Informatics

# Online Algorithm Selection of MPI Collective Communication Operations

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering und Internet Computing**

eingereicht von

**Sebastian Steiner**
Matrikelnummer 11777731

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dr. Sascha Hunold

Wien, 1. Mai 2023

Sebastian Steiner    Sascha Hunold

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Informatics

# Online Algorithm Selection of MPI Collective Communication Operations

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering and Internet Computing**

by

**Sebastian Steiner**
Registration Number 11777731

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dr. Sascha Hunold

Vienna, 1st May, 2023

Sebastian Steiner    Sascha Hunold

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Erklärung zur Verfassung der Arbeit

Sebastian Steiner

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Hiermit mache ich kenntlich, dass ChatGPT-4 zur Umformulierung häufig gebrauchter Phrasen an diversen Stellen dieser Arbeit angewendet wurde.

Wien, 1. Mai 2023

Sebastian Steiner

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Acknowledgements

Working on this thesis was quite a process with many steps, large and small, all of which crucial to the final outcome. The person most responsible for this positive result is my advisor, Prof. Sascha Hunold. I am deeply indebted and grateful for having had the opportunity to work together on such an enticing project. These past months were shaped by his excitement when new and unexpected turns lead the work along. Prof. Hunold found the time in his tightly packed schedule to discuss ongoing progress and to provide guidance for tackling problems whenever they emerged.

This thesis has given me opportunities I could not have seen coming, such as traveling to Texas and attending a presentation of this work, being able to discuss this and expand on my interests. Thank You, Sascha, for your continued support, your patience, and everything else over these last months.

I also want to thank my family and friends who had tremendous endurance and supported me when there was little else I would talk about but this thesis.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Kurzfassung

Das Message Passing Interface (MPI) stellt die Basis für den Großteil an Software, die für parallele Computercluster oder Supercomputer in verschiedenen Disziplinen erstellt wird. Operation mit kollektiver Kommunikation machen einen großen Anteil der Laufzeit von MPI-Anwendungen aus. Die Mehrheit von kollektiven Operationen können durch mehrere Algorithmen umgesetzt werden, wobei in der Regel kein einzelner Algorithmus für jede Eingabe die beste Leistung erzielt.Gängige MPI Bibliotheken inkludieren eine große Menge an Algorithmen für jede kollektive Operation, sowie eine Entscheidungslogik zur Auswahl. Die Standardauswahl erzielt jedoch häufig eine unzureichende Leistung und bietet somit Verbesserungspotenzial.

Aktuelle Ansätze nutzen typischerweise einen Tuning-Schritt, um die Entscheidungslogik an eine spezifische Maschine und Auslastung anzupassen. Ein Offline-Tuning-Ansatz leidet unter zwei Nachteilen: 1) einer potenziell lang andauernden Tuning-Phase und 2) der Notwendigkeit, im Voraus festzulegen, welche kollektiven Parameterfälle zu messen sind.

Um diese Einschränkungen zu beheben, schlagen wir in dieser Arbeit einen Autotuner mit niedrigem Overhead vor, der in die Ausführung von MPI Anwendungen integriert werden soll. Durch abfragen eines leichtgewichtigen Modell zur Algorithmenauswahl, wird ein Algorithmus abhängig seiner vorhergesagten Leistung zufällig gewählt. Die Laufzeit von kollektiven Operation wird mitgeschrieben und in periodischen Abständen genutzt, um ein Modell zur Laufzeitvorhersage für jeden Algorithmus zu trainieren. Anschließend werden die Modelle zur Algorithmenauswahl, basierend auf den Laufzeitvorhersagemodellen, aktualisiert und in die nächste Runde MPI Anwendungen eingefügt.

Wir demonstrieren die Anwendbarkeit dieses Auto-Tuners in einer quantitativen Studie und erreichen Leistungssteigerungen durch Tuning von ECP Proxy-Anwendungen an zwei unterschiedlichen Computerclustern.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Abstract

The Message Passing Interface (MPI) underlies most software run on large parallel machines or supercomputers, facilitating the development of highly parallel applications across various fields. Collective communication operations make up a large fraction of the runtime of MPI applications. These collective operations are generally implemented in multiple different algorithms, off which no single one performs the best for all inputs. Common MPI libraries include a large set of algorithms for each collective operation and decision logic for selection. The default selection, however, frequently underperforms, leaving room for performance improvements.

State-of-the-art approaches commonly apply an offline tuning step, to adapt the selection logic to a specific machine and workload. An offline approach suffers from two main drawbacks: 1) a potentially long-running tuning step and 2) the necessity to predefine the collective cases to be tuned.

To address these limitations, this thesis proposes a low-overhead online auto-tuner to be injected into the execution of MPI applications. Algorithms are selected randomly based on their predicted runtime, as indicated in the lightweight algorithm selection models. The actual runtime of collective operations is recorded and periodically used to train a runtime prediction model for each algorithm. Subsequently, these prediction models are used to update the algorithm selection models, which are injected into the next MPI applications.

We demonstrate the feasibility of this auto-tuner in a quantitative study, achieving performance improvements through tuning ECP proxy applications on two distinct compute clusters.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Contents

**List of Figures** xv

**List of Tables** xvii

**List of Listings** xix

**Nomenclature** xxi

**1 Introduction** 1
1.1 Goals of this Thesis . . . 2
1.2 Thesis Outline . . . 3

**2 Background** 5
2.1 System Architecture of Supercomputers . . . 5
2.1.1 Shared Memory Machine . . . 6
2.1.2 Distributed Memory Machine . . . 7
2.1.3 Programming Distributed Systems . . . 8
2.2 Message Passing Interface . . . 8
2.2.1 Message Passing Programming Paradigm . . . 9
2.2.2 Point-to-Point Communication . . . 9
2.2.3 Collective Communication . . . 9
2.2.4 Algorithms for MPI Collective Communications . . . 11
2.2.5 Collective Communication Algorithm Selection in MPI libaries 12
2.3 Algorithm Selection Problem . . . 13
2.3.1 Formal Problem Definition . . . 13
2.3.2 Meta-Learning . . . 14

**3 Problem Statement and Related Work** 17
3.1 Anatomy of Algorithm Selection for MPI Collective Communication . 18
3.2 Algorithm Selection Process for `MPI_Allreduce` . . . 19
3.2.1 Theoretical Analysis of `MPI_Allreduce` Algorithm Runtimes 19
3.2.2 Experimental Runtime Comparison of `MPI_Allreduce` Algorithms . . . 20
3.3 Algorithm Configuration Problem for MPI Collective Communication 21



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.4 State of the Art in Algorithm Selection . . . . . . . . . . . . . . . . . 23

4 **Iterative Online Tuning Approach** **25**
4.1 Architecture of Batched Algorithm Selection . . . . . . . . . . . . . . . 26
4.1.1 Online Algorithm Selection . . . . . . . . . . . . . . . . . . . . . 27
4.1.2 Offline Runtime Prediction Model Building . . . . . . . . . . . 27
4.2 Defining the Parameter Space for Algorithm Selection . . . . . . . . . 28
4.3 Defining a Runtime Metric for Collective Communication Operations . 29
4.4 Merging the ACP into the ASP . . . . . . . . . . . . . . . . . . . . . . 30
4.5 Collective Algorithm Performance Models . . . . . . . . . . . . . . . . 31
4.5.1 Regression Models for Collective Runtime Prediction . . . . . . 31
4.5.2 Deriving a Selection Model for Runtime Usage . . . . . . . . . 33
4.6 Online Algorithm Selection . . . . . . . . . . . . . . . . . . . . . . . . 34
4.6.1 Selecting an Algorithm Performance Vector . . . . . . . . . . . 35
4.6.2 Algorithm Selection Logic applied on Performance Vectors . . . 36
4.7 Application Overhead from Online Algorithm Selection . . . . . . . . . 37

5 **Experimental Evaluation** **41**
5.1 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.2 Collective Calls to Tune . . . . . . . . . . . . . . . . . . . . . . . . . . 41
5.3 Benchmark Applications to Trace . . . . . . . . . . . . . . . . . . . . . 42
5.4 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
5.4.1 Establishing a Baseline of the Maximum Attainable Performance Improvement . . . . . . . . . . . . . . . . . . . . . . . . . . 45
5.4.2 Comparing Instruction Overhead between Selection Logics . . . 47
5.4.3 Tuning miniAMR from different miniAMR cases on *Hydra* . . . 49
5.4.4 Tuning miniAMR from different miniAMR cases on *Irene* . . . 51
5.4.5 Tuning miniAMR from application group on *Irene* . . . . . . . 52
5.4.6 Transferring Learning between Compute Clusters . . . . . . . . 52
5.4.7 Simulating the Auto-Tuning Workflow . . . . . . . . . . . . . . 53

6 **Conclusion** **55**

A **Algorithm Mapping** **57**

B **Proxy Application Configurations** **61**

C **Building and Executing Custom Open MPI** **65**
C.1 Building Open MPI and the mpi_coll_tuner . . . . . . . . . . . . . . . 65
C.2 Slurm Job Template . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
C.3 *Irene* Job Template . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

**Bibliography** **69**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# List of Figures


1.1 Algorithm comparison for miniAMR on $32 \times 32$ procs. . . . . . . . . . . . 2
1.2 Individual algorithm comparison on *Hydra* . . . . . . . . . . . . . . . . . . . . 2

2.1 Simplified single-processor and dual-processor systems . . . . . . . . . . . 6
2.2 Connecting a distributed memory system using an interconnect . . . . . . 8
2.3 Visualization of select MPI collective operations . . . . . . . . . . . . . . . . 10
2.4 Visualization of Allreduce algorithms . . . . . . . . . . . . . . . . . . . . . . 11

3.1 Simplified parameter space partitioned by best algorithm . . . . . . . . . 18
3.2 Comparing Allreduce algorithm performances on *Hydra* . . . . . . . . . . 20
3.3 Visualization of message segmentation and chain algorithms . . . . . . . . 22
3.4 Speed-Up of pipelined Bcast algorithm using segmentation . . . . . . . . 22

4.1 Architecture of our novel auto-tuner system. . . . . . . . . . . . . . . . . . . 26
4.2 Performance impact of changing single parameters . . . . . . . . . . . . . 28
4.3 Various potential algorithm start/end time behaviors . . . . . . . . . . . . 31
4.4 Distribution of parameter values . . . . . . . . . . . . . . . . . . . . . . . . 33
4.5 Sampling from parameter space for lightweight algorithm model . . . . . 33
4.6 3D Parameter space containing probability distributions . . . . . . . . . . 36
4.7 Subsampling MPI calls and processes . . . . . . . . . . . . . . . . . . . . . . 38

5.1 Fraction of collective runtime per proxy application . . . . . . . . . . . . . 44
5.2 Comparing Allreduce algorithm performances on *Hydra* and *Irene* . . . . 44
5.3 Comparing Bcast algorithm performance on *Hydra* . . . . . . . . . . . . . 45
5.4 Default/Best found Allreduce algorithm on *Hydra* . . . . . . . . . . . . . 45
5.5 Default/Best found Bcast algorithm on *Hydra* . . . . . . . . . . . . . . . . 46
5.6 Default/Best found Allreduce algorithm on *Irene* . . . . . . . . . . . . . . 46
5.7 Default/Best found Bcast algorithm on *Irene* . . . . . . . . . . . . . . . . 47
5.8 Algorithm selection overhead for Allreduce . . . . . . . . . . . . . . . . . . 49
5.9 Algorithm selection overhead for Bcast . . . . . . . . . . . . . . . . . . . . . 49
5.10 Auto-tuning progress on *Hydra* . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.11 Speed-Up of auto-tuned algorithm over default on *Hydra* . . . . . . . . . 50
5.12 Auto-tuning progress on *Irene* . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.13 Auto-tuning progress from an application group on *Irene* . . . . . . . . . 52
5.14 Performance portability analysis on *Hydra* for Allreduce . . . . . . . . . . 52




Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.15 Performance portability analysis on *Hydra* for Bcast . . . . . . . . . . . . . 53

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# List of Tables

2.1 Number of collective algorithms in popular MPI implementations . . . . . 12

3.1 Algorithms for MPI_Allreduce implemented in Open MPI 4.1.x . . . . 19

4.1 AlgID Mapping for Allreduce . . . . . . . . . . . . . . . . . . . . . . . . . . 32
4.2 Converting performance values into prefix-summed weights. . . . . . . . . 38

5.1 Hardware and software overview. . . . . . . . . . . . . . . . . . . . . . . . . 42
5.2 Proxy applications used for benchmarking. . . . . . . . . . . . . . . . . . . 43
5.3 Collective operation breakdown per proxy application . . . . . . . . . . . 43

A.1 AlgID mapping for MPI_Allreduce . . . . . . . . . . . . . . . . . . . . 57
A.2 AlgID mapping for MPI_Bcast . . . . . . . . . . . . . . . . . . . . . . . . 58
A.3 AlgID mapping for MPI_Allgather . . . . . . . . . . . . . . . . . . . . 58
A.4 AlgID mapping for MPI_Alltoall . . . . . . . . . . . . . . . . . . . . . 59

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# List of Listings

1 OpenMP parallelized loop . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2 Function definitions of MPI's send and receive routines. . . . . . . . . 10
3 Auto-tuning code injected into Open MPI . . . . . . . . . . . . . . . . . 35
4 miniAMR run configurations . . . . . . . . . . . . . . . . . . . . . . . . 61
5 ExaMiniMD run configurations . . . . . . . . . . . . . . . . . . . . . . . 61
6 Nekbone run configurations . . . . . . . . . . . . . . . . . . . . . . . . 62
7 SWFFT run configurations . . . . . . . . . . . . . . . . . . . . . . . . . 62
8 miniVite run configurations . . . . . . . . . . . . . . . . . . . . . . . . 62
9 Laghos run configurations . . . . . . . . . . . . . . . . . . . . . . . . . 63
10 mpi_coll_tuner setup script . . . . . . . . . . . . . . . . . . . . . . . . 65
11 Slurm template file . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
12 Irene template file . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Nomenclature

| | |
|---|---|
| MPI | Message-Passing Interface |
| ASP | Algorithm Selection Problem |
| ACP | Algorithm Configuration Problem |
| MPI-ASP | MPI Collective Algorithm Selection Problem |
| MPI-ACP | MPI Collective Algorithm Configuration Problem |
| ECP | Exascale Computing Project |

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 1

# Introduction

Supercomputers, nowadays, are made up of many compute nodes like the Frontier supercomputer with 9,408 nodes [1] or the Fugaku supercomputer with 158,976 nodes [2], providing large degrees of parallelism. Multiple programming paradigms exist to make use of large amounts of parallel computing resources. The Message Passing Interface (MPI) is a popular standard specifying routines used in software to efficiently harness the parallel execution power provided by a supercomputer.

MPI defines point-to-point communication procedures focused on moving data and communicating between two processes and collective communication procedures for interactions between arbitrarily large amounts of processes. In typical usage, the former is responsible for transmitting most data, while the majority of application runtime can be accredited to the latter [3]. With the prevalence of MPI in high-performance computing, optimizing the performance of collective communication can noticeably improve the performance of a large number of programs.

As MPI only specifies which operations a conforming implementation is to support and what syntax and semantics these operations have, common implementations like Intel MPI, Open MPI, and MPICH provide multiple methods or algorithms for each procedure, all achieving the same results, however doing so in different ways. The performance characteristics of these algorithms can differ noticeably for different parameters of the procedure and when executed on different hardware. We emphasize the impact the algorithm selection has on the runtime of an MPI application in the following. We executed the miniAMR proxy application six times on a configuration of $32 \times 32$ processes on the *Hydra* compute cluster at TU Wien and overrode the algorithm choice for all `MPI_Allreduce` collective calls with each of the available algorithms irrespective of the given parameters. Figure 1.1 shows the resulting application runtime for all algorithms. As evident in this figure, the total runtime of an application depends highly on which algorithms are selected.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

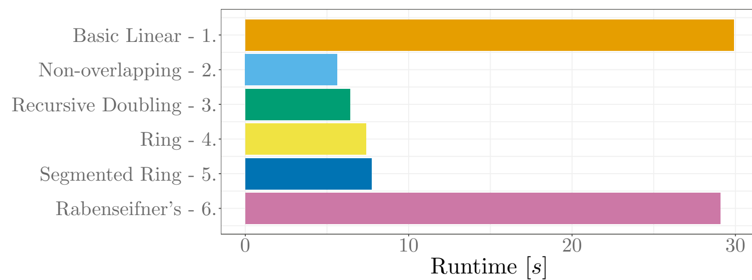



Figure 1.1: Total time spent in `MPI_Allreduce` for $32 \times 32$ processes running the miniAMR proxy application on *Hydra* varies greatly depending on the selected algorithm.

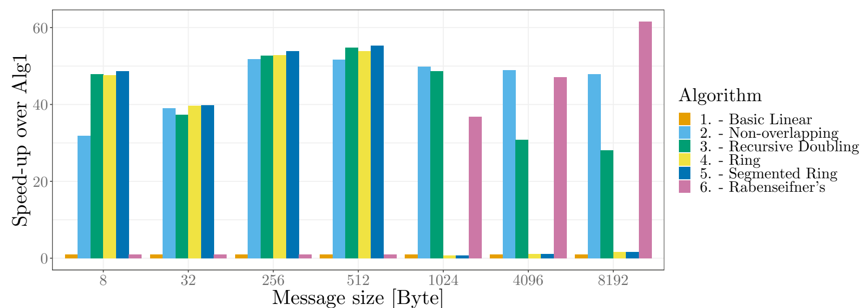



Figure 1.2: Comparing algorithm performance in `MPI_Allreduce` on different message sizes with the same operations on $32 \times 32$ nodes of *Hydra*.

Selecting appropriate algorithms for MPI collective communication operations is an important problem. In addition to runtime being dependent on the algorithm used, the size of the communicated message and the underlying hardware are both important. Figure 1.2 compares several `MPI_Allreduce` algorithms' runtimes with different message sizes on the *Hydra* cluster. Given the large differences in runtime across algorithms and message size, we plot the speed-up for each algorithm over the basic linear algorithm 1. Which algorithm performs the best does depend on the message size, with smaller values favoring algorithm 5, and larger values favoring algorithm 6. The default implementation in the given scenarios would select algorithm 2 for all the shown message sizes. This default selection does perform relatively well compared to the best-observed algorithms. However, by tuning the algorithm selection to the specific cluster, overall performance improvements are achievable.

## 1.1 Goals of this Thesis

In this thesis, we aim to improve the state of algorithm selection for MPI collective communication operations. Our contribution is a novel batched iterative auto-tuning approach which we implement for Open MPI 4.1.x. We build prediction models for the runtime of each algorithm by measuring a certain number of collective operations live



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

during production program execution, using this low-overhead measurement data to update the prediction model after each batch of application runs or time passed, and injecting our updated selection logic into the applications of the next batch. In contrast to previous work, we do not focus on micro-benchmarks as the primary source of timing values, instead directly measuring collective runtimes on the real applications used during typical workloads on each scientific computing cluster. The learned models and with it the prediction quality are expected improve over time as more timing data is collected from a user's real MPI application runs. This results in an auto-tuner improving the runtime of collective operations for applications a user aims to use, instead of solely tuning to improve benchmark results. In addition to the auto-tuner, our work also aims to improve the workflow for designing and evaluating potential tuning efforts in future projects. The following three research questions are to be addressed by this work:

**R1:** What type of model can be applied to online algorithm selection with minimal overhead over MPI applications?

**R2:** How much timing data is necessary to obtain stable performance improvements?

**R3:** Can performance portability across compute clusters and across applications can be achieved using a tuned algorithm selection model?

