# Optimal parameters determination for the execution of MPI applications on parallel architectures

Richard Sartori

▶ **To cite this version:**

Richard Sartori. Optimal parameters determination for the execution of MPI applications on parallel architectures. Artificial Intelligence [cs.AI]. Université de Bordeaux, 2024. English. NNT : 2024BORD0423 . tel-04907812

**HAL Id: tel-04907812**

**https://theses.hal.science/tel-04907812v1**

Submitted on 23 Jan 2025

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

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

Distributed under a Creative Commons Attribution 4.0 International License

# THÈSE

PRÉSENTÉE À

# L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE
DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Richard Sartori**

POUR OBTENIR LE GRADE DE

# DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

---

## Détermination de Paramètres Optimaux pour le Déploiement d'Applications MPI sur Architectures Parallèles

---

**Date de soutenance :** 19 Décembre 2024

**Devant la commission d'examen composée de :**

| | | |
|---|---|---|
| M. Guillaume MERCIER | Maître de conférences, Bordeaux INP | Directeur |
| Mme. Soraya ZERTAL | Professeur, Université Paris-Saclay | Rapporteuse |
| M. Philippe CLAUSS | Professeur, Université de Strasbourg | Rapporteur |
| M. Olivier BEAUMONT | Directeur de recherche, INRIA | Président |
| Mme. Emmanuelle SAILLARD | Chargée de recherche, INRIA | Examinatrice |
| M. Patrick CARRIBAULT | Directeur de recherche, CEA | Examinateur |

**Membres invités :**

| | | |
|---|---|---|
| M. Pierre LEMARINIER | Product Owner, EVIDEN | Co-Encadrant |
| M. Emmanuel JEANNOT | Directeur de recherche, DDN Japan | Co-Encadrant |

# Contents

**Remerciements** **7**

**Résumé en français** **9**

**Abstract** **10**

**Résumé étendu en français** **11**
Introduction . . . 11
Blackbox Optimization . . . 12
Squelettonisation . . . 13

**1 Introduction** **15**
1.1 The HPC context . . . 15
1.2 Evolution of Supercomputers . . . 16
1.3 MPI . . . 17
1.3.1 Open MPI . . . 18
1.4 Contribution . . . 19
1.5 Document Organization . . . 19

**2 An Overview of Open MPI Tuning** **21**
2.1 Open MPI Architecture . . . 22
2.1.1 MCA Parameters . . . 24
2.1.2 The `tuned` Module . . . 24
2.1.3 Performance Discrepancies between Implementations . . . 24
2.1.4 The Tuning File . . . 26
2.2 State of the Art and Related Work . . . 27
2.2.1 Brute Force Tuners . . . 27
2.2.2 Performance Prediction . . . 28
2.2.3 Incremental Tuners using Probing . . . 29
2.2.4 Black Box Optimization . . . 30
2.2.5 Machine Learning . . . 30
2.3 Problematic . . . 30

**3 Blackbox Optimization** **31**
3.1 Methodology of BBO . . . 32
3.2 State of the Art . . . 33
3.3 Formalization of the Problem . . . 34
3.3.1 The Parameter Space . . . 34
3.3.2 Choice of the Initial Sampling Point . . . 34
3.3.3 Choice of the Next Sampling Point . . . 35

3.3.4 Choice of the Stopping Criterion . . . . . 36
3.4 Agnosticism of the blackbox Approach . . . . . 36
3.5 Pre-existing Software used to Produce the Tuning File . . . . . 37
3.5.1 ACCO . . . . . 37
3.5.2 ShaMAN . . . . . 37
3.5.3 Benchmarks . . . . . 37
3.6 ACCO and ShaMAN Integration to Optimize an MPI Runtime . . . . . 38
3.7 Validity of the Bayesian Optimization over Brute Force . . . . . 39
3.7.1 Experimental Setup . . . . . 39
3.7.2 Experiment Plan . . . . . 39
3.7.3 Evaluation Metrics . . . . . 42
3.8 Comparisons between Brute Force and Baysian Optimizations . . . . . 42
3.8.1 Execution Time Comparison between Bayesian Optimization and Brute Force . . . . . 44
3.8.2 Tuning Time Comparison . . . . . 45
3.8.3 Scalability Study for a Single Collective Operation . . . . . 46
3.9 Conclusion . . . . . 48

**4 Skeletonization 49**
4.1 State of the Art . . . . . 50
4.2 Skeleton Generation . . . . . 51
4.2.1 Communication Pattern . . . . . 51
4.2.2 Communication Variable . . . . . 52
4.2.3 Types of Dependencies between Variables . . . . . 52
4.2.4 Slicing Criterion and Program Slice . . . . . 53
4.3 Application Skeleton . . . . . 54
4.4 A Complete Example (DGEMM) . . . . . 55
4.4.1 The Top-Down Phase . . . . . 57
4.4.2 The Bottom-Up Phase . . . . . 57
4.5 Automatization with LLVM . . . . . 58
4.5.1 LLVM . . . . . 58
4.5.2 An Implementation with LLVM . . . . . 61
4.5.2.1 A Simple Example of a Pass . . . . . 62
4.5.2.2 A More Detailled Example . . . . . 64
4.5.2.3 SkeletonPass Preparation . . . . . 65
4.5.2.4 Code Analysis . . . . . 67
4.5.2.5 Variables Status Propagation . . . . . 69
4.5.2.6 Removal of Unecessary Variables . . . . . 71
4.5.2.7 The LLVM IR Preparation . . . . . 73
4.5.2.8 Limitations of the Skeletonizer . . . . . 74
4.6 Tuning Time Reduction . . . . . 76
4.6.1 Workflow of the Tuning Process using Skeletonization . . . . . 77
4.7 Skeletonization Process Validation . . . . . 78
4.7.1 Viability of the Approach . . . . . 78
4.7.2 Experimental Setup . . . . . 78
4.7.3 Results Discussion . . . . . 78
4.7.3.1 FT . . . . . 78
4.7.3.2 EP . . . . . 78
4.7.3.3 CG . . . . . 79

4.7.3.4 MiniFE . . . . . . . . . . 79
4.7.3.5 Lulesh . . . . . . . . . . 80
4.7.3.6 Evaluation of the Skeleton . . . . . . . . . . 82
4.8 Conclusion . . . . . . . . . . 84

**5 Conclusion** **85**
5.1 Contributions . . . . . . . . . . 85
5.2 Discussion . . . . . . . . . . 86
5.3 Future Work . . . . . . . . . . 87
5.4 Perspectives . . . . . . . . . . 88

**A Benchmarks** **89**
A.1 OSU . . . . . . . . . . 89
A.2 NAS . . . . . . . . . . 89
A.3 MiniFE . . . . . . . . . . 90
A.4 Lulesh . . . . . . . . . . 90

**B User Manual of the Skeletonizer** **91**

**Publications** **99**
Articles in Peer-reviewed Journals . . . . . . . . . . 99

# List of Figures


1.1 Evolution of the computing power in the TOP500 . . . . . . . . . . . . . . 16

2.1 Schematic representation of the Modular Architecture of Open MPI . . . . 22
2.2 Visual representation of an hwloc output . . . . . . . . . . . . . . . . . . . 23
2.3 Communication scheme of a linear implementation of `MPI_Bcast` . . . . . . 25
2.4 Communication scheme of an implementation of `MPI_Bcast` using a binomial tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.5 Communication scheme of a hierarchical implementation of `MPI_Bcast` . . 26

3.1 Schematic representation of the optimization loop . . . . . . . . . . . . . . 33
3.2 Schematic Representation of the Workflow . . . . . . . . . . . . . . . . . . 39
3.3 Execution time gain of using the solution found by Bayesian Optimization compared to the default parametrization (`Pise` machine) . . . . . . . . . . 43
3.4 Execution time gain of using the solution found by Bayesian Optimization compared to the default parametrization (`Bora` machine) . . . . . . . . . . 43
3.5 Number of benchmark runs before reaching stop criterion . . . . . . . . . . 47

4.1 Schematic Representation of the Skeleton Usage . . . . . . . . . . . . . . . 50
4.2 Compilation process using LLVM . . . . . . . . . . . . . . . . . . . . . . . 59
4.3 The inheritance graph of the class `llvm::Constant` . . . . . . . . . . . . . 66
4.4 BasicBlocks with branching between them and their order of processing . . 71
4.5 MiniFE benchmark performance in diverse MPI configurations, `Pise` machine 80
4.6 Lulesh benchmark performance in diverse MPI configurations, `Pise` machine 81
4.7 Relative error in Lulesh performance for the best and worst configuration . 81
4.8 Lulesh benchmark, `Pise` machine . . . . . . . . . . . . . . . . . . . . . . . 82
4.9 Lulesh benchmark, (N=27, PPN=27, Pb_S=90), `Bora` machine . . . . . . 83

# List of Tables


2.1 MCA parameters that can be set using a tuning file (replace * by the name of a collective communication operation) . . . . . . . . . . 26

3.1 Hardware description . . . . . . . . . . 40
3.2 Collectives and their corresponding algorithms . . . . . . . . . . 41
3.3 Median difference in execution time and noise between best parametrization found by Bayesian Optimization and optimal parametrization (`Pise` machine) . . . . . . . . . . 44
3.4 Time to solution for each heuristic and each collective, rounded up to the nearest minute (`Pise` machine) . . . . . . . . . . 45
3.5 Time to solution for each heuristic and each collective, rounded up to the nearest minute (`Bora` machine) . . . . . . . . . . 45
3.6 Median number of iterations performed by Bayesian Optimization compared to exhaustive search . . . . . . . . . . 46
3.7 Tuning time (in minutes) vs. number of nodes (for the Reduce collective operation) . . . . . . . . . . 46

4.1 Expectations . . . . . . . . . . 77
4.2 FT benchmark, 32 nodes, 32 processes per node . . . . . . . . . . 78
4.3 EP benchmark, 32 nodes, 48 processes per node . . . . . . . . . . 79
4.4 CG benchmark, 32 nodes, 32 processes per node . . . . . . . . . . 79

# List of Listings


1.1 Signature of MPI_Send and MPI_Recv . . . . . . . . . . . . . . . . . . . . . . . . 17
2.1 Signature of MPI_Bcast . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.2 Example of an Open MPI configuration file . . . . . . . . . . . . . . . . . . . . . . 27
4.1 Signature of MPI_Send . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.2 Example of a data dependency . . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.3 Example of a control dependency . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.4 Example of a communication dependency . . . . . . . . . . . . . . . . . . . . . . 53
4.5 parallel matrix-matrix multiplication in C . . . . . . . . . . . . . . . . . . . . 56
4.6 Source code equivalent of Listing 4.5 after skeletonization . . . . . . . . . . . 58
4.7 Example of two pointers that are partial aliases . . . . . . . . . . . . . . . . . 59
4.8 Example of Alias Analysis Usage . . . . . . . . . . . . . . . . . . . . . . . . . 60
4.9 A Simple LLVM pass that prints function names . . . . . . . . . . . . . . . . . . 62
4.10 Convoluted hello_world program in C . . . . . . . . . . . . . . . . . . . . . . 62
4.11 Possible output . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
4.12 Human readable version of the LLVM IR . . . . . . . . . . . . . . . . . . . . . . 63
4.13 Portion of a pass eliminating conditional branches . . . . . . . . . . . . . . . 64
4.14 C code allocating memory with malloc . . . . . . . . . . . . . . . . . . . . . . 68
4.15 Generic function to add to the context . . . . . . . . . . . . . . . . . . . . . 69
4.16 Specialized functions to add to the context . . . . . . . . . . . . . . . . . . . 69
4.17 Specialized function for LoadInst . . . . . . . . . . . . . . . . . . . . . . . . 69
4.18 Specialized function for StoreInst . . . . . . . . . . . . . . . . . . . . . . . 70
4.19 LLVM IR of Listing 4.2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
4.20 Code responsible for removing instructions in a block . . . . . . . . . . . . . . 72
4.21 Specialized function to remove a ReturnInst from context . . . . . . . . . . . . 73
4.22 C code approximating $\sqrt{2}$ . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.23 C code using global variable . . . . . . . . . . . . . . . . . . . . . . . . . . 75

# Remerciements

QUATRES années de thèse se sont conclues sur un satisfaisant "Nous vous descernons le titre de Docteur en Informatique de l'Université de Bordeaux". Ce manuscrit a été rédigé, non sans peine, et il ne me reste donc plus qu'à écrire ces remerciements.

Tout d'abord, merci au jury. Merci à Soraya et Philippe pour avoir relu ce manuscript. J'espère que sa lecture a été plus facile pour vous que sa rédaction ne l'a été pour moi. Merci Emmanuelle pour avoir accepté d'être examinatrice, avoir un membre de l'équipe *Storm* dans le jury du thèse de l'équipe *Tadaam* me semble particulièrement intéressant. Merci également à Patrick, membre émérite du CEA pour qui j'espère que mon travail sera utile. Merci à Oliver pour avoir assuré la présidence de ce jury.

Les remerciements suivants s'addressent à ceux sans qui cette thèse n'aurait pas pu exister : Guillaume, Pierre et Emmanuel. Votre soutien et votre encadrement au cours de ces longues années ont été très important pour moi. Merci de m'avoir fait découvrir le monde de la recherche et le monde de l'entreprise. Merci de m'avoir transmis (une partie de) votre immense savoir, votre goût pour la recherche et la méthode scientifique du monde informatique. Vos exigences et les miennes différaient, merci donc de m'avoir enseigné la façon d'aller au bout de mes idées. Malgré les inévitables dissensions, travailler avec vous fût toujours un plaisir très enrichissant. Je me demande quelle part représente votre envestissement dans tous les résultats que nous avons produit. Cette thèse est tout autant la vôtre que la mienne, je vous laisse donc le soin de donner la réponse à cette question.

Un merci collectif à l'équipe qui a maintes fois changé de nom, de membres et même d'entreprise mais dont j'ai toujours fait partie. D'Atos à EVIDEN, un grand merci aux plus permanents François, Piotr, Florent et Emmanuel, mais également à tous ceux qui sont passés par chez nous, Guillaume, Yannis, Romain, Julien, Ahmed, et à tous ceux dont le nom m'est sorti de la tête. Un merci tout particulier à mes camarades de thèse du côté de Grenoble : Cassandra, Radja et Charles. La première m'a tout appris par son parcours et son abnégation, et j'espère avoir pu tout transmettre aux seconds. Je leur souhaite un grand courage pour la suite. Un grand merci également à Quentin, qui m'a apporté un soutien indéfectible et une aide inestimable, que cela soit en C++ ou autour d'un café. Mon petit doigt me dit que tu te trouveras une thèse.

Un merci tout aussi collectif à l'équipe *Tadaam*, sans nul doute la meilleure équipe le l'INRIA Bordeaux! Merci à Brice, Luan, Alexandre, Francieli, Guillaume, Mihail et François pour les échanges autour d'un rubik's cube ou d'un babyfoot, la médiathèque, le droit des logiciels, hwloc, les I/Os, les bancs de test MPI, le Brésil, les États-Unis et le fait de se moquer des étudiants de l'ENSEIRB (je vous laisse associer les activités aux permanents). Ensuite, merci à l'openspace pour m'avoir supporté, dans les deux sens du terme. La capacité de cet openspace à passer de l'ambiance la plus studieuse à la plus insouciante m'impressionera toujours. Merci à mes camarades de thèse du côté de Bordeaux : Alexis, Romain et Luc. Pas tous dans la même équipe mais tous dans le même bateau, vos travaux dans vos thèses m'ont aidé d'une façon ou d'une autre pour la mienne.

Merci aux autres qui sont passé par l'openspace, quel que soit votre statut : Clément, Clément, Corentin, Pierre, Robin, Julien, Jean-Alexandre, Mahamat, Thibaut et Méline. Pour finir avec les membres de l'équipe, merci à Catherine et Fabienne pour leur patience à toute épreuve dans mes démarches administratives.

Merci aux autres équipes HPC du centre, *Storm* et *Topal*. Merci en particulier à Emmanuelle, Hayfa, Alice, Romain et Vincent pour les bons moments passés à découvrir San Fransisco. Merci à Mathieu et Nathalie pour les discussions autour du babyfoot ou à la médiathèque.

Merci également aux autres membres du centre qui gravitent autour de la recherche : Philippe pour sa connaissance extensible de LLVM et du rubik's cube, Emmanuel pour ses conseils sur Guix. Merci également aux personnes qui maintiennent les plateformes de calculs, notamment PlaFRIM qui a été utilisé pour certaines expériences de cette thèse. Il est communément admis qu'il s'agit du seul outil nécéssaire à la recherche en informatique, et ce qui est bien c'est qu'il n'y a pas besoin de nettoyer après les expériences, à l'inverse de la physique-chimie.

On ne remerciera pas le Covid, qui aura grandement chamboulé le planning de cette thèse.

Un immense merci à ma famille, vous m'avez toujours soutenu, et même si vous m'avez posé des dizaines de questions et lu mes articles je doute qu'aucun d'entre vous ne puisse expliquer ce que j'ai fait exactement au cours de ces années. Merci à mes parents et ma tante pour avoir fait le déplacement depuis la Bourgogne-Franche-Comté pour assister à ma soutenance.

Pour finir, merci à toutes celles et ceux que j'ai pu oublier, mais qui ont été à mes côtés au cours de cette thèse.

# Résumé en français

**Résumé** Les supercalculateurs sont utilisés pour traiter des problèmes numériques complexes, comme les simulations, les prévisions météorologiques ou l'intelligence artificielle, nécessitant d'importantes ressources de calcul, inaccessibles aux ordinateurs traditionnels. Composés de multiples et puissants ordinateurs interconnectés, leurs capacités ne cesse de croître. Toutefois, développer des applications capables d'exploiter pleinement cette puissance devient de plus en plus difficile. En effet, divers facteurs doivent être pris en compte : des unités de calcul hétérogènes nécessitant des méthodes de programmation spécifiques, la hiérarchie mémoire, les transferts de données, les communications réseau et l'ordonnancement des tâches. Pour surmonter ces défis, le standard MPI a été créé, offrant une interface unifiée pour faciliter la programmation des supercalculateurs et la gestion des communications entre leurs composants.

Dans cette thèse, nous explorons les différentes façons d'améliorer les performances des applications utilisant MPI en ajustant divers paramètres afin d'exploiter au mieux les ressources matérielles disponibles sur les supercalculateurs. Le travail se concentre sur Open MPI, une implémentation open-source de MPI, et propose des techniques d'optimisation pour réduire le temps et les ressources nécessaires à cette tâche. Parmi celles-ci, nous explorons l'optimisation par boîte noire (BBO), une méthode agnostique vis-à-vis de l'application, qui permet d'explorer l'espace des paramètres de manière efficace. L'idée principale derrière la BBO est d'utiliser des heuristiques intelligentes pour sélectionner les points les plus prometteurs de l'espace des paramètres à évaluer. Plutôt que de tester toutes les combinaisons possibles, la méthode BBO permet d'explorer cet espace en minimisant le nombre d'évaluations nécessaires. En comparaison avec une exploration exhaustive, le temps requis pour réaliser le processus est largement réduit tout en fournissant une solution aux performances pratiquement identiques. Cette approche est particulièrement utile dans le contexte des applications MPI, où l'espace des paramètres peut être gigantesque.

D'autre part, nous introduisons également le concept de squelettisation des applications MPI, une nouvelle technique qui permet d'accélérer le processus d'optimisation en extrayant un "squelette" de l'application. L'extraction est automatisée à l'aide des outils fournis par LLVM, un ensemble d'outils fonctionnant autour d'une représentation intermédiaire. Cela rend l'extraction du squelette indépendante du language de programmation utilisé pour l'application. Ce squelette préserve les éléments essentiels à l'optimisation de l'application, notamment ses schémas de communication, tout en éliminant les aspects calculatoires. Il peut par la suite être utilisé en remplacement de l'application originale dans le processus d'optimisation, ce qui réduit le temps nécéssaire à son déroulement sans compromettre la validité des résultats obtenus.

Les contributions présentées montrent que finement ajuster les paramètres de l'implémentation de MPI a du potentiel pour améliorer les performances des applications HPC.

**Mots-clés** Calcul haute performance, MPI, optimisation, IA, squelettisation

# Abstract

**Abstract** Supercomputers are used to tackle complex numerical problems, such as simulations, weather forecasting, or artificial intelligence, that require significant computational resources, inaccessible to traditional computers. Composed of multiple powerful interconnected computers, their capabilities continue to grow. However, developing applications that can fully leverage this power is becoming increasingly difficult. Indeed, various factors must be taken into account: heterogeneous computing units requiring specific programming methods, memory hierarchy, data transfers, network communications, and task scheduling. To overcome these challenges, the MPI standard was created, providing a unified interface to facilitate the programming of supercomputers and the management of communications between their components.

In this thesis, we explore various ways to improve the performance of applications using MPI by adjusting various parameters to make the best use of the hardware resources available on supercomputers. The work focuses on Open MPI, an open-source implementation of MPI, and proposes optimization techniques to reduce the time and resources required for this task. Among these, we explore Black Box Optimization (BBO), an application-agnostic method that allows for efficient exploration of the parameter space. The main idea behind BBO is to use smart heuristics to select the most promising points in the parameter space to evaluate. Instead of testing all possible combinations, the BBO method enables exploration of this space while minimizing the number of necessary evaluations. Compared to exhaustive exploration, the time required to conduct the process is significantly reduced while providing a solution with virtually identical performance. This approach is particularly useful in the context of MPI applications, where the parameter space can be enormous.

On the other hand, we also introduce the concept of skeletonization of MPI applications, a new technique that accelerates the optimization process by extracting a "skeleton" of the application. The extraction is automated using tools provided by LLVM, a set of tools operating around an intermediate representation. This makes the skeleton extraction independent of the programming language used for the application. This skeleton preserves the essential elements for optimizing the application, including its communication patterns, while eliminating computational aspects. It can then be used in place of the original application in the optimization process, reducing the time required for execution without compromising the validity of the obtained results.

The contributions presented demonstrate that fine-tuning the parameters of the MPI implementation has the potential to improve the performance of HPC applications.

**Keywords** High-Performance Computing, MPI, optimization, AI, skeletonization

# Résumé étendu en français

## Introduction

Le calcul haute performance (HPC) constitue une pierre angulaire dans la résolution de problèmes calculatoires complexes, notamment dans des domaines tels que la simulation climatique, l'astrophysique, la découverte de nouveaux médicaments, et l'intelligence artificielle. Ces applications nécessitent des ressources de calcul massives, que seuls les supercalculateurs peuvent offrir. Ces machines, composées de milliers voire de millions de processeurs interconnectés, permettent d'effectuer des calculs à une échelle inatteignable pour des ordinateurs traditionnels. Les supercalculateurs, tels que Frontier, qui figure parmi les plus puissants au monde, sont devenus des outils incontournables, bien que leur conception et leur maintien représentent des coûts colossaux. Frontier, par exemple, rassemble plus de 8 millions de cœurs et utilise une combinaison de CPUs et de GPUs, marquant ainsi l'évolution vers des architectures de plus en plus hétérogènes.

L'augmentation exponentielle des performances des supercalculateurs a historiquement suivi la loi de Moore, qui prédisait un doublement du nombre de transistors sur une puce tous les deux ans. Cependant, cette progression a été freinée par les limites physiques des processeurs traditionnels, principalement en raison des problèmes liés à la dissipation thermique. En conséquence, la fréquence des processeurs ne peut plus être augmentée indéfiniment, entraînant une stagnation dans l'évolution des performances. Cette limitation a conduit à une réorientation vers des systèmes multiprocesseurs, où les gains de performance sont réalisés en combinant plusieurs processeurs dans des architectures massivement parallèles.

Pour tirer parti de cette évolution, des technologies comme les GPU (processeurs graphiques), capables d'exécuter des tâches massivement parallèles, sont de plus en plus intégrées aux supercalculateurs. Cette transition vers des architectures hétérogènes, combinant CPUs, GPUs et d'autres types d'accélérateurs spécialisés comme les FPGAs, a profondément modifié le paysage du calcul haute performance. Cependant, gérer cette diversité matérielle pose des défis importants, tant en termes de programmation que d'optimisation des performances.

C'est dans ce contexte que l'interface de passage de messages (MPI) s'est imposée comme un standard pour la programmation parallèle. Créée pour pallier les problèmes de portabilité entre les différentes bibliothèques propriétaires de communication inter-processus, MPI permet aux applications de communiquer efficacement sur des systèmes distribués. Son indépendance vis-à-vis du matériel sous-jacent en fait un outil clé pour le développement d'applications HPC. Les bibliothèques MPI permettent de gérer les communications entre processus répartis sur différents nœuds d'un supercalculateur, en offrant une interface standardisée pour les opérations telles que l'envoi et la réception de messages, la synchronisation et les communications collectives.

Nous nous intéressons principalement à l'implémentation Open MPI. Elle est largement

adoptée dans les environnements HPC pour sa modularité et sa capacité à s'adapter à différents matériels et réseaux. Open MPI est conçue pour être extensible, permettant d'ajouter des modules et des composants dynamiques qui optimisent les performances pour des environnements spécifiques. Elle est notamment utilisé dans des infrastructures à grande échelle où des réseaux rapides comme InfiniBand et Ethernet sont déployés. Cependant, bien que MPI fournisse les bases nécessaires pour des communications efficaces, la complexité des architectures modernes nécessite un réglage fin des paramètres afin d'exploiter pleinement les ressources matérielles.

L'objectif de cette thèse est d'explorer et de proposer des méthodes d'optimisation pour les applications utilisant MPI, notamment Open MPI. Parmi les approches étudiées, l'optimisation par boîte noire (BBO) se distingue comme une méthode puissante pour explorer efficacement l'espace des paramètres sans connaissance préalable des caractéristiques de l'application. Cette approche repose sur des heuristiques intelligentes qui permettent d'identifier rapidement les configurations les plus prometteuses, réduisant ainsi considérablement le temps nécessaire à l'optimisation par rapport à une exploration exhaustive de l'ensemble des paramètres.

De plus, la thèse introduit le concept de squelettisation des applications MPI, une technique innovante qui vise à simplifier le processus d'optimisation. Cette méthode consiste à extraire un squelette de l'application, c'est-à-dire une version allégée qui conserve les schémas de communication essentiels tout en éliminant les calculs lourds. Ce squelette est utilisé pour effectuer des bancs de tests et ajuster les paramètres, réduisant ainsi le temps d'optimisation sans compromettre la validité des résultats. L'extraction du squelette est réalisée de manière automatisée à l'aide des outils fournis par LLVM, une infrastructure de compilation.

Les contributions de cette thèse montrent que l'ajustement fin des paramètres de l'implémentation MPI, en particulier via des méthodes d'optimisation avancées comme la BBO et la squelettisation, peut significativement améliorer les performances des applications HPC, tout en minimisant le temps et les ressources nécessaires pour atteindre ces performances optimales.

## Blackbox Optimization

Le chapitre 3 de cette thèse porte sur l'application de l'optimisation par boîte noire (BBO) pour l'ajustement des paramètres de l'implémentation MPI. La complexité des architectures modernes et la diversité des configurations possibles rendent cette tâche difficile et fastidieuse, surtout si l'on considère une recherche exhaustive de l'optimum, car l'espace de paramètres croît de façon exponentielle. Pour répondre à ce défi, la méthode BBO est employée afin d'explorer seulement une partie restreinte de cet espace tout en maintenant la qualité du résultat.

L'approche par boîte noire se caractérise par son agnosticisme vis-à-vis de l'application optimisée, traitée comme une entité dont on peut uniquement évaluer les performances sur des points de l'espace des paramètres, sans connaissance préalable de sa structure ou de ses caractéristiques internes. Cette méthodologie repose sur une exploration guidée de l'espace des paramètres en se basant sur des heuristiques. L'heuristique utilisée ici est l'optimisation bayésienne, qui permettent d'identifier rapidement les configurations les plus prometteuses. Le processus d'optimisation utilise une fonction d'acquisition qui équilibre l'exploration et l'exploitation de l'espace des paramètres : il privilégie les zones peu explorées avec une grande incertitude tout en tenant compte des zones où les performances s'avèrent déjà

intéressantes. De plus, un modèle probabiliste (les processus gaussiens) est employé pour estimer la moyenne et l'écart-type des performances de chaque configuration testée. Ce modèle fournit une prédiction de l'efficacité des configurations potentielles et oriente le choix des points de l'espace des paramètres à évaluer.

