and completion loops have the same number of iterations, the same starting conditions, and that the nonblocking calls access the same elements of the request array. This ensures that there are enough completion calls in the execution path, and that each initiated communication will eventually be completed.

4.4 Defining the mandatory set

In Section 4.2, we developed a method to match nonblocking communications with the notion of post-domination. This concepts determines if a vertex or set of vertices in a CFG is a bottleneck for a reference vertex. Applied to nonblocking communications, it determines if the initiation call has the possibility of bypassing the potential completion calls, rendering the communication unsafe and susceptible of deadlocking. However, in Section 4.3, we identified situations where the post-domination approach cannot correctly match the nonblocking calls. With an analysis of the data flow, we are able to compare execution paths and loops to determine whether the execution of the initiation call leads to the execution of the completion call, thus ensuring the safety of the communication.

The matching of nonblocking communications relies on these two methods. In this context, we define the mandatory set of a basic block.

Definition 11 (Mandatory set)**.** Let $G = (V, E)$ a CFG. $\mathcal{M}(n) \subset V$ is a mandatory set of $n \in V$ if one of the following criterion are met:

1. $\mathcal{M}(n)$ post-dominates n,
2. $\forall x \in \mathcal{M}(n)$, x shares the same execution path as n.

The sets $\mathcal{M}(n)$ from Definition 11 are the basic blocks that are always executed following an execution of the basic block n, hence the "mandatory" name. The second criteria comes from the path sensitive analysis we developed in Section 4.3. By being on the same execution path, the execution of n always lead to the execution of the nodes of the mandatory set. The first criteria comes from the analysis of the CFG, developed in Section 4.2. By definition of the post-domination, all paths from n to the exit of the represented function have to contain a node from $\mathcal{M}(n)$, thus ensuring the execution of at least one basic block from the mandatory set.

From Definition 11, the following property can be deduced: an MPI nonblocking initiation call is safely completed only if it has a mandatory set where all of its basic blocks contains a completion call accessing the same request. By being on all vertices of the mandatory set of the initiation call, the completion calls are always executed after the initiation call, which ensures the correct completion of the communication.

Through this Chapter, we showed how compile-time analysis techniques can be leveraged to analyze MPI nonblocking communications. Section 4.1 focused on the analyses of the data flow, allowing the detection of completion calls and of data dependencies. Sections 4.2 and 4.3, which ultimately results in the mandatory set described in 4.4, described how the CFG and the data flow analyses of the IR can be used to match nonblocking calls. The point of performing all these analyses at compile-time is to provide solutions that can be seamlessly integrated to the development cycle of an MPI program, with an as low as possible impact on the development time. Using the techniques we described in this Chapter, we define a verification pass to detect mismatching nonblocking calls and data races at compile-time, and helps developers in fixing their code. It is the focus of Chapter 5. These techniques are also employed in Chapter 6, in which we define an optimization pass to create and extend the overlapping potential of a code.

Chapter 5

Verification of MPI Nonblocking Calls at Compile-Time

Debugging tools are great companions to developers and assist them during the development cycle, from the first prototypes to the production stage of a program. Automatic detection of errors relieves the burden put on developers when they are confronted to complex programming interfaces and obscure use cases of some interface. Debugging tools are also helpful in detecting common errors that could occur on more mundane code while the programmer's attention might be shifted to more demanding tasks. In the end, having a debugger around, provided that it delivers informative feedback without too many falsely flagged errors, can help in increasing code quality while reducing development time and cost.

In our study, we focus on static debugging methods. Relying on static analysis, they can operate without executing the program, intervening in the most early phases of a software's life cycle. This allows for a very early detection of errors, and not having to execute the actual program is a substantial benefit in terms of development time. Indeed, it removes the lengthy back and forth cycle in correcting the source code, compiling, and running the program until it reaches the ill point. These methods will usually stop this feedback loop at the compilation step or even during the initial coding phase, to warn developers of potential faulty lines of code. Furthermore, analyzing the code during or after its execution restrains the detection scope of debugging tools only to the executed portions of code. Static debugging tools allow the detection of errors regardless of the conditions of execution, and are able to cover the program as a whole.

The strengths of static detection of errors are especially suited for MPI programs which are expected for extended duration on multiple computing nodes.

Their parallel nature makes it hard for serial debugger to analyze their behavior at execution, and specialized tools must be used. Dynamic detection can provide accurate results, but requires the execution of the MPI code. Given the particularities of an HPC environment with limited access and quota to computing resource and time allocation, such approach is not desirable.

In this chapter, we propose a method to statically detect matching and data-race errors in the usage of nonblocking MPI communications. It relies upon some of the notions we introduced in Chapter 4. This method focuses on the detection of three kinds of errors:

1. Uncompleted communication;
2. Illegal access to communication buffers;
3. Request overwriting.

The first kind is the subject of Section 5.1 while the last two remaining are treated in Section 5.2. Most of the contributions of this Chapter have been published at the Correctness workshop in 2020 [74].

5.1 Detection of mismatched nonblocking calls

Because of their split form, MPI nonblocking communications are sensitive to mismatching errors. Every initiated communication must be completed. If this condition is not fulfilled deadlock errors can happen, preventing the program from progressing further. In this Section, we focus on asserting if all nonblocking communications in a code comply to this rule. Using the same method, it is also possible to determine if a completion call will catch an empty request. In other words, the completion call completes a request that has not been initialized, thus it does not complete any nonblocking communication. In such situations, the MPI specification allows the completion call to ignore the request, and to safely return control to its caller. Yet, it would still be interesting to highlight such completion calls, since they can help in correcting an unmatched initiation call, or contribute to improving coding practices.

The matching of nonblocking calls is performed in three steps. First we identify the *potential* completion calls, then the *capable* completion calls, and finally the *matching* completion calls. We define these concepts as follows:

Definition 12 (Potential set)**.** For a given nonblocking initiation call, a set $\mathcal{C}$ of completion points is a *potential set* of completion call if, $\forall w \in \mathcal{C}$, w uses the same request as the initiation call.

Definition 13 (Capable set)**.** For a given nonblocking initiation call, let $\mathcal{C}$ be its set of potential completion points. $\mathcal{C} \subset \mathcal{P}$ is a *capable set* if it is a mandatory set of the initiation call (see Definition 11.

Definition 14 (Matching set)**.** For a given nonblocking initiation call, the *matching set* of completion points is the capable set that will effectively catch and handle the request.

5.1.1 Identification of capable sets of completion calls

The initiation-completion matching operates on each nonblocking initiation in a function, and tries to determine such sets. Algorithm 1 is the top level algorithm for our matching method. It operates on each function defined in a translation unit. For each of these functions, the list of nonblocking initiations it calls is established. Its PDDAG and its MSSA representation are computed.

Matching nonblocking calls firstly relies on the successful identification of possible associations of initiation and completion calls. As we have detailed in Section 4.1.1, the `MPI_Request` objects indicate possible associations of initiation and completion calls. Using the request graph we defined, our approach determines all completion calls that use the same request, or base address in case of an array. This is performed inside the function `getWait` in Algorithm 1. Those are the *potential* completion calls for our communication, since we have no knowledge on their ability to catch request, either as a single call or as a set of calls.

Algorithm 1 Matching MPI initiations and completion calls

Require: mpi_i: list of MPI nonblocking initiation calls in f, mpi_w: list of MPI nonblocking completion calls, PDDAG: graph of generalized post-dominators of f
Ensure: Each MPI nonblocking call is properly matched
procedure MATCHMPINONBLOCKING(function f)
for all mpi_i ∈ f **do**
potentialWaits ← getWait(mpi_i.req)
capableWaits ← determineCapableWaits(mpi_i, PDDAG, AA, potentialWaits)
matchedWaits ← determineMatchedWaits(mpi_i, capableWaits)

Having determined which completion calls are likely to catch an initiation, the matching method now needs to determine if they can catch it before the execution flow reaches the end of the program. Indeed, a completion call using

the same request as the initiation call is not enough since it does not guarantee it will be executed after the initiation call. To that end, both the flow sensitive and the path sensitive analyses are used. These two methods complement each other and they are both needed. The flow sensitive approach only analyzes the CFG on a wider scale, and determines if there is a path bypassing some set of completion calls. It will not consider the conditions that defines a path for the execution flow. As a consequence, if our method only relied on this analysis, it would report a false positive when a nonblocking communication is split in two separate branches sharing the same condition, as explained in Section 4.3. On the other hand, the path sensitive approach will consider the conditions under which a path will be taken. It also considers looped calls and the number of iteration, thus checking if a completion call is able to catch all initiated communications. This step is performed inside the `determineCapableWaits` function in Algorithm 1, and is further detailed in Algorithm 2. It relies on two functions. One checks situations where the initiation call is looped, ensuring all initiated requests are getting caught. The other one manages the general case, and checks the control flow and the possible paths after the initiation call.

General case

The core of the matching relies on determining if, once the control flow has gone through an initiation call, it will go through a potential completion call before reaching the end of the program. These steps are performed in the `checkPath` function of Algorithm 2.

The matching begins by applying the flow sensitive analysis. Its goal is to find all sets of completion calls that can intercept the execution flow from the initiation call. Each of these sets, containing one or multiple basic blocks with a completion call, post-dominates the initiation call. By definition, there is no way to bypass them, thus ensuring that the communication request will be correctly terminated once initiated.

In a second time, the path sensitive analysis is applied on all completion calls. Its goal is to determine which completion calls share the same execution conditions as the initiation call. If they do, then executing the initiation call would mean the completion call will also be executed. At this point, it checks if the deepest control structure encompassing the initiation call is a loop. Such situations are handled differently, and will be the focus of the next Section. For each remaining initiation call, we build the list described in Section 4.3.1, which tracks the conditions allowing the execution of these calls. The same list is then built for each potential completion call.

The determination of the capable completion calls relies on the comparison of those lists. Two lists are equals if both are of the same size and if their elements

Algorithm 2 Determination of completion calls capable of intercepting the communication

```
procedure DETERMINECAPABLEWAITS(mpi_i, PDDAG, AA, potentialWaits)
    capableWaits ← ∅
    if mpi_i.isLooped then
        for all wait ∈ potentialWaits do
            if wait.loop ≠ mpi_i.loop then
                checkIteration(mpi_i, wait, AA, capableWaits)
            else
                checkPath(mpi_i, wait, PPDAG, AA, capableWaits)
    else
        checkPath(mpi_i, potentialWaits, PPDAG, AA, capableWaits)
    return capableWaits
procedure CHECKITERATION(mpi_i, wait, AA, capableWaits)
    Low_i, Up_i, Step_i ← mpi_i.loop iterator lower and upper bounds, and step
    if wait is an MPI_Wait or MPI_Waitany then
        Low_w, Up_w, Step_w ← wait.loop iterator lower and upper bounds, and step
        if Low_i = Low_w and Up_i = Up_w and Step_i = Step_w then
            capableWaits.add(wait)
    else if wait is an MPI_Waitall then
        l ← Waitall request array length
        p ← Waitall request array starting index
        if Low_i = p and Up_i = (l − p) and Step_i = 1 then
            capableWaits.add(wait)
procedure CHECKPATH(mpi_i, potentialWaits, PDDAG, AA, capableWaits)
    pdomWaits ← getPostDominatingSets(mpi_i, potentialWaits, PDDAG)
    capableWaits.append(pdomWaits)
    conditions_init ← buildAccessPath(mpi_i)
    for all wait ∈ potentialWaits do
        conditions_wait ← buildAccessPath(wait)
        if conditions_init = conditions_wait then
            capableWaits.add(wait)
```

are equal at each level. For both the initiation and completion calls, this would mean they need to go through the same number of checks, each sequence of check contains the same conditions put in the same order, and the resulting path must be the same. Thus, if the initiation call is executed, the completion call will also be executed, thus making it capable of terminating the communication.

Nonblocking communications in loops

Let us consider looped initiation calls separately. These communications require specific treatment because they initiate several requests, which all need to be caught be the capable completion calls. One "simple" enough situation arises when the potential completion point is inside the same loop as the initiation. In this case to ensure all requests are getting terminated, the completion should come after the initiation in the same iteration. This is verified by performing both the path sensitive and flow sensitive analyses inside the loop, as described in the general case. All single or set of completion points which satisfy these conditions are capable of properly catching the requests.

Completion calls residing outside the initiation's loop or in a distinct loop require more work from the matching method to assess their ability. These calls, if in a `Waitall` form, need to expect enough requests, and the same that were taken by the initiation call. If of the `Wait` or `Waitany` form, then they must belong to a loop distinct from the loop with the initiation call. Furthermore both loops must share the same properties (bounds, step, loop counter operator). Finally, to ensure both calls use the same section of the request array, they must access to its elements by using the loop counter without performing any further arithmetical operation. Indeed we know both calls share the same request array, and by knowing how the loop counter evolves throughout the iterations and its boundaries, we know if both the initiation and completion calls are accessing the same section of the request array. All `Wait` calls agreeing to these conditions are capable of completing the communication. Function `checkIteration` in Algorithm 2 is in charge of performing these checks.

Before considering the remaining forms of completion calls, let us note Up_i the upper bound of the initiation loop counter.

`Waitall` calls require specific considerations. These completion calls accept an array of requests and the size of the array. It waits and completes all requests from this array. To catch the initiated requests of such completion call, we must ensure the number of expected requests (i.e. the length of the array of requests), is at least greater than Up_i, and accesses the same section of the array than the initiation call. The completion call must be given the base address of the array with an

offset which is equal to the starting value of the loop counter. For simplicity, our implementation of the matching pass only considers loops whose counter starts at 0, and `Waitall` calls whose expected number of requests is equal to Up_i and whose request array is given by its base address without arithmetical operation. In this situation, the completion call would wait for all requests at indexes between 0 and $\mathrm{Up}_i - 1$, which matches with the indexes of initiated requests. Thus, the `Waitall` call becomes capable of terminating the nonblocking communications.

`Waitany` calls follow a slightly different procedure, albeit having a similar prototype to the `all` form. Contrary to the two previous forms, its behavior is undecidable at compile-time. This kind of completion call is given an array of requests, and each execution of it will complete one element of the array, whose position is unknown beforehand and only given after its execution. Consequently, it is impossible to know in advance which request will be completed. To circumvent this limitation, we require `Waitany` calls to be treated in the same way as `Wait` calls. They must belong to loops which share the same properties as the initiation loop. This constraint guarantees the completion of all initiated requests. `Waitany` completion calls that satisfy this condition, and use the same requests as the initiation calls, are capable of properly completing the communication.

`Waitsome` calls have a similar behavior to `Waitany` calls regarding the completion of its requests. They can catch several requests, but the position of the requests in the array is not known until after its execution. Similarly to the previous form of completion calls, it is possible to bypass this issue by adding the same constraint. All `Waitsome`s must be in loops sharing the same properties as the initiation loop, thus ensuring all requests will be completed. This last form has, however, not been implemented in our matching pass.

5.1.2 Identification of the matching set of completion call

The flow and path sensitive methods return sets of capable completion calls. Their existence is already a good omen for the correctness of the code and the well being of the MPI communication. Their absence, however, indicates an initiated nonblocking communication that will never be completed, at least in the scope of its caller function. The analysis pass should then emit a message warning the developers of this fault.

Yet the existence of these sets is not enough to perform analyses on the overlapping of nonblocking communications. Indeed, having multiple sets of completion calls is not enough to determine the overlapping interval of a nonblocking communication. Only one of these sets will actually complete the communication and mark

the lower bound of the overlapping interval, while the others might be assigned to other communications which reuse the same request, or might be completing empty requests.

In order to identify this interval, it is necessary to identify the most "immediate" capable set of completion calls coming after the initiation call. "Immediate" here is akin to the "immediate" multiple vertex post-domination[29], however the immediate multiple vertex post-dominating set of the initiation call does not necessarily contain the completion calls, neither must the completion calls be post-dominators of the initiation call as we emphasized on through the path sensitive analysis. By being the first capable set of completion calls coming after the initiation, they are guaranteed to catch and handle the emitted request, thus defining the overlapping interval. Such completion calls will be referred to as the "matching" calls. The identification of matching completion calls is performed by the function `determineMatchedWaits`, detailed in Algorithm 3.

The determination of this set relies on a traversal of the CFG using a breadth first search-like traversal. The simplest situation occurs when the basic block containing the initiation call also happens to contain capable completion calls. In this situation, the matching call is determined by simply iterating over the instruction list of the basic block, starting from the initiation call, until the first completion call.

Otherwise, the BFS-like traversal starts from the basic block containing the nonblocking call. It visits all succeeding basic blocks until a block containing a capable completion is found. It then checks how the completion call was deemed capable: by the path sensitive or by the flow sensitive analysis. In the first case, the matching call has been found, and the search stops. Being found by the path sensitive analysis means the completion call will be executed once the initiation call has been executed. On the other hand, if the call has been found by the flow sensitive analysis, it is marked as belonging to the matching set, and the search only stops in this branch. It resumes with the remaining nodes to visit. The search is guaranteed to end with the complete matching set. Indeed, having found one of the calls belonging to a post-dominator set of the initiation call, the BFS traversal has no way of finding an escape route, and will eventually meet the other matching calls in the remaining branches.

This determines the basic blocks containing the matching calls. Nonetheless it is possible for these basic blocks to have multiple capable completion calls in them. In this situation, only the first capable call in the instruction list of the basic block is saved and marked as matching. Thus is the matching set of completion calls found and the overlapping interval defined.