## 1.2 Thesis Outline

We structure this thesis with earlier chapters being more high-level and laying out information helpful to follow our approach and its discussion later on. In Chapter 2, we examine how supercomputers are designed, introduce the message-passing interface (MPI), and discuss the algorithm selection problem and its prevalence in machine learning. Following this, Chapter 3 serves as an exploration of the algorithm selection problem as applicable to MPI with a discussion of the complexity of selecting features to use for selecting appropriate algorithms. We describe how an MPI implementation needs to not just select appropriate algorithms but also configure those to attain the maximum performance. Having described previous approaches to the algorithm selection problem, we present our own in Chapter 4. Additionally, this chapter describes the practical process of implementing our auto-tuner for Open MPI 4.1.x and examines the challenges encountered during this, as well as the way we tackle each. This is put into practice in Chapter 5 where we describe the machines used during testing and go over all the experiments we performed for evaluating how well our approach works on actual systems consisting of different components. We go more into detail about what we can conclude from the experimental phase of this work and propose a simulation framework for researchers and implementers to model and investigate their work. We finish this work by concluding in Chapter 6 and giving an outlook where future work would be recommended.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

# Background

A broad range of domains make use of high-performance computing and aim to apply large amounts of computing resources efficiently to their problems. Supercomputers and general compute clusters offer parallelism to software and can execute far greater instances of problems in shorter times than regular single computers can. In the following, we present an overview of the architecture of supercomputers and discuss how software can be executed efficiently on such systems. Following this, we examine how MPI is relevant to writing programs for such systems and what features it does provide. Finally, we present the algorithm selection problem in a general sense and explore the presence of this problem in machine learning.

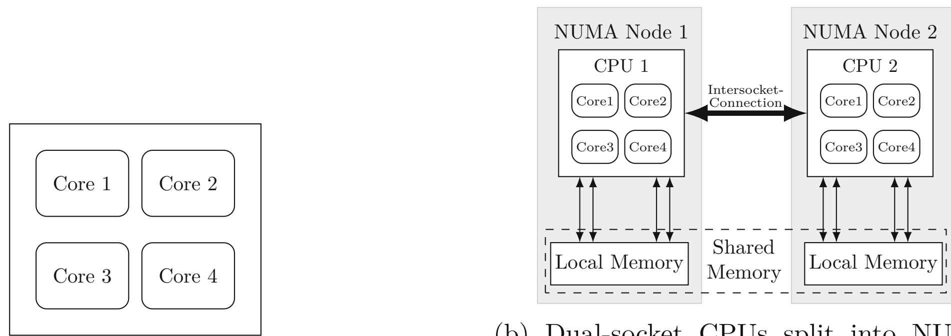
## 2.1 System Architecture of Supercomputers

A regular computer consists of multiple components, most importantly the Central Processing Unit (CPU) or processor and the Random Access Memory (RAM) or just memory. The processor handles instructions such as arithmetic and logical operations, making up any software. Memory holds data that the processor can operate on and allows the processor to read from and write to this memory, and offers higher speed than interacting with persistent storage mediums such as hard drives or SSDs [4]. A processor nowadays is not just a single piece for interpreting instructions, but, as visualized in Figure 2.1a, it is made up of multiple *cores* each capable of handling instructions independently of other cores. The one shown in Figure 2.1a contains four cores and is thus referred to as a quad-core processor.

A processor is installed in a system by being put into a socket, which provides many traces connecting the processor to outside components, such as the memory or in the case of having multiple sockets, links to an interconnect between these sockets. Figure 2.1b shows a system in which two processors are connected to memory and over an inter-socket connection to the other processor. A compute node or just node refers to a computer



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub




(a) A quad-core CPU with its four cores portrayed as boxes.

(b) Dual-socket CPUs split into NUMA nodes along with the RAM connected to each CPU.

Figure 2.1: Overview of the types of parallelism in a single processor and on a dual-socket system.

containing, at least, some processor and typically also memory attached to the processor. A general-purpose computer contains everything to be such a node. However, a node can also be a multi-socket system such as the one from Figure 2.1b.

A supercomputer differentiates itself from a regular computer in that it contains multiple nodes connected via some network. Frontier [1] a large supercomputer, leading the November 2022 TOP500 list of supercomputers[1], contains 9,408 individual compute nodes, each of which contains a single 64-core processor and 512 GB of memory.

To make use of all available resources, a normal software program is not enough. A program needs to be able to run multiple sub-tasks at the same time which then can be assigned to the appropriate processors. There exist various approaches to parallelizing software, depending on the details of the required resources and the task at hand. Hardware provides ways in which this parallelization can be realized in practice.

### 2.1.1 Shared Memory Machine

On some systems, a memory interconnect between the processors is provided, so that a CPU can access memory that is local to another CPU. The second CPU thus shares memory with the first CPU, through which the memory capacity of both CPUs can be increased, or information may be exchanged between the two processors. A machine supporting this concept is known as a *shared memory system*. In the machine shown in Figure 2.1b even though each processor has some local memory attached, it may still access the entirety of memory as this is shared.

In a shared memory system, parallel applications can exchange data between cores and coordinate execution, like splitting individual loop iterations across different cores, or assigning differing instructions to all cores. With shared memory between the two

[1] https://www.top500.org/lists/top500/list/2022/11/



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CPUs, coordinating work iterations could be done, e.g., by one core generating a work distribution mapping each iteration to a core and making this mapping available through the memory the two processors share.

If memory is shared between processors, each processor may access two different types of memory, the one directly connected to its socket and the memory accessible through a memory interconnect. However, as the latter memory access needs to pass through the interconnect in addition to the regular access path, performance is reduced [5]. This characteristic is called *non-uniform memory access* (NUMA) and refers to the observation that even though access to the shared memory is possible, the access time may differ significantly. We previously referred to the pairing of processors along with some memory connected as a node. In this setting, however, the term *NUMA node* refers to a processor and all memory it can access in uniform time. When multiple such NUMA nodes are connected, possibly inside a single compute node, the memory access speed inside a NUMA node is different from the speed of accessing memory across multiple NUMA nodes.

**Listing 1** Parallelizing a for loop on a shared memory system using OpenMP.

```
// serial for loop
for (int i = 0; i < n; i++) {
    perform_work(i);
    ...
}

// parallel for loop
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    perform_work(i);
    ...
}
```

The OpenMP API specification[2] defines compiler directives to the C, C++, and Fortran programming languages for parallelizing software on shared memory machines. Listing 1 showcases a simple directive for executing the iterations of a loop in parallel, with some iterations being assigned to cores as specified by a programmer.

### 2.1.2 Distributed Memory Machine

Not every machine does provide globally shared memory, instead there exists a type of machine in which processors each have memory local to them but keep access to this memory private. Memory is thus not available in a shared pool but is distributed across the machine. Distributed memory machines, as shown by an example in Figure 2.2 consist of multiple nodes with processors and private memory, which are then connected

[2] https://www.openmp.org/



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

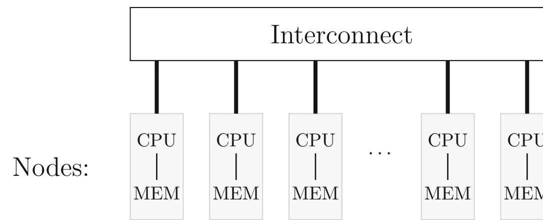



Figure 2.2: Multiple nodes are connected using an interconnect to obtain a network or multiprocessor computer system.

using an *interconnect*, with Ethernet being popular [4, pp. 15]. Frontier [1], consists of 74 cabinets each containing many compute nodes combining to 9408 processors in total. Such a system offers a large amount of distributed memory and a large amount of parallel work power to be utilized by parallel software. Interconnecting nodes in such a system, requires laying cables between the individual machines. The amount of cabling required depends on the structure of such a network and the way routers are set up. Multiple metrics are used to determine which network topology to use, such as hardware cost for running cabling and the achievable network throughput or latency. Modern supercomputers, such as the Frontier supercomputer are built on the Dragonfly topology, which makes use of multiple levels of hierarchy [6].

### 2.1.3 Programming Distributed Systems

Parallelization of software in distributed memory machines requires that data be exchanged, and the execution flow coordinated throughout participating nodes. Without the ability to exchange data using shared variables or global memory space, different systems for interaction need to be used. A more explicit interaction between nodes is provided by sending messages across the interconnect from one processor to another. Without the restrictions of running a single program, the software on distributed systems can be different on every processor involved and as long as the points of communication are coordinated correctly, they can exchange data with one another. Implementing ways to transmit these messages can be done in a variety of ways. Given such an implementation, if the goal is to port software to multiple other compute clusters, potentially with an interconnect from another provider or a different kind of computing hardware, the complexity of this porting process likely becomes a hurdle to the operation. Because of this, having a standardized way to perform such communication is a big advantage.

## 2.2 Message Passing Interface

The message-passing interface (MPI) is not a specific library or implementation to the message-passing paradigm, instead, the MPI constitutes an abstract, portable specification providing common routines required for message-passing in distributed memory computers to programmers in a portable and easy-to-use manner [7]. These routines range



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

from simply exchanging messages between two nodes to performing complex operations distributed across the system. MPI is the de-facto standard for the message-passing paradigm and provides implementations for a broad range of systems. This interface specifies the syntax and semantics libraries implementing message passing, need to provide and lists the operations to be supported [4, p. 228]. Libraries for distributed memory computing can be built on MPI and be ensured that they and user code built on top of such libraries, get portability across the broad range of systems supported by MPI. Many programming languages can interact with MPI implementations, for example, Open MPI can be used by any language with support for the foreign function interface (FFI).

### 2.2.1 Message Passing Programming Paradigm

Lacking globally shared memory, systems with a distributed memory architecture need another methodology for exchanging data with one another. If a processor $A$ wants to coordinate with processor $B$ and thus needs to transfer data, $A$ can send a message containing the data to be sent to $B$. The message passing paradigm assumes only that each processor can exchange data with every other processor, while not specifying exactly how this is to be done [4, p. 227]. With message passing, programmers may exchange data between processors or invoke functions from one processor on another one while being able to abstract over hardware details or the supercomputer network topology, depending on how much control is required.

In the message passing model defined for MPI the concept of a *collective* is introduced. This describes a set of processors all taking part in some procedure distributed over this collective. MPI specifies routines for performing communication between two nodes also called point-to-point communication and routines over some collective, called collective communication.

### 2.2.2 Point-to-Point Communication

The most basic way in which message passing can happen is if one processor needs to send some data to another processor. In this case, the sending processor executes a call to the MPI procedure `MPI_Send` which needs to be matched by an appropriate call on the receiving processor of `MPI_Recv`. We show function definitions of both routines in Listing 2. Both functions require a buffer, with `sendbuf` containing the data to be sent and received in `recvbuf`. On both sides, the `count` and `datatype` of the message to be transmitted are specified. Importantly, the communicator `comm` specifies in which context the two processes can talk to one another using the corresponding IDs in `dest` and `source`.

### 2.2.3 Collective Communication

The type of communication that does not necessarily involve exactly two processes, but instead an entire group of processes is called collective communication. While point-to-

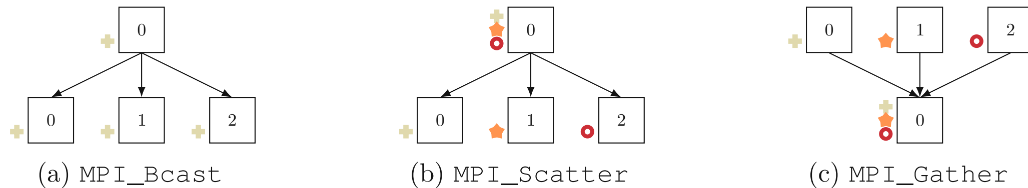


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing 2: Function definitions of MPI's send and receive routines.

```
int MPI_Send(void          *sendbuf,
             int           count,
             MPI_Datatype  datatype,
             int           dest,
             int           tag,
             MPI_Comm      comm)
```

```
int MPI_Recv(void          *recvbuf,
             int           count,
             MPI_Datatype  datatype,
             int           source,
             int           tag,
             MPI_Comm      comm,
             MPI_Status    *status)
```




(a) MPI_Bcast (b) MPI_Scatter (c) MPI_Gather

Figure 2.3: Three different MPI collective communication operations compared on simple cases.

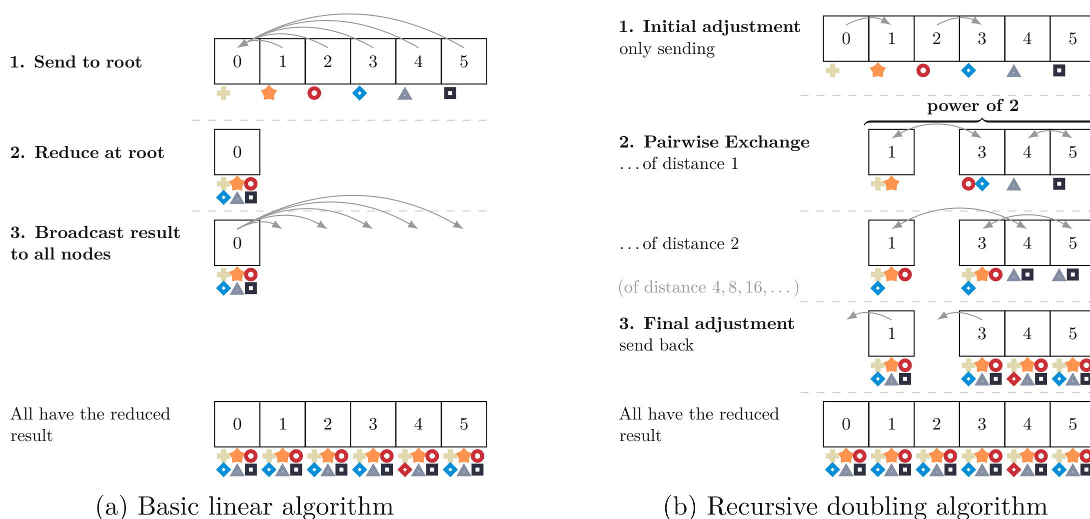
point communication can only exchange data, MPI provides collective communication routines for performing computation primitives for distributed operations.

Of the types of operations using collective communication, some are more communication-heavy, focusing on the transfer of data, such as MPI_Bcast. This defines the routine of broadcasting data from a single process to the collective of processes, so that each process ends up with this piece of data. MPI_Scatter is similar, in that one process starts with data and transmits this to the other processes, however, in this case, every receiving process ends up with a distinct, non-overlapping subset of the total data. A third operation of a similar function is MPI_Gather which acts as an inverse operation to MPI_Scatter, collecting data distributed across processes on one process. We visualize all three of these operations on simple settings in Figure 2.3. MPI routines such as these are called *rooted*, as there is one process that possesses a special role and is called the collective's root.

Another type of collective communication operations is provided by the collective routines MPI_Reduce and MPI_Allreduce. These accumulate data similarly to MPI_Gather, but additionally summarize this data using a reduction function. That is, these operations may compute the sum of a list of values spread out over multiple processes. As an illustrative example, multiple compute nodes could be tasked with simulating merchants trading produce with customers. At fixed intervals, the nodes evaluate how well they perform compared to all other nodes. They accomplish this, by comparing the sum of all sales made by any node with a statistic of their sales in a predetermined way and adapting their trading behavior depending on the outcome. Calculating the sum of all sales requires communication between the nodes, as the data is distributed across the nodes and not kept at a central location.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub




(a) Basic linear algorithm

(b) Recursive doubling algorithm

Figure 2.4: Two implementations for an `MPI_Allreduce` operation with six nodes, in which each node starts with a single element and ends up with the result of all six.

### 2.2.4 Algorithms for MPI Collective Communications

One collective operation, such as `MPI_Allreduce` may be implemented using different algorithms, with different ways of accomplishing their shared goal and along with this, different performance characteristics. A simple algorithm for `MPI_Allreduce` works by concentrating computation of the output value to one node and having this node distribute the result to the collective. This is shown in Figure 2.4a in which one node is marked as *root* of the collective and tasked with the described role. The root node gathers the data distributed over the collective, another operation common in MPI, it then performs the reduction as previously discussed and broadcasts the final value to the entire collective. This trivial approach does not concern itself with spreading work uniformly across all nodes, instead choosing to simplify the algorithm by centralizing the necessary work at the root.

In the recursive doubling algorithm, the aim is to increase performance by spreading work more evenly over the collective, consequently removing the special role of the root node. There is, however, an adjustment step so that the number of processes taking part in the communication steps are a power of two. To account for this, the extra nodes pass on their data early on and are not among those with work spread evenly across them. This algorithm performs three steps, visualized in Figure 2.4b: first the mentioned initial adjustment phase to change the number of active nodes to be a power of two, followed by an exchange phase between the nodes, and finally redistribution of the final output to the nodes, inactive from phase one. The exchange phase is where recursive doubling takes place, meaning that at first nodes exchange data with their immediate neighbors by ID, next with the nodes two steps away, then four, and recursively repeating this with the distance doubled each time. This goes to maximize parallel communication while



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Table 2.1: Number of algorithms for common MPI collective operations provided by popular MPI implementations.

| Implementation | Version | Allreduce | Bcast | Alltoall | Allgather |
|---|---|---|---|---|---|
| Open MPI | 4.1.x | 6 | 9 | 5 | 6 |
| Intel MPI | 2021.6 | 12 | 14 | 4 | 5 |
| MPICH | 4 | 8 | 8 | 6 | 5 |
| MVAPICH2 | 2.3.7 | 12 | 14 | 5 | 11 |

simultaneously minimizing repeated computation of the same value, as well as ensuring the order of operations, should the reduction operation not be commutative.

In contrast, to the basic linear algorithm, each node can reduce the partial amount of data it has at each step, e.g., in Figure 2.4b, node 1 can reduce the four values local to it in step 2.2 and then only needs to send one value to node 4, instead of an arbitrarily large list. Calculation thus can happen while data exchange takes place and is not an explicit step and calculations do not need to be repeated on every node.

### 2.2.5 Collective Communication Algorithm Selection in MPI libaries

The MPI standard is realized in multiple different implementations. In the following, we examine two popular open-source versions MPICH[3] and Open MPI[4]. Different implementations to MPI standards provide quite different amounts of algorithms for each of the most common collective operations. In Table 2.1, we list four of the most common MPI implementations and the number of algorithms supplied for each one of the four collective operations, which we examine in greater detail throughout this work.

An MPI implementation has to select an algorithm based on limited information. It may not be feasible to query every factor about the network topology, compute power on each node, network latency, or other specific data about the nodes in the collective call. What every implementation has access to by definition are all arguments to the function for the collective call and the communicator, including the size of the latter. The selection may pre-compute values used for algorithm selection at the start of each application run, such as estimating the current network topology. However, the work done at algorithm selection time is commonly reduced to maintain a low instruction overhead.

MPICH 4 uses a decision tree in which various factors lead to the final algorithm selection, in the current version a maximum of five decision points are taken before reaching a leave in the decision tree and obtaining an algorithm. For an `MPI_Bcast`, the decision is based on whether the communication is between nodes or localized to a single one, the communicator size, the communicator type (a way to encapsulate topology), and the message size. For an `MPI_Allreduce`, an additional important decision factor

[3] https://github.com/pmodels/mpich
[4] https://github.com/open-mpi/ompi



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

is whether the operation to be done is commutative, as an algorithm may perform re-orderings to improve speed, which can result in incorrect results should the operation not be commutative. MPICH exposes this decision logic as a JSON file with decisions being represented as nested branches in this tree. As a result, users may override the algorithm selection criteria.