Dans la phase expérimentale, des configurations optimales sont recherchées pour quatre opérations collectives courantes en MPI: broadcast, gather, reduce et allreduce. Les expérimentations sont conduites sur deux plateformes distinctes (Pise et Bora), chacune ayant des configurations matérielles spécifiques, avec un nombre variable de processus MPI par nœud et des tailles de messages allant de 4 octets à 1 Mo. Pour chaque opération collective, le temps d'exécution des bancs de tests est mesuré à l'aide de configurations par défaut et de configurations optimisées par BBO. Les résultats montrent que la BBO réduit le temps d'optimisation de 95% en moyenne par rapport à une recherche exhaustive, tout en atteignant une précision de réglage d'environ 6% par rapport à l'optimum global (avec une médiane à 0,7%).

Pour implémenter cette méthodologie, deux outils, ACCO et ShaMAN, sont intégrés pour former une boucle de réglage efficace et automatisée. ACCO pilote les bancs de tests OMB, collectant les données de performance des configurations testées, tandis que ShaMAN applique des modèles prédictifs pour estimer les configurations prometteuses. Ce flux de travail est conçu pour fonctionner efficacement sur des clusters en temps réel, en utilisant un gestionnaire de tâches (comme SLURM) pour distribuer les tâches MPI aux nœuds du cluster. La coordination de ces deux outils n'a besoin d'être lancée qu'une seule fois par cluster pour obtenir un réglage des paramètres MPI, applicable à une variété d'applications HPC.

L'analyse finale met en lumière la scalabilité de cette approche : les configurations optimisées par BBO s'adaptent bien à l'augmentation du nombre de nœuds et à la diversité des architectures matérielles. En moyenne, l'optimisation apporte une amélioration de 48,4% en performances par rapport aux paramètres par défaut d'Open MPI (52,8% en médiane), avec des gains plus marqués dans des configurations à fort parallélisme (par exemple, lorsqu'un seul processus MPI est assigné par cœur). Cette flexibilité et adaptabilité rendent cette approche particulièrement bien adaptée aux environnements HPC modernes, où les exigences de performance et de coût en ressources sont élevées.

En conclusion, l'utilisation de la BBO avec optimisation bayésienne dans le cadre de MPI permet de trouver un compromis optimal entre le temps de réglage et la qualité de l'ajustement des performances, offrant une solution efficace et extensible pour des clusters HPC de grande envergure.

## Squelettonisation

Le chapitre 4 de cette thèse introduit une méthode innovante de squelettisation pour optimiser les applications MPI, une technique conçue pour simplifier le processus de réglage des paramètres de communication dans les environnements HPC. Cependant, ce réglage peut être très long, surtout si l'application elle-même est lourde à exécuter. La squelettisation vise à résoudre ce problème en produisant une version simplifiée de l'application, dénommée squelette. Celui-ci conserve uniquement le schéma de communication de l'application, éliminant ainsi les calculs internes, ce qui réduit de manière significative le temps d'exécution lors des étapes de réglage.

Dans l'état de l'art des approches de parallélisation, les squelettes algorithmiques sont utilisés pour abstraire les schémas de calcul parallèles et réduire la complexité du code.

Cependant, les méthodes existantes nécessitent souvent que l'application soit structurée autour de ces squelettes, ce qui limite leur applicabilité. La méthode présentée dans ce chapitre permet de générer un squelette pour n'importe quelle application MPI sans que celle-ci soit conçue en amont pour s'y prêter.

La génération du squelette d'une application MPI repose sur plusieurs concepts : l'analyse du schéma de communication, la détection des variables de communication et le critère de découpage du programme. Le processus commence par une première phase descendante pour identifier les variables critiques des routines MPI (appelées variables de communication), suivie d'une phase ascendante où leur statut de dépendance est propagé pour déterminer quelles instructions doivent être conservées. Les instructions non essentielles au schéma de communication sont ensuite supprimées, ce qui aboutit à un programme minimal, capable de reproduire le comportement de communication de l'application d'origine, mais avec un temps d'exécution considérablement réduit.

Afin de démontrer la méthode, un exemple complet est présenté où les étapes de simplification du code sont décrites en détail. Cette approche est ensuite automatisée à l'aide de LLVM, un ensemble d'outils de compilation. L'utilisation de passes LLVM personnalisées permet d'analyser et de modifier le code de manière automatisée, rendant la méthode de squelettisation applicable même à de grandes bases de code. La passe de squelettisation est implémentée pour détecter les variables de communication et les instructions nécessaires, en marquant celles-ci pour garantir que le schéma de communication est préservé. Finalement, la structure du squelette est optimisée, garantissant une exécution rapide et ainsi une efficacité maximale pour le réglage des paramètres MPI.

Plusieurs limitations sont également discutées de cette approche dans ce chapitre. Par exemple, les applications dépendant de variables globales ou utilisant le système d'exceptions de C++ ne sont pas directement prises en charge, bien que certaines optimisations permettent de contourner ces limitations dans certains cas.

Les tests expérimentaux démontrent que l'utilisation du squelette dans le processus de réglage réduit le temps d'optimisation de manière significative par rapport à l'utilisation de l'application complète.

En conclusion, la squelettisation proposée dans cette thèse offre une méthode puissante et adaptable pour optimiser les applications MPI. Elle ouvre la voie à des réglages plus rapides et plus efficaces, réduisant ainsi les temps d'exécution des applications HPC.

# Chapter 1

# Introduction

## Contents

**1.1 The HPC context** . . . . . . . . . . . . . . . . . . . . . . . . . . . **15**
**1.2 Evolution of Supercomputers** . . . . . . . . . . . . . . . . . . . . . **16**
**1.3 MPI** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **17**
1.3.1 Open MPI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
**1.4 Contribution** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **19**
**1.5 Document Organization** . . . . . . . . . . . . . . . . . . . . . . . . . **19**

## 1.1 The HPC context

High-Performance Computing (HPC) is a scientific field using large scale computing platforms, called supercomputers, to solve advanced computational problems. HPC has become an indispensable framework across a broad spectrum of scientific and industrial applications, enabling the simulation and analysis of highly complex phenomena that are beyond the reach of conventional computing resources. Use cases range from climate modeling [9], astrophysics [48], and drug discovery [27] to large-scale data analysis in fields such as genomics [46] and artificial intelligence [56]. In this aspect, supercomputers must adapt to a wide variety of constraints and requirements, coming the from multiple independent applications that the computer can run concurrently. The building and maintaining costs of these platforms can only be justified by the opportunity of running a large panel of applications. The current leader of the TOP500 [60], Frontier, has an estimated cost of $600M. At the time of writing, it is composed of 9472 CPUs and 37888 GPUs, for a total number of cores exceeding 8 millions. With these extreme magnitude orders, new programming and optimization problems emerges.

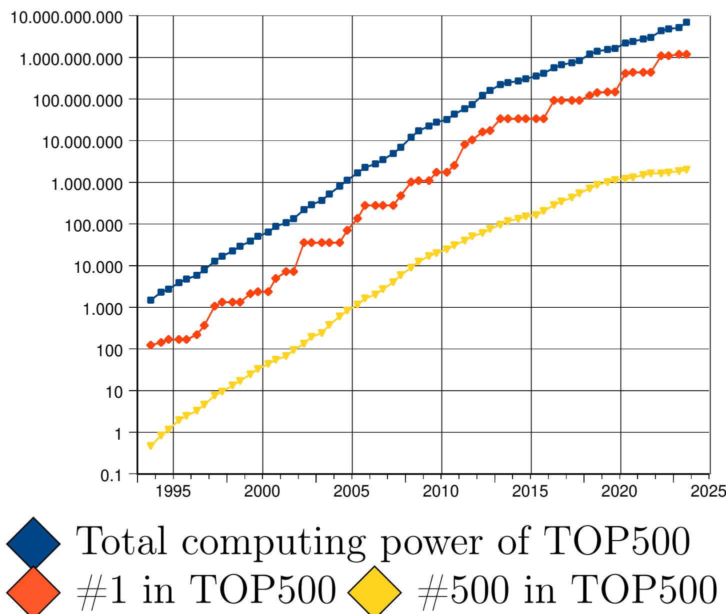



Figure 1.1: Evolution of the computing power in the TOP500[1]

## 1.2 Evolution of Supercomputers

Historically, *Cray Research* designed the first supercomputer in 1975. As adding new components to a single module became increasingly complex, they started to split their computer into smaller interconnected units, creating *Cray-1*, the first supercomputer.

Nowadays, the number of elements aggregated in a supercomputer are orders of magnitude higher, reaching a combined computing power over 1 EFLOP/s ($10^{18}$ FLOP/s). The FLOP/s, or Floating-Point Operations per Second, is the standard unit for measuring the performance of a supercomputer. For comparison, the Cray-1 had an estimated theorical peak performance of 160 MFLOP/s.

As presented in Figure 1.1, the evolution of supercomputers has been driven by the growing demand for increased computational power across a wide range of scientific, industrial, and defense applications. Historically, this demand was met through the consistent advancement of semiconductor technologies, as predicted by Moore's Law, which stated that the number of transistors on a chip would roughly double every two years, leading to exponential increases in computational performance. This fast scaling allowed for consistent improvements in processor speeds and capabilities, which fueled the development of ever more powerful supercomputers capable of tackling larger and more complex problems.

However, the era of continual CPU (Central Processing Unit) frequency scaling came to an halt due to fundamental physical limitations known as Dennard's scaling. According to Dennard's scaling theory, as transistors become smaller, power density would remain constant, allowing for higher clock speeds without significant increases in power consumption. But as transistors approached nanometer scales, heat dissipation issues and power leakage caused this principle to break down. As a result, further increases in CPU clock frequencies became impractical, creating a bottleneck in the continued evolution of computing power through traditional means.

[1] AI.Graphic, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=33540287`

To address this bottleneck, the solution to increase the computational power shifted from enhancing a single processor's capabilities to combining multiple processors into a unified system. This architectural change led to the rise of supercomputing clusters, where numerous processors, each housed in separate nodes, are interconnected through high-speed networks. This shift allowed for significant performance improvements without relying on clock speed increases.

Over time, multiple new components emerged, called accelerators. They are designed to perform a single task, but significantly faster than a CPU. The prime example is the Graphics Processing Units (GPUs) which can, with their highly parallel architecture, offer substantial performance improvements for many scientific applications that require massive parallelism. The architecture of supercomputers then stopped relying solely on CPUs to integrating various accelerators, most notably Graphics Processing Units, in ever increasing numbers. Supercomputers have progressively adopted a more heterogeneous architectures, extensively incorporating GPUs and, to a lesser extent, FPGAs (Field-Programmable Gate Arrays), a type of configurable integrated circuit.

With this shift towards heterogeneous architectures, programming models had to evolve to accommodate the complexity of managing diverse hardware components in supercomputing clusters. The Message Passing Interface has emerged as a key framework to address this challenge, enabling parallel processing across distributed and heterogeneous systems.

## 1.3 MPI

The Message Passing Interface, or MPI, is the *de facto* standard for parallel programming in distributed computing environments, currently in version 4.1. Introduced in 1993, it has become the backbone of HPC applications due to its ability to efficiently manage communications between processes running across multiple nodes in a supercomputing cluster. Unlike shared-memory models, where multiple processes access a common memory space, MPI enables communication between processes that have separate memory, making it well-suited for large-scale distributed systems where data needs to be exchanged between different compute nodes.

The MPI standard was introduced to adress the lack of portability of the different proprietary communication libraries that were historically used for inter-process communication. Thus, MPI is not tied to any specific hardware or architecture to ensure its portability across different computing systems. The standard provides a comprehensive set of functionalities for data transfer, synchronization, and collective communication operations, allowing developers to implement complex parallel algorithms.

```
int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest
    , int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
    int tag, MPI_Comm comm, MPI_Status *status);
```

Listing 1.1: Signature of `MPI_Send` and `MPI_Recv`

At its core, MPI is a hardware-agnostic interface for inter-process communication. Its most fundamental operation is the point-to-point communication, allowing two MPI processes to send and receive a message respectively. The signature of the two functions used to realize this simple data exchange can be seen in Listing 1.1. The `buf`, `count` and `datatype` parameters represent the data exchanged. The `count` is the number of elements of type `datatype` exchanged. The elements are read from the `buf` of the sender

and written in `buf` of the receiver, which needs to be large enough to contain the data. The value of `count` of the receiving process must be greater or equal to the `count` value of the sending process. The `datatype` is either a *basic datatype*, which corresponds to a type in the host language, or a *derived datatype*. A *derived datatype* is constructed with a sequence of types (basic or derived), each with a displacement. It allows the user to define their own datatypes and use them to send more complex elements, such as for example a C structure. The sequence of types, displacements omitted, is called the signature of the *derived datatype*. If a *basic datatype* is used, it must match for both the sender and the receiver, while in the case of *derived datatype* only the signature must be identical. The `comm` parameter represents an MPI communicator, a concept defined in the standard. It represents a communication context. This communicator must also be the same for both callers. The `dest` and `source` are the identifiers of the receiving process and sending process respectively, in the context of the communication (the communicator). The `tag` parameter is an additional information bundled with the message. The combination of `src`, `dest`, `tag` and `comm` is called the message enveloppe, and ensures correct ordering when multiple independents messages are exchanged by the same pair of processes. The `status` parameter is used to obtain informations about the exchange. Finally, the vast majority of MPI routines return an integer that indicates if the call was successful, allowing for error checking. The MPI standard includes hundreds of routines and defines official bindings in C and Fortran.

There are several open source MPI implementations, the most widespread being MPICH [30] and Open MPI [26], which will be presented in Section 1.3.1. These two projects serve as the basis for most of the other implementations of the standard including, without being limited to, commercial implementations from Intel, Microsoft or NEC.

### 1.3.1 Open MPI

Open MPI is an open source implementation of the MPI standard, that fully conforms to its 4.1 version. It is widely used by supercomputers listed in the TOP500. Developed and maintained by a consortium of partners from academia, research, and industry, Open MPI benefits from a community-driven development approach, facilitating the integration of new features. Open MPI is designed to be modular and extensible, supporting various components through dynamically loaded backends. For example, it provides support for multiple network interfaces (*e.g.,* InfiniBand, Ethernet) and plugins for different interconnects (*e.g.,* UCX, PSM2), enabling performance optimizations. This flexibility allows it to adapt to evolving hardware and software environments. Open MPI is suitable for clusters requiring adaptability to changing network hardware or where modularity and ease of integration with different systems are essential. It is also well-suited for mixed HPC environments that undergo frequent reconfigurations.

## 1.4 Contribution

In this thesis, we will investigate the Message Passing Interface, mainly focusing on Open MPI, which plays a fundamental role in parallel programming within HPC systems. Efficient communications between processes is crucial for achieving high performance. While MPI provides a standardized and portable way to handle this communication, the complexity of hardware architectures and network configurations can introduce bottlenecks. Factors such as network latency, bandwidth, and process placement can significantly impact the running time of an application. Achieving optimal performance thus requires careful tuning, that can be done by leveraging Open MPI 's flexibility and scalability. We will explore the various parameters within Open MPI that can be adjusted to enhance its efficiency, such as algorithms, segment sizes, and the configuration of collective operations. Beyond reviewing existing methods, ranging from brute-force approaches to more advanced black-box optimization techniques, we will introduce new strategies designed to streamline and automate the tuning process.

A key focus of this research will be the development of novel tuning techniques aimed at reducing the time and computational resources required for optimization. This includes the use of proxy applications through skeletonization, where a simplified version of an MPI application is employed during the tuning phase to significantly decrease the runtime without compromising accuracy. We will also investigate how machine learning and statistical models, such as Bayesian Optimization, can be integrated into MPI tuning to intelligently navigate the vast parameter space and identify optimal configurations faster than traditional methods. Through these innovations, we aim to contribute with new tools and methodologies that will make MPI tuning more adaptive, scalable, and applicable to a broader range of applications and hardware architectures, ultimately enhancing the performance of HPC systems in real-world scenarios.

## 1.5 Document Organization

The remainder of this document is organized as follows: Chapter 2 presents the architecture of Open MPI and provides an overview of the various tuning options available through its Modular Component Architecture. It also reviews the state-of-the-art methods for MPI tuning, including brute-force approaches and performance prediction techniques, establishing the context and challenges associated with the tuning process.

Chapter 3 introduces the Black Box Optimization approach for MPI tuning. This chapter details the methodology and formalization of the problem, describing how Bayesian Optimization can be used to efficiently explore the large parameter space without prior knowledge of the application's characteristics. It includes a comparative study of brute-force methods and black-box optimization, demonstrating the latter's efficiency in finding optimal configurations with reduced computational cost.

We extend the tuning approach with the concept of skeletonization in Chapter 4. We developped a method to extract the so-called skeleton of an MPI application, a simplified version that retains its core communication pattern. By using this skeleton for tuning instead of the full application, we significantly reduce the time and computational resources required. The chapter also covers the automation of the skeletonization process using LLVM and validates the effectiveness of this approach through experimental results.

In the conclusive Chapter 5, we make a synthesis of the contributions and discuss the implications of the work. This chapter also outlines potential future work, suggesting how

the methodologies developed can be further enhanced and applied to broader contexts, such as real-time adaptive tuning or integration with emerging heterogeneous architectures.

# Chapter 2

# An Overview of Open MPI Tuning

## Contents

**2.1 Open MPI Architecture** . . . . . . . . . . . . . . . . . . . . . . **22**

2.1.1 MCA Parameters . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

2.1.2 The `tuned` Module . . . . . . . . . . . . . . . . . . . . . . . . . 24

2.1.3 Performance Discrepancies between Implementations . . . . . . 24

2.1.4 The Tuning File . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

**2.2 State of the Art and Related Work** . . . . . . . . . . . . . . . . **27**

2.2.1 Brute Force Tuners . . . . . . . . . . . . . . . . . . . . . . . . . 27

2.2.2 Performance Prediction . . . . . . . . . . . . . . . . . . . . . . . 28

2.2.3 Incremental Tuners using Probing . . . . . . . . . . . . . . . . . 29

2.2.4 Black Box Optimization . . . . . . . . . . . . . . . . . . . . . . . 30

2.2.5 Machine Learning . . . . . . . . . . . . . . . . . . . . . . . . . . 30

**2.3 Problematic** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **30**

Tuning is a crucial part in the developpement of applications destined to run on the large scale supercomputers. It is done toachieve the most performance out of the underlying hardware and reach the smallest possible time to run. We mostly focus our efforts on tuning one implementation of the MPI standard: Open MPI. It seemed the easiest to tune, but most of our techniques could be applied to other implementations as well as to other runtimes.

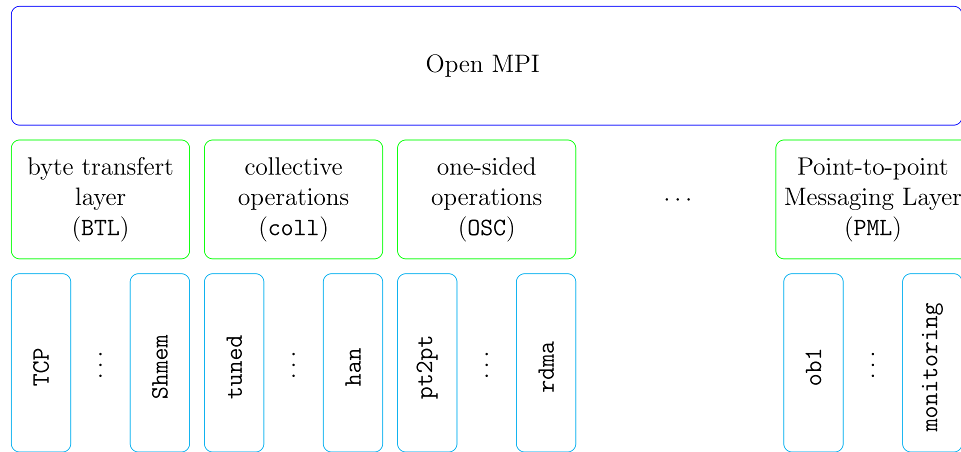



Figure 2.1: Schematic representation of the Modular Architecture of Open MPI

## 2.1 Open MPI Architecture

Open MPI is primarily intended for use on supercomputers. Architecture, hardware, interconnect, accelerators are among the many elements that can widely differ from one cluster to another, and its the role of the MPI implementation to allow all these components to communicate. To offer both performance and an hardware-agnostic implementation, Open MPI is designed around the Modular Component Architecture (MCA). It is a ensemble of *frameworks*, interfaces that collectively cover the entire MPI standard. Each framework is implemented by a *component*, itself composed of one or multiple *modules*. At launch time, before the application runs, modules are assembled into components, which are then assembled into a fully furnished MPI implementation.

**Frameworks** An MCA framework is an interface responsible for providing a specific element of the MPI standard, for example providing MPI collective operation functionality. The framework is also responsible, at runtime, for finding and dynamically loading components that implements it.

**Components** An MCA component is an implementation of a framework's interface. It is a standalone collection of code that can be bundled into a plugin. It is either a static library, that will be inserted into the application at compile-time, or a dynamic library which will be dynamically found and loaded into the application at runtime.

**Modules** An MCA module is an instance of a component, that is usually centered a very specific use case of the component. For example, the `coll` component can manage both the `Shmem` and `tuned` modules, the former being specifically designed for intra-nodes communications using shared memory and the latter being a more generic and widely usable implementation.

Frameworks, components, and modules can be dynamic or static, and can be inserted in the application either at runtime or at compile-time. Open MPI can be thus adapted to the architecture as the user can decide which component to use, to best suit their hardware and/or their application.

Figure 2.1 presents the modular architecture of Open MPI. Frameworks are represented in green, with an emphasis on the `coll` component, which is responsible for providing

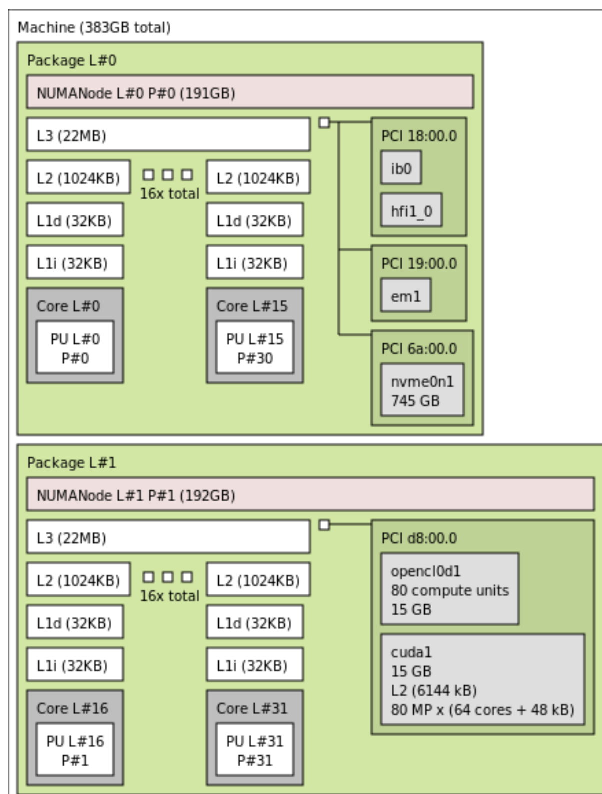



Figure 2.2: Visual representation of an hwloc output

the collective communication operations. The Point-to-point Management Layer (`PML`) component provides the basic functionality of allowing two MPI processes to send and receive messages. The One Sided Communication (`OSC`) component is responsible for giving the processes the ability to expose part of their memory for other processes to read and write to it. Modules are represented in light blue. We will focus on the `tuned` module, which offers multiple implementations for each collective communication operations. The user can chose which one to use depending on the circumstances through a tuning file, described in Section 2.1.4. The different implementations may yield different performances, there is no best algorithm, as can be seen in Section 2.1.3. It is up to the user to tune their system and find which implementation is best suited for their specific use case and hardware. The `coll` module is usually not used alone, as the shared memory (`sm`) module is oftenly loaded concurrently and used for intra-node communications.

Open MPI, along with MPICH, are usually coupled with hwloc [15]. From their online documentation: "The Portable Hardware Locality (hwloc) software provides a portable abstraction (across operating systems, versions, architectures, . . . ) of the hierarchical topology of modern architectures, including NUMA memory nodes (DRAM, HBM, non-volatile memory, CXL, etc.), processor packages, shared caches, cores and simultaneous multithreading." MPI implementation primarily use hwloc to acquire informations about the computing hardware, as its complexity increases, to make the most efficient use of its parallelism.

### 2.1.1 MCA Parameters

MCA parameters serve as the fundamental unit for runtime tuning in Open MPI. These parameters are simple key-value pairs widely used throughout the codebase to replace critical constants. A straightforward example would be the threshold between short and long messages. Short messages are transmitted eagerly without waiting for synchronization with the receiver, while long messages use a rendezvous protocol. The discriminating factor between these two protocols is the total size of the message (in bytes). By defining this threshold as an MCA parameter, users or system administrators can adjust it at runtime to suit a specific application or hardware. For instance, a value that works well for 100 Mbps Ethernet may be unsuitable for Gigabit Ethernet. MCA parameters can be set in four different ways, listed here by precedence:

- editting the command to be executed, by adding `-mca key "value"` to its arguments
- through an environment variable: `export OMPI_MCA_key="value"`
- by defining an MCA parameter file, where each line sets a value with `key = value`, and then supplying the file with the `-am` option
- editting one of the two MCA parameter files that are defined and automatically used by the implementation: user supplied values in `$HOME/.openmpi/mca-params.conf` and system supplied values in `$prefix/etc/openmpi-mca-params.conf`. The former takes precedence over the latter

### 2.1.2 The `tuned` Module

As explained in Section 2.1, the `tuned` module is an implementation of the `coll` component that provides several algorithms for each of the collective communication operations. If this module is selected by the implementation, or explicity required by the user, several MCA parameters become available to the user. They allow the user to set, for each collective communication operation, which algorithm is used, with which segment size and fan-in/out value. Table 3.2 list all possible algorithms for a selection of collective communication operations. Their performance might differ from hardware to hardware and from application to application, and are discussed in Section 2.1.3. The segment size is the size in bytes of the messages sent on the underlying hardware. Setting this value improves performance when messages requiring fragmentation are sent, as the supported message size varies with the interconnect used by Open MPI. The fan-in/out value dictates the maximal number of children when a tree-based algorithm is used, and is ignored otherwise. The available MCA parameters also allow the user to provide a *tuning file*, further described in Section 2.1.4, which is a more complete and succint way of setting up the Open MPI runtime environment. The `tuned` module is enabled by default, but as of Open MPI 5.0, the `han` [3] module will supersede it as the new default. This new module is specifically designed to handle hierarchical collective communication operations, with the possibility of using different algorithms depending on the level at which the operation takes place, similar to Figure 2.5.

### 2.1.3 Performance Discrepancies between Implementations

The performance of a collective communication algorithm depends on a number of parameters, such as the hardware, the message size, the number of nodes, etc. In this Section,

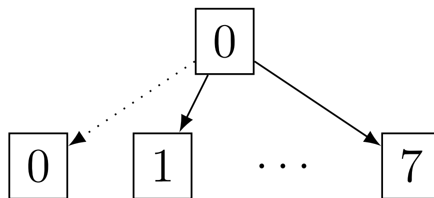



Figure 2.3: Communication scheme of a linear implementation of `MPI_Bcast`

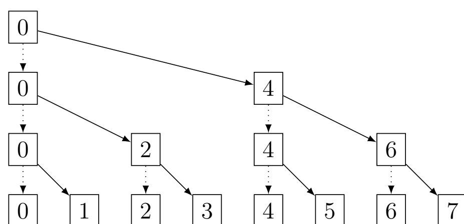



Figure 2.4: Communication scheme of an implementation of `MPI_Bcast` using a binomial tree

we present three possible implementations of the MPI_Bcast collective communication operation. MPI_Bcast, is the collective communication operation where the MPI process with rank `root` sends some data to all other MPI processes in a communicator. Its signature is presented in Listing 2.1. The data is described by the three parameters `buffer`, `count` and `datatype`. In the following examples, the `root` shall be rank 0 of the communicator used by the collective communication operation. Arrows represent point-to-point communications. If an arrow is dotted, it means that the algorithm requires a self communication for a process, and no actual data transmission occurs.