5.1.3 Reporting uncompleted communications

During the matching of nonblocking communications, we have determined the three sets of *potential*, *capable*, and finally the *matching* completion calls. The finding of a capable set inevitably leads to the determination of matching set, as there is bound to be a most immediate set. However, finding a potential set does not guarantee the presence of a capable set. Furthermore, it is possible that a code has an initiation call, but no completion call. In these two cases, a warning should

Algorithm 3 Determining the matching completion calls

```
procedure determineMatchedWaits(mpi_i, capableWaits)
    matchingWaits ← ∅
    visitedBB, toVisitBB ← ∅
    BB ← mpi_i.getBB()
    if ∃ wait ∈ BB, with wait ∈ capableWaits then
        wait ← first capable wait in BB after mpi_i
        matchedWaits.add(wait)
        return matchedWaits
    visitedBB.add(BB)
    for all succ ∈ BB.getSuccessors() do
        if succ ∉ visitedBB then
            toVisitBB.append(succ)
    while toVisit ≠ ∅ do
        BB ← toVisitBB.pop()
        visitedBB.add(BB)
        if ∃ wait ∈ BB, with wait ∈ capableWaits then
            wait ← first capable wait in BB
            if wait belongs to a post-dominating set then
                matchedWaits.add(wait)
                continue
            else
                matchedWaits ← wait
                return matchedWaits
        for all succ ∈ BB.getSuccessors() do
            if succ ∉ visitedBB then
                toVisitBB.append(succ)
    returnmatchedWaits
```

be emitted to the developers to alert them on a potential misuse. There are three kinds of warning reports.

The first one is the absence of completion call in a function containing an initiation call. This indicates that a communication has been initiated in a function, yet no completion point can catch it in this function. This warning can be a false positive in an interprocedural context, where another function would be in charge of the completion of the communication. This can be detected by an initial scanning of the function body to find the nonblocking calls it possesses.

The second is the absence of potential completion calls. In this situation, there are completion calls in the same calling function as the initiation point, however, none of those use the same request. As a consequence, the communication cannot be completed. The warning message should inform the developers to check if the nonblocking calls are using the correct requests. This message should be raised when the potential set is empty.

Finally the last kind of warning is the absence of capable completion calls. Here, potential calls have been found, but no subset of those is a mandatory set to the initiation call. In other words, for each potential call, there is a way to bypass it. The warning should invite developers to check if there is indeed a completion call on each possible path starting from the initiation call, and check the conditions of executing both the initiation and the completions. It is delivered when the capable set is empty.

5.2 Detection of communication buffers misuses

The successful matching of a nonblocking initiation with its completions (one or multiple) defines the overlapping interval of the communication. This interval represents the overlapping of the communication by other computing instructions, and is one of the reasons motivating the use of nonblocking communications. However, due to their nonblocking nature, the communication buffers are not protected inside this interval. There is no certitude on the state of an inbound buffer until the completion point, which eventually ensures the expected value has been saved. Likewise, there is no certitude on whether an outbound buffer has actually been sent. Only the completion call will provide these safeties, allowing their unimpeded usage by other instructions. Inside the interval however, any attempt at modifying or reading the concerned memory zones can result in errors.

5.2.1 Unsafe access of communication buffers

The detection of unsafe accesses to communication buffers is performed once the matching is done, and only for nonblocking communications whose overlapping interval has been defined. Performing such analysis on an uncompleted communication would be senseless, since their boundaries are unknown.

The detection of race conditions begins with the construction of the dependency slice of a nonblocking communication using the methods described in Section 4.1.2. It creates the list of instructions that either define or need an argument of the communication. The MPI specification mentions the behavior of each argument, whether they are accessed in read-only mode (outbound value), or as a read and write (inbound value). Using this information, and by knowing how an instruction from the dependency slice is related to an argument of the communication, it is possible to mark each of these statements. This marking conveys the link between the said instruction and the communication, that is the read or write nature of the argument using or needed by it. For example, if an instruction is using an outbound argument, which is an argument that is being sent or only read in the communication, then it is marked as read-safe.

Once the dependency slice has been determined and all of its elements marked with a read or write sensitivity, it is possible to look for potential race conditions inside the overlapping interval. To this end, every instruction between the initiation call and its completion calls is checked. This is done by performing the same BFS-like traversal of the CFG that was needed to determine the matching completion calls. Only this time, each instruction has to be considered, and tagged if it belongs to the dependency slice.

If one of the instructions of the overlapping interval belongs to the dependency slice, its read or write sensitivity is taken into account. Let us suppose, for example, the instruction is accessing a value that is only read in the communication. Then, we would need to look at the action it performs on the communication argument. If it only reads the argument, the dependency between the communication and this instruction is a "read after read", which is safe. On the other hand, if the argument is written over, the dependency becomes a "write after read", which is erroneous and the analysis pass should emit a message to inform developers about this issue. Finally, if the argument is inbound, the dependency becomes either "read after write" or "write after write", which are not safe. The same applies for inbound arguments. In this situation the dependency becomes either a "read after write" or "write after write", and neither are safe, thus a warning message must be emitted. Call sites must be considered separately from the other kinds of instructions. Without an interprocedural analysis it is impossible to determine

how an argument is used. As a consequence, our intra-procedural analysis has to assume all arguments are being written inside the function body. This will lead to a higher rate of false positive results, but ensures the safety of a program.

To sum up, the detection of race conditions is based on the identification of the dependency slices of the nonblocking communication. Based on the MPI specification and the operation performed in an instruction, it is possible to tell if there is an unsafe dependency. Such situations are reported to the developers to help them in fixing their code.

5.2.2 Request overwriting

`MPI_Request` objects carry important information about the nonblocking communication they were assigned to. As a consequence, this request must not be modified inside the overlapping interval, before reaching the completion call. Only the matching completion calls are allowed to access the request to identify which communication should be terminated. Any other modification of the request can result in a loss of information, preventing the proper completion of the communication, and eventually leading to deadlocks.

The detection of requests overwriting errors is a specific case of the detection of race conditions. Indeed, requests also appear in the argument list of MPI nonblocking calls. Furthermore, the dependency slice of the request has already been computed since it was needed for the identification of potential completion calls. It contained not only the completion calls, but all instructions accessing the request. Using this list, the method for detecting regular data races is applied, and all instruction accessing the request is flagged. These must be reported to the developers with an informative message, indicating a source of deadlock errors.

5.3 Experimental results

5.3.1 Implementation

The verification method we devised in Sections 5.1 and 5.2 have been implemented as an LLVM 12 pass plugin. It operates on each function of a translation unit, and its analysis scope is limited to the body of the function. As a pass for LLVM, it operates on the IR. Since all of the benchmarks used in this study are written in `C` or `C++`, we use Clang to generate the IR. We chose to disable most LLVM internal passes in order to operate on a mostly unchanged code, closer in structure to the original source code. The analysis pass we build still needs the

application of a few transformation passes which format the IR, most notably for the SCEV analysis. The needed transformation passes are: `instcombine`, `lowerinvoke`, `simplifycfg`, `sroa`, `loop-rotate`, and `loop-simplify`.

The most notable of those is the `lowerinvoke` pass. It removes all exception handling from `C++` codes. While this is a major infringement of the semantic of the program, it is necessary to allow the analysis of the IR. Without this pass, LLVM considers each function capable of emitting an exception. As a consequence, each call site would generate a fork node with one path leading to a landing pad handling the exception. Since LLVM does not know what kind of exception the function can emit, it ultimately leads to the abortion of the program. Statically this is an issue, as each call site have a path going straight to the end of the program, preventing any post-domination. It is also problematic for the comparison of branching conditions. The other transformation passes have no impact on the behavior of the program, and help in formatting the IR, easing the verification pass. The other passes are more thoroughly described in Section 2.2.3. `sroa, loop-rotate, loop-simplify`, format loops and their induction variables so they can be analyzed by the SCEV. The remaining transformation passes simplify the IR by removing code rendered inaccessible or useless by the `lowerinvoke` pass.

The verification pass also requires several other passes to determine pointer aliases. Those passes are: `basic-aa`, `globals-aa`, and `tbaa`.

Being an intra-procedural verification method, the analysis scope of our pass is limited to the body of the examined function. Consequently, it might struggle to determine the behavior of function calls regarding their accesses to the memory, their calls to MPI operations, or handling of global variables. This includes class attributes, since these are defined at the level of the structure rather than in the scope of the method. As a safety measure, the alias analysis assumes a function call touches all regions of the memory.

In LLVM, it is possible to reach a higher accuracy by telling the analysis that a function has a restricted reach on the memory. This is done by adding the `argmemonly` attribute to a function. It tells the alias analysis that the aforementioned function can only access memory locations which are given to it in its argument list. To add this attribute, we run a preliminary pass on the code to mark functions with this attribute. Since we know MPI calls only read or write the memory locations made known to them through the argument list, it is safe to add this attribute on them. It is probable they modify some variables internal to their implementation, but these are usually not accessible and usable in the source code. This is, however, not so obvious for other function calls. The resolution of their dependencies might require extensive interprocedural, and cross-module or

Table 5.1 – Matching statistics

	MCB	Pennant	Lammps	miniFE	miniMD
Number of initiation calls	6	4	114	5	7
True negatives	0	0	105	4	7
False positives	6	4	9	1	0

link-time analyses. In the end, only MPI functions are marked.

5.3.2 Results

The experiments have been carried on five benchmarks: miniMD, miniFE, Lammps, Pennant, and MCB. The first two are mini-apps from the Mantevo project[75]. They are scaled down versions of larger benchmarks, and focuses on a specific pattern of computation. The three remaining benchmarks are CORAL codes. They are significantly larger in scale, and aim at imitating production grade programs, with their computation and communication patterns. Full details on these codes are provided in Appendix A.

Matching of nonblocking calls

We first determine if our approach is able to correctly match nonblocking calls together. The Table 5.1 exposes the results obtained with our implementation of the matching pass. The "true negative" refers to nonblocking communications which we verified their correct matching with a manual analysis of the code, the dependencies, and the context of the function call. The "false positive" refers to the nonblocking communications our verification pass deemed as incorrectly matched, yet the communication can be safely matched through a manual verification of the code. We have not found any "true positive" or "false negative" as we did not identify any faulty nonblocking communication in these codes, and we did not introduce mutations.

Our implementation is able to match all of the nonblocking calls present in miniMD, and most of those in miniFE and Lammps. However, of all of the nonblocking communications found in MCB and Pennant, none could have been matched.

The reason of these false positives comes either from the failure of determining if two nonblocking communication calls refer to the same section of a request array, or from the intra-procedural nature of our method. Listing 5.1a contains

an example of a failed matching of an initiation call to its completion call, from Pennant, `Mesh.cc`. Our matching method is unable to associate the `MPI_Irecv` at line 620 to the `MPI_Waitall` at line 642. The initiation call is looped, and only relying on the `checkPath` function is not enough. The range of the used requests in the array must be determined with the help of the `checkIteration` function. The request array is being accessed with the `slvpe` variable, which happens to be the loop counter. The loop starts at 0 and ends at `numslvpe` with a step of 1, without any early exit or early back-edge. As a consequence, the `MPI_Irecv` initiates requests from indexes 0 to `numslvpe` $-$ 1, which matches with the range of read requests by the `MPI_Waitall` at line 642. The matching should be correct.

However, `numslvpe` is a global variable in the scope of this function, and LLVM struggles to determine if both uses of this variable will access the same value. This is highlighted in the SSA-form of the IR of function `parallelGather`, partially shown in Listing 5.2. It is the IR of the `for.cond.cleanup29` basic block containing the Waitall call. We can observe the computation of the address of `numslvpe` being performed in the GEP at line 118 from the class. This address is then used in line 120 to load the value into `%17`, which is then used in line 122. This computation is then performed once more at line 123 to be loaded in 125 and used in the `MPI_Waitall` line 127. Instead of using the previously loaded value, it loads another one. This might indicate that the alias analysis thinks the call at line 122 can modify the value of `numslvpe`, either because it is passed as an argument to it, or as a side effect, since `numslvpe` is a global variable. Moreover, instead of reusing an older load for the use in line 122, it recomputes the address and reloads the variable. This indicates that LLVM thinks it was modified between an older load, for example to compare the first loop counter, and line 118. To sum up, LLVM is unable to determine if the two loads of this variable would yield the same value. This makes our pass believe that the accessed requests are not within the same range, thus incorrectly flagging this nonblocking communication as unsafe.

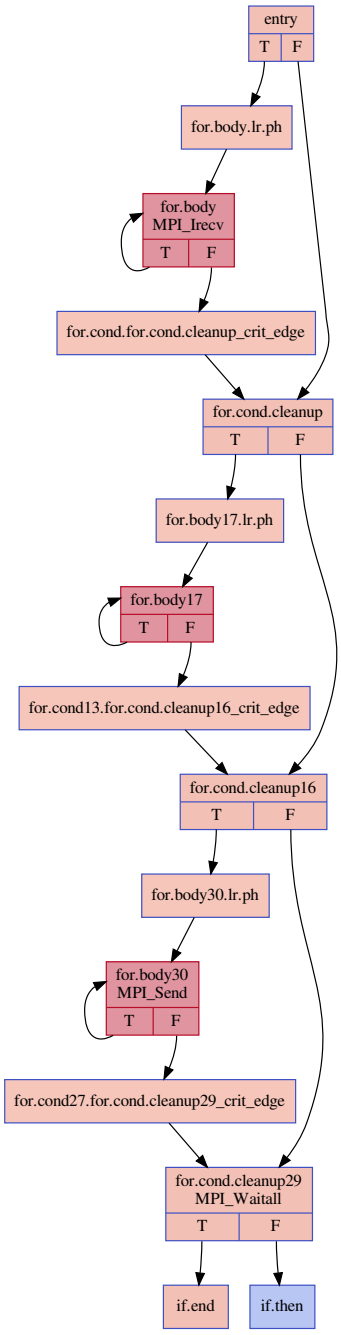
Detection of race conditions

The detection of race conditions, including the detection of request overwriting, is performed only for nonblocking communications which have been successfully matched. Thus, we will only consider results from Lammps, miniFE, and miniMD. The results are given in Table 5.2. Our implementation of the detection of race conditions display a very low accuracy. As mentioned in the matching of nonblocking calls, this inaccuracy can be caused by the lack of an interprocedural analysis to safely determine the aliasing of two pointers. The alias analysis chain we use might also not be able to correctly handle arrays, and considers that each access to

Figure 5.1 – Penannt, `Mesh.cc`, matching failure

```
601 template <typename T>
602 void Mesh::parallelGather(
603         const T* pvar,
604         T* prxvar) {
605 #ifdef USE_MPI
606     // This routine gathers slave values for which MYPE owns the
          masters.
607     const int tagmpi = 100;
608     const int type_size = sizeof(T);
609 // std::vector<T> slvvar(numslv);
610     T* slvvar = Memory::alloc<T>(numslv);
611
612     // Post receives for incoming messages from slaves.
613     // Store results in proxy buffer.
614 // vector<MPI_Request> request(numslvpe);
615     MPI_Request* request = Memory::alloc<MPI_Request>(numslvpe);
616     for (int slvpe = 0; slvpe < numslvpe; ++slvpe) {
617         int pe = mapslvpepe[slvpe];
618         int nprx = slvpenumprx[slvpe];
619         int prx1 = mapslvpeprx1[slvpe];
620         MPI_Irecv(&prxvar[prx1], nprx * type_size, MPI_BYTE,
621                 pe, tagmpi, MPI_COMM_WORLD, &request[slvpe]);
622     }
623
624     // Load slave data buffer from points.
625     for (int slv = 0; slv < numslv; ++slv) {
626         int p = mapslvp[slv];
627         slvvar[slv] = pvar[p];
628     }
629
630     // Send slave data to master PEs.
631     for (int mstrpe = 0; mstrpe < nummstrpe; ++mstrpe) {
632         int pe = mapmstrpepe[mstrpe];
633         int nslv = mstrpenumslv[mstrpe];
634         int slv1 = mapmstrpeslv1[mstrpe];
635         MPI_Send(&slvvar[slv1], nslv * type_size, MPI_BYTE,
636                 pe, tagmpi, MPI_COMM_WORLD);
637     }
638
639     // Wait for all receives to complete.
640 // vector<MPI_Status> status(numslvpe);
641     MPI_Status* status = Memory::alloc<MPI_Status>(numslvpe);
642     int ierr = MPI_Waitall(numslvpe, &request[0], &status[0]);
643     if (ierr != 0) {
644         cerr << "Error:␣parallelGather␣MPI␣error␣" << ierr <<
645                 "␣on␣PE␣" << Parallel::mype << endl;
646         cerr << "Exiting..." << endl;
647         exit(1);
648     }
649
650     Memory::free(slvvar);
651     Memory::free(request);
652     Memory::free(status);
653 #endif
654 }
```

(a) `parallelGather` function



(b) CFG of 5.1a

```
116 for.cond.cleanup29: ; preds = %for.cond27.for.cond.cleanup29_crit_edge, %for.cond.cleanup16
117 ; 20 = MemoryPhi({for.cond.cleanup16,22},{for.cond27.for.cond.cleanup29_crit_edge,7})
118   %numslvpe45 = getelementptr inbounds %class.Mesh, %class.Mesh* %this, i64 0, i32 22
119 ; MemoryUse(20) MayAlias
120   %17 = load i32, i32* %numslvpe45, align 4, !tbaa !11
121 ; 5 = MemoryDef(20)
122   %call46 = call %struct.ompi_status_public_t* @_ZN6Memory5allocI20ompi_status_public_tEEPT_i(i32
          %17)
123   %numslvpe47 = getelementptr inbounds %class.Mesh, %class.Mesh* %this, i64 0, i32 22
124 ; MemoryUse(5) MayAlias
125   %18 = load i32, i32* %numslvpe47, align 4, !tbaa !11
126 ; 6 = MemoryDef(5)
127   %call50 = call i32 @MPI_Waitall(i32 %18, %struct.ompi_request_t** %call2, %
          struct.ompi_status_public_t* %call46)
128   %cmp51.not = icmp eq i32 %call50, 0
129   br i1 %cmp51.not, label %if.end, label %if.then
```

Figure 5.2 – Pennant, `Mesh.cc`, excerpt of IR generated from the original source code

Table 5.2 – Data race detection statistics

	Lammps	miniFE	miniMD
Number of matched nonblocking communications	105	4	7
True positives	3	6	0
False positives	388	40	6
Total	391	46	6

a single element as an access to the whole array. It results in an overly conservative dependency analysis, flagging too many memory accesses as a potential threat to the safety of the communication.