The Open MPI 4.1.x implementation applies a similar approach to algorithm selection and equally bases the decision on a decision tree. In contrast to MPICH, the selection criteria are based directly on function arguments. For `MPI_Allreduce` the algorithm is selected based on whether the operator is commutative, the message size, and the communicator size. The decision tree in use by Open MPI 4.1 was constructed by running a set of micro-benchmarks on multiple compute clusters and evaluating at which message sizes and communicator sizes an algorithm performs best on average across the benchmarked systems [8]. In version 4.1 of Open MPI, segmentation of messages for segmented algorithms is not used, but users may set a different segment size or configure other parameter values using configuration files.

## 2.3 Algorithm Selection Problem

The problem of selecting appropriate algorithms is a common one and does appear in multiple domains. Given some information about a task to be solved, the goal is to select an algorithm to solve this task while optimizing for some objective like energy usage, running time, or error rate. When assigning resources to jobs in any setting, the way this assignment is done is optimized for some objective, for example, maximizing resource utilization at any given point in time. Game theory provides scenarios in which players must choose a strategy to win, like deciding the next move in a game of chess which may be done using various strategies or algorithms.

### 2.3.1 Formal Problem Definition

The algorithm selection problem (ASP) is highly dependent on the details of the specific domain and what is supposed to be optimized for. However, abstracting away this domain information, we can provide a general definition for the algorithm selection problem in Problem 2.3.1.

---

Problem 2.3.1 - Algorithm Selection Problem.

---

Given a portfolio of algorithms $\mathcal{A}$, a set of problem instances $\mathcal{I}$, and a cost function $f$, the algorithm selection problem is defined as finding a function $s : \mathcal{I} \mapsto \mathcal{A}$ such that the cost function $f$ is optimized across all instances: $\min \sum_{i \in \mathcal{I}} f\left(s(i), i\right)$.

An algorithm may not only cover a single method for solving a problem but instead cover a list of similar methods which are combined into one algorithm and can be selected by configuring parameters for this parent algorithm. An example of this is players in a



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

game applying a strategy in which typically a reliable, safe next step is preferred, but with a given probability $p$ a risky move is taken instead. These strategy variants can be selected by supplying the algorithm with some value for $p$ and measuring its performance. The space of possible parameter configurations is defined in Definition 2.3.1 again in an abstract manner.

**Definition 2.3.1.** Given an algorithm $a \in \mathcal{A}$ there exists a parameter space $\mathcal{P}_a$ for configuring $a$. This space may never be empty but can contain the empty configuration $\emptyset$, corresponding to the default, behavior, should no parameters be available. We denote the configuration of algorithm $a$ using parameter configuration $p \in \mathcal{P}$ as $a_p$.

Analogous to the ASP, we define the problem of configuring a specific algorithm in Problem 2.3.2 as the algorithm configuration problem or ACP.

---

Problem 2.3.2 - Algorithm Configuration Problem.

---

Given a portfolio of algorithms $\mathcal{A}$, a set of problem instances $\mathcal{I}$, a parameter space for each algorithm $\mathcal{P}_a$ with $a \in \mathcal{A}$, and a cost function $f$, the algorithm configuration problem is defined as finding a function $c : \mathcal{A} \times \mathcal{I} \mapsto \mathcal{P}$ such that the cost function $f$ is optimized over all algorithms and instances: $\min \sum_{a \in \mathcal{A}, i \in \mathcal{I}} f\left(a_{c(a,i)}, i\right)$.

### 2.3.2 Meta-Learning

In machine learning, the term *meta-learning* is used for observing the performance of different algorithms on given problems and learning which algorithms outperform others, leading to an overall increase in performance [9, p. 35]. Meta-data about the task at hand is collected, such as the algorithm's applicability, and the system in which the learning task is run. Based on this data, meta-learning can apply approaches from machine learning to improve the way algorithms are selected.

Meta-learning is about techniques for learning how to learn [9, p. 35]. In addition to applying this to the problem of selecting good algorithms for a well-known problem instance, it can be used for improving performance for other problem instance with little data. For the successful application of deep learning in particular, a requirement is access to a large dataset for this specific problem. In *few-shot-learning* or especially *one-shot-learning* this requirement is not satisfied, as only a few data points or even only a single one are available for any given classification or regression output, which results in a challenging task [9].

If, however, a closely related task with a large dataset or numerous closely related tasks each with a limited dataset exist, learning can still take place. In an approach similar to the way humans are able to extrapolate from some given information to apply this to related tasks, the learning progress across multiple tasks can be used for improving performance in each task [9]. From this, the meta-problem can be constructed and solved using a regular problem in machine learning, for example by learning common properties



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

of features encountered across tasks. The same machine learning techniques used for this application in meta-learning can be then applied to other domains encountering the algorithm selection problem.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 3

# Problem Statement and Related Work

In a study of open-source HPC applications done by Laguna et al. [10], nearly all of those reviewed make use of collective communication, with `MPI_Allreduce` being used the most. Additionally, Klenk et al. [3] show that MPI applications are likely to spend the majority of MPI time in collective operations. This combination of widespread use and high fraction of runtime indicates that increasing the performance of collective communication operations can net significant performance gains.

A large amount of general or highly-specific algorithms exist for commonly used collective operations and are offered in state-of-the-art MPI implementations. Table 2.1 provides an overview of the number of available algorithms in major implementations. By having a large set of algorithms to choose from, the choice of which algorithm to pick gets more complex. Consequently, if many algorithms may be selected, the logic of deciding whether an algorithm is appropriate for a problem instance gets more involved. Even if an algorithm is well suited to a large range of values, it may be out-performed for specific cases. Defining criteria for when to select which algorithm can, however, be challenging. Figure 3.1 spans a space of combinations between the number of processors and the message size. This space is partitioned into areas reflecting which algorithm has the best runtime for the number of processes and message size when running on some fictional machine. This is purely for visualization and should not imply that it is typical to draw such clean cuts across the space. In this example, algorithm $A$ is only feasible for small messages and small to medium number of processors, while algorithm $E$ dominates the field as soon as either of the dimensions is large enough. Between these two, other algorithms perform the best for small parameter ranges.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

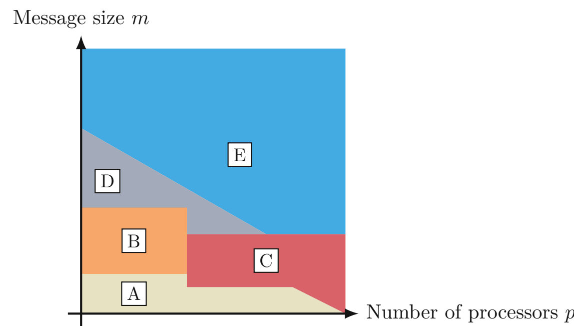



Figure 3.1: A potential, simplified 2D partitioning for the parameter space of message size × processor count depending on the best performing algorithm for every point.

## 3.1 Anatomy of Algorithm Selection for MPI Collective Communication

Given the general algorithm selection problem (ASP), we can equally define a formal version of the algorithm selection problem for MPI collective communication operations (MPI-ASP) as Problem 3.1.1. This definition is identical in structure to the generic one, with a measure of runtime $t$ being used as a cost function to minimize.

Problem 3.1.1 - MPI-Algorithm Selection Problem.

Given Problem 2.3.1, the MPI-Algorithm Selection Problem (MPI-ASP) is defined using a runtime measurement $t$ as finding a function $s : \mathcal{I} \mapsto \mathcal{A}$ such that the total runtime across all instances $\sum_{i \in \mathcal{I}} t(s(i), i)$ is minimized.

**Definition 3.1.1.** An instance $i$ of the MPI-ASP is made up of a collective operation, a set of parameters to this operation, and an execution context under which the operation is to run.

There are multiple important aspects of an MPI-ASP instance, which we specify in Definition 3.1.1. The collective operation to be executed is the main piece of this instance, as it defines the semantics an algorithm needs to exhibit. Each operation has parameters to further configure behavior, such as the size of the message to be transmitted for an `MPI_Bcast` operation or the operation and amount of data to be processed by an `MPI_Allreduce` operation. The last part of this problem instance is the execution context, which is the biggest and most complex aspect of an instance. The execution context contains all information about the environment in which the operation is to be run, for instance, the number of processes involved, or information about the underlying hardware like processor speed, the topology in which processes are assigned to cores in a compute cluster. Furthermore, information only available at runtime is also contained in this context like overall resource usage, or the amount of network congestion during an operation.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 3.1: Algorithms for MPI_Allreduce implemented in Open MPI 4.1.x

| ID | Algorithm |
|---|---|
| 1 | Basic Linear Reduce+Broadcast |
| 2 | Non-overlapping Reduce+Broadcast |
| 3 | Recursive Doubling |
| 4 | Ring |
| 5 | Segmented Ring |
| 6 | Rabenseifner's Algorithm |

## 3.2 Algorithm Selection Process for MPI_Allreduce

We have already introduced two algorithms for the MPI_Allreduce operation in Section 2.2.4. Table 3.1 lists all six algorithms available for this operation in Open MPI 4.1.x. In the following, we will reason first theoretically and later empirically which algorithms are expected to perform well, focusing especially on the basic linear and recursive doubling algorithms.

### 3.2.1 Theoretical Analysis of MPI_Allreduce Algorithm Runtimes

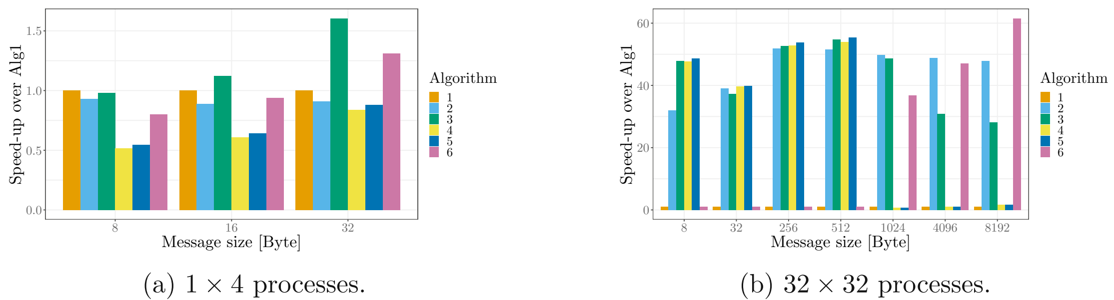
The runtime of algorithms is hard to predict exactly as it depends on many factors. A simple tool to evaluate how this runtime scales with increasing parameters is the big $O$ notation. If an algorithm's runtime is in the class $O(n^2)$ then this runtime is expected to scale depending on the square of $n$, so if $n$ is doubled then the runtime of this algorithm will take roughly four times as long. If one algorithm scales as $O(n \log n)$ and another as $O(n^2)$, then with large values of $n$, the first's runtime will not increase as much as the second's runtime.

The basic linear algorithm first gathers all data on the root, which scales linearly with the number of processes $p$ denoted as $O(p)$. In the course of this analysis, we assume that the reduction operation is done in constant time for each reduction step. Because of this, the reduction operation scales linearly with the amount of data per process $n$ and the number of processes, so $O(n \cdot p)$. Similarly to the gathering operation, distributing the data back to all processors again requires interaction from the root with each process, again scaling as $O(p)$. Combining all this, the algorithm scales with $p$ and $n$ as $O(n \cdot p)$, being dependent on their product.

In the recursive doubling algorithm, we examine the three major steps in isolation, after which we combine their runtime approximations. First, the initial adjustment reduces the number of processes to the closest power of two. This requires one round of communication to move data from the additional processes to those partaking in step two. In the pairwise exchange communication is intertwined with computation. In each step, every process communicates $n$ data elements, receiving $n$ new ones, and reduces these onto the existing $n$ elements. The recursive doubling constructs an inverted binary tree



TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.




(a) $1 \times 4$ processes.

(b) $32 \times 32$ processes.

Figure 3.2: Comparing algorithms using micro-benchmarking on *Hydra* shows how the best algorithm does depend on the operation parameters.

of height $\log p$, meaning that the pairwise exchange takes $O(n \cdot \log p)$ runtime. Finally, the data is re-distributed to those processes remaining from the first step, in another single, parallel communication round. In total, the recursive doubling algorithm scales as $O(n \cdot \log p + 1 + 1) = O(n \cdot \log p)$.

Based on the scaling behavior of both algorithms, the recursive doubling algorithm should always outperform the basic linear one. However, the big-$O$ notation does not take constant factors into account or the impact of factors scaling slower than the ones included in the final notational class. As the parameter values get bigger, this scaling approximation does, however, get more accurate.

### 3.2.2 Experimental Runtime Comparison of `MPI_Allreduce` Algorithms

To examine which `MPI_Allreduce` algorithm performs the best for some message sizes and the number of processors, we use the ReproMPI[1] benchmark on the *Hydra* cluster using Open MPI 4.1.x. By overwriting the default selection, we can instruct Open MPI to use the same specified algorithm for all parameters and call no further selection logic. We plot the median runtimes relative to the one of algorithm 1 in Figure 3.2 when executing on 1 node with 4 processes and 32 nodes with 32 processes each.

If we focus primarily on the basic linear algorithm (1) and the recursive doubling algorithm (3) in Figure 3.2a, we notice that for a very small message size on few processes, the basic linear one does match and even outperform all others. As soon as the message size increases, this does, however, change and the recursive doubling algorithm pulls into the lead and shows the best running time out of all algorithms. From this, we can conclude that the theoretical comparison of both algorithms does indeed predict the scaling behavior correctly. For small values, other factors not included in the big-$O$ notation come into play and the basic linear algorithm can perform better than predicted.

In Figure 3.2b, we ran the same test with a total of 1024 processes and with larger message sizes. When comparing the performance of algorithms 4 and 5 between the two

[1] https://github.com/hunsa/reprompi-dev



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

figures, they both appear to perform better in the case of more processes, consistently matching the performance of algorithm 3 until a message size of 512 B. Algorithm 3, which performs well for small to medium message sizes, does scale back compared to algorithm 1 for values larger than that, in which case algorithms 2 and 6 perform the best.

## 3.3 Algorithm Configuration Problem for MPI Collective Communication

If an algorithm has been selected to be used for some function call, the performance observed may still be highly influenced by selecting different values for the parameters available for this algorithm. Configuring algorithm parameters is a problem closely related to selecting an appropriate algorithm for a given instance. Problem 3.3.1 defines the algorithm configuration problem for MPI collective communication operations, with the goal of minimizing a measure of running time.

---

PROBLEM 3.3.1 - MPI-ALGORITHM CONFIGURATION PROBLEM.

---

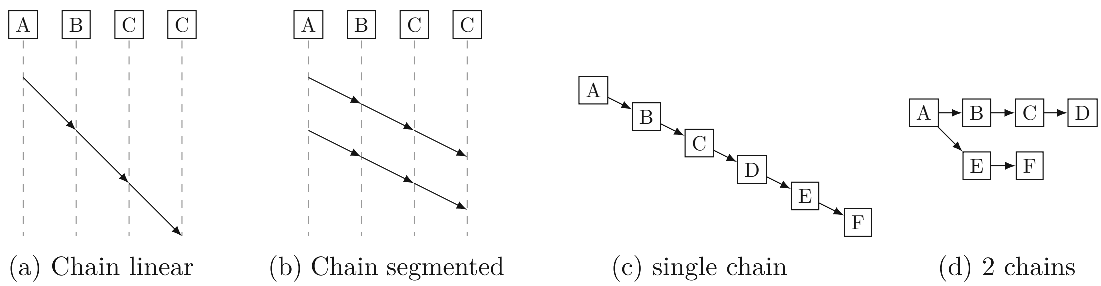
Given Problem 2.3.2, the MPI-ALGORITHM CONFIGURATION PROBLEM (MPI-ASP) is defined using a runtime measurement $t$ as finding a function $c : \mathcal{A} \times \mathcal{I} \mapsto \mathcal{P}$ such that the total runtime across all algorithms and instances $\sum_{a \in \mathcal{A}, i \in \mathcal{I}} t(a_{c(a,i)}, i)$ is minimized.

Algorithms for MPI collective operations, as can be found in different implementations, provide a range of general and algorithm-specific parameters. In Open MPI and MPICH, one parameter to be tweaked available for multiple algorithms is the *segment size* corresponding to how big the segments should be in which large messages sent across the network are to be split. The performance of sending such large messages can be improved if instead of sending one message, multiple smaller messages are sent. If a message is to be broadcast over a network using a *chain* algorithm, the communication may be modeled as in Figure 3.3a. As soon as one processor has finished receiving the message, it may be passed on to the next hop. By splitting the message into two parts, as shown in Figure 3.3b, node $B$ can start sending as soon as the first part has arrived and the total throughput across the network is increased. This process is also known as *pipelining*.

In contrast to general parameters, some algorithms expose parameters specific to their internal operations. Algorithms acting on the network in a tree topology provide parameters to configure the branching factor, specifying how many processors are connected to each one in the graph, or the type of tree to be built. Chain algorithms, like to one implemented in Open MPI, can provide a parameter similar to this tree branching factor. In Figure 3.3c we show a typical single chain, however, configuration may allow the creation of two chains starting at the root, as shown in Figure 3.3d. This may be combined with pipelining to further increase network throughput when sending large amounts of data.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub




(a) Chain linear (b) Chain segmented (c) single chain (d) 2 chains

Figure 3.3: A view of how the segmentation size and chain count parameters affect the way communication in a chain topology happens.

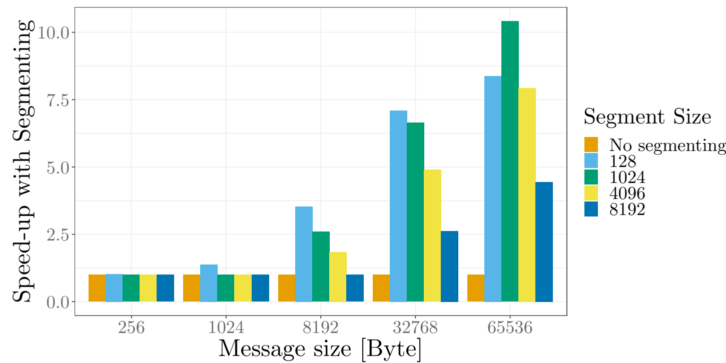



Figure 3.4: Speed-up in running time when enabling segmenting compared to the same pipeline algorithm (3) without segmenting enabled for `MPI_Bcast` with $32 \times 32$ processes on *Hydra*.

The attainable speed-up through segmenting depends both on the size of message to be sent, and the size of segments to split this message into. Figure 3.4 shows the influence of this factor on the running time of the pipelined algorithm implemented in Open MPI 4.1.x for an `MPI_Bcast` operation when run on $32 \times 32$ processes on the *Hydra* compute cluster. Segmentation does not perform well for small messages, however, with increasing message size, a large speed-up can be attained with different segment sizes.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

## 3.4 State of the Art in Algorithm Selection

There exists a broad range of research into the general ASP and also the MPI-ASP, with different approaches to tuning algorithm selection or parameters. In the following, we discuss a selection of work relevant to our own approach.

The ASP is prominent in machine learning and part of *meta-learning*. Hutter et al. [9] conduct a survey into this branch of machine learning, categorizing and discussing different approaches to tackling this task. Some approaches presented in their work can also be applied to the MPI-ASP. Guerra et al. [11] offer one approach specifically designed for the MPI-ASP, namely predicting the performance of algorithms by training regression models for each one individually, using features extracted from the problem to be solved. Their approach is to build these models as support-vector machines for their Meta-Regression task.