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
  MPI_Comm comm)
```

Listing 2.1: Signature of `MPI_Bcast`

Figure 2.3 presents the communication scheme of a very naive implentation, where the rank 0 sequentially sends the data to all other processes. It has the advantage of being very easily implemented, but its major drawback is its lack of parallelization, which severly impacts performance. This is usually not the preferred implementation, however it might be the best performing algorithm if there are very few proceses, or if the hardware is specifically designed for it.

Figure 2.4 shows the communication scheme of an implementation that maximizes the number of parallel point-to-point communications, using a binomial tree. First, the communicator is divided in two sub-communicators, with process rank 0 being one of the root. Then process rank 0 sends the data to the other root, which in this case is process rank 4. This routine can then recurse until every process has received the data. It is the preferred solution when all communications are equivalent, which is usually the case at the scale of a single node. When dealing with multiple nodes, inter-node communications are usually more time consumming than intra-node communications, rendering this implementation less appealing.

Finally, Figure 2.5 presents a hierarchical approach of the problem, where multiple implementations can be used at different scales on the machine. For example, an algorithm that minimizes the number of communications can be used at the swicth level, where

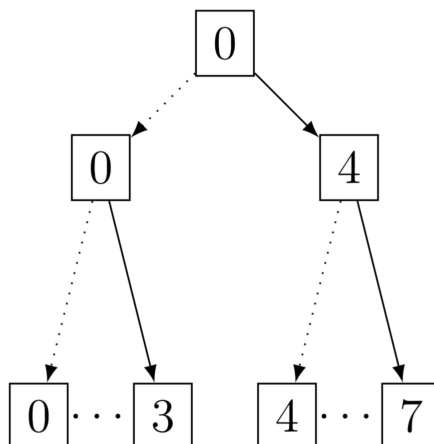



Figure 2.5: Communication scheme of a hierarchical implementation of `MPI_Bcast`

| Parameter name | Variable name | Description | Default |
|---|---|---|---|
| Algorithm | OMPI_MCA_coll_tuned_*_algorithm | Which * algorithm is used | 0 |
| Segment size | OMPI_MCA_coll_tuned_*_algorithm_ segmentsize | Segmentation size in bytes used by default for * algorithms | 0 |
| Fan in out | OMPI_MCA_coll_tuned_*_algorithm_tree_ fanout | Fanin/out for n-tree used for * algorithms | 4 |

Table 2.1: MCA parameters that can be set using a tuning file (replace * by the name of a collective communication operation)

communications are very time consumming, while a binomial tree is used at the node level, where communications are cheaper. This approach might be privilegied for large scale supercomputers. That type of algorithm is also provided by the `han` module.

As we can see, there is no best algorithm that outperforms all others in all situations, and that is why tuning is required for any application that is to be run on a supercomputer.

### 2.1.4 The Tuning File

The `tuned` module of Open MPI provides a simple interface for tuning its runtime environment. Several MCA parameters could be used, but the simpler approach involves the creation of a tuning file, which contains any number of rules for the values of the MCA parameters algorithm, fan-in/out value and segment size. Their full MCA name along with the default value are indicated in Table 2.1.

Listing 2.2 gives an example of such a tuning file. It indicates that in the case of an `MPI_Allreduce` operation, with 8 nodes or more and 64 MPI processes or more, two rules shall apply: if the message size is lower than 1024 bytes, algorithm number 7 shall be used,

with a fan-in/out value of 4 and a segment size value of 32. If the message size is greater than 1024 bytes, algorithm number 1 will be used, with a fan-in/out value of 4 and a segment size value of 64. This is a simplified example, more rules for other collectives, node counts or communicator sizes can be added. This file format allows to set any number of rules and conditions for the applications. To use this file as the Open MPI tuning file, the following MCA variables shall be set:

- `coll_tuned_dynamic_rules_filename` set to the path to your configuration file
- `coll_tuned_dynamic_rules_fileformat` set to 1
- `coll_tuned_use_dynamic_rules` set to 1