Nonetheless, upon closer inspection, the verification pass revealed several potentially dangerous situations in miniFE and Lammps. Note that it operates on the IR. This implies that one line in the source code can translate to multiple instructions in the IR, thus increasing the number of detected race conditions.

Our implementation of our verification method successfully outlined a race condition in miniFE. Figure 5.3 shows the relevant nonblocking communication, which is the nonblocking point to point reception at line 259. It is matched with the completion call at line 276, since both are in distinct loops and use the same section of the request vector, from index 0 to `num_send_neighbors` excluded. As a consequence, its overlapping interval starts after line 260, and ends before line

Figure 5.3 – miniFE, `make_local_matrix.hpp`, race condition on the reception buffer

```
257   std::vector<MPI_Request> request(num_send_neighbors);
258   for(int i=0; i<num_send_neighbors; ++i) {
259     MPI_Irecv(&tmp_buffer[i], 1, mpi_dtype, MPI_ANY_SOURCE, MPI_MY_TAG,
260               MPI_COMM_WORLD, &request[i]);
261   }
262
263   // send messages
264
265   for(int i=0; i<num_recv_neighbors; ++i) {
266     MPI_Send(&tmp_buffer[i], 1, mpi_dtype, recv_list[i], MPI_MY_TAG,
267              MPI_COMM_WORLD);
268   }
269
270   ///
271   // Receive message from each send neighbor to construct 'send_list'.
272   ///
273
274   MPI_Status status;
275   for(int i=0; i<num_send_neighbors; ++i) {
276     if (MPI_Wait(&request[i], &status) != MPI_SUCCESS) {
277       std::cerr << "MPI_Wait␣error\n"<<std::endl;
278       MPI_Abort(MPI_COMM_WORLD, -1);
279     }
280     send_list[i] = status.MPI_SOURCE;
281   }
```

Figure 5.4 – Lammps, `fix_neb_spin.cpp`, request overwriting

```
701   if (ireplica > 0) {
702     MPI_Irecv(xprev[0],3*nlocal,MPI_DOUBLE,procprev,0,uworld,&request);
703     MPI_Irecv(spprev[0],3*nlocal,MPI_DOUBLE,procprev,0,uworld,&request);
704   }
705   if (ireplica < nreplica-1) {
706     MPI_Send(x[0],3*nlocal,MPI_DOUBLE,procnext,0,uworld);
707     MPI_Send(sp[0],3*nlocal,MPI_DOUBLE,procnext,0,uworld);
708   }
709   if (ireplica > 0) MPI_Wait(&request,MPI_STATUS_IGNORE);
```

276. Inside this interval, the communication buffer, here `tmp_buffer` must not be accessed, since it is an inbound value. Yet the `MPI_Send` at line 266 reads at least a part of this array. Indeed, we noticed both loops start at 0, and thus the first few elements of the array are potentially going to be read while their value is still unknown. In the end, it is not known if the `MPI_Send` will send the value that has been received, or the value as it was defined before line 259. There are multiple solutions to fix this code. One is to move the sending operation out of the overlapping interval, before or after depending on what value of `tmp_buffer` must be sent. This will however end up in the shrinking of the overlapping interval of the `MPI_Irecv`, which is not desirable performance wise. The other solution would be to use a different buffer to receive data in. This results in a higher memory consumption, but preserves the length of the overlapping interval.

Figure 5.4 displays a dangerous usage of a request, including its overwriting, found by our verification pass. In this code, both `MPI_Irecv` are matched with the `MPI_Wait` at line 709. We can observe that both initiations are being called if `ireplica` is positive, and the completion call is also protected by the same condition. Furthermore, all values from the comparison are and remain the same, they are not being modified by any instruction between lines 701 and 709, thus ensuring the execution of the completion call after the initiations, and ultimately their matching. However, while the second `Irecv` is perfectly safe, the first has its request overtaken by the second. As a consequence, information about the non-blocking initiation at line 702 is potentially lost, preventing its completion, which can result in a deadlock. In order to fix this situation, the second communication must use a different request. Then it is possible to add another completion call for this new request, or replace the existing `MPI_Wait` by a `Waitall` call to catch both requests.

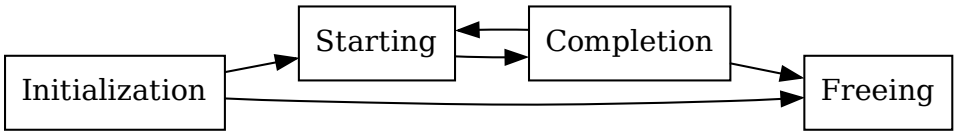


Figure 5.5 – Possible state transitions for a persistent MPI communication

5.4 Towards the correction of persistent communications

Persistent communication also adopted a split form, alike but more complex than their nonblocking counterparts. Instead of having only an initiation call and a completion call, persistent communications are split in four distinct functions: the initialization, the starting, the completion, and the freeing calls. The Figure 5.5 illustrates the possible transitions between persistent MPI calls. The main interest of this form of communication is the ability of initializing the request only once for multiple communications. That is the role of the initialization call: it attaches the communication buffers to the request. The communication is only performed upon reaching the starting call. The completion call, which is also of the form `Wait` and their `all`, `any`, and `some` versions, certifies the completion of the communication. They deactivate the persistent request. `Test` and its versions also apply to persistent communications. The difference lies in the ability to rerun the communication, with potentially different values as contents of the communication buffers. Eventually, once the sequence of exchanges is done, the persistent request must be deallocated. This operation is performed by the freeing call. This form of communication is particularly useful in loops of communications: the request and the communication buffers are initialized before the loop and each iteration starts and completes an exchange, possibly with different data each time. Finally, the request is only freed after the loop.

The verification of persistent communications can be performed using the same techniques which were laid out during this Chapter. Instead of matching an initiation with multiple completion calls, the matching must follow the pairings illustrated by Figure 5.5. Beginning with the initialization call, the analysis would need to find a matching call that is either a starting or a freeing call. To match

these calls, it is also possible to use the request object by identifying it with the methods described in Section 5.1. Once the potential candidates have been determined, both the path and flow sensitive approaches can be applied to determine the capable starting and freeing calls. Finally, the matching calls can be found by performing the same BFS-like traversal of the CFG, thus ensuring the initialization call is followed by either a starting call or a freeing call.

At this point, contrary to nonblocking communications, the matching method for the whole persistent communication is not yet complete. The correct follow-up for the starting and completion calls have not been checked yet. This is done by repeating the same matching procedure. The starting call must be followed by a completion call. The completion call must be followed by either a starting call, or a freeing call which indicates the end of the persistent communication window.

Communication buffers and requests corruptions can also cause errors in persistent communications. Their detection relies on the same principles as those described in Section 5.2, only the intervals to check change. The overlapping interval, which is the code portion during which the exchanging part of a communication can be executed in the background, is bounded by the starting and completion calls. Data races must be avoided in this interval, and the methods we proposed earlier can be applied between these two calls. The treatment of requests slightly differs. The communication is "alive" not only during its "exchange" section, but also between the initialization and the freeing calls. During this whole duration the request must remain protected, and only be accessed by the calls from the persistent communication, which are the starting and completion calls.

To sum up, the method we propose for the static verification of nonblocking communications can be adapted for persistent communications with minimal changes. The differences can be found in the matching it needs to perform, and the intervals in which the detection of race conditions must be performed. Contrary to the verification of MPI nonblocking communications, the propositions of this section have not been implemented.

5.5 Discussion and limitations

During this Chapter, we have proposed a method to verify the correctness of nonblocking communications in a program using information and analyses available at compile-time. This method is able to verify the matching of initiation and completion calls, and it is able to find potential race conditions in the overlapping intervals of nonblocking communications. We have implemented this method as a pass in LLVM. It has successfully detected several concrete situations where race

conditions are possible. However, it has a poor accuracy in the detection of race conditions, and suffers several limitations in the matching.

The fact that our pass is an intra-procedural pass is one of its main shortcomings. It prevents us from matching nonblocking communications across function boundaries, and it also hinders us in correctly determining the safety of a matching or a memory access inside an overlapping interval. Indeed, the impacts of a function call on the memory is unknown in the intra-procedural scope. The resolution of these dependencies might require extensive interprocedural, and cross-module or link-time analyses. Yet, there are function calls for which their behavior are known at compile-time in an intra-procedural context. MPI operations are an example of such functions. It is possible to tell LLVM that these functions only access memory locations that are passed as an argument thanks to the `argmemonly` attribute.

Despite the addition of this attribute to MPI calls, the alias analysis still struggles with global variables. It considers all global variables can be modified by any function call. Listing 5.1a, which we described in the previous Section, is a prime example. LLVM is unable to determine if `numslvpe`, a global variable, has been modified between the access at line 616 to serve as the upper bound of the initiation loop, and the access at line 642 in the `MPI_Waitall`. Yet, no call site nor instruction have a dependency relationship with this variable.

In order to pinpoint the cause leading to the missed matching, we have slightly modified the source code, and regenerated the IR. The modifications we introduced consists in copying `numslvpe` into a temporary `int` local variable between lines 614 and 615 in Listing 5.1a. Then, all subsequent uses of `numslvpe` are replaced with this temporary local variable. The resulting IR is shown in Listing 5.6a. We observe that this local variable is now only loaded once, at line 11 in the new IR. All subsequent uses refer to it by using `%1`, as partially shown in lines 13, 14, 114, and 116 which is the Waitall. By virtue of the SSA form, all these uses refer to the same value of `numslvpe`, allowing us to match the nonblocking communications. This lead us to think that LLVM believes the function calls in this code might have a side effect on `numslvpe`.

While the suspicion of a side effect on global variables in function calls is a reasonable assumption to make in the lack of an interprocedural analysis, in this situation it triggers a false positive. Looking at the code, the static `alloc` function from the `Memory`, defined in `Memory.hh`, only allocates memory location without modifying any variable. Similarly, MPI communications are not likely to have any side effect on global variables. One solution would be to limit the use of global variables, and copy their values into local buffers to mitigate risks and facilitate

static analyses.

One of the other limitations of this approach is the availability of the debugging and high level information of the source code. In LLVM IR, the source code location is not kept unless it was compiled with the `-g` option. The name of variables is also lost. All of this complexifies the process of informing developers about issues in their code. A better solution would be to work at the front-end level, where information about the source code is still complete and available. However, this would render the matching of nonblocking communications unavailable to other IR analysis and transformation passes.

Figure 5.6 – Pennant, `Mesh.cc`, IR of function `parallelGather`

```
3   entry:
4     %numslv = getelementptr inbounds %class.Mesh, %class.Mesh* %this, i64 0, i32 24
5   ; MemoryUse(liveOnEntry) MayAlias
6     %0 = load i32, i32* %numslv, align 4, !tbaa !2
7   ; 1 = MemoryDef(liveOnEntry)
8     %call = call double* @_ZN6Memory5allocIdEEPT_i(i32 %0)
9     %numslvpe = getelementptr inbounds %class.Mesh, %class.Mesh* %this, i64 0, i32 22
10  ; MemoryUse(1) MayAlias
11    %1 = load i32, i32* %numslvpe, align 4, !tbaa !11
12  ; 2 = MemoryDef(1)
13    %call2 = call %struct.ompi_request_t** @_ZN6Memory5allocIP14ompi_request_tEEPT_i(i32 %1)
14    %cmp33 = icmp slt i32 0, %1
15    br i1 %cmp33, label %for.body.lr.ph, label %for.cond.cleanup
```

```
111 for.cond.cleanup28: ; preds = %for.cond26.for.cond.cleanup28_crit_edge, %for.cond.cleanup15
112 ; 20 = MemoryPhi({for.cond.cleanup15,22},{for.cond26.for.cond.cleanup28_crit_edge,7})
113 ; 5 = MemoryDef(20)
114   %call44 = call %struct.ompi_status_public_t* @_ZN6Memory5allocI20ompi_status_public_tEEPT_i(i32
          %1)
115 ; 6 = MemoryDef(5)
116   %call47 = call i32 @MPI_Waitall(i32 %1, %struct.ompi_request_t** %call2, %
          struct.ompi_status_public_t* %call44)
117   %cmp48.not = icmp eq i32 %call47, 0
118   br i1 %cmp48.not, label %if.end, label %if.then
```

(a) Generated from the modified source code

Chapter 6

Automatic Exposition of Overlapping Potential

One of the main strengths of nonblocking communications is their ability to allow overlapping of communications with other operations, for example computations. This results in a better use of computing resources, which in turn leads to lower execution times of MPI programs. However, the use of this type of communications is still marginal. Codes still prefer blocking communications, and some only use nonblocking point-to-point communications to avoid deadlocks caused by disordered sends and receives. Not only the nonblocking operations are more complex to use since they are split between an initiation call and one or multiple completion calls, they are also more sensitive to programming errors which can lead to deadlocks and data corruption. These drawbacks are as many additional points the developers will have to watch for during the development of a program.

In order to completely relieve these burdens from the developers, we propose in this Chapter a method to automatically create and increase the overlapping potential of MPI communications in a function. First, we detect all blocking communication calls in a code, and transform them into their nonblocking version. This implies the insertion of a completion call and the creation of a request object that will tie the initiation with the newly inserted completion. Secondly, we rearrange the structure of the code in order to maximise the length of the overlapping intervals. The reorganization of the code is performed by moving the dependencies of the communication along with the initiation and completion calls. This step is also performed on existing nonblocking calls with the help of the matching method defined in Chapter 5.

Section 6.1 covers the boundaries of the overlapping intervals we create. In

Section 6.2, we describe our transformation method and its inner working. In Section 6.3 we introduce our approach to measure the effects of the automatic transformation method, which is then applied on several benchmarks.

The contributions of this Chapter have been published in an article at the Compiler-assisted Correctness Checking and Performance Optimization for HPC workshop (C3PO) in 2020[76].

6.1 Defining the boundaries of overlapping intervals

The goal of this study is to create and extend overlapping possibilities in a program with MPI communications. The method we propose in this Chapter is based on a reorganization of the code, more precisely we will move instructions of a function so that the communications are overlapped by independent computations. While Section 6.2 will focus on the movement of statements, this Section gives details on how the boundaries of overlapping intervals are defined in this study.

Existing transformation methods to create overlapping intervals in MPI codes are limited to the nearest dependent statement[72]. The solution we propose does not consider those statements as boundaries for overlapping intervals. Instead, these statements will be moved along with the nonblocking calls. This allows us to reach independent statements that might be beyond the dependencies. Thus the code is reorganized by gathering the independent statements inside the overlapping interval, while the dependencies are gathered outside the overlapping interval. While data dependencies are no longer limitations for the overlapping potential in our approach, there are still several obstacles related to the control flow. In the coming paragraphs, we detail each kind of boundary our transformation method faces.

Limits of the calling function

The application of the method we propose in this Chapter is limited to the body of the function calling the nonblocking communication. Considering the interprocedural context would require additional analyses which were not the focus of this study, including an interprocedural alias analysis to determine the dependent statements, and the handling of recurrent function calls. Unless the function has been inlined by a previous optimization pass, it is not possible to extend the overlapping interval across the boundaries of the calling function.

As a consequence, if no other limitation obstruct the movement of the nonblocking calls, the initiation call would need to be inserted above the first statement of the function, and the completion call right above the relevant procedure exiting call.

Other MPI communication calls

There are situations where an MPI nonblocking call cannot be moved beyond another MPI function call. Such situations are the displacement of an initiation call beyond another initiation call regardless of their dependency relationship, the displacement of a completion call beyond another completion call regardless of their dependency relationship, the displacement of dependent MPI calls and of several functions such as `MPI_Init`, `MPI_Finalize`, or `MPI_Barrier`. The first two functions define the MPI environment, which is the scope inside which communication calls are allowed. As for the `Barrier`, since it does not have any dependency, the transformation method is unable to determine a correct insertion position, and is unaware of the reason why a synchronization would be needed. Consequently, these calls should not be moved nor overlapped.

The remaining constraints, which are dependent MPI calls, other initiation calls when displacing an initiation call, and other completion calls when displacing a completion call, exist for two reasons. First is the order of collective communications. All MPI processes in a communicator must initiate the collective primitives in the same order, regardless of their blocking or nonblocking nature. In order to abide to this rule and avoid breaking the order of collective calls, the movement of initiation calls has to be limited. The second reason is to avoid the destruction of other overlapping intervals.

```
1 MPI_Ibcast(&A, ..., &req);
2 MPI_Bcast(B, ...);
3 B = ...;
4 MPI_Wait(&req, ...);
```

Figure 6.1 – Example of a stopped analysis because of an MPI communication call

Listing 6.1 shows a situation where the optimization of one communication can harm another. Indeed, the automatic optimization method cannot determine which communication should prevail. As a consequence, it handles MPI communications from a function in the order in which they appear in the IR and thus, as a default, prioritizes the first MPI communications. In this example, let us

suppose it just transformed the broadcasting call on variable `A`, resulting in the shown snippet of code, and is about to analyze the broadcasting call on variable `B`. Theoretically nothing prevents us from overlapping the newly inserted `MPI_Wait` since it is matched with the first broadcasting call, which is independent of the second broadcast. However, in doing so, it will have to move line 3 out of the overlapping interval of the first broadcast, rendering it empty, thus undoing the previous transformation.