To apply machine-learning techniques, Leyton-Brown et al. [12, 13] build models of *empirical hardness*. This is a metric for estimating the complexity of a problem instance given some select problem features. Using this hardness, they predict an algorithm's performance on problem instances. Hutter et al. [14] build upon the work by Leyton-Brown et al. to perform automated tuning of continuous algorithm parameters, using this approach to determine which problem features are most significant for the problem of parameter configuration. Machine learning techniques have also been shown to be useful for optimizing the parameter tuning process, among others by Pellegrini et al. [15]. They do not focus on tuning specific algorithm parameters, as we do, but instead on finding good values for parameters of the MPI runtime based on features of programs to be executed. In their work, they give an evaluation of decision trees and neural networks for improving the performance of MPI user programs on the specific hardware on which model learning was performed.

Pješivac-Grobvić et al. examine how to model decision boundaries for the collective algorithm selection problem using quadtrees [16] as well as in their later work on using C4.5 decision trees [17]. They show that both tree structures can model a mapping of $\langle \text{Communicator size}, \text{Message size} \rangle \mapsto \langle \text{Best algorithm} \rangle$ successfully, with the latter being able to represent this data with less complexity. These generated trees are then used to construct decision logic as is currently in use by Open MPI. Given that MPICH has similar decision trees, their approach could apply to this implementation as well.

Multiple papers have been written on the feasibility of timing micro-benchmarks and using measurements from this to obtain a model of which algorithm to select based on given parameters. Hunold et al. [18] run micro-benchmarks to evaluate all possible algorithm configurations and select the one performing best in each scenario. The benchmark results are used to train one random forest model for every implementation of a collective operation, resulting in collectives$\times$algorithms many models. Each model predicts the runtime of its assigned algorithms runtime as a ratio compared to the algorithm selected by default based on the sizes of the communicator and message. Hunold et al. [19] later on expand on this work to focus on well-tuned prediction on collective runtimes using



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

this prediction model to improve algorithm selection. Instead of creating and training the algorithmic model ahead of time, Wilkins et al. [20] apply their tuning effort directly before an application's execution. In their approach, called ACCLAiM, Wilkins et al. [20] introduce an additional step to each compute job in which algorithm tuning is performed on the specific (sub-)system the job is run on. A user-supplied list of collective calls, that should be representative of the tuned application, is benchmarked to obtain the tuned algorithm selection. Ahead of program execution time, a performance-guiding model is built, while this is done in an offline manner, it is very specialized and needs to be re-run with every job invocation, thus being closer to online model building.

Faraj et al. [21] propose Star-MPI a set of self-tuned adaptive routines to be used for collective operations. The approach injects tuning procedures online by measuring the performance of all possible algorithms for a required collective call and afterward using the algorithm configuration resulting in the best performance. The step to measure a set of possible algorithms is repeated multiple times should the chosen algorithm drop in performance in subsequent runs. Through this, Star-MPI is applicable to MPI code with many invocations to collective operations as it needs to obtain enough data. However, through the online tuning process, adaption to new system configurations per run or work-load changes within a single run is possible.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 4

# Iterative Online Tuning Approach

We propose a novel approach for addressing the MPI-ASP on supercomputers by building an auto-tuner learning which algorithm performs best for any given problem instance. We distinguish the applicability of our tuner from previous work. Some tuning efforts [20, 21] focus on improving the performance of a single application run which lends the ability to tune for a given hardware allocation in the case of workload managers. Other works [8, 17, 18] use a separate tuning phase to first collect information about algorithm performance, for example using micro-benchmarking, and then apply this information to select the best-performing algorithm during an application run. We combine these two approaches to obtain an iterative auto-tuner that uses performance measurements obtained from tracing real user applications, trains prediction models on this data, and re-uses it in subsequent application runs to select algorithm based on their observed performance, iteratively building more reliable models while improving the applications' performance in the process.

The decision to learn from user applications instead of micro-benchmarks has multiple reasons. Firstly, a micro-benchmarking approach would need to also define which problem instances of the MPI-ASP to benchmark so that any possible application is well tuned for. By directly learning from applications that are typically used, the performance dataset inherently contains many of those problem instances which are often observed and has a larger amount of data for these cases. We also do not need to spend tuning time on instances that do not come up in these applications.

The second reason against learning from micro-benchmarking data is that imbalance in the arrival times of processes in collective operations are implicitly present in applications. These can influence the runtime of collective operations significantly as shown by Faraj et al. [22]. Thus, having this implicit factor be part of training data may positively affect the resulting algorithm selection quality.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

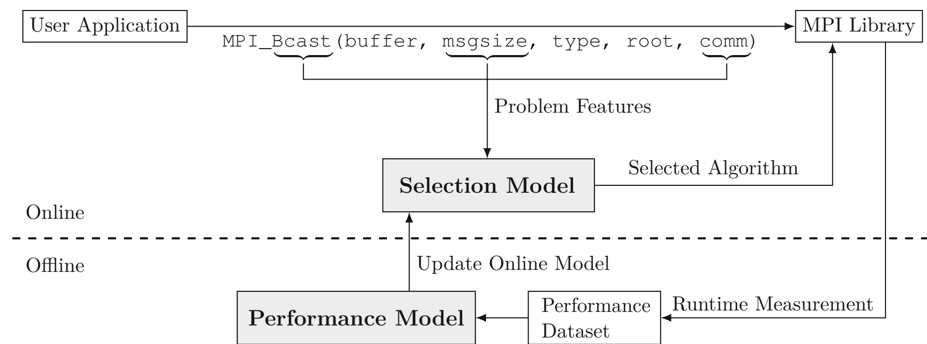



Figure 4.1: Architecture of our novel auto-tuner system.

In the following, we present an architectural overview of our iterative auto-tuner, showing what important parts make up the full workflow, followed by detailed discussion of significant aspects of the tuning process.

## 4.1 Architecture of Batched Algorithm Selection

An auto-tuner for the MPI-ASP aims to improve an application's performance by selecting fast algorithms during a user's application execution. Consequently, the maximum runtime overhead of such a system is to be kept low. This makes it intractable to query complex machine-learning models for every problem instance. Faraj et al. [21] approach this using a selection methodology with low computational intensity, selecting the best known algorithm intertwined with periodic checking if another algorithm might outperform the chosen one. Instead of following this approach, IOTA constructs an expensive prediction model offline, separate from a user's application, from which a lightweight algorithm selection model is constructed, to be used in the online tuning phase.

Figure 4.1 shows this distinction between our online and offline phases of operation. The online section of the auto-tuner hooks into an MPI library and intercepts collective calls, such as an `MPI_Bcast`. From this collective call, problem features are extracted, like the type of operation, the message size, and information about the communicator. Using these problem features, a lightweight selection model is queried for an algorithm expected to perform well. This is executed using the MPI library and the collective runtime traced. The offline section of the auto-tuner takes all timing data collected after a few application runs and uses this to train a performance model for predicting how long each algorithm will take for any collective call. This performance model is then compressed to the lightweight state we require for online selection and injected into subsequent application runs.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

### 4.1.1 Online Algorithm Selection

In order to optimize the performance of an application, our auto-tuner hooks into the MPI library and overrides the built-in algorithm selection logic for a subset of calls, by picking optimized algorithms for each scenario. Simultaneously, this selection gathers information about the feasibility of different algorithms to further improve performance of future runs. Four main steps make up this online selection.

The first operation, tracing MPI collective calls, involves intercepting these collective calls as they are invoked from a user application. During this step, required information of the call is extracted and converted into parameters for the MPI-ASP. Furthermore, it is at this point in the flow that the decision to sub-sample the calls, not tracing all that come in, is done to keep the overhead of sampling potentially slow algorithms low.

Secondly, the extracted parameters are used to query the online selection model for an algorithm expected to perform well in the given setting. This selection can be performed in a multitude of ways linking to the well-understood problem of deciding between exploiting current knowledge and exploring not yet fully understood options [23, p. 243].

In the third operation, the selected algorithm is injected back into the MPI library overriding the default selection logic. It is at this point, that it is executed as per any usual MPI run.

Finally, the fourth component involves measuring the runtime each process takes during execution of this algorithm. This data is fed into a performance dataset and then iterated on by the offline model to influence the behavior of future iterations of the online algorithm selection logic.

### 4.1.2 Offline Runtime Prediction Model Building

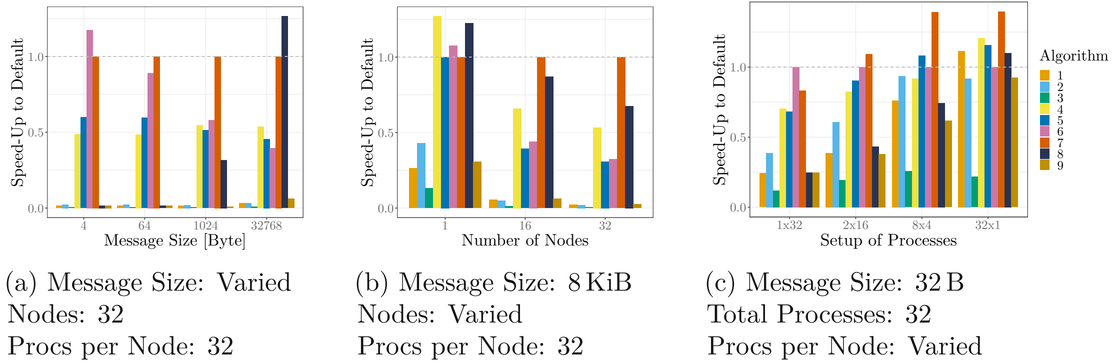
The offline part of our approach is invoked once enough data has been gathered or some pre-determined amount of time has passed. This step is critical to enabling our auto-tuner to iterate on the obtained performance traces and increase the probability of good algorithms being selected in future runs.

The timing measurements from the online tracing steps serve as the basis for building comprehensive prediction models for the expected performance of each algorithm given any problem instance. Using regression, a good estimate of how long an algorithm will take to execute some problem instance can be obtained. Querying these complex models during the online section of our auto-tuner would, however, introduce unwanted instruction overhead, in turn, a simplified version of this is used for the online selection.

This simplified representation is built by querying the full models only at specific data points from the pre-defined parameter space that are expected to be representative of the entire prediction space embedded in the offline models. By doing so, we obtain a simple view of how likely an algorithm is expected to perform well for any problem instance of the MPI-ASP.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub




(a) Message Size: Varied
Nodes: 32
Procs per Node: 32

(b) Message Size: 8 KiB
Nodes: Varied
Procs per Node: 32

(c) Message Size: 32 B
Total Processes: 32
Procs per Node: Varied

Figure 4.2: Executing `MPI_Bcast` benchmarks on *Hydra* leads to different best algorithms when one parameter is changed.

## 4.2 Defining the Parameter Space for Algorithm Selection

The MPI-ASP, as defined in Problem 3.1.1, is about finding a mapping from any problem instance to an algorithm, such that the runtime of the chosen algorithm on each problem instance is minimized. The problem instance as a whole is a complex structure of many features, some of which are given explicitly, such as the collective function to be called or the amount of data to be operated on. However, many more features are part of these problems, some of these implicitly such as the current (SLURM) resource allocation, the workload under which this operation takes place, or the network topology. A method for tackling the MPI-ASP needs to define which of these problem features to use as dimensions for selecting appropriate algorithms.

We pick three problem features to select an algorithm under the MPI-ASP. These features are extracted from the problem itself and converted into parameters for our algorithm selection function $s$ as given in Problem 3.1.1. Any function that might instantiate $s$ in an implementation of our approach operates on this three-dimensional parameter space.

Firstly, for any collective communication procedure, we can define a *message size* as the number of bytes transmitted or operated on by the procedure such as an `MPI_Bcast` or `MPI_Allreduce`. The message size used for collective communication typically occupies a narrow range of few distinct values for each application [24]. Despite that, the message size is the parameter with the largest range of possible values over multiple applications, as Klenk and Fröning [3] observe message sizes between single bytes and multiple megabytes, showing a range of possible values of six orders of magnitude for this parameter.

The other parameters make up the number of processes involved in a collective communication procedure, also called the communicator size. Instead of using a single number for this, we choose to split it up further, into the number of nodes and the number of processes per node, assuming even spread across the nodes. The *number of nodes* that are part of a collective communicator is defined as the number of compute nodes over which



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

the processes involved in the collective call are distributed. This number is dependent on the specific cluster used or possibly the maximum number of nodes assignable to a program running on this cluster. Based on the current top-performing supercomputers[1] Frontier [1] and Fugaku [2], this parameter currently goes up to the hundreds of thousand, yielding a range of four to five orders of magnitude.

Finally, the complementing value to the number of nodes is the number of *processes per node*. For this work, we assume that all nodes are assigned roughly the same number of processes and that a single CPU core is only allocated a single process. Frontier and Fugaku are made up of nodes each containing a single processor, coming in under 100 cores per node. Given that other supercomputers [25] may have up to multiple hundred cores per compute node, the possible range of values appears to be one to two orders of magnitude.

Our choice of parameters is based on picking those we expect to help with accurate modeling of the runtime of algorithms. However, it is equally essential that a parameter be obtainable at runtime with low instruction overhead, such as the message size that is directly passed as a function argument to the collective call, the number of nodes, or the number of processes per node which are contained in the current communicator. Other features of a problem instance, such as the current network congestion are hard to approximate at runtime, so we do not choose such a problem feature as one of our parameters. Figure 4.2 shows how changes to a single parameter while fixing the other two, does influence the algorithms' relative runtime and with it the best one to select.

## 4.3 Defining a Runtime Metric for Collective Communication Operations

The cost function to optimize for an MPI-ASP is a runtime metric of the used by a collective communication operation. To perform successful tuning, the auto-tuner needs to extract an accurate measurement of this runtime. In contrast to a single process, assigning a runtime to a parallel operation can be done in multiple, contradictory ways. Different metrics for a collective call's runtime may advantage or disadvantage an algorithm based on the way work is spread across all processes or the overall amount of work. The choice of runtime metric informs the auto-tuner on which algorithms to prefer, as such, defining a runtime metric representative of an overall goal is crucial.

To specify the runtime for each single process, we subtract the start time of a procedure from the end time, as reported by `MPI_Wtime` on this process. This is implemented using native high-precision timer functions on the tested systems, as reported by `ompi-info` and the `MPI_Wtick()` function. In case a system does not implement a high-precision `MPI_Wtime`, IOTA additionally supports `gettimeofday()`. Figure 4.3 shows multiple possible distributions of the way processes may start and end their part in some collective operation. Depending on the way work is distributed by an algorithm, a process may be

[1] https://www.top500.org/lists/top500/list/2022/11/



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

done significantly earlier than others, or all finishing at similar times. As an example, Figure 4.3a shows how a chain broadcast could be implemented in which the root "Proc 8" passes on data to 7 and finishes. In contrast to this, Figure 4.3b shows a pairwise exchange algorithm in which two processes exchange data, after which only one out of each pair continues, recursively.

A common way to specify the total runtime of such a collective operation is the *makespan*, defined as $\max_p(t_{\text{end}}(p)) - \min_p(t_{\text{start}}(p))$. This is the time from the first process starting to the final one completing the procedure. Alternatively, the runtime could be based on the mean time per process or overall total processing time spent in the collective call. We choose a third option, setting the auto-tuner to minimize the largest duration any individual process spends in a collective call. This consequently benefits algorithms which spread work very equally, whereas algorithms in which a few processes do the majority of work are scored less favorably.

Each process's start time may be slightly different in practice, like in Figure 4.3c, due to an imbalance in the work done before this operation. Performance implications as a result of this and the patterns such imbalance may show have been studied by Faraj et al. [22]. We only evaluate blocking collective calls during this thesis, however, the MPI standard up to and including MPI 4.0 [26] allows implementations to let a process exit the collective call as soon as their local work is done and the local buffer is ready to be reused, indifferent of the status of work in other processes.

The way we define our cost metric for the MPI-ASP is by selecting the largest duration a single process spends in an MPI collective communication procedure. The function $t$ is defined as $t = \max_{p \in P} [t_{\text{end}}(p) - t_{\text{start}}(p)]$ where $P$ denotes the set of processes. Compared to defining the runtime as the makespan, this methodology does not require a global time and is more compatible with drift of the processes' local clocks, as no synchronization requirements are imposed. During our experiments, we observed issues due to drifting clocks when using the makespan as our runtime metric, causing us to switch to the described alternative.

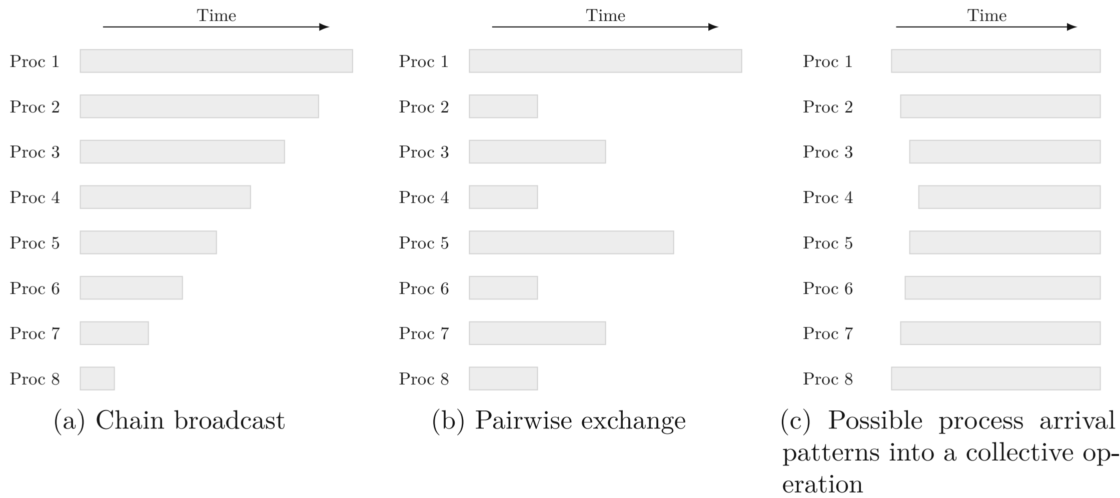
## 4.4 Merging the ACP into the ASP

The auto-tuner needs to not only solve the selection of algorithms (ASP) but also the configuration of these selected algorithms (ACP). When both of these problems are to be tackled, this may be done in a multitude of ways. It is possible to perform this in two separate steps, first selecting and then tuning an algorithm, or in the continuous tuning of both problems, e.g., by tuning parameters for multiple algorithms in tandem and noting which performs the best overall. We choose an approach, based on the work by Hunold et al. [19], in which the tuning process only needs to solve the selection problem instead of both.

A parameter exposed by an algorithm may either only take on one of a finite amount of possible values or one of a possibly contiguous range of infinite values. We require



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN




(a) Chain broadcast

(b) Pairwise exchange

(c) Possible process arrival patterns into a collective operation

Figure 4.3: Start and end times for all processes in an operation are dependent on the algorithm itself, as well as on the imbalance of processes arriving differently based on imbalance in previous work.

a domain expert to make the latter discrete so that the space of all possible parameter assignments is finite. For example, for an `MPI_Bcast` algorithm with pipelining, the segment size could only take on one of the allowed values $s = \{s_1, s_2, \ldots, s_n\}$. A custom identifier, the AlgID now maps to an algorithm of the MPI implementation, configured with one unique parameter configuration. The AlgID maps to each possible configuration of any algorithm with the allowed values. By performing this, we get a large MPI-ASP with already configured algorithms. Table 4.1 shows the mapping of our AlgID for the `MPI_Allreduce` call to the implemented algorithms and a configuration of the single parameter *segment size*. Only one algorithm uses segmentation of messages, while the other algorithms have a direct mapping between our custom identifier and the one provided by Open MPI. For space reasons, we include the equivalent tables for all implemented collective calls in Appendix A.