```
1  # number of rules for collectives
2  # Id of the collective (allreduce)
1  # Number of rules for nodes
8  # if nnodes >= 8
1  # number of rules for comm sizes
64 # comm size >= 64
2  # number of rules for message sizes
0 7 4 32     # size id faninout segsize
1024 1 4 64 # size id faninout segsize
```

Listing 2.2: Example of an Open MPI configuration file

## 2.2 State of the Art and Related Work

Over the years, several tuning methods have been developed, ranging from traditional brute-force approaches to more sophisticated automated and incremental tuning techniques. Each method addresses the challenge of searching a vast optimization space with different trade-offs between time, effort, and tuning precision.

In this Section, we will review the state-of-the-art MPI tuning methods, including brute-force tuning, auto-tuners, incremental tuners, and hybrid techniques, highlighting their approaches, strengths, and limitations. These methods provide valuable tools for improving the performance of MPI applications, enabling them to fully exploit the capabilities of modern supercomputing systems.

### 2.2.1 Brute Force Tuners

Brute force tuning is one of the most straightforward approaches to optimize MPI performance. In this method, all possible configurations of MPI parameters are systematically tested to identify the optimal settings for a given application and system. This includes varying communication protocols, buffer sizes, and process placement policies to find the combination that results in the best performance.

While brute force tuners are exhaustive and can guarantee finding the best configuration, they entail significant drawbacks. The primary limitation is the combinatorial explosion of tuning parameters, which makes the search space grow exponentially as more variables are considered. This results in high computational costs and long runtimes, especially on large-scale systems with numerous nodes.

Our approachs mitigate this issue by either exploring only a fraction of the vast tuning space, or by simplifying the application being run to drastically cut down on the time

required to explore it. Both approaches could even be combined for an even greater time save.

Despite these challenges, brute force tuners remain useful in cases where the number of tunable parameters is small. They also serve as a baseline for comparing the effectiveness of more advanced tuning methods.

**OTPO** The Open Tool for Parameter Optimization [17] (OTPO) is a new framework designed to aid in the optimization of the MCA parameters. OTPO systematically tests a large numbers of combinations of Open MPI's runtime tunable parameters based on a user input file to determine the best set for a given platform. It is the standard tool used by the Open MPI community for tuning MCA parameters.

**mpitune** Similar to OTPO, the `mpitune` [34] utility allows the user to automatically adjust MPI library parameters, such as collective operation algorithms, to the cluster configuration or application. However, it is restricted to Intel MPI.

### 2.2.2 Performance Prediction

To overcome the impossibility of running large scale tunning campaigns, an option is to lower the accuracy of the tuning by predicting the performance of an application instead of executing it at full scale. With enough accuracy, a model can be used to predict the effects of changing the tuning of MPI on the performances of the application, thus allowing for its fine tuning. However, this approach is usually very dependant on the application being tuned, and this technique can only be used for long-running applications.

**FACT** In [67], Zhai *et al.* present a method called FACT, designed to efficiently collect communication traces of large-scale parallel applications. The goal of FACT is to reduce the time and resources needed to obtain communication traces, which are crucial for optimizing and analyzing the performance of parallel applications.

Traditional communication trace collection methods for parallel applications are time-consuming and require significant computational resources. FACT addresses these limitations by enabling trace collection on small-scale systems while maintaining the accuracy of communication traces. It works by slicing the program using static analysis, removing unnecessary computations while preserving all communication-relevant parts. The reduced program slice is then executed to collect communication traces.

Experiments with programs such as the NAS Parallel Benchmarks [45] and Sweep3D [64] showed that FACT can significantly reduce resource consumption (by up to two orders of magnitude) while accurately preserving the spatial and volume communication attributes of the original programs. The traces collected by FACT can be used for process placement optimization, debugging, and communication patterns analysis.

FACT enables a better usage of resources by allowing large-scale applications to be traced using small-scale systems, reducing memory usage and execution time. It is also scalable, providing significant performance improvements, with experiments showing reductions in memory consumption and execution time when collecting communication traces.

While the predictions accurately describe the application, it only reduces the consumption of resources by a constant factor, which will not change with the growth of the parameter space. That is why this solution does not consider influential variables such as

the segment size or the fan-in/out value, as it would exponentially increase the resource consumption. In contrast, the time required by our solution does not scale with the size of the parameter space.

**PHANTOM** Following their previous paper, Zhai *et al.* presented a framework called PHANTOM [66]. It is designed to predict the performance of parallel applications on large-scale machines using only a single node. This approach is particularly valuable when large-scale machines are not yet available, or their entire capacity can't be used for testing.

PHANTOM aims to solve this by using a single node and a deterministic replay to measure sequential computation time accurately. By observing that processes in parallel applications tend to exhibit similar behavior, PHANTOM clusters them into groups and only replays a representative process from each group, which reduces the time and resources needed. It then combines this computation-time acquisition approach with a trace-driven network simulator to predict overall performance.

PHANTOM was validated on applications like ASCI Sweep3D [64], achieving less than 5% error on 1024 processor cores. The approach also demonstrated superior accuracy compared to regression-based models. Thus proving its usefulness for early-stage system design, cross-platform performance prediction, and helping developers optimize their applications before large-scale machines become available.

Similar to FACT, PHANTOM does not consider the segment size not the fan-in/out value. Additionaly, the approach requires a model that is specific to the application, which makes it application-dependant. On the contrary, our solution is truly application-agnostic as the tuning produced is tied to the hardware instead of the executable.

### 2.2.3 Incremental Tuners using Probing

Probing-based tuners build a performance model by selectively testing algorithms during the execution of MPI collective operations. This allows the tuner to adapt to varying hardware and network conditions, which can fluctuate during different runs or on different systems. The tuner incrementaly refines its algorithm selection by updating the model based on observed performance, reducing the likelihood of using inefficient algorithms over time.

This approach offers the advantage of ongoing optimization without requiring separate benchmarking phases, making it more flexible and effective in heterogeneous or variable environments, although the performance on the first executions and the exploration of all possible configurations might be suboptimal.

**OMPICollTune** The OMPICollTune [32] solution proposed by Hunold *et al.* started as an attempt at verifying the validity of an MPI implementation. Some collective communication operations can be implemented using the others, like `MPI_Allreduce` can be implemented using `MPI_Reduce` followed by `MPI_Bcast`. By identifying all such situations, monitoring the performances of both solutions and asserting that the specific function is more efficient than the composite implementation, they were able to validate an MPI implementation. Using the tools developped for this purpose, they also achieved complete MPI probing along with an embedded tool to select the algorithm to use, which can be used to effectively tune MPI from the inside.

Tuning MPI was not the primary goal of this approach, but rather a fortunate side effect. In addition, finding a situation that invalidates an MPI implementation, and thus

a possible optimization, is solely guided by randomness. The solutions presented in this thesis are specifically designed to optimize the MPI implementation in the most efficient way possible.

### 2.2.4 Black Box Optimization

Black-box optimization refers to the optimization of a function of unknown properties, most of the time costly to evaluate and which can only be evaluated a limited number of times. As no hypothesis is made on the optimized function, the only information available to the optimizer is the history of the black-box function, as the list of the inputs and the corresponding outputs. Black-box optimization has shown promising results in different optimization fields such as energy consumption [43], I/O accelerators [51] or high-end storage bays [16], and could be adapted to tune MCA parameters. This approach is further explored in Chapter 3.

### 2.2.5 Machine Learning

Machine Learning (ML) is a field of study in artificial intelligence concerned with the development and study of statistical algorithms that can learn from data and generalize to unseen data and thus perform tasks without explicit instructions. ML finds application in many fields, including natural language processing, computer vision, speech recognition, email filtering, etc [31].

With its wide applicability and performance, ML is a interesting candidate for MPI tuning. Its has already been used successfully for the optimization and tuning of NUMA systems [54]. The method shows an average 1.68x performance improvement over a locality-optimized NUMA baseline with all prefetchers enabled, and achieves 95% of the optimal performance while reducing the need to evaluate all configurations, saving time and resources.

## 2.3 Problematic

In conclusion, we have seen that MPI tuning is an active research field, with multiple angles from which to attack the central problem: How can MPI implementations be fine-tuned efficiently to maximize performance across various applications, architectures and runtime systems? Traditional methods of manually tuning MPI library parameters are insufficient due to the complexity and scale of modern HPC systems. We will investigate how automated techniques can streamline and enhance the tuning process, ensuring optimal communication performance without requiring deep, system-specific knowledge.

# Chapter 3

# Blackbox Optimization

## Contents


**3.1 Methodology of BBO** . . . . . . . . . . . . . . . . . . . . . . . **32**
**3.2 State of the Art** . . . . . . . . . . . . . . . . . . . . . . . . . **33**
**3.3 Formalization of the Problem** . . . . . . . . . . . . . . . . . . . . **34**
3.3.1 The Parameter Space . . . . . . . . . . . . . . . . . . . . . . . . . 34
3.3.2 Choice of the Initial Sampling Point . . . . . . . . . . . . . . . . . 34
3.3.3 Choice of the Next Sampling Point . . . . . . . . . . . . . . . . . . 35
3.3.4 Choice of the Stopping Criterion . . . . . . . . . . . . . . . . . . . 36
**3.4 Agnosticism of the blackbox Approach** . . . . . . . . . . . . . . . **36**
**3.5 Pre-existing Software used to Produce the Tuning File** . . . . **37**
3.5.1 ACCO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
3.5.2 ShaMAN . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
3.5.3 Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
**3.6 ACCO and ShaMAN Integration to Optimize an MPI Runtime 38**
**3.7 Validity of the Bayesian Optimization over Brute Force** . . . **39**
3.7.1 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.7.2 Experiment Plan . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.7.3 Evaluation Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
**3.8 Comparisons between Brute Force and Baysian Optimizations 42**
3.8.1 Execution Time Comparison between Bayesian Optimization and Brute Force . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.8.2 Tuning Time Comparison . . . . . . . . . . . . . . . . . . . . . . . 45
3.8.3 Scalability Study for a Single Collective Operation . . . . . . . . 46
**3.9 Conclusion** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **48**

This chapter presents the use of a classical technique to tune an MPI application. As described in Chapter 2, the number of configurations that the tuning profile can adopt is exponential, and finding the optimal one is a time consuming process, especially if the application itself takes a long time to execute.

In this chapter, we applied the technique called "Blackbox Optimization" (BBO) to explore a fraction of the parameter space, denoted $\Theta$, finding a compromise between the quality of the solution and the time to reach it. One of the main advantage of this technique over the others discussed in Section 2.2 is its agnocticism regarding the optimized application. Indeed, said application is treated as a *blackbox*, meaning that the only assumption made is that it can be evaluated at any point of $\Theta$, and thus the technique can be applied to optimize any MPI application.

## 3.1 Methodology of BBO

As stated in Section 2.2.4, exhaustively sampling the parameter space is not a viable option due to its size. To overcome this issue, the idea is to use a strategy, or heuristic, that decides which point of the parameter space is the most interesting to evaluate. Applying repeatedly this heuristic allows the algorithm to quickly converge toward a local optimum for the tuning file.

Black-box optimization refers to optimizing a function with unknown properties that are often costly to evaluate, resulting in a limited number of possible evaluations. These methods are promising for tuning various systems, including computer systems. When applied to computer system tuning, the approach treats the system as a blackbox, analyzing the relationship between input and output parameters as described in Figure 3.1. In this Figure, we denote $\theta_i$ the points of the parameter space that were evaluated. The set of all points $(\theta_i)_{1 \leq i \leq n}$ is the parameter space, noted $\Theta$. As stated in Section 3.3.1, there is a one-to-one relationship between the points of $\Theta$ and the tuning files, thus evaluating all $\theta_i$ is equivalent to testing all possible tuning files. The function that we aim to optimize is noted $f$, in this case it is the execution time of a benchmark application, discussed in Section 3.5, that we want to minimize. *BLACKBOX* represents the application being executed and optimized, along with every environment variable that may impact $f$. In our case the specific collective communication operation, the MPI implementation used and the execution context, which is composed of the hardware used and its current state. Since the *blackbox* does not require previous knowledge on the relationship between the parametrization of the collectives and the performance of the application, it is easier to implement than an analytical model. Finally, the *OPTIMIZER* is the mechanism that guides the exploration of the parameter space toward the most interesting points to evaluate using Bayesian Optimization. It is further detailled in Section 3.3.3.

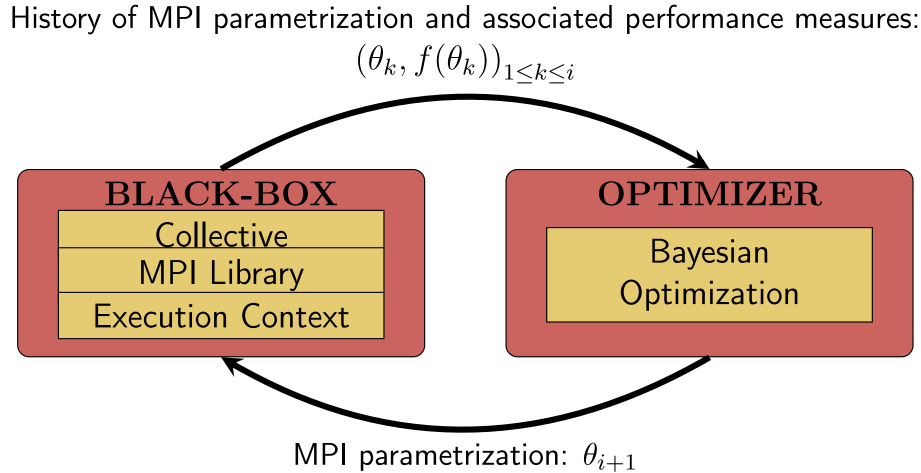



Figure 3.1: Schematic representation of the optimization loop

## 3.2 State of the Art

Auto-tuning using blackbox optimization has been used in several domains in the last years. It has yielded good results in very diverse situations and has been particularly helpful in computer science for finding optimal configurations of various software and hardware systems. In [23], the authors compare two derivative-free methods (Bounded Optimization by Quadratic Approximation method and Constrained Optimization by Linear Approximation method) to find the optimal configuration of the Hadoop framework. Bayesian Optimization has also been successfully used for finding the optimal configuration of Apache Storm computation system in [35]. A more general tuning framework, called BOAT, relying on structured Bayesian Optimization is described in [21]. Within the HPC community, auto-tuning has gained a lot of attention for tuning particular HPC applications and improve their portability across architectures [10]. In [55] and [39], a comparison of several random-based heuristic searches (simulated annealing, genetic algorithms, *etc.*) are provided when used for code auto-tuning while [13] yields good results with surrogate modeling using boosted regression trees. The energy consumption of the HPC machines has also benefited from Bayesian Optimization, as the "kukai" system [44] made the Green500 list after using an auto-tuner based on a combination of *Gaussian Process* regression and the *Expected Improvement* acquisition function. Reinforcement learning, which is another guided search method, has also been successfully used as an auto-tuner to optimize the performance of the Lustre filesystem in data center storage systems [2]. Also, an optimal parametrization for the several layers of the HDF5 library was found using genetic algorithms in [1]. An extension of this auto-tuner which selects the best parameters according to the I/O pattern is described in [11]. Prior studies regarding MPI application tuning have shown that the best communication algorithm for a collective communication operation highly depends on the message to be transferred [5, 58, 7]. Tuning an MPI application mostly relies on finding a suitable set of MCA parameters, described in Section 2.1, which is not an easy task considering the large number of available parameters.

## 3.3 Formalization of the Problem

Let $S$ be the MPI library to optimize and $\theta_i$ its parametrization, which belongs to a discrete subset of the possible parametrizations of the MPI libraries $\Theta = \{\theta_i\}_{i \in \mathbb{N}}$. Let $\mathcal{A}$ be a program measuring the performance of the collective communication operation chosen to be optimized within the library $S$. Let $\mathcal{E}$ be the execution context for which we want to optimize the benchmark. Let $F_s$ be the performance function associated with the application, the execution context and the MPI library:

$$F_S : (\mathcal{A},\ \mathcal{E}, \Theta) \longrightarrow \mathbb{R}$$

The optimization problem that we are trying to solve is:

$$\min_{\theta_i \in \Theta} F_{S,\mathcal{A},\mathcal{E}}(\theta_i)$$

In our situation, $\Theta$ is the Cartesian product of all possible values of parameters listed in the following Section 3.3.1, and $\mathcal{E}$ is the platform used for the experiments, presented in Section 3.7.1. $\mathcal{A}$ would be one of the OSU Micro Benchmarks (OMB), a benchmark suite developped by the Ohio State University (OSU), specifically targetting MPI collective communication performances. These benchmarks are furter detailed in Annexe A while our usage of them is described in Section 3.5.3.

### 3.3.1 The Parameter Space

The parameter space, denoted $\Theta$, is the cartesian product of all possible values of the MCA parameters considered and all possible combinations of number of nodes, number of MPI processes and message sizes that the application can be called with. As mentioned in Chapter 2, the combinatorial explosion of the size of this space will prohibit an exhaustive exploration. A bijection can be constructed between $\Theta$ and the set of all tuning files described in Section 2.1.4, as we can view $\theta_i$ as an set of tuples $(N, np, size, param_k, value_k)_{1 \leq k \leq n}$ where $k$ is the number of MCA parameters considered. In our case, $k$ is equal to 3, the collective communication algorithm identifier, the segment size and the fan-in/out value. These parameters are a subset of the ones available in the `tuned` component, described in Section 2.1.2. The segment size is the number of bytes of the messages sent by the MPI implementation on the underlying hardware, chopping messages whose payload is bigger than the segment size into chunks whose size is at most the segment size. Fan-in/out is a positive value that is used if the considered algorithm is tree-based, in which case it represents the target number of children for each node of the tree. For non tree based algorithms, this value is ignored, which renders the optimization of this parameter irrelevant and problematic. A tuning file can then be constructed for each $\theta_i$, according to the rules specified in Section 2.1.4.

### 3.3.2 Choice of the Initial Sampling Point

The first step of any blackbox optimization algorithm is the selection of the initial parameters to start the optimization process. An acceptable initialization starting plan must respect at least two properties [37, 53, 29]: the space constraints property and the non-collapsible property. The space constraints are shaped by the possible values that can be taken by the parameters. The non-collapsible property specifies that no parametrization can have the same value on any dimension. It ensures that if an axis of the parameters

space is removed, then no two points would have the same coordinates. This is especially important as we have no insight on the individual effect of each parameter, and we want to avoid evaluating points with similar performance as much as possible. For example, the dimension corresponding to the fan-in/out value can be flat, as this parameter is ignored in certain situations. In these cases, the property ensures that two parametrizations still can be distinguished. Latin Hypercube Sampling [37] is a usual choice as an initialization strategy, because of its simplicity and its efficiency [42]. We shall rely only on this method for initialization.

### 3.3.3 Choice of the Next Sampling Point

Besides the initialization plan, Bayesian Optimization requires two inputs to be instantiated: the acquisition function and the probabilistic model used to represent the performance function.

**The acquisition function** An acquisition function indicates for each configuration its potential performance improvement by being evaluated next, given the input of the probabilistic model. It should offer a trade-off between *exploration* of parameter zones where the model is uncertain (*i.e.,* zones with a high variance) and the *exploitation* of already promising well-explored zones (*i.e.,* zones where the mean is low). One of the most common acquisition methods is the *Expected Improvement* (EI), which computes the expected improvement from switching from $f^*$, the best configuration found so far:

$$I(\theta) = \begin{cases} f^* - f(\theta) & f(\theta) < f^* \\ 0 & f(\theta) \geq f^* \end{cases}$$

$$EI(\theta) = \mathbb{E}(I(\theta))$$

$I$ represents the relative improvement and $EI$ denotes the expected improvement, computed as the expectancy of the relative improvement. As EI is one of the most popular algorithms and has been proven to be an efficient acquisition function to solve a wide range of problems, we will focus solely on this acquisition function.

**The probabilistic model** A suitable probabilistic model should be able to give an estimation of the mean and the standard deviation for each possible parametrization. The most popular choice is Gaussian Processes [49] which generate distributions over functions used for Bayesian non-parametric regression. Other methods, such as Parzen trees [62] and Random Forests [12] were also implemented, our tests showed Gaussian Processes to be the most effective, so we focus on this model. The results of the experimentations are presented in Section 3.8. A Gaussian Process is fully characterized by a mean function $\mu$ at each parametrization, as well as a covariance function $\Sigma$ between all of the parametrizations of the parameters grid. Mean and variance predictions at parametrization $\theta_i$ are obtained as:

$$\mu(\theta_i) = k_* K^{-1} y$$

$$\sigma^2(\theta_i) = \Sigma(\theta_i, \theta_i) - k_*^T K^{-1} k_*$$

where $k_*$ denotes the vector of covariances between all previous observations, K is the covariance matrix of all previously evaluated configurations and $y$ are the observed performance values. The selected covariance function $\Sigma$ has a strong impact on the

performance of the model [33], and we opted for the common choice of a radial-basis function kernel (with $d$ the Euclidean distance): $\Sigma(\theta_i, \theta_j) = exp(d(\theta_i, \theta_j))$.

### 3.3.4 Choice of the Stopping Criterion

The stopping criterion dictates when we estimate that the current best parametrization is satisfaying enough. The optimization process runs until either the maximum number of steps is exceeded or a stopping criterion evaluates to `true`. Multiple criteria can be used, for example setting the number of steps to the size of $\Theta$ and the criterion to always return `false` will be equivalent of reverting back to an exhaustive search. For our use-case, we chose a maximum number of iterations set to 150. Additionally, we decided to stop the optimization process if the improvement over the last 15 iteration is less than 1%. Experimentations shows that the arbitrarily chosen value 150 is never reached, so its only used as a time saver to avoid exploring too much space in the case the convergence is slow.

## 3.4 Agnosticism of the blackbox Approach

Wilkins *et al.* proposed an approach called FACT [8] (Fast Communication Trace Collection) that focuses on minimizing the amount of data fed to the auto-tuners and that shows remarkable resilience to the decrease of the number of runs of the application. However, it does not consider other influential parameters such as the segment size nor fan-in/out and the employed performance model (*RandomForestRegressor*) deteriorates when the number of dimensions of the parameters space increases [12]. Last, FACT's approach is not fully application-agnostic: "*Considering training data must be recollected as frequently as every job allocation, FACT-based collective autotuning is only practical for longer-running jobs.*"

The successor to FACT, ACCLAiM [4], introduces an auto-tuning machine learning-based method for optimizing collective communication operations. This time, the method is effectively applied to real applications, but their focus is primarily on the type of collective communication operation, without considering other crucial parameters like the segment size or the fan-in and fan-out values. However, ACCLAiM's tuning process takes place at runtime and implies a model training for every job run on the target system, which may not be practical when dealing with a vast optimization space. In contrast, our approach is truly application-agnostic and optimizes independently each collective communication operation for the target system once and for all. An application is then able to select transparently the best algorithm when it calls a collective communication routine. The tuning in ACCLAiM is carried out on a per-application basis, necessitating adjustments whenever a new application is introduced or when there are changes in the communication pattern of an existing one. This stands in contrast to our approach, which requires a single tuning process for the entire cluster, applicable universally.

## 3.5 Pre-existing Software used to Produce the Tuning File

### 3.5.1 ACCO

ATOS Collective Communications Optimizer (ACCO) [24], is a specialized tool designed to optimize the collective communication operations in MPI. It leverages a brute-force exploration approach to evaluate all possible communication strategies for these collective operations, systematically identifying the most effective method for the specific architecture and communication patterns in use.

This emphasis on exhaustive exploration is especially crucial for the module `han`, described in Section 2.1.2, which is now the default implementation used in the `coll` in Open MPI version 5.0 and higher. Indeed, the hierarchical nature of these algorithms introduces a wide range of potential communication paths and strategies, which can differ significantly in performance. By rigorously testing every possible configuration, ACCO ensures that the optimal communication strategy is selected, leading to substantial performance gains. This makes ACCO an invaluable tool for HPC practitioners looking to fine-tune the performance of collective operations, particularly in environments that use advanced hierarchical communication models.

ACCO automates the submission and retrieval of benchmarks, parses their outputs to finally produce a tuning file, following the processus described in Section 2.1.4. It internally uses the OSU Micro Benchmarks suite, detailled in Section 3.5.3, to evaluate the performances of the underlying architecture.

### 3.5.2 ShaMAN

Smart HPC Application MANager (ShaMAN) [50, 52] is a framework to perform auto-tuning of configurable component running on HPC distributed systems. It performs the auto-tuning loop by parametrizing the component, submitting the job through the Slurm workload manager, and getting the corresponding execution time. Using the combination of the history (parametrization and execution time), the framework then uses blackbox optimization to select the next most appropriate parametrization, up until the number of allocated runs is over or the stopping criterion is reached.

The framework was originally designed to optimize I/O operations, but due to its genericity, it was extracted to be a standalone tool to perform optimizations, regardless of the underlying system being tuned.

ShaMAN comes out of the box with several optimization heuristics, several noise reduction strategies and pruning strategies. Most notably, alongside *genetic algorithms* and *simulated annealing*, the optimization heuristic *surrogate modeling* is implemented, and use the Bayesian Optimization detailled in Section 3.3.3. The other criteria and strategies presented in Section 3.1 are also implemented in ShaMAN.

### 3.5.3 Benchmarks

The OSU Micro Benchmarks (OMB) [47] are a widely recognized suite of performance tests developed by the Ohio State University (OSU) specifically for evaluating the performance of MPI implementations. These benchmarks are crucial in the field of HPC as they provide detailed measurements of various aspects of MPI communication, including point-to-point latency, bandwidth, and message rates. The OMB suite covers a broad spectrum of MPI

functionalities, such as collective operations or one-sided communication, making it an indispensable tool for assessing and optimizing the performance of parallel applications on various HPC architectures. By delivering fine-grained insights about the communication patterns and overheads, the OSU Micro Benchmarks help researchers and developers to fine-tune their systems and achieve optimal performance in distributed computing environments. However, they suffer from potential timing issues [22] for their use of unsynchronized clocks. This issue only affects precise timing, and was not considered here due to the high number of repetitions that were done for each benchmark (1000 repetitions for sizes below $2^{17}$ bytes, and 100 above). For each of the tuned collective and each tested size, we use the corresponding benchmark in the suite. To ensure stability and reduce the noise when collecting execution times, the OSU benchmark was parameterized to perform 200 warmup runs before performing the actual test.

## 3.6 ACCO and ShaMAN Integration to Optimize an MPI Runtime

The ShaMAN software, generic by design, is meant to be customized to solve optimizations problems, such as the execution time of MPI applications. Combined with the capabilities of ACCO, they would create a perfect framework to handle the problem at hand. The goal is to extract the core component of the two pieces of software and assemble them accordingly to facilitate the tuning process.

Figure 3.2 presents how the differents tools used to produce the tuning file were integrated together. The system is divided into three main components: the two pieces of software and the underlying cluster. ACCO handles the orchestration of OSU Micro Benchmarks, parsing the output results and providing the results to ShaMAN. It utilizes a Tuner module that generates configurations parameters, which are then saved in a tuning file, represented here by a red box. This tuning file is subsequently passed to the MPI runtime on the cluster, where a job manager, in our case SLURM [65], oversees its execution across multiple nodes. ShaMAN, on the other hand, integrates a regression model to predict the performance of different configurations. It calculates an Expected Improvement metric to guide the search for optimal parameters and includes the stopping criterion to determine when the optimization process should halt. The process is cyclical, with ShaMAN providing new configurations back to ACCO, which in turn produces a new tuning file for the MPI runtime until an optimal configuration is found. This coordinated effort allows for efficient tuning and execution of MPI programs across a parallel system, ensuring that the best possible performance is achieved.

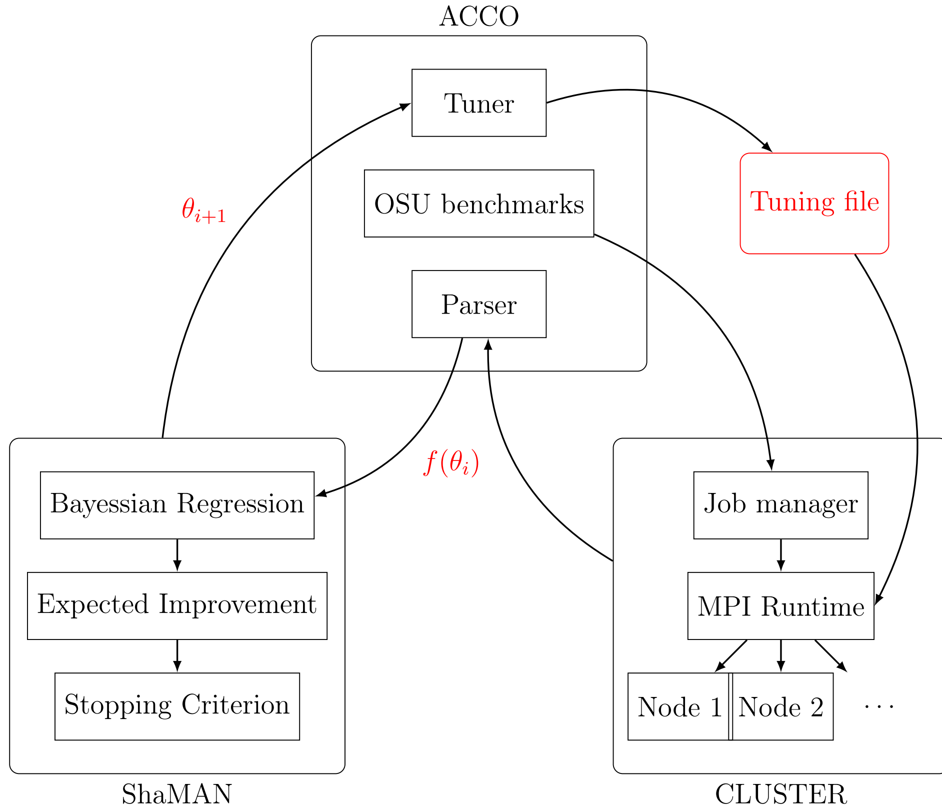



Figure 3.2: Schematic Representation of the Workflow

## 3.7 Validity of the Bayesian Optimization over Brute Force

The goal of the validation plan is to assess the performance of using Bayesian Optimization for tuning a set of benchmarks, both in terms of performance compared to the default parametrization and time to solution compared to an exhaustive search in the parametric space.

### 3.7.1 Experimental Setup

For our experimentations, we used the hardware described in Table 3.1. Both platforms feature different number of nodes and number of cores per processor, which lead to different number of experimentations on each. The `Pise` platform has 32 nodes with two 24-core AMD CPUs. On the other hand, the `Bora` cluster has only 24 nodes with two 18-core Intel CPUS. To limit the number of experimentations while fully utilizing both platforms, we decided to use nodes in increments of 6. Only the experimentations with 30 nodes could not be reproduced on the `Bora` cluster.

### 3.7.2 Experiment Plan

The evaluation of the performance of Bayesian Optimization is carried out by tuning four of the most commonly used collectives communication operations:

| Platform | `Pise` | `Bora` |
|---|---|---|
| **Node count** | 32 | 24 |
| **Open MPI version** | 4.0.4 | 4.1.5 |
| **CPU** | 2 x AMD Rome 24 cores (AMD EPYC 7402) | 2 x Intel Skylake 18 cores (Xeon Gold 6140) |
| **Interconnect** | Mellanox ConnectX-6 HDR200 (pcie4) | OmniPath 100GBit/s |

Table 3.1: Hardware description

- **Broadcast:** Broadcast is one of the collectives where one process sends the same data to all processes in a communicator. A classic use of broadcast is to send out user input to a parallel program, send out configuration parameters to all processes or simply send out the result of a local computation to multiple processes.
- **Gather**: The gather collective takes data from several processes and gathers them to one single root process. This operation is highly useful to many parallel algorithms, such as parallel sorting and searching.
- **Reduce**: The reduce collective operation is similar to the gather operation, with the addition of a user-defined operation to apply on the collection of data. It takes an array of input elements from each process and returns an array of output elements to the root process. These output elements contain the result of a user-defined operation such as the sum or the maximum, performed on the collection of data, which makes it very useful for parallel applications.
- **Allreduce**: Many parallel applications will require accessing the reduced results across all processes rather than only the root process, and the goal of the allreduce operation is to reduce the values and distribute the results to all processes. While being equivalent to a reduce followed by a broadcast, this explicit collective operation enables libraries to implement more efficient algorithms.

The `tuned` module, presented in Section 2.1.2, allows us to modify, along with the segment size and the fan-in/out values, the underlying algorithm used by the collective communication operation. The different algorithms and their names are listed in Table 3.2.

The sizes used for the messages sent through the MPI collective communication operation range from 4B to 1MB, with a multiplicative step of 2. Two hardware configurations are selected, to emulate two of the most common process placements policies encountered in HPC applications:

- *A single MPI process per node*: we run a single MPI process per node, for a total of 12 MPI processes. This type of setting is typical of hybrid applications relying on MPI for inter-node communications and on another solution for their intra-node communications, for instance OpenMP.
- *A single MPI process per core*: we used two platforms, described in Section 3.7.1. On the `Bora` machine, we run 36 MPI processes per node, for a total of 432 MPI processes, while on the `Pise` these numbers are 48 and 576 respectively. This type

| Collectives | Algorithm name and number |
| --- | --- |
| Broadcast | 1 - Basic linear<br>2 - Chain<br>3 - Pipeline<br>4 - Split binary tree<br>5 - Binary tree<br>6 - Binomial tree<br>7 - Knomial tree<br>8 - Scatter allgather<br>9 - Scatter allgather ring |
| Gather | 1 - Basic linear<br>2 - Binomial<br>3 - Linear with synchronization |
| Reduce | 1 - Linear<br>2 - Chain<br>3 - Pipeline<br>4 - Binary<br>5 - Binomial<br>6 - In order binary<br>7 - Rabenseifner |
| Allreduce | 1 - Basic linear<br>2 - No overlapping (tuned reduce + tuned broadcast)<br>3 - Recursive doubling<br>4 - Ring<br>5 - Segmented ring<br>6 - Rabenseifner |

Table 3.2: Collectives and their corresponding algorithms

of setting is typical of pure, MPI-only applications which rely on the MPI library for all their communications (inter-node and intra-node alike).

This results in a total of 160 optimization experiments (4 collectives, 20 sizes and 2 different topologies). The performance metric for tuning is the time elapsed by the benchmark for the selected message size given to the operation.

### 3.7.3 Evaluation Metrics

To evaluate the performance of our suggested method, the reference execution time is first computed which means running the different configurations of the benchmarks with the default parametrization. This default parametrization is run one hundred times to account for possible noise in the collected execution time. An exhaustive sampling of the parametric space is then performed, in order to get the corresponding execution time at each possible parametrization, and selects the parametrization with the minimal execution time as the optimal one, which acts as the baseline and is also run one hundred times for noise mitigation.

The tuning of the system is also performed using Bayesian Optimization, as described in Section 3.3.3. The best parametrization found by the optimization process is considered to be the best parametrization found by Bayesian Optimization and is also run one hundred times to account for noise. We are interested in comparing the trade-off between the tuning time and distance to optimal between both methods.

The performance of Bayesian Optimization in terms of elapsed time and number of benchmark runs required to reach the optimum is also discussed and compared to the number of iterations required by brute force. We also provide insights on the impact of the number of nodes on the elapsed time required to reach the optimum, in the case of Bayesian Optimization as well as for exhaustive search, for the reduce collective operation.

## 3.8 Comparisons between Brute Force and Baysian Optimizations

The execution time gain of using the best parametrization found by the Bayesian Optimization compared to the default one is represented in Figure 3.3 for `Pise`. Over all experiments, we find an average improvement of 48.4% (52.8% in median), using the best parametrization found with Bayesian Optimization. We find an average improvement of 38.42% (29% in median) for experiments with a single MPI process per node and of 58.9% (65.3% in median) when using a single MPI process per core, highlighting the efficiency of tuning the Open MPI parametrization instead of simply relying on the default one.

The time gain brought by Bayesian Optimization varies depending on the tuned collective communication operation, with some where the default parametrization is more adapted than others. It is the case for the *allreduce* collective when running one MPI process per node, where the Bayesian parametrization provides a median improvement of 0.9% (18% on average). Other collective operations have a default parametrization that is not adapted at all. It is for example the case of the *gather* collective with one MPI process per core, where we see an improvement of 91% in median and on average. The improvement of the default parametrization is strongly dependent on each evaluated parameter (message size, number of processes per node or type of operation) and is difficult

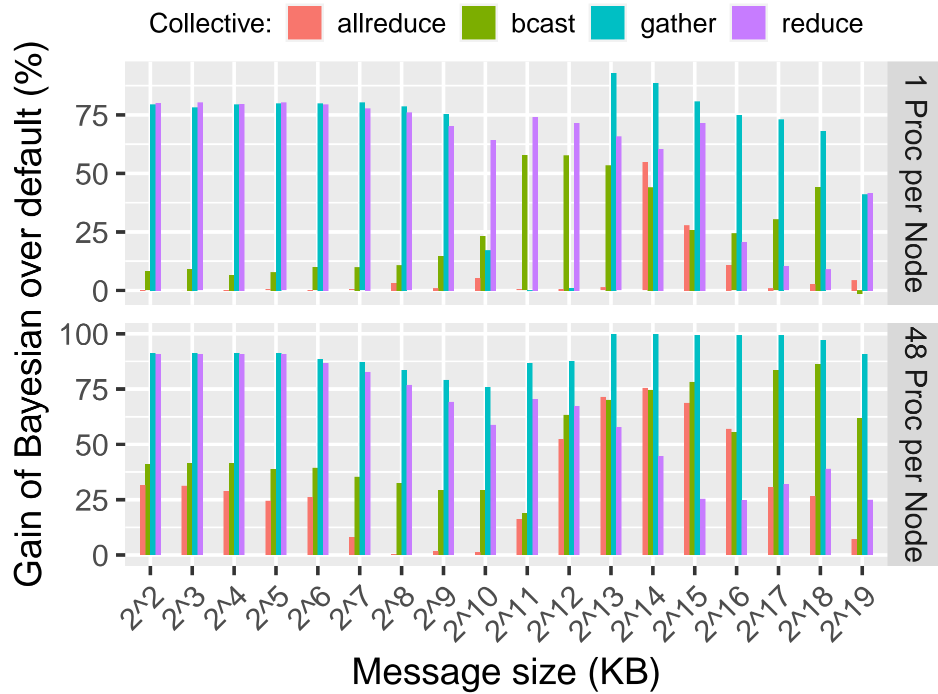



Figure 3.3: Execution time gain of using the solution found by Bayesian Optimization compared to the default parametrization (`Pise` machine)

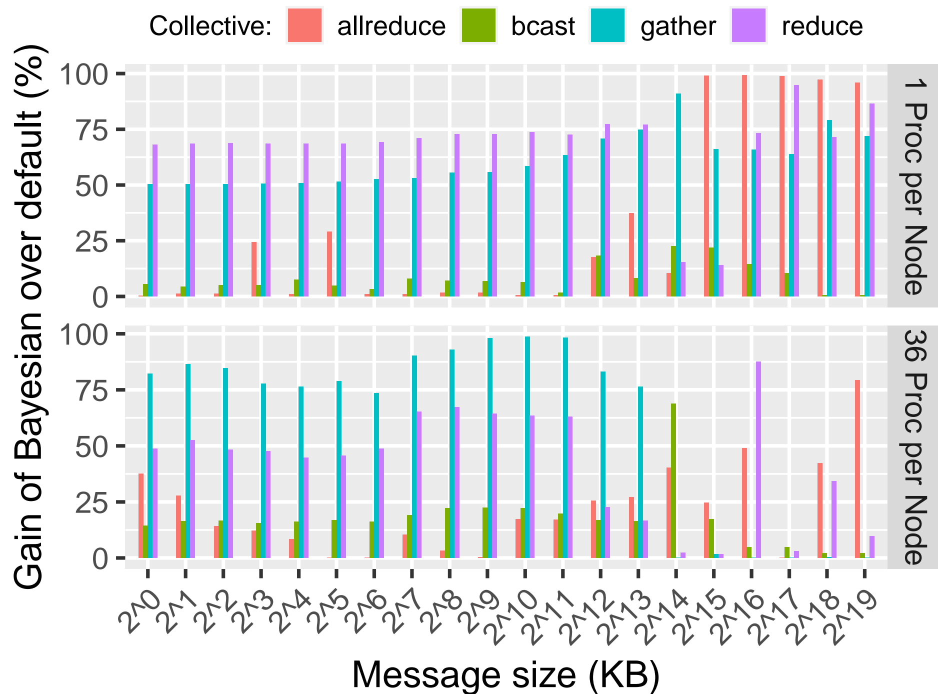



Figure 3.4: Execution time gain of using the solution found by Bayesian Optimization compared to the default parametrization (`Bora` machine)

to predict. This highlights the importance of tuning each configuration to get the best performance, and the need for an efficient tuning method.

For the `Bora` case, we also see a huge gain brought by the best parameterization, as shown in Figure 3.4, with an average gain of 39.3%. For the single process per node case, results are very similar to that of `Pise`, the gain is 45.0% using the best parameterization. We also see how machine-dependent is the tuning: the gain of allreduce is closer to 100% for large message on the `Bora` machine while the default parameterization was almost optimal on the `Pise` machine. For one process per core (36 processes per node), we observe that for small message sizes, the gain can be very large for gather but relatively small when dealing with large messages. Nevertheless, in this case, the average gain is 33.6%.

### 3.8.1 Execution Time Comparison between Bayesian Optimization and Brute Force

The median difference in elapsed time, along with the noise measurement, between the best parametrization found by Bayesian Optimization and the optimal parametrization found by exhaustive search is represented in Table 3.3. Over all optimization experiments, the average distance between the optimum and the result returned by Bayesian Optimization is of 5.71 microseconds (0.04 in median) for an average noise of respectively 2.03 microseconds in mean and 0.05 microseconds in median. This means that in median, the difference between using the best parametrization of our tuner compared to the true best parametrization is imperceptible from the noise. It indicates that the tunings produced by both methods are nearly indistinguishable in terms of performance, thus strongly recommanding our method as it does not degrade the result while being significantly faster. When looking at the relative difference between the optimum and the results from Bayesian Optimization, we find an average distance of 6% (0.7% in median) between the two.

| **Collective** | **# of MPI proc.** | **ΔT ($\mu$s)** | **Noise ($\mu$s)** |
|---|---|---|---|
| **allreduce** | 12 | 0.04 | 0.43 |
| | 576 | 3.80 | 1.05 |
| **bcast** | 12 | 0.26 | 0.32 |
| | 576 | 0.18 | 0.32 |
| **gather** | 12 | 0.01 | 0.04 |
| | 576 | 0.00 | 0.02 |
| **reduce** | 12 | 0.00 | 0.06 |
| | 576 | 0.00 | 0.03 |

Table 3.3: Median difference in execution time and noise between best parametrization found by Bayesian Optimization and optimal parametrization (`Pise` machine)

When looking at the different collective communication operations and hardware platforms, we find an average distance of 6% and the difference between the two optimal parametrizations to be inferior to the measured noise, for all collective operations except in the case of allreduce with 576 MPI processes. When looking at each optimization problem separately, we find that for 105 optimization problems out of 160, the distance of the performance returned by Bayesian Optimization to the optimum is below the measured noise of the system. For the problems where the difference between the results returned by the tuner and the optimum cannot be explained by noise, we find a quite low average difference of 1.90 microseconds (0.18 in median). The noise difference between collectives

is explained by multiple factors. Gather and reduce show low noise due to their simple communication pattern (all-to-one). On the opposite, the allreduce collective involves much more intertwined messages, which explains its higher noise and noise sensitivity. Broadcast's higher noise is explained by the best performing algorithm found (k-nomial tree) which, according to Subramoni *et al.* [6], introduces some noise due the imbalanced communication pattern. While the tuner has not been able to find the optimum in these cases, we find that the difference in performance is negligible for applications running in production and the gain compared to the default parametrization is enough to advocate for the benefits of Bayesian Optimization.

### 3.8.2 Tuning Time Comparison

The elapsed time required to reach the optimum for each of the collective communication operations and hardware configurations is reported in Tables 3.4 and 3.5. With a time gain of more than 85% for each collective operation, we see the difference coming from using guided search heuristics instead of testing every parametrization with exhaustive sampling. The time required to run all the 1296 optimization experiments ranges from a total of 8048 minutes (5 days and 14 hours) using brute force to 355 minutes (approximately 6 hours) using Bayesian Optimization, resulting in a total speed-up of 95%. The speed-up is relatively uniform across each collective communication operation and each target platform.

| Coll. | # proc | Brute force | Bayesian Opt. | Gain (%) |
|---|---|---|---|---|
| **Allreduce** | 12 | 53 | 5 | 91.40 |
| | 576 | 453 | 47 | 89.66 |
| **Bcast** | 12 | 745 | 24 | 96.82 |
| | 576 | 5098 | 134 | 97.39 |
| **Gather** | 12 | 24 | 4 | 85.42 |
| | 576 | 1041 | 78 | 92.56 |
| **Reduce** | 12 | 88 | 6 | 94.01 |
| | 576 | 551 | 62 | 88.82 |

Table 3.4: Time to solution for each heuristic and each collective, rounded up to the nearest minute (`Pise` machine)

| Coll. | # proc | Brute force | Bayesian Opt. | Gain (%) |
|---|---|---|---|---|
| **Allreduce** | 12 | 65 | 7 | 89.23 |
| | 432 | 515 | 49 | 90.49 |
| **Bcast** | 12 | 701 | 36 | 94.86 |
| | 432 | 4509 | 87 | 98.07 |
| **Gather** | 12 | 27 | 5 | 81.48 |
| | 432 | 930 | 83 | 91.08 |
| **Reduce** | 12 | 102 | 12 | 88.24 |
| | 432 | 585 | 46 | 92.13 |

Table 3.5: Time to solution for each heuristic and each collective, rounded up to the nearest minute (`Bora` machine)

| **Collective** | **Exhaustive search** | **Bayesian Optimization** | |
|---|---|---|---|
| | | Pise | Bora |
| **Allreduce** | 1400 | 29.0 | 30.5 |
| **Bcast** | 2000 | 30.0 | 31.0 |
| **Gather** | 800 | 26.0 | 28.0 |
| **Reduce** | 1600 | 29.5 | 31.0 |

Table 3.6: Median number of iterations performed by Bayesian Optimization compared to exhaustive search

The median number of iterations per collective operation is reported in Table 3.6. For Bayesian Optimization, the number of iterations is stable across each collective operation and each parametrization (mean number of approximately 30), but the size of the tuned message has an impact on the number of iterations, as shown in Figure 3.5(a) and Figure 3.5(b), respectively, for the Pise and Bora machines. Moreover, we see that the number of steps (*i.e.,* the tuning time) does not depend on the machine but on the number of MPI processes involved. The 150 steps limit mentionned in Section 3.3.4 has been chosen arbitrarily to set an upper bound on the number of iterations. The table shows that in practice, the process converges before that value is reached.

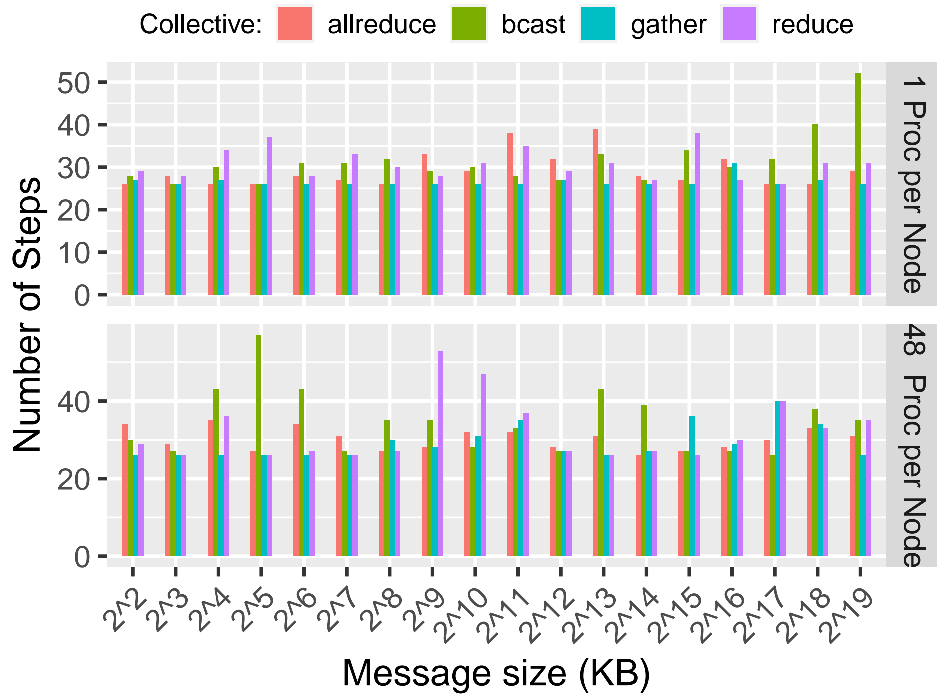
The key point when comparing Bayesian Optimization to brute force that we want to emphasize is the trade-off that we are introducing with the Bayesian Optimization between tuning time and accuracy of the solution. Since Bayesian Optimization does not explore all the parameter space, it may not be perfectly accurate but is much faster than a brute force method. Overall, we are speeding-up the tuning process by 95% while maintaining an accuracy of the solution that is 6% away from to the optimal solution in average (0.7% in median).

### 3.8.3 Scalability Study for a Single Collective Operation

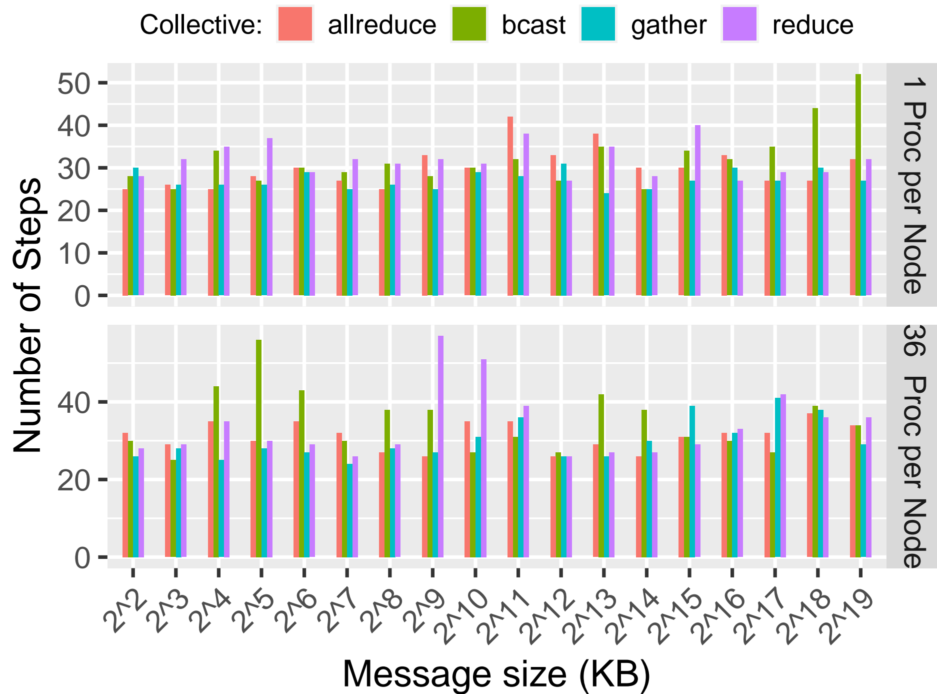
For the reduce collective operation and a single MPI process per node, the tuning time for different numbers of nodes is shown in Table 3.7. We see that the number of nodes has a non-linear impact on the convergence of the algorithms. The elapsed time for tuning using Bayesian Optimization is especially stable across the number of nodes because of the early detection of non-promising tuning, but also across the different architectures. The similarity between the two behaviors on very different architectures gives us confidence that our suggested solution is not only application-agnostic but also architecture-agnostic and shall adapt well on large-scale production HPC clusters.

| # **nodes** | **Brute force** | **Bayesian Optimization** | |
|---|---|---|---|
| | | Pise | Bora |
| 6 | 160.52 | 4.46 | 5.71 |
| 12 | 185.32 | 5.98 | 7.02 |
| 18 | 209.19 | 5.81 | 5.41 |
| 24 | 218.41 | 6.39 | 5.04 |

Table 3.7: Tuning time (in minutes) vs. number of nodes (for the Reduce collective operation)




(a) `Pise` machine




(b) `Bora` machine

Figure 3.5: Number of benchmark runs before reaching stop criterion

## 3.9 Conclusion

In this Chapter, we introduced the use of Bayesian Optimization to find the optimal parametrization of Open MPI collective communication operations within its component called `tuned`. We highlighted the importance of tuning every configuration and the need for an efficient tuning method which we proposed using a blackbox technique based on Bayesian Optimization. It induces a trade-off between the tuning time and accuracy of the solution that favors Bayesian Optimization over a brute force method.

We applied this method to optimize four MPI collective communication operations across two different hardware topologies and 20 message sizes. Our results show that Bayesian Optimization yields solutions that are, on average, within 6% of the optimal values (0.7% in median) identified through exhaustive brute-force testing, while achieving a 95% reduction in tuning time. This leads to an average performance improvement of 48.4% (52.8% in median) in collective operations compared to the default Open MPI MCA parameters.

Our scalability analysis demonstrates that the proposed tuner scales effectively, making this technique suitable for tuning parameters in large-scale HPC environments. As a result of this work, the ShaMAN optimizer module has been integrated as the primary method for exploring the parameter space in the ACCO software.

# Chapter 4

# Skeletonization

## Contents


**4.1 State of the Art** . . . . . . . . . . . . . . . . . . . . . . . . . . . **50**

**4.2 Skeleton Generation** . . . . . . . . . . . . . . . . . . . . . . . . . . **51**

4.2.1 Communication Pattern . . . . . . . . . . . . . . . . . . . . . . . . 51

4.2.2 Communication Variable . . . . . . . . . . . . . . . . . . . . . . . . 52

4.2.3 Types of Dependencies between Variables . . . . . . . . . . . . . 52

4.2.4 Slicing Criterion and Program Slice . . . . . . . . . . . . . . . . . 53

**4.3 Application Skeleton** . . . . . . . . . . . . . . . . . . . . . . . . . . **54**

**4.4 A Complete Example (DGEMM)** . . . . . . . . . . . . . . . . . . **55**

4.4.1 The Top-Down Phase . . . . . . . . . . . . . . . . . . . . . . . . . 57

4.4.2 The Bottom-Up Phase . . . . . . . . . . . . . . . . . . . . . . . . . 57

**4.5 Automatization with LLVM** . . . . . . . . . . . . . . . . . . . . . . **58**

4.5.1 LLVM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

4.5.2 An Implementation with LLVM . . . . . . . . . . . . . . . . . . . . 61

**4.6 Tuning Time Reduction** . . . . . . . . . . . . . . . . . . . . . . . . **76**

4.6.1 Workflow of the Tuning Process using Skeletonization . . . . . 77

**4.7 Skeletonization Process Validation** . . . . . . . . . . . . . . . . . **78**

4.7.1 Viability of the Approach . . . . . . . . . . . . . . . . . . . . . . . 78

4.7.2 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . . . . 78

4.7.3 Results Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

**4.8 Conclusion** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **84**

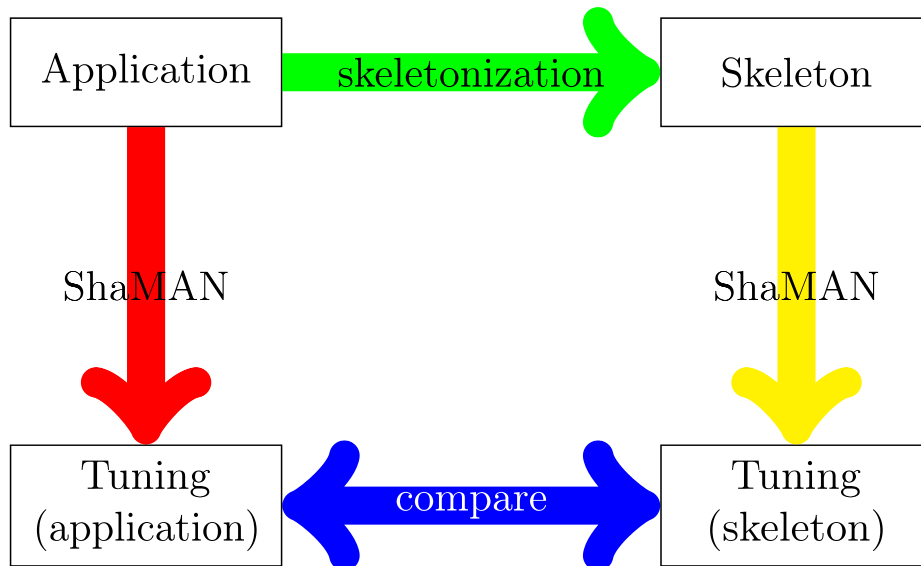



Figure 4.1: Schematic Representation of the Skeleton Usage

This chapter presents the main principles of a new method to tune an MPI application. As described in Chapter 2, producing a tuning file adapted to a specific application can be a time-consuming process, especially when the application itself takes a long time to execute.

The idea here, as shown in Figure 4.1, is to use a proxy application, denoted $skeleton(A)$, instead of the original application $A$ in the tuning process. The desired properties of $skeleton(A)$ would be similar to those of $A$ from an MPI point of view while having a smaller running time.

## 4.1 State of the Art

In the domain of parallel computing, algorithmic skeletons have emerged as an effective high-level abstraction for structured parallel programming. These abstractions allow programmers to focus on the algorithmic structure without managing low-level details like communication and synchronization, which can significantly reduce the complexity and error-proneness of parallel programming. Early skeletonization approaches like those proposed by Cole [20] outlined foundational patterns, such as divide-and-conquer, which remain prevalent in the field. Cole's work laid the groundwork for many skeleton frameworks by providing a formal, structured approach to parallel programming, abstracting frequently used computational patterns like task queues and pipelines into reusable higher-order functions that can be tuned in isolation.

Subsequent research expanded these skeletonization techniques, exploring their application to various domains. For instance, Bird [14] formalized higher-order functions in functional programming, which facilitated the design of parallel algorithms based on skeletons. These algorithms, that can be optimized independently, allow for an elegant and performant way of implementing parallel applications, which is particularly useful in parallel environments.

Moreover, there has been a significant focus on improving the performance and adaptability of skeletons. For example, systems like ASSIST [61] offer flexible parallel skeleton implementations that adapt dynamically to changing computational resources. Such ad-

vancements align with the need for resource-aware computing, particularly in heterogeneous environments like clusters and grids [28].

The refinement of skeleton libraries, like SkeTo [57] and Muesli [19], further promoted the adoption of parallel skeletons by providing practical implementations that integrate well with MPI. However, they are only tools that provide programmers with well established building blocks to create their applications, while our approach aims at reducing an application to its minimal communicating component. This method builds the skeleton, and therefore does not require the application to be based on one, which makes it available for all MPI-based applications.

In their paper [28], González-Vélez *et al.* provide a comprehensive review of algorithmic skeleton frameworks (ASKFs), which are high-level, structured parallel programming tools. They identified three types of ASKFs:

- **Data-parallel**: operations like `map`, `reduce`, and `scan`, typically applied to large data sets
- **Task-parallel**: Patterns like `farm` and `pipeline` that handle task distribution across processors.
- **Resolution skeletons**: Implement algorithms like `divide-and-conquer` and `branch-and-bound`, which recursively solve problems by splitting them into smaller parts

In the previously mentionned [67], Zhai *et al.* used the concept of a program slice, which gave us the idea for the skeletonization process. A program slice can be used to represent the minimal subset of instructions in a program that preserves an attribute of the complete program. In their case, the preserved attribute is the program trace, while in our approach we aimed at preserving the entire communication pattern. The concept is presented in details in Section 4.2.4.

## 4.2 Skeleton Generation

As described in Figure 4.1, an alternate path is taken to produce the tuning file. To reach this goal, the application $A$ undergoes a transformation named *skeletonization*, detailled in Section 4.3. To generate $skeleton(A)$, the application $A$ must be analyzed extensively to determine for each instruction if it can be removed or if it has to be preserved.

To perform the skeletonization process, we use the concepts of *Communication Pattern* and *Communication Variable* to evaluate the instructions in the application. These concepts are detailled in Sections 4.2.1 and 4.2.2 respectively.

### 4.2.1 Communication Pattern

Different MPI applications may exhibit different communication patterns, which can be characterized by three key attributes: volume, spatiality and temporality [18, 38]. The volume attribute encompasses both the number of messages and their sizes sent through the MPI library. The spatial attribute, or spatiality, is expressed in terms of the traffic pattern among the MPI processes whilst the temporal attribute, or temporality, is captured by the message generation rate. We need a concept to formalize whether or not a program statement does modify any one of these attributes, namely a *program slice* [63, 67].

### 4.2.2 Communication Variable

A *Communication Variable* (*Comm Variable* for short) is a parameter of an MPI communication routine which directly impacts either the volume, the spatiality or the temporality of the MPI program. For example, in the signature of the `MPI_Send` operation shown in Listing 4.1, the `buf` parameter is not a *Comm Variable*. Indeed, it holds a pointer to the data to send, and its content does not not change the communication pattern. On the other hand, all other parameters of `MPI_Send` are *Comm Variables*: the `count` and `datatype` parameters characterize the number of bytes to send, changing their values will modify the volume attribute. The combination of `dest` and `comm` determines which MPI process will receive the message, and their modification will change the spatial attribute. Finally, the `tag` value, if changed, might modify the message ordering and thus the spatial attribute.