To summarize, when moving the initiation call, it is only allowed to pass through independent completion calls. When moving the completion call, it is allowed to pass through other independent initiation calls (both blocking and non-blocking). The encompassing of a communication is permitted, resulting in total or partial overlapping of intervals. However, an inner communication cannot break out of the interval of another.

Other call sites

As a whole, the other call sites are also boundaries for the overlapping interval, partly because of the lack of interprocedural analyses.

The first reason is because of the side-effects a function call can have on the dependencies of the communication. Even if the dependency is not explicitly given as an argument to the call site, it is possible for the called function to access or modify it through memory operation or if it is a global variable or a class member. It is possible to embark the call site along with the other dependencies. However, since the effect of the function on other values is unknown, it would require us to embark every further statement, thus nullifying the point of overlapping.

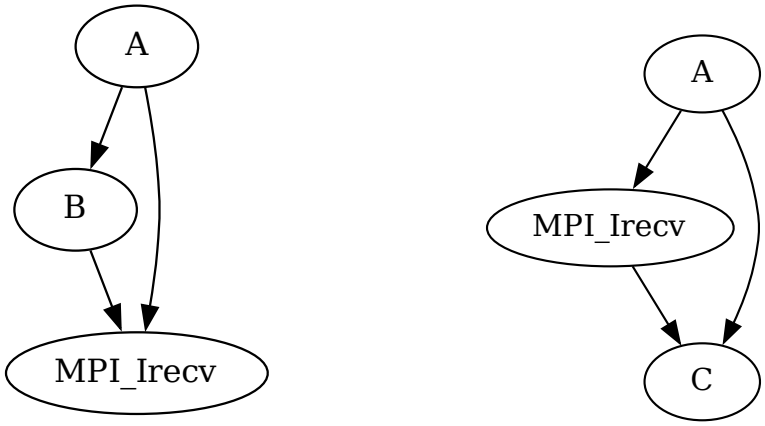
The second reason is the potential presence of collective MPI communications. As explained above, overlapping the call site can break the order of collective calls.

The last reason has more to do with the semantics of the program, and behaviors that cannot be determined at compile-time. Let us consider a function whose goal is to provide a timestamp that will be used to compute the execution time of a portion of code. These functions usually do not have any argument, and the compiler would fail to detect its dependencies. Yet, moving communication dependencies and other computations beyond these timing functions would break the measurements, and thus the overall behavior and expected output of the program.

Dependencies in control flow structures

Those are the dependencies which are in a control flow scope whose nesting level is deeper than that of the relevant nonblocking call. This is the case when a

dependency is found in a conditional branch or a loop, as shown in Figure 6.2a. In these situations, our approach would try to overlap the control flow structure. In this Figure, it would try to overlap the basic block B. However, if B contains a dependency, then overlapping it would lead to a race condition. Yet, this dependency cannot be moved either, since it is only executed if the control flow follows a path that includes the B.



(a) Overlapping conditional branches

(b) Limitation by a fork node

Figure 6.2 – Examples of MPI operations and control flow scopes

In these situations, the overlapping boundary is limited at the latest statement before entering this scope. In the example, the `MPI_Irecv` would need to be inserted right above the first instruction of its basic block, while remaining inside the block.

Hoisting out of a control flow structure

Inversely, if the communication to overlap is inside a control flow structure such as a loop or a conditional branch, then it cannot be hoisted to the outer scope. This is illustrated by Figure 6.2b. The `MPI_Irecv` is only executed if the branching condition in node A allows it. Hoisting the call up in that basic block would allow its execution in every circumstance, regardless of the branching

condition. A similar problem exists when communications are inside loops. They cannot be hoisted out of the loop since this transformation would change their execution count. In these situations, the nonblocking calls have to remain in their control flow scope.

6.2 Increasing overlapping potential through automatic reordering of instructions

Using nonblocking communications puts an additional burden upon the developers of MPI applications because of their complexity and because they offer less security guarantees. The goal of this contribution is to define a method to automatically add overlapping potential to an existing code, with minimal human intervention to reduce the workload for the developers. This is first done by transforming every existing blocking calls into their nonblocking version. Then, for all nonblocking calls found in a function, whether they were newly inserted or preexisting, the method we describe reorganizes the code in a way that would maximize the length of overlapping intervals. This step is done by moving dependencies along with the initiation and completion calls.

6.2.1 Enabling overlapping potential

Let us begin by considering the blocking calls from a function. Their overlapping potential is, by definition, null. In order to allow overlapping of the communication, we need to transform the blocking call into its nonblocking version. The conversion to the nonblocking form requires the insertion of three elements: the request, the initiation call, and the completion call.

First, the request is inserted right above the existing blocking call. This is performed by creating an instruction which will allocate memory on the stack for this request. Once the request has been inserted in the basic block containing the communication, the initiation and completion calls can be inserted. In MPI, the prototype of nonblocking initiation functions share many similarities with their blocking version. The argument list is the same, with the addition of a pointer to the request in the nonblocking version. Thus, a new call instruction is created. The called function is the nonblocking form of the targeted blocking call. The argument list is copied over, and is appended with a pointer to the newly inserted request. The old call site is then swapped with it. Finally, an `MPI_Wait` is inserted. This is done by creating a call site right below the previously inserted initiation. Its

argument list is made of a pointer to the same request, and a `MPI_STATUS_IGNORE` as the status.

There is an exception when replacing an `MPI_Recv`. The point-to-point reception has a distinctive characteristic of accepting an `MPI_Status*` as its last argument. However, its nonblocking form, `MPI_Irecv` delegates the handling of the status to the matched completion calls. Consequently, we first need to remove the pointer to the status from the original argument list before appending the request in. It is given to the completion call instead, which will use it rather than the `MPI_STATUS_IGNORE` default keyword, thus preserving the subsequent uses of the status.

6.2.2 Creating and increasing overlapping potential

The transformation of blocking calls leaves no blocking communications behind in the code of a function. However, those are still devoid of overlapping potential, since the completion call immediately follows the initiation call. There are no computations to hide the communication. Henceforth, the goal of this Section is to effectively create and extend overlapping potential, for both newly inserted and existing nonblocking communications.

The creation and extension of overlapping potential is done by increasing the length of the overlapping interval. This is done by moving both the initiation and completion calls so that computation ends up inside this interval. This method must preserve the dependencies and the semantics of the program.

First, the dependency slice of the communication must be built. As described in Section 4.1.2, we build a dependency graph. It contains all instructions that read or write to one of the arguments of the communication, be it a communication buffer or the metadata such as the root identifier, size of the message, or even the type of the data. Every variable required by the communication must be identified. The dependency slice is iteratively built, which means that it will also cover the dependencies of those variables, and so on.

With the identification of the communication dependencies, it is possible to move the nonblocking calls while preserving them and putting independent statements inside the overlapping interval. Algorithm 4 describes the moving of the initiation call and its dependencies. To that end, the notion of immediate dominance is extended from basic blocks to statements. A statement s_1 dominates another statement s_2 if s_1 belongs to a basic block dominating the basic block containing s_2 or, when s_1 and s_2 are in the same basic block, if s_1 precedes s_2. The post-dominance is also extended in a similar fashion.

Algorithm 4 Finding an insertion point for the initialization call

```
procedure INSERT_MPI_INIT_CALL(function)
    for all mpi_call ∈ function do
        list_stmt_init ← ∅
        V ← getDependencySlice(mpi_call)
        stmt ← mpi_call.get_stmt()
        while stmtImmovableInit(stmt, mpi_call.get_stmt(), V) = false do
            stmt ← immediate dominator(stmt)
            if stmt ∈ V then
                list_stmt_init ← list_stmt_init ∪ {stmt}
        insert_init ← stmt
        Move statements from list_stmt_init to the point of the code where stmt
is the immediate dominator, and insert the init call
procedure STMTIMMOVABLEINIT(stmt, call_stmt, V)
    if stmt is the first statement of the function then return true
    for all tstmt between stmt and its immediate dominator do
        if tstmt ∈ V then return true
    if call_stmt is between stmt and its immediate post-dominator then return
true
    if stmt is a call site then
        if not IsSafeMPI_up(stmt) then return true
    return false
```

Algorithm 4 consists in visiting every statement, going from immediate dominator to immediate dominator until one of the stopping points defined in Section 6.1 is found. The function `stmtImmovableInit` determines if the transformation method has reached such a point. The checks performed in that function correspond to one of those boundaries.

The first, which checks if the relevant statement is the first of the function, determines if we have reached the top of the function calling the communication.

The second checks if there is a dependency between the statement and its immediate dominator. It is the situations illustrated in Figure 6.2a. Assuming the insertion method has reached the top of the basic block containing the `MPI_Irecv`, its immediate dominator would be node A. Thus, to determine if the communication can be moved to that basic block, the safety of B must be ensured.

The next step determines if the insertion method is trying to go to the outer control scope. The insertion point must be control equivalent to the original call

site location. If the processed statement is not post-dominated by the initiation call, then the equivalence is broken and the statement is in another control scope, making the insertion site unsafe. As shown in Figure 6.2b, if the insertion method reached the bottom of the basic block A, then its immediate post-dominator would be C, and not the communication anymore. The previous two checks are also applicable on loops.

Finally, it checks if the statement is a call site. If so, and if the statement is not one of the "overlappable" functions by the initiation call, then the upper bound for the interval must be set below that statement. Dependencies and nonblocking calls should then be inserted there.

Data dependencies, which are put into the list V, are not boundaries. Instead, each time the method encounters an element of that list, it is appended to `list_stmt_init`, which is the queue of statements that need to be moved along with the initiation call. It also contains the newly inserted request. Once the boundary has been found, the initiation call is moved below it. Then, starting from the head of the queue, dependencies are inserted above the initiation call, and then above each other. This preserves the order of the dependencies.

The same method is applied on the completion call, with the post-domination replacing the domination, and vice-versa. The called function to check the "overlappability" of a call site invoking an MPI function (IsSafeMPI_up in Algorithm 4 for the initiation call) should also reflect the limitations set in Section 6.1.

The transformation method we described in this Section preserves the execution context of the communications, their dependencies and their execution order, and is conservative in regards of other MPI communication calls and of other function calls that might contain side-effects on dependencies or MPI communications, all the while reorganizing the code to favorise the overlapping intervals. Indeed, all independent statements, i.e. statements that were not in V, remain untouched, thus ending up between the initiation call and the completion call.

Processing of existing nonblocking calls

The matching of the newly inserted nonblocking calls is straightforward since the completion is right below the initiation call. The matching of existing nonblocking communications is less obvious, and the method proposed in Chapter 5 enables this. However, we only consider nonblocking communications whose matching is "simple", which are communications where an initiation call site is matched to one and only one completion call site. Inversely, the said completion call must complete that one initiation alone.

It is possible for nonblocking communications to be completed by multiple

distinct static completion points, as long as they post-dominate the initiation as a set. It is also possible for a completion call, for example `MPI_Waitall`, to terminate multiple distinct communications. As a consequence, we would need to consider the dependencies of all of those initiation calls, and move those calls accordingly. Since the "simply" matched nonblocking communications behave the most similarly to the converted ones, we decided to focus on those in this study.

The handling of the "simply" matched nonblocking communications is similar on almost all points to the handling of the newly inserted blocking communications. Only the creation step is skipped (creation of the request, initiation, and completion calls). The content of the existing overlapping interval is assumed to be safe. This can be ensured by executing the verification pass defined in Chapter 5 beforehand.

6.3 Applying automatic increase of overlapping potential on benchmarks

Very similarly to the verification pass described in Section 5.3.1, the transformation method we propose in this Chapter has been implemented as an LLVM 12 plugin pass. It operates on the IR of each function in a translation unit. The IR is also generated using Clang. This implementation uses the same pass pipeline to adapt the code to facilitate its handling, and the same alias analyses to build the dependencies. The transformation pass is then applied on this prepped IR with `opt`. This results in a modified version of the IR where all communications are transformed into their nonblocking version, and where overlapping intervals have been automatically extended according to the method we previously described. Finally, this IR is given to Clang, which detects the file type to carry on the compilation chain, resulting in a binary object, ready to be executed or linked.

6.3.1 Measuring the overlapping of a nonblocking communication

In order to evaluate the capacity of our transformation method to create overlap, we need to be able to measure the amount of overlap it introduces in a function. One solution would be to measure its execution time before and after performing the transformation. Indeed, an increase of the overlapping potential of MPI communications should lead to lower execution times thanks to the more efficient use of computing resources. However, studies on the ability of existing MPI imple-

mentations to progress nonblocking communications showed difficulties to improve performances[42]. Some implementations need to be specifically configured to enable asynchronous progression. Even with the correct setup, most situations show no performance improvement, and in several cases the use of nonblocking communications worsen the results compared to the use of their blocking version. The reason can be multiple: lack of a proper progression mechanism, or competition between computations and progression for resources.

Instead of measuring the execution time, we decide to evaluate the performance of our transformation method by using a static metric. We settled on the number of IR instructions between the initiation and the completion call. This corresponds to the length of the overlapping interval. Since multiple paths might exist in these intervals, we compute the length of the shortest path.

For example, when the interval contains conditional branches, it is, in most cases, impossible to determine which branch should be preferred since the branching condition might depend on the inputs of the program, and these branches might not have the same number of IR instructions. Similarly, if there is a path from the header of a loop to its exit node, which indicates the possibility of not executing the loop, then the length of the overlapping interval will not include any of the iterated instructions.

Consequently, when a communication has an overlapping interval of length n, then there is at least n IR instructions between its initiation and completion calls.

6.3.2 Results

We applied the transformation pass on the five benchmarks used in this study: miniMD, miniFE, Lammps, Pennant, and MCB. We first measure the length of existing overlapping intervals with a modified version of the verification pass we defined in Chapter 5. Since it tries to match all nonblocking communications, it is able to know which initiation is caught by which completion, and compute the shortest path between them. It is modified to output these distances in a table. The transformation pass is then applied. The intervals are measured again once each communication in a function has been transformed, and outputted in another table. We also collect statistics on the limitations of overlapping intervals.

The correctness of the transformations is ensured by executing the benchmarks before, and after the transformation pass. The results of both executions are compared, and for all five tested benchmarks, no discrepancies were found.

Table 6.1 and Table 6.2 respectively contain statistics on the length of overlapping intervals before and after the application of the transformation pass. It

succeed in significantly increasing the number of nonblocking communications, from a fourfold increase in Pennant to nearly thirty times more in Lammps. Note that while there was six nonblocking communications in MCB and four in Pennant, we could not compute their overlapping potential because of the initiation calls couldn't be matched together with their completion calls. However, with the addition on the transformed blocking communication and their overlapping interval, the median length of the intervals has been lowered. In each benchmark, the majority of those are at most one IR instruction long.

Table 6.3 shows statistics on the reasons why overlapping intervals are limited in these benchmarks. Each nonblocking communication has two stopping reasons: one for the upper bound of the overlapping interval (on the side of the initiation call), and one for the lower bound (on the side of the completion call).

The second to last row shows that our transformation pass encounter a substantive number of errors when analyzing the code. These errors prevent the transformation pass from further extending the overlapping interval. However, the placement of the initiation and completion calls, and their dependencies, is still valid, and will not cause critical errors when executing the program. These errors can come from failures to identify branches or loops, potentially because of a pattern in the control flow that our pass does not support, from unmatched and existing nonblocking communications, or from more complex matches that this pass does not yet handle.

6.4 Discussion

6.4.1 Measuring the overlap at compile-time

The method described in Section 6.3.1 to estimate the overlapping potential at compile-time has one major flaw. The measure we propose does not take the

Table 6.1 – Original overlapping interval length (in number of IR instructions)

	Lammps	Pennant	MiniMD	MCB	MiniFE
Nb. initiation calls	114	4	7	6	5
Total length	1916	0	88	0	44
Max length	65	0	15	0	13
Median length	12	0	14	0	11

Table 6.2 – Transformed overlapping interval length (in number of IR instructions

	Lammps	Pennant	MiniMD	MCB	MiniFE
Nb. initiation calls	3018	16	55	47	29
Total length	5413	15	188	44	84
Max length	84	4	17	3	13
Median length	1	1	1	1	1

Table 6.3 – Overlapping interval boundaries statistics

	Lammps	Pennant	MiniMD	MCB	MiniFE	TOTAL
Function Limits	271	0	4	3	4	282
Other MPI	2477	0	40	3	6	2527
Other call sites	5	2	2	2	16	31
Loop Dependency	46	0	0	0	0	46
Branch Dependency	935	0	19	8	16	978
Loop limits	160	12	3	1	6	182
Branch limits	852	14	16	61	2	944
Analysis errors	1290	4	26	16	8	1340
TOTAL	6036	32	110	94	58	6330

execution time of an IR instruction into account. Furthermore, the various kinds of IR instructions might not have an uniform execution time. For example, an `alloca`, a `store`, or a `load` might not take the same number of cycles to be processed. It might depend on the data types involved in the instruction, how it is translated to machine code, and the hardware capabilities to compute and store instructions and registers.

Ultimately, this prevent us from obtaining an estimate on the subsequent reduction of the execution time. Indeed, given the limitations of this measurement method, we are unable to estimate the execution time of an interval of n IR instructions. Moreover, it is also not possible to affirm that an interval of length $n + m$ will be better than an interval of length n in another snippet of code. It is possible that the instructions in the second interval weight more than the those in the first, resulting in more communication time being hidden.

Nonetheless, this method is still able to compare the gains in overlapping potential on each call site. An interval of length $n + m$, with m being a positive integer, on a call site will be better than an interval of length n for the same communication. The transformation method extends the intervals by moving the initiation and completion calls further away from each other. As a consequence, the first interval necessarily includes the first.

6.4.2 Automatic creation of overlapping intervals

As shown in Table 6.1 and Table 6.2, the transformation pass was able to expose overlapping potential in the tested benchmarks. It succeed in both extending the existing overlapping intervals, and creating new possibilities for overlapping previously blocking communications. Moreover, Table 6.3 contains hints to situations where the transformation pass can be improved. Almost a third of the reasons that prevent our approach from further expanding an overlapping interval is linked to conditional branches. Improving the handling of these situations with more complex transformations might conduct to more overlapping potential. The other noticeable obstacle is the presence of other MPI communications. It is caused by successions of communications whose interval obstruct each other. Other interesting trails for further improving the transformation method are an interprocedural analysis, and a better handling of loops. Although they represent a small amount of limiting factors, they can multiply the length of intervals. An interprocedural analysis can help, especially when MPI wrappers are involved.

However, the increase in the number of nonblocking communications does not translate to a similar increase in the overlapping potential of these benchmarks.

While it was able to introduce overlapping in places where there was no asynchronicity, the vast majority of the created intervals do not allow any possibility of hiding communication times. This is visible in the median length of these intervals. More than half of those are, at most, one IR instruction long. Furthermore, that instruction cannot be a call, because of the lack of an interprocedural analysis to determine its safety. Consequently, a one instruction long interval will most likely not be enough to hide the communication, resulting in a partial overlap at best.

Additionally, we do not have, at compile-time, an estimation of the duration of a communication. Yet, in order to have efficient overlapping, there must be enough computation to hide the communication. Thus, the length, or execution time, of the overlapping interval will not matter if the communication times are too short in comparison.

Chapter 7

Maximization of Overlapping Potential through Compiler-Assisted Code Transformation

In Chapter 6, we have proposed a method to automatically create and extend existing overlapping potential for both existing blocking and nonblocking MPI communications at compile-time. While it succeeds in exposing new overlapping possibilities in several benchmarks, and thus room for performance improvement, most of the created overlapping intervals offer very little opportunities. They often do not contain any instruction, other than the initiation and the completion calls, which means they are equivalent to a blocking communication. Nonetheless, the results from the automatic creation of overlapping intervals can give developers of MPI applications some insights to improve the asynchronicity of codes.

In this Chapter, we propose a method to further improve and maximize the overlapping potential gains achieved with the automatic transformation pass we defined in Chapter 6. It consists in providing feedback and suggestions of code transformations to the developers. It relies on information that can be retrieved by a static analysis tool to propose code modifications to create overlapping intervals. Since this approach is no longer a fully automatic transformation of the code, and given the size of HPC simulation codes, it is necessary to target the transformations to perform. This is why we also propose a method to select MPI communications whose overlapping interval will be optimized.

7.1 Providing code insights with compiler generated feedback

The provision of feedback, and more specifically of suggestions of code modifications, to the developers is another solution to the developers to help them in improving their code. The goal of the feedback and suggestions we define in this Section is to guide the developers in carrying on with the optimizations, and further improve the overlapping potential created and extended by the transformation pass described in Chapter 6.

7.1.1 Towards a tool assisted optimization method

The results of the automatic creation and extension of overlapping potential method we proposed in Chapter 6 reveals mixed results. Despite the benefits of an automatic approach in reducing the developer's burden and in being scalable in regards to the size of the code, the overlapping intervals it created are underwhelming. The results in Section 6.3.2 exhibit that, for most existing blocking communications, the created overlapping interval contained no other instruction than the initiation and completion calls, rendering it useless when compared to their blocking forms. The same effect is observed on existing nonblocking communications: the size of their overlapping interval did not increase with the application of the optimization pass.

In order to obtain larger overlapping intervals, we propose a method to provide information back to the developers, so that a human can intervene and overcome the limitations of the transformation pass detailed in Section 6.4.

One of the limitations that can be overcame by human intervention are the function calls. Our approach being intra-procedural, it is oblivious of the dependencies and effects a function call can have on a buffer. Having a human stepping in helps in disambiguating the dependencies and the behavior of the function. Given the necessary information, such as the dependencies, and with the knowledge on the semantics of the code, a developer should be able to rapidly determine if a function can safely be overlapped.

Looped communications and communications residing in conditional branches are also complex to handle in the intermediate representation. This would involve moving a communication and its dependencies out of a control structure, or require the cloning of the whole control structure to displace in order to gain overlapping potential. Because of this complexity, our method does not cover these situations. The developer intervenes directly on the source code, with the knowledge on which

branch or loop is limiting the overlapping interval, can result in a higher code quality with regards to its asynchronicity potential.

All these situations are complex and time consuming to automatically handle at compile-time. It would require extensive development time, notably due to the diversity of coding patterns that it would have to manage. Without prospect of improvements through an automatic transformation of codes, a tool-assisted approach is necessary. Retrieving feedback on the overlapping potential of a code can not only help developers of MPI applications in improving their code, but also help developers of static analyses and transformation tools in pinpointing communication and computation patterns where an optimization becomes worthwhile. Such approach would be more efficient in maximizing overlapping and asynchronous performance in an MPI program.

7.1.2 From automatic optimization to feedback generation

The approach we propose in this Chapter consists in building a feedback providing tool based on information that can be fetched by a compile-time analysis. The idea is that developers should be able to use this tool to learn more about their code potential and opportunities for improvement. Whereas the automatic transformation operates like a black box, this method involves the developers in the optimization method.

Such a tool can be built upon the automatic transformation pass we described in Chapter 6. Indeed, it holds all the information needed to provide feedback that will help developers improving their code. As explained in the previous Chapter, it is able to determine the dependencies of a communication. As a consequence, it is able to build a list of instructions that it would move in order to create and maximize the overlapping interval. This allows us to add the dependencies to the feedback message to inform the developers on the obstacles and the instructions they need to be aware of when manually modifying the code. Moreover, it is possible to base the feedback upon the boundaries identified by the automatic transformation pass, allowing the developers to intervene, decide the viability of further modifications, eventually leading to more overlapping potential.

All in all, this amounts to the feedback loop shown in Figure 7.1 which helps and incites developers to use nonblocking communications in their code. First, a source code file containing MPI communications, both blocking and nonblocking, is sent to the verification pass. This first step detects any misuse of the communication and their buffers. Once the code has been cleared of any potential mistakes, it is sent to the transformation pass which will create and extend over-

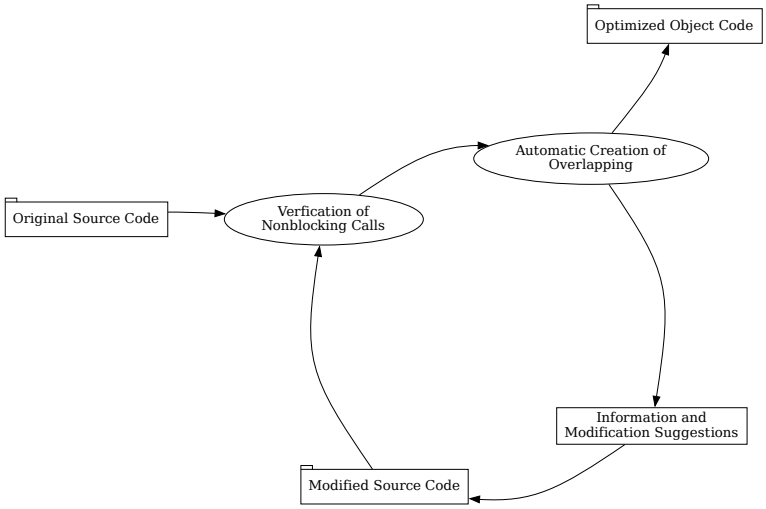


Figure 7.1 – Feedback loop to improve the overlapping potential of an MPI code

lapping possibilities. This results in an object file with all the transformations the automatic pass performed, and a list of suggestions. These suggestions should help the developers to manually modify their code to transform blocking communications into their nonblocking counterparts and achieve higher overlapping potential by overcoming boundaries that the automatic pass could not. It is then possible to validate these modifications by giving the modified source file to the verification pass once again.

Ultimately, the developers can either rely on the code created by the automatic optimization pass, or apply the suggestions given to them to manually improve the asynchronicity of their code. This method will incite them to use nonblocking communications thanks to the provision of feedback to correct and adapt MPI codes, which will also help in raising awareness on overlapping opportunities.

7.1.3 Emitting suggestions from limitations

Besides the analysis failures, several of the limiting factors identified by the automatic transformation method for the overlapping intervals can be leveraged to generate feedback and suggestions of code modification. As introduced in the previous Chapter, the transformation pass relies on an analysis of the CFG and of the dependency slices of the communication. Using this information, it transforms

and structures the IR to favor asynchronicity by displacing the nonblocking calls and their dependencies. The transformation of a communication stops when one of the limiting factors is found.

Before moving on to the next transformation, the transformation pass provides a justification why it decided to insert the nonblocking calls and the dependencies at a specific location in the IR. From each type of boundary found by the optimization method, along with the information on its location and cause, it is possible to infer suggestions of modifications and provide them to the developers. In this study, we focus on the following stopping reasons: other MPI communication calls, loop boundaries, branch boundaries, dependency in a branch or a loop, and dependency on a branching condition.

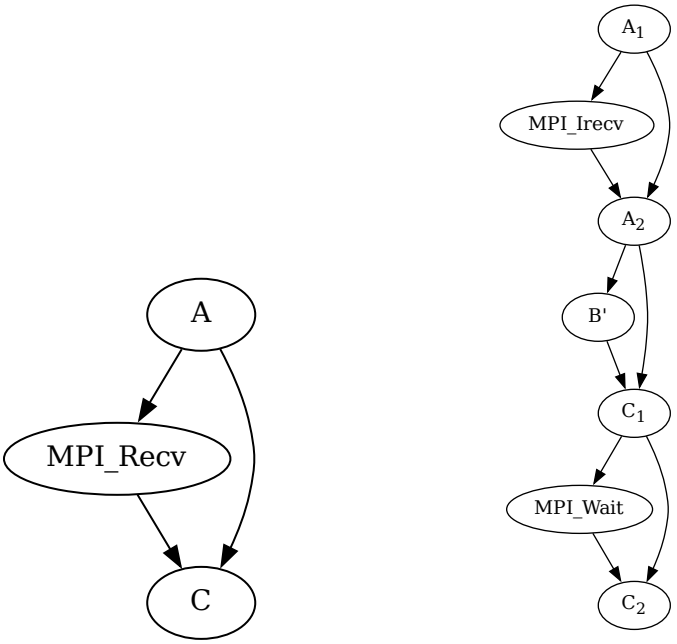
Other MPI communication calls

As explained in Section 6.1, there are several situations where the automatic transformation approach is unable to move nonblocking calls beyond other MPI communications. It lacks information on the worth of a communication, and defaults to a "first come, first serve" in the IR priority system. Having the developers intervene can help in improving the code, since they should have knowledge on which communication should get the priority for overlapping intervals.

Here, the feedback should contain the location of communication call preventing the analysis from further transforming the code. We would then suggest developers to decide if it is worth to manually overlap the said communication. Applied to Figure 6.1 for the second receive operation, the returned information should contain the location of the first initiation call. The suggestion given to the developers would be to look at the duration of the communications to determine which should overlap the other. If the second is more worthy, then the developer would have to manually transform it into its nonblocking form, and move the newly inserted `MPI_Irecv` and its dependency on the statement at line 2 above the first reception.

Branch or loop boundaries

Nonblocking calls whose overlapping interval is limited by the boundaries of branches are calls that are inside conditional branches. The automatic transformation pass cannot simply move the nonblocking calls out of the branch it is residing in without having to perform complex transformations on the CFG. Instead, a human eye can inspect the issue, determine the possibility of the transformation, its worth, and decide to apply the transformation directly on the source code.



(a) Original code

(b) Modified code with split branches

Figure 7.2 – Splitting and moving a conditional communication

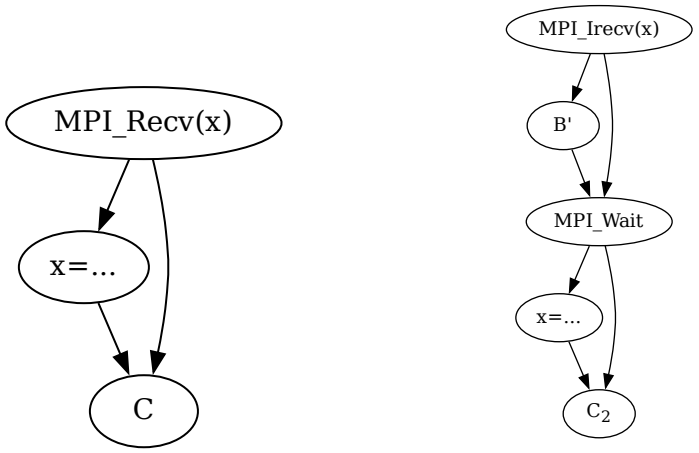
The information returned to the developers should contain the location of the incriminated conditional branch and the dependencies of the communication. From this feedback, we suggest developers to split the branching containing the MPI communication, as shown in Figure 7.2b. Manually performing this split is possible on the source code with the help of the information given by the feedback. This helps in increasing the overlap of the communication originally in node B. Vertices A_1 and C_2 would contain all dependent statements that were in vertices A and C respectively. A_2, B', and C_1 contain instructions independent of the communication, thus forming the overlapping interval. The nodes with the nonblocking calls contain dependencies that need to be executed under the condition. Obviously, the conditions leading to the execution of the initiation and completion calls must be the same as the condition leading to the execution of the `MPI_Recv` at the end of node A in Figure 7.2a. With the split form, it is possible to move the nonblocking calls and their dependencies while preserving the condition for executing the call.

Similarly to conditional branches, the automatic transformation pass does not move MPI communications in loops to the outer scope. The feedback for these communications must contain the location of the loop, its boundaries, and the dependencies of the communication, so that it helps developers in determining a course of action to increase the overlapping potential. The suggestion to resolve this issue is also similar. In this situation, the developer should fission the loop in multiple parts, one containing the independent statements that will end up in the overlapping interval, and the two remaining with the initiation, completion, and their dependencies. These two parts can then be moved, thus overlapping instructions that were originally in the outer scope, comparably to how the vertices with `MPI_Irecv` and `MPI_Wait` are moved in Figure 7.2b.

Loop or branch dependency

While our transformation method is able to recognize and move the dependencies along with the nonblocking calls, it cannot move those that are inside control flow structures, or loops and branches. This situation also requires extensive transformation of the CFG in order to move these dependencies without breaking the correctness of the program. For communications which are hindered by these dependencies, we also defined feedback and provide suggestions to overcome them. The feedback must contain the incriminated dependencies, as well as the location of the control flow structure containing these statements. The suggestion we give developers is to fission the loop or the branch in two parts.

Figure 7.3 illustrates the solution that is suggested to developers. Vertex B,



(a) Original code

(b) Modified code with a displaced dependency

Figure 7.3 – Splitting and moving a conditional dependency

which originally contained the statement that modifies the reception buffer, is split, with one part containing the instructions independent of the communication, noted B' in Figure 7.3b, while the other contains the dependencies, and thus the statement modifying x. The developer can then move the second part along with the completion call and the other dependencies from the outer scope. All subsequent modification or reading of x must also be displaced and the order of these operations preserved. The same kind of suggestion can be provided when x is in a loop.

There is one exception when the dependency is on the branching or looping condition. In other words, when the decision to go inside a conditional branch, or the number of iterations in a loop, is dependent on an MPI communication. In this situation, the fission of the control flow structure is impossible. Indeed, the whole structure, from the fork to the join nodes, is dependent on the communication. In these situations, it is necessary to move the whole loop or branch along with the nonblocking calls, and with all of their dependencies.

7.2 Determination of a metric to identify impactful transformations

In Chapter 6, we proposed a solution to automatically transform all existing blocking communications into their nonblocking forms in an MPI code, and to reorganize the code to favorise overlapping intervals. It was a very scalable approach since the automatic approach requires little involvement from the developers, and can be applied on large codes with minimal impact on the development times, since the transformations are performed at compile-time.

Contrary to that approach, what we propose in this Chapter requires human intervention. Yet, the size of HPC codes range from a few thousand to several hundred thousands lines of code, with dozens to thousand of static calls to MPI communication functions. For example, as illustrated in Table 6.2, the automatic transformation pass analyzed and modified more than three thousands MPI communications in Lammps. A developer cannot, in a reasonable time frame, analyze and transform all of the MPI communications in an HPC application. This is why it is necessary to help developers in picking a subset of communications on which they can focus on and analyze.

In this study, we reduce the number of MPI communications to address by sorting them by their communication time. For each static MPI call site, we determine the total time spent to perform the communication. The calls which have the biggest impact on the global execution time of the program will be considered.

This metric serves a dual purpose. First, it helps in determining if a communication is worth optimizing. The calls having a very low impact on the execution time of the application are not worth investigating. Should we succeed in fully overlapping the communication, the overall gain on the execution time of the application would still be very small, and barely noticeable. We would need to optimize the overlapping interval of several of such communications and fully overlap them in order to observe an effect on the execution time. On the opposite side, successfully overlapping one significant communication potentially leads to noticeable gains on the execution time. It is much more time efficient to analyze and optimize one communication which can take eight to ten percent of the execution time of an application, than analyzing and optimizing several communications with an impact of a few tenths of a percent each.

Secondly, when multiple MPI communications are competing for their overlapping interval, knowing the communication time, and thus the impact of each communication helps in prioritizing one over another. The more time consuming a communication is, the more overlapped it should be. For example in Figure 6.1, let

us suppose the broadcasting operation of buffer B has more impact on the performance of the code than the other broadcasting call. The reasons to this difference can be multiple (size of the expected message, or distance from the sender) and they cannot always be exploited at compile-time to determine the priority. In this situation, knowing that it is the more impacting communication, the developers can prioritize its overlapping interval, thus increasing the actual overlap.

7.3 Experimental results

In this Section, we first assess the ability of the filtering method described in Section 7.2 to reduce the number of MPI communications to consider and its impact on the optimization of the overlapping potential. In a second time, we evaluate the effects of the generation of feedback and their application on the five benchmarks considered for this study.

7.3.1 Determining the subset of communications to address

We first determine the subset of communications that are worth investigating in each benchmark by profiling them. To that end, we use mpiP [77] (version `3.4.1`), which is a profiling library for MPI programs. It can be linked to the executable of the targeted application, and instruments the behavior of MPI calls so they can output runtime statistics related to each communications call site per MPI processes. The returned data contain the number of times a call was executed, the mean (over its execution count) execution time for one call site, or the size of the message. The data also contain the ratio of execution time for a call site to the execution time of the whole program (App%), which is the metric we use to measure the impact of a communication.

We successfully instrumented the five benchmarks we considered for this study. They were run on a cluster composed of NUMA nodes with two AMD Epyc Rome processors, each one supporting 64 physical cores and 2 threads per physical core. Each core has access to 1875 megaoctets of memory. The NUMA nodes of this cluster are interconnected with BXI. Lammps, MCB, and Pennant have been run with 512 MPI processes, distributed over 4 NUMA nodes, each MPI process having access to one logical core. MiniMD and miniFE are run using half of these resources.

Table 7.1 displays the results of the profiling of MCB, Pennant, miniFE, and miniMD. Lammps is composed of multiple modules and several input data sets are available. Using different data sets leads to the execution of different sections of

Table 7.1 – Weight of MPI communication calls

	MCB	Pennant	miniFE	miniMD
Num. MPI comm. calls	37	18	30	56
Total App%	14.12	13.98	9.71	12.52
Num. MPI comm. calls ≥ 1 App%	5	2	2	6
Filtered Total App%	11.85	10.47	8.34	8.68

Table 7.2 – Weight of MPI communication calls (Lammps)

	①	SiH4	Si2H6	Li.bohr	Li.ang	②
Num. MPI comm. calls	108	111	111	89	94	97
Total App%	54.38	43.22	53.73	51.46	34.56	16.09
Num. comm. ≥ 1 App%	14	6	10	9	6	5
Filtered Total App%	40.99	29.81	40.55	42.33	28.99	8.18

① comb.Cu2O.elastic
② h2bulk.nve.ang

the benchmark, some being common while others being exclusive to a given input set. Thus, we decided on profiling Lammps using several data sets, whose name is in the first row of Table 7.2. MiniFE has been executed using a cube of dimensions 1024. MiniMD has been run with the default `in.eam.miniMD` input file. We used the `leblancx4.pnt` input file for Pennant, and MCB has been executed with 20000 Monte-Carlo particles per core.

First, we measure the weight of all communications for each benchmarks. This is done by averaging the App% of each call site for all MPI processes that execute it. This mean is then summed, thus resulting in the "Total App%" value. We decide on a threshold of 1% for the App% which will determine the call sites that are worth optimizing. The "Filtered Total App%" corresponds to the summed average App% of those calls.

7.3.2 Application of the feedback-based approach on benchmarks

The approach we propose in Section 7.1 has been implemented by adapting the automatic transformation pass we designed in Section 6.3. Operating as an

LLVM pass, it analyzes the IR of a function. The feedback providing part of the pass intervenes on each detected overlapping interval, including those that have been created by the transformation of blocking calls into their nonblocking form. For each interval, it analyzes the reason why the initiation and completion calls cannot be moved further. It matches this reason against a set of predefined suggestion messages according to the possibilities defined in Section 7.1.3. Each of those messages are then augmented with the location of the obstacle. However, the implementation we propose is not yet able to provide the list of dependencies. Working at the level of the IR, the dependent statements cannot be easily mapped back to their source code equivalent.

We applied this method on the five benchmarks that we successfully profiled with mpiP: miniMD, miniFE, Lammps, Pennant, and MCB. While we did not apply every suggestion returned by our implementation of the method, we looked for a diversity of feedback kinds to illustrate how each suggestion can be applied. The correctness of the code after manually transforming the code with the help of the suggestions is checked in two steps. First we run the verification pass defined in Chapter 5 in order to check if the newly inserted nonblocking calls can be matched together. Secondly, the modified benchmarks are compiled and run, and we compare the output against a base version to check if the numerical results of the tested problem are still valid and within the error margins. Since this study focuses on the creation of static overlapping, the run-time performance of the benchmarks was not measured, as detailed in Section 6.3.1.

MiniMD

The results from the automatic transformation pass applied on miniMD gave us an example of a communication whose overlapping interval is hindered by the limits of the branch it is residing in. Figure 7.4 is an extract from `comm.cpp` at line 311, the `MPI_Irecv` cannot be moved further up because it is in a branch. It has been successfully matched with the completion call at line 317, and it measured an interval of 17 IR instructions after applying the automatic transformation. The reception has a total impact of 1.26% on the execution time of the benchmark (1.01 from the completion call, and 0.25 from the initiation). We noticed in this code that instructions from line 289 to 295 are all independent from the reception, thus can be overlapped. A human analysis of the call to the function at line 295 shows it does not have any side effect on the reception buffer.

The suggestion given back by our method when a call is being blocked by the boundaries of a branch is to fission the control structure. Such transformation is possible here, and can lead to substantial and immediate gains. Indeed, having

Figure 7.4 – MiniMD, `comm.cpp` original code, boundaries of a conditional branch

```
289    pbc_flags[0] = pbc_any[iswap];
290    pbc_flags[1] = pbc_flagx[iswap];
291    pbc_flags[2] = pbc_flagy[iswap];
292    pbc_flags[3] = pbc_flagz[iswap];
293
294    //#pragma omp barrier
295    atom.pack_comm(sendnum[iswap], sendlist[iswap], buf_send, pbc_flags);
296
297    //#pragma omp barrier
298
299    /* exchange with another proc
300       if self, set recv buffer to send buffer */
301
302    if(sendproc[iswap] != me) {
303      #pragma omp master
304      {
305        if(sizeof(MMD_float) == 4) {
306          MPI_Irecv(buf_recv, comm_recv_size[iswap], MPI_FLOAT,
307          recvproc[iswap], 0, MPI_COMM_WORLD, &request);
308          MPI_Send(buf_send, comm_send_size[iswap], MPI_FLOAT,
309          sendproc[iswap], 0, MPI_COMM_WORLD);
310        } else {
311          MPI_Irecv(buf_recv, comm_recv_size[iswap], MPI_DOUBLE,
312          recvproc[iswap], 0, MPI_COMM_WORLD, &request);
313          MPI_Send(buf_send, comm_send_size[iswap], MPI_DOUBLE,
314          sendproc[iswap], 0, MPI_COMM_WORLD);
315        }
316
317        MPI_Wait(&request, &status);
318      }
319      buf = buf_recv;
320    } else buf = buf_send;
```

Table 7.3 – Overlapping interval length of `MPI_Irecv` from `comm.cpp` at line 311, miniMD

	Original code	After modification
Interval length	17	67

Figure 7.5 – MiniMD, `comm.cpp` modified code

```
286       if (sendproc[iswap] != me) {
287         #pragma omp master
288         {
289             if (sizeof(MMD_float) == 4) {
290                 MPI_Irecv(buf_recv, comm_recv_size[iswap], MPI_FLOAT,
291                         recvproc[iswap], 0, MPI_COMM_WORLD, &request);
292             } else {
293                 MPI_Irecv(buf_recv, comm_recv_size[iswap], MPI_DOUBLE,
294                         recvproc[iswap], 0, MPI_COMM_WORLD, &request);
295             }
296         }
297       }
298
299     /* pack buffer */
300
301     pbc_flags[0] = pbc_any[iswap];
302     pbc_flags[1] = pbc_flagx[iswap];
303     pbc_flags[2] = pbc_flagy[iswap];
304     pbc_flags[3] = pbc_flagz[iswap];
305
306     //#pragma omp barrier
307     atom.pack_comm(sendnum[iswap], sendlist[iswap], buf_send, pbc_flags);
308
309     //#pragma omp barrier
310
311     /* exchange with another proc
312        if self, set recv buffer to send buffer */
313
314     if(sendproc[iswap] != me) {
315       #pragma omp master
316       {
317         if(sizeof(MMD_float) == 4) {
318           MPI_Send(buf_send, comm_send_size[iswap], MPI_FLOAT,
319           sendproc[iswap], 0, MPI_COMM_WORLD);
320         } else {
321           MPI_Send(buf_send, comm_send_size[iswap], MPI_DOUBLE,
322           sendproc[iswap], 0, MPI_COMM_WORLD);
323         }
324
325         MPI_Wait(&request, &status);
326       }
327       buf = buf_recv;
328     } else buf = buf_send;
```

Table 7.4 – Overlapping interval length of `MPI_Reduce` from `main.cpp` at line 169, miniFE

	Original code	After modification
Interval length	3	9

noticed that the lines 289 to 295 are independent from the communication, the idea would be to split the reception from the sending parts, and have the reception parts overlap the identified portion of code. Figure 7.5 shows the result of such modification. The whole conditional structure containing the nonblocking reception call is split, and moved above the lines relating to the creation of the sending buffer, putting them inside the overlapping interval. Its length is of 67 IR instructions after applying this transformation. Note that the OpenMP pragma causes issues for the matching, since it introduces an additional branching condition to check if the current thread is the master thread. However, the comparison involves a global structure related to OpenMP, preventing the our passes from matching back the nonblocking calls. As a consequence, we had to comment out the pragma to measure the length of the overlapping interval.

MiniFE

Figure 7.6a shows a snippet from miniFE, `main.cpp`, where our method reported that a communication was being hindered by a dependency in a control flow structure. Both `MPI_Reduce` at lines 167 and 169 are affected. Both read from the same buffer, and writes to buffers that are going to be used in the branch starting at line 171. The second reduction operation is also being hindered by the first one, since the automatic transformation pass lacks awareness about the order of communications.

We focus on the second reduction. In this example, the automatic method returns two suggestions, one for the completion call, and the other for the initiation call. The first is about splitting the control flow structures to separate the dependent and independent parts. The second, on the initiation call, is to look at the impact of both calls to determine which should get the priority. Here, both `MPI_Reduce` have a very low impact on the execution time of miniFE, with an App% value that is less than 0.01. Yet, swapping the communications here can lead to a greater overlapping opportunity for the reduction on `global_rss`. Furthermore, there are no risk of mismatching the order of collectives in this examples, since there is only one possible execution flow.

(a) Original code, dependency in a branch

```
167   MPI_Reduce(&rank_rss, &global_rss, 1,
168        MPI_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
169   MPI_Reduce(&rank_rss, &max_rss, 1,
170        MPI_LONG_LONG, MPI_MAX, 0, MPI_COMM_WORLD);
171   if (myproc == 0) {
172        doc.add("Global␣All-RSS␣(kB)", global_rss);
173        doc.add("Global␣Max-RSS␣(kB)", max_rss);
174   }
```

(b) Modified code

```
167   MPI_Request req;
168   MPI_Ireduce(&rank_rss, &max_rss, 1,
169        MPI_LONG_LONG, MPI_MAX, 0, MPI_COMM_WORLD, &req);
170   MPI_Reduce(&rank_rss, &global_rss, 1,
171        MPI_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
172   if (myproc == 0) {
173        doc.add("Global␣All-RSS␣(kB)", global_rss);
174   }
175   MPI_Wait(&req, MPI_STATUS_IGNORE);
176   if (myproc == 0) {
177        doc.add("Global␣Max-RSS␣(kB)", max_rss);
178   }
```

Figure 7.6 – MiniFE, `main.cpp`

Table 7.5 – Overlapping interval length of `MPI_Send` from `Mesh.cc` at line 635, Pennant

	Original code	After modification
Interval length	1	3

Figure 7.6b exhibits the modifications that can be applied thanks to the feedback. The communications are swapped, and the conditional branch is split, and the completion call inserted before the dependency. The resulting length of the overlapping interval is 9 IR instructions.

Pennant

Figure 7.7a contains an example of a communication whose overlapping interval is obstructed by the boundaries of a loop. The complete body of the function containing this excerpt of code can be found in Figure 5.1a. The automatic transformation pass successfully determined that the `MPI_Send` at line 635 actually suffers from being trapped inside the loop starting at line 631. Indeed, once transformed into its nonblocking version, both its initiation and completion calls cannot be moved out of the current iteration by the automatic transformation pass, to preserve the execution count of the communication. This communication has an impact of 1.19% on the execution time of the benchmark.

Following the method we defined in Section 7.1.3, the suggestion that is returned to the developers is to fission the loop containing the communication to separate dependencies and the initiation from the completion. In this example, there are no dependencies across the iterations of the loop, and there are no write operations that could corrupt the outbound array `slvvar`. Consequently, it is possible to have each iteration of the communication overlap the others by separating the completion calls into another loop with the same properties. The resulting code is shown in Figure 7.7b. The completion loop is replaced by a call to `MPI_Waitall`. This allows an increase in the length of the overlapping interval, as seen in Table 7.5.

The computation of the length does not account for the number of iterations, and only considers the shortest path between the initiation and the completion, resulting in a low number of IR instructions. However, multiple iterations are actually being overlapped. Also note that the matching of the newly inserted nonblocking send and the Waitall calls suffers from the same limitations as those mentioned in Section 5.5 because of global variables. The measurement of the

(a) Original code, boundaries of a loop

```
631     for (int mstrpe = 0; mstrpe < nummstrpe; ++mstrpe) {
632         int pe = mapmstrpepe[mstrpe];
633         int nslv = mstrpenumslv[mstrpe];
634         int slv1 = mapmstrpeslv1[mstrpe];
635         MPI_Send(&slvvar[slv1], nslv * type_size, MPI_BYTE,
636                pe, tagmpi, MPI_COMM_WORLD);
637     }
```

(b) Modified code

```
631     MPI_Request req[nummstrpe];
632     for (int mstrpe = 0; mstrpe < nummstrpe; ++mstrpe) {
633         int pe = mapmstrpepe[mstrpe];
634         int nslv = mstrpenumslv[mstrpe];
635         int slv1 = mapmstrpeslv1[mstrpe];
636          MPI_Isend(&slvvar[slv1], nslv * type_size, MPI_BYTE,
637                 pe, tagmpi, MPI_COMM_WORLD, &req[mstrpe]);
638     }
639     MPI_Waitall(nummstrpe, req, MPI_STATUSES_IGNORE);
```

Figure 7.7 – Pennant, `Mesh.cc`

length of the interval was done by copying `nummstrpe` into a local temporary variable, and by replacing all its uses by this local variable.

MCB

The `MPI_Allreduce` at line 140 in `sumOverDomains.cc` from MCB is another example where our automatic transformation pass is limited by the boundaries of a conditional branch. The code excerpt containing this communication is shown in Figure 7.8. It represents 2.01% of the execution time of the benchmark. The Allreduce operation is only executed if the condition at line 137 is verified. We have noticed the instructions following this branch are independent from the communication, and can be safely overlapped. The function calls at lines 148, 156, and 158 do not have any side effect, neither on the outbound buffer `inputValue` or the inbound `summedValue`.

Using the suggestions returned by the approach we defined in this Chapter, the developers can amend their code to favorise overlapping. Figure 7.9 shows the results of such modifications. First the conditional branch is split, with the section containing the initiation call staying in place. The section containing the completion can be moved to overlap the independent instructions at lines 148 and 152 in the orignal code. We notice that the condition at line 156 is equivalent to the one guarding the communication. Therefore, the completion call can simply be moved inside the "`true`" branch and overlap line 158, resulting in the code shown in Figure 7.9. The `ASSERT` function call cannot be overlapped, since it contains an `exit` call, prematurely ending the execution. Improperly quitting the MPI runtime with pending communications might result in memory leaks.

While we could measure the interval length before applying the modifications (1 IR instruction), we could not match the inserted nonblocking calls together. Therefore, we are unable to give a size for the overlapping interval post modification.

Lammps

We will focus on one example in Lammps, at line 369 in `compute_reduce.cpp`, with a loop of `MPI_Allreduce` Figure 7.10a contains the relevant snippet of code. The weight of this communication on the execution time of the benchmarks varies depending on the input data, as shown in Table 7.6. The communication represents at most 1.49% of the execution time, and it goes as low as 0.14 when using the `h2bulk.nve.ang` data set. The automatic transformation pass correctly determined that this communication had its overlapping interval hindered by the

Figure 7.8 – MCB, `sumOverDomains.cc` original code, dependency in a branch

```
137         if( Mesh.isDomainMaster() )
138         {
139             const MPI_Comm &domainMastersComm = Mesh.
                    getDomainMastersCommunicator();
140             MPI_Allreduce( &inputValue, &summedValue,
141                            1,
142                            mpiType.get_MPI_type(),
143                            mpiType.get_MPI_SUM(),
144                            domainMastersComm );
145 
146         }
147 
148         const MPI_Comm &domainGroupComm = Mesh.getDomainGroupCommunicator();
149 
150         // get rank of master proc of this domain
151         // 0 is always the local ID of the master in the domain group
152         int domainGroupMasterRank = 0;
153 
154         // This code just checks convention that group master rank in domain
155         // group comm is 0
156         if( Mesh.isDomainMaster() )
157         {
158             MPI_Comm_rank( domainGroupComm, &domainGroupMasterRank );
159             ASSERT( domainGroupMasterRank == 0 );
160         }
161         else
162         {
163             int domainGroupRank;
164             MPI_Comm_rank( domainGroupComm, &domainGroupRank );
165             ASSERT( domainGroupRank != 0 );
166         }
```

Figure 7.9 – MCB, `sumOverDomains.cc` modified code

```
137         MPI_Request req;
138         if( Mesh.isDomainMaster() )
139         {
140             const MPI_Comm &domainMastersComm = Mesh.
                    getDomainMastersCommunicator();
141             MPI_Iallreduce( &inputValue, &summedValue,
142                             1,
143                             mpiType.get_MPI_type(),
144                             mpiType.get_MPI_SUM(),
145                             domainMastersComm, &req );
146 
147         }
148 
149         const MPI_Comm &domainGroupComm = Mesh.getDomainGroupCommunicator();
150 
151         // get rank of master proc of this domain
152         // 0 is always the local ID of the master in the domain group
153         int domainGroupMasterRank = 0;
154 
155         // This code just checks convention that group master rank in domain
156         // group comm is 0
157         if( Mesh.isDomainMaster() )
158         {
159             MPI_Comm_rank( domainGroupComm, &domainGroupMasterRank );
160             MPI_Wait(&req, MPI_STATUS_IGNORE);
161             ASSERT( domainGroupMasterRank == 0 );
162         }
163         else
164         {
165             int domainGroupRank;
166             MPI_Comm_rank( domainGroupComm, &domainGroupRank );
167             ASSERT( domainGroupRank != 0 );
168         }
```

Table 7.6 – Weight of `MPI_Allreduce`, `compute_reduce.cpp`, line 369, Lammps

	①	SiH4	Si2H6	Li.bohr	Li.ang	②
App%	1.49	1.29	1.39	0.62	0.19	0.14

① comb.Cu2O.elastic
② h2bulk.nve.ang

Table 7.7 – Overlapping interval length of `MPI_Allreduce` from `compute_reduce.cpp` at line 369, Lammps

	Original code	After modification
Interval length	1	3

boundaries of the loop it is residing in. The loop only contains the communication, and there are no dependencies between iterations.

The suggestion to separate the dependencies and the initiation from the completion also applies in this example. The result of the suggested manual transformation is shown in Figure 7.10b. Separating the completions from the initiations results in a loop of `MPI_Wait` calls with no dependencies. Such structure is equivalent to an `MPI_Waitall`, thus the code in Figure 7.10b. The subsequent interval length, shown in Table 7.7 shows a little increase in overlapping potential, but it does not reflect the overlapping of multiple communications at a time.

7.4 Discussion

In this Chapter, we leverage the limitations of the automatic transformation pass we defined in Chapter 6 to build a tool-assisted optimization method. Based on the boundaries of the overlapping intervals, it provides insights to the developers to guide them in improving their code for asynchronicity: location of the boundaries, data dependencies, and a suggestion on the required amendments. Since we strayed away from a fully automatic approach, the feedback needs to be analyzed by the developer, and the transformation manually applied accordingly. This is not a suitable approach for very large codes. In order to reduce the workload, we propose in Section 7.2 a method to filter the communications to optimize. It is based on a profiling of the program to determine the weight of each communication on the total execution time of the simulation.

(a) Original code, boundaries of a loop

```
368  if (mode == SUM || mode == SUMSQ) {
369    for (int m = 0; m < nvalues; m++)
370      MPI_Allreduce(&onevec[m],&vector[m],1,MPI_DOUBLE,MPI_SUM,world);
371
372  } else if (mode == MINN) {
```

(b) Modified code

```
368  if (mode == SUM || mode == SUMSQ) {
369    MPI_Request req[nvalues];
370    for (int m = 0; m < nvalues; m++)
371      MPI_Iallreduce(&onevec[m],&vector[m],1,MPI_DOUBLE,MPI_SUM,world, &req[m]);
372    MPI_Waitall(nvalues, req, MPI_STATUSES_IGNORE);
373
374  } else if (mode == MINN) {
```

Figure 7.10 – Lammps, `compute_reduce.cpp`

7.4.1 Reducing the number of calls to analyze

By profiling the benchmarks, we were able to determine the impact of each communication call on the overall execution time of the program. Table 7.1 shows that, at least for the four benchmarks upon which we performed this measurement, only a few MPI communications has a significant impact. As an example in MCB, if we only keep the communications whose impact is greater than one percent of the execution time, only five would remain. Those five MPI calls have a total impact of more than eleven percent, as compared with the fourteen percent when considering every call. This would mean that the thrity-two remaining ”only” have a total impact of three percent. It also means that the developer only needs to analyze and transform those five calls to have a significant reduction in the execution time of the application, especially if those communications can be fully overlapped. The same result can be observed on the other benchmarks we analyzed. The amount of weight filtered by the threshold of one percent varies, and can reach half of the total weight of communication calls when using the `h2bulk.nve.ang` data set for Lammps, as shown in Table 7.2. However, the actual value is still significant enough, with more than eight percent for five calls, while the other half is distributed over ninety-two calls. The threshold value can then be adjusted to have more or less calls to analyze.

Nonetheless, this approach requires a preliminary execution of the targeted

program, which defeats the advantages of having a compile-time method. As illustrated with Lammps, the number of calls and the amount of preserved weight highly depends on the picked entry data set. Furthermore, Table 7.6 shows the impact of one specific static call site also depends on the entry data, which complexifies the compile-time determination of the communications to prioritize. Preliminary studies to achieve such feat should focus on identifying the factors contributing to the impact of a communication. These factors might include the number of calls, the size of the message, the number of communicants, and their relative distance. Knowing these factors, it should be possible to determine an order of magnitude for the weight of each MPI call, and thus making it possible to estimate their weight at compile-time.

7.4.2 Pertinence of a tool-aided optimization for MPI calls

The proof of concept we described in Section 7.1.3 to provide suggestions of code modifications to improve the overlapping potential of MPI programs has been successfully applied on several benchmarks. We observed a threefold increase in the length of the overlapping interval compared to what the automatic transformation created, most notably in miniMD (see Table 7.3). On other occasions, it was able to suggest us to overlap loop iterations, even though the result was not reflected in the measurements of the intervals. This is due to the computation of this metric: it is based on the shortest path between the initiation and the completion. Indeed, at compile-time, the number of iterations is unknown, and there could be only one iteration before exiting the loop and reaching the completion call. These improvements are possible thanks to human intervention in the optimization pass to lift several limitations. For example the ordering of MPI calls to prioritize communications with more impact, or ensuring that more in-depth transformations were possible and to perform them, such as the fission of loops and of conditional branches.

Despite the positive results compared to those we found with the automatic approach, the way we measure these results face the same limitations as those described in Section 6.4, and more specifically on the weight of each instruction in the interval. As a consequence, a threefold increase of the overlapping potential in a section of the code does not necessarily leads to an equivalent reduction in the execution time of that section. Moreover, even with the help of a static solution to fetch the weight of each communication, the application of a suggestion ultimately requires human intervention, even though the workload is reduced.

Beyond these limitations, providing feedback to improve the asynchronicity of

a code is profitable for both the developers of MPI programs and developers of optimization solutions. For developers of HPC applications, it is about obtaining useful tips to write and adapt their code, especially during the early prototyping phases of the development cycle. On the other side, for developers of optimization solutions, it is about fetching statistics and information on communication and coding patterns. In the end, this should help them in a better allocation of development time for communication and coding patterns that are impacting and where overlapping can be achieved, such splitting control flow structures to separate dependencies. Finally, as the verification pass can be adapted to handle persistent communications, it is possible to analyze the code to detect situations where using persistent calls should be profitable, and suggest these modifications to developers.

Part III

Conclusion

Conclusion

In their quest for an exascale supercomputer, engineers have been resorting to massive parallelism by ever increasing the number of computing units within a system. As a consequence, handling the communications between these units is more than ever of the utmost importance. Yet, these communications can be source of performance degradation, especially in distributed memory architectures where they need to go through an interconnect. They can prevent the computing nodes from performing at their full potential because of communication delays, for example: latency and necessary synchronizations.

Nonblocking MPI calls is one of the solutions to this problem, by enabling overlapping of communication times by computations. In practice, it consists in defining an overlapping interval for each nonblocking communication, and in inserting instructions in this interval to hide the communication times. Ideally, these instructions should be concurrently executed to the progression of the data transfer. However, the nonblocking form is more complex to use than the "basic" blocking form. As they do not provide any safety measure against race conditions and are more prone to cause deadlocks, they require more attention from the developers. As a consequence, despite being introduced for more than a decade, the nonblocking collective calls are still not as popular as their blocking version.

We have developed two approaches to encourage developers to use nonblocking calls to fully exploit the computing potential of their hardware. The first is to identify the programming errors they could add when using nonblocking operations. The second is to assist in the creation and the maximization of the overlapping potential of such communications. The existing solutions to tackle these approaches are either not conveniently usable, not very well suited for the HPC environment, or incomplete in their coverage.

In this context, we proposed a new compile-time verification method to detect mismatching nonblocking calls and race conditions, an optimization pass that reorganizes the dependencies in a code to create and extend the overlapping potential of an MPI program, and a feedback-based method to maximize the overlapping

gains. The analyzing of an HPC code at compile-time is preferable despite the lower accuracy compared to dynamic or model checking approaches. Analyzing a code during its compilation results in a better integration into the development cycle of a program. Furthermore, it is much more scalable than model checking and symbolic execution.

Static verification of nonblocking communications

Based on an analysis of the control and data flow, we propose a method to determine at compile-time if a nonblocking communication can cause a deadlock or a race condition which eventually results in incorrect results.

For each completion call, it first identifies the completion calls using the same request object. In a second step, the sets of completion calls that are capable of catching the request are determined. To that end, our method resorts to an analysis of the control flow graph using the notion of post-dominance. It is coupled with an analysis of the data flow to determine the execution conditions of the nonblocking calls, and to retrieve the property of the loops they belong to. The matching completion calls of an initiation call are then determined by identifying the most immediate capable set of completion calls.

This matching allows the definition of an overlapping interval for each valid nonblocking communication. The identification of such an interval enables their analysis, and most notably, the detection of an eventual race condition. This detection is performed by visiting each instruction of the interval with an alias analysis and the dependency slice of the communication.

Implemented as an LLVM pass, this method is tested on several large scale benchmarks. While the matching pass was mostly successful, the detection of data races raised many false positives.

Automatic exposition of overlapping potential

The second major contribution of this work is a method to automatically create and extend the overlapping potential of an MPI code. It transforms the intermediate representation of a function to reorganize the order of statements in favor of larger overlapping intervals, while preserving the original behavior of the program.

First, we must identify all that might be a boundary of an overlapping interval. The identified reasons for limiting an interval are: boundaries of the caller function, some other MPI calls, other call sites, dependencies in a control flow structure, and the limits of the encompassing control flow structure. Contrary to existing works, our method do not consider data dependencies as boundaries.

All existing blocking communications are then transformed into their nonblocking version, with the insertion of a completion call, the allocation of a request, and the replacement of the blocking primitive by its nonblocking form. Existing nonblocking calls are matched using the methods developed for the verification pass. The optimization method consists in moving the completion and the initiation calls as far as possible from each other in the intermediate representation. To preserve the order of dependencies, which are not boundaries for the overlapping interval, they must be moved along with the nonblocking calls. As a consequence, only the independent statements remain untouched, and end up inside the overlapping interval and increasing its length.

The transformation pass is evaluated by measuring the static size of the overlapping intervals. It was able to create overlap in some programs that did not expose any. All benchmarks witnessed an increase of their overlapping potential. Despite the creation of some windows whose size is similar to that of the existing intervals in the code, the pass could not expose any interval in the majority of cases. The main reasons preventing it from further expanding an interval are related to the presence of another MPI communication call, to the presence of a dependency inside a conditional branch, or because a communication cannot be moved out of a branch.

Maximizing the overlapping potential through compiler assisted transformations

The third contribution of this work addresses the shortcomings of the transformation pass. It consists in providing feedback to the developers of MPI programs to help them in modifying their code to favorise the overlapping of communications.

This feedback should inform the developers on the limiting factors for the creation or the expansion of overlapping potential. Based on this, it suggests a transformation to apply to circumvent the obstacle, thus enabling longer overlapping intervals. The feedback exploits the statistics gathered by the transformation pass. It relies on the boundaries of the overlapping intervals to provide suggestions of modifications to the developers, along with information on the current limiting factor of the overlapping interval. With the help of these elements, they should be able to modify their code, and have their new transformations validated by the verification pass.

This method being manual and relying on an intervention from the developer, it is necessary to reduce the number of MPI calls to analyze and optimize. Large codes such as Lammps or MCB contain hundreds or thousands of MPI operations.

The reduction of the number of MPI calls is performed with the help of a profiling tool. It determines the time spent in each communication, and the calls with the most impact are prioritized.

By only considering the operations whose communication time exceeds 1% of the execution time of the application, we cut down the number of communications to examine in each benchmark to a dozen while preserving most of the impact of the communications. The successful overlapping of those communications should result in an observable impact on the performance of application. With the help of the suggestions, we applied several transformations which resulted in an increase of the static length of the intervals, compared to the automatically created intervals.

Perspectives

Our work and contributions are nonetheless challenged by many limitations. Addressing them provides interesting opportunities for future engineering and research projects. We identified three major axes of improvement which we develop in this Section. The first is on the direct improvements for our contributions, which should help in analyzing nonblocking codes. The second axis is related to the possible extensions to our contributions to support other MPI operations or programming interfaces. Finally, we conclude with perspectives on the application of our methods, and on the analysis of HPC codes at compile-time.

Improving the detection of race conditions

As noted in Section 5.5, the implementation of our verification pass struggles with the detection of race conditions. It has a poor accuracy with a high rate of false positive reports. In other words, it flags too many memory accesses as a potential race condition. This is the most immediate limitation to address, as it severely limits the verification pass. The causes for this flaw are most likely the alias analysis incorrectly or conservatively determines a dependency between a statement and a communication buffer, the lack of an interprocedural alias analysis to determine the impact of a call site, or an overlooked coding error in the pass itself. The addition of a more capable alias analysis to the middle-end pass pipeline is one of the planned solutions to address this issue. SVF[78] is an example of an interprocedural alias analysis with the ability to analyze the whole program at link-time. However, it is limited to the intermediate representation generated from C codes, barring it from analyzing most large scale benchmarks.

Interprocedural analysis

The lack of an interprocedural analysis is one of the major limitations of our methods. As mentioned above, it can be one of the causes for the low accuracy of the detection of data races. It is also limiting the effectiveness of the matching and the optimization methods, notably when global variables are involved. Without an interprocedural analysis of the dependencies, LLVM assumes, rightly so to provide a conservative result, that any function call can have a side effect, modifying the global variable. Moreover, in Section 6.3.2, we have noticed 200 nonblocking operations could not be moved further because it reached the end or the beginning of the caller function. As a consequence, adding an interprocedural analysis to both our verification and optimization passes should help in increasing their performance.

In a complex code, MPI operations might be spread over multiple code modules or classes that are not necessarily defined in the same translation unit. The interprocedural analysis should have the vision on the entire program, allowing the analysis of functions from other translation units. Ideally, the verification and transformation passes have to be performed at link time.

Verification and optimization of persistent and partitioned communications

With their introduction in the fourth major revision of the MPI specifications, persistent collective and partitioned point-to-point communications are expected to gain in popularity. In Section 5.4, we introduced an extension to our verification pass to detect mismatching persistent communications. Future efforts can encompass partitioned communications, and the actual implementation of these techniques as a compilation pass.

Extension to other programming interfaces

During our work, we focused on the verification and optimization of MPI nonblocking calls at compile-time. We did so by first matching the calls together through an analysis of the control flow and of the data flow. In a second time, we identified the dependencies of the communication, and reorganized the code to favorise longer overlapping intervals. The semantics of MPI nonblocking calls are not unique, there are other programming interfaces with a similar form.

The mechanism of futures and promises allows concurrent execution in some programming languages. A future can be defined as the *future* value of some

variable, while a promise provides the value of the future. This mechanism has been implemented in the C++11 standard. Figure 7.11 is an example of how the futures and promises can be used. The promise is first created, and a future is attached to it. The instruction at line 9 calculates the actual value of the future. Ideally, it should be executed on another thread or computing unit to fully benefit from the concurrency. Finally, the `get` method waits for the computation of the future, and stores it. In this example, everything is serialized.

Figure 7.11 – Example of a future and a promise in C++

```
1  #include <future>
2  #include <iostream>
3
4  int main()
5  {
6      std::promise<int> promise;
7      std::future<int> future = promise.get_future();
8
9      promise.set_value(10);
10
11     int result = future.get();
12     std::cout << result << "\n"; // 10
13
14     return 0;
15 }
```

The verification and optimization methods we proposed in this work can be adapted to this kind of semantic. In the code presented in Figure 7.11, the matching would no longer be performed on a request, but using the addresses of the `future` and `promise` objects. The detection of race condition can then be applied to determine misuses of the future. The optimization method would fetch the dependencies required to evaluate the promise, and try to put as many independent computations as possible between the creation of the future and its first use.

CUDA exposes a similar semantic with its asynchronous operations and streams. For example, we could check that the memory locations being used in a `cudaMemcpyAsync` is not facing a race condition before a "completion" call on the stream, such as `cudaStreamSynchronize`.

In addition, our work can also be extended to communication libraries acting as a wrapper above MPI. It is possible that a program relies such pre-compiled libraries to call MPI functions. In this situation, the link time analysis of the program we suggested earlier would not suffice. Provided that the semantics of

the functions defined in the library resemble those of the MPI nonblocking calls, and that these functions are sane (which can be asserted by applying the verification pass on the library itself), the verification and transformation passes can be extended to support the library.

Integrating the verification and optimization passes to ongoing development cycles

We evaluated our verification and optimization passes on existing benchmarks that have been in development and fine-tuned for years to achieve performance. While we did identify several errors and opportunities to increase the overlapping potential, all lead us to believe that these codes have already been proof-read for the most part, and fairly well optimized.

The methods we propose are not only aimed at analyzing and transforming existing codes, and most notably at creating overlapping potential in a quick fashion, they are also meant to guide developers during the whole lifetime of their program, from its prototyping phase to its deployment on production clusters. Consequently, it would be interesting to apply our methods on codes that are still in the early phases of their development cycle. This would allow us to assess our methods on environments that are less controlled: errors are more likely to be committed, and codes are not yet fully optimized.

Furthermore, since our methods are implemented as LLVM passes, they are easily deployable and usable, even by novice users. Students in high performance computing and aspiring developers can benefit from our contributions. In an academic context for example, the verification pass and the feedback-based optimization passes can help students to better understand their errors and how they can improve their code.

Communication performance prediction at compile-time

The feedback-based method we propose in this work has to rely on a preliminary execution of the code. It allows the fetching of performance data, especially on communication times, to determine the impact of a communication. The more impact, the more the communication should be prioritized for the optimization of its overlapping interval, for example, over another MPI communication as long as their dependencies allows the overlapping. Establishing this priority at compile-time is complex, and depends on factors unknown to compilers. The size of a message or the number of MPI processes involved in a collective operation are most likely one of the influential parameters. They might be estimated through a

thorough analysis of the dependencies, at least symbolically thus resulting in an order of magnitude for each parameter. However, MPI communications usually involve the cluster interconnect. Parameters about the network, and above all the configuration of the cluster and of the runtime are unknown to the compiler. Examples of these parameters are the number of MPI processes, the bandwidth, or the distance between two communicating nodes.

The interconnect plays a major role in the performance of an HPC application. As backends for each existing CPU architecture exist, providing specific information about the interconnect or the cluster in use to the compiler might help it in optimizing HPC codes. The information such "backend" might provide could include some of the elements we enumerated above to help the compiler in estimating the communication performances.

Appendix A

Benchmarks

The methods we propose and their implementation are tested and evaluated on five large scale HPC benchmarks. It includes two mini-apps, and three CORAL benchmarks. Mini-apps[75] are lighter programs with few library dependencies. Their objective is to allow developers to rapidly test various hardware and parameters configurations on small and fast codes, that still provide an accurate depiction of, for example, a numerical model that would be used in production codes. On the other hand, CORAL benchmarks[79][80] are larger in scale, and closely approximate production scale simulation codes. The codes originate from a collaboration of multiple American HPC research faculties, and the patterns of coding, of communication and of computations, are designed to be as close as possible to production programs.

MiniFE[81] is one of the mini-apps that approximates the computations of implicit finite elements problems. It is written in about 1500 lines of C++ codes[75]. We used version `2.2.0` released on 2017-11-23, which corresponds to the commit `a322da6`.

MiniMD[82] is the second mini-app we used. It is a scaled down version of Lammps[83], and both are simulations of molecular dynamics. Unlike Lammps, miniMD offers much less input parameters, and is written with less than 3000 lines of C++ code. The version we used in this study is from the commit `7cb2dd8`, released on 2020-02-29.

Lammps[83][84] is a CORAL-2 benchmark simulating molecular dynamics. Its code is modular, as in it is composed of several packages which an user can select to install when compiling the code. The implementations of our methods are tested on the commit `6fd8b2b`, which corresponds to the release tagged `stable_29Sep2021_update3`. The installed packages are: asphere, body, class2, colloid, coreshell, dipole, eff, extra-fix, extra-pair, granular, kspace, manybody, mc, misc,

molecule, mpiio, opt, peri, qeq, replica, rigid, shock, spin, and srd. Those were necessary to run most of the test cases distributed with the release we picked. Lammps is written in hundred of thousands of lines of C++ code. The exact number depends on the number of installed packages.

Pennant[85][86] is a CORAL-2 code benchmarking computations on unstructured meshes. The code we used is from the commit `8c8fec0`, release in February 2016. This benchmark is also written in 3300 lines of C++ code.

MCB[87][88] is a CORAL-1 benchmark simulating the wastes of a nuclear fission reaction using Monte Carlo techniques. It is written in roughly 13000 lines of C code. The version we used is from the commit `0d4535f` released on 2014-10-24.

Bibliography

[1] R. McHaney, *Computer simulation: a practical perspective*. Academic press, 1991.

[2] G. W. Platzman, "The eniac computations of 1950—gateway to numerical weather prediction," *Bulletin of the American Meteorological Society*, vol. 60, no. 4, pp. 302 – 312, 1979.

[3] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[4] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[5] T. N. Theis and H.-S. P. Wong, "The end of moore's law: A new beginning for information technology," *Computing in Science and Engineering*, vol. 19, no. 2, pp. 41–50, 2017.

[6] R. M. Hord, *The Illiac IV: the first supercomputer*. Springer Science & Business Media, 2013.

[7] T. Keller, "Cray-1 evaluation. final report," tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 1976.

[8] M. Flynn, "Very high-speed computing systems," *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[9] J. Hennessy and N. Jouppi, "Computer technology and architecture: an evolving interaction," *Computer*, vol. 24, no. 9, pp. 18–29, 1991.

[10] D. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[11] R. Buyya, "High performance cluster computing," *New Jersey: F'rentice*, vol. 2, 1999.

[12] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, pp. 47–47, 2004.

[13] N. DeBardeleben, S. Blanchard, L. Monroe, P. Romero, D. Grunau, C. Idler, and C. Wright, "Gpu behavior on a large hpc cluster," in *Euro-Par 2013: Parallel Processing Workshops* (D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Costan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, and J. Weidendorfer, eds.), (Berlin, Heidelberg), pp. 680–689, Springer Berlin Heidelberg, 2014.

[14] "top500.org." `https://top500.org/`. Accessed: 2022-10-07.

[15] "Frontier supercomputer debuts as world's fastest, breaking exascale barrier." `https://www.ornl.gov/news/frontier-supercomputer-debuts-worlds-fastest-breaking-exascale-barrier`. Accessed: 2022-11-09.

[16] Z. Majo and T. 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), Association for Computing Machinery, 2011.

[17] "top500 interconnect statistics." `https://www.top500.org/statistics/list/`. Accessed: 2022-10-07.

[18] D. W. Walker and J. J. Dongarra, "Mpi: a standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[19] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[20] "Mpich home page." `https://www.mpich.org/`. Accessed: 2022-11-09.

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

[22] M. Pérache, P. Carribault, and H. Jourdren, "Mpc-mpi: An mpi implementation reducing the overall memory consumption," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (M. Ropo, J. Westerholm, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 94–103, Springer Berlin Heidelberg, 2009.

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

[24] A. Skjellum, M. Rüfenacht, N. Sultana, D. Schafer, I. Laguna, and K. Mohror, "Exampi: A modern design and implementation to accelerate message passing interface innovation," in *High Performance Computing* (J. L. Crespo-Mariño and E. Meneses-Rojas, eds.), (Cham), pp. 153–169, Springer International Publishing, 2020.

[25] "Intel mpi home page." `https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html`. Accessed: 2022-11-09.

[26] J. Backus, "The history of fortran i, ii, and iii," *SIGPLAN Not.*, vol. 13, p. 165–180, aug 1978.

[27] W. A. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*. USA: Elsevier Science Inc., 1975.

[28] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[29] R. Gupta, "Generalized dominators and post-dominators," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 246–257, 1992.

[30] V. C. Sreedhar, G. R. Gao, and Y. Lee, "Dj-graphs and their applications to flowgraph analyses," in *ACAPS Tech. Memo 70*, Citeseer, 1994.

[31] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, (New York, NY, USA), p. 25–35, Association for Computing Machinery, 1989.

[32] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 12–27, 1988.

[33] "LLVM project home page." `https://llvm.org/`. Accessed: 2022-10-15.

[34] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86, IEEE, 2004.

[35] C. Lattner and V. Adve, "The llvm instruction set and compilation strategy," 2002.

[36] A. Diwan, K. S. McKinley, and J. E. B. Moss, "Type-based alias analysis," *SIGPLAN Not.*, vol. 33, p. 106–117, may 1998.

[37] "LLVM memory SSA documentation page." `https://www.llvm.org/docs/MemorySSA.html`. Accessed: 2022-10-15.

[38] R. A. van Engelen, "Efficient symbolic analysis for optimizing compilers," in *Compiler Construction* (R. Wilhelm, ed.), (Berlin, Heidelberg), pp. 118–132, Springer Berlin Heidelberg, 2001.

[39] O. Bachmann, P. S. Wang, and E. V. Zima, "Chains of recurrences—a method to expedite the evaluation of closed-form functions," in *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, ISSAC '94, (New York, NY, USA), p. 242–249, Association for Computing Machinery, 1994.

[40] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee, "A survey of mpi usage in the us exascale computing project," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 3, p. e4851, 2020. e4851 cpe.4851.

[41] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A large-scale study of mpi usage in open-source hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[42] A. Denis, J. Jaeger, E. Jeannot, and F. Reynier, "A methodology for assessing computation/communication overlap of mpi nonblocking collectives," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 22, p. e7168, 2022.

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

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

[45] R. L. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer, "Connectx-2 infiniband management queues: First investigation of the new support for network offloaded

collective operations," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 53–62, 2010.

[46] A. Denis, J. Jaeger, E. Jeannot, and F. Reynier, "One core dedicated to mpi nonblocking communication progression? a model to assess whether it is worth it," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 736–746, 2022.

[47] J. S. Vetter and B. R. De Supinski, "Dynamic software testing of mpi applications with umpire," in *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, pp. 51–51, IEEE, 2000.

[48] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch, "Marmot: An mpi analysis and checking tool," in *Advances in Parallel Computing*, vol. 13, pp. 493–500, Elsevier, 2004.

[49] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller, "Mpi runtime error detection with must: Advances in deadlock detection," *Scientific Programming*, vol. 21, no. 3-4, pp. 109–121, 2013.

[50] "Intel trace analyzer and collector home page." `https://www.intel.com/content/www/us/en/developer/tools/oneapi/trace-analyzer.html`. Accessed: 2022-10-18.

[51] E. M. Clarke, "Model checking," in *Foundations of Software Technology and Theoretical Computer Science* (S. Ramesh and G. Sivakumar, eds.), (Berlin, Heidelberg), pp. 54–56, Springer Berlin Heidelberg, 1997.

[52] S. F. Siegel, "Model checking nonblocking mpi programs," in *Verification, Model Checking, and Abstract Interpretation* (B. Cook and A. Podelski, eds.), (Berlin, Heidelberg), pp. 44–58, Springer Berlin Heidelberg, 2007.

[53] A. Pham, T. Jéron, and M. Quinson, "Verifying mpi applications with simgridmc," in *Proceedings of the First International Workshop on Software Correctness for HPC Applications*, Correctness'17, (New York, NY, USA), p. 28–33, Association for Computing Machinery, 2017.

[54] M. Laurent, E. Saillard, and M. Quinson, "The mpi bugs initiative: a framework for mpi verification tools evaluation," in *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*, pp. 1–9, 2021.

[55] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, may 2018.

[56] Z. Luo, M. Zheng, and S. F. Siegel, "Verification of mpi programs using civl," in *Proceedings of the 24th European MPI Users' Group Meeting*, EuroMPI '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[57] H. Yu, "Combining symbolic execution and model checking to verify mpi programs," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, (New York, NY, USA), p. 527–530, Association for Computing Machinery, 2018.

[58] H. Yu, Z. Chen, X. Fu, J. Wang, Z. Su, J. Sun, C. Huang, and W. Dong, "Symbolic verification of message passing interface programs," ICSE '20, 2020.

[59] F. Ye, J. Zhao, and V. Sarkar, "Detecting mpi usage anomalies via partial program symbolic execution," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, IEEE Press, 2018.

[60] A. Droste, M. Kuhn, and T. Ludwig, "Mpi-checker: Static analysis for mpi," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–10, 2015.

[61] P. Huchant, E. Saillard, D. Barthou, H. Brunie, and P. Carribault, "Parcoach extension for a full-interprocedural collectives verification," in *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)*, pp. 69–76, IEEE, 2018.

[62] P. Huchant, E. Saillard, D. Barthou, and P. Carribault, "Multi-valued expression analysis for collective checking," in *Euro-Par 2019: Parallel Processing*, pp. 29–43, 2019.

[63] J. Jaeger, E. Saillard, P. Carribault, and D. Barthou, "Correctness Analysis of MPI-3 Non-Blocking Communications in PARCOACH," in *European MPI Users' Group Meeting* , EuroMPI '15 The 22nd European MPI Users' Group Meeting, (Bordeaux, France), Sept. 2015.

[64] J.-P. Lehr, T. Jammer, and C. Bischof, "Mpi-corrbench: Towards an mpi correctness benchmark suite," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '21, (New York, NY, USA), p. 69–80, Association for Computing Machinery, 2021.

[65] T. Hoefler, P. Gottschling, W. Rehm, and A. Lumsdaine, "Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pp. 374–382, Springer, 2006.

[66] T. Hoefler and A. Lumsdaine, "Design, Implementation, and Usage of LibNBC," tech. rep., Open Systems Lab, Indiana University, Aug. 2006.

[67] D. J. Holmes, A. Skjellum, and D. Schafer, "Why is mpi (perceived to be) so complex? part 1—does strong progress simplify mpi?," in *Proceedings of the*

27th European MPI Users' Group Meeting, EuroMPI/USA '20, (New York, NY, USA), p. 21–30, Association for Computing Machinery, 2020.

[68] A. Danalis, L. Pollock, M. Swany, and J. Cavazos, "MPI-Aware Compiler Optimizations for Improving Communication–Computation Overlap," in *Proceedings of the 23rd international conference on Supercomputing*, pp. 316–325, 2009.

[69] A. Danalis, L. Pollock, and M. Swany, "Automatic MPI Application Transformation with ASPhALT," in *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–8, IEEE, 2007.

[70] J. Guo, Q. Yi, J. Meng, J. Zhang, and P. Balaji, "Compiler–Assisted Overlapping of Communication and Computation in MPI Applications," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 60–69, IEEE, 2016.

[71] D. Das, M. Gupta, R. Ravindran, W. Shivani, P. Sivakeshava, and R. Uppal, "Compiler–Controlled Extraction of Computation–Communication Overlap in MPI Applications," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, IEEE, 2008.

[72] H. Ahmed, A. Skjellum, P. Bangalore, and P. Pirkelbauer, "Transforming Blocking MPI Collectives to Non-Blocking and Persistent Operations," in *Proceedings of the 24th European MPI Users' Group Meeting*, pp. 1–11, 2017.

[73] D. Quinlan, "ROSE: Compiler Support for Object–Oriented Frameworks," *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000.

[74] V. M. Nguyen, E. Saillard, J. Jaeger, D. Barthou, and P. Carribault, "Parcoach extension for static mpi nonblocking and persistent communication validation," in *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*, pp. 31–39, 2020.

[75] P. S. Crozier, H. K. Thornquist, R. W. Numrich, A. B. Williams, H. C. Edwards, E. R. Keiter, M. Rajan, J. M. Willenbring, D. W. Doerfler, and M. A. Heroux, "Improving performance via mini-applications.," 9 2009.

[76] V. M. Nguyen, E. Saillard, J. Jaeger, D. Barthou, and P. Carribault, "Automatic code motion to extend mpi nonblocking overlap window," in *High Performance Computing* (H. Jagode, H. Anzt, G. Juckeland, and H. Ltaief, eds.), (Cham), pp. 43–54, Springer International Publishing, 2020.

[77] J. Vetter and C. Chambreau, "mpip: Lightweight, scalable mpi profiling," 2005.

[78] Y. Sui and J. Xue, "Svf: Interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, (New York, NY, USA), p. 265–266, Association for Computing Machinery, 2016.

[79] "Coral-1 benchmark list." `https://asc.llnl.gov/coral-benchmarks`. Accessed: 2022-10-20.

[80] "Coral-2 benchmark list." `https://asc.llnl.gov/coral-2-benchmarks`. Accessed: 2022-10-20.

[81] "miniFE github repository." `https://github.com/Mantevo/miniFE`. Accessed: 2020-03-31.

[82] "miniMD github repository." `https://github.com/Mantevo/miniMD`. Accessed: 2020-03-31.

[83] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of computational physics*, vol. 117, no. 1, pp. 1–19, 1995.

[84] "Lammps github repository." `https://github.com/lammps/lammps`. Accessed: 2022-10-20.

[85] C. R. Ferenbaugh, "Pennant: an unstructured mesh mini-app for advanced architecture research," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4555–4572, 2015.

[86] "Pennant github repository." `https://github.com/lanl/PENNANT`. Accessed: 2022-10-20.

[87] "MCB github repository." `https://github.com/arm-hpc/mcb`. Accessed: 2022-10-20.

[88] J. Cetnar, "Mcb: A continuous energy monte carlo burnup simulation code," Nov 1999.