## 4.5 Collective Algorithm Performance Models

In an offline step, separate from the execution of MPI applications, we train predictive models for the performance of all algorithms on collective calls. These models are not reused as-is for online algorithm selection but instead, serve as the baseline for constructing simpler algorithm selection models fitting to the online portion of IOTA.

### 4.5.1 Regression Models for Collective Runtime Prediction

To model how well each algorithm performs given some parameters, we train one regression model per AlgID on all the timing measurements obtained so far. This is done, instead of



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 4.1: Mapping of our AlgID to an Open MPI algorithm and parameter configuration for `MPI_Allreduce`.

| AlgID | Open MPI Algorithm | Segment Size |
|---|---:|---:|
| 1 | Basic Linear - 1 | / |
| 2 | Non-Overlapping - 2 | / |
| 3 | Recursive Doubling - 3 | / |
| 4 | Ring - 4 | / |
| 5 | Segmented Ring - 5 | 1 KiB |
| 6 | Segmented Ring - 5 | 1 MiB |
| 7 | Segmented Ring - 5 | 8 MiB |
| 8 | Rabenseifner's - 6 | / |

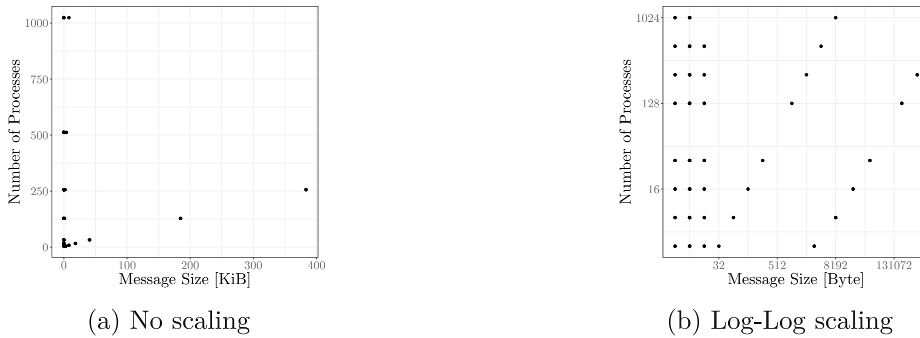
having one model per collective call with all possible algorithms, since a single algorithm may be outperformed in all but a few cases while still being the best in these very few cases. This would lead to significant selection bias, as good-performing algorithms are selected disproportionally to bad-performing ones, potentially leading to these algorithms not being regressed accurately and not getting considered for the top places.

For these regression models, we choose tree-based ones, as Hutter et al. [27] found random forest models to perform well for algorithm runtime prediction. Additionally, Pješivac-Grbović et al. [16, 17] observed good applicability of these types of models for predicting MPI collective algorithm runtime specifically. For our prediction trees, we make use of XGBoost[28] a popular implementation of boosted regression trees. Each regression model takes in the three parameters defined in Section 4.2 and outputs the expected runtime from Section 4.3. For our models, we use the Python package *xgboost 1.6.2* with 200 regression trees and a `Tweedie` objective function, given the long right tail we observe in algorithm runtime. We perform no further hyperparameter tuning, as we focus on a simple model.

To combat converging too quickly on a good algorithm measured at the beginning, we introduce guarding factors. In case a minimum threshold of timing data has been reached for an algorithm, we mark this one as *untrained* and do not train a model for this case. That is, if we assume that the noise in the training data would be too large for the model to accurately predict an algorithm's runtime. If fewer than five algorithms for a collective call are trained, the collective call is marked as *untrained*. This means that the trained models for algorithms with enough training data are ignored and all algorithms in this collective are continued to be sampled uniformly at random. This is repeated until enough training data is present.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub




(a) No scaling (b) Log-Log scaling

Figure 4.4: Distribution of observed parameter values for collective calls during ECP benchmark applications.

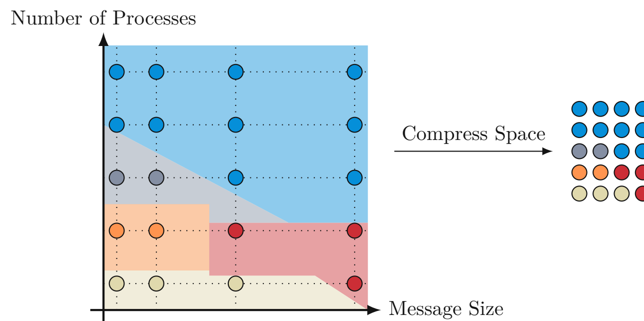



Figure 4.5: The machine learning model partitions the parameter space, to generate a model for usage at runtime, we discretize this space using pre-defined boundaries.

### 4.5.2 Deriving a Selection Model for Runtime Usage

Querying a regression model per algorithm available is too computationally expensive to be done at runtime, combined with our method of having as many models to query as algorithms for a collective call, this would impose significant overhead. We propose a method of porting the learning power of these models to a lightweight version capable of being queried at runtime.

To reduce the number of look-ups necessary to select an appropriate algorithm, we examine the distribution of our parameter values for typical applications IOTA is assumed to encounter. Figure 4.4a shows a two-dimensional scatter-plot of all values for message sizes and the number of processes involved when executing benchmarks from the ECP (Exascale Computing Project) [29], which we go into more detail in Chapter 5. This figure shows how only few regions of the total parameter space are actually used. Furthermore, by applying logarithmic scaling to both axes in Figure 4.4b, this data is shown to not be uniform but follow an exponential distribution.

If any one of the chosen three dimensions is only slightly changed, this will impact an algorithm's absolute runtime, however, the relative runtime and with it, the ranking of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

algorithms are expected to remain unchanged. Making use of the underlying parameter distribution and the chunking of values close to one another, we reduce the amount of data we store and bin the parameter space. We select a three-dimensional grid of representative data points, linearly distributed across the number of nodes and the number of processes per node, and exponentially distributed for the message size. Figure 4.5 shows a simplified, two-dimensional representation of this grid with the filled areas reflecting the regions in which our regression models found one algorithm to outperform others. A representative data point in this case does not just store the single best algorithm at this specific point but instead a performance vector.

**Definition 4.5.1.** A *performance vector* defines a list of values $\langle p_1, p_2, \ldots, p_n \rangle$ where each $p_i$ gives a probability in range 0 to 1, reflecting how well an algorithm is expected to perform relative to the others. The higher $p_i$, the better algorithm $i$ is expected to perform. This vector serves as the baseline for selecting a random algorithm weighted by predicted performance.

Given some parameter values, it is possible to look up which data point is representative of these binned values. Having looked up the according data point, the auto-tuner can query its performance vector, giving it a value in an abstract performance metric for each of the available algorithms. The metric we choose is $1/t_a$ where $t_a$ gives the modeled runtime for algorithm $a$. Each algorithm should thus maximize its inverse runtime, so in turn minimizing the runtime is desirable. Based on this performance metric, an algorithm is then sampled.

## 4.6 Online Algorithm Selection

During an application's execution, the auto-tuner is tasked with sampling a few collective calls and selecting algorithms for this that are expected to perform well. Multiple steps, that hook into the MPI library at different points, are necessary to perform this algorithm selection. At first, during initialization up-front setup work is done to ensure fast algorithm selection. In the finalize step of MPI, the collected timing data is gathered from all processes and packaged into a single output file. Most important to this thesis, however, is the actual algorithm selection once an application invokes a collective call.

Listing 3 shows our altered version of the Open MPI library for injecting a broadcast algorithm selection. This minimal tracing harness is added for any collective call which should be auto-tuned. The first decision before any auto-tuner code is executed, is whether to trace this specific call on the calling process. If this process should not trace, it still calculates which algorithm to execute, as all processes need to select the same algorithm. However, this process does not measure the algorithm's runtime. The algorithm selection model is queried for an AlgID, which is unpacked into the algorithm ID as well as its parameter values. At this point, execution is passed back to the typical flow in Open MPI, which executes the collective call with the chosen algorithm. To finalize, all selected processes report the end time of the algorithm.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

**Listing 3** Code injected into Open MPI to get the sampled parameters and unpack the chosen parameters.

```
// Sub-sample the collective calls
// Only trace a fraction of MPI calls
if ( AT_is_collective_sampling_enabled()
     && AT_is_collective_sampling_possible() ) {

    // Query custom AlgID using algorithm selection logic
    alg_id = AT_get_bcast_selection_id(count * type_size, comm_size);

    // Unpack AlgID to
    //     - Open MPI algorithm id
    //     - Parameter values
    AT_col_t at_alg = AT_get_bcast_our_alg(alg_id);
    algorithm = at_alg.ompi_alg_id;
    segsize = at_alg.seg_size;
    faninout = at_alg.faninout;

    // Start time measurement
    call_id = AT_record_start_timestamp(...);
}
switch (algorithm) {
    // Execute the chosen algorithm (from Open MPI 4.1.x)
    ...
}
if ( AT_is_collective_sampling_enabled()
     && AT_is_collective_sampling_possible() ) {
    // Stop time measurement
    AT_record_end_timestamp(..., call_id);
}
```

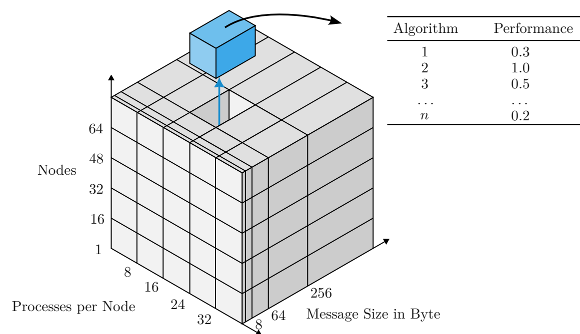
### 4.6.1 Selecting an Algorithm Performance Vector

A collective call that has been traced enough to exceed the minimum thresholds is enabled for auto-tuning. At runtime, the MPI application is hooked and corresponding calls are intercepted. From this call, the auto-tuner extracts problem features of an MPI-ASP instance. In our case, the three parameters, message size, number of nodes, and the number of processes per node are extracted from the collective call arguments and the communicator. These raw values are incompatible with the lightweight selection model, so they are binned into pre-defined values. A triple of binned values can then be used to query a performance vector for the current collective call.

A simplified example of how this process is done is shown in Figure 4.6. The parameter space is visualized as a three-dimensional cube with lines marking the boundaries at which we bin each dimension. The axes *Nodes* and *Processes per Node* are scaled linearly, despite the communicator size observed in proxy applications growing faster than linear. We conclude that by splitting it into two dimensions we are likely to manage this non-linear



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub




| Algorithm | Performance |
|---|---|
| 1 | 0.3 |
| 2 | 1.0 |
| 3 | 0.5 |
| ... | ... |
| $n$ | 0.2 |

Figure 4.6: By specifying three dimensions for binning, a probability distribution is obtained.

scaling. The message size is, however, also scaled logarithmically in this parameter space. The bins reach for each dimension range from the lower to upper mark along each axis. The final bin in each dimension acts differently, it is unbounded so that values it captures all values larger than its lower bound. A message of size 1 GiB would land in the bin starting at 256 B. In this example, a specific bin is selected for 96 nodes with 20 processes each and a message size of 128 B. Once the appropriate range for each dimension has been established, these are combined to obtain a single three-dimensional bin. Each such bin contains a performance vector mapping some performance metric to each algorithm, as shown in Figure 4.6. This vector gives a measure of relative performance for all algorithms and is used by our algorithm selection logics to pick the most appropriate algorithm for the given problem instance and current goal. Each implemented collective call possesses one such three-dimensional binned cube.

### 4.6.2 Algorithm Selection Logic applied on Performance Vectors

During the learning process, it is typically desirable to always select the algorithm performing the best, reducing the application's runtime. This choice can, however, be suboptimal if the current model does not accurately reflect which algorithm is the fastest in each scenario. This may be due to a good algorithm not being selected often enough to have an accurate enough model or a non-static distribution of the algorithm's runtimes. There, consequently, may exist situations in which selecting an algorithm expected to perform worse can improve the total performance. As a result, the need for a more nuanced selection logic or even multiple different ones arises.

Faraj et al. [21] seek to resolve this problem in their work STAR-MPI by introducing multiple phases in which algorithms are selected in different manners. At first, all algorithms are measured, next only the top-performing one is picked, followed by a loop of checking if the next-best algorithm might outperform this one and then again only selecting the best one.

We propose a modular architecture in which the algorithm selection logic is separated from all other parts, enabling us to easily switch out how this selection should be performed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

An algorithm selection logic only concerns itself with taking in a performance vector and selecting one specific algorithm from this vector to execute. For this thesis, two different selection logics are implemented and used to inform the tuning process. In the first, the inverse modeled runtime per algorithm is normalized to obtain a probability of selecting each algorithm based on how well the regression models predict each algorithm to perform relative to all others. These probabilities are then capped to be in the range $[3\%, 50\%]$. This capping is performed to prohibit rapid convergence on a suboptimal algorithm. As alternative algorithm selection logic, it is assumed, that the model is accurate enough and only the best algorithm is selected for any traced call.

## 4.7 Application Overhead from Online Algorithm Selection

For a tool that is built to run concurrently with time-sensitive MPI applications, having an accurate model of the amount of overhead this imposes is essential. To begin, we present an overview of the types of overhead stemming from our approach. We follow this by more closely reasoning through the overhead of selecting an arbitrarily bad algorithm and what measures our auto-tuner takes to reduce the impact of such bad selections. IOTA imposes three types of overhead on the applications it is injected into: *memory*, *instruction*, and *slow algorithm* overhead.

The memory overhead stemming from storing performance vectors for each possible bin is tuned to be inconsequential for a typical MPI application. It does depend on the selection mode in use, as when always picking the best-performing algorithm, only one integer is needed to be stored per parameter bin, resulting in roughly 10.76 KiB with 18 bins for the message size, 17 for the number of nodes, and 9 for the number of processes per node. In the alternative case, when the selection logic may choose any algorithm, storing the full performance vectors is necessary. When tuning the Open MPI algorithms for `MPI_Allreduce`, `MPI_Bcast`, `MPI_Allgather`, and `MPI_Alltoall`, with a total of 52 algorithms, this needs to store 559.4 KiB to encompass the whole space.

The different selection logics presented in this thesis were tuned to have negligible instruction overhead when used for any type of MPI application. Optimizations for having quick access to the fastest algorithm in the given mode or random selection have been performed to obtain an algorithm quickly. For example, in the weighted random selection we select an algorithm randomly based on the weight or performance associated with it. To make this efficient, the individual weights are normalized, giving them a sum of 1. An example for six algorithms of this process is shown in Table 4.2. We now calculate the prefix-sum of the normalized vector, resulting in a monotonically increasing vector of values with the largest one being 1. To select one of these six algorithms based on their performance a single random value $r$ between 0 and 1 is needed and the algorithm $a$ with the smallest value $p_a$ with $p_a \geq r$ is selected. In the example shown in Table 4.2 using a random value of 0.645 the selection picks $A4$.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Table 4.2: Converting performance values into prefix-summed weights.

| | A1 | A2 | A3 | A4 | A5 | A6 |
|---|---|---|---|---|---|---|
| Performance | 6 | 4 | 2 | 2 | 4 | 2 |
| Normalized | 0.3 | 0.2 | 0.1 | 0.1 | 0.2 | 0.1 |
| Prefix Sum | 0.3 | 0.5 | 0.6 | 0.7 | 0.9 | 1.0 |

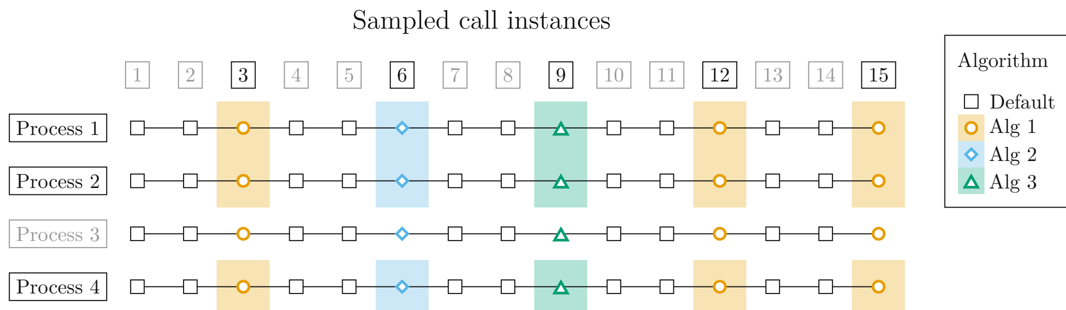



Figure 4.7: Only a sample of calls selected for injecting our algorithm selection and from those only a sample of processor's times are measured.

A bigger part of memory can be the amount of timing data we collect for iterated tuning, which also requires some amount of work to gather from all processes and write to a file. Each measurement stores start and end times as `double`, as well as metadata about the collective call. This contains four `int` values: the collective ID, AlgID, message size, and communicator size. In total, per measurement and traced process 24 B, meaning that the memory impact depends on how many such measurements are taken. The primary way to reduce the amount of memory and time required for this is by restricting the number of calls we intercept and trace during an application. Especially for decreasing the amount of post-processing time in gathering time, we can select a subset of all processes instead of the global communicator for which we measure and store runtimes. All processes still need to perform the same steps for selecting an algorithm, as otherwise, execution would diverge. Figure 4.7 shows an example execution of an MPI application with a total of 15 collective calls using four processes. In this case, only processes 1, 2, and 4 are selected for measuring time and do so only for five of the call instances. Limiting how many MPI collective calls the auto-tuner is supposed to tune per application run has another effect. In addition to shrinking the amount of timing data needed to be kept, these two measures reduce the amount of time a very slow algorithm can contribute.

It is important to not eagerly rule out an algorithm after just a few rounds, given a high variability in the timing results, so even badly performing algorithms will be kept around and given a small probability of being selected. A user may choose to blacklist ones that are already known to be undesirable in certain scenarios or apply the learnings from the tuning process to extend this blacklist with more entries over time. Even with these steps, it could still be possible that for instance a linear broadcast algorithm would be selected



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

for transmitting a message of multiple megabytes slowing down execution of this call by multiple orders of magnitude. We try to reduce the overall impact on the application of such a bad selection by only allowing the tuning process to select algorithms for 1000 calls instead of every single one.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

# Experimental Evaluation

The following chapter concerns itself with describing the process of experimentation performed to evaluate the real-world feasibility of an auto-tuner based on our prototype. At first, setup on all used machines is presented. After that, we discuss which of all the available collectives we select to tune and why the chosen subset is representative of expected usage. Finally, the performed experiments are discussed and the obtained results presented.

## 5.1 Experimental Setup

For evaluating whether our prototype could produce positive tuning results in different settings, we made use of two different clusters: *Hydra* and *Irene*, summarized in Table 5.1. The *Hydra* system comprises 36 nodes each with two Intel Xeon Gold 6130 processors for 32 cores on each node, situated at TU Wien. This cluster was used for the initial experimentation and prototyping of the IOTA auto-tuner. To verify proper scaling behavior and ensure that this tuner would not only function on a single cluster, later experimentation was additionally performed on the *Irene* cluster hosted at the TGCC at CEA. The Skylake portion of this cluster consists of 1,653 nodes comprising two Intel Xeon Platinum 8168 processors with 24 cores each for a total of 48 cores per node.