```
1 int MPI_Send(const void* buf, int count, MPI_Datatype datatype, int dest
    , int tag, MPI_Comm comm)
```

Listing 4.1: Signature of `MPI_Send`

It is possible to determine whether or not a parameter of an MPI communication routine is a *Comm Variable* by reading the MPI standard, as the role and effect of each parameter is defined in it. As explained before, some parameters fall into the *Comm Variable* category, while others need more examination. One of the most notable example is the parameter of type `MPI_Op`, present in multiple MPI collectives communication operations such as `MPI_Reduce`. It is an MPI structure, wrapping the function that is called when doing the reduction operation. We initially supposed that the function used would not change the communication pattern, as all predefined `MPI_Op`s like `MPI_SUM` are commutative. However, after carefully examining the Open MPI source code, we found that user-defined `MPI_Op`s are allowed to call other MPI functions. Moreover, the predefined `MPI_NO_OP`, doing nothing when invoked, is a special case as MPI is allowed to return early from an `MPI_Reduce` using it, thus not performing any communication. In either case, the value of the parameter of type `MPI_Op` is able to change the communication pattern, earning the *Comm Variable* status. Conversely, output parameters and return values are not provided by the user, they are the result of the function call, therefore they are not marked as *Comm Variables* by the function call.

However, variables not marked as *Comm Variables* in a specific MPI call might be marked as such at a later point in the code. For example, it is often the case with the output parameter of `MPI_Comm_rank`, which contains the rank of the calling MPI process. As an output parameter, it is initially not marked as a *Comm Variable*, but offset computation usually depend on its value, therefore marking it as a *Comm Variable*. This dependency transmission is further described in Section 4.2.3. This mechanism allows the transmission of the status to a variable $v$ if the value of a *Comm Variable* depends on the value of $v$. This definition of a *Comm Variable* is used throughout the entire description of the skeletonization process to refer to variables, registers, statements or instructions of $A$ that should be preserved in $skeleton(A)$.

### 4.2.3 Types of Dependencies between Variables

Dependencies are relations between variables where the value of one influences the value of the other. Three types of dependencies may be detected:

- $X$ has a *Data dependency* on $Y$ if the program's computation might change if/when the statements where $X$ and $Y$ appear are reversed.
- $X$ has a *Control dependency* on $Y$ if the value of variable $Y$ determines whether or not the statement containing variable $X$ is executed.
- $X$ has a *Communication dependency* on $Y$ if $X$ has a data dependency on $Y$ and the statement where $Y$ appears is an MPI communication operation.

In Listing 4.2, the relation of `x` toward `y` is an example of a data dependency. If the statements on Lines 2 and 3 were reversed, the value of `x` after their execution would certainly not be 42.

```
int x, y;
y = 42;
x = y;
```

Listing 4.2: Example of a data dependency

In Listing 4.3, the relation of `x` toward `y` is an example of a control dependency, the value of `y` is controlling a possible increment of `x`.

```
int x, y;
if (y > 256) {
  x++;
}
```

Listing 4.3: Example of a control dependency

In Listing 4.4, the relation of `x` toward `y` is an example of a communication dependency, as the value of `x` depends on the value of `y` which itself is the result of an MPI operation, in this case `MPI_Recv`.

```
int x, y;
MPI_Recv(&y, 1, MPI_INT, src, tag, comm, MPI_STATUS_IGNORE);
x = y;
```

Listing 4.4: Example of a communication dependency

### 4.2.4 Slicing Criterion and Program Slice

A *slicing criterion* of a program $P$ is a pair $\langle p, V\rangle$, where $p$ is a statement in $P$, and $V$ is a subset of the variables in $P$. A *program slice* based on the slicing criterion $\langle p, V\rangle$, is the minimal subset of statements that preserves the behavior of the original program up to the statement $p$ with respect to the program variables in $V$. These concepts are used in conjunction to identify the specific portions of a program that exhibit a specific behavior. In our case, the behavior that we want to identify and isolate is the part that determines its communication pattern.

## 4.3 Application Skeleton

We aim at determining the slice of the MPI application $A$ based on the slicing criterion $\langle a, V \rangle$ where $a$ is the last statement of $A$ and $V$ is the subset of the *Comm Variables* of $A$. We use the last statement to cover the entire program. This program slice is therefore the minimal program that exhibits the same communication pattern and behavior than $A$. To achieve this objective, we identify the variables of $A$ that are *Comm Variables*, using our knowledge of the MPI routines that are present in the codebase, and we then propagate this status to all variables of $A$ that have a dependency on them so as to preserve the program slice. At this point, every variable that modifies the behavior of $A$ is marked with the *Comm Variable* status, thus removing all unmarked variables and the operations performed on them will result in the slicing criterion $\langle a, V \rangle$. This new program, that exhibits the same communication pattern without the computational part of the application, is the desired skeleton of the application $A$. It will be referred to as $skeleton(A)$ in the following Sections.

## 4.4 A Complete Example (DGEMM)

For a more concrete description of this process, we present a complete example of a linear algebra general parallel matrix-matrix multiplication written in C (BLAS DGEMM), presented in Listing 4.5. The skeletonization process requires two phases, the first one analyzes all instructions from top to bottom to indentify the *Comm Variables*, the second one propagates the status from bottom to top.

```c
#include <mpi.h>
#include <stddef.h>

#define N 80
#define MAT(m, i, j) (m)[(j)*N+(i)]
#define COMM MPI_COMM_WORLD

int main(void) {
  int err, myid, nprocs, cols, mysize, tag, master, offset;
  MPI_Status st;
  double A[N*N], B[N*N], C[N*N];
  err = MPI_Init(NULL, NULL);
  err = MPI_Comm_rank(COMM, &myid);
  err = MPI_Comm_size(COMM, &nprocs);
  cols = N / (nprocs - 1);
  mysize = cols * N;
  tag = 1;
  master = 0;
  if (myid == master) {
    for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
        MAT(A, i, j) = i + j;
        MAT(B, i, j) = i * j;
      }
    }
    for (int dest = 1; dest < nprocs; ++dest) {
      offset = (dest - 1) * mysize;
      err = MPI_Send(A, N*N, MPI_DOUBLE, dest, tag, COMM);
      err = MPI_Send(B+offset, mysize, MPI_DOUBLE, dest, tag, COMM);
    }
    for (int source = 1; source < nprocs; ++source) {
      offset = (source - 1) * mysize;
      err = MPI_Recv(C+offset, mysize, MPI_DOUBLE, source, tag, COMM, &
   st);
    }
  } else {
    err = MPI_Recv(A, N*N, MPI_DOUBLE, master, tag, COMM, &st);
    err = MPI_Recv(B, mysize, MPI_DOUBLE, master, tag, COMM, &st);
    for (int k = 0; k < cols; ++k) {
      for (int i = 0; i < N; ++i) {
        MAT(C, i, k) = 0.0;
        for (int j = 0; j < N; ++j) {
          MAT(C, i, k) += MAT(A, i, j) * MAT(B, j, k);
        }
      }
    }
    err = MPI_Send(C, mysize, MPI_DOUBLE, master, tag, COMM);
  }
  err = MPI_Finalize();
  (void) err;
  return 0;
}
```

Listing 4.5: parallel matrix-matrix multiplication in C

### 4.4.1 The Top-Down Phase

We start with the identification of the *Comm Variables*. They are listed here, along with their line number to avoid confusion between multiple occurrences of the same label: (`myid`, 13), (`nprocs`, 14), (`dest`, 28), (`tag`, 28), (`mysize`, 29), (`dest`, 29), (`tag`, 29), (`mysize`, 33), (`source`, 33), (`tag`, 33), (`st`, 33), (`master`, 36), (`tag`, 36), (`st`, 36), (`mysize`, 37), (`master`, 37), (`tag`, 37), (`st`, 37), (`mysize`, 46), (`master`, 46), and (`tag`, 46). `N`, `MPI_COMM_WORLD` and `MPI_REAL` are not variables, but constant values, and are therefore left unmarked. The `err` variable, the output result of MPI routines, behaves similarly to `buf`, and is thus not marked with the *Comm Variable* status, as it does not impact the communication pattern unless an error occurs and is properly handled.

### 4.4.2 The Bottom-Up Phase

In this phase, all dependencies involving a *Comm Variable* are determined in order to propagate this status upward in the program. In this example program, the *Comm Variable* (`mysize`, 16) has a Data dependency on the regular variable (`col`, 16), promoting it to a *Comm Variable*. It is the sole occurrence of a dependency that changes the status of a variable. Now that all *Comm Variables* are known, the statements that do not modify them can be removed. For instance, the assignments to the variable `offset` on Lines 27 and 32 can be removed. Similarly, the initial construction of $A$ and $B$ on Lines 20 to 25 can be deleted, along with the assignments to $C$ on Lines 38 to 45. The remaining statements constitute the skeleton of the program. The code produced after manually performing the two steps is presented in Listing 4.6.

```
1  #include <mpi.h>
2  #include <stddef.h>
3
4  #define N 80
5  #define MAT(m, i, j) (m)[(j)*N+(i)]
6  #define COMM MPI_COMM_WORLD
7
8  int main(void) {
9    int myid, nprocs, cols, mysize, tag, master, offset;
10   offset = 0;
11   MPI_Status st;
12   double A[N*N], B[N*N], C[N*N];
13   MPI_Init(NULL, NULL);
14   MPI_Comm_rank(COMM, &myid);
15   MPI_Comm_size(COMM, &nprocs);
16   cols = N / (nprocs - 1);
17   mysize = cols * N;
18   tag = 1;
19   master = 0;
20   if (myid == master) {
21     for (int dest = 1; dest < nprocs; ++dest) {
22       MPI_Send(A, N*N, MPI_DOUBLE, dest, tag, COMM);
23       MPI_Send(B+offset, mysize, MPI_DOUBLE, dest, tag, COMM);
24     }
25     for (int source = 1; source < nprocs; ++source) {
26       MPI_Recv(C+offset, mysize, MPI_DOUBLE, source, tag, COMM, &st);
27     }
28   } else {
29     MPI_Recv(A, N*N, MPI_DOUBLE, master, tag, COMM, &st);
30     MPI_Recv(B, mysize, MPI_DOUBLE, master, tag, COMM, &st);
31     MPI_Send(C, mysize, MPI_DOUBLE, master, tag, COMM);
32   }
33   MPI_Finalize();
34   return 0;
35 }
```

Listing 4.6: Source code equivalent of Listing 4.5 after skeletonization

## 4.5 Automatization with LLVM

The skeletonization process presented in the previous Section is both error-prone and time-consuming when performed manually, which makes it an excellent candidate for automation. The initial realization is presented in Section 4.7.1, however the viability of this approach is largely undermined by the fact that it is carried out manually, thus rendering it not applicable to applications with a large codebase. To carry out the skeletonization automatically, tools to realize code analysis and transformation were required, and the LLVM suite was selected. The complete automatization process is detailled in Section 4.5.2, while LLVM itself is presented in Section 4.5.1.

### 4.5.1 LLVM

The LLVM project [59, 40] is a collection of modular and reusable compiler and toolchain technologies, that can be used to develop a frontend for any programming language and a backend for any instruction set architecture. LLVM is designed around an *intermediate*

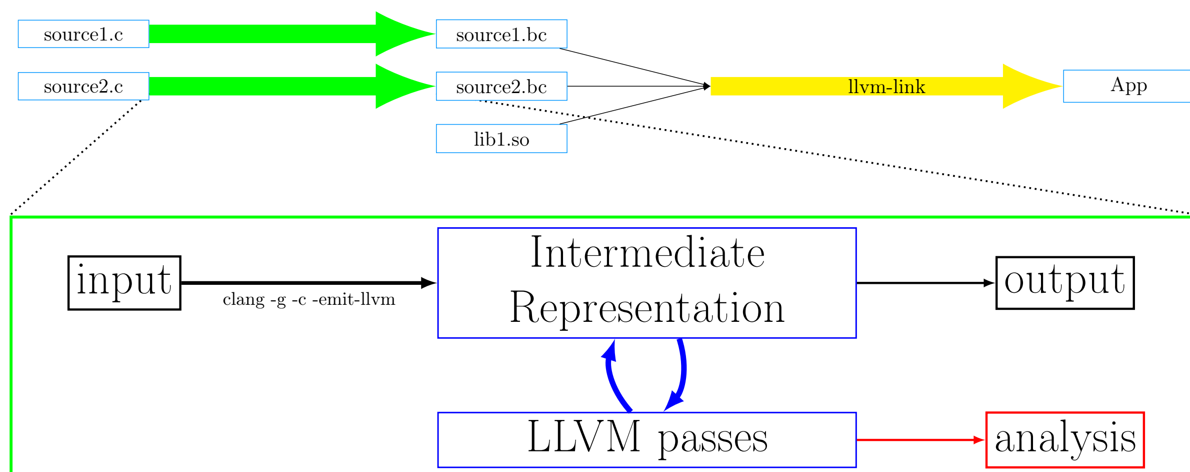



Figure 4.2: Compilation process using LLVM

*representation* (LLVM IR, or simply IR) that is a language independent and portable representation of the code. One of its most notable tool is `clang`, a compiler for the C programming language. It includes numerous passes, transformation functions that run on the IR to perform various optimizations, such as eliminating dead code or propagating constants values.

Figure 4.2 shows a compilation example using `clang`. The green box represents what occurs inside the green arrows. The `clang` executable is a wrapper around multiple binaries inside LLVM that are invoked sequentially such as a parser, a lexer, an optimizer or an assembler. Each of those binaries is a standalone and can be called independently, or by providing the appropriate flags to `clang`.

The frontend (`clang -cc1`) is used to convert C source code into the LLVM IR. It first parses the file to generate the Abstract Syntax Tree (AST), emitting syntax errors if the code does not respect the C language syntax. It is eventually preceded by a precompiler phase where C macros are expanded. If the code is syntaxically correct, the LLVM IR is generated. It can be written in two different formats, `.bc` and `.ll`. The first one is a binary file (bitcode) and is memory efficient, while the second is a regular text file that is human-readable.

Then, the optimizer `opt` is invoked on the resulting IR. LLVM contains multiple transformation functions, known as LLVM passes, that take the IR or part of it as a parameter to produce a result. The immutable passes do not modify the input but analyse the IR to produce an object containing useful informations about it. For example, the `BasicAliasAnalysis` pass is a stateless machine running on the IR that produces an object of the generic type `AAResults`, object that can be queried to know if two memory locations can be accessed from one another. For example, Listing 4.7 presents two pointers that can both be used to access the value 3, through `array[3]` and `*(two+1)`. In this situation, we say that `array` and `two` are *aliases* (partial aliases in that case).

```
1 int array[5] = { 0, 1, 2, 3, 4 };
2 int* two = &array[2];
```

Listing 4.7: Example of two pointers that are partial aliases

The query results of an `AAResults` are of type `AliasResult`, an enumeration with values:

- `NoAlias`: The two locations are not *aliases* at all
- `MayAlias`: The two locations may or may not be *aliases* (this is the least precise result)
- `PartialAlias`: The two locations are *aliases*, but only due to a partial overlap
- `MustAlias`: The two locations are *aliases*

These informations produced by this pass are very useful for other passes to perform their own transformation, but said transformation will invalidate the `AAResult`. Other passes producing alias analysis exist and are used in more specific cases, the `BasicAliasAnalysis` is the least elaborated one, but also one of the cheapest to compute and other alias analyses may rely on it to handle simple cases.

The informations produced by immutable passes can be later used by the other kind of passes, the transformation passes, that modify the IR to achieve a desired property. For example, the `LoadStoreVectorizer` transformation pass is able, under some conditions, to merge multiple small load instructions into a single wider load instruction (known as a vector load, hence the name). It operates in a similar way on store instructions, and is usually applied to codes that are intended to run on GPU devices, where loading multiple elements at once is very beneficial on their architecture. To operate properly, this pass requires, among other prerequisites, an alias analysis that is able to prove that two memory locations are disctinct (they are not aliases) so that they can be combined.

```
1 void increment(int* ptr1, int* ptr2, int amount) {
2   *ptr1 += amount;
3   *ptr2 += amount;
4 }
```

Listing 4.8: Example of Alias Analysis Usage

In Listing 4.8, the result of `clang` will depend on the result of the alias analysis. If it detects that `ptr1` and `ptr2` are not aliases, their load/store instructions can be combined. If it can be proven that `ptr1` and `ptr2` are aliases (`ptr1 == ptr2`), then another pass (`InstructionCombine`) will merge the two add instructions together. In all other cases, such optimization can not be performed.

As a side note, even an advanced alias analysis may not be able to find any aliases, thus leading to no optimization. To influence the analysis and help the optimization, developpers have one keyword at their disposal: the `restrict` type qualifier. It only applies to pointers and indicates that within the scope where a pointer `p` is `restrict`-qualified, any object accessed through `p` (directly or indirectly) is only accessible through `p` and no other pointer. If any of `ptr1` or `ptr2` was marked `restrict`, the alias analysis will be able to deduce that they are not aliases of one another and perform the combination.

Each pass can be invoked independently, but some have dependencies, as seen with the `LoadStoreVectorizer` pass. An other example of dependency would be the transformation pass that unrolls loops (`LoopUnrollPass`), which require the `IndVarSimplify` pass to run before it, and that its result were not invalidated. The `IndVarSimplify` pass transforms the induction variables (variables used as loop indices) into simpler forms, potentially allowing them to be removed when trying to unroll the loops. Each pass is responsible for updating the intermediate results and notify when they are invalidated. These dependencies are all resolved by the `opt` tool, which tries to run as few passes as possible, executing required passes first, precomputing intermediate results and feeding them to passes that do not invalidate them in priority.

The passes can thus be invoked in sequences, denoted pipelines. Several predefined pipelines exist, the most notable being of the form `default<OX>` with `X` corresponding to an optimization level (0 to 3, or `s` to optimize for binary size). As `opt` is the backbone of `clang`, all these pipelines are by extension available as options to the compiler.

Finally, the linker `llvm-link` analyzes the IR at the end of the `opt` pipeline. It is responsible for aggregating different IRs, resolving symbols and producing an executable file. The LLVM IR is, similary to the input language, independent of the output format, or instruction set. That allows `llvm-link` to produce files for different architectures, including (but not restricted to) `x86-64`, `ARM` and `WebAssembly`. At the end, `llvm-link` outputs an executable binary using as default the host's instruction set.

LLVM allows the user to define its own passes, that can be independently applied to the IR as part of a compilation pipeline (a sequence of LLVM passes). The skeletonization process was implemented as a transformation pass.

### 4.5.2 An Implementation with LLVM

The manual process of skeletonization, as outlined at the beginning of this Section, is both labor-intensive and error-prone, making it unsuitable for complex and large codebases. To address these issues, we implemented an automated solution using the previously mentioned LLVM compiler infrastructure. In this Section, we detail step-by-step this new LLVM pass, along with the specific challenges that we encountered to ensure the correctness and efficiency of the skeletons produced.

Here are the main steps of the core of our solution, encapsulated in the `SkeletonPass::run()` function, which implements the skeletonization process:

- step 1 builds the data structure that will contain the informations (`FuncMap`)
- step 2 finds the `main` function and adds it to the data structure
- step 3 fills the data structure by recursively exploring the IR (starting with `main`)
- step 4 marks *Comm Variable* as such using the knowledge of the MPI standard, as discussed in Section 4.2.2
- step 5 recursively propagates the *Comm Variable* status using dependencies that can be obtained with LLVM
- step 6 recursively removes instructions that were not marked as *Comm Variable*
- step 7 adjusts and prepares the resulting IR

The implementation consists of a single core function (`SkeletonPass::run`) whose sole parameter is a reference to the LLVM IR (`llvm::Module&`). It is compiled into a shared library that can be loaded dynamically by the `opt` tool, and the function `SkeletonPass::run` is executed, with its parameter being the current state of the IR.

The LLVM IR is comparable to a matryoshka doll, as it is composed of containers included in other containers. The topmost container is the previously mentioned `llvm::Module`, which contains global elements like functions, aliases and global variables.

#### 4.5.2.1 A Simple Example of a Pass

```
using namespace llvm;

PreservedAnalyses FunctionNames::run(Module& M) {
  for (Function& F : M) {
    errs() << F->getName() << '\n';
  }
  return PreservedAnalyses::all();
}
```

Listing 4.9: A Simple LLVM pass that prints function names

In the simple pass example shown in Listing 4.9, we use C++ *range-based for loop* to iterate over the `Module` container, more specifically over the `Function` elements. For each of them, we print their name on the standard error stream of `opt`. `PreservedAnalyses::all()` is returned to indicate that no analysis pass (such as `BasicAliasAnalysis`) was invalidated by executing this plugin. The boilerplate code required for the pass to cleanly fit in `opt` is not shown here.

```
void greet(char* who) {
  printf("Hello %s\n", who);
}

int main() {
  greet("world");
  return 0;
}
```

Listing 4.10: Convoluted hello_world program in C

```
greet
main
```

Listing 4.11: Possible output

When this plugin code is executed over the LLVM IR produced by Listing 4.10, the output prints the two function names, in unspecified order, as shown in Listing 4.11.

```
@.str = private unnamed_addr constant [10 x i8] c"Hello %s\0A\00", align
    1
@.str.1 = private unnamed_addr constant [6 x i8] c"world\00", align 1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @greet(ptr noundef %0) #0 {
  %2 = alloca ptr, align 8
  store ptr %0, ptr %2, align 8
  %3 = load ptr, ptr %2, align 8
  %4 = call i32 (ptr, ...) @printf(ptr noundef @.str, ptr noundef %3)
  ret void
}

declare i32 @printf(ptr noundef, ...) #1

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  call void @greet(ptr noundef @.str.1)
  ret i32 0
}
```

Listing 4.12: Human readable version of the LLVM IR

Listing 4.12 presents the human-readable version of the non optimized LLVM intermediate representation of the code in Listing 4.10. Some of the irrelevant part were omitted. It reads as follows: Lines 1 and 2 hold the constant string variables, under the names `@.str` and `@.str.1`, later referenced in both `greet` and `main`. Lines 4 to 11 is the definition of the `greet` function. It has several attributes on Line 4, most of them being here because no optimization was performed. The complete list of attributes attached to a function can be found by looking at the # after the function signature and locating in the file the line starting with the same # identifier (not shown in the Listing). Text starting with a semicolon is a comment, LLVM only adds it for readability but ignores it when using the IR. The `noinline` attribute is present by default and indicates that the function may not be inlined. It disappears if the code is compiled with any higher level of optimization, as inlining a single function is often possible. The `nounwind` simply indicates that the function does not use exceptions, and therefore that no special unwinding instructions need to be emitted when calling it. Unwinding is detailled further in Section 4.5.2.8. `greet` takes a single pointer argument, denoted `%0` and marked `noundef`, meaning that if the compiler manages to prove that an undefined pointer ends up in this place, it must emit an error. Line 6 corresponds to an `llvm::AllocaInst`, an instruction that reserves memory for a variable by making space on top the stack. In this case, the variable is a pointer, most likely a 64 bits wide as the alignment is 8, but the exact width used is internal to LLVM. The result of the `llvm::AllocaInst` can be referred to as `%2`. It is used in the very next line, as the value of the parameter `%0` is stored in the reserved memory. A store instruction (`llvm::StoreInst`) does not have a return value, one can only access the results with the pointers. Line 8 is a `llvm::LoadInst`, the exact opposite of a store. It loads a value from memory, usually in preparation for the next instruction. The loaded value is a pointer, loaded from `%2` and accessible through `%3`. Line 7 and 8 could easily be removed, they are present here solely because no optimization was done. Line 9 calls the function `printf` using the previously mentionned `@.str` and `%3`. The syntax `i32 (ptr, ...)` indicates that `printf` is a variadic function, taking a pointer and list of arguments of unknown

length as inputs to return an integer (`i32`). The full list of arguments is indicated after it. The return value is stored in `%4`, but in this case it is not referenced later. Finally, the last line returns from the function, it corresponds to a `llvm::ReturnInst` and in this case it returns nothing (`void`). Line 13 declares the `printf` function. Let's remember that when the LLVM IR is produced and worked with, the linking phase has not been performed yet. The declaration is simply indicating a requirement for the linker to resolve. Lines 16 to 21 are the definition of the `main` function. The first instruction in it allocates memory on the stack to hold the return value, in this case an integer. This instruction is present for every non-void function, and the 4 bytes alignment is a prerequisite specific to the `main` function for compatibility with 32 bits architectures. They could easily be removed, along with the store instruction next line storing 0 in the allocated memory, because the last line directly returns 0. These redundant instructions are present once again because of the absence of optimizations. Line 19 is a `llvm::CallInst`, invoking the function `printf` with the argument `@.str.1`. Finally, as previously stated, the next line returns the value 0.

#### 4.5.2.2 A More Detailled Example

Listing 4.13 presents parts of a more advanced pass that is able to remove conditional branches whose condition can be determined at compile time.

```
1  static bool eliminateCondBranches(Function& F) {
2    bool Changed = false;
3
4    for (BasicBlock& BB : F) {
5
6      // Skip blocks without conditional branches as terminators.
7      BranchInst* BI = dyn_cast<BranchInst>(BB.getTerminator());
8      if (!BI || !BI->isConditional())
9        continue;
10
11     // Skip blocks with conditional branches without ConstantInt
       conditions.
12     ConstantInt* CI = dyn_cast<ConstantInt>(BI->getCondition());
13     if (!CI)
14       continue;
15
16     // Remove dead branch
17     Changed = true;
18     ...
19
20   }
21   return Changed;
22 }
```

Listing 4.13: Portion of a pass eliminating conditional branches

It works by taking as input a `Function` and iterating over its `BasicBlocks`, the containers at the base of the LLVM IR, representing blocks of instructions without jumps. They must however end with a jump instruction, represented in the IR by a `BranchInst`. They are instructions containing the addresses of the next `BasicBlock` the control flow has to jump to, and an optional condition if there are multiple successor blocks. If after some optimization the condition can be reduced to a compile time constant, the following pass will be able to remove the dead branch.

LLVM IR makes extensive use of the C++ class capabilities. Every compilation concept is implemented as a class to encapsulate its characteristics and functionalities, as can

be seen in the case of the `BranchInst`. It can also be seen in Figure 4.3, which shows the inheritance graph of the `llvm::Constant` class, that many elements inherit from this interface, most notably `ConstantData` which refers to constant values declared in the code. Their main commonality is the possibility for the user of the class to access informations that are known at compile time, which explains the presence of `GlobalVariable` among the descendants of `Constant`. Indeed, a `GlobalVariable` may not be constant but fits the `Constant` requirements of having its information available at any time, including whether or not it is constant. The `GlobalVariable` has an exclusive method `isConstant` to access this information. Through optimizations, a variable value can be promoted to a `ConstantData` or one of its descendant like `ConstantInt`.

Presented in Listing 4.13, the `dyn_cast<>` operator is used throughout the LLVM codebase. It checks to see if the operand is of the specified type, and if so, returns a pointer to it (this operator does not work with references). If the operand is not of the correct type, a null pointer is returned. Thus, this works very much like the `dynamic_cast<>` operator in C++, and should be used in the same circumstances. Typically, the `dyn_cast<>` operator is used in an `if` statement, as seen in Listing 4.13. This operator was introduced for compile time performance, and works by making use of the LLVM built-in type informations instead of relying on the much more costly C++ `dynamic_cast<>` which require the compiler to generate a virtual table at compile time and dereference several pointers through the table at runtime.

The cast at Line 7 accesses the last instruction of the `BasicBlock` through the `getTerminator()` method and checks if it is a `BranchInst` or derived from it. The `BranchInst` are instructions that will cause a jump in the code execution, like an `if` statement or a `return` instruction. `BasicBlocks` are a list of instructions with the condition that:

- no instruction but the last one is a `BranchInst`
- the last instruction must be a `BranchInst`

They are the basic blocks between which the control flow of the program jumps. If the conditions are not respected, the LLVM assembler will refuse to create a binary and abort. Line 8 verifies that the instruction is a `BranchInst` by checking if it is not `nullptr`, and then checks if the branch is not conditional. If we enter this `if` statement, we know that there is no conditional branch to remove, and move to the next block.

If the execution reaches below Line 10, we know that we have a conditional branch that could be removed. Line 12 casts the condition of the branch to a `ConstantInt`, and the following `if` checks the result for `nullptr`. If the casts results in a `nullptr`, we know that the optimization could not reduce the condition to a compile-time constant, and we can not remove the branch. However, if the cast succeeds, we have a an opportunity for optimization as the branch is conditional (thus having multiple successors), but we can remove successors that can not be reached (as we know the value of the condition).

#### 4.5.2.3 SkeletonPass Preparation

At first, the SkeletonPass initializes the data structures that we use in the following phases to keep track of every information required. A `struct VarInfos` is used to store the following informations:

- `Trilean commVar`: is the variable a *Comm Variable*

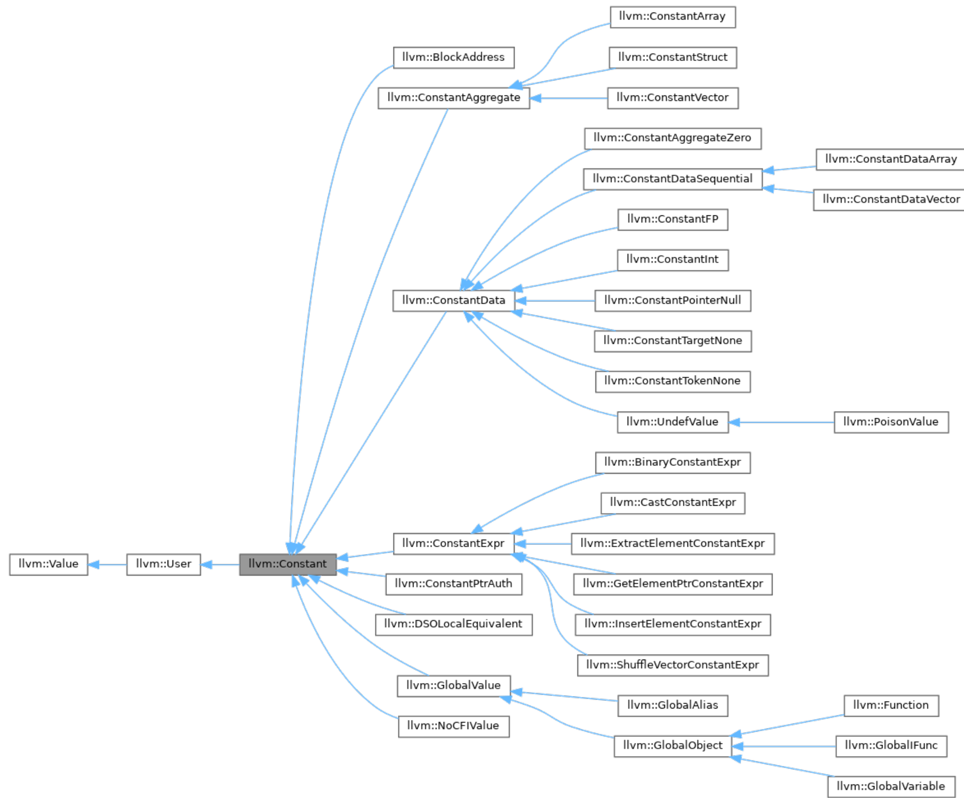



Figure 4.3: The inheritance graph of the class `llvm::Constant`

- `Trilean memVar`: is the variable the output of a memory allocation call
- `Trilean returnVar`: is the variable returned by the function
- `Trilean resultVar`: is the variable the result of a function call

The meaning of these informations will be detailled further in Section 4.5.2.4. Within the LLVM IR, an `llvm::Instruction` is itself a variable that can be manipulated and that, most of the time, corresponds to a register once compiled in the binary. Therefore, the term `Variable` will be used to refer to an `llvm::Instruction` and its associated informations. We do not know *a priori* the type of the `llvm::Instruction`, so we store only the informations that apply to all types. The handling of specific types of instruction is delegated to a specialized function.

The statuses are not stored as booleans but as trileans, which have an additional `maybe` value. This special state is used when the status is ambiguous, most notably when analyzing a conditional loop where the status of the variables depends on values that are outside the body of the loop. The default value of a trilean matches that of the boolean (`false`), which means that the default value of a `varInfo` represents a variable that has no special status.

Pointers to an `llvm::Instruction` and `VarInfos` structures holding the associated informations are stored as key-value pairs in an map. The type `std::unordered_map` was chosen for the two following reasons: first, we only have access to the Intermediate Representation, which is mostly composed of pointers to llvm structures scattered across the memory, and we need to match these pointers to their `VarInfos`. Second, the number of instructions in a function can grow to arbitrarily large sizes, and we needed a data

structure that allows to quickly access the informations associated with an instruction. The unordered map provided by the standard library was appropriate as we can customize it to use any hashable type as the key (and pointers are numbers, thus hashable), and the map allows access to every element in a constant amortized time. This map is denoted as the `VarMap` of the function.

The `FuncInfos` works very similarly to the `VarInfos`, but at the `llvm::Function` level. It is composed of:

- `VarMap varMap`: informations about the variables of the function
- `std::vector<VarInfo> parameters`: informations about the parameters of the function
- `VarInfo retvalStatus`: informations about the returned value

The default value has both `varMap` and `parameters` empty, they will be filled later on. All parameters can be only be determined in advance for MPI functions.

Finally, the most important data structure is the `FuncMap`, associating pointers to `llvm::Function` to the `FuncInfo` with their relevant values. It is this structure, named `context`, that we initialize to an empty map at the very beginning of the function `SkeletonPass::run`.

#### 4.5.2.4 Code Analysis

The first iteration through the LLVM IR fills the structure defined in the previous Section with the relevant informations. It starts by finding the `main` function, a symbol that must exist for the `Module` to be correct. Once found, we iterate over the `BasicBlock`s of the function, and then over the `Instruction`s of each `BasicBlock`.

The most important instruction types have a specialized overloaded function to add them to the `context`, as some properties are to be preserved for the IR to remain valid. Other informations are also stored to preserve some of the code, most notably memory allocations and deallocations so as to not cause any SIGSEGV or OOM killers.

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  int value;
} myint_t;

// opaque pointer
typedef void* myint;

myint myint_allocate(void) {
  return malloc(sizeof(myint_t));
}

void myint_free(myint i) {
  free(i);
}

int myint_get(myint const i) {
  myint_t* ptr = (myint_t*) i;
  return ptr->value;
}

void myint_set(myint i, int v) {
  myint_t* ptr = (myint_t*) i;
  ptr->value = v;
}

int main(void) {
  myint i = myint_allocate();
  myint_set(i, 42);
  printf("%d\n", myint_get(i));
  myint_free(i);
  return 0;
}
```

Listing 4.14: C code allocating memory with malloc

Listing 4.14 demonstrates the use of the `memVar` status, added to fix one of the main problem of the skeletonizer: memory allocation. It is an additional status that variables can have, indicating that they are the result of a memory related function call. Similarly to MPI functions, memory allocating and deallocating functions can be known by reading the documentation. Typical buffer are not marked with the *Comm Variable* status, leading to their initialization being removed from the skeleton. However, in the common case of mallocated buffers, this behavior is problematic as we need the memory to exist for the MPI functions to access it without triggering a *segmentation fault* signal (SIGSEGV). In this example, the call to the function `myint_allocate` and its return value will be marked with the `memVar` status as it arises from a call to `malloc`. Similarly, `myint_free` and its argument will be marked due to the `free` call. Currently, the hardcoded database for memory-related functions only contains functions from the `malloc`, `mmap`, `brk` and `posix_memalign` families, but can easily be extended.

As the iteration progresses, the `FuncMap` gradually fills up with informations by repeatedly calling a specialized function on every traversed instruction. Listing 4.16 shows the specialized functions to add an `llvm::Instruction` to the `FuncMap`. If no specialization matches the current instruction, a generic one is called which only performs the addition to the `context`, without any additional operation, as presented in Listing 4.15.

```
template<>
void add(Instruction* I, FuncMap& context) {
  // Retrieve parent function
  Function* parent = I->getFunction();
  // add default empty varInfo to the context
  context.at(parent).variables[I] = {};
}
```

Listing 4.15: Generic function to add to the context

```
template<>
void specialized_add(ReturnInst* RI, FuncMap& context) {
  Function* parent = RI->getFunction();
  // create and retrieve varInfo
  VarInfos& varInfos = context.at(parent).variables[RI] = {};
  // tag the instruction as "being returned"
  varInfos.returnVar = true;
}
```

Listing 4.16: Specialized functions to add to the context

#### 4.5.2.5 Variables Status Propagation

Once the analysis of the `Module` is completed, all the informations required to propagate the *Comm Variable* status are gathered in the `FuncMap`, and the process can begin. We start from the last instruction of the `main` function and work our way upward. This direction was chosen for simplicity: an instruction usually depends on something that precedes it, therefore working from bottom to top minimizes the number of out-of-order operations that the pass has to perform.

This step works by iterating backward on every instruction, calling for each a function to perform that propagation. The function is given informations through the `context`, and a specialized function is called for instruction types that require a more careful treatment. For example, the `CallInst`, which represents the call of another function $F$, requires to recursively invoke the propagation process inside $F$.

```
template<>
bool specialized_propagate(LoadInst* LI, FuncMap& context, CallStack& cs
    ) {
  (void) cs;
  // if the LoadInst itself is a commVar, tag the pointer operand as
    commVar
  if (getVarInfos(LI, context).commVar) {
    Instruction* pointer = dyn_cast<Instruction>(LI->getPointerOperand()
    );
    VarInfos& varInfos = getVarInfos(pointer, context);
    if (varInfos.commVar) return false; // no rerun needed
    // else set comm var to true and rerun propagation
    varInfos.commVar = true;
    return true;
  }
  return false;
}
```

Listing 4.17: Specialized function for `LoadInst`

```
template<>
bool specialized_propagate(StoreInst* SI, FuncMap& context, CallStack&
    cs) {
  (void) cs;
  Instruction* pointer = dyn_cast<Instruction>(SI->getPointerOperand());
  if (!getVarInfos(pointer, context).commVar) return false; // no rerun
    needed
  // the pointer operand is commVar, tag self and value operand as
    commVar
  Instruction* value = dyn_cast<Instruction>(SI->getValueOperand());
  bool rerun = false;
  VarInfos& valueInfos = getVarInfos(value, context);
  if (!valueInfos.commVar) {
    valueInfos.commVar = true;
    return = true;
  }
  VarInfos& selfInfos = getVarInfos(SI, context);
  if (!selfInfos.commVar) {
    selfInfos.commVar = true;
    return = true;
  }
  return rerun;
}
```

Listing 4.18: Specialized function for `StoreInst`

Listing 4.17 presents the specialized function used to propagate informations from a `LoadInst` to the `context`. The parameter of type `CallStack` is voided as it is only used for `CallInst`. The same description fits the Listing 4.18, showing the specialized function for the `StoreInst`.

For example, if we run the skeletonization process on Listing 4.2, that illustrated the data dependency, up to the propagating step, the corresponding LLVM IR will ressemble to Listing 4.19. If we suppose that `x` is a *Comm Variable*, as `x` has a data dependency on `y`, then `y` must inherit the *Comm Variable* status. In Listing 4.19, `%1` and `%2` correspond respectively to the declarations of `x` and `y`, and `%1` is marked as *Comm Variable*. As we go from bottom to top, the first line to examine is the number 5, where the value of `%4` is stored at `%1`. As `%1` is a *Comm Variable*, `%4` will be marked as a *Comm Variable*. Next is the line number 4, which loads the value of `%2`. As the instruction itself is a *Comm Variable*, it will mark its argument, `%2` as a *Comm Variable*. Finally, the line number 3 stores a constant value at `%2` which is a *Comm Variable*, thus marking itself as *Comm Variable*.

```
%1 = alloca i32, align 4
%2 = alloca i32, align 4
store i32 42, ptr %2, align 4
%4 = load i32, ptr %2, align 4
store i32 %4, ptr %1, align 4
```

Listing 4.19: LLVM IR of Listing 4.2

All specialized functions return a boolean, which indicates whether or not the propagation should be run again on this block. It is used to handle cases where an instruction depends on something happening after it, the most common cases are loops, where the loop condition depends on its body. For these situations, the `Trilean` type is used. We entrer a loop where the status of each instruction that is not a *Comm Variable* is marked with the value `maybe`. This is to account for a potential modification of the status after the inital execution of the pass. During this phase, if a status is modified, an extra execution of

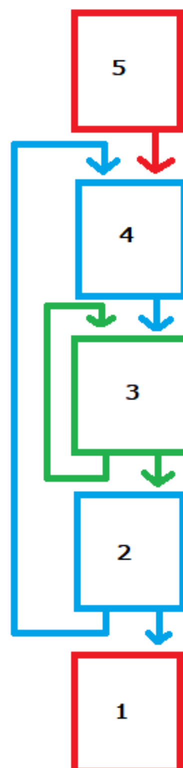



Figure 4.4: BasicBlocks with branching between them and their order of processing

the loop is needed, and the function returns `true`. The loop continues until no additional executions are needed, at which point all statuses that are still classified as `maybe` are reverted back to `false`. Figure 4.4 shows the organisations of `BasicBlocks` in the case of two nested `for` loops.

In this situation, processing instructions in the block number 4 (the begining of the outer loop) might depend on instructions in the block number 3 (the inner loop). This is usually a control dependency, that is linked to a pair of `LoadInst` and `StoreInst`. If the status of any of the two instructions changes, it means that a control dependency occured, and that another execution is needed. The specialized function to propagate from these instruction types will thus return `true`, as can be seen at Line 10 of Listing 4.17.

`CallInst` also requires careful processing, as its status dictates if the function call must be kept. If it is to be kept, the status will be used to qualify the return value of the function before executing the propagation from it. The return value of `main` is not considered as a *Comm Variable*. The status of all other function calls is known before applying the propagation inside the function, and this status should not change (unless recursion was used, that is one of the limitations of the processus discussed in Section 4.5.2.8).

#### 4.5.2.6 Removal of Unecessary Variables

Once the propagation has finished, we know for each instruction if it has an impact on the communication pattern, and thus if it needs to be kept or not. This next step is thus rather simple: if an instruction does not need to be kept, it is removed.

Several properties of the IR, required for LLVM to correctly work, must be preserved through the removal process:

- Functions that feature a non-void return type in their signature must return something (an `llvm::ReturnInst` without value is disallowed)
- Memory locations (pointers and arrays) must keep their initialization (as to not have uninitialized pointers that are very likely to cause a SIGSEGV when used)
- BasicBlocks must end with an instruction that is derived from `llvm::BranchInst`, which includes `llvm::ReturnInst`
- The remaining IR must not contain `llvm::UndefValue`, which is the LLVM way of representing an undefined value or an invalid reference.

The first property is the reason for the presence of a member `returnVar` in the `VarInfos` struct. Similarly to *Comm Variable* and `memVar`, it stores a Trilean information indicating if the relevant variable is returned from the function, in which case it has to be preserved (even if said variable is not a *Comm Variable*). It is used to mark variables that are returned by the current function, in order to preserve the associated `llvm::ReturnInst`. For variables that are returned from a function, an additional status, `resultVar` is used in the `VarInfos` struct. This trilean is used during the removal process to check if the returned variable is assigned to another variable. If that is the case, removing the assignment will be linked to the removal of the function call. Indeed, removing the function call without removing the following assignment will result in an invalid reference. Conversely, removing the assignment without removing the function call is a violation of either a data dependency or a communication dependency

The second property is handled with the `memVar` member of the `VarInfos` and is discussed in details in Section 4.5.2.4.

```
auto iterator = block->begin();
while (iterator != block->end()) {
  Instruction* I = &*iterator;
  if (remove(I, context)) {
    I->replaceAllUsesWith(UndefValue::get(I->getType()));
    iterator = I->eraseFromParent();
    continue;
  }
  ++iterator;
}
```

Listing 4.20: Code responsible for removing instructions in a block

Listing 4.20 shows how the skeletonizer proceeds to remove instructions within a block. An iterator is created to iterate over every instruction of the block. Dereferencing this iterator does not produce an `Instruction*` but an LLVM struct that override `operator&`, and that is why the instruction is acquired with the syntax `&*`. The `remove` function is tasked to call the specialized functions for instructions that must be handled carefully. These functions simply return a boolean indicating whether or not their instruction must be removed, depending on the `context`. If no specializations were made for an instruction type, the `remove` function simply checks if the instruction is marked with any status (*Comm Variable*, `memVar`, `returnVar` or `resultVar`), in which case it returns `true`, otherwise it returns `false`.

If the `remove` function indicates that the instruction must be removed, we first call the LLVM function to replace in the IR all uses of the instruction with an undefined value of the same type. That replacement is made to avoid complications with missing references later on. Next, as the instruction is no longer used anywhere, we can safely remove it

using `EraseFromParent`. As removing an element will invalidate the iterator, this function returns a new valid iterator that is assigned to the previous one. Finally, we increment the iterator to proceed to the next instruction.

```
template<>
bool specialized_remove(ReturnInst* RI, FuncMap& context) {
  // never remove ReturnInst
  (void) RI; (void) context;
  return false;
}
```

Listing 4.21: Specialized function to remove a `ReturnInst` from context

As an example of a specialized function, Listing 4.21 presents the one responsible for checking if an `llvm::ReturnInst` must be removed. In our case, to satisfy the third property, such instruction must never be removed.

To remove all `llvm::UndefValue` and satisfy the fourth property, an optimization pass is run at the end of the skeletonizer. This last pass will additionally remove all code that was rendered dead due to the skeletonization process and is discussed in further details in Section 4.5.2.7.

The ratio between the number of instructions kept to the initial number of instructions is not directly correlated to the improvement in execution time. As the example in Section 4.4 shows, removing the entire body of a loop will cause the removal of the entire loop, and depending on the number of times this loop is executed, it can have a much larger impact than removing an instruction executed only once. Similary, removing a `CallInst` only accounts for one instruction removed, but the real number of instructions removed is that of the function not called.

#### 4.5.2.7 The LLVM IR Preparation

Finally, some preparations are now done to the IR before producing the skeleton. Firstly, to prevent the use of potentially uninitialized variables, all variables declared with `AllocaInst` are initialized to 0. Indeed, the skeletonizer might remove the initalization of some variables, as it can not distinguish initialization from any other computation. Thus, if the variable is later used as an offset, it might lead to SIGSEGV because the offset could have any numerical value. Initializing to 0 solves this issue.

Secondly, an optimization pass, `default<O3>`, is run on the IR. Indeed, the initial LLVM IR of the application is not optimized, to give the skeletonizer all possible opportunities to remove instructions. Once the skeletonizer is done, the optimization pass is run to align the performances of the skeleton with that of the original application. Most importantly, it eliminates all the `llvm::UndefValues` added by the skeletonizer where it decided that an instruction should be removed, and it will also has the added benefit of removing dead code potentially left with all the cut instructions.

Listing 4.6 shows the result of the skeletonization process on the code presented in Listing 4.5, if it were reverse engineered from LLVM IR to C source code. We can see that the initialization of the matrices $A$ and $B$ has been completely removed, both buffers will now contain uninitialized data. Similarly, the triple nested loops doing the matrix multiplication is missing, no computation involving the uninitialized data is done. Finally, the computation of the offset has been removed, but its value is used as the index of the elements to access within the buffers. As described in Section 4.5.2.7, the prepararation step added an instruction to initialize the value to 0, to avoid using uninitialized data as an offset.

#### 4.5.2.8 Limitations of the Skeletonizer

These steps make the skeletonizer able to handle a majority of codes produced for the HPC field. However, some code patterns or language features still limits its capacity to process every possible application, and they will be discussed in this section.

```c
#include <mpi.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define TRIES_PER_ROUND 1000
#define MAX_ROUNDS 1000
#define EPSILON 1E-5

int main() {
  int my_rank, i, rounds = 0;
  float x, sqrt2;
  long A = 0, B = 0;
  long world_A, world_B;

  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  srand(my_rank + time(NULL));

  do {
    for (i = 0; i < TRIES_PER_ROUND; i += 1) {
      x = (float)rand() / (float)(RAND_MAX/2);
      if (x * x < 2.0) {
        A += 1;
      }
    }
    B += TRIES_PER_ROUND;
    rounds += 1;

    MPI_Allreduce(&A, &world_A, 1, MPI_LONG, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce(&B, &world_B, 1, MPI_LONG, MPI_SUM, MPI_COMM_WORLD);

    sqrt2 = 2.0 * (float)world_A / (float)world_B;

  } while (rounds < MAX_ROUNDS && fabs(sqrt2 * sqrt2 - 2.0) > EPSILON);

  if (my_rank == 0) {
    printf("sqrt(2) ~ %f, found in %d rounds\n", sqrt2, rounds);
  }

  MPI_Finalize();
  return 0;
}
```

Listing 4.22: C code approximating $\sqrt{2}$

**Sending *Comm Variables* through MPI buffers** In Listing 4.22, we use the Monte-Carlo method to approximate the value of $\sqrt{2}$ to demonstrate one of the limitations of the skeletonizer: *Comm Variables* sent through MPI buffers. Indeed, the condition at Line 36 is a convergence criterion whose value dictates the number of loops to do. The

condition depends on the value of `sqrt2`, which depends on `world_A` and `world_B` which in turn have a communication dependency with `A` and `B` on Lines 31 and 32. As a result, all these variables are considered *Comm Variables*, which will turn practically all variables in the program into *Comm Variables*. This is an issue for the skeletonizer as it loses all possibility to remove any instruction, and the resulting skeleton is almost identical to the original application. The choice that was made is to not propagate the *Comm Variable* status through communication dependencies. In this case, it frees the skeletonizer from propagating the status to the variables `A` and `B`, thus allowing to remove the entire loop at Lines 22 to 28. However, the behavior of both applications might diverge, as some variables could evolve differently in the original application and in the skeleton. This discrepancy could possibly impact the communication pattern, as it is the case for the variable `sqrt2`, because it depends on variables sent through MPI buffers. Removing the propagation in these situations is still a reasonable choice to make as in most cases, the data sent through the buffers does not impact the communication pattern. The most common exception is a convergence critertion, as seen in the example, but these cases are usually protected with a static condition on the number of loops, condition that is preserved by the skeletonizer. However, it has the infortunate side effect of fixing the number of loops that the skeleton will perform, and thus requiring the user to also fix this number in the original application for them to have the same number of loops.

```c
1  #include <stdio.h>
2  #include <mpi.h>
3  #define ROOT 0
4
5  int global;
6
7  int main(void) {
8    MPI_Init(NULL, NULL);
9    int size; MPI_Comm_size(MPI_COMM_WORLD, &size);
10   int rank; MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11   if (rank == ROOT) {
12     global = 42;
13     MPI_Bcast(&global, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
14   } else {
15     MPI_Bcast(&global, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
16   }
17   printf("[%d] %d\n", rank, global);
18   MPI_Finalize();
19 }
```

Listing 4.23: C code using global variable

**Mutable global variables** are a notorious example of a poor code practice, their use is allowed but strongly discouraged. In the case of the skeletonizer, the `AliasAnalysis` discussed in Section 4.5.1 is not always able to detect to which global variable a pointer refers to, thus the skeletonizer is unable to know whether or not is should apply the *Comm Variable* status on a `LoadInst`. A naive but working solution would be to consider every global variable as being a *Comm Variable*, thus marking any load of a global object as being required. This approach was our initial solution, but was replaced by a new method, which features a much simpler implementation: if at any point LLVM assures that an instruction loads information from a mutable global object, the skeletonizer aborts, indicating that the use of global variable is not supported. It is a sensible path to take as the use of such a variable can be avoided in almost all situations.

**Function calls through function pointers** If the compiler is not able to know in advance which function will be called, then the skeletonizer is unable to correctly recurse and analyze the function called. This situation oftenly arise in C++ when inheritance is used. An LLVM pass that performs constant propagation could help in this situation: it is an optimization pass that moves computations from run-time to compile-time when possible. It mostly applies to variables that can be proven to be constants, in which case the pass substitutes the computations with the result at compile-time. It is, under some conditions, able to determine which function will be called through a pointer, thus proving the pointer to be constant, and modifies the IR accordingly. In this situation, the limitation is lifted, and that is why this pass is run before the skeletonization.

**Variadic functions** Similarly, the skeletonizer is unable to correctly mark the arguments of a function if the function has a variable number of arguments. Therefore, variadic functions are not supported. This choice is reasonable as their use in functions that perform MPI operation is rare.

**Exceptions** The C++ exceptions system, when used, require special instructions in the LLVM IR. Indeed, a function that can throw an exception is a function that the control flow can exit in two ways, through a regular return statement or through the exception system. Such a function requires various modifications to be called, such as a special call instruction (`llvm::InvokeInst`), along with a *landing pad*, a special label to go to in case of an exception. This renders the analysis considerably more difficult, especially if we consider that *Comm Variables* can be thrown through the exception mechanism. As a result, we use the LLVM pass `lower-invoke`, which transforms a code with exceptions to one without them, simply by removing any potential throws. It was originally designed for platforms that do not support stack unwinding, which is the process of rewinding back in the stack to run the destructor of still allocated elements in the case of an exception, as specified in the C++ standard. All exception-handling code becomes dead code after this pass. It is not a perfect solution, but it allows us to perform our tests on some C++ codebases that use exceptions, such as MiniFE [41].

## 4.6 Tuning Time Reduction

As shown in Figure 4.1, the objective is to use the automatically produced proxy application in place of the original application in the tuning process. To gain time during this optimization, we want to compare the performance of the skeleton relative to the application, along with the gains brought by the tuning in both cases.

To ease the reading of the following sections, we introduce several notations:

- skeleton($A$) : the application obtained after applying the process described in Section 4.3 to the application $A$
- $(A \mid B)$ : the execution time of application $A$ with the tuning produced using the application $B$.
- $(A \mid \varnothing)$ : the execution time of application $A$ without tuning (MCA parameters are thus set to their default values)

| | | |
|---|---|---|
| $\frac{(app\|\varnothing)}{(app\|app)}$ | $< 1$<br>$= 1$<br>$> 1$ | $\Longrightarrow$ degradation<br>$\Longrightarrow$ neutral tuning<br>$\Longrightarrow$ improvement |
| $\frac{(app\|\varnothing)}{(app\|\text{skeleton}(app))}$ | $< 1$<br>$= 1$<br>$> 1$ | $\Longrightarrow$ degradation<br>$\Longrightarrow$ neutral tuning<br>$\Longrightarrow$ improvement |
| $1 - \frac{(\text{skeleton}(app)\|\varnothing)}{(app\|\varnothing)}$ | $\approx 0$<br>$\approx 1$ | $\Longrightarrow$ moderate acceleration<br>$\Longrightarrow$ considerable acceleration |
| $\frac{(app\|app)}{(app\|\text{skeleton}(app))}$ | $\approx 0$<br>$\approx 1$ | $\Longrightarrow$ tuning not applicable<br>$\Longrightarrow$ tuning applicable |

Table 4.1: Expectations

The ratio $\frac{(app|\varnothing)}{(app|app)}$ represents the gain brought by an optimal tuning by comparing the application without tuning with its tuned version. We want this ratio to be at least strictly superior to 1, and as high as possible in the best case scenario. It also represents the upper limit for the next ratio, $\frac{(app|\varnothing)}{(app|\text{skeleton}(app))}$, which represents the gain brought by a tuning using $skeleton(app)$, the result that our approach aims to produce. The ratio $1 - \frac{(\text{skeleton}(app)|\varnothing)}{(app|\varnothing)}$ is the most interesting, it represents the fraction of the time gained by executing $skeleton(app)$ instead of $app$. The closer this ratio is to 1, the greater the time saved. Finally, the last ratio $\frac{(app|app)}{(app|\text{skeleton}(app))}$ indicates how the tuning produced using $skeleton(app)$ compares to the optimal tuning. For the best results, we want it to be as close as possible to 1. These expectations are summarized in Table 4.6

### 4.6.1 Workflow of the Tuning Process using Skeletonization

As described in Figure 4.1, we take the original application ($app$) and compile it to LLVM IR using `clang`, without optimizations. If $app$ makes use of C++ exceptions, the `lower-invoke` pass is also run on the IR. At this point, we can invoke the SkeletonPass on the IR to perform the skeletonization process and produce $skeleton(app)$. Then, the optimization pass $O2$ or $O3$ is run on the skeleton, as stated in Section 4.5.2.7. Finally, the resulting IR must be linked in the same way $app$ is, to produce the final binary that will execute the skeleton.

This binary, $skeleton(app)$, is then used in any tuning process, exactly how $app$ would be used to produce a tuning. As shown in Section 4.7.3.6, this step is significantly faster than if $app$ was used. In this case, we used the tuning process described in Section 4.3 to produce the tuning file relative to $skeleton(app)$. Finally, we can validate that the tuning file is correct by using it with $app$ and find that it reduces the execution time.

| Class | $(FT \mid \varnothing)$ (sec) | $\frac{(FT\|\varnothing)}{(FT\|FT)}$ | $\frac{(FT\|\varnothing)}{(FT\|\text{skeleton}(FT))}$ | $1 - \frac{(\text{skeleton}(FT)\|\varnothing)}{(FT\|\varnothing)}$ | $\frac{(FT\|FT)}{(FT\|\text{skeleton}(FT))}$ |
|---|---|---|---|---|---|
| **C** | 1.67 | 1.018 | 1.012 | 0.054 | 0.994 |
| **D** | 38.36 | 1.016 | 1.012 | 0.056 | 0.996 |
| **E** | 233.59 | 1.004 | 1.001 | 0.015 | 0.997 |

Table 4.2: FT benchmark, 32 nodes, 32 processes per node

## 4.7 Skeletonization Process Validation

### 4.7.1 Viability of the Approach

To prove the viability of the approach, a few short benchmarks were selected to manually perform the skeletonization process and assess the applicability and complexity of this technique. We selected the NAS Parallel Benchmarks [45] as our testing suite and have arbitrarily choosen the CG (Conjugate Gradient), FT (Fast Fourier Transform) and EP (Embarassingly Parallel) kernels for the experiment. Their behavior has been studied thoroughly [25, 38] and is therefore well-known, allowing a more extensive discussion of the results. Following the workflow detailled in Section 4.6.1, we evaluated the performances of the skeletons, and then used them in replacement of the original NAS applications in the tuning process. The results obtained with the skeleton manually created are detailled in Section 4.7.3.

### 4.7.2 Experimental Setup

For our experimentations, we used the hardware described in Section 3.7.1. They are two very different platforms, one uses Intel CPUs coupled with an OmniPath interconnect, while the other one uses AMD CPUs with a Mellanox interconnect. Having comparable results on both architectures will prove that the approach is hardware-agnostic.

### 4.7.3 Results Discussion

#### 4.7.3.1 FT

Results for the FT benchmark are presented in Table 4.2. FT is communication bound and does very little computation [25], therefore skeleton($FT$) behaves very similarly to FT, which explains the near-identical performance. Tuning the skeleton instead of the full application leads to similar tuned parameters values and thus does not degrade performance, but it does not reduce the exploration time to find those parameters either. If using the proposed skeletonization approach brings no real benefit in this case, it can still be employed on applications with a similar behavior as it includes no drawback either.

#### 4.7.3.2 EP

Results for the EP benchmark are presented in Table 4.3. EP is computation bound with very few communications performed [25]. That is why the skeleton appears to be so much faster. However, no tuning of its communications would heavily change the behavior of that type of applications, which makes the approach irrelevant in this case (which could be well detected by a static analysis). One can note however that when approaching an application without knowing its communication pattern, using our proposed skeletonization would

| Class | $(EP \mid \varnothing)$ (sec) | $\frac{(EP\mid\varnothing)}{(EP\mid EP)}$ | $\frac{(EP\mid\varnothing)}{(EP\mid\text{skeleton}(EP))}$ | $1-\frac{(\text{skeleton}(EP)\mid\varnothing)}{(EP\mid\varnothing)}$ | $\frac{(EP\mid EP)}{(EP\mid\text{skeleton}(EP))}$ |
|---|---|---|---|---|---|
| **A** | 0.03 | 1.500 | 1.001 | 0.667 | 0.667 |
| **B** | 0.08 | 1.143 | 1.143 | 0.875 | 1.000 |
| **C** | 0.33 | 1.179 | 1.139 | 0.970 | 0.966 |
| **D** | 4.50 | 1.011 | 1.004 | 0.998 | 0.993 |
| **E** | 72.08 | 1.011 | 1.003 | 1.000 | 0.992 |

Table 4.3: EP benchmark, 32 nodes, 48 processes per node

| Class | $(CG \mid \varnothing)$ (sec) | $\frac{(CG\mid\varnothing)}{(CG\mid CG)}$ | $\frac{(CG\mid\varnothing)}{(CG\mid\text{skeleton}(CG))}$ | $1-\frac{(\text{skeleton}(CG)\mid\varnothing)}{(CG\mid\varnothing)}$ | $\frac{(CG\mid CG)}{(CG\mid\text{skeleton}(CG))}$ |
|---|---|---|---|---|---|
| **C** | 2.55 | 1.045 | 1.024 | 0.200 | 0.980 |
| **D** | 55.32 | 1.066 | 1.040 | 0.637 | 0.976 |
| **E** | 576.93 | 1.055 | 1.053 | 0.789 | 0.998 |

Table 4.4: CG benchmark, 32 nodes, 32 processes per node

drastically reduce the exploration time, even if the resulting parameters values would not bring much benefit.

#### 4.7.3.3 CG

Results for the CG benchmark are presented in Table 4.4. CG is the most interesting case, which falls between the two previous ones [25]. skeleton($CG$) is significantly faster, taking only a fraction of the time of CG, therefore quickly yielding a tuning. Moreover, the tuning obtained performs nearly exactly like the tuning obtained using CG. The small difference (less than 0.5%) is explained by further analysis of the respective tuned parameters obtained with ShaMAN for CG and skeleton($CG$): they actually differ on the broadcast algorithm, but the broadcast function is only used once during the application. This explains why the ratio differs from 1, but the difference is negligible. Tuning using skeleton($CG$) instead of CG leads to a tuning with near identical performance and is much faster to obtain.

#### 4.7.3.4 MiniFE

MiniFE [41] is a mini-application using several MPI collective communication operations (`MPI_Allgather`, `MPI_Allreduce`, `MPI_Bcast`, and `MPI_Reduce`) to compute differential equations using the finite elements method. It is further detailled in Annexe A. We executed various configurations with different algorithms for these collectives, adjusting segment sizes and fan-in/out values. This led to a substantial reduction in execution time, averaging a 32% improvement.

Figure 4.5 displays results for a grid of $200^3$ and 64 processes (8 nodes with 8 processes per node). We projected the results to evaluate the effectiveness of the skeletonizer in identifying the best configuration for each (segment size, fan-in/out value) pair. In most cases, the skeletonizer accurately identified one of the optimal configurations of the original miniFE, with an average error of 1.4%. For detecting the worst configuration, the accuracy was slightly lower but generally within a 10% margin (6.5% on average).

Figure 4.5: MiniFE benchmark performance in diverse MPI configurations, `Pise` machine

#### 4.7.3.5 Lulesh

The Lulesh [36] benchmark simulates a hydrodynamics problem using a numerical method. It is further detailled in Annexe A.

We ran the Lulesh benchmarks in two setups: the original version and a skeletonized variant. These benchmarks were evaluated across diverse MPI configurations, including the number of processors (N), processes per node (PPN), and varying problem sizes (Pb_S), and were carried out on the `Pise` platform.

Figure 4.6 presents a comparison between timings obtained from the skeletonized version and the original Lulesh application, using different `MPI_AllReduce` algorithms. Our objective is to assess if the optimal ($\bigtriangledown$) and worst-case ($\bigtriangleup$) timings for each collective communication algorithm align across both versions.

Results show that the skeletonized version performs significantly faster than the original, achieving speedups of 7.5x, 28x, and 49x for problem sizes 30, 60, and 90, respectively. In most cases, the skeleton accurately identifies both the worst and best cases for MPI configurations. When discrepencies occur, they are generally within the noise margin. The main exception is for the combination (N=8, PPN=1, Pb_S=90), where the skeletonized configuration's optimal timing corresponds to the worst case in the original application.

To further assess the skeletonizer's reliability for configuration selection, we analyzed the error in selecting configurations based on the skeletonized version compared to the real application, as shown in Figure 4.7. Two cases are illustrated: "*min_ error*", which measures the discrepency when selecting the skeletonizer's best configuration relative to the actual best runtime of the original, and "*max_ error*", which demonstrates the skeletonizer's effectiveness in ruling out poor configurations.

In the majority of cases, error rates remain well below 10%, with many showing no discrepancy when using the skeletonized configuration. The only notable exception occurs with (N=8, PPN=1, Pb_S=90), where the skeletonizer's suggested worst configuration is approximately 10% off from the original's actual worst.

For clarity, the figures primarily focus on variations in collective communication algorithms used in Lulesh. In Figure 4.8, we explore an additional scenario in which both fan-in/out values and segment sizes for two collective communication operations (`MPI_Reduce` and `MPI_Allreduce`) are varied, focusing on the case (N=8, PPN=1, Pb_S=90). Here,

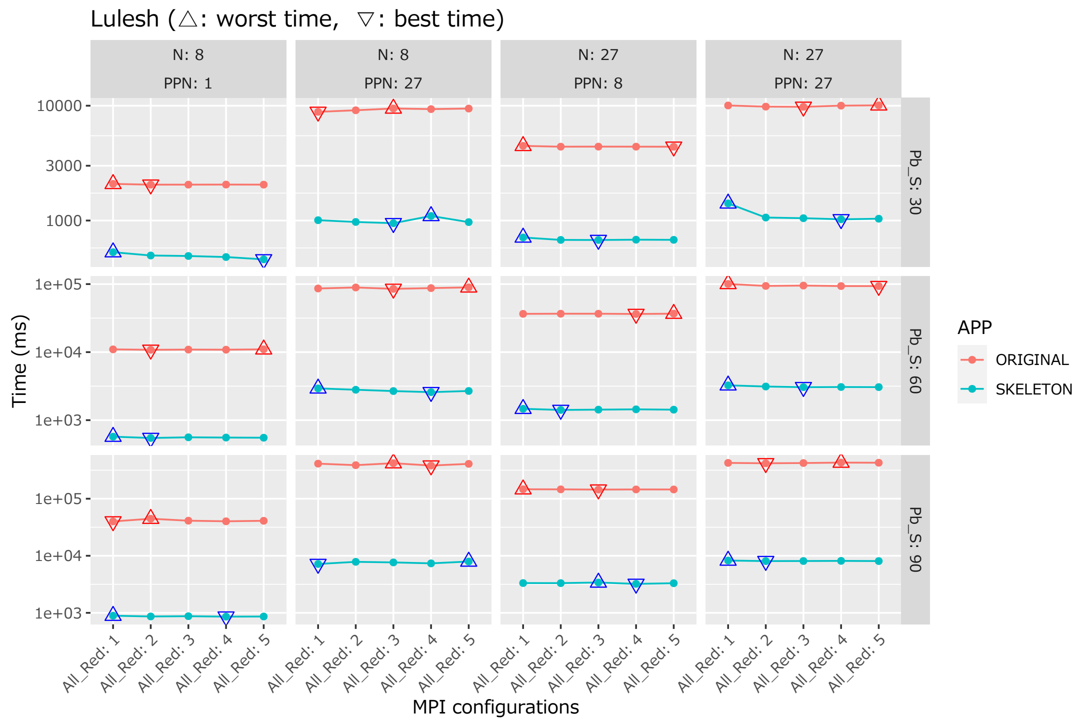



Figure 4.6: Lulesh benchmark performance in diverse MPI configurations, `Pise` machine

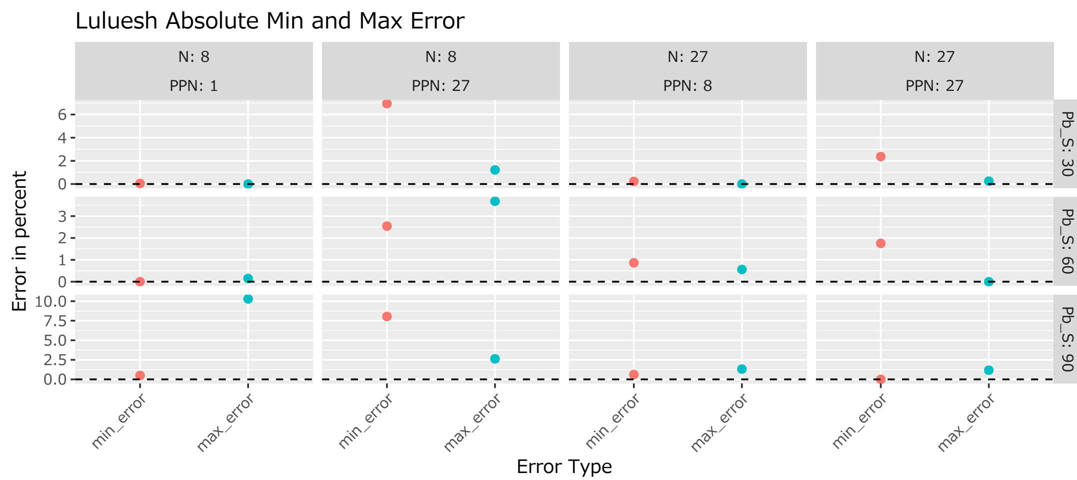



Figure 4.7: Relative error in Lulesh performance for the best and worst configuration

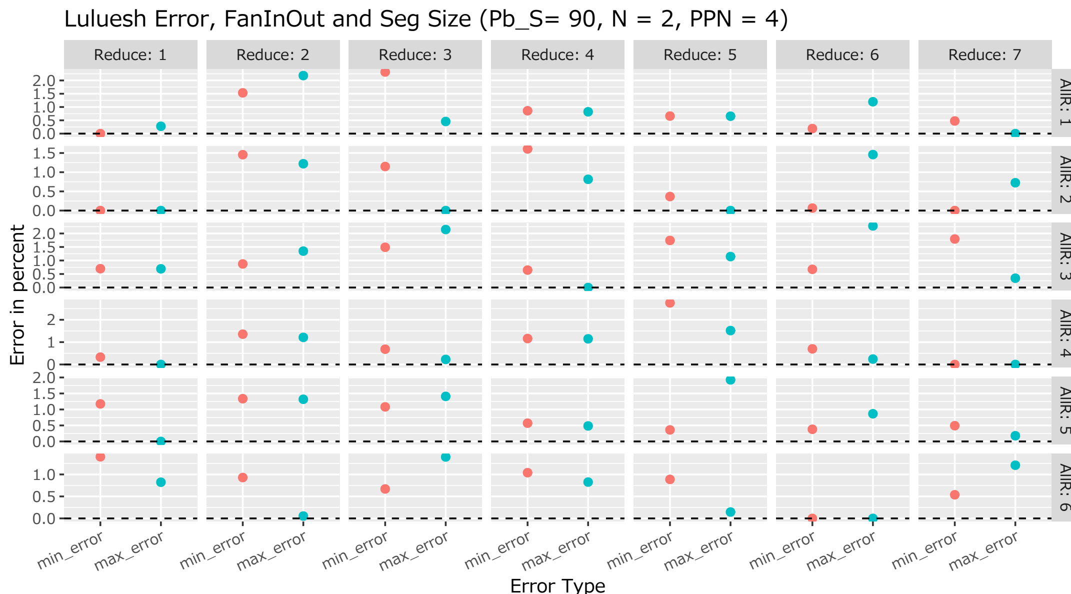



Figure 4.8: Lulesh benchmark, `Pise` machine

the error between the best skeletonized configuration and the best original configuration is consistently below 2.8%, while error in detecting the worst configuration remains under 2.5%. If skeletonization is employed to find optimal or avoid suboptimal configurations (considering a mix of collective algorithms, fan-in/out, and segment size), the error remains below 3% in both cases.

Finally, we conducted an experiment on the second platform, `Bora`, concentrating on the specific case (N=27, PPN=27, Pb_S=90), one of the largest test case. We exhaustively explored all the combinations of algorithms, fan-in/out values and segment sizes. Multiple plots in this figure illustrate the efficiency of the skeletonizer at identifying the worst configuration, highlighting consistent performance. It is especially visible in the case fan-in/out= 64 and segment size= 1048576, correctly identifying the spike in execution time with this configuration. Due to the flatness of the graphs, pinpointing the best configuration is a complicated process, but it can be observed that the selected configuration features similar performance of that of the best one. Unlike Figure 4.7, no notable irregularities were observed, confirming the stability of the skeletonizer on this platform. The results demonstrate a relative speedup of approximately 8.5x, further underscoring the effectiveness of the method in accelerating execution time while maintaining accurate configuration selection. This lesser speedup is a factor in the higher stability of the behavior on the `Bora` machine.

#### 4.7.3.6 Evaluation of the Skeleton

The results above demonstrate that the automatic skeletonization method proposed effectively simplifies non-trivial MPI programs while achieving significant execution speedups. This acceleration allows application tuning to be conducted much more rapidly without compromising quality. Indeed, in most cases, the MPI configuration that delivers the fastest execution time with the skeletonized version also ranks among the top-performing configurations for the original application. When discrepancies do occur, they typically arise because of the minimal differences between configurations, result of the high speedup

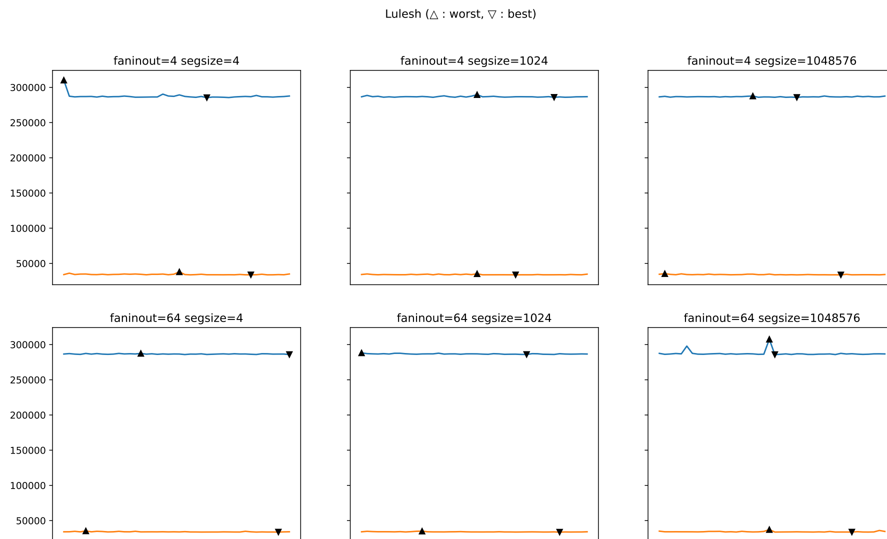



Figure 4.9: Lulesh benchmark, (N=27, PPN=27, Pb_S=90), `Bora` machine

brought by the skeleton.

The most important fact to validate about the resulting skeleton is to verify that it features the same communication pattern as the original application. This was verified on every automated test run using a built-in tool of Open MPI: the PML monitoring. As stated in Section 2.1, Open MPI has a modular architecture, which leaves us the choice of the PML (Point-to-point Management Layer) to use. The one we chose is the `monitoring` as it allow the user to query each MPI process for the number of messages and quantity of bytes sent to any other MPI process. All skeletons exhibited the exact same communication pattern as their original applications.

Additionally, one expected feature of the skeleton is its rapidity of execution relative to its source. As can be seen for all three NAS benchmarks, the ratio $1 - \frac{(\text{skeleton}(app)|\varnothing)}{(app|\varnothing)}$ is always strictly inferior to 1 regardless of *app*, which means that the execution time of $skeleton(app)$ always represents less than 100% of the execution time of *app*, thus implying that the skeleton always executes faster.

## 4.8 Conclusion

In this chapter, we presented a novel approach for the optimization of MPI applications through a process called skeletonization. By generating a simplified version of the original application which retains only the communication behavior, we have demonstrated a method to significantly reduce the time required for tuning MPI parameters.

We detailed an automatic process for the generation of these skeletons, highlighting the challenges of manually creating them. The LLVM-based implementation allows for consistent and accurate generation of skeletons, even for complex applications, by systematically removing unnecessary computations while preserving essential communication patterns.

Our experimental results have shown that the skeletons produced by our method are effective in maintaining the communication characteristics of the original applications while offering substantial reductions in execution time. This enables rapid exploration of the tuning space and application of optimizations that would otherwise be infeasible due to the high computational cost of the original application.

While the skeletonization process has proven effective for the set of tested applications, certain limitations remain, particularly when dealing with complex code structures such as recursive functions, or global variables. These limitations suggest areas for future work, including the development of more advanced analysis techniques and the expansion of the skeletonizer's capabilities.

Overall, the skeletonization approach provides a promising direction for optimizing MPI applications by offering a scalable and automated way to minimize the tuning overhead. This work lays the foundation for further research into automated optimization techniques, potentially extending beyond MPI to other parallel computing paradigms.

# Chapter 5

# Conclusion

## Contents

5.1 **Contributions** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.2 **Discussion** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
5.3 **Future Work** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.4 **Perspectives** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

High-Performance Computing platforms used to run parrallel applications grow more and more heterogeneous, and with it the need to optimize the MPI implementations that HPC rely upon. The current thesis is a part of the recent developpement aiming at fine tuning Open MPI, mostly focusing on two distinct approaches, Blackbox Optimization and Skeletonization.

## 5.1 Contributions

**Blackbox Optimization** One significant contribution lies in the development and application of blackbox optimization techniques for MPI parameter tuning, as detailed in Chapter 3. This approach models the MPI application as a black box, employing Bayesian Optimization to navigate the parameter space effectively without exhaustive exploration. The proposed optimization framework reduces the need for intensive computational resources by intelligently narrowing down possible configurations based on prior results. By incorporating Bayesian models, which iteratively improve predictions based on observed data, we achieve high-performance configurations with minimal tuning overhead. This is particularly beneficial for complex applications or architectures where traditional tuning methods are impractical. Moreover, the blackbox approach enables adaptability across different MPI implementations and architectures, making it a versatile solution for a range of HPC environments.

**Skeletonization** In Chapter 4, we introduced the skeletonization process, another core contribution of this thesis. Skeletonization generates a simplified version of an MPI application, referred to as its "skeleton," which retains only the communication characteristics of the original application while eliminating other computational complexities. By preserving the core communication pattern, the skeleton enables rapid performance tuning without the computational cost typically associated with full-scale application execution. This process leverages the LLVM compiler infrastructure for automated skeleton extraction,

ensuring consistency and accuracy in skeleton generation, even for large and intricate applications. Skeletonization thus provides a practical and scalable approach for MPI tuning, allowing for reduced overhead in the optimization process while maintaining fidelity to the communication behavior of the original application.

The skeletonization method also enhances usability by integrating seamlessly into existing software infrastructures, requiring no expertise in compiler theory or manual code modification. As a result, non-specialist users can apply sophisticated tuning techniques to their applications, which broadens the accessibility and potential impact of this work. This automation, combined with the high fidelity of skeletons to their original counterparts, ensures that performance optimizations derived from skeleton-based tuning are reliable and effective.

Collectively, the contributions in this thesis represent a significant advancement in the domain of MPI application tuning. By reducing tuning time and computational costs, the blackbox optimization and skeletonization techniques introduced here have the potential to enhance performance in various MPI applications, particularly those running on large-scale HPC systems. Additionally, these methods pave the way for future research, including dynamic adaptive tuning and the application of these techniques to other parallel computing paradigms. This work establishes a foundation for further exploration into automated tuning solutions that could ultimately contribute to more efficient and accessible HPC application optimization.

## 5.2 Discussion

The central problem tackled in this thesis was how to efficiently fine-tune MPI implementations to maximize performance across various applications and architectures. Both Blackbox Optimization and Skeletonization provided answers to this problem from different perspectives and brought substantial benefits by addressing key performance bottlenecks without relying on exhaustive manual tuning or requiring deep system-specific knowledge

The Blackbox Optimization approach developed in this thesis effectively manages the complexities of MPI parameter tuning across various architectures and application types. Its core strength lies in its agnostic approach, treating applications as opaque functions that can be optimized based solely on performance outputs. By applying Bayesian Optimization to MPI parameters, we achieve a performance tuning process that is adaptable and computationally efficient. Unlike traditional brute-force methods, BBO intelligently samples the parameter space, directing resources only towards configurations with high performance potential. This adaptability proves especially valuable in large-scale systems, where exhaustive tuning would be computationally prohibitive. Consequently, BBO is not only faster but also widely applicable across different types of MPI applications.

In parallel, the skeletonization approach offers a complementary solution by allowing rapid performance evaluations through simplified application models. The skeletons generated through the LLVM-based automation are particularly effective in accurately representing the communication patterns of the original application, which ensures that optimizations made on the skeleton translate directly to performance gains on the original application. Moreover, this skeletonization process enables scalable tuning on larger, more complex applications that would otherwise be impractical to optimize exhaustively.

While both BBO and skeletonization offer efficient alternatives to exhaustive tuning, each method has unique trade-offs. BBO achieves a broad applicability by treating applications as black boxes, making it a flexible solution for tuning without needing

detailed application-specific knowledge. However, skeletonization offers more speed by simplifying the application itself, which accelerates the tuning process but requires a careful balance to ensure that the skeleton retains enough fidelity to accurately represent the original application's performance dynamics.

The combination of these two methods thus creates a highly adaptable tuning framework: BBO provides a systematic way to approach optimization without application-specific modeling, and skeletonization accelerates the process by reducing the computational load of each tuning evaluation. Together, these methods contribute a versatile solution to the longstanding problem of efficiently tuning MPI applications on ever-evolving HPC systems.

## 5.3 Future Work

This thesis opened numerous research paths, and some of them could not be fully explored.

A promising avenue for future work involves leveraging Machine Learning models to further classify MPI applications based on their communication patterns, computational intensity, and hardware dependencies. Using ML for classification could help group applications with similar tuning requirements, potentially creating "tuning profiles" that can be reused across applications with comparable behavior. By clustering applications into relevant tuning classes, it may be possible to rapidly approximate optimal configurations for new applications by aligning them with an existing profile, reducing the tuning time and and the need for computational ressources.

Integrating dynamic adaptive tuning mechanisms is another exciting prospect. Dynamic tuning would involve continuous monitoring of application performance metrics during execution, allowing for dynamic adjustments to MPI parameters. Such an approach could react to fluctuations in workload or system states, adapting configurations to optimize performance. Implementing feedback loops that modify MPI parameters based on dynamic data could make tuning both more responsive and resilient, particularly for applications deployed in evolving HPC environments.

An automated framework encompassing BBO, skeletonization, and ML classification could serve as a powerful toolkit for MPI tuning. Such a framework would enable non-specialist users to automatically adjust configurations for optimal performance, democratizing access to sophisticated tuning techniques. Integrating these tools into a cohesive, user-friendly platform could facilitate broader adoption and make optimization more accessible to developers across various scientific and engineering domains.

In summary, the methodologies developed in this thesis lay a solid foundation for future research. By expanding machine learning applications, developing adaptive tuning mechanisms, and addressing the needs of heterogeneous systems, future work can further enhance the versatility and impact of MPI tuning techniques, pushing the boundaries of HPC performance optimization.

## 5.4 Perspectives

Looking ahead, the future of MPI tuning lies in the integration of more intelligent, automated systems. Both Blackbox Optimization and Skeletonization provide foundational steps toward this goal, but there remains significant potential for innovation.

Furthermore, as HPC systems continue to evolve towards more heterogeneous architectures, the ability to tune applications across diverse hardware environments will become increasingly important. The methods developed in this thesis provide a solid starting point for such advancements, offering scalable, adaptable, and efficient tuning solutions for the next generation of MPI applications. Future research will push MPI tuning techniques to new heights, enabling HPC systems to handle increasingly complex applications with greater efficiency, flexibility, and accessibility.

# Appendix A

# Benchmarks

## A.1 OSU

The OSU Micro Benchmarks (OMB) [47] suite is a comprehensive set of tests designed to evaluate the performance of key communication primitives in HPC environments, particularly those using MPI. Developed by the Ohio State University (OSU), OSU Micro Benchmarks provides fine-grained insights into the behavior of MPI implementations across different platforms. It includes a variety of benchmarks that measure latency, bandwidth, and message rate for point-to-point and collective communication operations. These benchmarks help researchers and system architects identify bottlenecks in network communication and optimize both hardware and software performance. Due to its focus on low-level communication patterns, the OSU Micro Benchmarks suite is widely used to evaluate the efficiency of MPI libraries, network hardware, and HPC system configurations, making it an essential tool in the fine-tuning of distributed applications.

## A.2 NAS

The NAS Parallel Benchmarks (NPB) [45] suite is a collection of programs developed by NASA Advanced Supercomputing (NAS) to evaluate the performance of parallel supercomputing systems. They are usually reffered to as the NAS benchmarks. These benchmarks simulate a variety of computational problems which are common in aerospace and scientific applications, making them highly relevant for real-world HPC workloads. The suite includes several benchmarks, each designed to test different aspects of system performance, such as computation, communication, and memory access. The benchmarks are implemented using multiple runtimes, including MPI, allowing them to assess the scalability and efficiency of shared and distributed memory systems. The NAS benchmarks provide a standardized way to compare the performance of HPC systems and optimize parallel algorithms, compilers, and hardware configurations. Their behavior and limitations have been thoroughly studied [25], allowing their results to offer researchers and system architects valuable insights for enhancing the performance of parallel applications.

## A.3 MiniFE

The miniFE [41] benchmark is a simplified finite element code that serves as a proxy application for testing and evaluating HPC systems. The finite element method (FEM) is a numerical technique used to solve complex partial differential equations (PDEs) by dividing a large problem domain into smaller, simpler subdomains called finite elements. By approximating the solution over these elements and assembling them, FEM provides an efficient way to model physical phenomena in engineering and scientific problems. MiniFE emulates the performance characteristics of more complex finite element analysis applications, allowing for the investigation of issues such as parallel scalability, memory access patterns, and communication efficiency in distributed systems. The benchmark leverages MPI to distribute its workload across multiple processors, with possible imbalances in workload, to solve a sparse linear system using the Conjugate Gradient method with an incomplete lower-upper (ILU) preconditioner. MiniFE provides a valuable framework for optimizing MPI implementations and hardware performance in scientific and engineering simulations.

## A.4 Lulesh

The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) [36] benchmark is a representative application for evaluating the performance of parallel computing systems, particularly in HPC environments. It simulates a shock hydrodynamics problem using a Lagrangian numerical method, which is common in scientific computing workloads. Designed by the Lawrence Livermore National Laboratory (LLNL), LULESH is widely used to test and optimize MPI implementations. Its structured yet computationally intensive nature makes it an ideal benchmark for assessing the scalability and efficiency of communication patterns, load balancing, and memory access across distributed systems. By tuning parameters like mesh size and decomposition, LULESH allows researchers to investigate performance bottlenecks and evaluate improvements in MPI-based systems, offering key insights into real-world HPC application behavior.

# Appendix B

# User Manual of the Skeletonizer

# Glossary

**communication dependency** $X$ has a *Communication dependency* on $Y$ if $X$ has a data dependency on $Y$ and the statement where $Y$ appears is an MPI communication operation. 50, 69

**control dependency** $X$ has a *Control dependency* on $Y$ if the value of variable $Y$ determines whether or not the statement containing variable $X$ is executed. 50

**data dependency** $X$ has a *Data dependency* on $Y$ if the program's computation might change if/when the statements where $X$ and $Y$ appear are reversed. 50, 67, 69

**DGEMM** In the BLAS naming convention: D=double GE=general matrix MM=matrix-matrix product. 52

**program slice** the minimal subset of statements that preserves the behavior of the original program up to the statement $p$ with respect to the program variables in $V$. 48, 50

**SIGSEGV** Signal sent by the operating system to an offending program when it attempts an invalid memory access. more commonly known as segfault. 64, 65, 69, 70

**slicing criterion** a pair $\langle p, V \rangle$, where $p$ is a statement in $P$, and $V$ is a subset of the variables in $P$. 50

**TOP500** Project ranking the 500 most powerful (in FLOP/s) non-distributed computers in the world. 13, 14, 16

# Acronyms

**ACCO** ATOS Collective Communications Optimizer. 11, 34, 35, 45

**BLAS** Basic Linear Algebra Subprograms. 52

**CPU** Central Processing Unit. 9, 13, 14

**FLOP/s** Floating-Point Operations per Second. 13

**FPGA** Field-Programmable Gate Arrays. 9, 14

**GPU** Graphics Processing Unit. 9, 13, 14

**HPC** High-Performance Computing. 6, 8–13, 15–17, 27, 30, 34, 35, 45, 71, 82–86

**LLNL** Lawrence Livermore National Laboratory. 86

**LULESH** Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics. 86

**MCA** Modular Component Architecture. 17–24, 26, 30, 31, 45

**ML** Machine Learning. 26, 27, 84

**MPI** Message Passing Interface. 5, 7–12, 15–19, 21, 24, 26, 27, 29–31, 34–37, 39, 41, 43, 45, 47–51, 54, 58, 64, 65, 71–73, 76, 77, 79–86

**NAS** NASA Advanced Supercomputing. 25, 75, 80, 85

**NPB** NAS Parallel Benchmarks. 25, 75, 85

**NUMA** Non-Uniform Memory Access. 27

**OMB** OSU Micro Benchmarks. 11, 31, 34, 35, 85

**OOM** Out Of Memory. 64

**OSU** Ohio State University. 31, 34, 35, 85

**PlaFRIM** Plateforme Fédérative pour la Recherche en Informatique et Mathématiques. 6

**ShaMAN** Smart HPC Application MANager. 11, 34, 35, 45

# References


[1] Behzad et al. "Taming Parallel I/O Complexity with Auto-tuning". In: *SuperComputing*. SC '13. Denver, Colorado, 2013, 68:1–68:12.

[2] Li et al. "CAPES: Unsupervised Storage Performance Tuning Using Neural Network-based Deep Reinforcement Learning". In: *SuperComputing*. 2017.

[3] Luo et al. "HAN: a Hierarchical AutotuNed Collective Communication Framework". In: *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 2020, pp. 23–34. DOI: 10.1109/CLUSTER49012.2020.00013.

[4] M. Wilkins et al. "ACCLAiM: Advancing the practicality of MPI collective communication autotuning using machine learning". In: *IEEE Cluster 2022*. IEEE. 2022, pp. 161–171.

[5] Pješivac-Grbović et al. "Performance Analysis of MPI Collective Operations". In: *Clust. Comput.* Vol. 2005. Jan. 2005. DOI: 10.1109/IPDPS.2005.335.

[6] Subramoni et al. "Design and Evaluation of Network Topology-/Speed- Aware Broadcast Algorithms for InfiniBand Clusters". In: *Proceedings - IEEE International Conference on Cluster Computing, ICCC*. Sept. 2011, pp. 317–325. DOI: 10.1109/CLUSTER.2011.43.

[7] Tu et al. "Multi-core aware optimization for MPI collectives". In: *ICCC*. Sept. 2008, pp. 322–325. DOI: 10.1109/CLUSTR.2008.4663789.

[8] Wilkins et al. "A FACT-based Approach: Making Machine Learning Collective Autotuning Feasible on Exascale Systems". In: *2021 Workshop on Exascale MPI (ExaMPI)*. 2021, pp. 36–45. DOI: 10.1109/ExaMPI54564.2021.00010.

[9] Jean-Claude André et al. "High-Performance Computing for Climate Modeling". In: *Bulletin of the American Meteorological Society* 95.5 (2014), ES97–ES100.

[10] P. Balaprakash et al. "Autotuning in High-Performance Computing Applications". In: *Proceedings of the IEEE* 106.11 (2018), pp. 2068–2083. URL: https://ieeexplore.ieee.org/abstract/document/8423171.

[11] B. Behzad et al. "Pattern-driven parallel I/O tuning". In: *Proceedings of the 10th Parallel Data Storage Workshop*. Nov. 2015, pp. 43–48.

[12] Mariana Belgiu and Lucian Drăguţ. "Random forest in remote sensing: A review of applications and future directions". In: *ISPRS Journal of Photogrammetry and Remote Sensing* 114 (2016), pp. 24–31. ISSN: 0924-2716. DOI: https://doi.org/10.1016/j.isprsjprs.2016.01.011. URL: https://www.sciencedirect.com/science/article/pii/S0924271616000265.

[13] J. Bergstra, N. Pinto, and D. Cox. "Machine learning for predictive auto-tuning with boosted regression trees". In: *2012 Innovative Parallel Computing (InPar)*. 2012, pp. 1–9.

[14] Richard S Bird. "Lectures on constructive functional programming". In: *Constructive Methods in Computing Science: International Summer School directed by FL Bauer, M. Broy, EW Dijkstra, CAR Hoare*. Springer, 1989, pp. 151–217.

[15] François Broquedis et al. "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications". In: *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. Ed. by IEEE. Pisa, Italy, Feb. 2010. DOI: 10.1109/PDP.2010.67. URL: https://inria.hal.science/inria-00429889.

[16] Z. Cao et al. "Towards Better Understanding of Black-box Auto-tuning: A Comparative Analysis for Storage Systems". In: *USENIX*. USENIX ATC '18. 2018, pp. 893–907.

[17] M Chaarawi et al. "A Tool for Optimizing Runtime Parameters of Open MPI". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 210–217. ISBN: 978-3-540-87475-1.

[18] S. Chodnekar et al. "Towards a communication characterization methodology for parallel applications". In: *Proceedings Third International Symposium on High-Performance Computer Architecture*. 1997, pp. 310–319. URL: https://doi.org/10.1109/HPCA.1997.569693.

[19] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. "The münster skeleton library muesli: A comprehensive overview". In: (2009).

[20] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[21] V. Dalibard, M. Schaarschmidt, and E. Yoneki. "BOAT: Building Auto-Tuners with Structured Bayesian Optimization". In: *the 26th International Conference on World Wide Web (WWW'17)*. 2017, pp. 479–488.

[22] Alexandre Denis and François Trahay. "MPI Overlap: Benchmark and Analysis". In: *International Conference on Parallel Processing*. 45th International Conference on Parallel Processing. Philadelphia, United States, Aug. 2016. URL: https://inria.hal.science/hal-01324179.

[23] D. Desani et al. "Black-Box Optimization of Hadoop Parameters Using Derivative-Free Optimization". In: *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. 2016, pp. 43–50.

[24] *EVIDEN software suite*. URL: https://eviden.com/solutions/advanced-computing/bullsequana-ai/.

[25] Ahmad Faraj and Xin Yuan. "Communication characteristics in the NAS parallel benchmarks". In: *IASTED PDCS*. Citeseer. 2002, pp. 724–729.

[26] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 97–104. URL: https://link.springer.com/chapter/10.1007/978-3-540-30218-6_19.

[27] Hu Ge et al. "Molecular dynamics-based virtual screening: accelerating the drug discovery process by high-performance computing". In: *Journal of chemical information and modeling* 53.10 (2013), pp. 2757–2764.

[28] Horacio González-Vélez and Mario Leyton. "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers". In: *Software: Practice and Experience* 40.12 (2010), pp. 1135–1160.

[29] R.B. Gramacy. *Surrogates: Gaussian Process Modeling, Design, and Optimization for the Applied Sciences*. Chapman & Hall/CRC Texts in Statistical Science. CRC Press, 2020. URL: https://books.google.fr/books?id=F9LVDwAAQBAJ.

[30] William Gropp et al. "A high-performance, portable implementation of the MPI message passing interface standard". In: *Parallel Computing* (1996), pp. 789–828. URL: https://www.sciencedirect.com/science/article/pii/0167819196000245.

[31] Junyan Hu et al. "Voronoi-Based Multi-Robot Autonomous Exploration in Unknown Environments via Deep Reinforcement Learning". In: *IEEE Transactions on Vehicular Technology* 69.12 (2020), pp. 14413–14423. DOI: 10.1109/TVT.2020.3034800.

[32] S Hunold and S Steiner. "OMPICollTune: Autotuning MPI Collectives by Incremental Online Learning". In: *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2022, pp. 123–128. DOI: 10.1109/PMBS56514.2022.00016.

[33] F. Hutter, L. Kotthoff, and J. Vanschoren, eds. *Automated Machine Learning: Methods, Systems, Challenges*. en. The Springer Series on Challenges in Machine Learning. Cham: Springer, 2019. ISBN: 978-3-030-05317-8. DOI: 10.1007/978-3-030-05318-5. (Visited on 02/24/2021).

[34] *Intel mpitune*. URL: https://www.intel.com/content/www/us/en/docs/mpi-library/developer-reference-linux/2021-8/mpitune.html.

[35] P. Jamshidi and G. Casale. "An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems". In: *CoRR* (2016).

[36] Ian Karlin, Jeff Keasler, and J Robert Neely. *Lulesh 2.0 updates and changes*. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2013.

[37] A Keane and A Sóbester. "Engineering Design via Surrogate Modelling". In: John Wiley & Sons, Ltd, 2008. Chap. 1, pp. 1–31.

[38] JunSeong Kim and David J. Lilja. "Characterization of communication patterns in message-passing parallel scientific application programs". In: *Network-Based Parallel Computing Communication, Architecture, and Applications*. Ed. by Dhabaleswar K. Panda and Craig B. Stunkel. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 202–216. URL: https://link.springer.com/chapter/10.1007/BFb0052218.

[39] P. Knijnenburg, T. Kisuki, and M. O'Boyle. "Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation". In: *The Journal of Supercomputing* 24 (Jan. 2003), pp. 43–67.

[40] 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.* 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[41] Paul T Lin et al. "Assessing a mini-application as a performance proxy for a finite element method engineering application". In: *Concurrency and Computation: Practice and Experience* 27.17 (2015), pp. 5374–5389.

[42] M. D. McKay, R. J. Beckman, and W. J. Conover. "A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code". In: *Technometrics* 21.2 (1979), pp. 239–245. DOI: 10.2307/1268522.

[43] T. Miyazaki, I. Sato, and N. Shimizu. "Bayesian Optimization of HPC Systems for Energy Efficiency". In: *High Performance Computing*. Springer International Publishing, 2018, pp. 44–62.

[44] T. Miyazaki, I. Sato, and N. Shimizu. "Bayesian Optimization of HPC Systems for Energy Efficiency". In: *High Performance Computing*. Springer International Publishing, 2018, pp. 44–62.

[45] *NAS Parallel Benchmarks*. URL: https://www.nas.nasa.gov/software/npb.html.

[46] NICULESCU. "On the Impact of High Performance Computing in Big Data Analytics for Medicine". In: *Applied Medical Informatics* 42.1 (Mar. 2020), pp. 9–18. URL: https://ami.info.umfcluj.ro/index.php/AMI/article/view/766.

[47] *OSU Micro-Benchmarks*. URL: http://mvapich.cse.ohio-state.edu/benchmarks/.

[48] Simon Portegies Zwart. "The ecological impact of high-performance computing in astrophysics". In: *Nature Astronomy* 4.9 (2020), pp. 819–822.

[49] C. Rasmussen and C. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005. ISBN: 026218253X.

[50] S. Robert, S. Zertal, and P. Couvee. "SHAMan: A Flexible Framework for Auto-tuning HPC Systems". In: *MASCOTS 2020, Nice, France*. 2020. DOI: 10.1007/978-3-030-68110-4_10.

[51] S. Robert et al. "A comparative study of black-box optimization heuristics for online tuning of high performance computing I/O accelerators". In: *Concurrency and Computation: Practice and Experience* (2021). DOI: https://doi.org/10.1002/cpe.6274.

[52] Sophie Robert. "auto-tuning of computer systems using black-box optimization : an application to the case of i/o accelerators". Theses. Université Paris-Saclay, Nov. 2021. URL: https://theses.hal.science/tel-03507465.

[53] J. Sacks et al. "Design and Analysis of Computer Experiments". In: *Statistical Science* 4.4 (1989), pp. 409–423. DOI: 10.1214/ss/1177012413.

[54] Isaac Sánchez Barrera et al. "Modeling and optimizing NUMA effects and prefetching with machine learning". In: *Proceedings of the 34th ACM International Conference on Supercomputing*. ICS '20. Barcelona, Spain: Association for Computing Machinery, 2020. ISBN: 9781450379830. DOI: 10.1145/3392717.3392765. URL: https://doi.org/10.1145/3392717.3392765.

[55] K. Seymour, H. You, and J. Dongarra. "A comparison of search heuristics for empirical code optimization". In: *2008 IEEE International Conference on Cluster Computing*. 2008, pp. 421–429.

[56] Yun Su et al. "Computing infrastructure construction and optimization for high-performance computing and artificial intelligence". In: *CCF Transactions on High Performance Computing* (2021), pp. 1–13.

[57] Haruto Tanno and Hideya Iwasaki. "Parallel skeletons for variable-length lists in sketo skeleton library". In: *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings 15*. Springer. 2009, pp. 666–677.

[58] R. Thakur, R. Rabenseifner, and W. Gropp. "Optimization of Collective Communication Operations in MPICH". In: *Int. J. High Perform. Comput. Appl.* 19.1 (Feb. 2005), pp. 49–66. ISSN: 1094-3420. DOI: 10.1177/1094342005051521.

[59] *The LLVM Compiler Infrastructure*. URL: https://llvm.org/.

[60] *Top 500*. URL: https://top500.org/.

[61] Marco Vanneschi. "The programming model of ASSIST, an environment for parallel and distributed portable applications". In: *Parallel computing* 28.12 (2002), pp. 1709–1732.

[62] Shuhei Watanabe. "Tree-structured parzen estimator: Understanding its algorithm components and their roles for better empirical performance". In: *arXiv preprint arXiv:2304.11127* (2023).

[63] Mark Weiser. "Program Slicing". In: *IEEE Transactions on Software Engineering* SE-10.4 (1984), pp. 352–357. URL: https://doi.org/10.1109/TSE.1984.5010248.

[64] Brian J. N. Wylie et al. "Performance analysis of Sweep3D on Blue Gene/P with the Scalasca toolset". In: *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 2010, pp. 1–8. DOI: 10.1109/IPDPSW.2010.5470816.

[65] Andy B Yoo, Morris A Jette, and Mark Grondona. "Slurm: Simple linux utility for resource management". In: *Workshop on job scheduling strategies for parallel processing*. Springer. 2003, pp. 44–60.

[66] Jidong Zhai, Wenguang Chen, and Weimin Zheng. "PHANTOM: Predicting Performance of Parallel Applications on Large-Scale Parallel Machines Using a Single Node". In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, 2010, pp. 305–314. URL: https://doi.org/10.1145/1693453.1693493.

[67] Jidong Zhai et al. "FACT: Fast Communication Trace Collection for Parallel Applications through Program Slicing". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. Association for Computing Machinery, 2009. URL: https://doi.org/10.1145/1654059.1654087.

# Publications

## Articles in Peer-reviewed Journals

- Emmanuel Jeannot, Pierre Lemarinier, Guillaume Mercier, Sophie Robert-Hayek, Richard Sartori. Application-Agnostic Auto-tuning of Open MPI Collectives using Bayesian Optimization. In IWAPT 2024 - International Workshop on Automatic Performance Tuning
- Quentin Buot, Emmanuel Jeannot, Pierre Lemarinier, Guillaume Mercier, Richard Sartori. Automatic Skeletonization of MPI Applications Applied for Accelerating Tuning. In IPDPS 2025 - International Parallel & Distributed Processing Symposium. *Submitted*