## 5.2 Collective Calls to Tune

MPI offers a fair selection of collective calls, with varying purposes and performance traits. In this thesis, we do not implement tuning capabilities to all of these calls. This is fine since the overall premise can be shown by tuning some operations with a significant share of the overall application runtime. We implement the functionality for `MPI_Allreduce`, `MPI_Bcast`, `MPI_Allgather`, and `MPI_Alltoall`. We chose these specific collective calls because of their high usage throughout different domains,

Table 5.1: Hardware and software overview.

| Machine | $N$ | $ppN$ | Processors | Interconnect | GCC | Slurm |
|---|---|---|---|---|---|---|
| *Hydra* | 36 | 32 | 2x Intel Xeon 6130 | Intel Omni-Path | 10.2.1 | 22.05.3 |
| *Irene* | 1653 | 48 | 2x Intel Xeon 8168 | EDR InfiniBand | 8.3.0 | 20.11.9 |

for example, the distributed deep learning framework "Horovod" uses exactly these in its MPI backend[1]. More importantly, Laguna et al. [10] find that nearly all MPI programs make use of collective operations, and among those who do, the four collectives chosen are in the top ten of all collective operations with `MPI_Allreduce` being used by nearly every application in their study. Klenk and Fröning [3] further show that collective operations make up a majority of application runtime, making them a more worthwhile optimization target than point-to-point operations.

We base our selection of proxy suites on the research of Klenk et al. [3], picking those with a high communication time in at least one of the collective operations we choose to tune.

## 5.3 Benchmark Applications to Trace

This section provides an overview of the applications we use for benchmarking different algorithms and the ECP, from which the applications stem. The Exascale Computing Project (ECP) covers a multitude of initiatives of the U.S. Department of Energy (DOE) to lay the groundwork for achieving exascale compute power and delivering large speed-ups over previous supercomputers.[2] Among the contributions are multiple applications designed for use as benchmarks of parallel systems. These applications are known as *proxy applications* since they model performance-critical applications and "mimic" the programming patterns and styles of such applications. A proxy is, in a way, a minified application, with a large portion of runtime allocated to high-intensity parallel programming, reducing the amount of start-up, preprocessing, and finalizing of results.

We make use of six proxy applications, from the ECP, for measuring algorithm runtime and evaluating our novel concepts. A short description for each proxy is given in Table 5.2. We select this list of proxy applications as they provide a broad overview of different compute patterns and rely on different collective operations to accomplish this, however, this selection is discussed in more detail in Chapter 5.

The Exascale Computing Project (ECP)[3] offers simplified versions of real-world high-performance applications. Each ECP proxy app represents the most important aspects of an exascale application, with a high focus on preserving the performance characteristics closely [29]. In Release 4.0, the latest available to date, ECP offers 14 proxy app suites,

[1] https://horovod.readthedocs.io/en/stable/concepts.html
[2] www.exascaleproject.org
[3] https://www.exascaleproject.org/



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table 5.2: Proxy applications used for benchmarking.

| Application | Description and context |
|---|---|
| miniAMR | 3D stencil calculation with Adaptive Mesh Refinement |
| ExaMiniMD | Particular Molecular Dynamics |
| miniVite | Louvain method in distributed memory for graph community detection |
| SWFFT | Distributed variant of fast Fourier transform |
| Nekbone | Computational fluid dynamics simulation |
| Laghos | Solver for the Euler equation of compressible gas dynamics in a Lagrangian frame using unstructured high-order finite elements |

Table 5.3: Breakdown of how common each of the traced collective operations are in the ECP proxy applications used for auto-tuning.

| Application | `Allreduce` | `Bcast` | `Allgather` | `Alltoall` |
|---|---|---|---|---|
| miniAMR | 94.5% | 1.9% | 3.6% | n.a. |
| ExaMiniMD | 98.7% | 1% | 0.3% | n.a. |
| Nekbone | 97.3% | 2.7% | n.a. | n.a. |
| SWFFT | 100% | n.a. | n.a. | n.a. |
| miniVite | 1.7% | n.a. | n.a. | 98.3% |
| Laghos | 93.6% | 6.4% | n.a. | n.a. |

each with different workloads and usage characteristics of MPI collective/point-to-point calls.

Sultana et al. [30] as well as Klenk and Fröning [3] analyzed how these and other proxies make use of MPI operations, as well as the kinds of collective calls and their respective total runtime per application. Using this data we can pick a subset of proxy suites with distinct quality that makes use of at least some of the calls we implement tuning for.

We provide Figure 5.1 and Table 5.3 as illustrations of why and how we make use of the six different proxy suites we choose. We require applications in which the collective operations we tune make up a significant part of the overall application runtime. If this is true, we can ensure that enough relevant timing data will be generated to build a model of good quality. Additionally, the applications function as benchmarks with a large part of the runtime being modifiable by providing different algorithm selections, which in turn amplifies the resulting difference of a good or bad selection model. Figure 5.1 shows for each proxy application suite how much of its overall MPI time, as measured from `MPI_Init` until `MPI_Finalize`, is taken up by collective calls we tune in this work. From this, we can make out the *Laghos* suite as an especially promising candidate for tuning. This sum of runtimes is further split up into percentages of each collective call



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

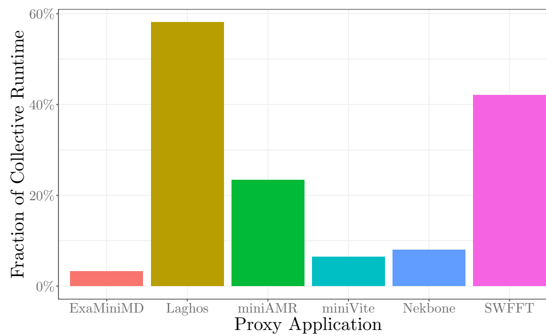
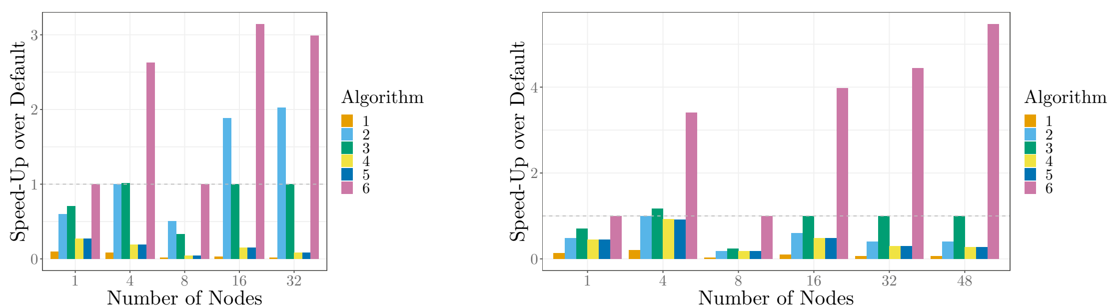



Figure 5.1: Time spent in `Allreduce`, `Bcast`, `Allgather`, or `Alltoall` collective calls per proxy application, as a fraction of overall MPI time.




(a) *Hydra* with $N \times 32$ processes.

(b) *Irene* with $N \times 48$ processes.

Figure 5.2: Comparing the performance of different algorithms for `MPI_Allreduce` with 10240 Bytes.

in Table 5.3. To obtain the data needed for both plots, we executed each application suite with parameters as described in Chapter 5 with setup $32 \times 32$ (32 nodes with 32 processes per node) and selecting time as reported on rank= 0.

The ECP proxy applications mimic the behavior of more complex MPI applications. For representative timing data, these applications need to be called with

## 5.4 Experimental Results

We now present the results of applying our auto-tuner on the two different compute clusters *Hydra* and *Irene* by learning from ECP proxy applications. In addition to this, we build a reference frame for what performance improvements are possible on both systems in various cases and closely examine the overhead of our algorithm selection logic itself.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

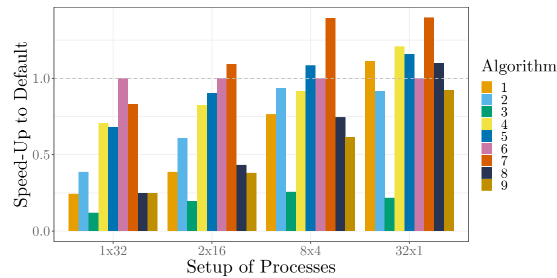
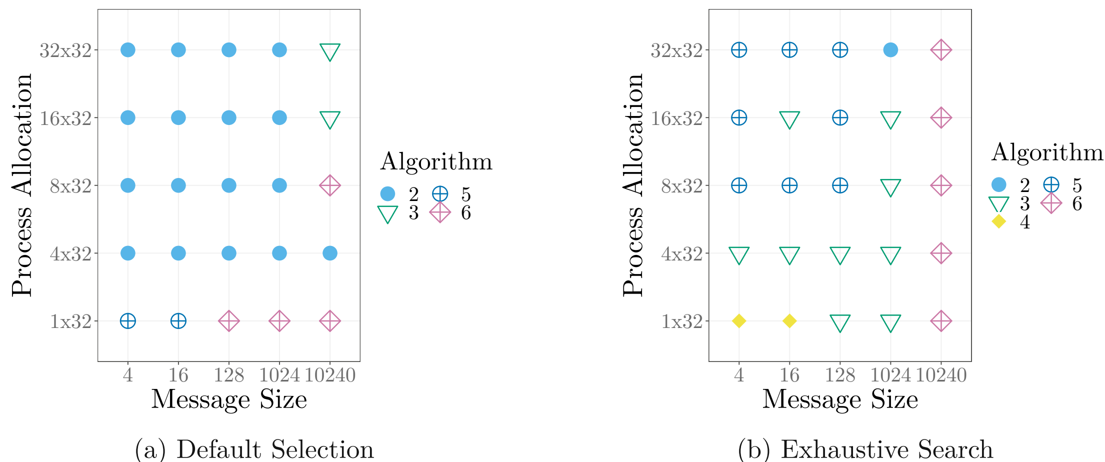



Figure 5.3: Comparing `MPI_Bcast` performance on *Hydra* with fixed total number of processes.




(a) Default Selection

(b) Exhaustive Search

Figure 5.4: Visualizing selected algorithm from default selection compared to the best found by exhaustive search on *Hydra* for `MPI_Allreduce`.

### 5.4.1 Establishing a Baseline of the Maximum Attainable Performance Improvement

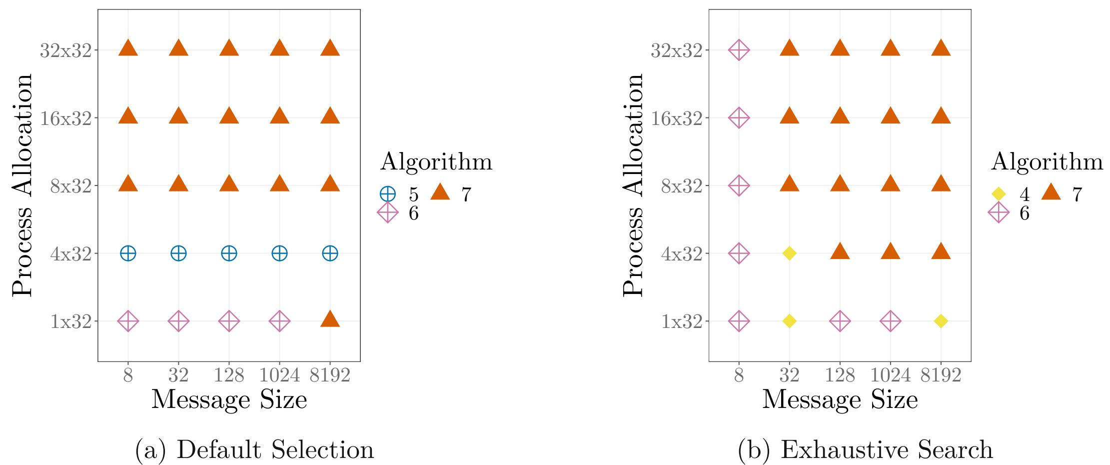
The default algorithm selection logic for collective communication procedures, as established, does not always select the best algorithm. To put the auto-tuner's performance gain into relation we compare the performance of all available default algorithms for a variety of parameter values. To accomplish this, we selected a range of values for message size, number of nodes, and processes per node, and executed the `MPI_Allreduce` operation for these values for 5000 iterations each with each algorithm as configured per default. This shows the performance range of the algorithms the default selection logic may select during regular operation. To execute the benchmarks we ran each case with ReproMPI[4] with the reduction operation `MPI_SUM`.

Figure 5.2 depicts each algorithm's runtime depending on the number of nodes and with that the total number of processes for `MPI_Allreduce`. Comparing the differences

[4]https://github.com/hunsa/reprompi-dev

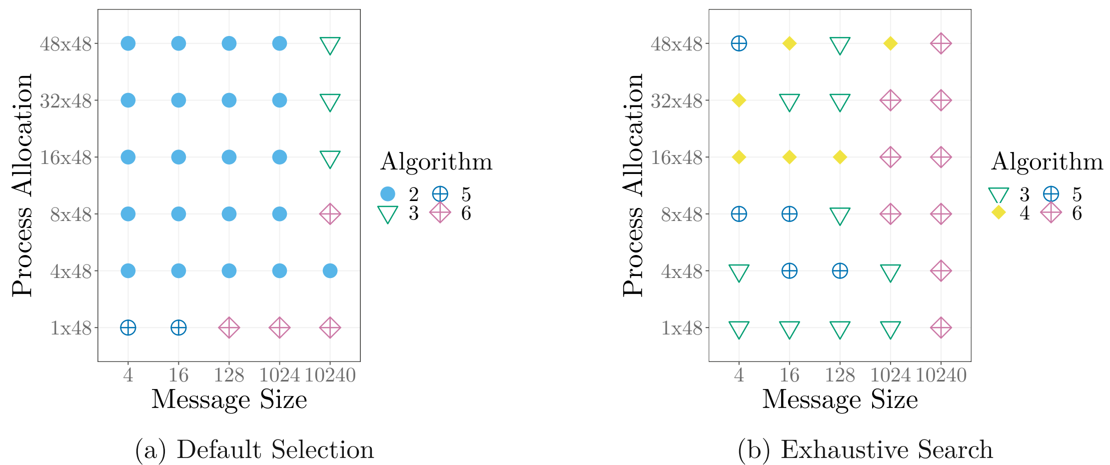


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub




(a) Default Selection (b) Exhaustive Search

Figure 5.5: Visualizing selected algorithm from default selection compared to the best found by exhaustive search on *Hydra* for `MPI_Bcast`.




(a) Default Selection (b) Exhaustive Search

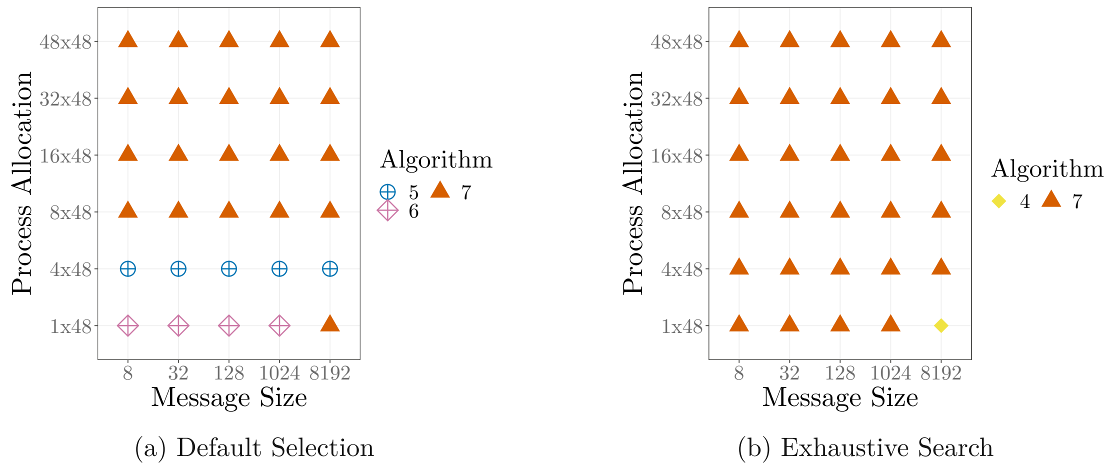
Figure 5.6: Visualizing selected algorithm from default selection compared to the best found by exhaustive search on *Irene* for `MPI_Allreduce`.

between both tested systems shows that equally, the default selection is bested for the most part by Rabenseifner's algorithm, number six. However, on *Hydra* the non-overlapping algorithm does beat out the default selection for a larger number of nodes, a type of behavior that cannot be observed on *Irene*.

Another factor of interest to the implementation of our auto-tuner is how high the impact of external factors, such as specifics of SLURM allocations, is on the runtime of collective operations. Figure 5.3 shows how different algorithms for `MPI_Bcast` perform the best with a fixed message size and fixed number of processes, only altering over how many nodes these processes are evenly spread. As this is a factor ignored in the Open MPI default selection, this is taken into account in our tuning efforts.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN




(a) Default Selection (b) Exhaustive Search

Figure 5.7: Visualizing selected algorithm from default selection compared to the best found by exhaustive search on *Irene* for `MPI_Bcast`.

Finally, for a comprehensive overview of the potential performance gains, Figure 5.4 and Figure 5.5 compare the best-found algorithm with the default selected one for `MPI_Allreduce` and `MPI_Bcast` on *Hydra*, respectively. The same information for *Irene* is shown in Figure 5.6 and Figure 5.7.

### 5.4.2 Comparing Instruction Overhead between Selection Logics

There are multiple potential sources of overhead an online auto-tuner could introduce to an MPI application. We compare the time taken by a collective algorithm plus all instructions before that algorithm which performed the selection logic. This is done using ReproMPI, which measures the entire process of selection and execution of a collective algorithm. By repeating this experiment and only swapping out the selection logic, we can compare the runtime of all available algorithm selection logics. The choice of algorithm does not affect this experiment, since we force each selection to always return the same algorithm.

Open MPI implements multiple ways of selecting an algorithm, of which we examine the following two:

- *default*: The default way how Open MPI selects an algorithm. This logic queries a decision tree for each collective call and decides based on the number of processes and the message size.
- *mca*: The Modular Component Architecture (mca) is a framework included in Open MPI for users to override the default selection. We choose a decision file in which user-defined decision boundaries are used to select an algorithm. At runtime this is converted into a decision tree and all branches which are not reachable are trimmed, e.g., due to too many processes being required.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

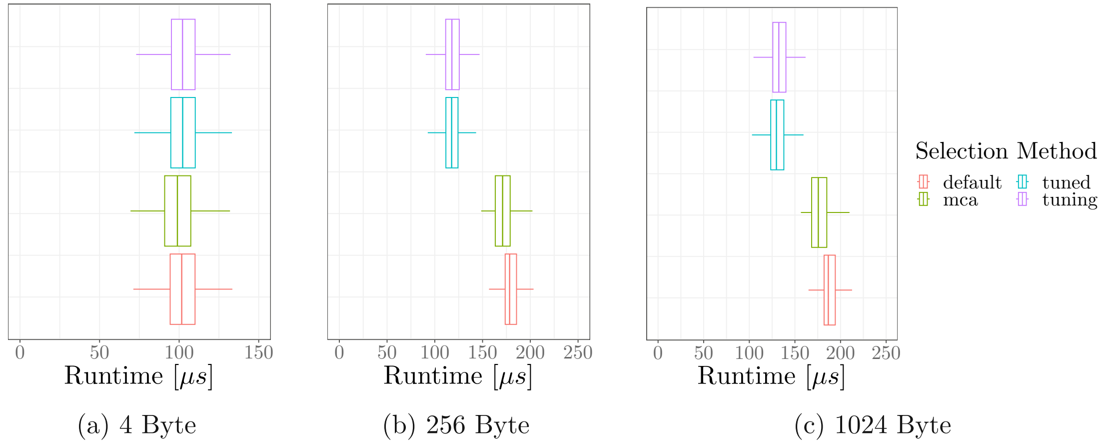
Our auto-tuner itself has two different ways of selection an algorithm, which are called depending on the user's requirements:

- *tuning*: A weighted random selection process. Each algorithm is assigned a performance value based on the problem instance and queried from the algorithm selection model. These performance values are used as weights when randomly sampling an algorithm.
- *tuned*: Selection logic to pick the best known algorithm at the current time. This takes in the same performance vector as *tuning* uses. However, before any collective call is queried, these vectors are pre-processed to only keep the algorithm with the highest weight. At runtime, the problem instance is used to read this single algorithm and select it.

We compare the full runtime of an `MPI_Allreduce` and `MPI_Bcast` collective call in Figure 5.8 and Figure 5.9 respectively. The comparison is executed on three different message sizes, to evaluate that this does not impact the relative values. For `MPI_Allreduce`, at 4 B the *mca* selection performs slightly faster than the others. For all other message sizes, however, the auto-tuned selection logics beat out the Open MPI ones. As expected, we see *tuned* being consistently slightly faster than *tuning*, as it does not need to sample a random value.

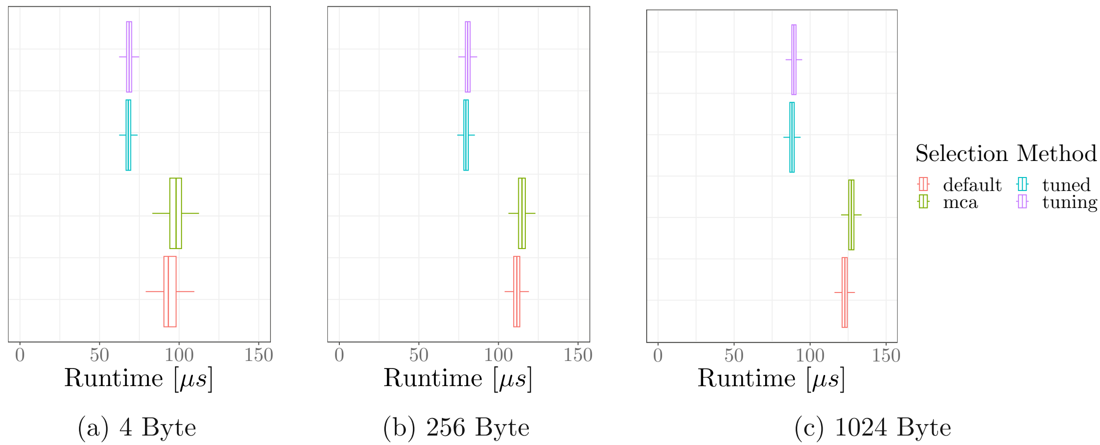
The analysis for `MPI_Bcast` is conclusive again, in that our custom selection logics performed the best for all message sizes we tested. These results show that the runtime overhead of using IOTA's auto-tuning selection logics is either negligible, or even a slight speed-up instead. Another interesting result of this experiment is seeing that the relative performance of *default* and *mca* change depending on the chosen collective call. By examining the source code of both methods, we suspect this is due to the size of each decision tree. The performance gap is, however, very narrow for all observed cases and should not be expected to significantly affect performance.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub




(a) 4 Byte (b) 256 Byte (c) 1024 Byte

Figure 5.8: Comparing the runtime of different selection methods for `MPI_Allreduce` at $32 \times 1$ processes.




(a) 4 Byte (b) 256 Byte (c) 1024 Byte

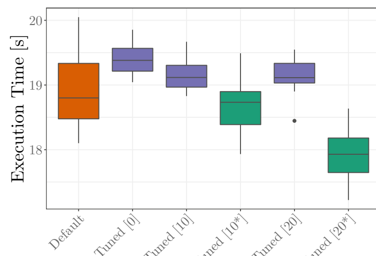
Figure 5.9: Comparing the performance of different selection methods for `MPI_Bcast` at $32 \times 1$ processes.

### 5.4.3 Tuning miniAMR from different miniAMR cases on *Hydra*

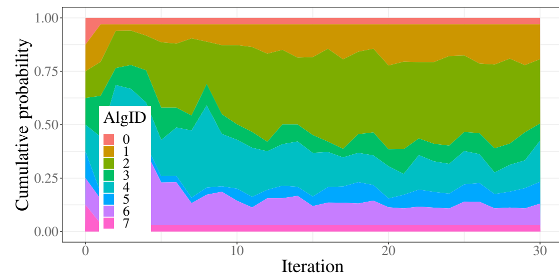
We now present the result of performing a full tuning run on *Hydra* with the miniAMR ECP proxy application. The basis for such a tuning run is selecting which applications at which sizes and parameter configurations to execute. This is to construct work similar to real-world data in a reproducible and understandable way. For this, we select running miniAMR on multiple different numbers of nodes and different numbers of processes per node. For tuning, we randomly selected ten runs of miniAMR with 4, 8, 16, or 32 nodes and 1 or 32 processes per node, per iteration. The exact commands to run miniAMR at these problem sizes are located in Appendix B for space reasons and to avoid duplication. A single training iteration consists of ten miniAMR runs as well as a model update phase. Every ten iterations, we evaluate the maximum achievable performance, by using the



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

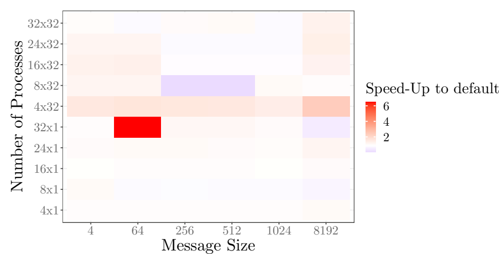



(a) Boxplots showing progress running miniAMR on $32 \times 32$ processes.

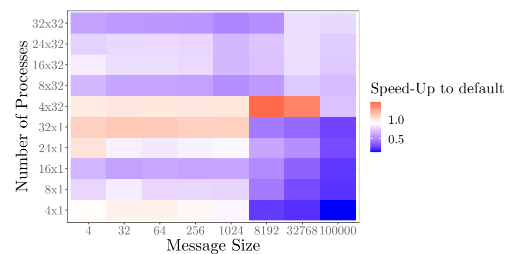



(b) Probability trend for 16 bytes of `MPI_Allreduce` on $24 \times 32$ processes.

Figure 5.10: Tuning progress on *Hydra* after 20 training iterations as shown in our previous work [31].




(a) `MPI_Allreduce`




(b) `MPI_Bcast`

Figure 5.11: 2D Grid comparing how well the selected algorithm compares to the default selection with different parameter values.

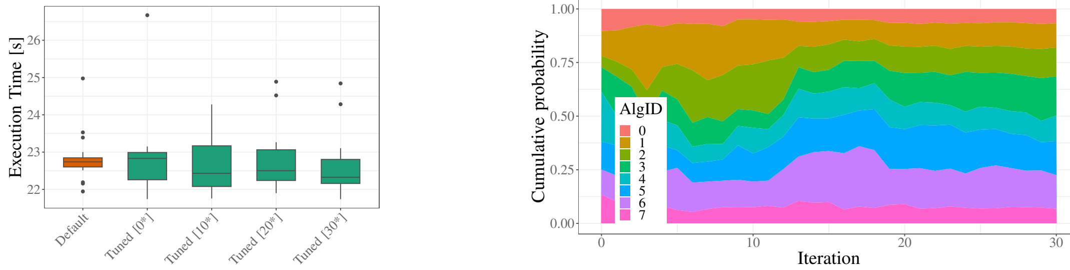
*tuned* algorithm selection logic. A known problem instance of miniAMR at size $32 \times 32$ processes is executed, and the total time logged without using this timing data to inform the future learning process.

Figure 5.10a shows these results, which we take have previously presented [31]. In this visualization, the three different colors of box plots refer to the used algorithm selection logic. Red refers to the *default* selection. Violet selects algorithms using the *tuning* selection logic based on the selection models as they are present after $i$ iterations. Finally, the green box plots with asterisks in their label represent the evaluation runs which use the *tuned* selection logic, also based on the current selection model. Green always picks the best known algorithm, whereas violet still gathers information. After 20 tuning iterations, our auto-tuned selection strategy can outperform the one shipped with Open MPI on the same MPI application but with unseen parameter values. Additionally, the overhead in the tuning steps compared to the default selection does keep relatively low but can be lowered further by specifying meta-parameters like the number of algorithmic choices to inject or the number of processes that are to measure the time.

To visualize the trend of which algorithm to prefer over time, Figure 5.10b visualizes the cumulative probability mapping a larger probability to a broader colored band. To more



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub




(a) Boxplots from *Irene* executing unseen miniAMR configuration.

(b) Probability trend for 16 bytes of `MPI_Allreduce` learned on miniAMR.

Figure 5.12: Tuning progress on *Irene* after 30 training iterations from miniAMR, evaluated on configuration unseen during training phase at $24 \times 48$ processes.

closely analyze how the tuning step has influenced the algorithm choice throughout the different possible parameter values, we compare the performance of this tuned selection to the default one across a suite of benchmarked data points in Figure 5.11a and Figure 5.11b. The message sizes were selected independently of the exact values used for miniAMR, as the goal is to infer which areas are highly tuned. It is not given that each cell in this grid was actually seen during training, leading to the need for inference. Given that miniAMR uses many `MPI_Allreduce` calls with a message size of 8 B, these evidently inform the case for 64 B.

The data shown in Table 5.3 indicates, that the majority of collective calls in miniAMR are `MPI_Allreduce`, which `MPI_Bcast` getting a far smaller fraction. This underlines why the predictions for `MPI_Allreduce` are shown to fit noticeably better than those for `MPI_Bcast`. A further point highlighted by this, is that the auto-tuner focuses training efforts directly on the cases most likely to impact overall running time. Through the adaptive nature of an auto-tuner, such a deficit could be alleviated, however, by measuring more `MPI_Bcast` collective calls.

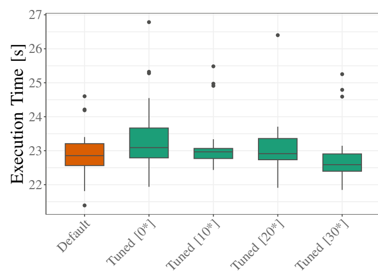
### 5.4.4 Tuning miniAMR from different miniAMR cases on *Irene*

The *Irene* compute cluster is significantly larger than *Hydra* at 1656 nodes, because of that, by tuning similarly to *Hydra*, subsequent training iterations are likely to have very different hardware allocations and resource utilization, leading in-turn to noise in the measured collective runtimes. We acknowledge that tuning this requires further changes to the auto-tuner. To counter this noise, we performed all following tuning runs on *Irene* by reserving a single resource allocation and performing all training iterations on the same allocation.

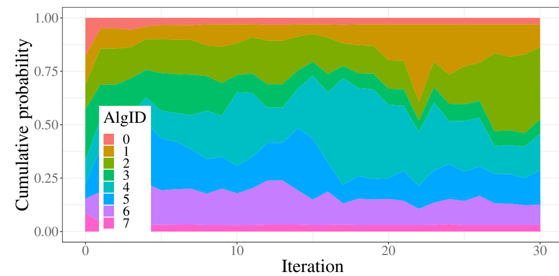
Figure 5.12a reflects the same training scenario as is presented in Figure 5.10a, except for changing the possible values for processes per node to 1 and 48 instead of 1 and 32. We see that after 30 training iterations, the tuned selection logic manages to outperform the default one. Showcasing similar results to the ones seen on *Hydra*.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN




(a) Boxplots from *Irene* learned on various proxies.




(b) `MPI_Allreduce` Probability trend for 16 bytes on $24 \times 48$ learned on various proxies.

Figure 5.13: Tuning progress on *Irene* after 30 training iterations from various proxies.

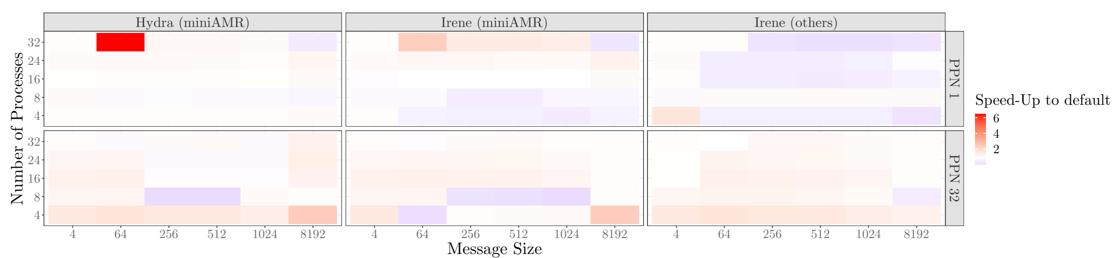



Figure 5.14: 2D Grid comparing how well the selected one compares to the default selection for `MPI_Allreduce` on 32 processes per node on *Hydra* trained on different systems.

### 5.4.5 Tuning miniAMR from application group on *Irene*

We now show the potential of our auto-tuner to apply tuning information gathered from a group of proxy applications to another unseen application, by performing tuning with all applications in Table 5.3 except for miniAMR and then predicting the algorithm selection on miniAMR with the same unseen processor setup $24 \times 48$. The application group tuning is performed on consists of: Laghos, Nekbone, miniVite, SWFFT, ExaMiniMD.

Similar to the tuning run on *Irene* in which inference from a set of miniAMR cases to an unseen miniAMR case was required, we observe an improvement in performance after 30 training iterations, as shown in Figure 5.13a. However, comparing the trend of selection probabilities for `MPI_Allreduce` on 16 bytes in Figure 5.13b, we can see that in this setting, the auto-tuner ended on a different distribution of values.

### 5.4.6 Transferring Learning between Compute Clusters

It is of interest how well training effort can be ported from one supercomputer to another one. For this, we execute the auto-tuner tasked with optimizing the performance of multiple proxy applications, not including miniAMR, on the *Irene* supercomputer and extract the probability distributions after 100 iterations. These are then used to



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

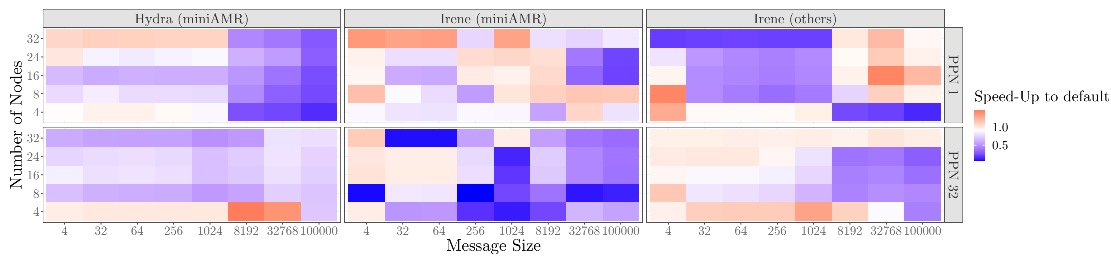



Figure 5.15: 2D Grid comparing how well the selected algorithm compares to the default selection for `MPI_Bcast` on 32 processes per node on *Hydra* trained on different systems.

execute benchmarks on the *Hydra* cluster and these results are compared with probability distributions obtained from 30 iterations of purely learning on miniAMR on *Hydra*.

For the `MPI_Allreduce` collective, in Figure 5.14, the best performance overall is achieved by tuning on the compute cluster that is used for execution later on, however, both versions tuned on *Irene* show generally to be tuned relatively close to the default selection logic, with only a few darker spots on the map. Whereas, `MPI_Bcast`, in Figure 5.15, shows a different scenario in which especially tuning from other proxy applications, making more use of this collective, manages to find good algorithms for some section of the parameter space.

### 5.4.7 Simulating the Auto-Tuning Workflow

The auto-tuner is optimized to run alongside MPI applications for an extended period of time, iteratively improving the algorithm selection quality. When determining whether this auto-tuner does function as expected and developing IOTA, it was necessary to verify this behavior in controlled runs. For this we used ECP proxy applications with various problem sizes and set IOTA up to re-train the models after a certain number of application runs. Such a process is the closest feasible one to deploying the auto-tuner on a production system, while still enabling iterative development. It does, however, come at the cost of requiring compute resources as well as a large amount of execution time, to mimic long term deployment.

Throughout the work on this thesis, the specifics of how the runtimes of a single collective call may start, end, and generally be distributed, were of interest. Just as, analyzing the imbalance in the processes' arrival patterns. To enable both fast test runs of the auto-tuner using faked data, and a better understanding of the data to fake, we developed a simulation backend for IOTA.

The simulation workflow fakes execution of an MPI application and tries to act as closely as possible to the way an actual MPI application would behave. The range of ways to tweak this includes defining the actual performance vectors for all algorithms and problem instances or specifying noise for the faked timing data. Meaning that each simulation run could train the auto-tuner to approximate a different true distribution of



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

arbitrarily selectable complexity. As such, it can act as a testing harness for IOTA and did help in determining the origins of buggy behavior.

When in the process of trying to determine whether some supposed property of the timing values does impact the tuning progress, the simulation workflow can be consulted to test the given scenario. As an example, when determining how big the impact of imbalance in the arrival times of processes in collective calls is on some specific algorithms, the simulation workflow could be used as a dry-run. From the learnings obtained here, better experiments can be performed in the future.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 6

# Conclusion

This thesis aimed to improve the quality of MPI collective algorithm selection through automatic tuning for a specific compute cluster. To assess this, three research questions were specified to be answered:

**R1:** What type of model can be applied to online algorithm selection with minimal overhead over MPI applications?

We constructed an auto-tuner injecting algorithms into proxy applications on a best-guess approach, recording the runtimes during each run. These runtimes are then used to build a model of collective runtimes which is compressed to be useful for online usage and repeated by injecting the choices of the updated model. Quantitative studies were performed to compare the predictions of this approach on two systems to the default algorithm selection.

The findings from applying our prototype indicate that live measurement of collective runtimes can be done with little overhead. Our approach of splitting the runtime modeling into two steps, first building a large accurate model and later compressing this for live selection, yields promising results.

**R2:** How much timing data is necessary to obtain stable performance improvements?

Our iterative online auto-tuner is shown to perform well on *Hydra* and *Irene*, with fixed allocations, outperforming the default selection on both tested systems after 30 training iterations on unseen parameter values. This was possible on both compute clusters, when sub-sampling the number of collective calls traced in each application run.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

**R3:** Can performance portability across compute clusters and across applications can be achieved using a tuned algorithm selection model?

We have observed that the auto-tuner, when trained on a list of proxy applications, can match or outperform the default selection in Open MPI 4.1.x for an unseen proxy application when executing on an unseen number of nodes on *Irene*. Similarly, evaluating the performance on micro-benchmarks after having tuned on different systems with different workloads, showed the auto-tuner to closely match the default selection for `MPI_Allreduce` and infer some tuning knowledge for `MPI_Bcast`, which was tuned for fewer collective calls throughout the training process.

We have shown that there exists a performance gap in the current default selection in Open MPI that can be addressed using auto-tuning. Experiments with a prototype for such a system demonstrate the feasibility of our auto-tuning approach on different systems. While this work provided answers, new avenues for further research have opened up. Some possible directions to evaluate in future works include:

- Reduction of noise in runtime in between tuning iterations.
- Improvement of runtime modeling assisted by the simulation workflow.
- Evaluate different machine learning models and pre-processing steps.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

APPENDIX A

# Algorithm Mapping

Table A.1: Mapping of our AlgID to an Open MPI algorithm and parameter configuration for `MPI_Allreduce`

| AlgID | Open MPI Algorithm | Segment Size |
|---|---|---|
| 1 | Basic Linear - 1 | N/A |
| 2 | Non-Overlapping - 2 | N/A |
| 3 | Recursive Doubling - 3 | N/A |
| 4 | Ring - 4 | N/A |
| 5 | Segmented Ring - 5 | 1 KiB |
| 6 | Segmented Ring - 5 | 1 MiB |
| 7 | Segmented Ring - 5 | 8 MiB |
| 8 | Rabenseifner's - 6 | N/A |

Table A.2: Mapping of our AlgID to an Open MPI algorithm and parameter configuration for `MPI_Bcast`

| AlgID | Open MPI Algorithm | Segment Size | Faninout |
|---|---|---|---|
| 1 | Basic Linear - 1 | 1 KiB | N/A |
| 2 | Chain - 2 | 1 KiB | 1 |
| 3 | Chain - 2 | 1 KiB | 4 |
| 4 | Chain - 2 | 1 KiB | 16 |
| 5 | Chain - 2 | 1 MiB | 1 |
| 6 | Chain - 2 | 1 MiB | 4 |
| 7 | Chain - 2 | 1 MiB | 16 |
| 8 | Chain - 2 | 8 MiB | 1 |
| 9 | Chain - 2 | 8 MiB | 4 |
| 10 | Chain - 2 | 8 MiB | 16 |
| 11 | Pipelined - 3 | 1 KiB | N/A |
| 12 | Pipelined - 3 | 1 MiB | N/A |
| 13 | Pipelined - 3 | 8 MiB | N/A |
| 14 | Split Binary Tree - 4 | 1 KiB | N/A |
| 15 | Binomial - 5 | 1 MiB | N/A |
| 16 | K-Nomial - 6 | 8 MiB | N/A |
| 17 | Scatter Allgather - 7 | 1 KiB | N/A |
| 18 | Scatter Allgather - 7 | 1 MiB | N/A |
| 19 | Scatter Allgather - 7 | 8 MiB | N/A |
| 20 | Scatter Allg. Ring - 8 | 1 KiB | N/A |
| 21 | Scatter Allg. Ring - 8 | 1 MiB | N/A |
| 22 | Scatter Allg. Ring - 8 | 8 MiB | N/A |

Table A.3: Mapping of our AlgID to an Open MPI algorithm `MPI_Allgather` without parameter configuration

| AlgID | Open MPI Algorithm |
|---|---|
| 1 | Basic Linear - 1 |
| 2 | Bruck's - 2 |
| 3 | Recursive Doubling - 3 |
| 4 | Ring - 4 |
| 5 | Neighbor Exchange - 5 |



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Table A.4: Mapping of our AlgID to an Open MPI algorithm and parameter configuration for `MPI_Alltoall`

| AlgID | Open MPI Algorithm | Max Requests |
|---|---|---|
| 1 | Basic Linear - 1 | N/A |
| 2 | Pairwise - 2 | N/A |
| 3 | Bruck's - 3 | N/A |
| 4 | Linear Sync - 4 | 0 |
| 5 | Linear Sync - 4 | 64 |
| 6 | Linear Sync - 4 | 128 |
| 7 | Linear Sync - 4 | 512 |

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX B

# Proxy Application Configurations

**Listing 4** miniAMR parameter configurations used for auto-tuning.

```
srun -N 4 -n 4 ./miniAMR.x --npx 2 --npy 2 --npz 1
    --nx 32 --ny 16 --nz 16 --num_tsteps 120
srun -N 4 -n 128 ./miniAMR.x --npx 8 --npy 4 --npz 4
    --nx 32 --ny 16 --nz 16 --num_tsteps 120
srun -N 8 -n 8 ./miniAMR.x --npx 4 --npy 2 --npz 1
    --nx 32 --ny 16 --nz 16 --num_tsteps 120
srun -N 8 -n 256 ./miniAMR.x --npx 8 --npy 8 --npz 4
    --nx 32 --ny 16 --nz 16 --num_tsteps 120
srun -N 16 -n 16 ./miniAMR.x --npx 4 --npy 2 --npz 2
    --nx 32 --ny 16 --nz 16 --num_tsteps 120
srun -N 16 -n 512 ./miniAMR.x --npx 8 --npy 8 --npz 8
    --nx 32 --ny 16 --nz 16 --num_tsteps 120
srun -N 32 -n 32 ./miniAMR.x --npx 4 --npy 4 --npz 2
    --nx 32 --ny 16 --nz 16 --num_tsteps 120
srun -N 32 -n 1024 ./miniAMR.x --npx 16 --npy 8 --npz 8
    --nx 32 --ny 16 --nz 16 --num_tsteps 120
```

**Listing 5** ExaMiniMD parameter configurations used for auto-tuning.

```
srun -N 4 -n 4 ./ExaMiniMD -il in.lj --comm-type MPI
srun -N 4 -n 128 ./ExaMiniMD -il in.lj --comm-type MPI
srun -N 8 -n 8 ./ExaMiniMD -il in.lj --comm-type MPI
srun -N 8 -n 256 ./ExaMiniMD -il in.lj --comm-type MPI
srun -N 16 -n 16 ./ExaMiniMD -il in.lj --comm-type MPI
srun -N 16 -n 512 ./ExaMiniMD -il in.lj --comm-type MPI
srun -N 32 -n 32 ./ExaMiniMD -il in.lj --comm-type MPI
srun -N 32 -n 1024 ./ExaMiniMD -il in.lj --comm-type MPI
```



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

**Listing 6** Nekbone parameter configurations used for auto-tuning.

```
srun -N 4 -n 4 ./nekbone ex1
srun -N 4 -n 128 ./nekbone ex1
srun -N 8 -n 8 ./nekbone ex1
srun -N 8 -n 256 ./nekbone ex1
srun -N 16 -n 16 ./nekbone ex1
srun -N 16 -n 512 ./nekbone ex1
srun -N 32 -n 32 ./nekbone ex1
srun -N 32 -n 1024 ./nekbone ex1
```

**Listing 7** SWFFT parameter configurations used for auto-tuning.

```
srun -N 4 -n 4 ./TestDfft 2 1024
srun -N 4 -n 128 ./TestDfft 2 1024
srun -N 8 -n 8 ./TestDfft 2 1024
srun -N 8 -n 256 ./TestDfft 2 1024
srun -N 16 -n 16 ./TestDfft 2 1024
srun -N 16 -n 512 ./TestDfft 2 1024
srun -N 32 -n 32 ./TestDfft 2 1024
srun -N 32 -n 1024 ./TestDfft 2 1024
```

**Listing 8** miniVite parameter configurations used for auto-tuning.

```
srun -N 4 -n 4 ./miniVite -l -n 512000 -w
srun -N 4 -n 128 ./miniVite -l -n 1024000 -w
srun -N 8 -n 8 ./miniVite -l -n 512000 -w
srun -N 8 -n 256 ./miniVite -l -n 1843200 -w
srun -N 16 -n 16 ./miniVite -l -n 512000 -w
srun -N 16 -n 512 ./miniVite -l -n 1843200 -w
srun -N 32 -n 32 ./miniVite -l -n 1024000 -w
srun -N 32 -n 1024 ./miniVite -l -n 18432000 -w
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

**Listing 9** Laghos parameter configurations used for auto-tuning.

```
srun -N 4 -n 4 ./laghos -pa -p 1 -rs 3 -tf 0.8 -m cube_311_hex.mesh
    --max-steps 100
srun -N 4 -n 128 ./laghos -pa -p 1 -rs 3 -tf 0.8 -m cube_311_hex.mesh
    --max-steps 250
srun -N 8 -n 8 ./laghos -pa -p 1 -rs 3 -tf 0.8 -m cube_311_hex.mesh
    --max-steps 250
srun -N 8 -n 256 ./laghos -pa -p 1 -rs 3 -tf 0.8 -m cube_311_hex.mesh
    --max-steps 250
srun -N 16 -n 16 ./laghos -pa -p 1 -rs 3 -tf 0.8 -m cube_311_hex.mesh
    --max-steps 250
srun -N 16 -n 512 ./laghos -pa -p 1 -rs 3 -tf 0.8 -m cube_311_hex.mesh
    --max-steps 250
srun -N 32 -n 32 ./laghos -pa -p 1 -rs 3 -tf 0.8 -m cube_311_hex.mesh
    --max-steps 250
srun -N 32 -n 1024 ./laghos -pa -p 1 -rs 3 -tf 0.8 -m cube_311_hex.mesh
    --max-steps 250
```



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX C

# Building and Executing Custom Open MPI

## C.1 Building Open MPI and the mpi_coll_tuner

**Listing 10** Commands necessary to compile the customized Open MPI implementation and the mpi_coll_tuner.

```
git clone git@github.com:sebastian-steiner/ompi_pmbs.git
git clone git@github.com:hunsa/mpi_coll_tuner.git

export PATH=$HOME/ompi_pmbs/build/bin:$PATH
export LD_PRELOAD=$HOME/mpi_coll_tuner/libmpi_coll_tuner.so

# edit $HOME/mpi_coll_tuner/CMakeLists.txt
# to set AT_CORES_PER_NODE and AT_TOTAL_NODES correctly

cd $HOME/ompi_pmbs
./autogen.pl
CFLAGS="-I$HOME/mpi_coll_tuner/include" ./configure
    --prefix=`pwd`/build --with-slurm=...
    --with-psm2=... --with-hwloc=... --with-pmix=...
make
make install -j

cd $HOME/mpi_coll_tuner
cmake .
make
make install -j
```



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

## C.2 Slurm Job Template

**Listing 11** Template for a job file used by the Slurm workload manager.

```
#!/bin/bash

#SBATCH -N 32
#SBATCH --ntasks-per-node 32

set -x

export PATH=$HOME/ompi_pmbs/build/bin:$PATH
export LD_LIBRARY_PATH=$HOME/ompi_pmbs/build/lib:$LD_LIBRARY_PATH
export LD_PRELOAD=$HOME/mpi_coll_tuner/libmpi_coll_tuner.so

export AT_MPI_COLL_TRACER_ENABLED=1
export AT_MPI_SELECT_MODE=1
export AT_MPI_COLL_TRACER_NB_PROC_SET=1024
export AT_MPI_TOTAL_TIMINGS_ENABLED=1
export AT_MPI_COLLS_TO_TRACE=1
export AT_MPI_COLL_SEED=`date +%s`

# execute traced application

export -n AT_MPI_COLL_TRACER_ENABLED
export -n AT_MPI_SELECT_MODE
export -n AT_MPI_COLL_TRACER_NB_PROC_SET
export -n AT_MPI_TOTAL_TIMINGS_ENABLED
export -n AT_MPI_COLLS_TO_TRACE
export -n AT_MPI_COLL_SEED
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

## C.3 *Irene* Job Template

**Listing 12** Template for a job file used by *Irene*'s version of the Slurm workload manager.

```
#!/bin/bash

#MSUB -n 1024
#MSUB -N 32
#MSUB -x

set -x

export SELFIE_MSUB=0
export SELFIE_MPRUN=0

export PATH=$HOME/ompi_pmbs/build/bin:$PATH
export LD_LIBRARY_PATH=$HOME/ompi_pmbs/build/lib:$LD_LIBRARY_PATH
export LD_PRELOAD=$HOME/mpi_coll_tuner/libmpi_coll_tuner.so

export AT_MPI_COLL_TRACER_ENABLED=1
export AT_MPI_SELECT_MODE=1
export AT_MPI_COLL_TRACER_NB_PROC_SET=1024
export AT_MPI_TOTAL_TIMINGS_ENABLED=1
export AT_MPI_COLLS_TO_TRACE=1
export AT_MPI_COLL_SEED=`date +%s`

# execute traced application

export -n AT_MPI_COLL_TRACER_ENABLED
export -n AT_MPI_SELECT_MODE
export -n AT_MPI_COLL_TRACER_NB_PROC_SET
export -n AT_MPI_TOTAL_TIMINGS_ENABLED
export -n AT_MPI_COLLS_TO_TRACE
export -n AT_MPI_COLL_SEED
```



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Bibliography


[1] Oak Ridge National Laboratory, "Frontier user guide." `https://docs.olcf.ornl.gov/systems/frontier_user_guide.html`, 2022. [Online; accessed 2023-01-18].

[2] RIKEN Center for Computational Science, "About Fugaku." `https://www.r-ccs.riken.jp/en/fugaku/about/`, 2022. [Online; accessed 2023-01-18].

[3] B. Klenk and H. Fröning, "An overview of MPI characteristics of exascale proxy applications," in *Proceedings of the 32nd ISC High Performance* (J. M. Kunkel, R. Yokota, P. Balaji, and D. E. Keyes, eds.), vol. 10266 of *LNCS*, pp. 217–236, Springer, 2017.

[4] G. Rünger and T. Rauber, *Parallel Programming - for Multicore and Cluster Systems; 2nd Edition*. Springer, 2013.

[5] C. Lameter, "An overview of non-uniform memory access," *Commun. ACM*, vol. 56, no. 9, pp. 59–54, 2013.

[6] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *35th International Symposium on Computer Architecture (ISCA 2008), June 21-25, 2008, Beijing, China*, pp. 77–88, IEEE Computer Society, 2008.

[7] D. W. Walker, "The design of a standard message passing interface for distributed memory concurrent computers," *Parallel Comput.*, vol. 20, no. 4, pp. 657–673, 1994.

[8] W. Zhang, "OpenMPI 4.1.x - Change the default collective algorithm selection." `https://github.com/open-mpi/ompi/commit/5a13c5352f5aa054fcc0222d5db66791602a6231`, 2020.

[9] F. Hutter, L. Kotthoff, and J. Vanschoren, eds., *Automated Machine Learning - Methods, Systems, Challenges*. The Springer Series on Challenges in Machine Learning, Springer, 2019.

[10] I. Laguna, R. J. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A large-scale study of MPI usage in open-source HPC applications," in *Proceedings of*




Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

*the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019* (M. Taufer, P. Balaji, and A. J. Peña, eds.), pp. 31:1–31:14, ACM, 2019.

[11] S. B. Guerra, R. B. C. Prudêncio, and T. B. Ludermir, "Predicting the performance of learning algorithms using support vector machines as meta-regressors," in *Artificial Neural Networks - ICANN 2008 , 18th International Conference, Prague, Czech Republic, September 3-6, 2008, Proceedings, Part I* (V. Kurková, R. Neruda, and J. Koutník, eds.), vol. 5163 of *Lecture Notes in Computer Science*, pp. 523–532, Springer, 2008.

[12] K. Leyton-Brown, E. Nudelman, and Y. Shoham, "Learning the empirical hardness of optimization problems: The case of combinatorial auctions," in *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings* (P. V. Hentenryck, ed.), vol. 2470 of *Lecture Notes in Computer Science*, pp. 556–572, Springer, 2002.

[13] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham, "Understanding random SAT: beyond the clauses-to-variables ratio," in *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings* (M. Wallace, ed.), vol. 3258 of *Lecture Notes in Computer Science*, pp. 438–452, Springer, 2004.

[14] F. Hutter and Y. Hamadi, "Parameter adjustment based on performance prediction: Towards an instance-aware problem solver," *Microsoft Research, Tech. Rep. MSR-TR-2005-125*, 2005.

[15] S. Pellegrini, J. Wang, T. Fahringer, and H. Moritsch, "Optimizing MPI runtime parameter settings by using machine learning," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7-10, 2009. Proceedings* (M. Ropo, J. Westerholm, and J. J. Dongarra, eds.), vol. 5759 of *Lecture Notes in Computer Science*, pp. 196–206, Springer, 2009.

[16] J. Pješivac-Grbović, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, "MPI collective algorithm selection and quadtree encoding," *Parallel Comput.*, vol. 33, no. 9, pp. 613–623, 2007.

[17] J. Pješivac-Grbović, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, "Decision trees and MPI collective algorithm selection problem," in *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007, Proceedings* (A. Kermarrec, L. Bougé, and T. Priol, eds.), vol. 4641 of *Lecture Notes in Computer Science*, pp. 107–117, Springer, 2007.

[18] S. Hunold and A. Carpen-Amarie, "Algorithm selection of MPI collectives using machine learning techniques," in *Proceedings of the IEEE/ACM Workshop on Per-*



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

*formance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS@SC)*, 2018.

[19] S. Hunold, A. Bhatele, G. Bosilca, and P. Knees, "Predicting MPI collective communication performance using machine learning," in *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER )*, pp. 259–269, IEEE, 2020.

[20] M. Wilkins, Y. Guo, R. Thakur, P. A. Dinda, and N. Hardavellas, "Acclaim: Advancing the practicality of MPI collective communication autotuning using machine learning," in *IEEE International Conference on Cluster Computing, CLUSTER 2022, Heidelberg, Germany, September 5-8, 2022*, pp. 161–171, IEEE, 2022.

[21] A. Faraj, X. Yuan, and D. K. Lowenthal, "STAR-MPI: self tuned adaptive routines for MPI collective operations," in *Proceedings of the International Conference on Supercomputing (ICS)*, pp. 199–208, ACM, 2006.

[22] A. Faraj, P. Patarasuk, and X. Yuan, "A study of process arrival patterns for MPI collective operations," in *Proceedings of the 21th Annual International Conference on Supercomputing, ICS 2007, Seattle, Washington, USA, June 17-21, 2007* (B. J. Smith, ed.), pp. 168–179, ACM, 2007.

[23] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *J. Artif. Intell. Res.*, vol. 4, pp. 237–285, 1996.

[24] K. B. Ferreira and S. Levy, "Evaluating MPI message size summary statistics," in *EuroMPI/USA '20: 27th European MPI Users' Group Meeting, Virtual Meeting, Austin, TX, USA, September 21-24, 2020* (W. Bland, K. Mohror, and T. Pena, eds.), pp. 61–70, ACM, 2020.

[25] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, and G. Yang, "The sunway taihulight supercomputer: system and applications," *Sci. China Inf. Sci.*, vol. 59, no. 7, pp. 072001:1–072001:16, 2016.

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

[27] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods & evaluation," *Artif. Intell.*, vol. 206, pp. 79–111, 2014.

[28] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016* (B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, and R. Rastogi, eds.), pp. 785–794, ACM, 2016.

[29] Exascale Computing Project Team, "ECP Proxy Applications." `https://proxyapps.exascaleproject.org/`. [Online; accessed 2022-11-24].



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[30] N. Sultana, M. Rüfenacht, A. Skjellum, P. V. Bangalore, I. Laguna, and K. Mohror, "Understanding the use of message passing interface in exascale proxy applications," *Concurr. Comput. Pract. Exp.*, vol. 33, no. 14, 2021.

[31] S. Hunold and S. Steiner, "Ompicolltune: Autotuning MPI collectives by incremental online learning," in *IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, PMBS@SC 2022, Dallas, TX, USA, November 13-18, 2022*, pp. 123–128, IEEE, 2022.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub