University of Innsbruck

Doctoral Thesis

On Simplifying and Optimizing Message Passing Programs: A Compiler and Runtime-Based Approach

Author:
Simone Pellegrini

Advisor:
Prof. Dr. Thomas Fahringer

submitted to the faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfilment of the requirements for the degree of doctor of science

in the

Institut für Informatik

November 2011

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wórtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht. Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum	Unterschrift

"Die schlechtesten Leser sind die, welche wie plündernde Soldaten verfahren: Sie nehmen sich einiges, was sie brauchen können, heraus, beschmutzen und verwirren das übrige und lästern auf das Ganze."

"The worst readers are those who behave like plundering troops: they take away a few things they can use, dirty and confound the remainder, and revile the whole."

Friedrich Wilhelm Nietzsche

UNIVERSITY OF INNSBRUCK

Abstract

Doctor of Philosophy

On Simplifying and Optimizing Message Passing Programs: A Compiler and Runtime-Based Approach

by Simone PELLEGRINI

In High-Performance Computing (HPC), the high demand for computational power is satisfied only by large distributed memory systems. In order to fully harvest the potentials of such super-computers, programmers resort to low-level programming models such as message passing. However, writing portable and scalable distributed applications is known to be difficult, error-prone, and time consuming.

This thesis explores static and dynamic techniques to automatically improve the performance behaviour of existing distributed memory programs. Moreover, it discusses novel ideas aiming at both simplifying the programming effort and improving code scalability. Throughout this thesis, the Message Passing Interface (MPI) is considered as an implementation of the message passing model.

Among others, the thesis presents a distributed runtime system, called *libWater*, aimed at heterogeneous systems. It combines the message passing and the Open Computing Language (OpenCL) programming model to realize a simple, but yet powerful, Application Programming Interface (API) which hides the distributed nature of the underlying system to the programmer. High scalability is made possible by various optimizations within the runtime system which dynamically detect and rewrite communication patterns in a more efficient way.

This thesis also examines ways to improve the performance of a message passing applications without the need of recompiling it. We propose a technique which aims at the customization of an instantiation of the MPI library, by tuning available runtime parameters, to suite the target architecture and input program.

Among the static approaches we present a novel analysis which uses elements of the Polyhedral Model (PM) to match communication statements. By using heuristics, our method can analyze rather complex codes which cannot be handled by any of the previous approaches. This analysis represents an important pre-requisite for any static compiler transformation which can be leveraged for future work.

Acknowledgements

When looking back throughout my years of study, I clearly see few people who showed me the direction and shaped me into the individual I am today; and many other who supported me during though periods. First of all, I would like to express my gratitude to my supervisor Prof. Thomas Fahringer for the useful comments, remarks and engagement through the learning process of this doctorate thesis. Without his direction this achievement would not be possible. Similar gratitude goes to Prof. Torsten Hoefler who constantly challenged me to be the best researcher I could be. It has been an honor working so closely together. Among the mentoring figures I would also like to thank Hans Moritsch, Biagio Cosenza and John Thomson for the interesting and constructive discussions.

I also had the fortune to meet several great people who helped me to keep my mental sanity during these long years. A special thank goes to some of my closest friends, Simon Ostermann, Kassian Plankensteiner and Ivan Grasso. Peter Thoman who, beside his preferences of using STL's lists instead of vectors, I highly esteem. Ferdinando Alessi, Timo Schneider and the Insieme compiler team members (current and past) which put great efforts in making this project a success.

Finally, my biggest thank goes to my family and my wife Lia. Without their support and love this achievement would be meaningless. I dedicate this thesis to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The needs of many scientific applications often cannot be satisfied by a single computing system. Typically, the limited amount of memory that can be fitted in a single system bounds the input problem size used to run an application. Additionally, these algorithms are usually computational intensive (or *computational-bound*) and they can be significantly improved if *parallelized*. That means that a sequential algorithm is split into many parallel flows which collaborate together to produce a final result. Symmetric Multiprocessor System (SMP) machine allows for multiple flows of execution within a single machine. However, there is a constraint on the number of computational units due to physical and energy factors making scaling the computational power of a single computing system expensive and not practical.

The problem of assigning many computing resources to an application has been addressed by connecting many homogeneous systems together to form a so called *distributed memory system*. A generic definition for a distributed memory system was introduced by [2]:

> *A distributed memory system is a collection of independent computers that appears to its users as a single coherent system.*

One class of distributed memory systems which are of particular importance for us, and the field of HPC in general, are *Clusters*. Clusters are characterized by the underlying hardware consisting of a collection of similar homogeneous workstations, running the same operating system, closely connected by means of high-speed local-area networks.

Effectively programming such distributed memory systems poses several challenges which will be the focus of this thesis. In this work we look at one of the prevalent paradigms used to program applications for a cluster system, i.e., *message-passing*. It provides a

minimal abstraction layer leaving the full control over communication and synchronization to the developer. MPI [3] is on one implementation of the message passing paradigm which is a *de-facto* standard in HPC systems.

1.1 Motivation

The message passing paradigm offers few basic primitives. This allows for a very low learning curve; however writing *efficient* and *error-free* message passing programs is known to be a difficult task. Moreover, the interfaces are designed with the goal of making distributed memory programs agnostic to the underlying architecture. However, features of the target platform, or the implementing library may prefer a particular style to another.

The effort of the MPI forum is to standardize a generic message passing interface for HPC, not its implementation. As of today, several production-quality implementations exist, e.g., Open MPI, MPICH, and MPIVACH. Internally, these implementations employ various kind of optimizations making it strenuous for developers to achieve the same level of performance across them. For example, MPI libraries often use buffering to reduce latency. This means that for a program written to rely on a particular buffering amount will be difficult, if not impractical, to be ported to a different implementation.

Usually a program needs tuning to best fit a specific target platforms. This work is conducted by experts which can achieve that by matchmaking the characteristics of the algorithm, the features of the underlying platform and message passing library. Tuning of a parallel distributed memory program can be very challenging and expensive. Tracing and post-analysis tools are often employed wasting precious cycles of production clusters.

The huge efforts required to write efficient and error-free message passing programs could be dramatically reduced by relying on automatic tools. A significant similar problem is the translation of high-level programming language constructs to assembler instructions the physical processing unit understands. In particular application areas is not uncommon, for specialists, to tune the code to address specific architectures. However, most of the applications rely on compilers which, beside mapping complex constructs into basic operations, they try to optimize the process leveraging unique features of the underlying processor to fully exploit its capabilities. Compilers are often coupled with runtime systems which can deliver further optimizations during the execution of a program. These are two major areas where the HPC community has been focusing its efforts for several decades. The object of these thesis is to contribute with new approaches and improve

upon existing techniques with particular regard to distributed memory programs based on the message passing paradigm.

1.2 State of the Art

Supporting the development and deployment of distributed memory programs has been a research topic for decades. In particular we intent to contribute upon three particular areas:

Simplification: As stated previously, the message passing paradigm has a very low learning curve since it involves very few concepts (i.e. processes, communication channels and messages) for a programmer to grasp. However, writing an error-free and performance efficient program can be challenging since the compiler often does not understand the structure of the parallel algorithm. For this reason efforts has been done in order to simplify its usage. Some of the approaches presented in literature focus on the API of the message passing paradigm [4]. The idea is to design the interfaces in a way that errors can be avoided, or enable the compiler – by means of annotations – to capture them. Alternative interfaces for MPI have been presented over the years, and example is represented by Boost.MPI [5] and the Object-Oriented MPI (OOMPI) [6].

A more elaborated effort is instead the definition of new programming models for distributed memory introducing higher level abstractions which are easier to manage with in a program. For example the Partitioned Global Address Space (PGAS) model [7] belongs to this category. PGAS-based languages introduce few extensions which make distributed memory programming easier (since most of the effort of producing low-level communications is offloaded to a compiler), however they did not reach the level of adaptation expected due to the poor scalability of the generated code. One of the latest advancement in this area is the definition of the OpenCL programming model OpenCL. OpenCL targets non-distributed heterogeneous architectures, in this thesis we realize a distributed runtime system which enables OpenCL programs to target heterogeneous clusters.

Runtime: In HPC, implementations of the message passing paradigm try to adapt to the underlying cluster architecture in a way which is transparent to the program. Many efforts exist in literature which target this aspect of message passing libraries. For example, coalescing of many short messages, is employed to reduce latency and network injection rate [8]. Detection of communication patterns is also another interesting research area which replaces single point-to-point communications into

more complex collective patterns (e.g. broadcast, gather or scatter) which can be natively supported by an advanced message passing library like MPI [9, 10]. These runtime systems also offer several dozens tunable parameters which can be customized to improve the performance of a given application on a selected target architecture. The advantage of this solution is that the performance of a program can be improved without the need to change or recompile the input program. Work in literature use rudimentary techniques to address this problem [11]. Thus several hundred evaluations of the input program are needed to find a satisfying setting of the input parameters. In this thesis we set to explore more sophisticated techniques, based on evolutionary algorithms, ML and statistical analysis [12–14], with the aim to reduce the search costs.

Compiler: Several work exist which approach the problem of optimizing message passing programs using compiler technology. In runtime based approaches, analysis has to be conducted during the execution in order to determine when a certain optimization can be carried out. It is important that any of the overhead introduced by this dynamic analysis is neutralized by the optimization. Static approaches do not suffer from this problem since the cost of the analysis is transfered at compilation time with no impact on the execution. Several work exist in literature using compiler technology for hoisting and coalesce communications [15, 16]. Also detection of communication patterns and their replacement has been the subject of many work in literature [17, 18]. However, compiler based approached did not find large application in mainstream compilers due to their complexity and scarce maturity.

1.3 Thesis Goals

Writing a deadlock-free distributed memory program is often cumbersome and the compiler provides very little help in spotting parallelization errors. Furthermore, programs often use the SPMD paradigm, leveraging control-flow statements to assign work to each processor thereby affecting performance. In this thesis we use features of the C++ programming language, such as template meta-programming and Object-Oriented Programming (OOP), to introduce new abstractions **with the purpose of transforming the message passing programming model from an imperative to a declarative style** offloading generation of optimized message passing code to the C++ compiler.

The OpenCL programming model emphasizes a programming style which relies on a runtime system to dispatch commands asynchronously to devices. Although OpenCL was thought with a single heterogeneous computing system in mind, its extension to

distributed memory systems can be achieved with an enhanced runtime system without the need of altering the programming interface. This thesis presents an **OpenCL distributed runtime system which aims at latency hiding and low resource utilization**.

The performance of message passing programs is very sensitive in terms of the values settings for its runtime parameters. Tuning these parameters manually is infeasible as there are hundreds of them and it requires weeks and months to determine efficient parameter values. In this thesis we describe a novel method for **auto-tuning of message passing runtime parameters** that deals with this problem.

Placing of communication statements within a program can be not trivial if caching effects are taken into account. However, one of the generalizations of this problem states that send operations should be issued as soon as possible and receives right before the utilization of the received value. In this thesis we explore the effects of message passing on caches and we **elaborate communication statement placement strategies that improve cache behaviour**. Furthermore, we present a mechanism to **maximize, automatically at compile time, communication over computation overlap** for any message passing program based on point-to-point communications.

One of the main challenges for the analysis of message passing programs is to statically determine, given a communication statement S, the set of statements that *match* with S. This information is important in order to understand how data is exchanged among processes, i.e., the *communication graph*. Since the evolution of a message passing program is non-deterministic, this problem presents serious challenges if addressed in a static approach. In this thesis we present **a novel algorithm based on a heuristic capable of determining matching information** eliminating some of the limits imposed by previous work.

1.4 Thesis Organization

This thesis is divided into four additional chapters. Chapter 2 introduces the concepts and the formalism which will be used throughout this work. In Section 2.1, the elements of a distributed memory system are introduced and formalized. Section 2.2 introduces the abstractions and the basic routines utilized to write parallel programs based on the message passing paradigm. The chapter concludes with Section 2.3 which introduce the concept of program and its representation within a compiler.

Chapter 3 focuses on simplification of distributed memory programming, three separate works are presented. First, in Section 3.2, some of the pitfalls common in the usage

of MPI are discussed and an alternative interface called MPP, based on C++, is presented. MPP's design goals are on minimal overhead, native support to user datatypes and simplicity of use. Section 3.3 deals with the message passing programming model, some of the typical performance problems caused by the use of message passing and the SPMD programming model are shown. A library approach, which leverages template meta-programming capabilities of the C++ compiler, is presented. This allows generation of optimized communication code from a declarative specification provided by the programmer. The last work, presented in Section 3.4 borrows concepts from the OpenCL programming model and it combines them with message passing to realizes a framework called libWater. libWater consists of a minimal API and a powerful distributed runtime system which embodies a technology that automatically recognizes inefficient communication patterns and transparently optimizes them at runtime.

Chapter 4 investigates optimization capabilities of the MPI library through the tuning of available runtime parameters. A motivational study is presented showing the impact that these parameters have on performance. Moreover, it is shown that best parameter settings usually depend on the application code, the target architecture and the input data. Three approaches are proposes in this chapter which aim to MPI runtime parameter tuning, each better suited for a different use case scenario. The first method, presented in Section 4.5, is based on evolutionary techniques. The given application code is executed against randomly generated parameter settings. Combinations with fastest execution times are recombined, by means of the crossover and mutation operators, to form new settings. In Section 4.6, the second technique is discussed which drastically reduces the number of executions of the given input code to one by employing advanced machine learning techniques to build a prediction model from training data. Last approach, in Section 4.7, moves the focus from single application codes to a class of applications with similar workload. With this method, based on statistical techniques, a single parameter setting, which is an optimal trade-off among a class of applications, is derived. Applications with similar workload automatically take advantage of this parameter setting, no additional runs are required.

In Chapter 5 focuses on static approaches leveraging compiler technology. Several analyses and transformations are presented. At first, in Section 5.2, the effects of point-to-point communications on the processor cache are analyzed and a series of optimization guidelines are derived, from benchmark data, that can be incorporated into a compiler to place communication statements in a way that cache reuse is maximized. In a second work, presented in Section 5.3, the advanced analysis capabilities of the PM are utilized to analyze point-to-point message passing statements within computation loops. By computing the instance-based data dependencies within the loop, the earlies and

latest possible placement of communication statements is detected. This method improves over previous techniques which use a more coarse-grained analysis result. Code is transformed so that the computation/communication overlap ratio is maximized and thus communication overhead is minimized. Section 5.4 discusses the last work of this thesis: a novel analysis for static matching of point-to-point communication statements. This approach uses elements of dataflow analysis and the polyhedral model to divide the input code into independent regions. Within those regions it discovers all possible matchings between communication statements through the use of the network flow algorithm. Compared to the state-of-the-art, this work is the first that can deal with non-blocking routines and wildcards.

Chapter 6 concludes this thesis providing the list of peer-reviewed publications which support the findings in this thesis and discussing future work.

Chapter 2

Model

In this chapter we present a formal model that defines the main concepts used throughout the whole thesis. Three separate models are presented: the hardware, the message passing and the program model. In Section 2.1, we formally describe the structure of a Cluster computing system, an important class of distributed memory systems employed in HPC. This represents the machine architecture targeted by the analyses and optimizations presented in this thesis. The message passing model, presented in Section 2.2, formally provides the paradigm used to program such distributed memory architectures. At last, in Section 2.3, the program model introduces the data structure typically employed by compilers to represent and manipulate programs.

2.1 Hardware Model

In this section, the elements composing a generic cluster computing system are defined using a bottom-up fashion. We thereby introduce the terminology which will be used throughout the rest of this thesis.

Definition 1 – *Computing Unit*

Let a *computing unit*, *cu*, be part of a computer system that carries out arithmetic, logical, and decision-making operations. Physically, such entity is implemented by a single core of a Central Processing Unit (CPU) or an accelerator core of a GPU. Typical a *cu* contains a number of registers and redundant functional units to achieve an increased Instruction Level Parallelism (ILP) which is typical of *superscalar* architectures. The set of all computing units is denoted by $\mathcal{CU}$. A *cu* is mapped to a physical location within a topology, represented by an integer tuple, through the bijective function $\mathsf{map} : \mathcal{CU} \rightarrow \mathbb{N} \times \mathbb{N} \times \ldots \times \mathbb{N}$. The function is generic to allow the representation of arbitrary *topologies*.

A *cu* is usually characterized by the peek amount of performed instructions per second (IPS). In HPC, the capabilities of a *cu* is better defined by the number of FLoating-point Operations Per Second (FLOPS). Scientific applications heavily rely on single- and double-precision floating point values. FLOPS is an important unit which is used to rank world's fastest supercomputers [19].

Definition 2 – *Chip*

Let a *chip*, c, be an integrated circuit packaging multiple instances of the same *cu*. Examples of such entity are multi-core CPUs and GPUs. The set of chips within a computing system is denoted by $\mathcal{C}$. Within a chip, a *cu* is identified by a unique identifier. For such purpose we can define the bijective function $\mathsf{map_{cu}} : \mathcal{CU} \rightarrow \mathcal{C} \times \mathbb{N}$. Since multiple chips can be hosted in a computing system, addressing is done through the function $\mathsf{map_c} : \mathcal{C} \rightarrow \mathbb{N}$ function. Hence, within a computing system, mapping of a *cu* is obtained by combining the functions $\mathsf{map} = \mathsf{map_{cu}} \circ \mathsf{map_c} : \mathcal{CU} \rightarrow \mathbb{N} \times \mathbb{N}$. In the produced tuple (i, j) the first dimension, i, refers to the chip identifier whereas the second dimension, j, is the position of the mapped *cu* within the chip.

For example a computing system composed of two dual-core CPU chips (i.e., 4 *cu*s) and one GPU with a single computing unit (i.e., gu_1) is represented as follows:

$$\mathsf{map} : \begin{cases} cu_1 \rightarrow (0,0) \\ cu_2 \rightarrow (0,1) \\ cu_3 \rightarrow (1,0) \\ cu_4 \rightarrow (1,1) \\ gu_1 \rightarrow (2,0) \end{cases}$$

As a convention, the GPU chips are always listed after the CPUs.

In this thesis we solely address *homogeneous* chips, this means that within a chip the computing units are all of the same type, e.g., multi-core CPUs such as AMD Opteron, Intel Xeon and IBM Power. It is worth noting that, the architecture and programmability aspects of homogeneous chips are different from *heterogeneous* ones, also known in literature as System on Chips (SoCs). In such systems, different *cu* types are combined into a single package. SoCs represent an important development of micro-processor technology which is being driven by embedded systems.

Definition 3 – *Memory Unit*

Let a *memory unit*, *mu*, be the part of a computer system where information and instructions are fetched and stored. The set of all memory units in a system is denoted by $\mathcal{MU}$. A *mu* can be private to a *cu* or shared among several of them.

A *mu* is mapped to the set of entities (i.e., computing or memory units) which can access its content. Hence, $\mathsf{map_{mu}} : \mathcal{MU} \rightarrow \{(\mathcal{CU} \cup \mathcal{MU})\}$.

For example, a system which contains a single memory, *mm*, shared among all computing units, i.e., cu_1, cu_2, cu_3, has a mapping function defined as follows: $\mathsf{map_{mu}} : \{mm\} \rightarrow \{cu_1, cu_2, cu_3\}$. We identify with the term *main memory*, *mm*, an instance of *mu* which is directly or indirectly accessible by every chip of a computing system.

Memory units can be organized into *hierarchies* such that small, fast memories (i.e., registers and caches) nearby a *cu* act as staging areas for a subset of the data and instructions stored in the relatively slow main memory. Coherency between such hierarchies is maintained by means of hardware *cache coherency protocols* (e.g., MESI, MOESI) [20].

Definition 4 – *Cache*

We denote with the term *cache* a generic memory unit whose coherence with the main memory is automatically managed via hardware protocols and thus not controlled by the programmer. Caches are usually structured into hierarchies. *Cache levels* are identified with the abbreviation `Li`, $i \in \mathbb{N}$. Each *mu* is mapped to a cache level via $\mathsf{map_{cl}} : \mathcal{MU} \rightarrow \mathbb{N}$. We identify with $\mathcal{MU}_{\texttt{Li}}$ the set of *mu*s mapped to cache level i. Cache hierarchies are structured in a way such that the content of memory units at level i, i.e., $cnt(\mathcal{MU}_{\texttt{Li}})$ have the following property: $cnt(\mathcal{MU}_{\texttt{L1}}) \subseteq cnt(\mathcal{MU}_{\texttt{L2}}) \subseteq \ldots \subseteq cnt(\mathcal{MU}_{\texttt{LN}})$.

For example, in a system with three computing units, $\mathcal{CU} = \{cu_1, cu_2, cu_3\}$, where each core has a private `L1` cache, $\mathcal{MU}_{\texttt{L1}} = \{\, mu_1, mu_2, mu_3 \,\}$, the mapping function is defined as follows:

$$\mathsf{map_{mu}} : \begin{cases} \{mu_i\} \rightarrow \{cu_i\} \; \forall \, i \in [1,3] \\ \{mm\} \rightarrow \{mu_1, mu_2, mu_3\} \end{cases}$$

Lower cache levels are usually small and *private* to a single *cu* while higher level ones are significantly larger and *shared* among all computing units in a chip. In a hierarchy, memory modules in a cache level $\texttt{L}_i$ are directly interfaced with level $\texttt{L}_{i+1}$, *mu*s at the higher cache level are interfaced with the main memory. This type of memory structure has the characteristic that a processor can access its own local memory faster than non-local memory (i.e., memory local to another processor or memory shared between processors). This is known in literature as Non-Uniform Memory Access (NUMA). In particular, the class of coherent memory systems are known as Cache coherent NUMA (ccNUMA). The number of levels and their private/shared nature may change between

chips, therefore throughout the rest of the thesis such details will be provided for the different architectures used in the experimental sections.

Definition 5 – *Non-coherent memory unit*

A *scratchpad* memory or *local store* is a non-coherent memory unit which must be managed by the programmer and it is directly interfaced with the main memory. Scratchpad memories are usually utilized within accelerator devices to mitigate for the slow rate at which data can be transfered to and from a device. In such context, local store is referred to as *device memory*.

The principal purpose behind the use of caches and local stores is to address the so called *memory wall* problem. The growing disparity of speed between chips and main memory makes memory latency an overwhelming bottleneck in computer performance. An important reason for this disparity is the limited communication bandwidth beyond chip boundaries. By interposing several buffering stages between chips and *mm*, the time for which a *cu* is waiting for instructions and data is optimized. It is therefore the duty of the programmer (and the compiler) to take into account the memory layout (even in the presence of cache-coherency) of the target machine in order to produce performance efficient code.

Definition 6 – *Computing Node*

Let a *computing node*, *cn*, be a computer system (e.g., a workstation or a server) containing a number of chips, $\subseteq \mathcal{C}$, and memory units, $\subseteq \mathcal{MU}$, which share information through access to the *main memory*. The set of all computing nodes is denoted by $\mathcal{CN}$. Additionally, a Network interface Controller (NiC) allows a *cu* to physically interface to a network which interconnects the set of $\mathcal{CN}$. As for computing units, computing nodes are identified by a unique number by the function $\mathsf{map}_{\mathsf{cn}} : \mathcal{CN} \rightarrow \mathbb{N}$. Throughout this thesis we also refer to the subset of $\mathcal{CN}$ represented by homogeneous multi-chip systems with the term SMPs.

As part of this thesis we consider the network as a *transparent link* which enables the exchange of information between the main memory modules of two, or more, distinct computing nodes. In practise, network interconnects reflects an active field of research; these aspects are orthogonal to the techniques presented in this thesis. The arrangement of the various elements (e.g., computing nodes, links and cables) of a cluster is called a *network topology*. Examples of topologies for HPC systems are: *fat-trees*, *multi-dimensional meshes* and *toroids*, and *hyper-cubes* [21]. Beside the topology, another important characteristic of a network is the interconnecting *medium*.

Definition 7 – *Bandwidth*

A medium defines the rate at which information can travel from one computing node

to another and the allowed maximum possible distance a signal can travel without being compromised. Such transmission rate is also called network *bandwidth*, which is measured in bits per second (bps).

Definition 8 – *Latency*

Another important characteristic of the interconnection medium is the *delay* occurring between the transmission of a packet of data and its arrival to the destination node. This is also called *network latency*, which is measured in seconds. Latency is not only a function of the medium but also of the topology, NiC and software stack.

High-speed interconnects, like InfiniBand QDR, have a bandwidth of around 96Gbps and a latency of 140 ns. Cheaper network medium, such as Ethernet, currently has a bandwidth of 10Gbps and a latency of 2 to 4 μs.

The building blocks of a generic distributed memory system have been presented. Within this thesis we solely focus on a class of distributed memory systems called *clusters*.

Definition 9 – *Cluster*

Let us define a *cluster computer* that consists of a set of locally connected homogeneous computing nodes each of them running its own instance of an operating system (OS). Data, d ,between two nodes, a and b, is exchanged by sending a message containing a *copy* of data d from node a to node b through the network. Within a cluster, computing units are topologically mapped to a unique location through function $\mathsf{map} : \mathcal{CU} \rightarrow \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ function. Elements of the triplet (i, j, k) represent, respectively, the node identifier, i, the chip identifier within a computing node, j, and the relative position of a *cu* within the chip, k. Since we are not interested in modeling network topologies, the three-dimensional space defined by our mapping function suffices to describe physical cluster systems that we will be utilized in this thesis.

In Figure 2.1 an example of an heterogeneous cluster is shown. The cluster is composed of 2 compute nodes, cn_i and cn_j. The first node has 2 CPUs and one GPU. Each CPU is composed of 4 compute units which have identical architectures. Each *cu* has a private L1 cache (i.e., mu_{0-3} and mu_{5-8}) and the 4 *cu*s in one chip share an L2 cache (i.e., mu_4 and mu_9). mu_{11} is the main memory of the first compute node and it connects to the gpu_0 using a scratchpad memory, mu_{10}. The GPU is composed of 8 compute units. The second node of the cluster, cn_1 has a slightly different architecture. A much smaller GPU with no scratchpad memory is present and only a single quad-core CPU called cpu_3. Every compute unit in the diagram is annotated with a triplet representing the mapping onto this topology. It can be seen that every element can be addressed univocally.

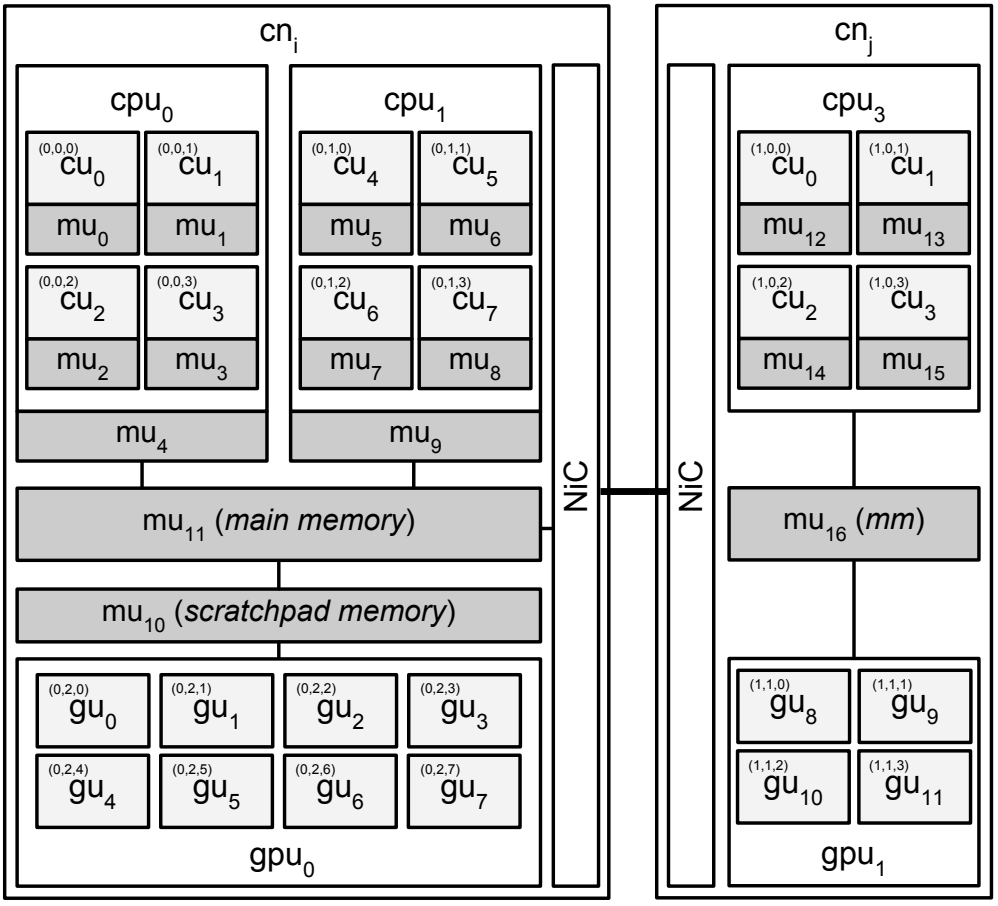


FIGURE 2.1: Example of a concrete cluster composed of 2 compute nodes, 3 CPUs and 2 GPUs and several memory modules.

2.2 The Message Passing Model

Programmability of distributed memory systems often relies on the message passing paradigm. A basic assumption is that the OS running in each computing node provides the means to access and manage node *resources*. The set of resources within a node is defined by the subset of $\mathcal{CU} \cup \mathcal{MU}$ which are mapped to the same node identifier. Since programming models do not directly address hardware entities, abstractions, usually provided by an OS, need to be defined.

Definition 10 – *Process*

Let a *process*, p, be an entity (or object) that models the *execution context* of a program. A program is composed of a sequence of instructions which are loaded into the main memory, see Section 2.3. A program becomes a process once moved into main memory by the OS scheduler. A process instance is created when a program is executed and terminated when the execution is complete. The OS allocates various *resources* to the process as the program execution evolves, e.g., memory units, file descriptors and compute units. The *execution context* of the process consists of this

resource allocation information, the current state of the program execution (e.g., the register values of the *cu* executing that program), the instruction that is to be executed next, and other information related to the program execution. The set of all processes is denoted by $\mathcal{P}$. A process is identified by a number, *pid* or *rank*, which is unique. With np we express the number of processes in $\mathcal{P}$, its cardinality $|\mathcal{P}|$, such as $0 \leq p < np$.

A process needs a *cu* to execute assigned instructions. In general, a process may use multiple *threads* of execution; however in this thesis we only focus on processes which are single threaded. Since a *cu* is an *exclusive* resource (because it cannot be used by more the one process at the time), the OS takes care of allocating these resources to processes in a *fair* and *effective* manner. This is also known as *resource scheduling*. Multiple *cu*s in a computing node allow for many processes to execute simultaneously. However, normally the number of processes in a system is larger than the number of *cu*s hence resources need to be *shared* or *multiplexed*. There are two ways to share or multiplex a resource: (*i*) *time-multiplexing*, where processes take turns in using the resource (e.g., processor), and (*ii*) *space-multiplexing*, where processes occupy different parts of the resource simultaneously (e.g., main memory).

In the context of cluster systems, we assume that there are always as many computing units as processes available. Practically, this means that a process is scheduled to a *cu* and owns that resource until completion. Instantiating more processes than available resources is a practice called node *over-subscription*. It is particularly useful for workloads which are I/O bound, HPC workload tend to be more CPU bound and therefore this practise is highly discouraged.

Definition 11 – *Processor Affinity*

The mapping of processes to a *cu*, which is done by the OS resource scheduler, can be customized by means of user policies. *Processor affinity* forces a process to be bound to a specific *cu* or a set of *cu*s. This allows the implementation of scheduling strategies which are aware of the application semantics.

Beside modeling processes, an OS also abstracts the *storage area* allocated to a process where information (such as code and data) are kept during execution.

Definition 12 – *Address Space*

Let an *address*, $addr_p$ with $p \in \mathcal{P}$, be a name or a number which uniquely identifies an *entity* in the main memory such as a constant, a variable or an instruction of process p. Let the set of entities used by a program p be $\mathcal{A}_p$, or *address space*, an area of the main memory for which the following property holds $\forall p_1, p_2 \in \mathcal{P}$ if $p_1 \neq p_2 \Rightarrow \mathcal{A}_{p_1} \bigcap \mathcal{A}_{p_2} = \emptyset$.

Definition 13 – *Sequential/Parallel Program*

A program which uses a single process with a single flow of execution is also called a *sequential* program. On the contrary, an application which requires multiple processes working together (referred to as *cooperating processes*) is a *parallel* program. In our model, cooperating processes are *loosely* connected in the sense that they have independent private address spaces and may run at different speeds.

We distinguish two main categories of parallel programs:

SPMD: A single program is executed by multiple process instances. Control statements select different parts to execute.

MPMD: Potentially separate programs which are executed by separate processes.

In both cases, processes *interact* among themselves by exchanging information. In shared memory programs this is done through writing and reading data onto a location shared among interacting processes. In distributed memory programs this information is packaged into *messages* which are exchanged through an interconnect medium (the computing network or the computing node bus). The producer of a message is called *sender*, and the consumer the *receiver*. A process may act both as a producer and a consumer during its lifetime.

Definition 14 – *Message*

Let us define a *message*, msg, as the tuple $(envelope, data)$. The *envelope* (or *header*) is the part of a message that contains information necessary for the correct delivery of the message from the sender to the receiver. This includes the sender and receiver process *ranks*, or its unique identifier, and the *communication context*. In our context we often assume there is always one communication context and it is represented by the set of all processes $\mathcal{P}$. The *data* (or *payload*) is instead the actual transmitted information. This part consists of a sequence of elements (which may be empty), each of them carrying information on its data type.

2.2.1 Point-to-Point Primitives

As stated before, messages travel thought a network, if the processes are not located in the same computing node. Since the programming model abstracts from such low-level hardware details we introduce the concept of *communication channel* which models the medium interconnecting two process instances. Practically, a channel is said to be *intra-node* if it logically connects two processes in the same computing node; an *inter-node* channel connects instead processes belonging to different computing nodes.

Definition 15 – *Channel*

Let us define a *channel*, $ch(p_i, p_j)$ with $0 \leq i, j < np$, as a bidirectional virtual connection between two *endpoints*, p_i and $p_j \in \mathcal{P}$. An endpoint is always a process, hence p_x is a process identifier. The purpose of a channel is to deliver a message from one endpoint to the other. Moreover, the channel abstraction provides two important properties on which the message passing programming model relies: (*i*) *guaranteed delivery* of a message, and (*ii*) *non-overtaking* of messages. Therefore a channel guarantees that messages arrive with the same order of dispatch in a First In, First Out (FIFO) fashion. Many channels can exist between the same pair of processes, we call this set $\mathcal{CH}(p_i, p_j)$. The *commutative* property holds, such that $\mathcal{CH}(p_i, p_j) = \mathcal{CH}(p_j, p_i)$. The set of all channels within a parallel application is denoted with $\mathcal{CH}$.

A channel provides five fundamental primitives:

Definition 16 – *send*

Let us define the routine

$$\textbf{send}(addr_p, size, p_{dest}, ch)$$

When invoked by a generic process $p \in \mathcal{P}$, the semantics of the communication primitive is the following. p transfers $size$ bytes from the address $addr_p \in \mathcal{A}_p$ towards destination process p_{dest} through the channel ch_c. This assumes that there exists a channel $ch(p, p_{dest}) \in \mathcal{CH}(p, p_{dest})$. If not, the channel is created. The semantics of the *send* primitive can be formally represented as follows:

$$\forall i \in [0, size) \Rightarrow addr_{p_{dest}}[i] := addr_p[i]$$

Where the := symbol denotes the *assignment operator*.

If p_{dest} is not ready for receiving the data then p blocks (within the *send* routine) until p_{dest} has posted a receive operation (i.e., *recv*) on the same communication channel.

Definition 17 – *recv*

Let us define the routine

$$\textbf{recv}(addr_p, size, p_{src}, ch)$$

When invoked by a generic process $p \in \mathcal{P}$, the semantics of the communication primitive is the following. p receives $size$ bytes from source process p_{src} and stores them starting from the address $addr_p \in \mathcal{A}_p$. The transfer happens through the

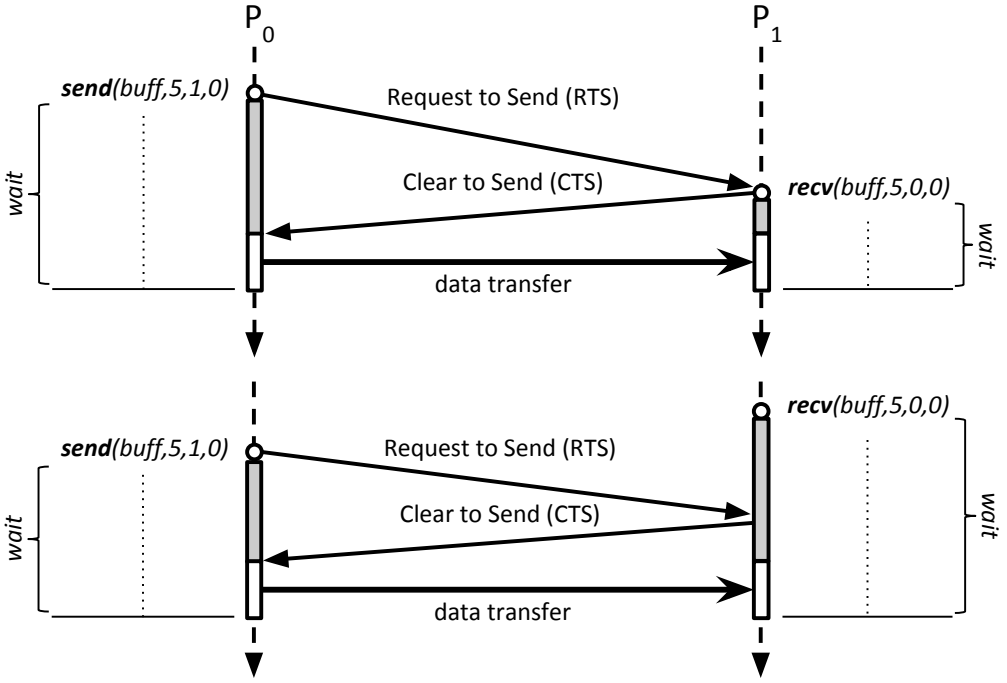


FIGURE 2.2: *Communication protocol* used for coordinating sender and receiver for point-to-point blocking communications.

channel $ch(p_{src}, p) \in \mathcal{CH}_(p_{src}, p)$. A *wildcard*, *any* (or $*$) is used to allow a receive operation to match a message which may be delivered from an unknown source process or channel. Hence both p_{src} and the channel can be set to the wildcard *any*. For example, with a $recv(a_1, size_1, p_{dest}, *)$ operation, process p accepts messages issued by process rank 2 incoming from the subset of channels of $\mathcal{CH}(p_{dest}, p)$. Similarly, the operation $recv(a_2, size_2, *, c)$ accepts messages from any process dispatched by the set of channels $\{ch(x, p) \,|\, x \in \mathcal{P}\} \subseteq \mathcal{CH}$. The semantics of the *recv* primitive can be formally represented as follows:

$$\forall i \in [0, size) \Rightarrow addr_p[i] := addr_{p_{src}}[i]$$

send and *recv* are *blocking* primitives. This means that the control is given back to the caller when the data transfer between the two processes is completed. Since the sender cannot start the transfer before the receiver is ready to store the data, a *network protocol* is necessary. The *rendezvous protocol* is often used in message passing libraries, it is schematically depicted in Figure 2.2. The protocol dictates that the sender process sends a message to the receiver requesting the permission to send data (i.e., `RTS` message). When the receiver is ready (i.e., the *recv* primitive has been issued), it sends back an acknowledgment, the `CTS` message, and prepares itself to receive the message data which is transfered immediately after.

One of the major drawbacks of the *rendezvous* protocol is its costs in terms of latency, since several messages are exchanged during the initial negotiation process, or *handshaking*. As explained in Appendix A message passing libraries for HPC use several strategies and *buffering* to reduce communication latency.

Similar to the *send* routine, a *recv* blocks if p_{src} has not issued a *send* operation. Because of the blocking semantics, the improper use of the *send* and *recv* routines may lead to a situation in which two processes are both waiting for the other to issue a receive operation before proceeding. The result is that both processes wait indefinitely. This condition is called *deadlock*. It is the responsibility of the programmer to prevent such situation from happening.

send and *recv* also have a *non-blocking* (or *asynchronous*) version called *isend* and *irecv*, respectively.

Definition 18 – *isend*

Let us define the routine

$$\textbf{isend}(addr_p, size, p_{dest}, ch, h)$$

The difference with the *send* primitive, in Definition 16, is the additional h argument which represents a *handle*, returned by the function, used for checking the completion of the data transfer. The semantics is similar to the one described in Definition 16, however the *isend* immediately completes without waiting for a matching receive operation.

Definition 19 – *irecv*

Let us define the routine

$$\textbf{irecv}(addr_p, size, p_{src}, ch, h)$$

As described in Definitions 17 and 18 the difference is the presence of the h handle which is used to check whether the data has been received. Also in this case the semantics is non-blocking, which means that the *irecv* routines immediately completes.

The advantage of using non-blocking routines is that a program can do some useful computation while the data transfer is being performed in the background. This optimization technique is also known in literature as the *communication/computation overlap*. It is however important that the area of memory $[addr_p, addr_p + size) \subseteq \mathcal{A}_p$ is not accessed during the transfer. In particular, in the case of a *isend*, the memory can be read but not written; during an asynchronous *irecv* the memory area being transfered should neither

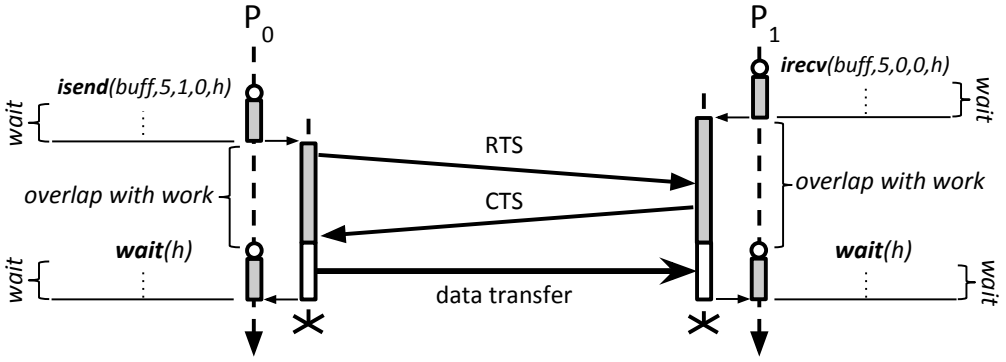


FIGURE 2.3: Sequence diagram for *non-blocking* operations.

be read nor written. This is because the data transfer is offloaded to an auxiliary process (or a specialized hardware such as a Direct Memory Access (DMA) processor) which executes in parallel to process which has issued the communication. If the input buffer is changed before the auxiliary process performs the data transfer, then the receiver obtains the updated value and not the one at the moment the operation was issued. Access to the buffer data is allowed only after checking for its completion, this is done through the *wait* routine.

Definition 20 – *wait*

Let us define the routine

$$\mathbf{wait}(h)$$

This operation, given an handle from a non-blocking routine, checks whether the transfer is completed. If not, the routine blocks and waits until the data has been effectively sent or received. Otherwise it simply returns.

In Figure 2.3 a sequence diagram which explains the semantics of non-blocking operations. When a non-blocking operation is issued the control is immediately returned to the caller and the burden of executing the handshake protocol is delegated to a second flow of execution which is either a newly spawned process, or a specialized hardware (e.g., DMA). This allows the worker to overlap computational work while in the background the message is being transferred. The *wait* primitive allows the programmer to check for completion of a communication routine.

The programming model introduced until here allows exchange of messages whose size is already known a priori. In order to overcome this limitation, a routine is utilized to probe a channel for incoming messages.

Definition 21 – *probe*

Let us define the routine

$$\mathbf{probe}(p_{src}, c)$$

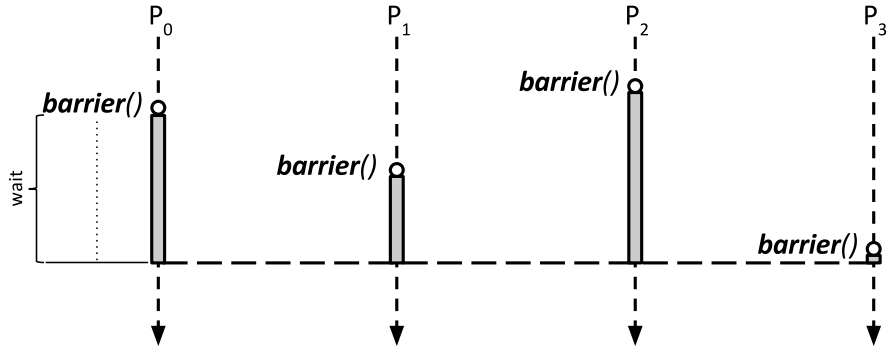


FIGURE 2.4: Sequence diagram for *barrier* primitive given $|\mathcal{P}| = 4$.

Given the channel $ch_i(p_{src}, p)$, this routine says whether there is a pending message. If found it returns the identifier of the sender process, the amount of data and the corresponding type. Similarly to the *recv* routine, wildcards can be used to match any sender and any available channel (see Definition 17).

It is worth noting that the *probe* does not receive the message, in order to retrieve the content of the pending message the *recv* or *irecv* routines must be utilized.

2.2.2 Collective Primitives

On top of point-to-point communication routines, we can define more complex abstractions which involve the entire group (or a subset) of the available processes. Such operations are called *collective* primitives. We define 4 primitive operations which must be invoked by all processes, collective routines are always blocking and all processes need to contribute in order for the routine to complete. We assume in our model that such operations always require all processes in a parallel program to take part in the computation.

Definition 22 – *barrier*

Let us define the primitive

barrier()

The first routine is a synchronization primitive which is utilized to make sure that all processes reached a program point. The routine is blocking and it returns the control only when all processes reached the barrier.

It is worth noting that a synchronization point is not given by the statement itself but instead multiple *barrier* operations, in different program points, must contribute to the same logic barrier instance. A sequence diagram of the *barrier* operations is depicted in Figure 2.4. It is interesting to note that the *barrier* is a *pure synchronization* routine.

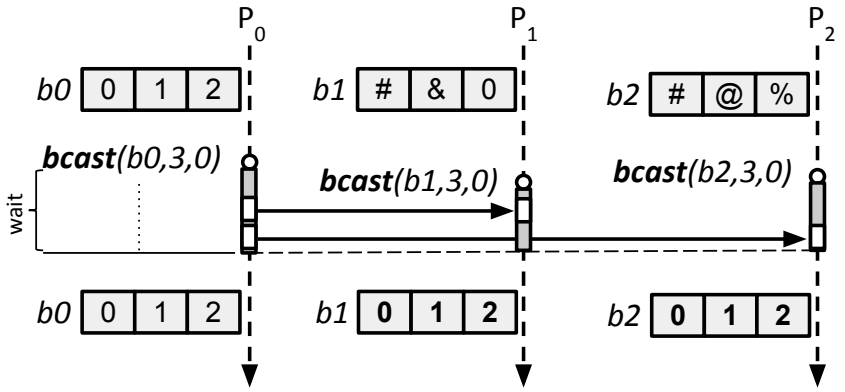


FIGURE 2.5: Sequence diagram for *bcast* primitive with p_0 being the root process; moreover $|\mathcal{P}| = 3$ and $size = 3$.

This means that no program data is exchanged and the time spent inside the routine is pure overhead. Therefore it is considered a good practice to avoid its usage.

Definition 23 – *bcast*

Let us define the primitive

$$\textbf{bcast}(addr, size, p_{root})$$

This operation copies $size$ bytes, starting from local memory address $addr$ of process p_{root}, i.e., $[addr_{p_{root}}, addr_{p_{root}} + size) \subset \mathcal{A}_{p_{root}}$, to processes, $\mathcal{P} - p_{root}$, address memory spaces, i.e., $\forall\, p \in \mathcal{P} - p_{root} : [addr_p, addr_p + size) \subset \mathcal{A}_p$. It therefore results in the following operation across memory address spaces:

$$\forall i \in [0, size), \forall\, p \in \mathcal{P} - p_{root} \Rightarrow addr_p[i] := addr_{p_{root}}[i]$$

While for the root process the operation is a read operation of the provided memory segment, it is a write for the remaining processes. In Figure 2.5 a diagram depicts the semantics of the *bcast* operation. The content of the buffer of the root process p_0, i.e., b_0, is copied to buffer locations provided by the remaining processes p_1 and p_2.

Definition 24 – *scatter*

Let us define the primitive

$$\textbf{scatter}(addr, size, p_{root})$$

Similar to a *bcast*, this routine allows a root process, i.e., p_{root}, to distributed local data to processes $\mathcal{P} - p_{root}$. In particular, the *scatter* takes the $[addr_{p_{root}}, addr_{p_{root}} + size)$ bytes and distribute the content to remaining processes as described by the

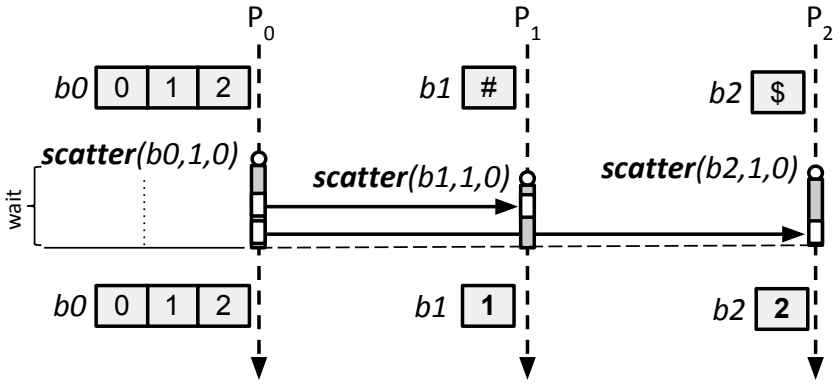


FIGURE 2.6: Sequence diagram for *scatter* primitive with p_0 being the root process; moreover $|\mathcal{P}| = 3$ and $size = 1$.

following formula:

$$\forall\, p \in \mathcal{P} - p_{root} \Rightarrow [addr_p, addr_p + size/np) := [addr_{p_{root}} + size/np * p, addr_{p_{root}} + size/np * (p+1))$$

The sequence diagram in Figure 2.6 depicts an example of a *scatter* operation performed by 3 processes, p_0, p_1 and p_2. All processes in $\mathcal{P}$ invokes the collective routine providing a buffer. The *root* process, i.e., p_0, provides a buffer, b_0, with $(|\mathcal{P}|) * size$ elements while the other processes buffers with $size$ elements. As a consequence of the *scatter* operation of Figure 2.6, having $size = 1$, the content of b_0 at position i is copied to the first element of buffer b_i belonging to process p_i.

Definition 25 – *gather*

Let us define the primitive

$$\textbf{gather}(addr, size, p_{root})$$

This is the inverse of the *scatter* operation. It gathers $size/np$ chunks of data coming from the memory address spaces of np distinct processes into the memory space of the root process, i.e., p_{root}. Formally we can describe the operation as follows:

$$\forall\, p \in \mathcal{P} - p_{root} : [addr_{p_{root}} + size/np * p, addr_{p_{root}} + size/np * (p+1)) := [addr_p, addr_p + size/np)$$

Figure 2.7 shows the sequence diagram of the *gather* operation. The way and the order data is gathered towards the root node p_0 is not necessary the one depicted in Figure 2.7. Several strategies are often employed by concrete implementations which takes care of various aspects, e.g., the network topology.

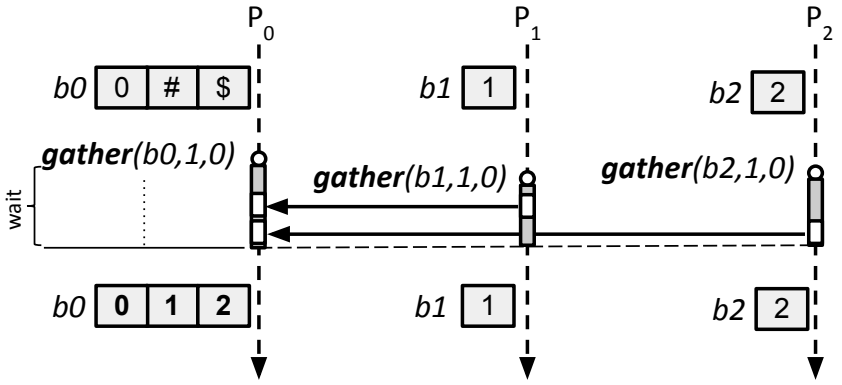


FIGURE 2.7: Sequence diagram for *gather* primitive with p_0 being the root process; moreover $|\mathcal{P}| = 3$ and $size = 1$.

In our model *scatter* and *gather* operations assume that all processes contribute with the same amount of data and the *root* process with none. However, in concrete implementation of the message passing paradigm, e.g., MPI, the semantics is slightly different since processes can contribute with different chunk sizes to the collective call and moreover, the *root* process must also contribute. Differences and analogies between our formal model and the concrete API provided my the MPI library are discussed in Appendix A.

Last primitive of our formal model combines data transfer and computation.

Definition 26 – *reduce*

Let us define the primitive

$$\textbf{reduce}(addr, size, func, p_{root})$$

This routine combines data from all the processes (except from p_{root}) using *func* and stores the combined value in the buffer provided by p_{root}. Each process can provide one element, or a sequence of elements, in which case the *func* operation is executed element-wise on each entry of the sequence. The formal semantics is the following:

$$\forall i \in [0, size), \forall\, p \in \mathcal{P} : addr_{p_{root}}[i] := func(addr_p[i], \ldots)$$

We assume the arity of *func* to be $np-1$ however, to simplify its concrete definition, a binary function is allowed so that the final value of the reduction can be computed by composing the results of the reductions at intermediate steps. In order for the result to be correct the provided *func* must be *associative*. Additionally, since the implementation can decide the order on which preliminary results are reduced together, *func* must also be *commutative*.

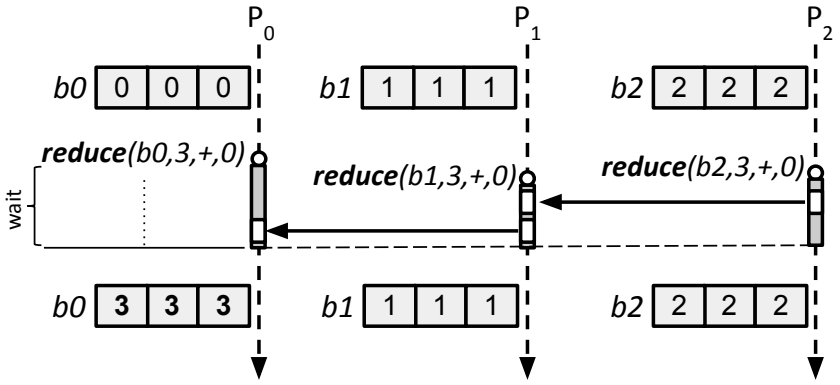


FIGURE 2.8: Sequence diagram for *reduce* primitive with p_0 being the root process; moreover $|\mathcal{P}| = 3$ and $size = 3$.

A sequence diagram of the *reduce* operation between $|\mathcal{P}| = 3$ processes is shown in Figure 2.8. The root process is p_0 and the reduction operator, i.e., *func*, is the binary plus operator. In this specific case the reduction is done by summing up, element-wise, the elements of array b_1 and b_2. The result is then sent to the root process and combined with the local array b_0 which is also utilized to store the final result. As previously stated, an implementation of this model may use a different strategy, therefore the reduction operator needs to be associative and commutative.

2.3 The Program Model

In Section 2.2, we described a program as a sequence of basic machine instructions (such as *load*, *store*, *move*) which are executed by a process. Programs are encoded in a way that are directly executable by a designated *cu*, i.e., *machine code*. Depending on the *cu* architecture a different version of the machine code must be provided.

In order to allow portability, programs are usually written using a higher-level, human-readable representation, a *source code*, that a separate program, known as the *compiler*, converts into machine code. During the conversion, the compiler analyses the source code and transforms it in order to improve its performance.

Computer programs can be categorized by the programming language paradigm used to produce them [22]. Two of the main paradigms are imperative and declarative:

Imperative: Programs written using an imperative language specify an algorithm using *declarations*, *expressions* and *statements*. These entities are used to define the data- and control-flow of a program.

Declarative: Programs written using a declarative language specify the properties that have to be met by the output. They do not specify details expressed in terms of the control flow of the executing machine but of the mathematical relations between the declared objects and their properties.

In this thesis we pose our focus on imperative paradigms only. However, in Section 3.3, we propose an extension of the message passing programming model which uses a declarative paradigm to hide low level generation of communication routines.

Definition 27 – *Variable*

Let us define a *variable*, var, as a storage location ($\in \mathcal{A}$) and an associated *symbolic name* (an *identifier*) which contains some quantity or information, a *value*. The variable name can be used to reference the stored value, the binding between the variable name and its location within the address space is done by the compiler when machine code is generated. The identifier in source code can be bound to a value during run-time or at compile-time. In both cases this value may change during the program execution.

A variable storage location can be referenced by multiple identifier simultaneously. Such situation is known as *aliasing*, i.e., $alias(a, b) \Rightarrow var_i\{id = a, addr = a_0\} \wedge var_j\{id = b, addr = a_0\}$. Assigning a new value through one of the identifier will change the value accessed through the other identifiers.

Definition 28 – *Function*

Let a function $f(v_0, \ldots, v_N)\{\, body \,\}$ be a sequence of program instructions that performs a specific task, packaged as a unit. The content of a function is called *body*. A function may expect to obtain one or more data values from the calling program which are intercepted by variables, called *formal parameters* $v_0, \ldots, v_N$. A function call may also return a computed value to its caller (called the *return value*). Throughout this thesis we use the terms: *function*, *procedure*, *routine* and *method* with the same semantics.

A special function, called the *main* function, represents the entry (or starting) point of every program. Therefore it must always be present.

Definition 29 – *Declaration*

A declaration d announces the existence of a *variable* or a *function* to the compiler.

A declaration also binds the variable identifier to a *type* which represents the storage size associated with that variable. Discussion of datatypes and more general the type system of a language is beyond the scope of this thesis.

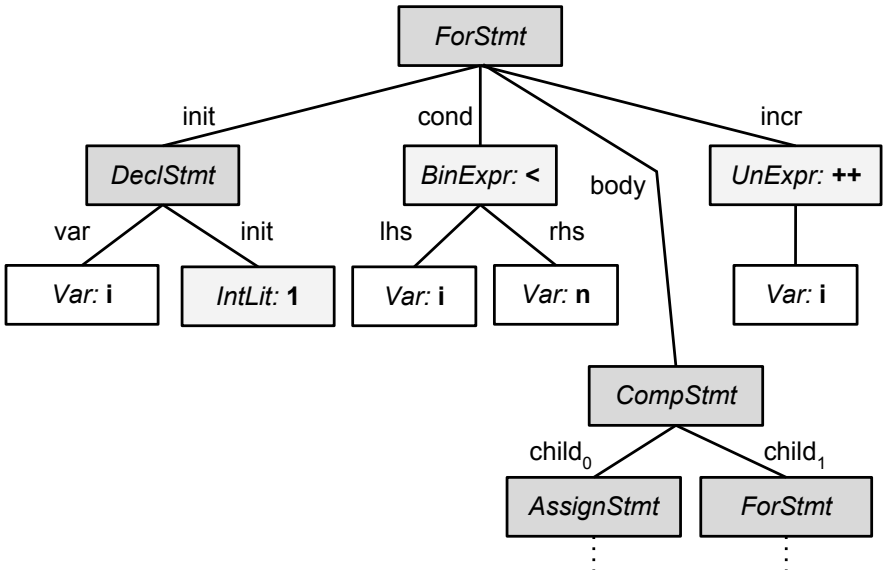


FIGURE 2.9: Partial AST of the program code in Listing 2.1

Definition 30 – *Statement*

A *statement* is the smallest standalone element of an imperative programming language. A program is formed by a sequence of one of more statements. Statements are used to describe both the data-flow (e.g., by means of assignment operation, i.e., `v := 0`) and the control-flow (e.g., by means of `IF`, `SWITCH`, `FOR`, `WHILE`, `BREAK` and `CONTINUE` statements) of a program.

We expect the reader of this thesis to be familiar with the semantics of control-flow statements commonly used in imperative programming languages.

2.3.1 The Program Abstract Syntax Tree (AST)

The *syntactic structure* of a program is often represented within the compiler on the basis of a syntax tree. Each node denotes a construct occurring in the source code. Since not every detail is represented, e.g., braces, semicolons and parentheses, this is an *abstract* representation. This data structure is widely used in source-to-source compilers due to their property of representing the structure of a program code in a way that syntactically equivalent source code can be reproduced. ASTs often serve as an Intermediate Representation (IR) of the program through several stages that a compiler requires. The structure of the IR can have strong impact on the final output of the compiler.

An example of the structure of an AST is depicted in Figure 2.9. This tree represents the program source code in Listing 2.1. The program root node is a `ForStmt` which is composed of 4 children nodes. The first child of the `Fortmt`, the initialization of the loop

contains the declaration of the variable `i` (the loop iterator) which is set to be 0 (the type information is omitted from the `ForStmt`). The second child of the `ForStmt` is the exit condition of the loop, this is a binary expression (i.e., `BinExpr`) comparing the values of variables `i` and `n`. Then follows the body of the statement which contains the statements which are executed in each loop iteration. In this case the body is a compound statement (i.e., { }, `CompStmt`) which contains two statements: an assignment (line 4) and a second for loop with iterator variable `j`.

The AST is an important representation on top of which many semantic program analyses can be realized. Additionally, a tree is a data structure which is easy to manipulate and which is commonly used as a base structure for program transformations. A concrete implementation of an AST is provided in Appendix C which gives an overview of the Insieme compiler project. However, while ASTs are a mean to represent the structure of a program, they lack the capability to represent program execution. For example, in the AST of Figure 2.9 the loop body is represented by a single tree node. When the program is executed, the loop body is likely to be executed more than once, therefore multiple dynamic instances of that AST node may exist. The AST gives no means to distinguish between dynamic instances associated with a single statement. This poses a limit on the accuracy of the analysis results and the transformation capabilities since transforming a statement in the AST affects all dynamic instances of it.

Over the last decade, alternative ways of representing programs have been proposed with the goal to overcome the limitations of ASTs.

2.3.2 The Polyhedral Model (PM)

The PM represents the execution of a program in an algebraic way. It captures both the control-flow and data-flow behaviour using three compact algebraic structures, described in the following subsections. The main idea is to define, for a statement S, a *space* in Z^n where each point corresponds to an execution, or *instance*, of S. The value of the coordinates of a point within this space represents the value of the n nested loop iterators spawning statement S. We call this space *polytope*.

Definition 31 – *Polytope*

The set of all vectors $\vec{x} \in \mathbb{Z}^n$ such that $A\vec{x} + \vec{b} \geq 0$, where A is an integer matrix, defines a convex integer polyhedron. A bounded polyhedron is also called polytope.

The polytope associated with a program statement is also called its *Iteration Domain*.

```
1 for (unsigned i=0; i<n; ++i) {
2    A[i][i] = 0;                                                    S0
3    for (unsigned j=i; j<n-1; ++j) {
4       A[i][j+1] = i+j;                                             S1
5       if ((i+j)%2) { A[n-j][n-i] = j-i; }                          S2
6 }
```

LISTING 2.1: A simple sequential code example

Definition 32 – *Iteration Domain*

Let use define an *Iteration Domain*, $\mathcal{D}_S$, as the polytope (in Z^n) in which a statement S is defined.

For example let us consider the code in Listing 2.1. This loop nest contains 3 assignments which are referred to as $S0$, $S1$, and $S2$. Control-flow statements (e.g., `IF`, `FOR`, `SWITCH`) are used to define the surrounding domain. For example the iteration domain for $S0$, $S1$, and $S2$ is defined as follows:

$$\mathcal{D}_{S0} := \{\, i \mid 0 \leq i < n \,\}$$
$$\mathcal{D}_{S1} := \{\, i, j \mid 0 \leq i < n \wedge i \leq j < n-1 \,\}$$
$$\mathcal{D}_{S2} := \{\, i, j \mid 0 \leq i < n \wedge i \leq j < n-1 \wedge \exists e \in \mathbb{Z} \mid i + j - 2e = 1 \,\}$$

As in Definition 31, iteration domains are represented by the integer matrix A multiplied by a so called *Iteration Vector* $\vec{x}$. The iteration vector determines the dimensionality of the iteration space for a statement and therefore it is composed of the loop iterators enclosing that statement and *parameters* (also known in literature as *free variables*), which are unknown integer values constant within the loop nest. For example iteration vector for statements $S1$ and $S2$ in Listing 2.1 is defined by the vector $\vec{x_{S1}} := (i, j, n)^\intercal$. Where i and j are the loop iterators of the enclosing loop nest and n is a parameter (existentially quantified variables have been projected out). Conventionally the matrix A is represented using a so-called *homogeneous* coordinates so that vector $\vec{b}$ is added as its last column. The iteration domain for statement S_1 is therefore represented as follows:

$$\mathcal{D}_{S_1} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2, \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & 1 & -2 \end{bmatrix} \cdot \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \geq \vec{0} \right\}$$

The second piece of information which is required to describe the semantics of a program is the so-called *scheduling* (or *scattering*) *function*. Intuitively, statements belonging to

a loop body, and subject to the same control flow, will share identical iteration domains. However, the information of the order on which statement instances are executed is not represented.

Definition 33 – *Schedule*

Let us define a *schedule*, $\theta(\vec{x})$, as a function which associates a logical *execution time* to each instance of a statement. This allows the ordering of the instances defined by the iteration domain and an ordering for instances belonging to different domains. A schedule $\theta(\vec{x})$ has the following shape: $\theta_S(\vec{x}) = T_S\vec{x} + \vec{t_S}$ where $\vec{x}$ is the iteration vector, T_S is an integer transformation matrix and $\vec{t_S}$ is a constant vector. T_S and $\vec{t_S}$ can be merged together into a matrix $\mathcal{S}$ if the system is represented on the basis of homogeneous coordinates.

It is worth noting that two or more statement instances that have the same execution time can be executed in *parallel*. For example, the scheduling functions for statements $S0$, $S1$, and $S2$ of the code in Listing 2.1 can be defined as follows:

$$\theta_{S0} = \{\, i, 0, 0 \,\} \qquad \theta_{S1} = \{\, i, 1, j, 0 \,\} \qquad \theta_{S2} = \{\, i, 1, j, 1 \,\}$$

T_S and $\vec{t_S}$ can be easily extracted by factorizing the iteration vector $(i, j, n, 1)^\intercal$. We call $\mathcal{T}$, the matrix representation of the schedule in the homogeneous form:

$$\mathcal{T}_{S_0} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \qquad \mathcal{T}_{S_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \qquad \mathcal{T}_{S_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}$$

Many strategies can be utilized to derive scheduling functions based on the structure of an input program. A widely used one is presented in [23], it uses the AST representation of a program. Multiplication of the scheduling function with the corresponding iteration domain produces a sequence of tuples, or logic dates, representing the execution order of each statement instance. By *lexicographically ordering* the set we can derive the exact sequence of statement instances executed by the program. The precedence relationship, $\prec$, is used to chronologically order statement instances regarding their execution times. The formal definition follows: $(a_1, \ldots, a_n) \prec (b_1, \ldots, b_m)$ *iff* $\exists\ i \in \mathbb{Z} \,|\, 1 \leq i \leq min(n, m) \wedge (a_1, \ldots, a_{i-1}) = (b_1, \ldots, b_{i-1}) \wedge a_i < b_i$. The precedence relationship applied to the statement instances of Listing 2.1 generated by the scheduling functions defined

above, produces the following sequence:

$$S0:(0,0,0) \prec S1:(0,1,0,0) \prec S2:(0,1,0,1) \prec \ldots \prec S2:(0,1,n-2,1) \prec$$
$$S0:(1,0,0) \prec S1:(1,1,0,0) \prec S2:(1,1,0,1) \prec \ldots \prec S2:(1,1,n-2,1) \prec \ldots$$

One last function is also required to capture the data locations on which a statement operates.

Definition 34 – *Access Function*

The *access* (or *subscript*) *function* describes the index expression utilized to access arrays, and therefore memory locations, within a statement. Representation of access functions is similar to what has been described for scheduling functions. Each dimension of a multi-dimensional access is represented by an affine expression which can be put into matrix form. Scalars, e.g., `s`, are supported by treating them as one-dimensional arrays with 1 single element (therefore accessed as `s[0]`),

For example array access in statement $S1$, `A[i][j+1]`, can be represented in a matrix form with one row for each dimension being accessed:

$$\mathcal{A}_{USE_A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}$$

Access functions also store the information whether a particular memory location is being read (i.e., `USE`) or written (i.e., `DEF`). This kind of information is utilized by the polyhedral model to compute exact dataflow and dependence analysis for a given code region, see Section 2.3.3.

2.3.2.1 Limits of the Polyhedral Model

Because of the affine linear nature of constraints utilized to describe control- and data-flow, not every program can be represented in the polyhedral model. The PM is limited to so called Static Control Parts (SCoPs) of a program which are defined to be maximal sets of consecutive instructions such that: for-loop bounds, conditionals and subscript expression are all affine functions of the surrounding loop iterators and global variables (or parameters); for-loop iterators and parameters cannot be modified [24] (apart from the implicit update of iterators within for-loops). Although this represent the main critic to the PM, SCoPs have been found to capture a large portion of the computation

time in scientific application [25]. Furthermore, in many cases, the input code can be rewritten to fit such constraints.

Compiler transformations implemented in many mainstream compilers (such as *constant propagation*, *function inlining* and *loop-invariant code motion*) can be applied to increase the applicability of the PM.

2.3.2.2 Counting integer points in a polytope

One feature of the theory behind the PM is the possibility to efficiently count, in a symbolic way, the number of *integer* points inside a polytope [26]. Since every statement in a program has an iteration domain associated (deriving from the structure of the control flow), this gives us the capability to know, statically, the number of dynamic instances for that statement. We refer to the number of integer points within an iteration domain as its *cardinality*. With $|\mathcal{D}_{S1}|$, we refer to the cardinality of statement `S1`. Within the rest of the paper we refer to an instance of a communication statements as a *communication operation*.

2.3.3 Data Dependencies

Assignment statements within a program define the data-flow. Two or more statements accessing the same variable are said to be *dependent*. If a dependency is broken, the outcome of a program, or its *semantics*, may differ. Dependence information are useful at several levels. Compilers often reorder execution of statements, or their schedule, to reduce the number of cycles a CPU spends waiting for certain type of costly instructions (e.g., fetching of data from the main memory). Additionally, *super-scalar* CPUs allow *out-of-order* execution of instructions, meaning that the processor executes instructions in an order governed by the availability of input data, rather than by their original order in a program.

Three types of dependencies may occur in a program:

Read-After-Write (RAW) : Also called *true-dependence*, it is identified by the symbol δ. A RAW dependence occurs when a statement $S1$ depends on the result of a previous statement $S0$, i.e., $S0\,\delta\,S1$.

Write-After-Read (WAR) : Also called *anti-dependence*, it is identified by the symbol δ^{-1}. A statement $S1$ is anti-dependent on $S0$ (written $S0\,\delta^{-1}\,S1$) if and only if $S1$ modifies a resource that $S0$ reads and $S0 \prec S1$.

Write-After-Write (WAW) : Also called *output-dependence*, it is identified by the symbol δ^o. It occurs when two statements $S0$ and $S1$ both modify the same resource and $S0 \prec S1$.

Formally we can state that a statement S_j *depends* on a statement S_i if there exists an operation $S_i(\vec{x_i})$, an operation $S_j(\vec{x_j})$, and a memory location m such that:

- $S_i(\vec{x_i})$ and $S_j(\vec{x_j})$ refer to the same memory location m, and at least one of them writes to that location;
- $\vec{x_i}$ and $\vec{x_j}$ respectively belong to the iteration domain of S and R;
- $S_i(\vec{x_i}) \prec S_j(\vec{x_j})$.

Depending on the nature of the memory operation in S and R (either a read or a write operation) one of the dependencies defined above can be identified. Due to the properties of the PM representation, it is possible to build a polytope which contains a point for each dependence between two array accesses. A formal description of how the so called *dependence polyhedron* is built, is presented in [23]. Intuitively a system of inequalities is built where an equality of all the access functions to the same array is imposed, i.e., $\mathcal{A}_{arr}(\vec{x_i}) = \mathcal{A}_{arr}(\vec{x_j})$ to enforce that the same memory location is read or written. Then using the scheduling functions we project those accesses in a new space where lexicographically ordering the accesses yields the set of all dependencies instances and their type (i.e., RAW, WAR or WAW). In this thesis we utilize the complex capability to perform data dependence analysis on SCoPs, however it is not the focus of this work to provide details on such method which is illustrated in [23, 27].

Dependencies are particularly important for array accesses since they are often used within for-loops (which account for most of the execution time in a program). For example a loop may write or read different memory locations during its execution. Depending on the expressions used to access the array, data dependencies between iterations, also called *loop-carried* dependencies, may or may not exist. Checking for dependencies between two statements requires to check for the existence of integer solutions to a set of linear constraints. This is known to be an NP-complete problem. Traditionally, approximate and/or incomplete methods with fast worst-case performance have been used for dependence analysis [28, 29].

2.3.4 Control Flow Graph (CFG)

Compilers usually complement the structural representation of a program given by the AST with an additional graph representing the control-flow of the program. This data

structure, called the CFG, can be directly be generated from the AST and it represents all paths that might be traversed through a program during its execution. The CFG is essential to many compiler optimizations and static analysis tools.

Definition 35 – *CFG*

Let us define a generic CFG $= (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges. An edge e is defined as a tuple $(v_i, v_j) \in \mathcal{E}$ with $v_i, v_j \in \mathcal{V}$ and e is directed from vertex v_i to v_j. In a CFG two special designated vertices *entry* and *exit* exist. The control of a program enters through the *entry* vertex and exits through the *exit* vertex.

In a CFG vertices and edges are different from the ones used in the construction of an AST. In the AST the vertices of the tree represent concepts related to the structure of a programs such as: statements, declarations, expressions and variables. Edges put those components into a relationship *container/contained* object. In the CFG, the focus is on the control-flow therefore vertices and edges have different semantics. The CFG generated from the source code in Listing 2.1 is depicted in Figure 2.10.

Definition 36 – *Basic Block*

Vertices of a CFG are also called *basic blocks*. A basic block is a maximal group of consecutive statements that are always executed together with a strictly sequential control flow between them. Multiple flows can either enter or exit from a basic block.

In Figure 2.10, the CFG is composed of eight basic blocks (i.e., $B0, \ldots, B7$). It can be noted that some of the basic blocks may generate multiple exit flows. These blocks (i.e., $B1, B3, B4$) correspond to the control flow statements, i.e., `FOR` and `IF`, used within the code in Listing 2.1. These nodes have a special statement (the last) called *terminator*, identified by a `T`, which contains the expression used in the program to decide which of the exit branches should be taken. Edges of a CFG represent the flow of the control, thus they are *directed*.

Definition 37 – *Path*

Let us define a $path(v_i, v_j)$ in a CFG as a sequence of vertices (or basic blocks) which connect vertex v_i to v_j, i.e., $path(v_i, v_j) := (v_i, v_0, v_1, \ldots, v_k, v_j)$. A *path* is characterized by its *length* which is represented by the number of vertices being traversed. Let us also define $path(v_i)$ as a shortcut for $path(entry, v_i)$. $path(v_i)$ identifies a path in the CFG whose root is always the *entry* vertex.

For each vertex of a CFG, two additional sets can be defined representing the predecessors and successors:

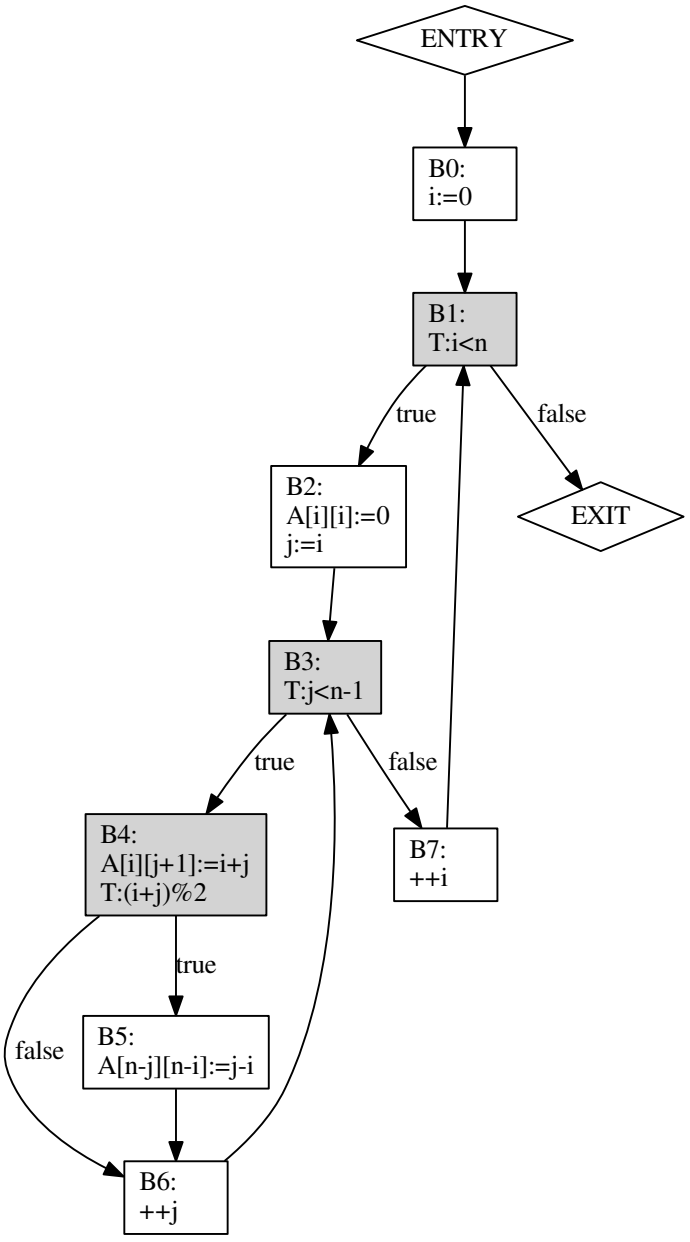


FIGURE 2.10: Complete CFG of the program code in Listing 2.1

$Pred(v_i)$**:** This is the set of all vertices preceding v_i, the set is formally defined as follows: $Pred(v_i) := \{\forall v_j \in \mathcal{V} \,|\, (v_j, v_i) \in \mathcal{E}\ \}$.

$Succ(v_i)$**:** This is the set of all vertices successive to v_i. The set is formally defined as follows: $Succ(v_i u) := \{\forall v_j \in \mathcal{V} \,|\, (v_i, v_j) \in \mathcal{E}\ \}$.

The CFG represent a powerful instrument for program analysis. Unlike the AST which is composed of many different node types, analysis built on top of the CFG only deals with

basic blocks. An example is represented by the iterative dataflow analysis framework which is formally described in Appendix D.

Chapter 3

Simplification of Distributed Memory Programming

The message passing programming model has a low level of abstraction, for this reason it is also referred to as the assembly language *of distributed memory programming. The model, as presented in Section 2.2 puts the burden on the developer to achieve correctness (e.g., lack of deadlocks) and performance.*

In this chapter we consider the programmability issues of the message passing paradigm from three different angles. At first, (i) we analyze the API offered by a popular message passing interface like MPI [3]. We show how the low level of the provided routines is inadequate for modern object-oriented languages such as C++ and propose an interface which improves programmability while keeping performance mostly unchanged. Secondly, (ii) we propose an object oriented programming model for distributed memory systems which raises the abstraction level for message passing languages. We achieve this by relying on meta-programming capabilities of the C++ language which enables the production of distributed memory code as part of the compilation process. At last, (iii) we propose a combined approach which relies on an minimal API and a powerful distributed runtime system which hides the distributed nature of the underlying computing system to the programmer. The library interface adopts and extends concepts of the OpenCL programming model making this approach suitable for coarse-grained parallel programs.

3.1 Introduction

3.1.1 A Lightweight Programming Interface

MPI is the *defacto* standard for writing parallel programs for distributed memory systems. An overview of MPI's principal routines and their relation with the generic message passing model presented in Section 2.2 is presented in Appendix A. As its focus is on HPC, MPI offers an API for C, C++ and Fortran; the most widely used languages for HPC. Unfortunately, since the definition of the first standard in 1994 [30], MPI did not keep the pace with the evolution of the underlying languages, such as OOP in Fortran 2000 and templates in C++.

Nowadays, this problem is mostly perceived in C++ which, unlike Fortran and C, provides much higher-level abstractions which are not reflected in the design of the MPI interface [4]. MPI is poorly integrated into the C++ environment thus many programmers prefer to use, even in C++ programs, the C interface. Furthermore, to map common C++ constructs onto MPI, programmers are forced to weaken the language type safety. As a consequence, errors that could be easily detected by the compiler are no longer captured leading to runtime failures. These issues led the MPI committee to the decision of deprecating C++ bindings in the version 2.2 of the MPI standard. However, because of the growing interest and use of C++ in HPC, several third-party wrappers to MPI have been proposed [31], the most important being Boost.MPI [5] and OOMPI [6].

In Section 3.2, we combine some of the concepts presented in Boost.MPI and OOMPI and propose an advanced lightweight interface called MPP that aims at transparently integrating the message passing paradigm into the C++ programming language without sacrificing performance. Our approach focuses on point-to-point communications and integration of user data types which, unlike Boost.MPI, relies entirely on native `MPI_Datatypes` for better performance. Our interface also utilizes advanced concepts from other parallel programming languages, e.g., *future objects* [32], which simplifies the use of MPI asynchronous routines.

3.1.2 Towards a Simplified Message Passing Programming Model for C++

Compared to other existing parallel programming models such as Open Multi-Processing (OpenMP), message passing offers two basic primitives: *send* (Definition 16) and *recv* (Definition 17). The burden of managing almost every aspect of the program execution

including data partitioning, communication, and synchronization between processes is left to the programmer. A low-level of abstraction is helpful in writing highly optimised programs, however, it makes distributed memory programming very complex, time-consuming and error-prone.

Recently, new programming models are increasingly being used aiming at simplifying distributed programming. An example is the PGAS model, which provides the programmer with a logically global memory address space where variables may be directly read and written by any process. Below the logical view, each variable is physically associated with a single process. Any attempt to read or write memory locations physically allocated on a different process results in a communication operation generated either by a runtime environment (e.g., in the Global Array library [33]) or during the compilation process in the Co-array Fortran and Unified Parallel C (UPC) [34, 35].

In Section 3.3, we show how a similar approach can be realized only relying on features of the C++ programming language. We employ techniques from OOP to provide an abstraction to homogeneously access the memory address space of a distributed application, $\mathcal{A}$. The proposed abstraction allows a process p to access the entire address space $\mathcal{A}$ in a transparent way. The C++ compiler then takes care of rewriting the access to memory address a in terms of local read/write operations, if $a \in \mathcal{A}_p$; or using an explicit data exchange over the network, if $a \notin \mathcal{A}_p$. We use meta-programming techniques, implemented with C++ templates, for the generation of such code for two reasons. Firstly, to spot parallelization errors at compile time. Secondly, to make the generated code efficient by rewriting the SPMD input code to an MPMD program which is specialized for each process identifier.

3.1.3 A Uniform Approach for Heterogeneous Distributed Memory Programming

The last contribution of this chapter is towards simplification of distributed memory programming focuses on the realization of a *runtime system*. With *libWater*, Grasso [36] proposes an alternative interface for programming distributed heterogeneous systems based on the OpenCL programming model. The novel interface allowed a powerful runtime system to be built achieving two design goals: (*i*) transparent abstraction of the underlying distributed architecture, such that compute units belonging to a remote node are accessible like local devices; (*ii*) enables the access to performance-related details since it supports the OpenCL kernel logic.

In Section 3.4 the *libWater* runtime system is discussed. An overview of Grasso's interface concepts will be briefly discussed in Section 3.4.3 to justify some of the design

```
1  if (rank==0) {
2     MPI_Send((const int[1]) { 2 }, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
3     std::array<float,2> val{3.14f, 2.95f};
4     MPI_Send(&val.front(), val.size(), MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
5  } else if (rank==1) {
6     int n;
7     MPI_Recv(&n, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
8     std::vector<float> values(n);
9     MPI_Recv(&values.front(), n, MPI_FLOAT, 0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
10 }
```

Listing 3.1: Simple MPI program using C bindings

principles adopted within the runtime system. The rest of Section 3.4 focuses on the runtime system being one of the contributions of this thesis. Calls to the library routines are internally represented as command descriptors which are being enqueued to a command queue. Dependencies between commands are expressed by the programmer through event objects. This enables the runtime system to structure such commands into a DAG which is used for *scheduling* and *optimization* purposes. An example is the realization of a novel hierarchical scheduling strategy which avoids the bottleneck caused by a centralized scheduling by delegating synchronization of commands targeting a remote node to that node's local scheduler. Additionally analysis of the DAG allows the recognition of communication patterns such as *broadcast*, *gather* and *scatter* and their replacement with a more efficient implementation. We demonstrate the scalability of our approach on homogeneous and heterogeneous clusters up to 32 nodes.

3.2 A Lightweight Interface for MPI

3.2.1 Background and Motivation

Listing 3.1 shows a simple MPI program sending two floating point values from process rank 0 to rank 1. A problem of this code snippet is that the programmer is forced to unnecessarily declare a temporary variable `val` to store the values being sent by `MPI_Send` (line 4). Although the C99 standard [37] introduced compound literals to avoid such unnecessary memory allocations (line 2), they are not widely used because of the decreased code readability. Because the compiler is not aware of the semantics of `MPI_Send`, which guarantees that the `val`'s value is not modified, no memory optimizations can be performed. A second problem is that the signatures of all MPI routines require the programmer to provide the size and the type (i.e., one `MPI_FLOAT`) of the data being sent, which is error-prone and can be avoided in C++ by inferring them at compile-time.

```
1 if (world.rank()==0) {
2    world.send(1, 1, 2);
3    world.send(1, 0, std::array<float,2>{3.14f, 2.95f});
4 } else if (world.rank()==1) {
5    int n;
6    world.recv(0, 1, n);
7    std::vector<float> values(n);
8    world.recv(0, 0, values);
9 }
```

Listing 3.2: Boost.MPI version of the program from Listing 3.1

Boost.MPI [5] tries to simplify the MPI interface by deducing several of those parameters at compile-time through C++ template techniques. For example, the size of the data sent and its associated `MPI_Datatype` is strictly related to the type of the object being sent and, therefore, deducible at compile-time from the C++ typing system. The `send` and `recv` routines in Boost.MPI require only three parameters, as shown in Listing 3.2 (lines 2, 3, 6, and 8): the source/destination rank, the message tag, and the message content. This not only simplifies the usage of the routines, but also improves their type safety. Although Boost.MPI is a consistent improvement over the standard MPI C++ bindings, it is not widely accepted within the MPI community because of two reasons: (*i*) the dependency on the Boost C++ library and accompanying licensing issues; (*ii*) the use of a serialization library [38] to handle transmission of user-defined data types (i.e., merging of objects with a sparse memory representation into a continuous data chunk) that negatively impacts the performance.

An object-oriented approach to improve the C++ MPI interface is OOMPI [6] which specifies send and receive operations in a more user friendly way by overloading the insertion `<<` and extraction `>>` C++ operators. In OOMPI, a `Port` towards a process rank is obtained by using the array subscript operator, i.e., `[ ]`, (on a communicator object (see line 2 in Listing 3.3). A further advantage is the convenience to combine these operators in one C++ instruction when inserting or extracting data to/from the same stream. A drawback of OOMPI is the poor integration of arrays and user data types in general. For example, sending an array instance requires the programmer to explicitly instantiate an object of class `OOMPI_Array_message`, which requires the size and type of the data to be manually specified as in the current MPI specification (line 4). The support for generic user data types requires the objects being sent to inherit from the `OOMPI_User_type` interface. This is a rather severe limitation as it does not allow any legacy class (e.g., the Standard Template Library (STL)'s containers) to be directly supported.

In this Section we propose an alternative interface for MPI which overcomes many of the previous mentioned problems. We introduce MPP [39], an header-only interface,

```
1  if (OOMPI_COMM_WORLD.rank()==0) {
2     OOMPI_COMM_WORLD[1] << 2;
3     std::array<float,2> val{3.14f, 2.95f};
4     OOMPI_COMM_WORLD[1] << OOMPI_Array_message(&val.front(), val.size());
5  } else if (OOMPI_COMM_WORLD.rank()==1) {
6     int n;
7     OOMPI_COMM_WORLD[0] >> n;
8     std::vector<float> values(n);
9     OOMPI_COMM_WORLD[0] >> OOMPI_Array_message(values, 2);
10 }
```

Listing 3.3: OOMPI version of the program from Listing 3.1

therefore lightweight and easy to use, which aims at a better integration with the C++ language. Overall, MPP is designed with a specific focus on performance. As we target HPC systems, we understand how critical performance is and several efforts have been spent in reducing the interface overhead. We compare the performance of MPP with Boost.MPI and show that, for a simple ping-pong application, MPP achieves a throughput (in terms of messages per seconds) which is 4 times larger than Boost.MPI. Compared to the pure C bindings, MPP has an increased latency of only 9%. As far as the handling of user data types is concerned, MPP is shown to reduce transfer time of a linked list (i.e., `list<T>` from C++ STL) up to 20 times compared to Boost.MPI. In order to determine the benefit of using MPP for real applications, we rewrote the computational kernel of QUAD_MPI [40] to use Boost.MPI and MPP. Results show a performance improvement of around 12% with respect to Boost.MPI.

3.2.2 MPP: C++ Interface to MPI

We use object-oriented programming concepts and C++ templates to design a lightweight wrapper for MPI routines that simplifies the way in which MPI programs are written. Similar to Boost.MPI, we achieve this goal by reducing the amount of information required by MPI routines and by inferring as much as possible at compile-time. By reducing the amount of code written by the users, we expect less programming errors. Furthermore, by making type checking safer, common programming mistakes can be captured at compile-time. In this work, we focus on point-to-point operators, as the specialised semantics of collective operations has no counterpart in C++'s STL. We also present a generic mechanism of handling C++ user data types which allows for easy transfer of C++ objects to any existing MPI routine (including collective operations).

```
1  namespace mpi {
2     struct comm {
3        mpi::endpoint operator()(int) const;
4     };
5  } // end mpi namespace
6
7  template <class InStream, class T>
8  void read_from(InStream& in, T& val) { in >> val; }
9
10 int val[2];
11 // reads the first element of the val array from std::cin
12 read_from(std::cin, val[0]);
13
14 // receives 2nd element of val array from rank 1
15 read_from(mpi::comm::world(1), val[1]);
```

LISTING 3.4: Example of usage of endpoints in a generic function.

3.2.2.1 Point-to-Point Communication

While Boost.MPI maintains in its API design the style of the traditional send/receive MPI routines, our approach is more similar to OOMPI aiming at a better C++ integration by defining these basic operations using *streams*. A stream is an abstraction that represents a device on which input and output operations are performed. Therefore, sending or receiving a message through an MPI channel can be seen as a stream operation. We introduce an `mpi::endpoint` class which has the semantics of a bidirectional stream from which data can be read (received) or written (sent) using the `<<` and `>>` operators respectively. The concept of endpoints is similar to the `Port` abstraction of OOMPI, however, because our mechanism is based on generic programming, user-defined data types can be transparently handled. In contrast, OOMPI is based on inheritance which forces the programmer to instantiate an `OOMPI_Message` class containing the data type and size required by the MPI routines underneath [31] (see line 4 in Listing 3.3).

Because an MPI send/receive operation offers more capabilities than C++ streams (e.g., tags for messages, non-blocking semantics), endpoints cannot be directly modelled using an *"is-a"* relationship. Fortunately, STL's utilities (e.g., algorithms) are mostly based on templates and endpoints can be passed to any generic function which relies on the `<<` or `>>` stream operations. Listing 3.4 shows an example that uses an endpoint as argument to a generic `read_from` function. An endpoint is generated from a communicator using the procedure call operator, i.e.,`()`, to which the process rank is passed (line 3). The `mpi::comm` class is a simple wrapper for an `MPI_Communicator` with the capability of creating endpoints, retrieving the current process rank and the communicator size. The `mpi::world` refers to an instance of the `comm` class which wraps the `MPI_COMM_WORLD` communicator.

```
1  using namespace mpi;
2  if (comm::world.rank()==0) {
3     comm::world(1) << std::array<float,2>{3.14f, 2.95f};
4     comm::world(1) << msg(2, 1);
5  } else if (mpi::world.rank()==1) {
6     int n;
7     comm::world(0) << msg(n, 1);
8     std::vector<float> values(n);
9     comm::world(0) >> values;
10 }
```

Listing 3.5: MPP version of the program from Listing 3.1.

```
1  float real;
2  mpi::request<float> req = mpi::comm::world(mpi::any) > real;
3  // ... do something else ...
4  use( req.get() );
```

Listing 3.6: Non-blocking MPP endpoints.

Listing 3.5 shows how the program in Listing 3.1 can be rewritten with MPP. First of all, objects are either sent or received using stream operations which allows for a more compact code compared to C MPI bindings (half in size) or to Boost.MPI. Secondly, objects are automatically wrapped by a generic `mpi::msg<T>` object, which does not need to be specified by the user (as opposed to OOMPI). Adding this level of indirection allows MPP to handle both primitive and user data types in a way transparent to the user. R-values (i.e., values with no address such as constants) are handled similar to any regular L-value (e.g., variables) using C++ constant references via the `msg` class, which avoids unnecessary memory allocation. The interface also allows specifying message tags by manually allocating the message wrapper (example in line 3).

MPP also supports non-blocking semantics for the send and receive operations through the overloaded `<` and `>` operators. Unlike blocking send/receives, asynchronous operations return a future object [32] of class `mpi::request<T>` which can be polled to test whether the pending operation has completed or not. An example of non-blocking operations in MPP is shown in Listing 3.6. For non-blocking receives, the method `T& get()` waits for the underlying operation to complete (line 2) and, upon completion, it returns a reference to the received value. The `mpi::request<T>` class also provides a `void wait()` and a `bool test()` method implementing the semantics of `MPI_Wait`, respectively `MPI_Test`. The example also shows MPP's support for receive operations which listens for messages coming from an unknown process using the `mpi::any` constant rank when creating an endpoint (line 3).

Errors, which in MPI are returned by every routine as an error code, are handled in MPP via C++ exceptions. Any call to MPP routines can potentially throw an exception which

```
template <class T>
struct mpi_type_traits<std::vector<T>> {
   static inline const T* get_addr(const std::vector<T>& vec) {
      return mpi_type_trait<T>::get_addr(vec.front());
   }
   static inline const size_t get_size(const std::vector<T>& vec) {
      return vec.size();
   }
   static inline MPI_Datatype get_type(const std::vector<T>&) {
      return mpi_type_trait<T>::get_type(T());
   }
};
...
typedef mpi_type_traits<vector<int>> vect_traits;
vector<int> v = { 2, 3, 5, 7, 11, 13, 17, 19 };
MPI_Ssend(vect_traits::get_addr(v), vect_traits::get_size(v),
          vect_traits::get_type(v), ... );
```

LISTING 3.7: Example of using `mpi_type_traits` to handle STL vectors.

is a sub class of `mpi::exception`. The method `get_error_code()` of this class allows the retrieval of the native error code.

3.2.2.2 User Data Types

OOMPI is one of the first APIs trying to introduce support for user data types through inheritance from an `OOMPI_User_type` class. Unfortunately, this mechanism is relatively weak because, by relying on inheritance, it does not allow the handling of class instances provided by third-party libraries (e.g., STL containers). Another attempt is the use of serialization in Boost.MPI which, although elegant, introduces a high runtime overhead. The objective of MPP is to reach the same level of integration with user data types as Boost.MPI without performance loss, which we achieve by relying on the existing MPI support for user data types, i.e., `MPI_Datatype`. See Appendix A.2.4 for an overview of MPI datatypes and their implementation within the standard. The definition of an `MPI_Datatype` is rather cumbersome and therefore not commonly used. Defining an `MPI_Datatype` requires the programmer to specify several information related to its memory layout which often leads to programming errors that are very difficult to debug. However, because operations on data types are mapped to DMA transfers by the MPI library, the use of an `MPI_Datatype` outperforms any other techniques based on software serialization.

The integration of user data types is achieved by using a design pattern called *type traits* [41]. An example is illustrated in Figure 3.7 for C++ STL's `vector<T>` class. We let the user specialise a class which statically provides the compiler three pieces of information required to map a user data type to `MPI_Datatype`s: (i) the memory address

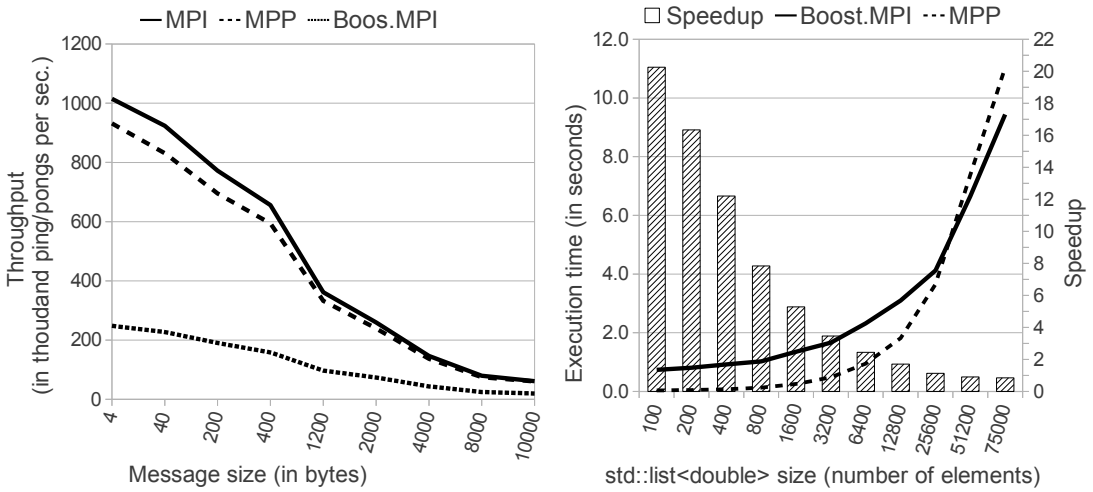
from which the data type instance begins; (*ii*) the type of each element; and (*iii*) the number of elements. Because a C++ vector is contiguously allocated in memory, the starting address of the first element has to be recursively computed for handling generic regular nested types (e.g., `vector<array<float,10>>` in lines $3-5$). The length is the number of elements present in the vector (line 9) and the type is the data type of a vector element (line $9-11$). Because our mechanism is not based on inheritance (like in OOMPI), it is open for integration and use with third party class libraries. Lines $14-17$ show how the introduced type traits can be used with the MPI C binding. This can also be used for collective operations or for one of the several flavors of `MPI_Send` for which an appropriate operator cannot be defined. MPP provides several type traits for some of the STL containers (i.e., vector, array and lists).

3.2.3 Performance Evaluation

In this section we compare the performance of MPP against Boost.MPI and the standard C binding of MPI. Open MPI version 1.4.2 runtime has been used to execute the experiments. We did not consider OOMPI for performance evaluation since its development has been stopped since several years. We compare the MPI bindings firstly by using micro-benchmarks, successively with a real MPI application called QUAD_MPI. QUAD_MPI is a C++ program which approximates an integral using a quadrature rule [40].

3.2.3.1 Micro Benchmarks

The purpose of the first experiment is to measure the latency overhead introduced by MPP over the standard C interface to MPI. Boost.MPI and MPP are both implemented on top of MPI C bindings. We implemented a simple ping-pong application which we executed on a shared memory machine with a single AMD Phenom II X2 555, 3.5 GHz dual-core processors, 1MB of L2 cache, and 6MB of L3 cache. This way, any data transmission overhead is minimized and the focus is solely on the interface overhead. Figure 3.1(a) displays the number of ping-pong operations per second for varying message sizes. MPP has approximately 9% larger latency for small messages compared to the native MPI routines. This overhead is due to the creation of a temporary status object corresponding to the `MPI_Status` returned by the MPI receive routine containing the message source, size, tag, and error (if any). Compared to Boost.MPI, MPP shows nevertheless a consistent performance improvement of around 75% for small message sizes. Because both implementations use plain vectors to store the exchanged message, no serialization is involved to explain the overhead difference. We believe that



(a) Number of ping/pong operations per second.

(b) Comparison of Boost.MPI and MPP for STL's linked list (`list<T>`).

FIGURE 3.1: MPP performance evaluation results.

the main reason for this overhead comes from the fact that Boost.MPI is implemented as a library and every call to MPI routines pays the overhead of an additional function call. We solved the problem in MPP by designing a pure header-based implementation, this allows all MPP routines to be inlined by the compiler and therefore eliminating any overhead. The graph also illustrates that, as expected, the overhead decreases for larger messages as the communication time becomes predominant.

In the second experiment, we compared MPP with Boost.MPI for the support of user-defined data types. We used a `list<double>` type of varying size exchanged between two processes in a loop repeated one thousand times. The code of the type trait defined to support any generic linked list is showed in Listing 3.8. We executed the experiment on an IBM blade cluster with a quad-core Intel Xeon X5570 processors interconnected through Infiniband network. We allocated the two MPI processes on different blades in order to simulate a real use case scenario. Figure 3.1(b) shows the time necessary to perform this micro-benchmark for different list sizes and the speedup achieved by MPP over Boost.MPI. For small lists of 100 elements, the speedup is approximately 20, however, the performance gap closes by increasing the list size. The reason is the `list` implementation in MPP using `MPI_Type_struct`, which requires enumerating all memory addresses that compose the object being sent. To create an `MPI_Datatype` for a linked list, three arrays have to be provided: (*i*) the displacement of each list element relative to the starting address; (*ii*) the size of each element; and (*iii*) the data type of each element (i.e., $O(3 \cdot N)$ of memory overhead). We observe in Figure 3.1(b) that building such a data type becomes more expensive as the list size increases, so that for large linked lists of over 50,000 elements, the software serialization outperforms the MPI

```
template <class T>
struct mpi_type_traits<std::list<T>> {

   static inline size_t get_size(const std::list<T>& vec) { return 1; }

   static MPI_Datatype get_type(const std::list<T>& l) {
      std::vector<MPI_Aint> address(l.size());
      std::vector<int> dimension(l.size());
      std::vector<MPI_Datatype> types(l.size());

      auto dim_it = dimension.begin();
      auto address_it = address.begin();
      auto type_it = types.begin();

      MPI_Aint base_address;
      MPI_Address(const_cast<T*>(&l.front()), &base_address);

      *(type_it++) = mpi_type_traits<T>::get_type(l.front());
      *(dim_it++) = static_cast<int>(mpi_type_traits<T>::get_size(l.front()));
      *(address_it++) = 0;

      typename std::list<T>::const_iterator begin = l.begin();
      ++begin;
      std::for_each(begin, l.cend(), [&](const T& curr) {
            assert( address_it != address.end() &&
                    type_it != types.end() &&
                    dim_it != dimension.end() );

            MPI_Address(const_cast<T*>(&curr), &*address_it);
            *(address_it++) -= base_address;
            *(type_it++) = mpi_type_traits<T>::get_type(curr);
            *(dim_it++) = static_cast<int>(mpi_type_traits<T>::get_size(curr));

         }
      );
      MPI_Datatype list_dt;
      MPI_Type_create_struct(static_cast<int>(l.size()), &dimension.front(),
                             &address.front(), &types.front(), &list_dt);
      MPI_Type_commit(&list_dt);
      return list_dt;
   }

   static inline const T* get_addr(const std::list<T>& list) {
      return  mpi_type_traits<T>::get_addr(list.front());
   }
};
```

Listing 3.8: Type trait for a generic `std::list<T>`.

data typing mechanism. Future optimization could improve the support of large data structures integrating in MPP a mechanism that switches from the use of `MPI_Datatype` to serialization starting from a critical size.

```
1  double my_a, my_b;
2  my_total = 0.0;
3  if (rank == 0)  {
4    for (unsigned q = 1; q < p; ++q) {
5      my_a = ((p - q) * a  + (q - 1) * b) / (p - 1);
6      MPI_Send(&my_a, 1, MPI_DOUBLE, q, 0);
7
8      my_b = ((p - q - 1) * a + q * b) / (p - 1);
9      MPI_Send(&my_b, 1, MPI_DOUBLE, q, 0);
10   }
11 } else {
12   MPI_Recv(&my_a, 1, MPI_DOUBLE, 0, 0, status);
13   MPI_Recv(&my_b, 1, MPI_DOUBLE, 0, 0, status);
14
15   for (unsigned i = 1; i <= my_n; ++i) {
16     x = ((my_n - i) * my_a + (i - 1) * my_b) / (my_n - 1);
17     my_total = my_total + f(x);
18   }
19   my_total = (my_b - my_a) * my_total / (double) my_n;
20 }
```

Listing 3.9: Computational kernel of QUAD_MPI.

3.2.3.2 QUAD_MPI Application Code

The micro-benchmarks highlighted the low latency of the MPP bindings. However this does not say much about the benefits of using MPP for real application codes. For this purpose we took a simple MPI application kernel called QUAD_MPI and rewritten using Boost.MPI and MPP. QUAD_MPI is a C program which approximates an integral using a quadrature rule [40]. The computation is done in parallel by using MPI. The original code can be found in [40], the computational kernel has been extracted and depicted in Figure 3.9. Process rank 0 assigns to each other process a sub-interval of $[A, B]$. The bounds are then communicated using message passing. The number of communication statement in the code is limited, i.e., $2(P-1)$, where P is the number of processes. Therefore this code represents a good balance between communication and computation making it ideal to determine the benefits of MPP bindings.

This code can be easily rewritten to use Boost.MPI and MPP. The manually rewritten code is shown respectively in Listings 3.10 and 3.11. In both cases we removed the necessity of assigning the value being sent to the *my_a* and *my_b* variables. This is because both Boost.MPI and MPP support sending R-values, the computed value is directly send to destination (lines 4 and 5). The code at the receiver side is similar, the only difference is that now we can restrict the scope of *my_a* and *my_b* variables to the else body only. This allows for a faster machine code as the compiler can utilize CPU registers in a more efficient way. Additionally, MPP allows for a further reduction of the code as shown in Listing 3.11. The two sends can be combined together into a single

```
1  my_total = 0.0;
2  if (rank == 0) {
3    for (unsigned q = 1; q < p ; ++q) {
4      world.send(q, 0, ((p - q) * a + (q - 1) * b) / (p - 1));
5      world.send(q, 0, ((p - q - 1) * a + q * b) / (p - 1));
6    }
7  } else {
8    double my_a, my_b;
9    world.recv(0, 1, my_a);
10   world.recv(0, 2, my_b);
11
12   for (unsigned i = 1; i <= my_n; ++i) {
13     x = ((my_n - i) * my_a + (i - 1) * my_b) / (my_n - 1);
14     my_total = my_total + f(x);
15   }
16   my_total = (my_b - my_a) * my_total / (double) my_n;
17 }
```

Listing 3.10: Computational kernel of QUAD_MPI rewritten using Boost.MPI.

```
1  my_total = 0.0;
2  if (rank == 0)  {
3    for (unsigned q = 1; q < p; ++q) {
4      comm::world(q) << ((p - q) * a + (q - 1) * b) / (p - 1)
5                     << ((p - q - 1) * a + q * b ) / (p - 1);
6    }
7  } else {
8    double my_a, my_b;
9    comm::world(0) >> my_a >> my_b;
10
11   for (unsigned i = 1; i <= my_n; ++i) {
12     x = ((my_n - i) * my_a + (i - 1) * my_b) / (my_n - 1);
13     my_total = my_total + f(x);
14   }
15   my_total = (my_b - my_a) * my_total / (double) my_n;
16 }
```

Listing 3.11: Computational kernel of QUAD_MPI rewritten using MPP.

statement (line 4), as well as the receives (line 9). MPP also relieves the programmer from the burden of specifying a message tag, the tag 0 is always utilized by default. With MPP we are able to shrink the input code by 30% (in terms of number of characters) and less code means less errors and overall more productivity.

We ran the three versions of the QUAD_MPI program on a machine with 16 cores (a dual socket Intel Xeon CPU). We utilized shared memory communication in order to minimize communications costs and therefore highlight the library overhead. The input programs have been compiled with optimizations enabled (i.e., -O3). In Figure 3.2 the average execution time and standard deviation obtained out of 10 runs of the three codes are depicted. Because of the optimization we were able to perform on the input code (i.e., removing the superfluous, assignment to the *my_a* and *my_b* variables), the MPP

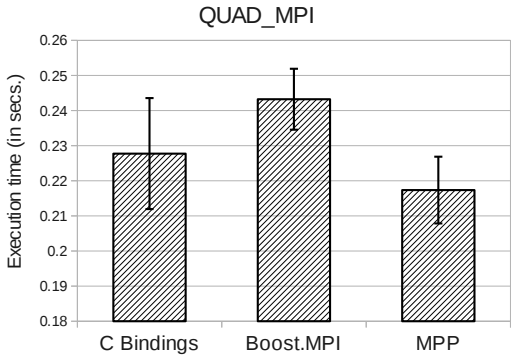


FIGURE 3.2: QUAD_MPI performance evaluation of the three code versions.

version performs slightly faster than the original code. It is worth noting that the same optimization has been applied to the Boost.MPI version. However, the large overhead of the Boost.MPI library cancels any of the benefits making the resulting code the slowest. Compared to Boost.MPI the application written with MPP bindings has a performance improvement of around 12%.

3.3 Towards a Simplified Message Passing Programming Model for C++

Simplifications to the programming interface makes message passing programs easier to read and write. However, some of the main issues which make programs difficult to analyze, and errors to detect, from a compiler point of view remain unchanged. In this section, we propose the use of a declarative programming model which enables a programmer to specify the type data movement he wants to achieve and offload to a compiler the burden of generating low-level intra- or inter-node data transfers.

3.3.1 Motivation

The motivation for the research presented in this section is based on the observation that sending a message from two processes is semantically equivalent to an assignment operation. Consider a matching *send*/*recv* operation between two processes p_i (the sender) and p_j (the receiver of the message). The content of the memory cells owned by process p_j is overwritten with data residing on process p_i's memory space as described in Section 2.2. In a programming language this is the semantics of the assignment operator.

Since C++ allows the redefinition of operators, we use the C++ *operator overloading* mechanism and *template meta-programming* techniques [42] to enable the automatic

generation of low-level communication primitives by the standard C++ compiler. For example, whenever an assignment operator involving memory cells residing on different processes' address spaces is encountered, the compiler generates the required communication statements. Additionally, we generate for each process rank a separate executable containing only those operations involving the assigned memory cells, which eliminates the control flow overhead incurred by the SPMD nature of the input program. The main advantage of our approach is the fact that it achieves a level of abstraction similar to PGAS-based languages by only exploiting features of the standard C++ language and compiler. Furthermore, because the underlying programming model is based on message passing, the programmer still retains full control over the resulting performance.

3.3.2 The PGAS Programming Model

One of the major efforts in improving and simplifying the message passing programming model was the definition of the PGAS programming model [35]. PGAS attempts to combine the advantages of an SPMD programming style for distributed memory systems with the data referencing semantics of shared memory systems. It assumes a *global* memory address space which is *logically* partitioned and a portion of it is local to each process. Each process has private memory for local data items and shared memory for globally shared data values. While the shared-memory is partitioned among the cooperating processes (each process will contribute memory to to the shared global memory), a process can directly access any data item within the global address space with a single address. Communication between processes is introduced, when needed, by the compiler. The PGAS model is the basis of UPC [35], Coarray Fortran [43], Chapel [44], X10 [45] and Global Arrays [33].

One of the main critic against PGAS-based languages is their low scalability when compared to highly optimized message passing programs. One advantage is however the low latency associated with the communication statements used by PGAS. This is due to the fact that PGAS-based languages often rely on a communication layer which is based on *one-sided communications* called GASNet [46]. The main difference with standard message passing libraries is that each process can update or interrogate the memory of another process without any intervention from the destination process. This is also known as Remote Direct Memory Access (RDMA). While this mechanism is usually faster (with lower latency) when a limited number of processes is used, it has been shown to have serious scalability issues. Moreover, the model offers no support for collective-like communications making those collaborative patterns highly inefficient.

```
1 float pi;
2 if (rank == 0) {
3    pi = calc_pi();
4    MPI_Send(&pi, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
5 } else if (rank == 1)
6    MPI_Recv(&pi, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
7 use(pi);
```

LISTING 3.12: Simple message passing program in MPI

3.3.3 Overview

This section gives a brief overview of our technique while further details will be given in Section 3.3.4 and 3.3.5. Let us consider in Listing 3.12 a simple message passing program written in MPI [3]. Two processes are involved in this example: process rank 0 computes the value of the π constant (`pi`) and sends it to process rank 1. The computed value is then used by both processes for further computation. One of the first characteristics of the program is the use of the SPMD technique, which generates a single executable that is spawned on multiple processors. To customize the program behaviour for a specific process rank, the programmer needs to continuously use control statements to guide the specific process flow of execution (lines 2 and 5). The use of control flow statements is in general the source of many inefficiencies and limits compiler analysis and optimizations. Additionally, miss-predicted branches cause significant performance penalties on modern pipelined CPU architectures [20]. Because the generated executable contains code which is never executed on a particular process rank, the L1 instruction cache may be not optimally used too.

A second observation is that message passing programs are often complex to read and, more importantly, to analyse. Because the programmer is forced by the programming model to describe the low-level operations (i.e., the *"how"*), the semantics of the program (i.e., the *"what"*) is mostly hidden. For example, although a connection between the send and receive operations in lines 4 and 6 exists, it is implicitly in the mind of the programmer and not made explicit in the code. We will focus on the matching problem of communication statements in Chapter 5 of this thesis. This hidden knowledge could be used by the compiler to improve error checking and program performance, but it is unfortunately very complex to be captured by static analysis [18, 47]. For example, the compiler could enforce the amount of received data to be not less than the amount of data sent, or use constant propagation to remove communication statements in case the transmitted value is constant (detected by compiler dataflow analysis).

We propose a different approach which lets the programmer focus on the program semantics (the *"what"*) and lets the compiler deal with the generation of the required

```
1 mem_wrap<float> pi;    // manages memory allocation in the distributed env.
2 pi[r0] = calc_pi();    // Rank 0 executes calc_pi() and writes the returned value
3                        // into its own copy of pi
4 pi[r1] = pi[r0];       // Copies the value of pi owned by process rank 0 onto the
5                        // memory cell owned by process rank 1 (by using send/recv)
6 use(*pi);
```

Listing 3.13: Overload of assignment operator in C++

Rank 0	Rank 1
1 float pi; 2 pi = calc_pi(); 3 MPI_Send(&pi,1,MPI_FLOAT,1,0,...); 4 use(pi);	1 float pi; 2 MPI_Recv(&pi,1,MPI_FLOAT,0,0,...); 3 use(pi);

Table 3.1: Compiler generated codes for process rank 0 and 1.

communication operations. The idea is not entirely new [35], however, instead of introducing a new programming model (e.g., PGAS) and an underlying language support (e.g., UPC), we exploit the capabilities of the standard C++ language and compiler. Listing 3.13 shows a simple C++ program semantically equivalent to the previous example. The first aspect is the lack of any control flow statements, which is achieved by offloading all memory operations to a new data type, i.e., `mem_wrap`, acting as a memory wrapper for distributed memory environments. The input program is compiled multiple times, each time for a different process rank. Keeping the value of the process rank constant at compile-time allows meta-programming techniques to be used for specializing the semantics of operations involving `mem_wrap` instances. For example, the initialisation of a memory cell owned by the process rank 0 results in a *no-operation* (`NOP`) when the program is compiled for process rank 1 (line 2). Assignment operations involving memory cells residing on different address spaces are replaced by communication statements (line 4). Table 3.1 shows the codes generated at compile-time by our approach for the processes with rank 0 and 1 from program code in Listing 3.13. The SPMD input program is compiled into multiple executables (as many as the number of processes) and successively executed using the MPMD paradigm.

MPI can be used to run both SPMD and MPMD programs through the `mpirun` command. In order to support the MPMD model, the MPI standard defines the ':' command line separator used to specify multiple executables [3]. Running the MPMD program generated by our technique produces very promising results. We executed both the SPMD and MPMD executables on an Intel Xeon X5570 CPU and an AMD Opteron 2435, both compiled with GCC 4.5.3 and optimization enabled (-O3). We repeated the

	SPMD		MPMD		
	Exec. time [milisec.]	*Standard deviation*	*Exec. time [milisec.]*	*Standard deviation*	*Speedup*
Intel Xeon	8180	440	6162	129	**1.32**
AMD Opteron	9638	166	9296	177	**1.04**

TABLE 3.2: Execution time for each process of the program in Listing 3.13 using SPMD and MPMD models.

Hardware counter	SPMD	MPMD
L1 Instruction Cache misses	4253718	**4246317**
L2 Instruction Cache misses	681689158	681689158
Conditional branch instructions	4260166	**4254384**

TABLE 3.3: Performance counter values for the Intel architecture.

code snippet one thousand million times and used shared memory communication, instead of the network, to reduce the communication overhead. The main program loop has been executed 10 times, the average value of execution time and standard deviation are depicted in Table 3.2. A considerable performance improvement, of around 30%, is observed for the Intel architecture, while on the AMD CPU, the improvement was of around 5%.

In order to explain the performance improvement we executed the code snippet enabling performance counters on the Intel CPU by using the PAPI library [48]. The measured values for three performance counters are depicted in Table 3.3. We measured the instruction cache misses for both level 1 and 2 and the total amount of conditional branch instructions. The code snippet is small to easily fit into L2 cache, therefore no differences in terms of L2 cache misses are visible. However, the utilization of the L1 cache is improved for the MPMD code as we were able to reduce the amount of cache misses by a 0.5%. This is because, by removing unreachable branches, code locality is improved. Additionally, also the amount of conditional branch instructions is reduced by the same amount. This alone cannot however explain the 32% speedup which we believe to be the result of optimizations (e.g., loop unrolling and constant propagation) performed by the compiler on the MPMD code. As a matter of fact, thanks to the simplification to the control flow obtained with our meta-programming technique, we enable the compiler analysis to perform more aggressive optimizations which are not forseable on the SPMD version.

3.3.4 The `mem_wrap` Object

Meta-programming is the practice of writing a computer program that manipulates other programs (or themselves) as their data. Meta-programming can be used to perform part

```
1  template <class T, template <class> class Sel, class R>
2  struct mem_wrap {
3    T& operator*();  // Access to managed memory
4
5    mem_wrap<T,Sel,R>& operator=(const T&);
6    template <template <class> class Sel2, class R2>
7    mem_wrap<T,Sel,R>& operator=(const mem_wrap<T,Sel2,R2>&);
8
9    template <class R2> mem_wrap<T,Sel,R2> operator[ ](const R2&);
10 };
```

Listing 3.14: `mem_wrap` object interface.

of the computation at compile-time instead of runtime. By combining templates and meta-programming, it is possible in C++ to specialize the implementation of generic functions based on particular properties of the input parameters. For example, a generic function can have two implementations depending on whether the input parameter is a pointer or a value type. Because these checks are conducted at compile-time, it is necessary that the expressions used to select a particular implementation involve compile-time constants only.

Our approach is based on a similar mechanism. The objective is to introduce an *enhanced* assignment operator which, depending on the type of the left and right hand side expressions, is specialized to implement different semantics. We introduce a new data type called `mem_wrap` illustrated in Listing 3.14 that manages the allocation and accesses to memory locations in the distributed memory environment. The first template parameter `T` represents the wrapped type which allows the management of single elements (e.g., `mem_wrap<float>`) or of more complex data types such as arrays (e.g., `mem_wrap<vector<float>>`). The second parameter `Sel` is the selector, which decides whether the wrapped object (of type `T`) has to be allocated on the particular process rank for which the input program is being compiled. For example, by using the expression `Rank%2==0` as a selector, we enforce only even process ranks to allocate the memory to host the object of type `T`. This means that when the program will be compiled for an even rank process, the generated code for the `mem_wrap` class will contain an initialization of an object of type `T`. We refer to these instances of `mem_wrap` as *active*. Odd ranks, for which the selector is not satisfied, allocate an empty wrapper instance called *shadow*. A shadow wrapper acts as a pointer to a memory location on a different machine and can be used to read data from it. To note that `mem_wrap` does not perform any data partitioning, the programmer is still responsible to divide the memory space among the processes. Because a `mem_wrap` instance can refer to memory locations on multiple address spaces, the `R` parameter is used to address the copy owned by that specific (i.e., `R`) process rank. The `mem_wrap` also provides three basic methods among several others: a dereferencing operator, i.e., `*`, used to directly access the memory managed by the

```
1 template <class RR = mpl::int_<MY_RANK>>
2 struct even {
3    template <class Rank>
4    struct apply : public mpl::bool_<Rank::value%2==0> { };
5 };
6 mem_wrap<std::vector<float>, even> vect(100);
7 for (unsigned int i=0; i<100; ++i) { vect(i) = MY_RANK; }
8 vect[r0] = vect[r2];
```

LISTING 3.15: Example of using selectors.

wrapper (line 3), an assignment operator, i.e., `=`, overloaded to work with data type instances of type `T` (line 5) or `mem_wrap` instances (line 7), and a subscript operator, i.e., `[]`, used to select a copy of the wrapped data which belongs to a particular address space.

There are two specializations of the `mem_wrap` class: one for active and the other for shadow wrapper instances. We define a pre-processor directive called `MY_RANK` as the rank of the process for which code is being generated. During the compilation process, for every instantiation of a `mem_wrap`, the selector is applied to the value of `MY_RANK`. Depending on the result, one of the two specializations is used. Furthermore, methods of the `mem_wrap` class have multiple specializations depending on the type of the input parameters.

To better understand how selectors work, we illustrate in Listing 3.15 a slightly more complex example of a program that allocates a vector (`vect`) of 100 floating point numbers on every even process rank, initialises it, and then copies its value from rank 2 to rank 0. We define the class `even` as a selector for even rank values. For simplicity, we use utilities (i.e., types and *meta-functions*) from the Boost Meta-Programming Library (MPL) [49], on which also the implementation of `mem_wrap` heavily relies. The selector is applied to a rank value using the `apply` generic inner class defined in line 2. We follow the naming convention used in MPL which enables us using existing meta-programming utilities from the MPL library. In line 6, the allocation of the variable `vect` is managed by our memory wrapper which enables the compiler to select the type of wrapper to instantiate (i.e., active or shadow) depending on the rank for which the code is being compiled.

Accessing array elements from a wrapper instance is allowed using the `()` operator which, instead of returning directly the indexed value, instantiates the target wrapper referring to the addressed memory cell. For shadow wrappers, an assignment operator of a value of type `T` resolves to a `NOP` (e.g., loop iteration in line 7 compiled for odd processors) that the compiler optimizations can easily detect and safely remove as dead code. Therefore, line 7 will be preserved for even processes but it will be removed for

```
1  template <class RR = mpl::int_<MY_RANK>>
2  struct top_neigh {
3     template <class Rank>
4     struct apply : mpl::bool_<RR::value+1 == Rank::value> { };
5  };
6  const size_t size = N/NUM_PROCS+2;
7  mem_wrap<carray<float>> u(size,N), tmp(size-2,N);  // Active wrapper
8  mem_wrap<carray<float>, top_neigh> top( u );          // Shadow wraper
9  mem_wrap<carray<float>, bottom_neigh> bottom( u );    // Shadow wrapper
10 // Initialize matrix u...
11 for(unsigned int it=0; it<MAX_ITER; ++it) {
12    u[ slice(size-1, size, 0, N) ] = top[ slice(1, 2, 0, N) ];
13    u[ slice(0, 1, 0, N) ] = bottom[ slice(size-2, size-1, 0, N) ];
14    for (unsigned int i=1; i<size-1; ++i)
15       for (unsigned int j=1; j < N-1; ++j)
16          tmp(i-1,j-1) = 1/4 * ( *u(i-1,j) + *u(i,j+1) + *u(i,j-1) + *u(i+1,j) );
17 }
```

Listing 3.16: C++ Jacobi relaxation

odd ranks. Finally, the assignment operator in line 8, involving the two wrappers, is rewritten as send/receive. For odd ranks, the operation results again in a `NOP`. The `r`n constants, where n is an integer value representing the rank, are defined to easily refer to a process rank. The `[ ]` operator is used to specialize a generic wrapper to refer to a particular memory address space, method signature is shown in line 9 of Listing 3.13. Since 0 is within the set of processes selected by the `even` selector, code will be generated for this process. Because the wrapper object for this process is in the left-hand-side of an assignment, a receive operation is generated, the source of the receive is given by the wrapper on the right-hand-side of the assignment operator, i.e., $r2$. When code is generated for rank 2, a similar mechanism is used that generates a send operation towards the process rank 0.

3.3.5 Jacobi Relaxation

In this section we show how an important class of HPC stencil operations can be expressed in our framework. We use as example the Jacobi relaxation method based on the nearest neighbour communication. A two-dimensional matrix is distributed among the processes, each process having a dependency to the memory cells owned by its direct neighbours. When the data is distributed in a row-wise manner, each process needs to access the memory allocated in the top `MY_RANK+1` and bottom `MY_RANK-1` neighbours. Every process allocates an equal portion of `N/NPROCS+2` matrix rows, where `N` is the matrix size. The two additional rows are used to store the first and last row received from the top and the bottom neighbors.

```
1 shared [N*N/THREADS] float u[N][N];
2 shared [N*N/THREADS] float tmp[N][N];
3 // Initialize matrix u...
4 for( unsigned int it=0; it < MAX_ITER; ++it)
5    upc_forall(unsigned int i=1; i<N-1; i++; &tmp[i][0]) {
6       for (unsigned int j=1; j < N-1; ++j)
7          tmp[i][j] = 1/4 * ( u[i-1][j] + u[i][j+1] + u[i][j-1] + u[i+1][j] );
8    }
```

Listing 3.17: UPC based Jacobi relaxation method

Listing 3.16 shows the Jacobi relaxation algorithm expressed using our method. In lines 8 and 9, two shadow wrappers are generated referring to the `top` and `bottom` neighbours. The top processor selector `top_neigh` is defined in lines 2-5. The selector for the bottom processor `bottom_neigh` is similar with the difference that the expression `RR::value-1 == Rank::value` is used as a selector. Both `top` and `bottom` are instantiated as shadow wrappers on every processor rank because the selector expressions always evaluate to false when applied to the current rank (`MY_RANK`). Lines 12 and 13 implement the neighbor communication. In line 12, a receive operation is generated for the incoming data from the top neighbor process. Unlike previous examples, the rank is not statically specified and the source rank of the message is automatically computed at compile-time in order to avoid any runtime overhead. This is done with the following procedure. The selector of the right hand side expression `top_neigh` is applied to a list of process ranks `PL` generated at compile-time as follows `PL: {0,1,...,MY_RANK-1, MY_RANK+1,..., NUM_PROCS-1}`, where `NUM_PROCS` is the total number of processes defined via a pre-processor directive, and `RR` (i.e., `RefRank`) is set to be `MY_RANK`. The selector is invoked several times as follows: `top_neigh<MY_RANK>::apply<R>`, $\forall R \in PL$. The receive operation is generated using, as a source rank, the value `R` which satisfies the selector, (i.e., `MY_RANK+1`). Speculatively, a send operation is generated towards the bottom neighbor. This requires to invert the `top_neigh` selector previously used to generate the receive operation. We achieve this by invoking the selector in the following way: `top_neigh<R>::apply<MY_RANK>`, $\forall R \in PL$. The semantics is the following, find the processes for which the `top_neigh` selector is satisfied when applied to the current rank value (i.e., `MY_RANK`). For rank values which satisfy the selector, a send operation is generated using as target rank the value of `R` (i.e., `MY_RANK-1`). The communication statements for line 13 are generated similarly but using `bottom_neigh` as selector. The `slice` function indicates the start and end rows and columns of a matrix partition which has to be either transmitted or overwritten by the incoming data.

We compared our Jacoby relaxation implementation with an UPC-based version on a shared memory machine with 10 AMD Opteron cores. The UPC implementation of Jacobi (from [50]) utilized in our experiments is depicted in Listing 3.17. The code uses

Matrix size	MPMD	UPC	Speedup
10x10	0.0129	0.0022	0.14
100x100	0.018	0.023	**1.28**
500x500	0.098	0.205	**1.84**
1000x1000	0.20	0.74	**3.7**
2000x2000	0.61	2.98	**4.9**

TABLE 3.4: Jacobi relaxation execution time (in seconds) and speedup comparison.

a memory layout specifier (i.e., `[...]`) which allows the UPC runtime to distribute the `u` and `tmp` matrices assigning an equal amount of rows to each UPC process (similar to the MPMD code). For a fair comparison, we forced UPC to use MPI as the underlying communication library (`-network=mpi`). Furthermore, we utilized the `-T` flag which enables the UPC compiler to create an executable which runs with a fixed number of threads (i.e., `-T=10`). The Berkley UPC compiler version 2.12.2 with experimental optimization enabled (`-opt`) was utilized. GCC version 4.5.3, with optimization flag `-O3`, was used to compile the MPMD version of the Jacobi in Listing 3.16.

Table 3.4 shows that UPC performs slightly better for very small matrix sizes but, as the problem size increases, the MPMD version significantly outperforms UPC. Unfortunately we could not compile the UPC code for larger matrix sizes as the UPC compiler does not support, in the layout specifier, a block size which is greater than 1MB. We believe that the main source of inefficiency in UPC is the fact that the compiler is not able to vectorize the accesses to neighbor memory cells. Therefore every access to remote memory locations results in a separate communication operation. It is also worth noting that compared to an SPMD-based MPI implementation of the Jacobi, the MPMD version presented here only marginally improve performance. The main advantage is in the simplified programming model which, as the experiments show, does not cause any performance penalty.

3.4 LibWater: A Uniform Approach for Heterogeneous Distributed Memory Programming

3.4.1 The OpenCL Programming Model

OpenCL is an open industry standard for programming heterogeneous systems. The language is designed to support devices with different capabilities such as CPUs, GPUs and accelerators. The platform model comprises a *host* connected to one or more *compute devices*. Each device logically consists of one or more compute units which are further divided into processing elements (PEs). Within a program, the computation is

expressed through the use of special functions called *kernels* that are, for portability reason, compiled at runtime by an OpenCL driver. Interaction with the devices is possible by means of *command-queues* which are defined within a particular OpenCL *context*. Once enqueued, commands – such as the execution of a kernel or the movement of data between host and device memory – are managed by the OpenCL driver which schedules them on the actual physical device.

When a kernel is submitted for execution by the host, an index space is defined and an instance of the kernel (i.e., *work-item*) is executed for each point in that space. Work-items are also organized into work-groups, which provide a more coarse-grained decomposition of the index space. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit and therefore, the size of a work-group can have a significant effect on the runtime performance. Moreover, work-groups share memory therefore synchronization is allowed.

Commands can be enqueued in blocking or non-blocking mode. A non-blocking call places a command on a command-queue and returns immediately to the host, while a blocking-mode call does not return to the host until the command has been executed on the device. This is similar to non-blocking semantics of communication statements explained in 2.2. For synchronization purpose, within a context, *event* objects are generated when kernel and memory commands are submitted to a queue. These objects are used to coordinate execution between commands and enable decoupling between host and devices control flows.

Despite being a well designed language that unleashes the compute power of heterogeneous devices from a single, multi-platform source code base, OpenCL has some drawbacks and limitations. One of the major drawback is that, because being created as a low-level API, a significant amount of boilerplate code is required even for the execution of simple programs. Developers have to be familiar with numerous concepts (i.e., *platform*, *device*, *context*, *queue*, *buffer* and *kernel*) which make the language less attractive to novice programmers. Another important limitation is that, although it was designed to address heterogeneous systems, in case of devices from different vendors, objects belonging to the context of one vendor are not valid for other vendors. This limitation clearly becomes a problem when synchronization of command queues across different contexts is needed.

Moreover OpenCL applications can only take advantage of the local devices present on a single node of a heterogeneous cluster. To exploit devices on different nodes another appropriate programming paradigm is needed. Different paradigms, however, do not have a common memory model. That leaves to the programmer the responsibility to maintain the memory consistency when switching from one model to another.

3.4.2 Related Work

Several projects have been recently proposed to facilitate the programming of clusters with heterogeneous nodes [51–54].

Kim et al. [51] proposed the *SnuCL* framework that extends the original OpenCL semantics to heterogeneous cluster environments. Their work is closely related to ours. SnuCL relies on the OpenCL language with few extensions to directly support collective patterns of MPI. In SnuCL is the programmer responsibility to take care of the efficient data transfers between nodes. In that sense, end users of the SnuCL platform need to have an understanding of MPI collective calls semantics in order to be able to write scalable programs. SnuCL poses a limit to the number of devices which can be addressed by their runtime system, i.e., NVidia accelerators and CPUs.

Also other works have investigated the problem of extending the OpenCL semantics to access a cluster of nodes. The Many GPUs Package (*MGP*) [52] is a library and runtime system that using the MOSIX VCL layer enables unmodified OpenCL applications to be executed on clusters. *Hybrid OpenCL* [53] is based on the FOXC OpenCL runtime and extends it with a network layer that allows the access to devices in a distributed memory system. The *clOpenCL* [54] platform comprises a wrapper library and a set of user-level daemons. Every call to an OpenCL primitive is intercepted by the wrapper which redirects its execution to a specific daemon at a cluster node or to the local runtime. While the objectives of these approaches are similar to ours, none of them provides an abstraction layer to reduce the complexity associated with the OpenCL development and, furthermore, they show a very limited scalability in clusters of 4 to 8 compute nodes.

Besides OpenCL-based approaches, also Compute Unified Device Architecture (CUDA) solutions have been proposed to simplify distributed memory systems programming. *CUDASA* [55] is an extension of the CUDA programming language which extends parallelism to multi-GPU systems and GPU-cluster environments. *rCUDA* [56] is a distributed implementation of the CUDA acAPI that enables shared remote GPGPU in HPC clusters. *cudaMPI* [57] is a message passing library for distributed-memory GPU clusters that extends the MPI interface to work with data stored on the GPU using the CUDA programming interface. All of these approaches are limited to devices that support CUDA, i.e., NVidia GPU accelerators, and therefore they cannot be used to address heterogeneous systems which combines CPUs and accelerators from different vendors.

Device Management (wtr_)	
`void` **init_devices**`('DQL', ...)`	`device` **get_device**`('DQL', ...)`
`int` **get_num_devices**`()`	`void` **release_devices**`()`
`void` **print_device_infos**`(device)`	
Buffer Management **(wtr_)**	
`buffer` **create_buffer**`(device, mem_flag, size, evt)`	
`void` **write_buffer**`(buffer, size, source_ptr, wait_evt, evt)`	
`void` **read_buffer**`(buffer, size, dest_ptr, wait_evt, evt)`	
`void` **release_buffer**`(buffer, wait_evt, evt)`	
Kernel Management **(wtr_)**	
`kernel` **create_kernel**`(device, name, kernel_name, build_options, flag, evt)`	
`void` **run_kernel**`(kernel, work_dim, global_work_size, local_work_size, wait_evt, evt, num_args, ...)`	
`void` **release_kernel**`(kernel, wait_evt, evt)`	
Event Management **(wtr_)**	
`event` **create_event**`()`	`void` **release_event**`(evt)`
`event` **merge_events**`(num, ...)`	`void` **wait_for_events**`(num, ...)`
`void` **init_event_array**`(num, evt)`	
`void` **release_event_array**`(num, evt)`	

TABLE 3.5: The complete *libWater* API.

3.4.3 The LibWater Programming Interface

In this section we present an overview of *libWater*'s interface. A more detailed description of the rationale behind the design of the API can be found in [36].

libWater is a C/C++ library-based extension of the OpenCL programming paradigm. It inherits the main principles from the OpenCL programming model trying to overcome its limitations. While maintaining the notion of host and device code, *libWater* exposes a very simple programming interface based on four key concepts: *device*, *buffer*, *kernel* and *event*. A *device* represents a compute device, but differently from the original paradigm this single object is an abstraction of the OpenCL platform, device, queue and context concepts.

Table 3.5 presents the *complete* API of the *libWater* library. The prefix `wtr_` and the C language pointer syntax has been removed from the table for readability reasons. Initialization and selection of devices is done, respectively, by using the `wtr_init_devices` and the `wtr_get_device` routines. Once a *device* is created, it is possible to allocate data and execute computation on it. In *libWater*, this is done through the use of the

buffer and the *kernel* concepts. These two objects are similar to their respective OpenCL versions, with the main difference that, during their creation, they are bound to a specific device. For this reason no device must be specified for buffer and kernel related functions. The principal kernel functions are `wtr_create_kernel` and `wtr_run_kernel`. The former receives as parameter a `flag` that specifies whether the `name` input argument contains the kernel code or it is the name of a file containing the OpenCL kernel. The latter is used for executing a kernel in the previously bound device. The parameters `work_dim`, `global_work_size` and `local_work_size` are the same specified in the OpenCL `clEnqueueNDRangeKernel`. The `num_args` parameter states the number of input arguments accepted by the kernel. This parameter is followed by a list of a variable number of pairs. Each pair consists of a size (in bytes) and a pointer to the corresponding kernel argument. The first value of the pair distinguishes between buffers – when is equal to 0 – or a valid address in the host memory. The fourth concept in *libWater* is the *event* object. Most of *kernel* and *buffer* functions have one or two parameters called `wait_evt` and `evt`. The latter is an output argument which is used by the invoked command to generate an event object. If not specified, *libWater* assumes blocking semantics for the routine. The former specifies the event object on which the execution of the command depends. If not present, the command has no dependencies and thus it can be immediately executed. Since there can be a dependency between several commands, the `wtr_merge_events` function can be used to merge multiple event objects into one.

The last major difference between *libWater* and the OpenCL model is the fact that initialization and release of buffers and kernels can be invoked using a non-blocking semantics. The main reason for this is to increase the amount of operations that the runtime system can overlap. In the next section we explain how dependency information enforced by events are then exploited by *libWater*'s runtime system.

3.4.4 The LibWater Distributed Runtime System

While the main focus of the programming interface of *libWater* is on simplicity and productivity, the underlying runtime system aims at low resource utilization and high scalability. Calls to *libWater* routines are forwarded to a distributed runtime system which is responsible for dispatching the OpenCL commands to the addressed devices and for transparently and efficiently moving data across the cluster nodes. The *libWater* distributed runtime is written in C++ and internally uses several paradigms, such as pthreads, OpenMP and MPI for parallelization.

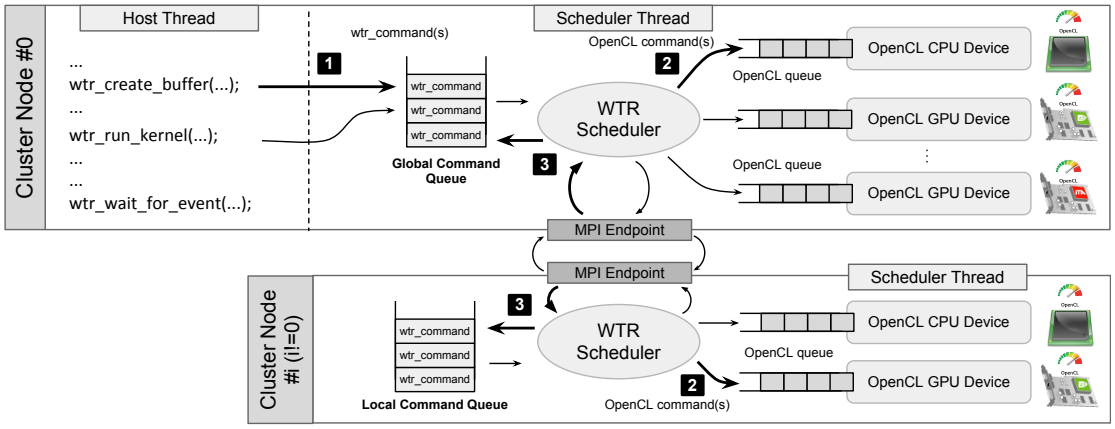


FIGURE 3.3: *libWater*'s distributed runtime system architecture.

3.4.4.1 Runtime System Architecture

Figure 3.3 shows the organization of the *libWater* distributed runtime system. The *host code*, which directly interacts with *libWater*'s routines, runs on the so called *root node*, which by default is the cluster node with rank 0. This thread will be referred to as the *host thread*. In the background, a second thread, i.e., the *scheduler thread*, is allocated to execute an instance of the `WTRScheduler`. On the remaining cluster nodes, a single *scheduler thread* is spawned independently of the number of available devices (only one MPI process is allocated per node). This thread executes an instance of the `WTRScheduler` which represents the backbone of *libWater*'s distributed runtime system.

Each `WTRScheduler` continuously dequeues `wtr_commands` from the local command queue. `wtr_commands` in the system are generated in two ways, either by (*i*) *libWater*'s routines (step 1), or (*ii*) by delegation from the root scheduler (step 3). Calls to the *libWater*'s interface are converted into command descriptors (i.e., command design pattern) and immediately enqueued into the root node local command queue (step 1) of Figure 3.3. Since all `wtr_commands` are generated by the root node itself, we refer to its queue as the runtime *global* command queue.

`wtr_commands` are either wrappers for OpenCL commands or data transfer jobs (i.e., `send_job` or `recv_job`) which are generated by the library routines whenever the device addressed by a read or write buffer operation is located in a remote (i.e., $rank \neq 0$) compute node. The descriptor of a `wtr_command` is *self-contained* since it carries all the information necessary for its execution. To be portable across cluster nodes, OpenCL objects such as kernels, buffers and events are identified, within the `wtr_command` object, by a unique ID. The root scheduler continuously fetches the `wtr_commands` from the global command queue, decodes its content and – depending on the targeted device – dispatches the command to the correct node. When the `wtr_command` addresses one of

the local OpenCL devices, the corresponding OpenCL command is created and enqueued into the device command queue (step 2). When a remote OpenCL device is addressed, an MPI message is generated – serializing the content of the `wtr_command` descriptor – and dispatched to the cluster node hosting the requested device. The `WTRScheduler` of the target node then de-serializes the `wtr_command` and, instead of immediately executing it, enqueues the `wtr_command` instance into the local command queue (step 3). The same `WTRScheduler` is then responsible to dispatch the corresponding OpenCL command into one of its local device queues (step 2).

The heartbeat of the `WTRScheduler` is an advanced event system which allows the management of an entire compute node – hosting multiple OpenCL devices – using only a single application thread. Because one instance of the `WTRScheduler` runs on every cluster node, trying to keep the resource usage as low as possible is of paramount importance in order to avoid wasting CPU cycles which can be used to run an OpenCL kernel. Different from related work, e.g., the SnuCL runtime system [51], which exclusively reserves an entire cluster node and a physical CPU core in each compute node only for scheduling purposes, our system does not exclusively reserve any user resources for scheduling. Furthermore, using a single thread, for both executing local `wtr_commands` and for performing scheduling decisions, reduces the amount of synchronization since accesses to event and the command queues do not need to be synchronized.

Relying on a single thread can however easily become a performance bottleneck. An interesting example is the interaction with MPI routines. By default many MPI implementations realize blocking behaviour with a *spin-lock* mechanism in order to minimize latency. This means for example that a blocking receive, waiting for a message from the communication channel, continuously checks for incoming data usually saturating a CPU core. In an environment like ours, where CPU cores may be used to run OpenCL kernels, this behaviour must be avoided. Our solution is to avoid in every event handler routine any call to blocking MPI or OpenCL routines and always use the non-blocking semantics. The main idea is the creation of periodic events, handled by the event system using a priority queue based on timestamps, to check for the completion of pending operations. For OpenCL routines, we exploit the OpenCL event system and the associated callback mechanism. In this way, the `WTRScheduler` is able to dispatch several commands on the OpenCL devices, or MPI data transfers, which although being issued sequentially (by the single flow of the execution) are concurrently executed by the available resources (i.e., OpenCL devices and the network controller). The same event-based technique utilized to manage multiple OpenCL devices in a single node is also exploited on the large scale across cluster nodes.

```
1  wtr_init_devices(WTR_ALL);
2  wtr_event* evts[get_num_devices()];
3  for (int i=0; i<get_num_devices(); ++i) {
4     size_t offset=size/2*i;
5     wtr_device* dev = wtr_get_device(i);
6     assert(dev != NULL && "Device does not exist!");
7     wtr_event* e[8];
8     wtr_init_event_array(7,e);
9     wtr_kernel* kern = wtr_create_kernel(dev,"kernel.cl","fun", "", WTR_SOURCE, e+0);
10    wtr_buffer* buff = wtr_create_buffer(dev, WTR_MEM_READ_WRITE, size/2, e+1);
11    wtr_write_buffer(buff, size/2, ptr+offset, e+1, e+2);
12    e[7] = wtr_merge_events(2, e+0, e+2);
13    wtr_run_kernel(kern,1,(size_t[1]){size/2},NULL,e+7,e+3,2,
14                         0, buff,
15                         sizeof(size_t), &offset);
16    wtr_read_buffer(buff, size/2, ptr+offset, e+3, e+4);
17    wtr_release_buffer(buff, e+4, e+5);
18    wtr_release_kernel(kern, e+3, e+6);
19    evts[i] = wtr_merge_events(2, e+5, e+6);
20    wtr_release_event_array(8, e);
21 }
22 /* Blocks until buffers and kernels are released */
23 wtr_wait_for_events(2, evts+0, evts+1);
24 wtr_release_event_array(2, evts);
```

LISTING 3.18: A complete multi-device program example using *libWater*'s routines

3.4.4.2 Event-based Command Scheduling

As already explained in the previous Section, *libWater* puts a strong emphasis on events. Following the semantics of OpenCL, dependency information enforced by programmers are used to select `wtr_commands`, which can be safely enqueued into one of the cluster nodes. *libWater* provides an event object, i.e., `wtr_event`. Internally, `wtr_events` are mapped either to an OpenCL `cl_event` object, or to a `wtr_command` identifier which is automatically generated for each `wtr_command` enqueued into the system. These dependencies allow the runtime system to organize enqueued `wtr_commands` into a DAG.

A complete multi-device *libWater*-based host program is shown in Listing 3.18. This code initializes all the available devices. For each device the code in Listing 3.18 does the following: create a kernel (i.e., `kern`, in line 9) and a read/write buffer (i.e., `buff`, line 10). Then the contents from the host memory is written into the device buffer by the `wtr_write_buffer` command (line 11) and the `wtr_run_kernel` command is issued providing `buff` as an input argument (lines 14-16). The computed result is then retrieved by the `wtr_read_buffer` command (line 16) which moves data from the device memory back to the host memory.

From the runtime system point of view, the execution of the previous code generates a set of dependent commands structured as the DAG depicted in Figure 3.4. The DAG

$G(V, E)$ is composed of vertices, i.e., `wtr_commands` $\in V$, interconnected through directed edges $(a, b) \in E \mid a, b \in V$, or *events*, which guarantee that the correct order of execution, and therefore the semantics of the input program, is maintained. The *set* of dependencies associated with a command $c \in V$ is defined as $c.deps = \{v \in V \mid (v, c) \in E\}$. It is worth mentioning that not all *libWater* library routines generate a corresponding `wtr_command`. For example, creation, merging and release of events are only meaningful in the root node, therefore there is no need for serializing them. In Figure 3.4, each `wtr_command` carries a descriptor in the form $x|y$ where x represents the node rank, $c.node\_id$, on which the targeted device, $c.dev\_id$, is hosted and y is the unique command identifier assigned by the runtime system. As already mentioned, for buffer operations on remote devices (i.e., device on node 1) explicit data transfers are automatically inserted by the *libWater* library (e.g., `wtr_commands` 10 and 14).

Events determine when a `wtr_command` can be scheduled for execution. The scheduler uses a *just-in-time* strategy to select the next `wtr_command` from the local command queue. The logic works as follows: enqueued `wtr_commands` are analyzed in a FIFO fashion and, for each ready command, the scheduler checks whether dependencies – explicitly specified by event objects – are satisfied. If a command has no dependencies, it can be executed. Since the host program generates all the commands solely on the root node, scheduling is done at this node. However, a centralized scheduler on a single node is not an effective strategy since it limits command throughput and thus the overall scalability of the system.

In order to solve this problem, we rely on the fact that the OpenCL runtime system already has the capability of scheduling commands and handling dependencies by using events. It is worth noting that in OpenCL this mechanism is limited since events cannot be used to perform command synchronization across different contexts. *libWater* unifies event handling through `WTRScheduler` instances which manage inter-context synchronization and offload intra-context synchronization to the OpenCL driver.

We implemented a *three-level hierarchal scheduling* approach as described in Algorithm 1. At the top level, the root node of the *libWater* runtime system *pro-actively* schedules `wtr_commands` from the global queue to the targeted cluster nodes. *cmd*, fetched from the command queue, is sent to the target node (i.e., *cmd.node_id*) only if each of its dependent commands (i.e., the set *cmd.deps*) are to be executed on the same remote node (lines 7–10). The second level scheduling is local to each node, lines 12–15. The scheduler checks whether *cmd* only depends on `wtr_commands` addressing the same OpenCL device. In such case, the command is enqueued into the corresponding device queue (i.e., *dev.dev_id*) and dependencies are mapped to local OpenCL events. Alternatively,

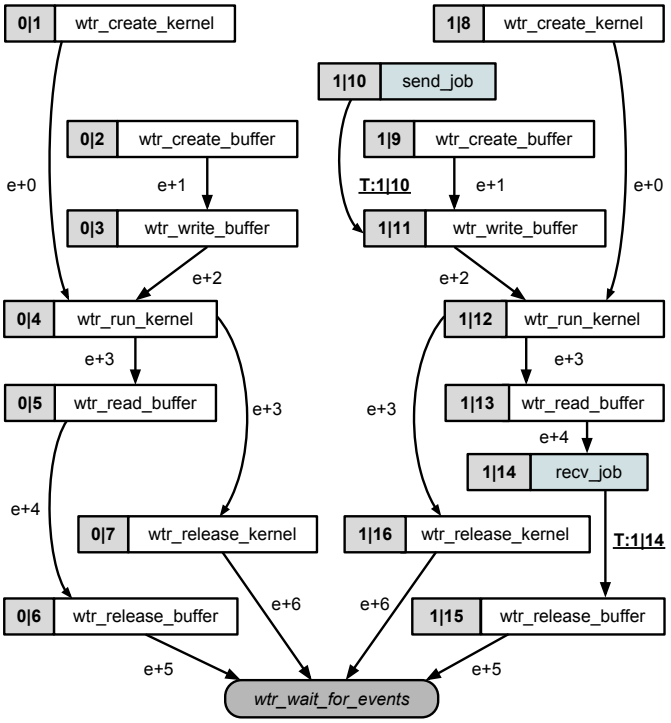


FIGURE 3.4: DAG of `wtr_commands` generated during the execution of the code snippet in Listing 3.18.

if a `wtr_command` C_1 depends on a second `wtr_command` C_2, scheduled in another context (of the same node), the local `WTRScheduler` ensures that C_1 is not enqueued into the OpenCL device queue before C_2 is completed. The third-level scheduling is implemented by the OpenCL runtime system itself which is responsible of managing single device queues. If *cmd* cannot be scheduled, due to unsatisfied dependencies, then it is pushed back in the command queue.

Command dependencies are automatically updated when a `wtr_command` c completes. Locally, a *command completion event* is generated. The associated *callback* function is depicted in Algorithm 2. The function removes, for every command in the local queue, any dependence on c. Additionally, nodes notify the root scheduler with a message (lines 8–10) triggering a similar completion event internally at node 0. In such a way, commands in the global queue waiting for the completion of c can be scheduled – depending on the targeted device – either to a local device or to a remote node.

This multi-level scheduling allows the runtime system to hide the costs of the scheduling, as well as data transfers, with the actual work being done by the devices in the background. The main idea is to use non-blocking semantics when OpenCL commands are scheduled in the corresponding devices. In this way, the `WTRScheduler` can continuously dispatch commands to other devices or move data from and to the root node. In the example in Figure 3.4, commands `0|1` and `0|2` can be executed in parallel. Events

Algorithm 1 The `WTR_Scheduler`'s algorithm

```
1: Input: cmd_queue                                   ▷ Local FIFO wtr_command queue
2: Input: my_rank                                     ▷ MPI process rank
3: procedure ScheduleCMD(cmd_queue : input, my_rank : input)
4:   while true do
5:     cmd ← cmd_queue.pop();
6:     if cmd.node_id ≠ my_rank then
7:       if ∀ d ∈ cmd.deps | d.node_id = cmd.node_id then
8:         send(cmd, cmd.node_id, SCHED)              ▷ Delegates cmd to node
9:         continue
10:      end if
11:    else
12:      if ∀ d ∈ cmd.deps | d.dev_id = cmd.dev_id then
13:        issue(cmd.cl_cmd, cmd.deps)                ▷ Delegates to corresp. dev.
14:        continue
15:      end if
16:    end if
17:    cmd_queue.push(cmd)                            ▷ Failed to schedule event due to deps.
18:  end while
19: end procedure
```

Algorithm 2 Update `wtr_command` dependencies

```
1: Input: c                                           ▷ Completed command
2: Input: cmd_queue                                   ▷ Local command queue
3: Input: my_rank                                     ▷ Rank associated to executing process
4: procedure CallbackCmdCompletion(c : Input, cmd_queue : Input, my_rank : Input)
5:   for cmd in cmd_queue do
6:     cmd.deps.remove(c)                             ▷ Removes c from the dependencies
7:   end for
8:   if my_rank ≠ 0 then
9:     send(c, 0, DONE)                               ▷ Notifies the root node of c completion
10:  end if
11: end procedure
```

at addresses $e+0$ and $e+1$ are handled by the root `WTRScheduler` since the OpenCL standard does not allow non-blocking semantics for these operations. The remaining commands (i.e., `0|3`, `0|4` and `0|5`) are inserted asynchronously into the OpenCL device queue of node 0, upon completion of commands `0|1` and `0|2`. Events `e+2` and `e+3` are therefore handled directly by the OpenCL runtime system. Following the same logic, `wtr_commands` addressing the second OpenCL device (i.e., $1|*$) are sent to the node with rank 1. The blocking function `wtr_wait_for_events` stops the execution of the host until the release operations on both nodes have completed.

3.4.4.3 The DCR Optimization

The underlying architecture of the *libWater* runtime system and the emphasis on events, promoted by its interface, enables several runtime optimizations which are transparent to the user. This capability is a direct consequence of adhering to the OpenCL queuing

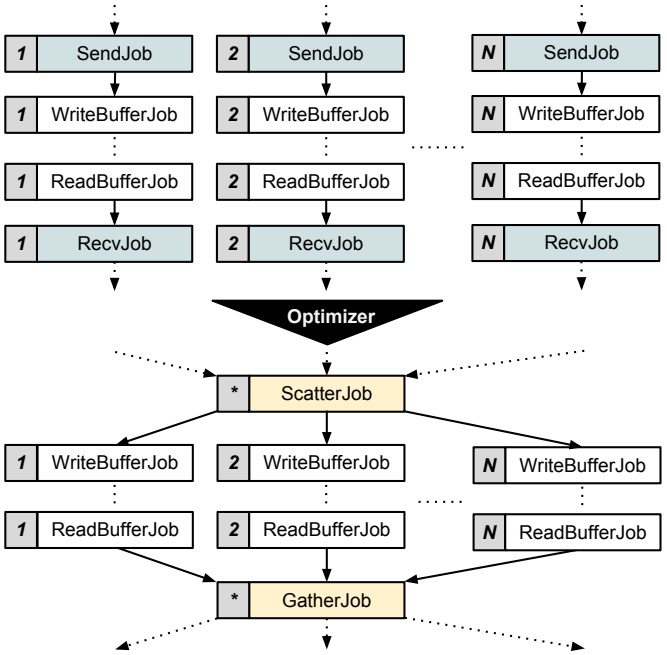


FIGURE 3.5: Dynamic collective communication pattern replacement (DCR) optimization.

semantics. While commands are being enqueued into the system, a command DAG (as shown in Figure 3.4) is internally created. Since OpenCL issues commands to the appropriate device only when an explicit flush is invoked by the programmer, the runtime system can analyze large portions of the application DAG and optimize it for improving scalability.

An optimization which has been implemented in the *libWater* runtime system is the dynamic detection and replacement of collective communication patterns (DCR). Whenever the addressed device is not hosted in the root node, a call to `wtr_write_buffer` and `wtr_read_buffer` respectively generates an MPI send and receive operation. When an OpenCL application is distributed among all available devices, input buffers are usually either split or replicated between compute nodes. This parallelization strategy is common and it results in a DAG containing several send/receive transfer operations for every device of the cluster. An example is depicted in Figure 3.5 which represents a realistic DAG resulting from the splitting of an input and output buffer among a set of N OpenCL devices.

Point-to-point data transfers performed by the *libWater* runtime system imply an increased latency when compared with the native MPI send or receive routines. The reason for that is the polling mechanism implemented by the *libWater* runtime system – mainly employed to save node resources – which replaces the spin-lock mechanism commonly used by MPI libraries. Additionally, the number of required data transfers is directly proportional to the cluster nodes (and thus devices). This results in a large number of

Algorithm 3 DCR pattern recognition

```
1: Input/Output: G(V, E)                                          ▷ Command DAG
2: Input: root                                                    ▷ DAG's entry point
3: function REPLACE_COLLECTIVE_PATTERNS(G(V, E) : input/output, root : input)
4:   for ∀t in BFS(G(V, E), root) do
5:     if t.type ∈ {SendJob, RecvJob} then
6:       jobs[t.node_id].append(t)                                ▷ Orders transfer jobs
7:     end if
8:   end for
9:   for i ← 0 to min({jobs[k].length : ∀k | 0 ≤ k < N}) do
10:    pattern ← 0
11:    for j ← 1 to N do
12:      if jobs[j][i].type ≠ jobs[j − 1][i].type then break
13:      if jobs[j][i].buf = jobs[j − 1][i].buf ∧ jobs[j][i].size = jobs[j − 1][i].size then
14:        if pattern = 2 then break
15:        pattern ← 1                                  ▷ Sequence recognized as a broadcast
16:      end if
17:      if jobs[j][i].buf = jobs[j − 1][i].buf + jobs[j − 1][i].size then
18:        if pattern = 1 then break
19:        pattern ← 2                                  ▷ Sequence can be either scatter or gather
20:      end if
21:    end for
22:    if j ≠ N ∨ pattern = 0 then continue
23:    if pattern = 1 ∧ jobs[0][i].type = SendJob then
24:      replace_with_broadcast(jobs, i)
25:    else
26:      if jobs[0][i].type = SendJob then replace_with_scatter(jobs, i)
27:      else replace_with_gather(jobs, i)
28:    end if
29:  end for
30: end function
```

commands being dispatched by the runtime system and consecutively negatively impacts the overall scalability. MPI offers a large set of communication patterns called *collective operations* [3]. These routines are highly efficient since nearly all modern supercomputers and high-performance networks provide specialized hardware support for collective operations [58]. Additionally, the implementation of such collective operations employs dynamic runtime tuning techniques which choose, among a set of semantically equivalent algorithms, which best fit the underlying network topology and architecture [59–61].

Related work analyzed the problem of automatic detection of collective patterns from a set of point-to-point communications. This technique is common in MPI performance tools which are capable of detecting such patterns via post-mortem analysis of program traces [10]. The general problem of collective communication pattern detection is NP-hard, however, under particular restrictions the problem can be solved in polynomial time. A more recent work [9] proposed a fast solution, with a complexity of $\mathcal{O}(n \log n)$, which makes the approach more suitable for runtime systems.

The goal of our DCR optimization algorithm is to analyze the command DAG isolating

point-to-point data transfers and detect whether a subset of those resembles one of the collective patterns supported by MPI. This is possible since – if the application is carefully written using events for command synchronization – the command DAG will be available to the runtime system scheduler before the first blocking command is invoked (e.g., `wtr_wait_for_event`(s)). Since data transfers in our environment have all the same root (the node 0), the analysis for patterns is simplified. The pattern recognition is presented in Algorithm 3. The command DAG is traversed once in breadth-first order (lines 4–8), transfer commands are collected into N separate lists (i.e., variable *jobs*), one per device. On the extracted N lists, pattern analysis is performed, lines 9–29. The check is done by considering elements having the same position within the transfer job lists. Furthermore, the check is simplified by the fact that every send and receive `wtr_command` carries information of the buffer location (*buf*) and the amount of bytes being transfered (*size*). The pattern analysis starts by taking the first transfer `wtr_command` from the N lists and by checking against a supported pattern, i.e., *broadcast*, *scatter* or *gather*. For instance, in a broadcast N send operations are expected where $\forall\, i \mid 0 \leq i < N-1,\ buf_i = buf_{i+1} \vee size_i = size_{i+1}$. If the check fails, the transfer jobs are tested against a scatter or gather pattern $\forall\, i \mid 0 \leq i < N-1,\ buf_i + size_i = buf_{i+1}$.

Once a pattern is recognized, single point-to-point transfers are removed from the command DAG and replaced by the corresponding collective communication operation, lines 24, 26 and 27. A visual example of this optimization is depicted in Figure 3.5, where multiple send operations are collapsed into a single scatter operation and correspondingly, receives are rewritten as a gather operation. By doing so, dependencies between successive commands are updated in order to keep the semantics of the input program unchanged.

Since collective operations must involve all the processes in a communicator, the current implementation of the DCR optimization works when all the initialized devices participate in the computation. Therefore, the analysis is limited to regular applications which *must* involve all OpenCL devices in data transfers. This is important to keep the pattern recognition algorithm simple and fast, since this optimization is applied during runtime. In the future, we plan to improve this mechanism by extending the pattern recognition also to sub-groups of devices.

3.4.5 Experimental Evaluation

We used *libWater* to encode 6 computational codes, some of them taken from various OpenCL benchmarking suites (i.e., AMD and IBM), and studied their scalability. Four of the OpenCL kernels were optimized for local memory, i.e., `PerlinNoise` (from IBM),

Site	Vienna Scientific Cluster	Barcelona Supercomputing Center
Cluster	VSC2	MinoTauro GPU Cluster
Max # of nodes	1.314	128
Processors	2 x AMD Opteron 6132 HE	2 x Intel Xeon E5649
Cores per node	2 x 8	2 x 6
Clock Frequency	2.2 GHz	2.5 GHz
Memory per Node	32 GB DDR3	24 GB DDR3
GPUs	–	2 x Nvidia M2090
Interconnection	Infiniband 4x QDR	Infiniband 4x QDR
Open MPI version	1.6.1	1.6.1
OpenCL version	AMD APP 2.6	CUDA 4.1
Top500 (June 2013)	238th	442th

TABLE 3.6: The VSC2 and BSC experimental target architecture.

`NBody` (from AMD), `Floyd` and `kNN` manually written by us. For the remaining two codes, `MatrixMul` and `LinReg` we used a naive implementation unoptimized for what concern local memory. The Table 3.7 shows, for each code, the number of input and output buffers used by the application. We define a buffer as *splittable* when its content can be distributed among the devices. The nature of a buffer is strictly related to the algorithm being implemented within the OpenCL kernel, and thus the application. Non splittable buffers are always replicated on every device.

For the scalability analysis we used two large-scale production clusters, the Vienna Scientific Cluster VSC2 [62] and the BSC's MinoTauro GPU Cluster [63]. Details of the systems and their respective positions in the Top500 [19] are depicted in Table 3.6. A second study was conducted to test the suitability of *libWater* to exploit the computational capabilities of a heterogeneous cluster configuration. For this purpose we used a cluster, composed of 3 compute nodes (i.e., `mc1`, `mc2` and `mc3`), custom made with *off-the-shelf* GPU accelerators. The hardware details are depicted in Table 3.8.

The six applications utilized for our study are listed in Table 3.7. We started from a pure OpenCL implementation and rewrote them using *libWater*. In Table 3.7, we show the reduction, in terms of lines of code, achieved when the application is written using our library. It is worth mentioning that while the original OpenCL applications were single device codes, the *libWater* based implementation is instead multi-device code. On average, we were able to reduce the lines of the host code by approximately a factor of 2 due to the higher level abstractions provided by *libWater*.

Application	OpenCL LOC	*libWater* LOC	Input size	Input/Output buffers (*splittable*)	Short Description
`PerlinNoise`	412	**301**	20K x 20K	0(0) / 1(1)	Gradient noise generator
`Nbody`	450	**324**	600K bodies	2(0) / 2(2)	N-body simulation
`kNN`	234	**101**	*ref*: 8M, *query*: 80K	2(1) / 2(1)	k-nearest neighbor
`Floyd`	222	**113**	8K, Adj. mat. 64K	1(0) /1(0)	Floyd-Warshall
`MatrixMul`	219	**104**	7Kx7K ($A = B = C$)	2(1) / 1(1)	Matrix Multiplication
`LinReg`	298	**149**	1000K	4(2) / 1(1)	Linear regression

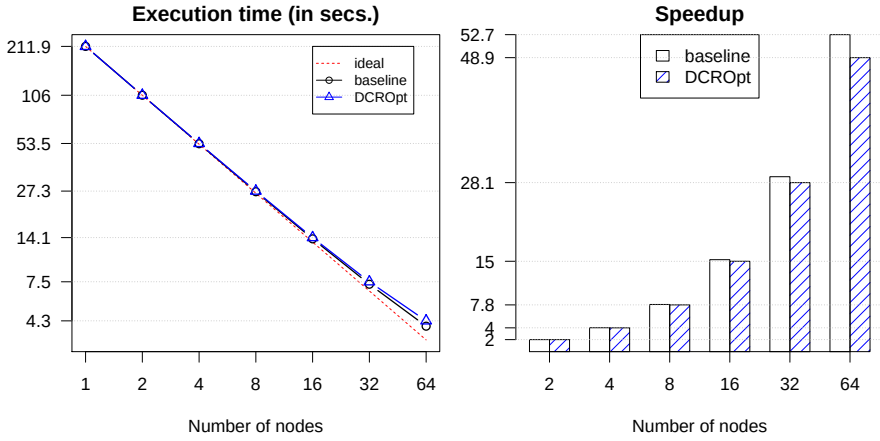
TABLE 3.7: Application codes used for *libWater* evaluation.

3.4.5.1 Homogeneous CPU cluster

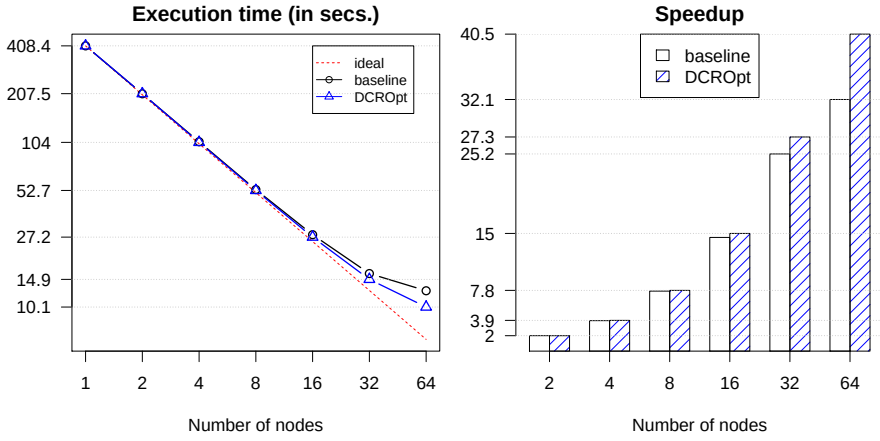
The applications shown in Table 3.7 were executed on the VSC2 homogeneous CPU cluster. We were able to access up to 64 compute nodes with a total of 1024 CPU cores. Since the 2 AMD CPUs which are hosted per node are considered by the OpenCL driver as a single device, the speedup was computed based on the number of compute nodes (and thus OpenCL devices) instead of single CPU cores. The workload partitioning is implemented, for each test case, by assigning to each OpenCL device an equal amount of work.

The scalability tests were performed in the following way: the original OpenCL version of the applications were executed in a single node and their execution times used as a reference measurement. *libWater* was then used for node numbers ranging from 2 to 64. The main differences between the original version of the application codes and the one written using *libWater* are mainly in the host code. The kernel code was slightly modified only to forward the *offset* value used by the workload partitioning (as shown in Listing 3.18). We computed the ideal scaling for each application using the reference execution time and dividing it by the number of nodes. We conducted experiments with *libWater* by using two different settings: the first, named *baseline*, uses the runtime system without dynamic optimizations enabled; the second, DCR, uses the collective pattern replacement mechanism as described in Section 3.4.4.3. The results of our experiments are depicted in Figures 3.6 and 3.7.

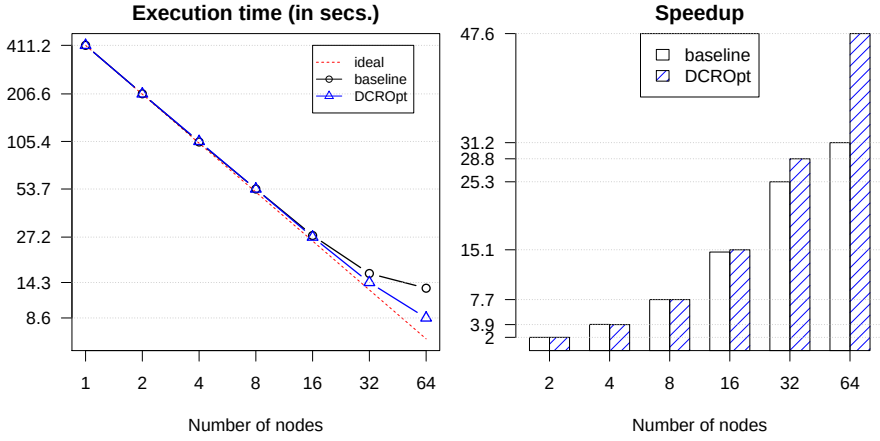
For each of the six applications, we show the execution time (in seconds) for up to 64 devices and the corresponding speedup with respect to a single node. Overall, we observe that our approach scales almost linearly, especially for those codes using few input/output buffers. `PerlinNoise`, Figure 3.6(a), is an example of those, since it has no dependencies on input buffers and the data produced by the kernel is distributed between the devices. For such code, the baseline configuration of our runtime system achieves a speedup of 53 for 64 nodes, and thus an efficiency of 83%. When the number and size of the input/output buffers increases, the efficiency of our system decreases.



(a) `PerlinNoise`



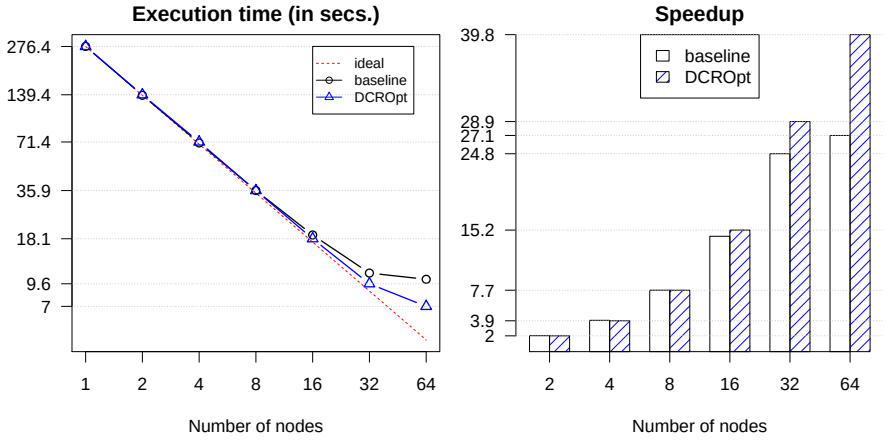
(b) `NBody`



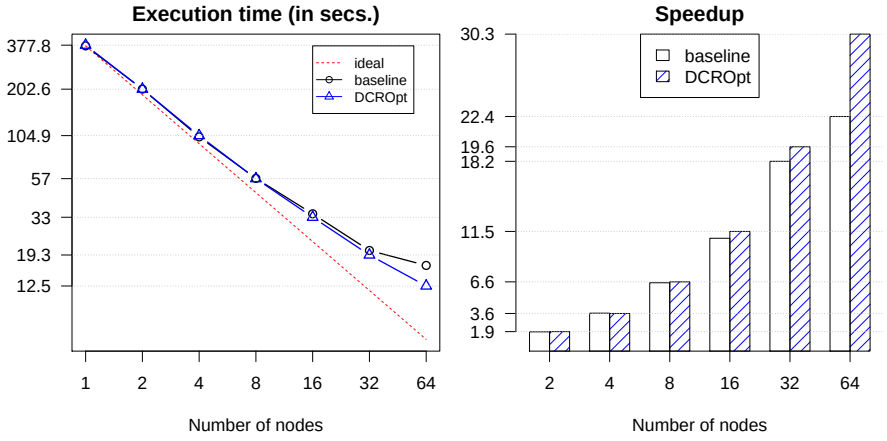
(c) `floyd`

FIGURE 3.6: Strong scaling of *libWater* on the VSC2 (1 of 2)

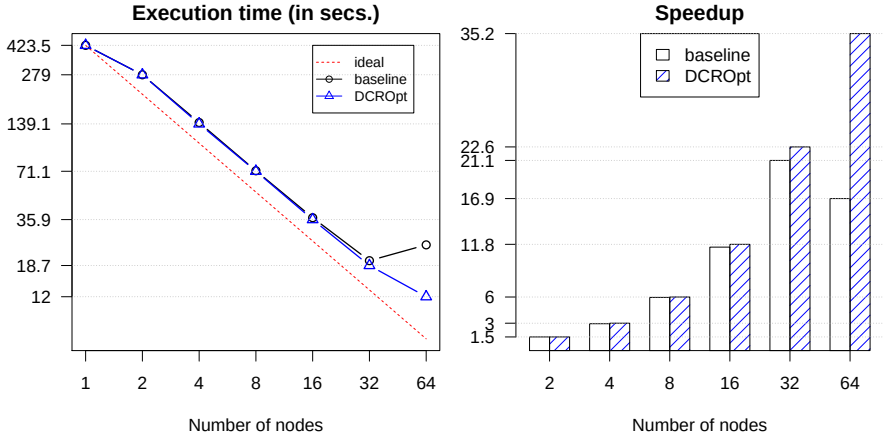
The worst case is represented by the `LinReg` application, Figure 3.7(c), which stops scaling after 32 nodes. This kernel has 4 input buffers, 2 of them are not splittable (because of dependencies within the kernel code) and therefore must be replicated on every node. The remaining 2 input and output buffers are instead splittable. For such



(a) `kNN`



(b) `MatrixMul`



(c) `LinReg`

FIGURE 3.7: Strong scaling of *libWater* on the VSC2 (2 of 2)

code we have an immediate decrease (75% on two nodes) of the efficiency. This is because the kernel execution is delayed due to the fact that several `wtr_commands` are executed (and transfered to the target nodes) to create and initialize the input/output buffers. However this delay is a constant and system efficiency remains almost unvaried

up to 16 nodes. On 32 and 64 nodes the efficiency of the baseline runtime system starts decreasing significantly.

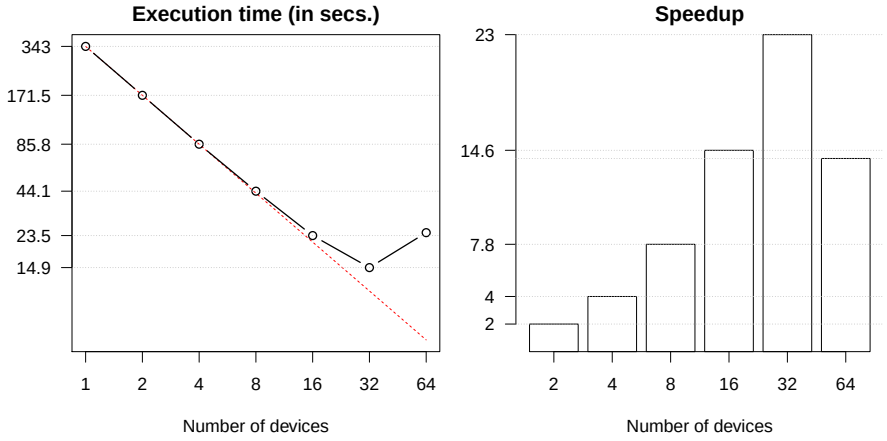
This problem is largely addressed by the dynamic collective pattern replacement, i.e., DCR, optimization which was introduced in Section 3.4.4.3. This optimization reduces the load on the scheduler since it replaces several single transfer jobs with one collective operation. In `LinReg` this optimization improves the scalability of the system by a factor of 2 achieving an efficiency of 55%. A small effect of this optimization can be observed for smaller node configurations because collective operations are optimized for a large number of nodes. An interesting result is the effect of the DCR optimization on the `PerlinNoise` test case. In such a case, the DCR optimization fails to improve performance over the baseline. The reason is that collective operations are blocking while point-to-point communications in the runtime system are non-blocking thereby allowing overlapping of multiple transfers. The synchronization costs introduced by the gather operation is therefore not properly compensated by the amount of exchanged data. We believe that this problem can be eliminated by using *non-blocking collective* routines which have been introduced in the latest MPI standard [3] and will soon be available in mainstream MPI libraries. Additionally, since this optimization is done dynamically, and therefore the amount of data being transfered is known by the scheduler, heuristics can be integrated to decide when such optimization should be applied.

On average, *libWater* achieves an efficiency of 80% on 32 nodes and 64% when 64 nodes are used. Without the DCR optimization the system has an efficiency of 47% on 64 nodes. This means that the DCR optimization improves the system efficiency by 17% on 64 nodes and we expect this value to increase proportionally with the number of nodes.

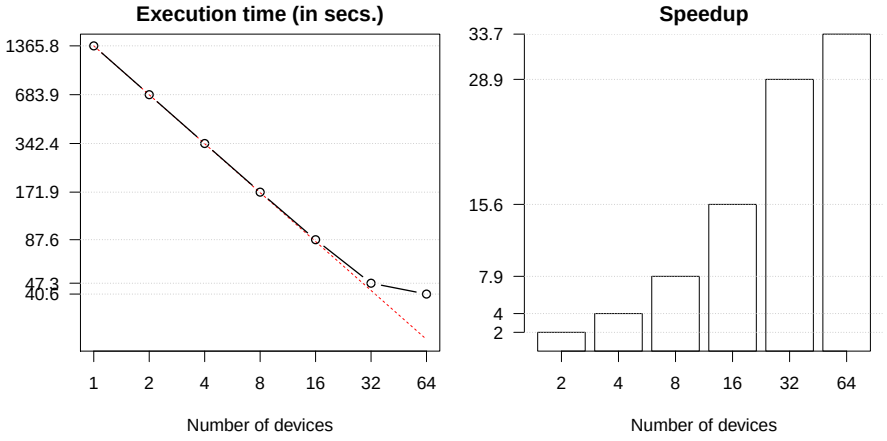
3.4.5.2 Homogeneous GPU cluster

The second experiment assesses the scalability of the *libWater* runtime system on the MinoTauro GPU cluster, hosting two OpenCL devices per single node. The N-body simulation described in Table 3.7 was executed with multiple problem sizes. We were able to access up to 32 nodes of the MinoTauro cluster with a total of 64 GPU devices. In all the experiments, the workload was equally partitioned between the available devices. The optimization of the N-body simulation on the GPU processor is an active research problem [64–67]. The problem is well known to be suitable for the GPU architecture and in case of an high number of particles for cluster of GPUs.

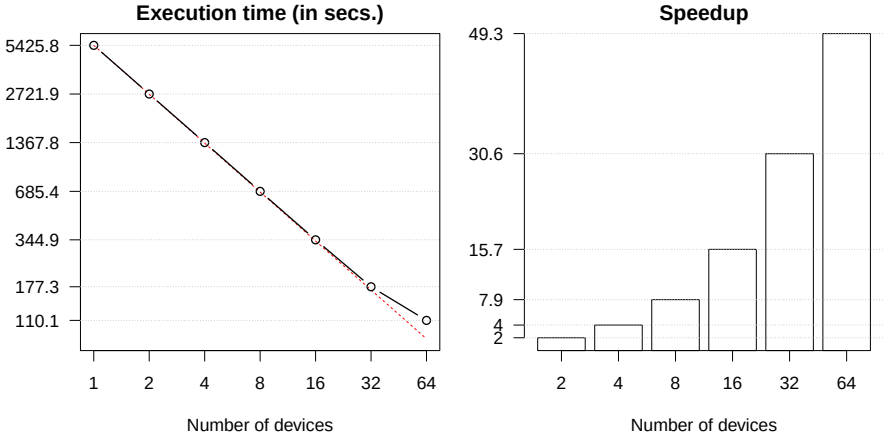
We ran the `NBody` test case using 3 different input sizes that show the benefit of using an high number of GPUs in case of large number of bodies in the test. The results of our



(a) `NBody 2400K`



(b) `Nbody 4800K`



(c) `NBody 9600K`

FIGURE 3.8: Strong scaling of `NBody` on the BSC's MinoTauro GPU Cluster

experiments are depicted in Figure 3.8. The 3 tests were conducted respectively with an input size of 2 (Figure 3.8(a)), 5 (Figure 3.8(b)) and 10 (Figure 3.8(c)) Million bodies. With the smallest input size the application scales almost linearly up to 16 GPUs and stops scaling after 32 GPUs. Increasing the input size by a factor 2 increases the

	mc1	mc2	mc3
CPUs	2 x AMD Opteron(tm) 6168 @1.9GHz	2 x AMD Opteron(tm) 6168 @1.9GHz	2 x Intel(R) Xeon(R) X5650 @2.67GHz
GPUs	2 x ATI Radeon HD 5870	1 x NVIDIA GTX 480	1 x NVIDIA GTX 460
RAM	24 GB DDR3		
Interconn.	Infiniband QDR		
Open MPI	1.6.1		
OpenCL	AMD APP 2.6	CUDA 5.0	CUDA 5.0

TABLE 3.8: The architecture of `mc1`, `mc2` and `mc3` heterogeneous compute nodes.

execution time by a factor 4, due to the quadratic complexity of the N-body algorithm. With an input size of 5 and 10 million bodies the application becomes more suitable for a GPU cluster and with the biggest tested input size achieves a speedup of around 49 on 64 GPUs with an efficiency of 77%. It is worth mentioning that in such environment is important from a user prospective to find a trade-off between the number of devices and the desired efficiency. The results show that the hierarchical scheduling approach described in Algorithm 1 is able to handle multiple devices per node without compromising the overall scalability of the system.

3.4.5.3 Heterogeneous CPU/GPU cluster

Since OpenCL allows access to heterogeneous devices we conducted a second experiment which demonstrates *libWater* on a heterogeneous GPU cluster as described in Table 3.8. The application codes were rewritten in order to control the workload distribution via command line arguments. It is worth mentioning that workload partitioning for heterogeneous architectures is an active research problem [68–71]. However, this aspect is orthogonal to our library and for the sake of this experiment, we derive workload partitionings in an empirical way.

We ran the `MatrixMul` and the `Floyd` test cases using different combinations of devices. For each device configuration, several different workload splittings were tested and the fastest one was chosen. The partitionings and their corresponding execution times, are shown in Table 3.9. For example, in `MatrixMul`, configuration `C1` assigns all the workload to the first GPU of node `mc1`. The execution time for this configuration is 63.3 seconds. By equally splitting the workload between the two accelerators on the same node, i.e., `C2`, we double the performance. Between the GPUs, the NVidia GTX 480 is the fastest device requiring only 29.4 seconds to complete the work. However *libWater* can be used to improve the execution time even further. The overall execution time can be reduced by

	Device	Workload Partition Configurations							
		C1	C2	C3	C4	C5	C6	C7	C8
MatrixMul	mc1-GPU1	100%	50%	-	-	35%	25%	-	22%
	mc1-GPU2	-	50%	-	-	-	25%	-	22%
	mc2-GPU3	-	-	100%	-	65%	50%	75%	44%
	mc3-GPU4	-	-	-	100%	-	-	25%	12%
	Exec. time (in secs.)	63.3	32.5	29.4	68.4	23.7	19.0	26.6	**17.3**
Floyd	mc1-GPU1	100%	50%	-	-	2%	1%		0.5%
	mc1-GPU2	-	50%	-	-	-	1%	-	0.5%
	mc2-GPU3	-	-	100%	-	98%	98%	99%	98%
	mc3-GPU4	-	-	-	100%	-	-	1%	1%
	Exec. time (in secs.)	101.6	51.3	14.9	58.3	17.3	16.0	**13.1**	16.3

TABLE 3.9: Performance of `MatrixMul` and `Floyd` on the heterogeneous cluster for different combination of GPUs.

50% by using the workload partition as described by configuration `C8` which assigns 22% to each GPU in `mc1`, 44% to the NVidia GTX 480 and the remaining 12% to the NVidia GTX 460 accelerator. For the `Floyd` kernel, the results are different. Its execution on the GTX 480 is 8 times faster than the AMD GPU and 4 times better than the GTX 460. However, performance can still be improved by splitting the workload between the two NVidia accelerators by assigning 99% of the work to the faster GTX 480 and 1% to the GTX 460. This experiment demonstrates, despite higher latencies caused by additional data transfers between host and device memory, non-blocking communication yields good scalability behaviour even for heterogeneous architectures. However, scalability on such environments depends on several factors and we plan to investigate this issues in future work.

3.5 Summary

In this chapter we proposed several approaches to simplify distributed memory programming and improve productivity. Proposed ideas span from an advanced lightweight C++ interface to MPI (Section 3.2), to the use of meta-programming techniques (Section 3.3) and a powerful distributed runtime system (Section 3.4) which automatically generates communication statements in a way which is transparent to the programmer.

In HPC, experienced distributed memory programmers often prefer to retain full control over communication generation and placement. Our proposed MPP interface has been successfully received by the MPI community and immediately adopted in an *experimental* way by the Gromacs particle simulator for molecular dynamics [72].

However we also believe that programming models and systems that hide the complexity of message passing semantics are needed. On the one hand simplified programming models enable fast prototyping of new ideas and applications. On the other hand, they allow unexperienced programmers to easily address small to medium cluster systems with very little effort. At last, providing an higher abstraction level (also through the use of Domain Specific Languages (DSLs)) can have the effect of improving the adoption of the message passing paradigm for parallel programming since it is usually perceived as *difficult* (with an high entry barrier) and often ditched by newcomers in favor of shared memory parallel languages and paradigms such as OpenMP and pthreads.

Chapter 4

Runtime Parameter Tuning of Message Passing Programs

Support for the message passing programming model is provided either at the programming language level (e.g., Erlang [73] and Scala [74]) or through third-party libraries (e.g., MPI [3]). In both cases, a runtime system is included which realizes the abstractions, such as channels and communication contexts, necessary for two or more processes to communicate. These runtime systems are usually complex and highly optimized software layers designed to be portable on many architectures; to this end, they expose a high level of customizability through the use of a set of parameters to suite various hardware configurations.

In this chapter of the thesis we focus on the runtime parameter tuning of one of the most widely used implementations of the MPI standard, i.e., Open MPI. We analyze how parameter settings can negatively and positively influence the performance of an application and that optimal parameter values may depend on many factors: such as the application code, the input data and the target cluster configuration. Once analyzed these parameters, we derive methods delivering near-optimal parameter setting for a single or a class of applications. We use several different strategies to accomplish this: evolutionary, machine learning and statistical methods designed to address different use-case scenarios.

4.1 Introduction

In HPC, MPI is the de-facto standard for programming distributed memory systems. Because of the wide range of hardware configurations that MPI implementations have to address, MPI libraries allow customization through user-configurable parameters to better fit the characteristics of the underlying cluster architecture (e.g., cache size, type and topology of interconnections). Tuning these parameters is, however, a time-consuming task and requires detailed knowledge of the underlying architecture (i.e., interconnection and node architecture). For production clusters (e.g., BlueGene, RoadRunner), parameter tuning is often performed by the vendor itself. In other cases, parameter values are manually set to achieve a performance trade-off across a set of micro-benchmarks. However, as more and more small- to mid-size in-house commodity clusters with off-the-shelf components are being used by application groups and universities, parameter tuning is slowly becoming a critical issue. As no standardized methods exist, the default settings provided by MPI libraries are often employed, which can lead to poor performance.

Generally, we can distinguish two kinds of parameters:

1. *Compile-time* parameters, which are set during compilation of the MPI library, They are generally used to enable/disable features such as the level of support should be provided for multi-threaded execution (e.g., in OpenMPI the `MPI_THREAD_MULTIPLE` thread level, which allows multiple threads to invoke MPI routines without worrying about possible race conditions) or RDMA support (see Section 3.3.2). In general these parameters are more oriented towards functionality rather than performance.

2. *Run-time* parameters, they are used to adapt an instantiation of the MPI environment to better fit the characteristics of a target system, e.g., size of the internal buffers or processor affinity binding. Parameters of this class are generally easier to tune by the end-user (no particular administration privileges are required). This thesis focuses on the setting of runtime parameters.

From the most widely used MPI library implementations, MVAPICH [75] allows users to tune runtime parameters through environment variables, while Open MPI offers command line options to the Open MPI's Modular Component Architecture (MCA) [76], which will be presented in Section 4.2. Most of the distributed software developers are unaware of these parameters. However, as shown in Section 4.3, the impact of these parameters on the execution time of a program can be substantial, e.g., a reduction of

the execution time by half. Moreover, libraries usually expose several dozens of tunable runtime parameters making the optimization space infeasible to be exhaustively explored.

Besides the existence of basic guidelines on how and when to tune particular parameters [76], only one work in literature has been proposed to address automatic runtime parameter tuning. This tool is called the Open Tool for Parameter Optimization (OPTO) [11]. It determines the parameter combination that optimizes the execution of a program on a specific target machine. OPTO is an Iterative Feedback-driven Tuning (IFT) tool which, based on heuristics, tries to find "near-optimal" parameter values through several hundreds of program executions. The same approach has been employed for compilers, also known as Iterative Compilation (IC) [77], to find a "good" sequence and values of compilation flags optimizing the execution time (or the power consumption) of sequential programs for a target processor architecture. In IC, the compiler iteratively explores the optimization space defined by the input program by applying transformations and evaluating the resulting binaries on a target system. The main drawback of the IC and IFT-based techniques is the dramatic increase in compilation or optimization time. A large number of program execution (50 to 200) is usually needed to find a good transformation sequence or parameter combination. Over the years, IFT-based techniques have been improved in several ways, either by introducing better algorithms to reduce the number of program executions [13], or by using ML techniques which, by means of a training phase that learns the behavior of the machine to focus the search towards the optimal setting [78]. The main disadvantage of the presented techniques is, nevertheless, the additional executions of the input program required. While paying this cost may still be acceptable for compile-time or off-line training, it becomes less convenient for runtime parameters, as the performance gain resulting from tuned settings may not justify additional program runs. Furthermore, these techniques are sensitive to the input data (e.g., input data size, number of MPI processes), often leading to solutions which are optimized for a specific input data set only.

The goal of this chapter is to propose three methods to determine the MPI runtime parameter values that significantly improve the performance of an application on a target cluster. In other words, we aim at customizing the value of the default setting of any MPI library for a specific target architecture *fully automatically*, without any knowledge of the underlying hardware.

- The first method, presented in Section 4.5, considers evolutionary techniques [79] to explore the large optimization space generated by the available runtime parameters. We show that with this technique it is possible to converge to a near-optimal

solution using a smaller number of evaluations compared to OPTO, which involves simple heuristics to guide the search process.

- The second method, in Section 4.6, is based on ML techniques. The underlying idea is to gather knowledge of a target system by running kernel codes with varying settings of the runtime parameters. During a *training phase*, data of the executed configurations and relative performance is gathered. From that data a model is extrapolated, or *trained*, using ML algorithms. The optimized parameter setting for a new input program is determined by querying the model once.

- Our last approach for runtime parameter tuning is presented in Section 4.7. While the previous methods focus on delivering parameter settings which optimize a given application code and architecture; this method aims at finding a *trade-off* across application kernel codes characterizing a particular workload (e.g. HPC) which improves the execution of a *class* of programs based on a similar workload. To do that, we rely on statistical analysis, in particular we use the ANOVA [80] to (*i*) find the set of parameters with meaningful impact on the execution time, and (*ii*) to determine the values which, on average, improve the program execution time with respect to default settings.

4.2 The Modular Component Architecture

In this thesis, we focus on performance tuning of the runtime parameters provided by Open MPI's MCA [76]. MCA consists of a set of *frameworks*, *components*, and *modules* which are assembled at runtime to create an MPI implementation. A framework is dedicated to a specific task such as providing data transfer primitives for a particular network interconnect (i.e., Byte Transfer Layer (BTL)) or MPI collective operations (i.e., `COLL`). An MCA component is the specific implementation of a framework interface. Typically, the same framework can have multiple implementations, e.g., BTL includes support for Transmission Control Protocol (TCP), Infiniband (`OpenIB`), shared memory (`SM`), and others. Each module defines a set of runtime parameters whose values can be specified when an MPI application is started via the `mpirun` command, therefore, no *dynamic* tuning is possible.

The current development version of Open MPI has several hundreds of MCA runtime parameters. Features of the runtime environment such as processor and memory affinity can be enabled or disabled by using specific parameters. Other parameters refer to internal buffers (e.g., TCP send/receive buffer) which are allocated by the Open MPI library as specified by the user provided parameter values. Lastly, the behavior of

the runtime environment can be customized by means of threshold values. A good example is the opportunity to change the semantics of the *send* operation in relation to the transmitted message size (i.e., the *eager limit*). For small messages, the data is directly transmitted to the receiver without any acknowledgement from a matching *recv* operation (also called *eager* send). When the message size exceeds a threshold value, a different protocol is utilized such that an acknowledgement from a matching receive is required in order for the send operation to complete (*rendezvous*). A default setting for these parameters is provided by the Open MPI library. In the rest of the chapter we refer to this as the *default setting*.

Adding buffering to communication routines is a common practice to improve the latency of the communication routines. The MPI standard itself does not acknowledge the existence of these runtime parameters. This means that there is no standardization of their names and semantics. However, since they are widely used in all major implementations of the MPI standard (e.g., MPICH2, Open MPI, MVAPICH) similarities can be detected. Optimization techniques used within the libraries are often similar, leading to parameters which differ in the name but similar semantics (e.g., `MP_EAGER_LIMIT` in MPICH2 is semantically equivalent to `sm_eager_limit` in Open MPI). Therefore, the findings of this thesis, which are based on the parameters offered by the Open MPI platform, can be extended to other major MPI implementations.

4.3 Motivation

In this section we analyze the impact of MPI runtime parameter tuning on the execution time of parallel codes. We examined codes from the NPB suite [81] and studied their performance behaviour on three different cluster systems.

4.3.1 Experimental Setup

We considered the kernel codes from the NPB suite version 3.3 which consists of three benchmarks (`BT, LU` and `SP`) and five kernels `CG`, `EP`, `FT`, `IS`, and `MG`. We chose these kernels due to their importance for performance sensitivity for point-to-point and collective communication patterns [82].

We used, in our evaluation, the three SMP clusters summarized in Table 4.1. For each cluster, we considered two node sizes and evaluated each experiment for a small and a large input data size. The following scheduling policy, or mapping function, was used that allocates an MPI process $p_i \in \mathcal{P}$ (see Definition 10) to a computing unit $cu_j \in \mathcal{CU}$

System Name	**LEO 2**	**IBM Blade**	**Karwendel**
# of compute nodes	8-32	2-4	2-8
Chips per node	2	2	4
cus per chip	4	4	2
Core Architecture	Harpertown L5420	Nehalem X5570	Opteron 880
Clock Frequency	2.5GHz	2.93GHz	2.4GHz
shared L2/L3 cache	4MB (shared by 2)	8MB (shared by 4)	2x1MB (not shared)
Symmetric Multi-Threading (SMT)	–	2-fold	–
Memory per Node	32GB DDR2	32GB DDR3	16GB DDR2
Interconnection	Infiniband x8 QDR	Infiniband x8 QDR	Infiniband x4 SDR
Operative Systema	CentOS 5.3	CentoOS 5.4	CentOS 4.7
Kernel Version	2.6.18	2.6.18	2.6.9
Open MPI version	1.2.6	1.4.2	1.3.3
Compiler	GCC 4.1.2	GCC 4.4.3	GCC 4.3.3
Optimization Flags	-O3	-O3	-O3

TABLE 4.1: Experimental target architectures.

(see Definition 1):

$$p_i \longrightarrow cu_j \,|\, i = j$$

Note that $|\mathcal{P}| = |\mathcal{CU}|$ represents the number of processes being used to execute a program, we denote this quantity as the *communicator size*. We assume always to run as many processes as computing units available (no *over-subscription* of nodes). Given a compute unit, cu_i, its physical position within the topology is determined by a tuple (see Definition 9) determined as follows:

$$\mathsf{map} : cu_i \rightarrow (\frac{i}{cus\_per\_node}, \frac{i \ mod \ cus\_per\_node}{cus\_per\_chip}, i \ mod \ cus\_per\_chip)$$

We used the IBM Blade with 2 and 4 nodes (i.e., 32 respectively 64 MPI processes), the Karwendel cluster with 2 and 8 nodes (i.e., respectively 16 and 64 processes) and LEO2 with 8 and 32 nodes (i.e., 64 respectively 256 MPI processes). Although the node configurations of the parallel computers are significantly different, the type of interconnect and the SMP nature of the computing nodes are identical. Each cluster supports Open MPI for writing parallel algorithms configured to use the Infiniband network for inter-node communication and shared memory for intra-node communication. Therefore, we focused on the tuning of the Open MPI runtime environment on the runtime parameters of the `OpenIB` and `SM` modules (see Section 4.2).

The CPU's cores of the IBM cluster are capable of handling multiple flows of executions, or *threads*. In particular, the Nehalem core architecture can run two *logical* flows in

parallel per each physical computing unit (see Definition 1). Therefore a quad-core CPU with 2-fold SMT can handle 8 distinct execution flows. In our experiments these additional *logical* flows are considered as normal *cu*s enabling each node of the IBM cluster capable of running 16 MPI processes.

4.3.2 Performance-Oriented Runtime Parameters

Based on our experimental platforms, we selected 27 MPI runtime parameters with a clear performance-oriented semantics. This list was obtained from online documentation [76] and related work on the topic of Open MPI runtime parameters tuning [11]. They are listed in Table 4.2. A description is provided in Appendix B extracted from the documentation provided by the MCA framework [76]. The default parameter values given by the Open MPI library is shown on the second column of the Table 4.2. Some of the parameters can be assigned with a predefined number of values, for instance `mpi_yield_when_idle` can be either 0 or 1. Others (e.g., `btl_sm_eager_limit`) can assume an arbitrary value range, only limited by the available computer resources (e.g., memory size and number of Infiniband links). For the latter we considered only a subset of the possible values as specified by the *value range* column of the Table 4.2. The stride operator $x : y : *z$ is used for compactness, representing a geometric progression [83] of exponentially growing values, i.e.,

$$\left[x, x \cdot \left(z^1\right), x \cdot \left(z^2\right), x \cdot \left(z^3\right), \ldots, y\right]$$

Programs from the NPBs are executed by providing a value to each of those parameters. If a value is not provided, the default setting will be enforced by the library. The vector which assigns a value to each of the parameters is also referred to as a *configuration*.

Definition 38 – *configuration*

Let us define a *configuration* (C) that consists of a tuple of n runtime parameters $(p_1, p_2, \ldots, p_n)$, where $p_i \in \mathbb{P}arams$. A *configuration instance*, c, is a value setting where every p_i gets a value assigned, such as $c := (v_1, v_2, \ldots, v_n)$ where $v_j \in \mathbb{V}alues(p_j)$ is a value (a buffer/threshold size or a flag value) associated with the parameter p_j. Throughout this chapter we use the term *parameter setting* as a synonym for configuration instance.

Parameter name	*Default*	*Value range*
`mpi_yield_when_idle`	0	$0, 1$
`mpi_paffinity_alone`	0	$0, 1$
`mpi_preconnect_mpi`	0	$0, 1$
`mpi_leave_pinned`	0	$0, 1$
`COLL`: Collective operation tuning (`coll_*`)		
`sm_tree_degree`	4	$2 : 8 : *2$
`sm_control_size`	4096	$512 : 256K : *2$
`sm_fragment_size`	8192	$512 : 256K : *2$
`sync_barrier_after`	0	$0, 500, 1K, 5K, 10K$
`sync_barrier_before`	1000	$0, 500, 1K, 5K, 10K$
`tuned_init_tree_fanout`	4	$4 : 16 : *2$
`tuned_init_chain_fanout`	4	$2 : 16 : *2$
`SM`: Shared memory communication tuning (`btl_sm_*`)		
`eager_limit`	4096	$512 : 256K : *2$
`max_send_size`	32768	$512 : 256K : *2$
`rndv_eager_limit`	4096	$512 : 256K : *2$
`fifo_size`	4096	$512 : 256K : *2$
`num_fifos`	1	$2 : 16$
`OpenIB`: InfiniBand communication tuning (`btl_openib_*`)		
`eager_limit`	12288	$512 : 256K : *2$
`max_send_size`	65536	$512 : 256K : *2$
`rndv_eager_limit`	12288	$4K : 256K : *2$
`use_message_coalescing`	1	$0, 1$
`user_eager_rdma`	0	$0, 1$
`eager_rdma_num`	16	$2 : 64 : *2$
`use_async_event_thread`	1	$0, 1$
`ib_max_rdma_dst_ops`	4	$1 : 8$
`rdma_pipeline_send_lenght`	1048576	$4K : 2M : *2$
`rdma_pipeline_frag_size`	1048576	$4K : 2M : *2$
`min_rdma_pipeline_size`	262144	$4K : 512K : *2$

TABLE 4.2: List of 27 performance runtime parameters meaningful for our target architectures.

4.3.3 Random Space Exploration

Finding the optimal parameter setting by exhaustive exploring of the optimization space is an exponential problem that can require millions of runs for every input program, rendering it impossible. A common solution is to employ randomized algorithms (e.g., simple, simulated annealing) for finding approximate solutions as close as possible to the optimal one.

In this section we randomly explore the optimization space for each of the five computational kernels by evaluating 1000 distinct configuration instances on each target clusters. We executed the same configuration instances for different input data and communicator

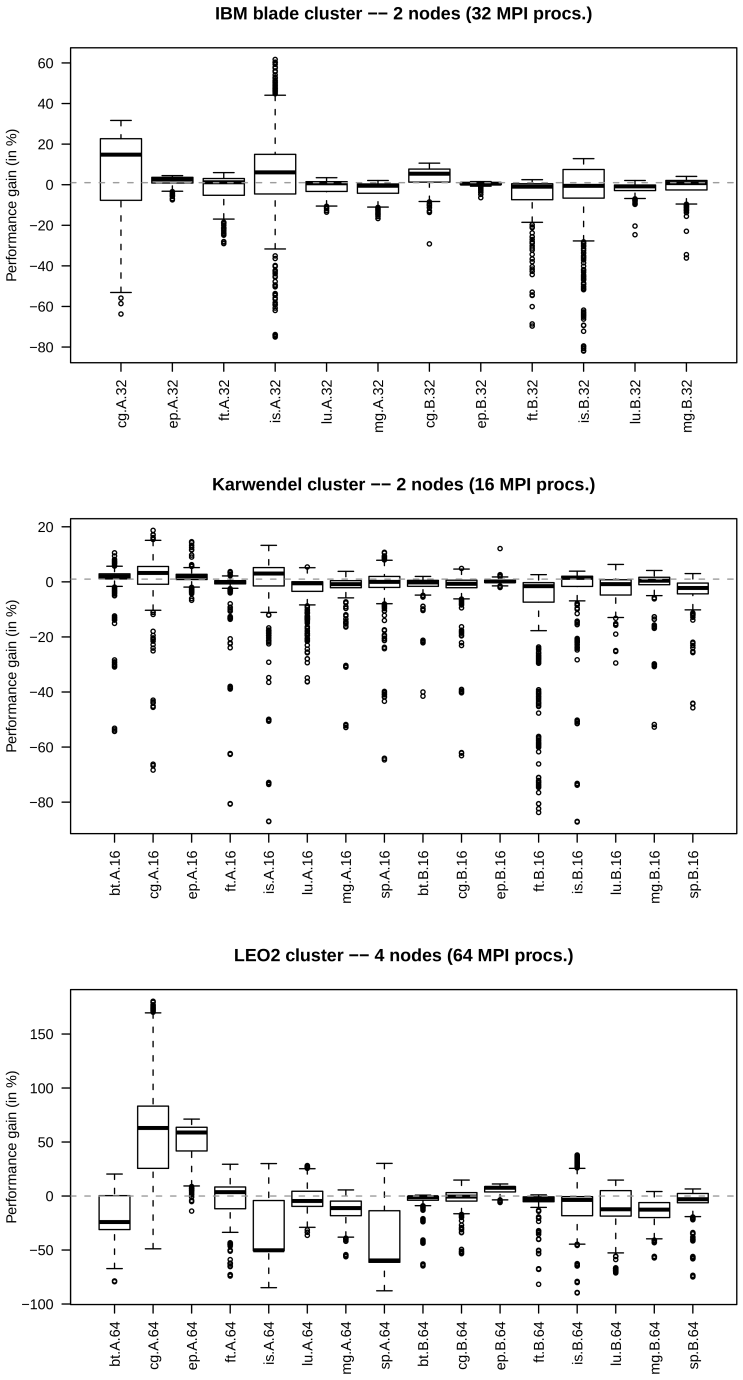


FIGURE 4.1: Performance variance with respect to Open MPI's default setting registered for 1000 parameter configurations when used to run the NPBs on our target architectures with the small node setting.

sizes that yielded a total of 20000 runs on each cluster. We computed the performance gain by comparing each experimental run against the one using the default Open MPI parameter setting. The bar plots in Figures 4.1 and 4.2 depict the variance of the performance gain by running each benchmark with different parameter settings. The points inside the boxes are within the lower and upper quartile (the median is represented by

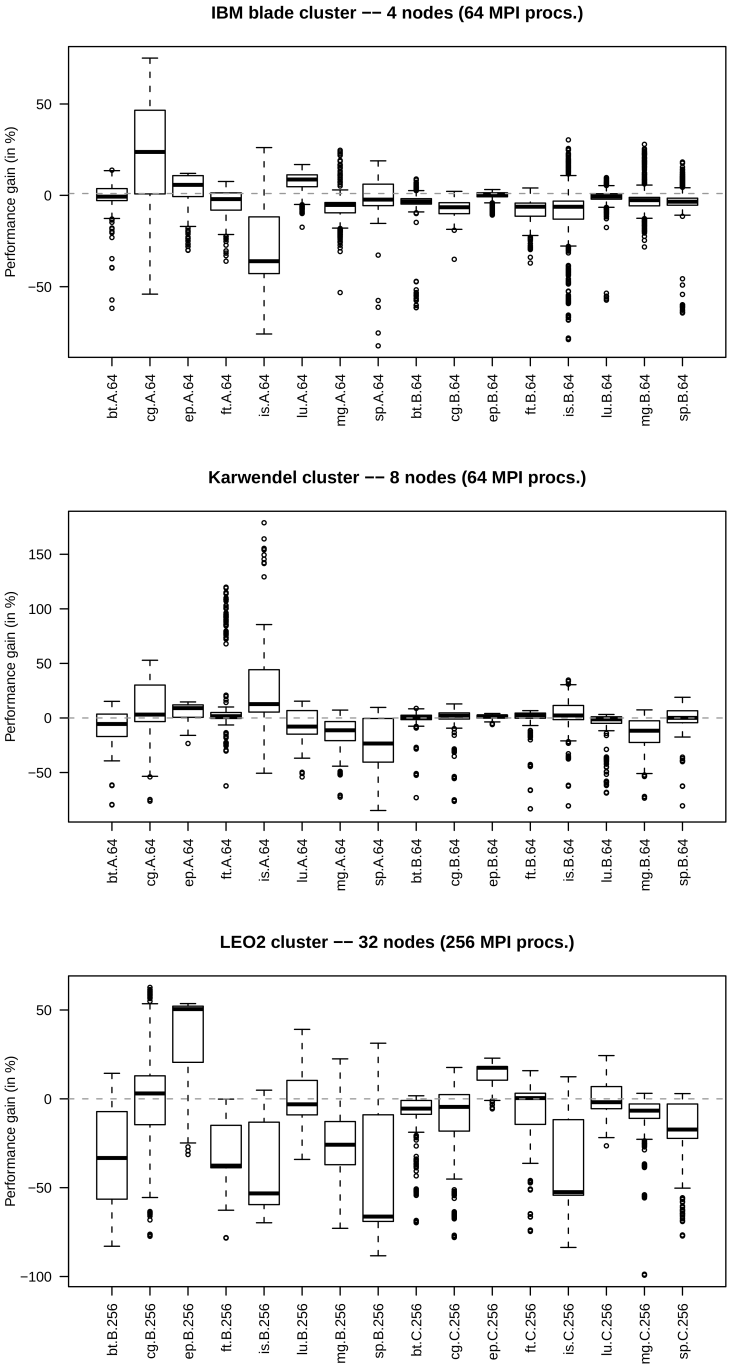


FIGURE 4.2: Performance variance with respect to Open MPI's default setting registered for 1000 parameter configurations when used to run the NPBs on our target architectures with the large node setting.

the bar inside each box), ranges outside the box represent the minimum and maximum samples, while the single points outside the ranges are the outliers.

First, we can observe that, on average, the Open MPI's default settings are a fairly good choice for most benchmarks. In fact, 80% of the tested configuration instances have an increased execution time with respect to the default settings. For example,

IS has a sensible performance slowdown on each target architecture, but achieves a speedup (with respect to the default settings) of around 10% by parameter tuning. Interesting performance improvements are also visible for other kernel codes such as CG and EP on every architecture. On average, the performance gained by the "best" random configuration compared to the default setting is of approximately 26% on LEO2, 15% on the IBM Blade and 17% for the Karwendel cluster. This means that with the "best" parameter setting, found by the random sampling, a program executes 15 to 30% faster with respect to the default values. In the rest of the chapter, we refer to the configuration instance delivering the fastest, measured, execution time for a particular benchmark and target architecture as its OPUB. Important to note is that this is not the optimum c, as we only explored a subset of the optimizations space. However, since the random probing was designed to cover the optimization space homogeneously, we expect the global optima not to be far too from the OPUB. Throughout the chapter, the word "best" is used as a synonym for OPUB.

This exploration of the optimization space also highlighted several properties of the OPUB. For example, a configuration instance which is shown to be the "best" choice for one kernel may be suboptimal for another. The "best" parameter values could also change for different input data or communicator sizes. Figure 4.3 highlights this aspect. We selected four kernels FT, EP, IS and MG, and their relative OPUB configurations resulting from the random exploration on the LEO2 cluster. We used the data from LEO2 since the larger number of nodes results in higher performance gains, and thus, the aforementioned performance behaviour is better visible. Afterwards, we used the OPUB configuration instance of each kernel to run other kernels, measured the execution time and compared the results. Each of these configuration instances are represented as a bar in the graph in Figure 4.3 (in total we have 4 distinct instances. i.e., $i = 1, \ldots, 4$). Each benchmark B_i (on the X-axis) has been executed with configurations $c_j : \forall j \in \{1, \ldots, 4\}$. We can observe that the "best" setting for a specific kernel can result in a significant performance loss when used to run other applications. For example, FT always exhibits a performance loss when executed with configuration instances from other benchmarks. Executing FT with its own OPUB setting yields, however, a performance improvement of around 2%. On the other hand, using the "best" configuration instance for FT to run other benchmarks yields a suboptimal improvement or even a slowdown for MG. The other kernels show similar behavior.

These experiments demonstrate the complexity that system administrators are facing during parameter tuning of MPI libraries for different architectures and the performance gain which is achievable by fine tuning of these parameters. Eventually, administrators manage to reach a satisfiable parameter setting after a tedious and time consuming

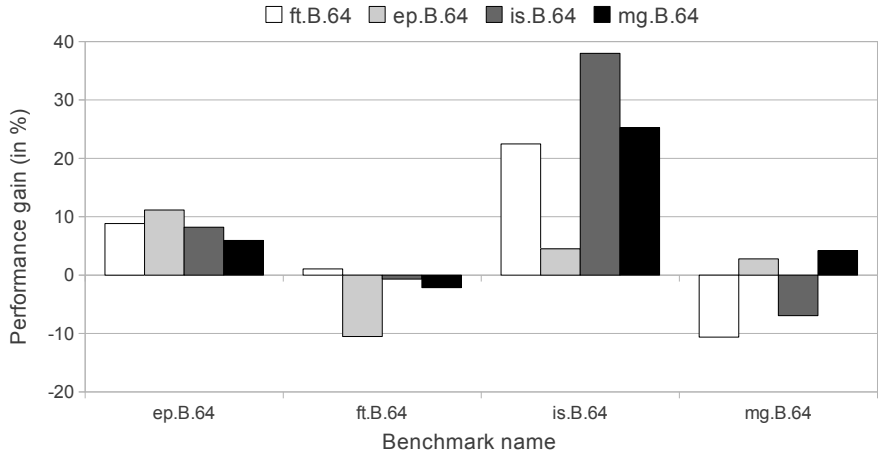


FIGURE 4.3: Performance gain when using the "best" configuration of an NPB on LEO2 for running the other three NPBs.

manual tuning, a process which is likely to be driven by *chance*. Our goal is to propose automatic tools which automatically deliver near-optimal parameter values.

4.4 Related Work

To the best of our knowledge, only few works related to the optimization of MPI runtime parameter settings exist in literature. This problem is similar to other fields, such as the optimization of compiler flags. We can divide the approaches in literature into three separate categories:

Search-based: In this group we consider all techniques which employ heuristics to search the optimization space generated by the large combination of parameter values. All of the IFT-based methods are included in this category.

ML-based: It represents an improvement over the previous group; techniques based on ML uses previous knowledge (gathered during the training phase) to narrow the search space.

Statistical-based: This class of algorithms considers techniques which are not limited to a single application. The goal is to find a configuration instance which suites the underlying architecture independently on the application code.

OPTO can be classified among the search-based approaches [11]. OPTO systematically tests large numbers of combinations of the Open MPI's runtime parameters *for common communication patterns* and *performance metrics* to determine the "best" set for

a specific benchmark under a given platform. OPTO is based on a library incorporating various search algorithms, namely the Abstract Data and Communication Library (ADCL). ADCL evaluates the performance of some (or each) configurations of the provided runtime parameters, and by using heuristics, chooses the version leading to the lowest execution time.

In [12], Cooper at al. used a *genetic algorithm* to find compiler optimization sequences that generate small object codes. Instead of finding parameter values, the algorithm proposed in [12] was designed to determine the compiler optimizations and the order which delivers smaller object code for a given application. Experiments showed that for most of the codes, the genetic search was able to outperform (both in terms of generated object code size and performance) the outcome of a fixed optimization sequence (similar to what it is offered by the `-O3` flag of a compiler). Moreover, compared to random probing, the genetic algorithm converged faster to quality solutions.

Search techniques have also been employed within libraries to adapt the implementation of an algorithm to the underlying architecture. Relevant examples are the Automatic Tuned Linear Algebra Software (ATLAS) [84] and the Fastest Fourier Transform in the West (FFTW) [85]. The idea underlying the architecture of ATLAS and FFTW is similar. The libraries include several implementations (or versions) of the same algorithm and the "best" version is chosen at runtime by an optimizer across several invocation of the routine. In order to do this efficaciously, the *planner* uses dynamic programming techniques to prune the search space. This paradigm is also called Automated Empirical Optimization of Software (AEOS).

ML techniques have been also used in MPI programs for optimizing collective operations [86]. The bottom line is that collective operations can be implemented by using several algorithms (e.g., *2D/3D mesh*, *recursive doubling*, *bruck*, *ring* and *pair*) and depending on the network topology, communicator and message size, some implementations perform better than others. The problem of selecting the algorithm for a specific collective operations which is optimal for the communicator size, topology and message size is addressed by using decision trees. Off-line training builds a tree which selects the optimal algorithm corresponding to a particular collective routine, communicator and message size. At runtime, whenever a collective operation is called, the model is queried with the current message and communicator size. The query yields the best algorithm to be employed.

In the context of statistical based methods, several work exist related to the optimization of compiler flags. These techniques have never been applied to MPI's runtime parameters, nevertheless the problem they aim to solve is similar. In [13], Eigenmann proposed an algorithm, called Combined Elimination (CE); CE is an iterative algorithm,

at each step it recognizes the compiler flags with negative effects on performance and turns them off. The algorithm has been proved to converge to an optimal configuration quickly but still several dozens of iterations, and thus evaluations of input programs, are required. In [14], statistical techniques have been employed to improve iterative compilation. Also this approach deals with compiler flags and thus it assumes binary variables (i.e., on/off). The *full factorial design* of an experiment with k parameters (or flags) is 2^k combinations, in order to reduce the number of configuration a *fractional* factorial design is defined where orthogonal arrays (OA) are considered. The algorithm proposed is iterative, it first identifies options with a large overall effect and switch them to on; then it looks at the remaining options to see what improvement they can produce given the partial setting already constructed.

The aforementioned approaches either cannot be directly applied to the MPI runtime parameter tuning problem (e.g., because only dealing with binary variables or they require too many executions to explore the optimization space). In the rest of this chapter we propose three novel approaches which focus on runtime parameter tuning for different use-case scenarios.

4.5 Auto-Tuning with Evolutionary Techniques

In this section we describe the first of the three methods proposed for tuning MPI runtime parameters. This method is similar to the technique used by the OPTO optimizer. The main idea is to *explore* the optimization space, for a given input code, generated by varying runtime parameter settings. In order to minimize the number of evaluations, the search is focused on those areas likely to give shortest execution time. We derived an algorithm based on *evolutionary techniques* which is tailored to the runtime parameter problem [79].

In general, an evolutionary search algorithm uses mechanisms inspired by biological evolution, such as *reproduction*, *mutation*, *recombination*, and *selection*. Candidate solutions to the optimization problem play the role of *individuals* in a population, and the *fitness function* determines the quality of the solutions. Individuals are composed by a set of *genomes* which represent the *atomic building blocks* which are recombined across generations to form new individuals or *offspring*. Evolution of the population takes place after the repeated application of the above operators. Evolutionary based search techniques are shown to be successful for very large and complex optimization spaces [87]. A drawback of any evolutionary algorithm is that a solution is "better" (in terms of fitness) only in comparison to other, presently known solutions; such an algorithm actually has no concept of an "optimal solution", or any way to test whether

a solution is optimal. This also means that an evolutionary algorithm never knows for certain when to stop.

In our context, an individual is a particular configuration instance of MPI runtime parameters. We define the recombination operator (or *crossover*) in a way such that good parameters are preserved across generations. This section continues describing how the generic evolutionary algorithm has been tailored to the MPI runtime parameter problem. The proposed algorithm has been evaluated on the three experimental platforms presented in Table 4.1.

4.5.1 Tournament Selection

Tournament selection is a method of selecting an individual from a population of individuals in an evolutionary algorithm. An evolutionary algorithm starts by generating an initial population of individuals (or configuration instances), E, of a determined size. Each individual i is composed of a variable number of *genomes* randomly generated. In our context, a genome g is defined by the following structure:

$$i := (g_0, \ldots, g_n) \in E, \textit{where}$$
$$g := (p, v_p) \,|\, p \in \mathbb{P}\textit{arams} \wedge v_p \in \mathbb{V}\textit{alues}(p)$$

Genome g is composed of the MPI parameter name p, chosen within the set of parameters $\mathbb{P}arams$ depicted in Table 4.2; and a value, in $\mathbb{V}alues(p)$, which is in the set of p's possible values (column *value range* of Table 4.2). Given an individual of the population, a parameter setting is generated by using the parameter values v_j in the correct order (dictated by the configuration).

Once the initial population is created, the *tournament selection* strategy is applied [88]. In each *generation*, a subset of size t individuals, with $t << G$, is randomly selected and their *fitness* is evaluated. In this context, the fitness is given by the *performance gain* (with respect to the Open MPI's default parameter setting) of a program executed with the parameter values contained in the selected individual genomes. The two individuals with best fitness, i.e., higher performance gain, inside the tournament group are chosen to create *two* offspring by applying a *crossover* operator (see later). To escape local performance maxima, a mutation operator – selected with low probability ($\leq 0.2\%$) – is also applied to the newly created offspring. A tournament completes with the replacement of the elements with lowest fitness with the newly generated offspring.

The algorithm iterates until a maximum number of generations has been reached, or no improvement of the overall fitness is contributed by the new offspring for at least 10 generations. Tournament selection has been chosen because of its properties. It is efficient, with a computational complexity of $O(n)$, and it allows the *selection pressure* to be easily adjusted by tuning the tournament size. On the one hand, a small tournament leads to a low selection pressure as the probability of a genome present in the tournament to be carried on to the new offspring is high. On the other hand, a larger tournament results in a higher selection pressure, which makes the algorithm more unstable. Due to the algorithm formulation, the genetic search ideally converges to individuals with high fitness value whose genomes are composed of only runtime parameters that are meaningful for the underlying architecture. Parameters with no performance improvements will be lost in the genetic pool of the population as poor performing individuals are always replaced with newly created offspring.

4.5.2 Crossover and Mutation Operators

Of extreme importance for a genetic algorithm to converge to optimal solutions is the definition of the *crossover* and *mutation* operations. The crossover operator takes the two configuration instancess with highest fitness in the current tournament ($\mathbb{T}urn$) and recombines their genomes to create new offspring. Crossover has to carry on those parameters which are the cause of the high fitness value of tournament winners. The genome structure of the offspring generated by a crossover operation (i.e., os) has been defined, for our context, as follows:

$$os := \{g_0, \ldots, g_i, \ldots, g_n\}, \textit{where}$$

$$g_j := (p_j, v_{p_j}) \begin{cases} 0 \leq j \leq i & \exists\ g_{par_0}, g_{par_1} \in \mathbb{T}urn : \\ & p_j \in g_{par_0} \wedge p_j \in g_{par_1} \Rightarrow v_{p_j} := rand(g_{par_0}[p_j], g_{par_1}[p_j]) \\ i < j \leq n & \exists\ g_{par_0}, g_{par_1} \in \mathbb{T}urn : \\ & (p_j, v_{p_j}) \in g_{par_0} \wedge p_j \notin g_{par_1} \vee (p_j, v_{p_j}) \in g_{par_1} \wedge p_j \notin g_{par_0} \end{cases}$$

In our implementation, those parameters which are present in both parent genomes (i.e., g_{par_0} and g_{par_1}) are always carried on to the children, i.e., $0 \leq j < i$, the parameter values are chosen (randomly) to be one of their parent values. The size of the new offspring n is chosen randomly, however, offspring are never smaller or larger than the parents. We allow variable size individuals so that runtime parameters which have been recognized not to have any effect on the execution time of an application are lost across

generations. The remaining genomes, i.e., $i \leq j < n$, are randomly chosen from both the parents amongst the ones which are not in common.

The *mutation* operator, applied with a probability of 0.2%, randomly changes parameter values, i.e., v_j, with a probability of 0.5%. Mutation is important in order to avoid local maxima.

4.5.3 Experimental Evaluation

The genetic search algorithm has been executed multiple times on each architecture of Table 4.1. The population and tournament sizes have been set empirically, they are respectively $E = 50$ and $t = 5$. The initial population is generated in a way that all of the 27 runtime parameters of Table 4.2 appear in at least one individual. Additionally, the tournament selection makes sure that individuals whose fitness has not been yet evaluated are selected for the next tournament. This guarantees that every individual fitness is evaluated once.

The graph in Figure 4.4 shows, for each generation, the fitness value (i.e., performance gain) of the best configuration in the population. For the LEO2 cluster, the genetic search, discovers configurations which outperform the upper bound previously found with the random exploration, i.e., the OPUB, with almost a tenth of the evaluations. Furthermore, the search reaches a maximum after approximatively 50 generations which, considering an average number of 3 evaluations per tournament, results in a total of 150 program evaluations. Also for the IBM Blade system and the Karwendel cluster, evolutionary search returns an average performance gain which is close to the OPUB value found with random exploration. A performance value which is within 80% of the OPUB can be obtained with 15 generations (i.e., around 45 evaluations).

Unfortunately we could not directly compare the result of our evolutionary tuning algorithm with OPTO since its code is not available. Comparing with their performance results presented in [11], our tool requires a similar amount of evaluations to OPTO. However, in our setup we considered a larger number of parameters, 27 versus 4, making the optimization space to explore much larger.

4.6 Auto-Tuning with Machine Learning

The genetic search approach presented above is best when a program with a specific input configuration instance (input data and communicator sizes) has to be tuned for a given target machine and the user can afford several executions of this program. Sometimes

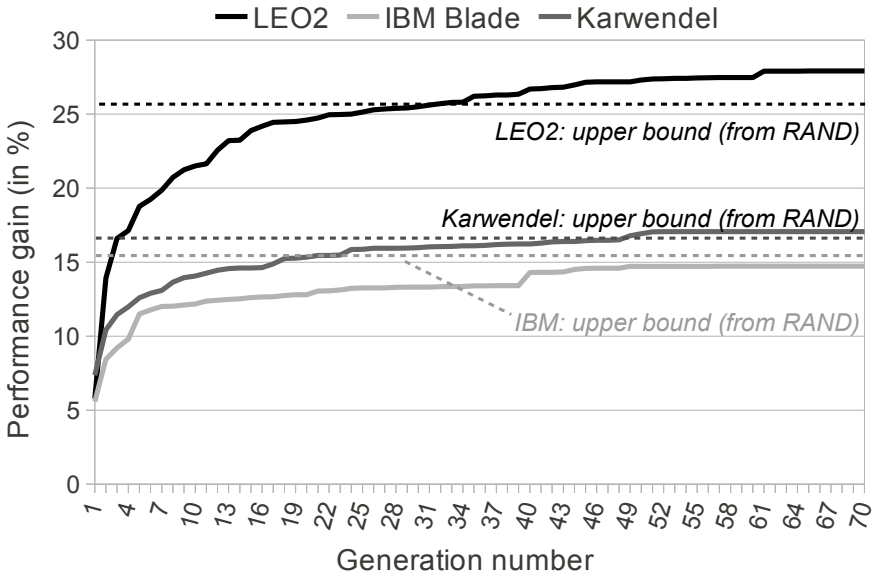


FIGURE 4.4: Average performance gains of the configurations with best fitness in the population related to a generation number.

this is not feasible since a single execution of the input code may be too long making the tuning process span over a long period of time.

In this section we propose a novel approach for MPI runtime parameter tuning based on ML. The main goal is to reduce the time needed for the estimation of the parameter setting which best suites a given application and the underlying target architecture. The main idea is to derive a *model* which delivers *near-optimal* parameter settings given an input program. Since deriving such a model based on an analytical method would require to take into account the complex interactions between the hardware architecture characteristics and the application code features; we use ML technique to *learn* a prediction model from data.

The basic assumption behind ML-based approaches is that similar codes have similar behaviour. Moreover, the runtime parameter setting derived for one code is likely to have also positive effect for a similar code. The efficacy of the method and its prediction accuracy depends on the way differences between two codes are measured. For this, metrics are used to characterize a message passing program. These *program features* can be both *static* and *dynamic*. Static features, e.g., number of point-to-point and collective communication statements, can be extracted by analyzing the input code. Whereas the extraction of dynamic features (e.g., average amount of data exchanged in communication) requires one profiled execution of the input code.

This section describes how the runtime parameter tuning problem can be expressed as a *predictive* modeling problem and introduces the ML-based framework by describing the *feature extraction*, *training* and *deployment* of the model.

4.6.1 Parameter Selection and Experimental Setup

During the training phase, the optimization space is explored exhaustively (i.e., all combinations of runtime parameter values are tested and measured). We used all 8 codes of the NPBs as the program training set. In order to make this exploration feasible, we reduced the number of parameters from 27 to the most meaningful 4. The selection has been done by computing the *Pearson correlation coefficient* [89] between the parameter values and the execution time from the data gathered during the random exploration described in Section 4.3. The four parameters with the absolute highest correlation value have been selected:

`sm_eager_limit:` eager threshold for communication done over shared memory (`SM`)

`btl_openib_eager_limit:` eager threshold for communication done over InfiniBand (`OpenIB`)

`mpi_paffinity_alone:` binds the MPI process rank to a physical *cu*

`mpi_yield_when_idle:` MPI processes frequently yield the CPU to its peers (*degraded mode* [76])

Also, because of time constraints, we limited our experiments to a single architecture, we chose the LEO2 cluster, see Table 4.1, since it provides the largest number of nodes.

4.6.2 The Prediction Model

The predictive modeling problem can be formally represented as follows. Let x be a vector of program features extracted from an MPI program p, s be the performance curve (speedup) of p and c be the runtime parameter setting with the "best" measured speedup. We want to build a model, f, which predicts the configuration instance, $\hat{c}$, i.e., $f(x) := \hat{c}$. The closer the speedup resulting from the execution of the program with the predicted runtime parameter setting (i.e., $\hat{c}$) is to s, the more accurate the model will be.

Figure 4.5 depicts an overview of the proposed ML framework for MPI runtime parameter tuning. It consists of two major phases: the (*i*) *training process* and the (*ii*) *model deployment*.

- During the training phase data is collected by running a set of training programs on the target machine against several configuration instances of a set of runtime parameters. The program features together with the current runtime parameter setting and the achieved speedup is stored in the training data *repository*. This

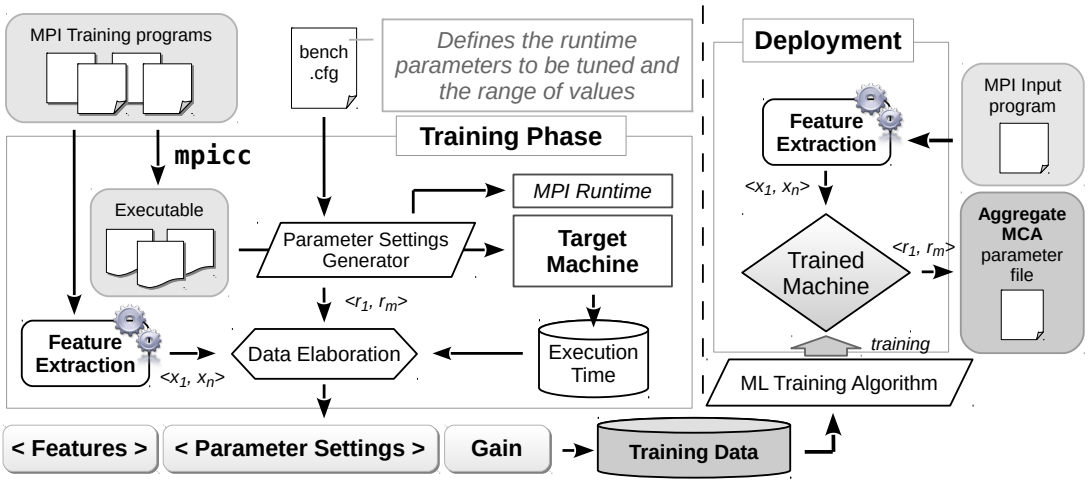


FIGURE 4.5: Overview of the ML-based method.

phase can be expensive, however it only needs to be executed once before the input code to be optimized is presented to the tool. For this reason it is also referred to as *off-line* training. Using *supervised learning* techniques [90] the predictive model is then built from the collected data.

- When the model is deployed, *on-line*, it is used to predict the optimal runtime parameter settings for *unseen* input programs presented to the tool. In this phase, the system automatically extracts the input program features and presents them to the trained machine (or model) which returns the predicted setting as an Aggregate MCA (AMCA) parameter file [76].

In the following section we describe the feature extraction phase and an overview of the ML algorithms used to build the predictor.

4.6.3 Feature Extraction

Figure 4.6 depicts a classification of the 19 features herein employed to represent MPI programs. Both *point-to-point* and *collective* communication patterns are described. The point-to-point communication graph is described by the average number of receivers, for each process in the communicator, and the average distance between the source and the destination of a message. These metrics are weighted according to the number of messages exchanged. In this way, patterns which are more frequently used within the code are recognized to be more relevant. Additionally the average point-to-point message size in bytes is stored. Collective communications are divided into three different categories: *one-to-n*, *n-to-one* or *n-to-n*. Routines which are included in the *one-to-n* category are for example `MPI_Bcast` and `MPI_Scatter`. `MPI_Reduce` and `MPI_Gather`

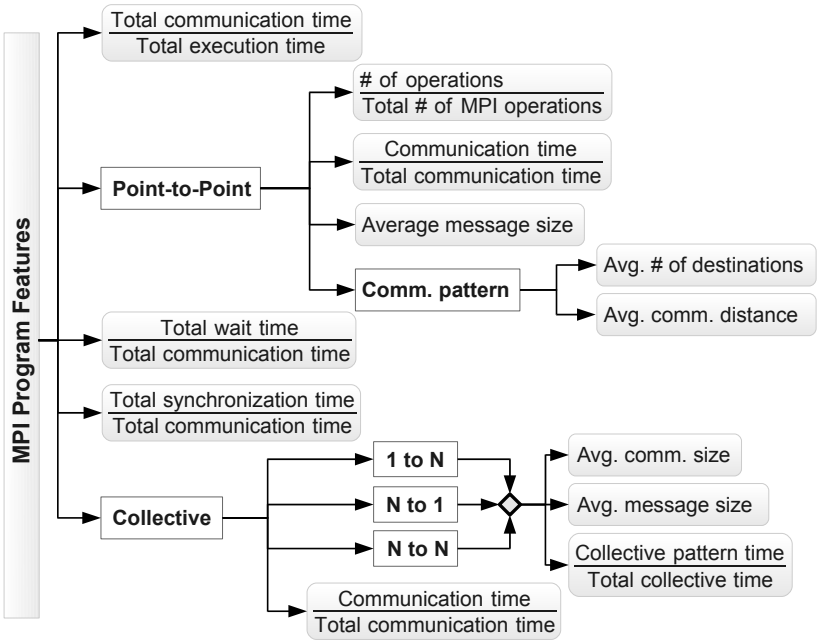


FIGURE 4.6: Classification of the 19 program features used to characterise MPI programs

are instead classified as *n-to-one* pattern. The most expensive collective pattern is represented by *n-to-n* and it refers to routines like `MPI_Alltoall` and `MPI_Allreduce`. For each of the patterns the average communicator and message size are measured. The remaining features are used to measure the communication-over-computation ratio, the wait and synchronisation time over the total communication time ratio and, to conclude, the ratio between the time spent in point-to-point and collective operations over the total communication time.

As most of the features depend on the input data and communicator size, the extraction is done dynamically. *Only one run* of the input program is needed in order to extract its feature set. MPI input programs are compiled and linked against a tracing library which, using the MPI profiling interface (PMPI) [3], writes information about every MPI call performed by each process of the program into trace files. For each operation a time-stamp value, the execution time and the amount of data sent by a process is stored. Specifically, the receiver is stored for point-to-point communications and the communicator size for collective operations. *Trace analysis* is then performed in order to extract features described in Figure 4.6. As each MPI process rank produces its own trace file, analysis is done in parallel.

4.6.4 Training Prediction Models with Machine Learning Techniques

Supervised learning techniques [90] allow the deduction of prediction models starting from training data by means of different algorithms. Commonly, supervised learning generates a model that maps input features into desired outputs. As shown in Figure 4.5 each entry, td, of the training data, TD, is a vector of n values structured as follows:

$$td := (x_1, \ldots, x_f, v_1, \ldots, v_p, Speedup)$$

Where $f = 19$ (number of the program features), $p = 4$ (number of runtime parameters) and therefore $n = 24$. Training data cannot be used in this form as the speedup is neither an input nor an output of the prediction model, i.e., $f(x) = \hat{c}$. Currently, the training set contains the speedup values obtained by running the training set programs against several runtime parameter settings. As we want to optimize the speedup, only the configuration instances with the highest performance gain must be filtered to be used by the learning algorithm.

$$TD_{best} := \{(x_i, c_{best}) \mid \Rightarrow speedup((x_i, c_{best})) = max(speedup((x_i, c_k)) \ \forall k)\}$$

In this work we compare the prediction accuracy of two ML algorithms: k-Nearest Neighbours [90] and Artificial Neural Networks [91].

4.6.4.1 k-Nearest Neighbours (k-NN)

The k-NN is amongst the simplest of all ML techniques and allows object classification based on closest training examples in the feature space. An object is classified by a majority vote of its neighbours, with the object being assigned to the class most common amongst its k nearest neighbours. This technique does not require the construction of a model as the classification of a new input can be done by determining its k closest – by means of a metric defined on the program features – neighbours in the training set. Despite its simplicity, the time used to classify a new input linearly grows with the size of the training set leading to slow prediction performance with very large training sets.

Euclidean distance has been used as a metric and the best prediction results have been obtained with $k = 3$. New input programs are classified by measuring the Euclidean distance between their features and the feature vectors contained in the training set. The three nearest neighbours are selected and for each parameter the predicted value is determined as follows:

$$R_{best} := \left(\frac{\sum v_1(i)}{k}, \ldots, \frac{\sum v_p(i)}{k} \right) \ \forall i = 1, \ldots, k$$

4.6.4.2 Artificial Neural Networks (ANNs)

The second ML algorithm which fits the formulated prediction model is represented by ANNs. ANNs have the ability to derive knowledge from complicated or imprecise data and can be used to extract patterns and detect trends. ANNs are particularly robust to noise and can be used to model both linear and non-linear classification and regression problems. In this thesis we use a type of neural network known as *feed-forward* Multi-Layer Perceptron (MLP). An MLP can be easily configured, in terms of input/output units and number of layers, in a way to meet the requirements of the prediction model, i.e., 19 inputs (program features) and 4 outputs (runtime parameter values). Supervised training with MLP is possible by means of the *back-propagation* algorithm. Unlike k-NN, ANNs require to be trained. Training time depends on the size of the training data. Once trained, ANNs are very fast in delivering the prediction with a deployment time which only depends on the number of layers.

A *four-layer feed-forward back-propagation network* is used, the ANN structure with the best performance for our problem is chosen as below. The number of units in the input/output layers is defined by the problem, thus 19 neurons in the input layer (equal to the number of program features) and 4 in the output layer (equal to the number of runtime parameters to estimate) are needed. The two hidden layers (H_1 and H_2) of the network contains, respectively, 16 and 10 units. The *hyperbolic tangent Sigmoid* transfer function has been used for H_1 and the output layer, the Sigmoid has been used for H_2. A *learning rate* in the range $[0.2, 0.3]$ and a *momentum* of 0.8 has been used for training. Configuration of the parameters of the ANN used in our prediction problem has been derived empirically by testing different setups and choosing the one with highest prediction accuracy.

4.6.5 Experimental Evaluation

In this section we evaluate the accuracy of the ML-based model for 8 of the NPBs (i.e., `BT`, `CG`, `EP`, `FT`, `IS`, `LU`, `MG`, `SP`). The experiments have been conducted on the LEO2 cluster.

The prediction accuracy has been measured using the Leave-One-Out Cross Validation (LOOCV) method [90]. When the runtime parameter values for benchmark x are predicted, the model is built by using the training data obtained by removing x from the training set. x's feature vectors – resulting from varying input data and communicator sizes – are then used to query the trained prediction model. The closer the execution time – measured by running x with the estimated runtime parameters – is to the upper

bound in performance, the more accurate the prediction is. The raw training data (TD) consists of ∼9500 points while $|TD_{best}| = 44$.

In Figure 4.7, the two prediction models are shown and their prediction accuracy compared with random selection. The random selection has been executed by randomly choosing a value for the four selected runtime parameters described in Section 4.6.1. The produced parameter setting was then used to run the input code. We repeated this process five times and computed the average performance improvement with respect to the Open MPI default parameter setting.

In Figure 4.7 the graph represents the percentage of the available speedup achieved by the estimated runtime parameter settings for random selection, k-NN and ANNs when the LOOCV method is used. The performance value is relative to the OPUB which, in this scenario, is represented by the execution time of best configuration instances contained in TD_{best}.

The prediction always achieves a performance improvement which is, on average, within 92%, for k-NN, and 96%, for ANNs, of the OPUB. This means that the configuration instances predicted by our ML-based model yields a performance improvement which is very close to the best possible achievable via runtime parameter tuning. For the LEO2 cluster this means a 20% performance improvement over the default setting provided by the Open MPI library. If compared to a random selection, the performance of the configuration instances predicted using ML is generally higher. In fact, by averaging the absolute performance gain value resulting from random parameter settings, the overall performance is worse than what is achieved using the Open MPI's default settings.

For benchmarks `BT`, `MG` and `SP`, the prediction is accurate as these applications are based on point-to-point communications with very similar feature vectors. For example the three benchmarks have an average point-to-point communicator size of 6 and the distance between communicators is 9.9; as a consequence, the configuration instances derived by the model are very similar. Degradation in the prediction accuracy is measured for benchmarks `EP` and `IS`. `EP`, for example, is an *embarrassingly parallel* application and it presents unique input features across the training set. Only few messages are exchanged among the processors and we measured that 50% of the overall communication time is spent in synchronisation barriers. As none of the other benchmarks in the training set presents a similar behaviour, the prediction accuracy is affected. Prediction accuracy can be improved by introducing new benchmarks (or feature vectors) in the training data. Benchmarks must be selected in a way the program feature space is homogeneously covered.

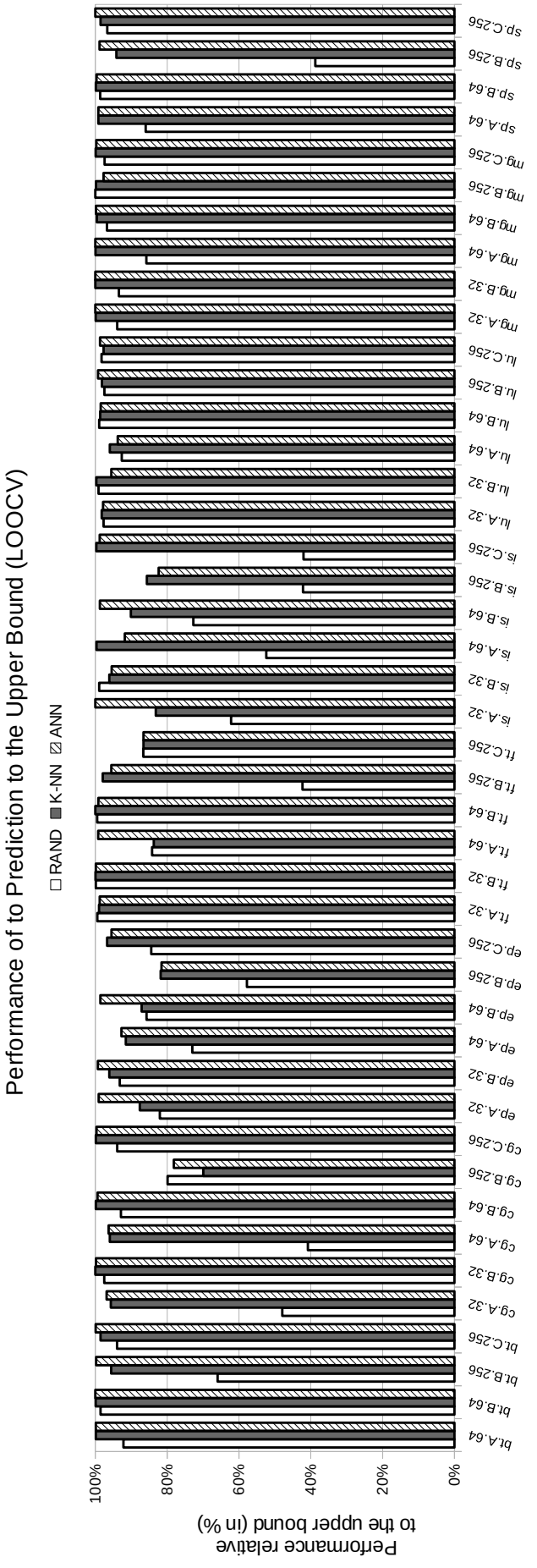


FIGURE 4.7: Performance gain, relative to the OPUB, for the 8 NPBs using parameter settings estimated with random selection (RAND), k-NN and ANNs.

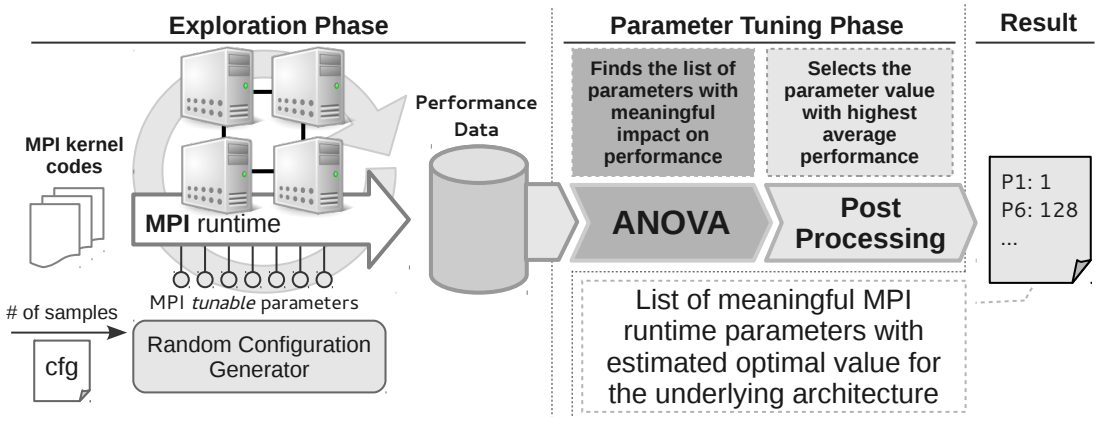


FIGURE 4.8: Parameter tuning and application optimization design.

4.7 Auto-Tuning with ANOVA

In this Section, we propose a two-phase method for tuning MPI runtime parameters for individual clusters based on the execution of a set of kernels and micro-benchmarks which characterize an application workload. We employ a statistical method called ANOVA [80] to identify the parameters with the highest effect on the execution time, and subsequently the parameter values which deliver on average better performance. Differently from the approaches presented so far, this method delivers a single parameter setting which is defined as the optimizing trade-off for a class of applications with a workload similar to the one used during the characterization phase. An overview of our method is described in Figure 4.8.

In a first *exploration* phase, those selected kernels are executed on the target machine with varying configuration instances of the MPI runtime parameters. Because the exploration space can be too large, and an exhaustive exploration is infeasible, random settings are generated up to a number defined by the user, as shown in Section 4.3. During the second *parameter tuning phase*, the training data is collected and analyzed using ANOVA. The outcome of this phase is a set of parameters (or a configuration C) with a major impact on the overall performance, for which the "best" setting with the maximum performance improvement is estimated for every considered cluster. Differently from previous approaches, our method requires to be executed only once, for every cluster architecture. The derived parameter setting can then be utilized to replace default parameter settings provided by the Open MPI library. Applications executing on the cluster will automatically benefit from the new parameter values.

We used this approach to tune parameter settings for two of the clusters, the IBM Blade and the LEO2 clusters, as listed in Table 4.1. For the exploration phase we employed 5 kernels of the NPBs. Estimated parameter settings by ANOVA have been tested

against a set of real codes from the SPEC MPI 2007 benchmark suite [1]. For the first exploration phase we used the same data gathered during the random exploration of the optimization space discussed in Section 4.3. As a consequence, with the OPUB we refer to the configuration instances reaching maximum performance gain seen during that exploration.

4.7.1 Parameter Selection

Factorial ANOVA takes n independent variables (or *factors*), each of them assuming a number of *levels* (or values). The method then measures for each level how different the mean values are and how much of the observations are spread out around their respective means. ANOVA is mainly utilized to verify whether the *null hypothesis* (i.e., there is no statistical difference between the means of each level) can be rejected with a determined confidence level. The analysis of variance can be conducted either by looking at single factors independently (i.e., 1-way ANOVA) or by capturing interdependencies among multiple factors (i.e., 2-way or in general n-way ANOVA).

For example, a specific MPI runtime parameter may not show any meaningful correlation with the execution time when analyzed alone, but may become important when coupled with other parameters. Unfortunately, the algorithm which computes the n-way ANOVA is of complexity $O(s^n)$, where s is the number of samples and n the number of factors. Since in our settings, $s \approx 20000$ and $n \approx 27$, the problem becomes largely infeasible to be solved. For this reason, we resort on *1-way ANOVA* that is of linear $O(S)$ complexity and, as we will show later in this section, delivers satisfactory results.

We used in our experiments the ANOVA package from `R` [92]. Before running the analysis, we computed the performance gain of each experiment by dividing the execution time of each parameter configuration instance with the Open MPI default execution and then normalized this value with respect to the OPUB. As depicted in Figures 4.1 and 4.2, different kernels achieve different levels of performance improvement, therefore, by dividing the performance gain with the relative OPUB, each kernel will contribute to the final solution with the same weight. Table 4.3 displays the parameters with a meaningful impact on the execution time resulting from 1-way ANOVA under a confidence level of 99.9% on LEO2 and IBM blade machines. A dash '–' means that the *null hypothesis* of no variance between the parameter values can not be discarded, meaning that changing the parameter value does not cause modifications in execution time. In such cases, the default Open MPI's setting is used. For the other parameters recognized to have a meaningful impact, we show the value with the highest mean performance gain.

Table 4.3 also shows that different parameters are selected to impact the execution time on the two different clusters. Nevertheless, eight of them significantly influences the performance of both computers, as follows (a complete description of the parameters is given in Appendix B):

- `mpi_paffinity_alone` for controlling affinity mapping of processes to cores;
- `btl_sm_eager_limit` and `btl_openib_eager_limit` for setting the eager limits for both intra-node and inter-node communications;
- `mpi_yield_when_idle` used by some applications to oversubscribe cores with MPI processes. In the Open MPI library, when a process blocks because of synchronization issues, spin-lock (or active wait) is always utilized as MPI assumes node cores to be exclusively used and not shared with other programs. However, for some applications it makes sense to spawn more MPI processes than available cores, a.k.a. *over-subscription* of nodes. The `mpi_yield_when_idle` flag forces the MPI library to work in *aggressive* (when disabled) or *degraded* mode. In aggressive mode, a blocking MPI processes never voluntarily gives up a core to other processes, while in degraded mode each process frequently yields the processor to its peers, thereby allowing all processes to make progress;
- `mpi_preconnect_mpi` for establishing a fully-connected topology during the `MPI_Init()` initialization phase which is beneficial for communication-intensive applications but detrimental to embarrassingly parallel ones;
- `mpi_leave_pinned` used to pre-register user message buffers so that the RDMA protocol can be used[1];
- `max_send_size` that defines the maximum chunk size used to send large messages via the `OpenIB` or `SM` layers for intra- and inter-node communications. By default, Open MPI splits messages into smaller chunks to enable a pipelining effect between sender and receiver. Large messages usually lead to better efficiency but with the drawback of inhibiting the pipelining effect.

4.7.2 Parameter Optimization

We utilize the ANOVA's results to find the parameter setting which represents a good compromise across the computational kernels. In `R`, we use the `model.table(anova_output, "means")` command to produce, for each meaningful parameter, the average

[1] Unfortunately, we could not explore this parameter on the LEO2 machine because user authorization reasons.

Parameter name	*IBM* ANOVA	*IBM* BEST	*LEO2* ANOVA	*LEO2* BEST
`mpi_paffinity_alone`	1	1	1	1
`mpi_yield_when_idle`	0	1	1	0
`mpi_preconnect_mpi`	1	1	1	1
`mpi_leave_pinned`	1	1	NA	NA
`coll_*`				
`sm_tree_degree`	–	8	–	8
`sm_control_size`	–	64K	128K	4K
`sm_fragment_size`	–	16K	4K	8K
`sync_barrier_after`	–	0	1000	1000
`sync_barrier_before`	–	0	–	5000
`tuned_init_tree_fanout`	–	16	–	8
`tuned_init_chain_fanout`	–	8	–	4
`btl_sm_*`				
`eager_limit`	–	8K	16K	32K
`max_send_size`	8K	16K	128K	32K
`rndv_eager_limit`	–	1K	1024	16K
`fifo_size`	–	1K	128K	16K
`num_fifos`	–	13	11	7
`btl_openib_*`				
`eager_limit`	64K	64K	128K	128K
`max_send_size`	64K	64K	128K	128K
`rndv_eager_limit`	–	8K	–	16K
`use_message_coalescing`	–	0	–	1
`user_eager_rdma`	–	0	–	0
`eager_rdma_num`	–	2	–	8
`use_async_event_thread`	–	0	–	1
`ib_max_rdma_dst_ops`	–	4	–	4
`rdma_pipeline_send_len.`	–	1M	–	512K
`rdma_pipeline_frag_size`	–	4K	–	4K
`min_rdma_pipeline_size`	–	16K	–	65K

TABLE 4.3: "Best" parameters values derived by ANOVA and BEST for both target architectures.

performance value associated with each parameter level. The optimizing parameter setting is therefore determined by selecting, for each parameter, the value with the highest average performance. An example is depicted in Figure 4.9, where the average performance gain associated with each level of the `btl_openib_eager_limit` parameter is shown for both target architectures. On each cluster, no performance gain relative to the Open MPI's default value can be observed up to a parameter value of 12288 Bytes. For LEO2 an average performance gain of 10% is registered with a parameter value of 128 KB, while for IBM blade the optimal setting is 64 KB.

Figure 4.9 also shows how overestimating or underestimating the eager limit usually

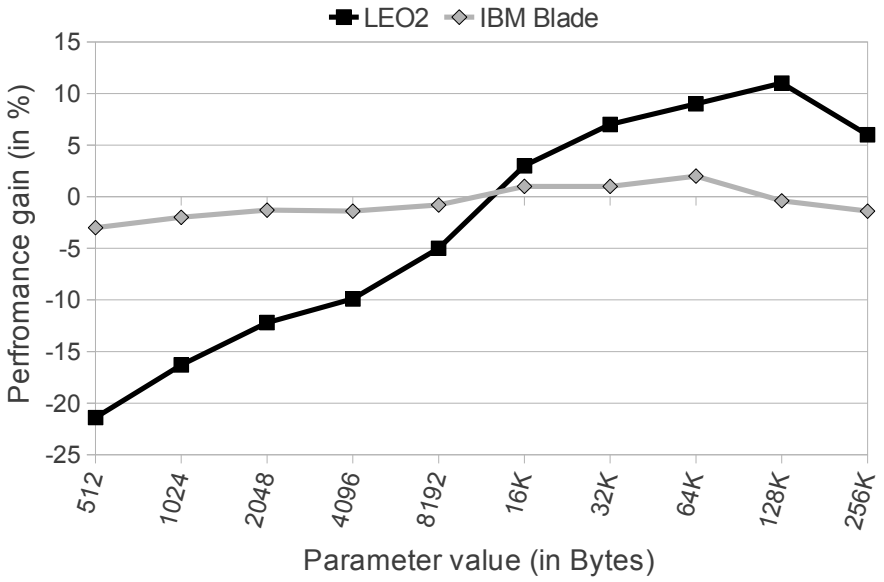


FIGURE 4.9: Mean performance gain for `btl_openib_eager_limit` parameter levels.

leads to severe performance losses. The eager limit should be set in a way that frequent messages are always delivered using the eager protocol which reduces the synchronization costs. Setting the threshold too high leads to a performance loss since larger buffers are allocated on the nodes. The optimal eager limit value generally depends on several factors such as the amount of cache per core, the latency, the bandwidth, and the interconnection topology. Therefore, finding the optimizing parameter setting by an analytical model often requires detailed knowledge of the target machine and the execution behavior of the input program. Our method automatically data mines those values by using a statistical method which does not require any expert knowledge. Similar information with the one depicted in Figure 4.9 for the `btl_openib_eager_limit` parameter can be extracted for the other parameters with meaningful impact on performance. We report in Table 4.3 the parameter values with the highest mean performance gain.

Changing affinity (i.e., `mpi_paffinity_alone = 1`) and pre-connect of MPI nodes (i.e., `mpi_preconnect_mpi = 1`) is beneficial for every architecture. Interesting is the fact that enabling the `mpi_yield_when_idle` flag on processors with SMT should in general be avoided because an SMT-based processor already has an efficient thread scheduling mechanism. Therefore, by forcing threads to yield we only increase the context switching overhead. On LEO2, which does not support SMT, the overall execution time improves when forcing threads to yield before suspending. Interesting is also that the value of the `btl_openib_eager_limit` parameter is half on IBM than on LEO2, which is a direct consequence of the CPU's cache sizes. IBM Blade has quad-core CPUs supporting 2-fold SMT with a shared level-3 cache of 8 MB leading to an ideal amount of 1 MB of cache per MPI process. On the other hand, LEO2 uses older Intel quad-core CPUs with 12MB

of level-2 caches shared among two cores, leading to an ideal 3 MB per MPI process. This extra amount of cache allows LEO2 to more efficiently use larger buffers and to achieve a higher performance gain.

Finally, we should note that the parameter settings derived by our method strongly depend on the workload type (i.e., the set of characteristic kernels) and only marginally on the input data size and communicator sizes being used during the exploration phase. To further investigate this aspect, we ran ANOVA on multiple subsets of the tuning data gathered during the training phase and reported the derived parameter values in Table 4.4, where each column represents the ANOVA output for a specific data subset:

- *A*, *B*, and *C* use the data generated by running the five NPB kernels with the input data size `A`, `B`, respectively `C`. Each subset contains performance data of both small and the large communicator sizes.
- *32*, *64*, respectively *256* use the data generated by the experiments run with a communicator of size 32, 64 respectively 256. As before, each subset contains both the performance data resulting from the small and the large input data sizes.

Table 4.4 demonstrates that for each subset of the tuning data, the number of relevant parameters may vary, however, their value across different data subsets remains largely constant. This insight can be used to drastically reduce the duration of the tuning phase, avoiding kernel executions for multiple input data and communicator sizes. Large input data sizes are nevertheless preferable, since for large messages the size of Open MPI's runtime buffers converges towards a dimension that optimizes the cache use on the target architecture, as indicated by the `btl_openib_eager_limit` threshold.

4.7.3 Experimental Evaluation

We employ the parameter settings determined by ANOVA based on the five NPB kernels at the end of the parameter tuning phase to optimize the execution of 10 applications from the SPEC MPI 2007 suite [1]. The goal is to demonstrate that the parameter values delivered by our technique outperform Open MPI's default setting for new unseen input codes. We also validate the results against two other tuning strategies for runtime parameters:

RAND: Randomly selects a parameter combination for every input program. To properly simulate this selection strategy, 5 random configurations are generated (for each input code) and the average performance gain value computed.

	IBM Blade				*LEO2*				
Parameter name	*A*	*B*	*32*	*64*	*A*	*B*	*C*	*64*	*256*
`mpi_paffinity_alone`	1	1	1	1	–	1	–	1	–
`mpi_yield_when_idle`	0	–	0	–	1	1	–	1	1
`mpi_preconnect_mpi`	1	1	1	1	1	1	1	1	1
`mpi_leave_pinned`	1	1	1	1	NA	NA	NA	NA	NA
`coll_*`									
`sm_fragment_size`	–	–	–	–	–	4K	–	4K	4K
`sync_barrier_after`	–	–	–	–	–	1000	–	1000	1000
`btl_sm_*`									
`eager_size`	–	16K	–	–	–	–	–	–	–
`max_send_size`	–	8K	–	–	128K	128K	128K	128K	128K
`rndv_eager_limit`	–	–	–	–	1024	1024	–	1024	1024
`fifo_size`	–	–	–	–	128K	128K	128K	128K	128K
`num_fifos`	–	–	–	–	–	6	–	3	6
`btl_openib_*`									
`eager_limit`	16K	64K	64K	64K	128K	128K	128K	128K	128K
`max_send_size`	–	64K	128K	–	128K	128K	128K	128K	128K

TABLE 4.4: ANOVA results on a selected subset of tuning data for both target architectures.

BEST: Randomly generates 80 training configurations selects the combination with the largest performance gain across the entire set of kernels.

First, we show in Figure 4.10 the performance relative to OPUB obtained by executing the five NPB kernels with the configurations delivered by RAND, BEST and ANOVA based on parameter values from Table 4.3. ANOVA reaches an average performance gain of around 13% with respect to the default setting on IBM Blade, and of around 24% on LEO2, i.e., 90% of the performance improvement available on the machine. On the other hand, RAND delivers on both machines almost no performance gains as expected, while BEST achieves an average performance gain similar to ANOVA of 10% on IBM Blade and 24% on LEO2. Compared to the approaches presented in Sections 4.5 and 4.6, which ideally deliver a performance improvement close to OPUB, our approach yields approximatively 88% of the OPUB without requiring any additional run of the input program.

Finally, we conducted an experiment to understand how the parameters tuned for one particular architecture affect the performance of running new codes. We selected ten applications from the SPEC MPI 2007 [1] suite displayed in Table 4.5 and executed them on the two target architectures using the parameters tuned by RAND, BEST and

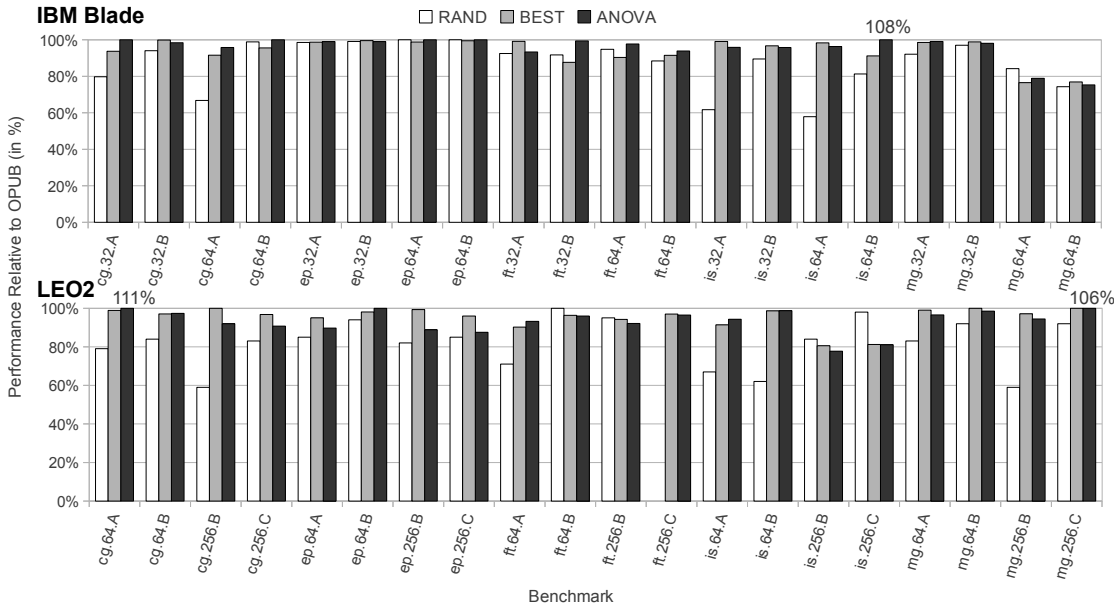


FIGURE 4.10: Performance relative to OPUB for the NPB kernels executed with the parameters tuned by RAND, BEST and ANOVA.

ANOVA. We simulated new and not previously encountered runtime conditions by running the SPEC benchmarks using a communicator size of 48 on IBM Blade and 128 on LEO2. We chose the `mref` input data size with a runtime of approximatively two hours per iteration on our two machines. We repeated each experiment three times and reported the average performance values. Since we did not explore the optimization space for these benchmarks, we have no knowledge of the OPUB and, therefore, report only the absolute performance gains obtained. The results show that RAND does not achieve any improvement over the Open MPI default settings, as already observed in the previous experiment. On the IBM Blade, the parameter tuning with ANOVA only slightly outperforms BEST, however, ANOVA delivers delivers substantially better results on LEO2. The reason in our belief is the larger number of relevant parameters detected to by ANOVA on LEO2, which increases the probability of BEST to select less performance-efficient parameter values. Because ANOVA determines parameter settings by combining information gathered from many executions of the characteristic kernels, the outcome is more reliable. The average performance improvement measured for the SPEC codes is about 20%, which is comparable to the one observed for the kernel codes in the tuning phase.

4.8 Summary

In this chapter we analyzed the importance of runtime parameter tuning for MPI applications. They provide a way to significantly reduce the execution time of a program

	IBM Blade				*LEO2*			
Benchmark	*Baseline [sec.]*	*RAND*	*BEST*	*ANOVA*	*Baseline [sec.]*	*RAND*	*BEST*	*ANOVA*
104.milc	437	-2%	13%	14%	323	-5%	-21%	9.0%
107.leslie3d	1737	4%	6%	6%	770	1%	-20%	13.0%
113.GemsFDTD	1072	-3%	-0.4%	-0.3%	624	2%	-11%	-3.4%
115.fds4	741	0.7%	-3 %	-1%	–	–	–	–
122.tachyon	762	0.2%	0.4%	1.2%	247	-1.5%	-34%	-1.4%
126.lammps	767	7%	7%	7%	311	-12%	-36%	8.0%
128.GAPgeofem	572	-1%	4%	4%	–	–	–	–
130.socorro	1025	-0.5%	1.2%	2%	391	-7%	-3%	37.6%
132.zeusmp2	1070	-0.7%	-0.4%	0.2%	743	15%	25%	52.6%
137.lu	1573	0.1%	-2%	3%	388	12%	14%	45.1%
AVG GAIN		0.4%	**2.5%**	**3.4%**		0.6%	-11%	**20.07%**

TABLE 4.5: Performance gain for the SPEC MPI 2007 [1] applications executed using the parameter settings estimated by RAND, BEST and ANOVA on the two target architectures.

without changing the application code. We derived three methods for estimating optimal parameter settings which are suited for different application scenarios.

The evolutionary method delivers the best parameter settings specific to the application (for a particular input configuration) and the target machine. The major drawback is the fact that the application code to be optimized has to be executed multiple times (50 to 150 depending on the required performance). Derived parameter settings deliver on average a performance improvement of around 27% on the LEO2 cluster and 15% for the Karwendel and IBM clusters. Use of the evolutionary search is suggested for those cases where an in-house cluster is being used to run a handful of application codes with a stable input configuration (communicator and input data sizes). In such a case, time invested in the exploration of the search space is amortized across several executions of the program.

For those scenarios where many runs of the input code cannot be afforded, the ML-based method is preferred. A training phase is used to create a prediction model which delivers optimized parameter settings based on a set of program features. The training phase is usually expensive but it is executed only once, when the cluster is deployed. Input programs must also be executed once in order for the feature extraction to take place on the collected execution traces. As shown in the experiments, the parameter settings delivered in this way are very close to the performance upper-bound uncovered during the exploration of the optimization space, i.e., the OPUB. One limitation of this method is the amount of parameters that can be estimated by means of ML algorithms. The number should be kept small otherwise prediction accuracy may dramatically decrease.

The last approach for parameter tuning aims at replacing the role of default parameter setting to match a given cluster system. This method is based on an, expensive,

exploration phase which needs to be conducted only once. During this phase computational kernels which characterize the workload of a cluster system are executed with varying parameter settings. The method delivers a single parameter setting which is, by definition, a trade-off across the computational kernels that are examined during an exploration phase. Unseen input program with similar workload characteristics benefits from the derived parameter values. This last method is well suited for those scenarios where a cluster is shared among many users belonging to the same application area.

Chapter 5

Message Passing-Aware Compiler Analyses and Transformations

Message passing programs are often challenged to scale up to several million cores. In doing so, the programmer tunes every aspect of the application code. However, for large applications, this is not practical and expensive tracing tools and post-mortem analyses are employed to guide the tuning efforts finding hot-spots and performance bottlenecks.

In this chapter, we revive the use of compiler analysis techniques to automatically detect and apply optimizations to communication statements of a distributed memory program. We present two optimizations which deal with communication statement placement. The first, empirically studies the interactions between CPU caching and communications for several different scenarios. Gained insights are then used to formulate a set of intuitive rules for communication calls placement. The second, focuses on increasing the communication/computation overlap of a message passing program using the result of exact data dependence analysis provided by the polyhedral model. This is obtained by employing a novel approach, presented in this chapter, which allows the representation of message passing routine semantics within the PM's constraints. Finally we examine the problem of static communication matching of message passing statements. In this thesis we present a novel static analysis which can establish a communication match for many real codes which is a prerequisite for several compiler transformations.

5.1 Introduction

5.1.1 Message- and Cache-Aware Compiler Optimizations

When writing a message passing program, the programmer must guarantee the semantic correctness ond position the communication routines within the code. There are many architectural aspects which may play a role in determining a good program position to place send or receive calls. For example, the request management overhead of asynchronous routines – which are commonly used to hide communication costs – may penalize performance for small message sizes. Runtime systems for distributed memory libraries (e.g., MPI [93] and UPC [7]) usually employ optimizations to hide many architectural details from the programmer. For example long messages may be split into smaller chunks to enable pipelining [94]. On the contrary, when too many short messages are sent, the runtime system may try to coalesce information into larger messages reducing the injection rate [8, 95]. Optimizations done at runtime are highly effective since the system is fully aware of the underlying architecture. However, most of the decisions have already been made by the programmer in the source code and therefore, at this stage, it is often too late to overcome performance problems. For these reasons, production codes are usually hand-tuned for particular target architectures.

In Section 5.2, we study the impact of CPU cache on MPI communication routines. We measured, with a synthetic benchmark, the differences in terms of execution time, for point-to-point operations performed when the data being sent is fully loaded into the CPU cache or not. We repeated the experiment with multiple configurations, i.e., intra-node and inter-node. In the same way we measured the impact of point-to-point communication routines on the application cache by accessing application data, previously loaded into the cache, right after the communication is performed. From the gathered data we derived a set of rules and guidelines which can be utilized to transform the input program for improved cache utilization and thus performance. To the best of our knowledge, this aspect has been largely neglected until now. Work in literature focuses on quantifying the impact of local memory on communications [96]. Those works are largely concerned with *non-regular* data types which involve expensive packing/unpacking operations and optimizing the way the MPI library handles them; whereas our work focuses on contiguous data and how the impact of communication routines can be exploited, by a programmer or a compiler, to optimize the input code.

5.1.2 Exact Dependence Analysis for Increased Communication Overlap

Compiler technology has been used in the past to optimize message passing programs [15, 16, 97–99]. The main idea is to extend the compiler analysis module to understand the semantics of message passing routines and treat them not just like a library call but as a language construct. In doing so, existing compiler analysis can be utilized to uncover optimization potentials hidden within the input code.

In Section 5.3 we show an approach based on compiler analysis, and specifically *exact* data dependence analysis to maximize the computation/communication overlap for a given input code. Indeed, increasing the time window on which computation and communication can be performed in parallel (or overlapped) is one of the well known rules of thumb used to optimize message passing codes. As opposed to the previous compiler-based approaches, we utilize finer-grain *exact* analyses produced by the PM [100]. Unlike the traditional dependence graph, which contains data dependency information between the program statements, the *dependence polyhedron* lists dependencies on the basis of *statement instances* [101] (see Section 2.3.2). By using this more detailed analysis our approach increases the time interval between generating the data or buffer availability and the final consumption of the data.

5.1.3 Static Matching of Communication Statements

A main challenge for message passing programs is the *static matching* of messages, i.e., to determine if a given send statement may form a communication channel (*match*) with a receive statement. This problem is similar to the serial alias analysis [102] and thus of paramount importance for many higher-level analyses and transformations. For example, well-known approaches for communication tiling [15] and message coalescing [103] rely on a matching analysis.

The idea behind tiling is to replace a single data transfer of N elements between a pair of processors with N/T message exchanges, each with T elements. Coalescing corresponds to a dual approach by aggregating N distinct sends operations into one. It is easy to understand that any update to a send or receive operation, which changes the amount of data being transferred, requires to update all matching communication statements and their *transitive closure*. In most previous works, e.g., [15, 103], the matching problem has been neglected assuming that the user specifies all matches explicitly. This essentially prevents any automated optimization and increases code development and maintenance significantly. Thus, those promising transformation techniques did not find wide adoption.

In this thesis we present a novel approach for static matching of communication statements. We use the PM [23, 24] to augment the communication statements in a program with the knowledge of the subset of processes involved in the communication. We use polyhedral techniques to symbolically compute the number of instances (each set's *cardinality*) of all communication statements. We then model the matching problem as a *bipartite graph*, and use the cardinality information as capacity; hence we transform the matching problem into a parametric *maximum network flow problem* [104]. A matching is said *valid* only if the computed flow uses all the available capacity. The algorithm guarantees that none of the possible matches is excluded.

5.2 Message- and Cache-Aware Compiler Optimizations

As shown in Section 2.2.1, the *send* routine transfers a buffer, $buff_i$, allocated within a process', p_i, memory space, i.e., $buff_i \subseteq \mathcal{A}_{p_i}$ to a receiver process, p_j, which stores the received data in a buffer, $buff_j \subseteq \mathcal{A}_{p_j}$. These sender ($buff_i$) and receiver ($buff_j$) buffers, are stored in the main memory of processes p_i and p_j, respectively. However, hardware caches are interleaved between the computing units and the main memory. This means that when a *cu* accesses elements of a buffer, these elements are automatically copied (by the hardware coherency protocol) onto the caches and thus made available to the requesting *cu*. In this section we investigate the impact of the caches on point-to-point on point-to-point communication routines. communication routines.

5.2.1 Analyzing MPI Cache Behaviour

In order to highlight the effects of CPU caches on MPI communication routines we wrote a synthetic benchmark. The main goal of the MPI cache benchmark is to capture differences in terms of execution time between communication routines with multiple configurations of the CPU cache and additionally, to measure their impact on the cache. In doing so, we also collected the value of several performance counters using the PAPI library [105]. Many benchmarking suites for MPI exist in literature[106, 107]. Coddington et al. wrote a survey of benchmarking tools for MPI's point-to-point communications [108]. However none of those is designed to capture cache behaviour of MPI routines. Some of the tools, e.g. MPIBench[107] and SKaMPI[106], provide options to pre-load messages into the cache before performing the communication but they do not provide a way to precisely capture the level of cache pollution caused by MPI communication routines. The benchmark code which has been developed for this purpose follows the guidelines for reproducibility of measurements described in [109]; the code is publicly available from [110]. Beside the execution time, the benchmark registers the

values of multiple PAPI performance counters which will be used to understand low level implementation details of the underlying MPI library.

The benchmark is split into two scenarios, `SCN1` and `SCN2`, which are further described in this section.

5.2.1.1 Scenario 1 – `SCN1`

`SCN1` studies the behaviour of single MPI send/receive routines. A skeleton of the benchmark code is shown in Listing 5.1. With this benchmark we are interested in capturing the behaviour, in terms of performance counters and runtime, of two basic MPI routines, i.e. `MPI_Send` and `MPI_Recv`, considering different states of the cache. The benchmark performs a ping-pong operation with three different initial cache states. In the first case, `INV` (lines 34-37), we make sure all the content in the cache is wiped out and none of the data elements being sent or received are present into any of the CPU caches. The second cache configuration, `EXCL` in lines 39-43, entirely pre-loads into the cache the message data right before the communication is performed. Data elements are only read which means the corresponding cache lines are in the "exclusive" state. In the last cache configuration, `MOD` in lines 45-49, cache lines are preloaded in the "modified" state.

5.2.1.2 Scenario 2 – `SCN2`

In the second scenario, referred as `SCN2`, we want to capture the level of cache pollution caused by send/receive communication routines. The skeleton of the benchmark code for this setting is shown in Listing 5.2. This is obtained by measuring the time, together with other performance counters, required to traverse the array containing the message data previously exchanged in the ping-pong operation. This is done considering multiple configurations of the cache. In `INV`, we start by cleaning the caches, we then perform the message exchange and, upon competition of the send/receive, data is traversed and the measurement is performed (lines 23-27). In the second configuration, `PRE`, we pre-load the message data into the cache before performing the message exchange (lines 29-34). It is worth noting that, in both cases, the code for which we perform the measurements does not contain any communication statements. The obtained data is compared with the values measured while traversing the message buffer without previously performing any communication. Also in this case we consider two cache configurations, i.e., cache is invalidated before the array elements are accessed, `BASE_INV` in lines 11-14, or the array is fully pre-loaded into the cache before being traversed, `BASE_PRE` in lines 16-20.

```
#define CL 64 // Size of a Cache Line (in bytes)
#define CS    // Total Cache Size (in bytes)

void cache_bench(unsigned N) {
   assert( rank < 2 && "Only two MPI processes allowed");
   // @@ Warms up communication channels
   for(size_t ci=128, end=CS*4; ci<=end; ci*=2) {
      // @@ Allocates Data buffers
      char* msg = new char[ max(ci,CS)*2 ];
      char* buff = &msg[ max(ci,CS) ];
      // @@ Performs Measurments
      for (size_t i=0; i<N; ++i) { test_inv(msg,buff,ci); }
      for (size_t i=0; i<N; ++i) { test_excl(msg,buff,ci); }
      for (size_t i=0; i<N; ++i) { test_mod(msg,buff,ci); }
      delete[] msg;
   }
}
inline void comm(char* msg, size_t n) {
   sync();
   if (rank==0) {
      START_INSTRUMENTATION();
      MPI_Send(msg,n,MPI_BYTE,1,0,comm);
      END_INSTRUMENTATION();
   } else {
      START_INSTRUMENTATION();
      MPI_Recv(msg,n,MPI_BYTE,0,0,comm,MPI_STATUS_IGNORE);
      END_INSTRUMENTATION();
   }
}
inline void clean(char* buff, size_t size) {
   for(long i=0, i<max(CS,size); i+=CL) { load(buff[i]); }
}
// Communication from 'invalid' cache
inline void test_inv(char* msg,char* buff,size_t size) {
   clean( buff, size );
   comm( msg, size );
}
// Communication from warm cache ('exclusive')
inline void test_excl(char* msg,char* buff,size_t size) {
   clean( buff, size );
   for(long i=size-1; i>=0; i-=CL) { load(msg[i]); }
   comm( msg, size );
}
// Communication from warm cache ('modified')
inline void test_mod(char* msg,char* buff,size_t size) {
   clean( buff, size );
   for(long i=size-1; i>=0; i-=CL) { write(msg[i]); }
   comm( msg, size );
}
```

Listing 5.1: MPI Cache Benchmark Code Skeleton for `SCN1`

We repeated the experiment with two different process allocations in order to test intra-node and inter-node point-to-point communications. This was obtained by allocating the two MPI processes respectively on different computing nodes or on the same multi-processor machine. In both cases, the use of affinity settings ensured the MPI processes

```
#define CL 64 // Size of a Cache Line (in bytes)
#define CS    // Total Cache Size (in bytes)

// traverse the message buffer
inline void traverse(char* msg, size_t size) {
   START_INSTRUMENTATION();
   for(long i=0; i<sizel i+=CL) { load(msg[i]); }
   END_INSTRUMENTATION();
}
// Traverse the msg array when not in cache
inline void test_base_inv(char* msg, char* buf, size_t size) {
   clean(buff, size);
   traverse(msg, size);
}
// Traverse the msg array when preloaded into cache
inline void test_base_pre(char* msg, char* buf, size_t size) {
   clean(buff, size);
   for(long i=size-1; i>=0; i-=CL) { load(msg[i]); }
   traverse(msg, size);
}

// Communication from 'invalid' cache and access to data
inline void test_inv(char* msg,char* buff,size_t size) {
   clean(buff, size);
   comm(msg, size);
   traverse(msg, size);
}
// Communication from warm cache ('exclusive') and access to data
inline void test_pre(char* msg,char* buff,size_t size) {
   clean(buff, size);
   for(long i=size-1; i>=0; i-=CL) { load(msg[i]); }
   comm(msg, size);
   traverse(msg, size);
}
```

Listing 5.2: MPI Cache Benchmark Code Skeleton for `SCN2`

are bound to a specific core of distinct CPUs. This is done in order to take full advantage of the CPU cache and avoid conflicts which arise when multiple processes share the same last level cache.

5.2.1.3 Hardware Platforms

We evaluated the code on 2 computing platforms summarized in Table 5.1. The LEO3 cluster system consists of 162 compute nodes (with a total of 1944 cores). All nodes are connected through an Infiniband 4x QDR high speed interconnect. Each node contains two Intel Xeon CPU based on the Nehalem architecture where Hyper Threading (HT), or 2-fold SMT, has been disabled. The VSC2 cluster has been already presented in Section 3.4.5. CPU cache layout for the two system is summarized in Table 5.1. These are both production clusters and the measurements have been taken while the clusters were

System name	**LEO3**	**VSC2**
Max # of *cn*	162	1.314
Chips per node	2	2
*cu*s per chip	6	8
Core Architecture	Intel Xeon X5650	AMD Opteron 6132 HE
Clock Frequency	2.67GHz	2.2GHz
L1 cache	32KB + 32KB	64KB + 64KB
L2 cache	256KB (private)	512KB (private)
L3 cache	12MB (shared by 6)	2x6MB (shared by 4)
SMT	Disabled	NA
Memory per Node	24GB DDR3	32GB DDR3
Interconnection	Infiniband 4x QDR	Infiniband 4x QDR
Kernel Version	2.6.32	2.6.32
Open MPI version	1.5.5	1.5.4
SM module	OpenMPI default	KNEM[111]

TABLE 5.1: Experimental target architectures.

fully operational, therefore some level of noise is expected to show up in the measurements. In order to reduce it we repeated each measurement 100 times and considered the median.

5.2.1.4 MPI Communication Protocols

The cache benchmark treats the underlying MPI library as a *black box*. This allows us to make considerations which are not biased towards a particular feature of an MPI implementation. However, MPI libraries are very complex and in order to correctly interpret the gathered data, implementation details cannot be completely neglected. Every MPI library exposes several "knobs" which can be used to tune the performance of a particular application on the underlying target platform, see Chapter 4. One of the most relevant threshold for point-to-point communication is the so called "eager limit". The eager protocol is not standardized by the MPI specification, however it is an implementation technique utilized by all MPI implementations. Every message exchanged between peer processes is subject to this protocol. MPI libraries typically use (at least) two algorithms, *eager* and *rendezvous*. When the size of the transmitted message is smaller than the specified threshold value, the message (together with an MPI header) is eagerly sent to the receiver. For larger messages the *rendezvous* protocol is utilized instead, see Section 2.2.1. The eager protocol is useful when latency is important because it avoids

CTS/RTS round-trip overhead. However it requires additional buffering at the receiver side. Rendezvous protocols are typically used when resource consumption is critical.

For example, the Open MPI library [112] uses multiple protocols, detailed in [113]. In the case of eager send, the behaviour is the same as described before. The rendezvous protocol however enables better latency hiding. When the communication is performed over RDMA-enabled networks (such as an OpenFabrics-based network, e.g., InfiniBand) the protocol is divided into three phases. In the first phase the RTS message is sent to the receiver, while the sender is waiting for the CTS message, it starts "registering" the rest of the large message with the OpenFabrics network stack. Since the registration (or *pinning*) is slow the process is pipelined so that registration latency is hidden.

In shared-memory, the rendezvous protocol can use several implementation mechanisms which have been presented over the last decade because of the increasing relevance of multi-core systems. Most shared-memory message passing implementations, such as Nemesis [114] device in MPICH2 and the SM component in Open MPI, depend on a double buffering memory scheme. An extra memory buffer is pre-allocated as an exchange zone between processes. Communication between the processes is performed using the so called copyin/copyout semantics (CICO). The sender process copies from the message buffer into the shared memory and in the same way the receiver reads it out and copies into the receiver buffer. In order to reduce latency, the copy happens in a pipelined way. However approaches exist, such as KNEM [111], which via an OS kernel extension, allows the direct copy from the sender to the receiver buffer. This mechanism has the advantage to eliminate the additional memory copy and therefore reduces both latency and cache pollution.

We perform our measurements using the default settings provided by the chosen MPI library. We use Open MPI with the default eager limit, which is set by default to 12 KiB for communication over Infiniband and to 4 KiB for intra-node communication, on both systems. In the LEO3 cluster we used the default shared memory provided by the Open MPI library which is based on the CICO mechanism. On the VSC2 cluster shared memory communications are performed using the KNEM kernel extension.

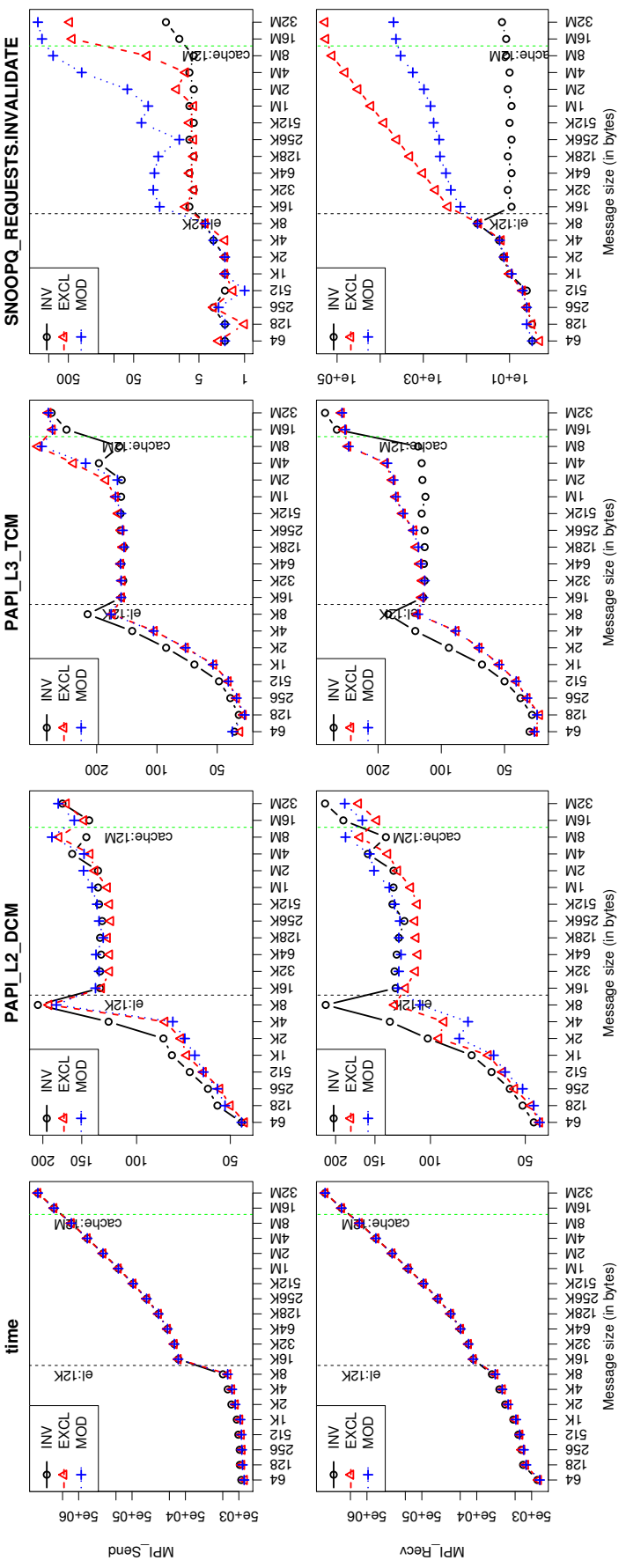


FIGURE 5.1: LEO3 Inter-node – SCN1 – Send/Receive

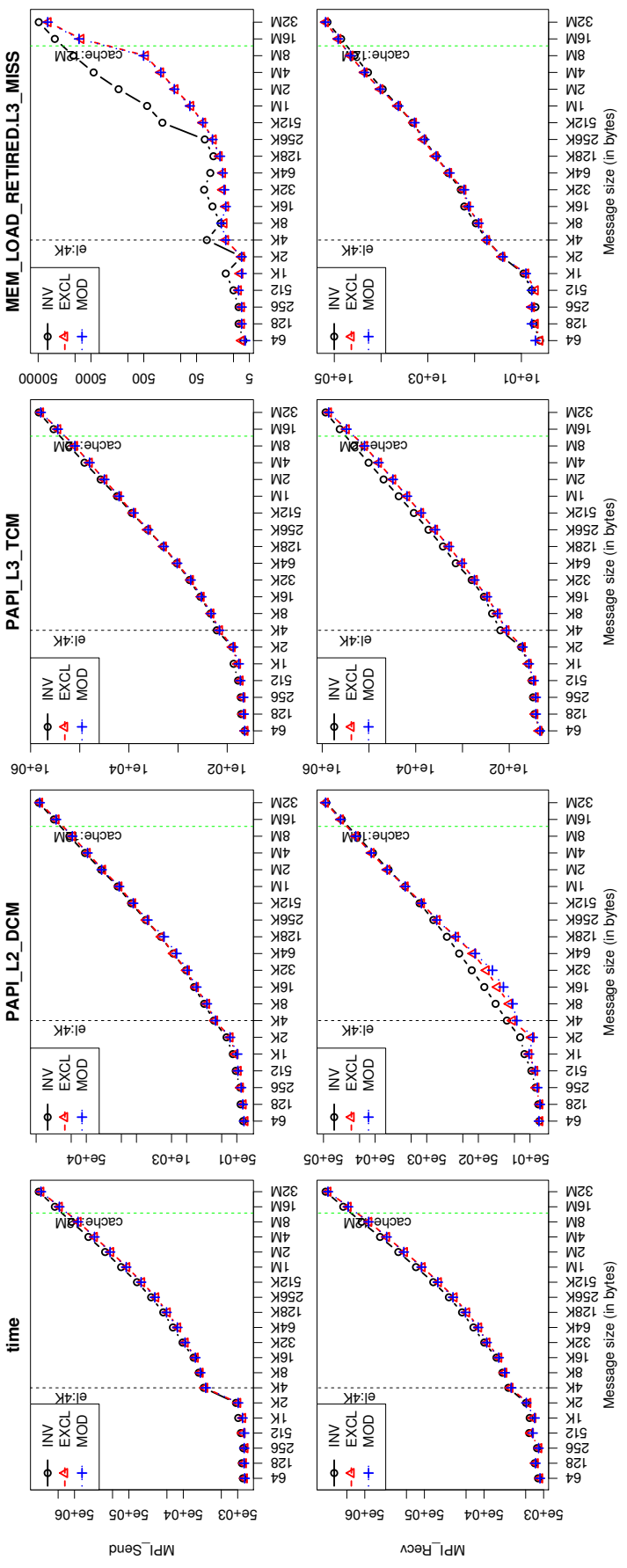


FIGURE 5.2: LEO3 Intra-node – SCN1 – Send/Receive

5.2.2 Benchmark results

In this section, the data gathered by running our cache benchmark for the LEO3 cluster architecture is shown and analyzed.

5.2.2.1 Inter-node communication – Infiniband

Figures 5.1 and 5.3 depict the values obtained by the two benchmark scenarios (`SCN1` and `SCN2`) using inter-node communication, over Infiniband.

Figure 5.1 shows several performance counters associated with the `MPI_Send` operation, in the first line, and `MPI_Recv`, in the second line, using the three cache configurations: `INV`, `EXCL` and `MOD`. The first column shows the execution time which, in order to be as precise as possible, is expressed in terms of number of CPU clock cycles. Differences in terms of the execution are barely noticeable. However, we can note that data preloading into the cache (as in `EXCL` and `MOD`) reduces the amount of L2 data cache misses (`PAPI_L2_DCM` counter in the second column) up to the eager limit, this is visible especially at the receiver side where buffering happens. The two routines have a reduced execution time, which reaches its peak of around 20% for messages of 8KiB, when the message data is preloaded into the cache.

After the eager threshold is exceeded, we still have better behaviour of L2 cache however there is an increase of L3 cache misses (`PAPI_L3_TCM` hardware counter) which is similar for the `EXCL` and `MOD` cache states. While the reduced cache misses in L2 cache are constant for increasing message sizes, L3 cache misses proportionally grow with the message size. To better understand the reason for this, we show another performance counter, in the last column of Figure 5.1, which depicts the number of *snoop* invalidation requests addressing the CPU. The snoop is a signal used to maintain cache coherency among processors in a SMP machine [20]. It can be noted that during the rendezvous protocol, the number of invalidation requests increases considerably if the message data is preloaded into the cache. This is more noticeable for the receiver as the NiC driver updates the message buffer in main memory and therefore eventual dirty copies in the cache need to be invalidated.

The measurements for the second scenario, `SCN2`, are depicted in Figure 5.3. As previously stated, this benchmark measures the performance resulting from accessing the message buffer right after being sent/received. We keep performance values for `BASE_INV` and `BASE_PRE` as a upper and lower bound for what we expect to be the performance from this scenario. Interesting is the number of L3 cache misses, in the case of the sender process, accessing the data after the send operation (`INV`) causes the same amount of

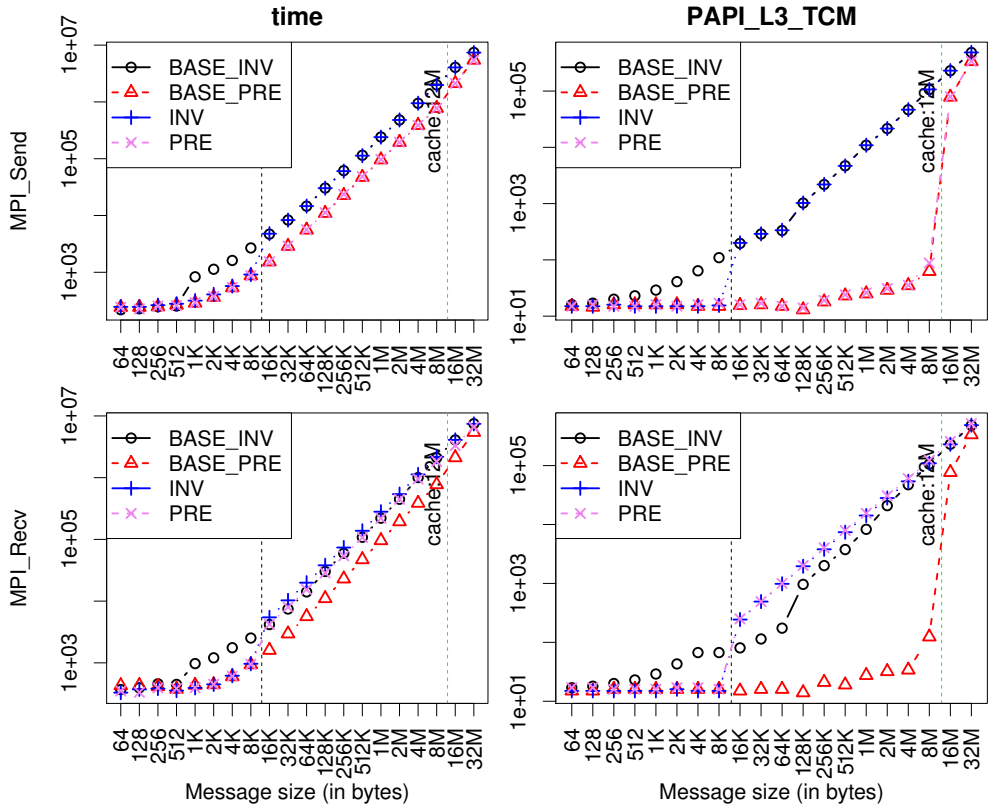


FIGURE 5.3: LEO3 Inter-node – SCN2 – Cache Pollution

misses measured for BASE_INV. This means the send operation does not pollute the application cache. However this is not true for messages which are smaller than the eager limit. In that case, there are no L3 cache misses for both INV and PRE configurations.

Major differences between sender and receiver happen beyond the eager threshold. In PRE, while at the sender side the amount of cache misses is comparable with the one measured for the BASE_PRE configuration; the receiver behaviour is instead similar to the BASE_INV case. The receive operation invalidates the entire L3 cache (as suggested by the memory bus snoop operations shown in Figure 5.1) and accessing the received elements costs as many memory operations as accessing it from a completely invalid cache (BASE_INV). Additionally, loading the data after the receive routine causes more misses than the BASE_INV configuration (which should be the performance upper-bound). Unfortunately we could not find a reasonable explanation for this. The increased amount of L3 cache misses has also a significant impact on the execution time which for INV and PRE is slightly higher than BASE_INV. In our opinion, the reason for this is consequence of the *memory pinning* operation performed by the MPI library. Also it is worth saying that the same kind of behaviour has been observed at the sender side when the data is preloaded in a "modified" state. In that case, the send operation invalidates all the preloaded cache lines and therefore accessing the buffer data after the communication

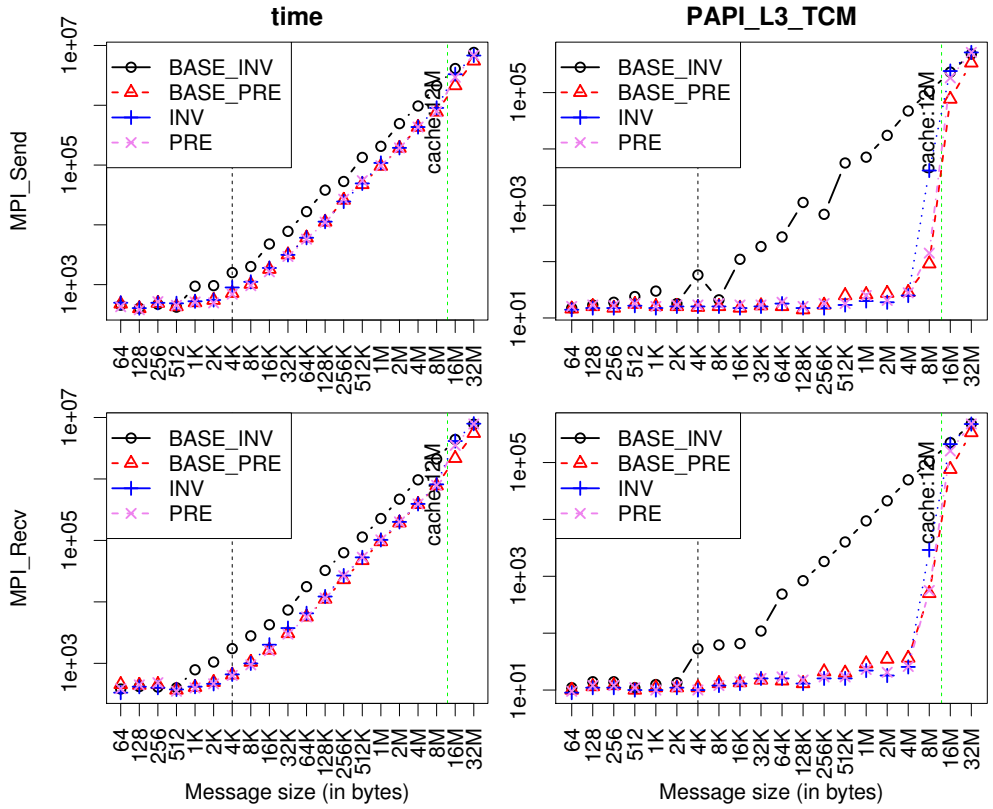


FIGURE 5.4: LEO3 Intra-node – SCN2 – Cache Pollution

routine is slower.

5.2.2.2 Intra-node Communication – SM

In Figures 5.2 and 5.4, the data obtained for shared memory configuration for the LEO3 cluster is shown.

Figure 5.2 depicts the measurements for SCN1. In this case we observe overall a much higher number of cache misses since the actual data exchange between the two MPI processes happens in shared memory. However, for the sender process, we see only small differences among the three configurations. We show the value of the MEM_LOAD_RETIRED:L3_MISS performance counter which proves the advantage, i.e., reduced number of memory load misses, due to fact of having the message buffer available in the cache. At the receiver side instead, we observe a smaller number of both L2 and L3 cache misses for messages up to the last level cache size. Overall, the performance of MPI routines is improved when data is preloaded into the cache and the gain reaches its peak, around 25%, before the cache size is exceeded. As already stated, in this machine shared memory communication is performed using a CICO mechanism. Because the transfer between

sender and receiver is done using a shared buffer, which for the Open MPI library is of 32 KiB, only a portion of the data cache gets polluted during the transfer.

This is visible in Figure 5.4. Differently from what observed for inter-node communications, in shared memory the message buffer is fully loaded into the cache for both `INV` and `PRE` configurations. However while the amount of L3 cache misses for `PRE`, `BASE_PRE` and `INV` is almost the same up to 4 MiB, at 8 MiB we start seeing a gap between the three configurations. The amount of cache pollution is higher at the sender side since the difference in terms of cache misses between `PRE` and `BASE_PRE` is noticeably higher than the receiver side. This is unexpected since the data transfer from the user buffer to the shared memory segment should be implemented using non temporal move instructions (e.g., `MOVNTDQ`), which avoids the target address to be loaded into the CPU cache. However, this penalty happens only for message sizes which are larger than half of the last level cache size.

5.2.3 Considerations and Optimization Guidelines

From the output of the MPI cache benchmark we derive, in this section, a set of intuitive rules to find a good placement for send/receive communication statements which better exploit the properties of the CPU caches. We divide our consideration into three subsections applying to specific ranges of the transmitted data, i.e., (*i*) from 1 byte up to the eager limit, (*ii*) from the eager threshold up to the last level cache size and (*iii*) beyond the available cache size.

5.2.3.1 From 1 Byte to the Eager Threshold

When the eager protocol is utilized, messages are transfered to the NiC using a `memcpy()` operation which has the side-effect of loading the content of the send buffer into the CPU cache. Therefore if the transmitted data is accessed right after the send operation, the data will be still available in one of the CPU caches. Additionally the `memcpy()` routine also benefits from having the source and target buffers preloaded into the cache. However, the input program could present dependencies (e.g., a read operation right after a receive statement) which do not allow this transformation to be applied. In such situation, the sent/received data should be accessed immediately after the communication routines or as late as before the message buffer content gets eliminated out from the caches.

For messages up to the eager limit, it is always preferable to perform the communication when the message data is cached. Received data should be immediately accessed.

5.2.3.2 From the Eager Limit to the Last Level Cache Size

We now consider the second message range, from the eager limit up to the cache size. In this range intra-node and inter-node communication differ and we treat them separately.

As far as inter-node send operations are concerned, we observe an increase in the number of L3 cache misses which is proportional to the message size in Figure 5.1. However the overall number of cache misses is small thus the execution time is not affected by it. More interesting effects can be seen in Figure 5.3. At the sender side, there is no cache pollution caused by the send operation. Therefore we expect no changes in the application performance from changing send statements placement.

However, things change dramatically at the receiver side. The receive operation invalidates all the preloaded cache lines in the case the message data was preloaded into the cache. Additionally, because of the memory pinning, utilized by the rendezvous protocol in Open MPI, accessing the received data right after the receive statement has a negative impact on performance. A similar behaviour was also observed for the sender process when the data is preloaded in a "modified" state as discussed in Section 5.2.2.2.

Inter-node: Avoid to access the transfered data immediately after the communication routines. If possible, perform all the computations on the message buffer before issuing a send operation. At the receiver side, delay the access to the receiver buffer by overlapping it with other computations.

For shared memory communication there is, in Figure 5.2, a reduction of L2 and L3 cache misses which is proportional to the size of the message being transferred. This has positive effects on the execution time which reaches a maximum improvement, of around 25%, both for sender and receiver processes, for 8 MiB messages. For shared memory communications, both the send and the receive routines populate the cache with the content of the message buffer and in the case the data is preloaded before the communication routine, the cache lines will not be invalidated. However, when the CICO mechanism is utilized, cache pollution may occur for large messages.

Intra-node: Access the message data after the communication statements, if the data is not already loaded into the cache, when

```
1  for(unsigned iter=0; iter<MAX_ITERS; ++iter) {
2
3     MPI_Sendrecv(&A[0][0], COLS, MPI_DOUBLE, bottom, 0,
4                  &A[ROWS-1][0], COLS, MPI_DOUBLE, top, 0,
5                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
6
7     for(unsigned i = 0; i<ROWS-1; ++i)
8        for(unsigned j = 0; j<COLS-1; ++j)
9           tmp[i][j]=A[i][j]+(A[i+1][j]+A[i][j+1])/4;
10
11    double** ttemp=A; A=tmp; tmp=ttemp; // swap arrays
12 }
```

LISTING 5.3: 3-point stencil code

the message size is smaller than $LAST\_LEVEL\_CACHE\_SIZE/2$ bytes. If the data is already in the cache, perform all the computations before invoking any communication routine.

5.2.4 Beyond the Last Level Cache Size

Beyond the cache size the behaviour of our benchmarks tend to converge, therefore no meaningful optimization rule can be defined. However, large messages can be divided into smaller chucks using a well known MPI code transformation referred in literature as software pipelining or message strip mining [98]. If the splitting size is chosen accordingly, the cache effects can be enabled.

5.2.5 A Case Study: 3-point Stencil

Following the optimization guidelines derived in the previous section, we manually tuned a 3-point stencil code which encodes a pattern commonly utilized in many HPC codes. A common way of parallelizing such stencil operation in MPI is shown in Listing 5.3. The code has a communication statement at the beginning of the loop which exchanges the first and last row of a 2-dimensional matrix which is updated by the following stencil computation. It is worth noting that while the receive operation must be performed before the last iteration of the `i` loop, the send operation has no dependencies and can be therefore issued at any program point, but before the swap procedure (line 11). We derived two versions of the stencil code depicted respectively in Listings 5.4 and 5.5.

Based on the our observations, the code has a poor cache behaviour as the array elements being sent, which are in a "modified" state, are accessed right after the communication statement, therefore after being flushed out from the CPU cache (when the rendezvous protocol is utilized). In order to optimize this aspect we can rewrite the code by moving

```
for(unsigned iter=0; iter<MAX_ITERS; ++iter) {

   for(unsigned j = 0; j<COLS-1; ++j)
         tmp[0][j]=A[0][j]+(A[1][j]+A[0][j+1])/4;

   MPI_Sendrecv(&A[0][0], COLS, MPI_DOUBLE, top, 0,
                &A[ROWS-1][0], COLS, MPI_DOUBLE, bottom, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);

   for(unsigned i = 1; i<ROWS-1; ++i)
      for(unsigned j = 0; j<COLS-1; ++j)
         tmp[i][j]=A[i][j]+(A[i+1][j]+A[i][j+1])/4;

   double** ttemp=A; A=tmp; tmp=ttemp; // swap arrays
}
```

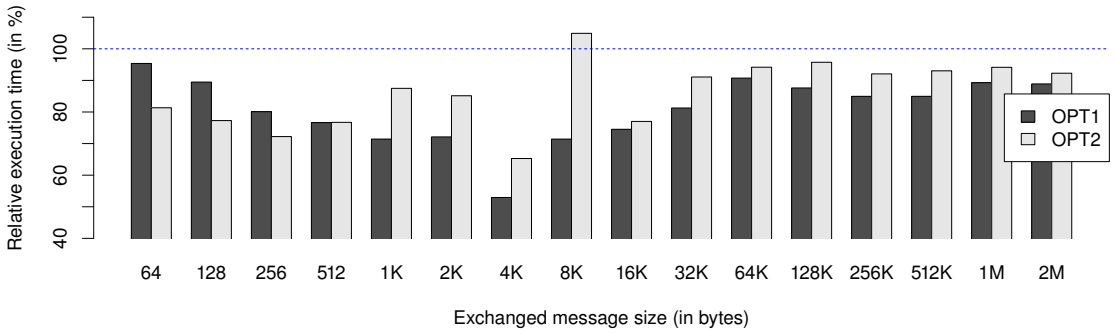
LISTING 5.4: Tuned 3-points stencil code ($OPT1$)

```
for(unsigned iter=0; iter<MAX_ITERS; ++iter) {

   for(long i = ROWS-3; i>=0; --i)
         for(long j = COLS-2; j>=0; --j)
               tmp[i][j] = A[i][j] + 1/4*(A[i+1][j]+A[i][j+1]);

   MPI_Sendrecv(&A[0][0], COLS, MPI_DOUBLE, top, 0,
                &A[ROWS-1][0], COLS, MPI_DOUBLE, bottom, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);

   for(unsigned j = 0; j<COLS-1; ++j)
         tmp[ROWS-2][j]=A[ROWS-2][j]+
                           (A[ROWS-1][j]+A[ROWS-2][j+1])/4;

   double** ttemp=A; A=tmp; tmp=ttemp; // swap arrays
}
```

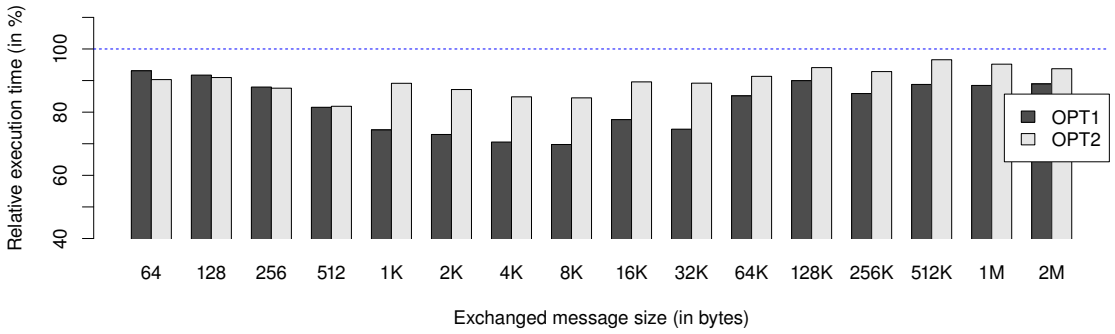
LISTING 5.5: Tuned 3-points stencil code ($OPT2$)

the communication right after the first iteration of the loop. This has two advantages: (i) it guarantees that the matrix rows which are going to be sent/received are freshly loaded into the cache; (ii) it avoids to access the received data right after the communication routine. The transformed code is depicted in Listing 5.4, we refer to this code version as `OPT1`.

The `OPT1` code version can however be further improved for the receive operation. As a matter of fact, the received data is not immediately consumed but accessed only in the last iteration of the stencil loop. This may not be optimal for messages which are smaller than the eager limit. For optimizing this aspect we can derive a second code version which utilizes the received data immediately after the data is available in the receiver buffer. This is obtained by reversing the order of execution of the stencil code. We traverse the 2-dimensional matrix from `ROW-3` backwards until the first row. In this



(a) Intra-node



(b) Inter-node

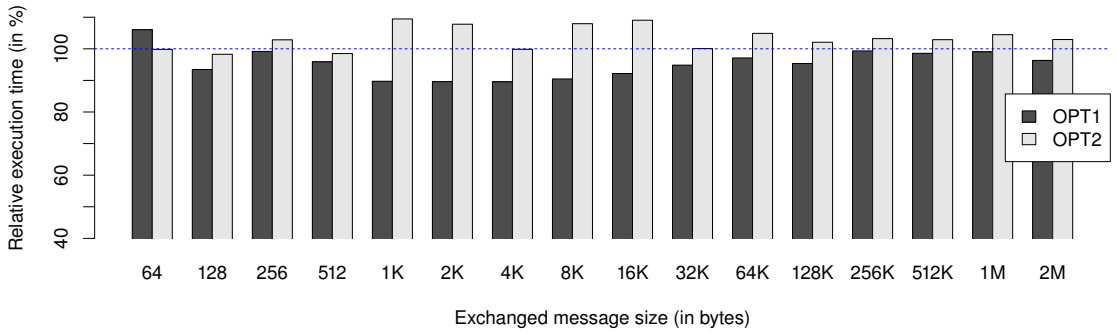
FIGURE 5.5: LEO3 – Evaluation of tuned 3-point stencil application code

way we make sure to have the send data already into the cache. We then perform the communication and successively complete the stencil by updating the last row. We refer to this code version as `OPT2`, the code is depicted in Listing 5.5. It is worth noting that in this version, the receiver buffer may not be loaded into the cache before the message exchange if the entire problem does not fit into the cache.

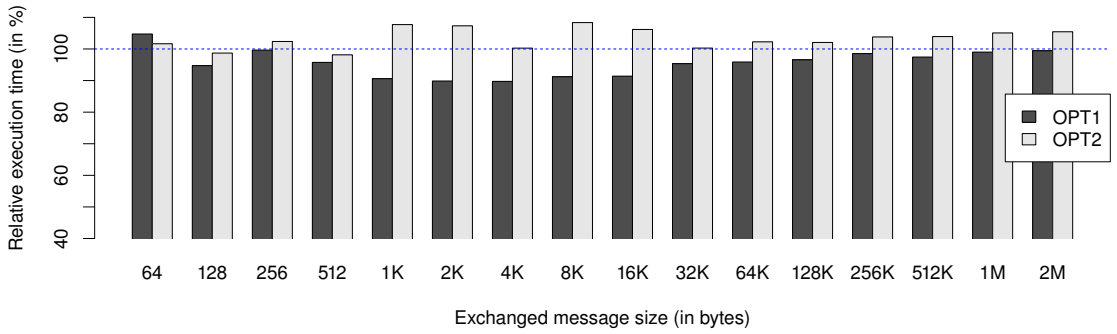
5.2.5.1 Evaluation

We evaluated the three versions of the stencil code on the two clusters described in Table 5.1. Each version has been executed multiple times with different problem sizes using two different process allocations, i.e. intra-node and inter-node. We ran the stencil code using two MPI processes to correlate the outcome with the results gathered by the cache benchmark. We measured the execution time of each code versions and used the value of the median obtained from 100 repetitions of the program.

Figure 5.5 shows the execution time of code versions `OPT1` and `OPT2` relative to the baseline solution, i.e.,5.3 for the LEO3 cluster. The x axis refers to the size of the message (in bytes) being exchanged by the stencil computation in every iteration. As expected, the `OPT2` version has better performance for small message sizes reaching, for shared memory,



(a) Intra-node



(b) Inter-node

FIGURE 5.6: VSC2 – Evaluation of tuned 3-point stencil application code

a performance improvement of around 20% for 256 bytes messages. For larger messages `OPT1` has a better performance reducing the execution time of the stencil code by 40%. For larger message, the advantage becomes smaller as the communication/computation ratio diminishes.

Figure 5.6 shows the results for the VSC2 cluster for both intra- and inter-node communications. Also on this machine, `OPT2` has an advantage over the original stencil code for very small message sizes. However, for larger messages this version is noticeable slower. The `OPT1` version, on the contrary, is faster for both inter- and intra-node communication. However, the measured performance improvement reaches approximatively 10%. We believe the poor performance of the `OPT2` version is due to the reversed access of array elements which may inhibit the CPU data pre-fetcher from correctly determining the data access pattern.

We showed the performance improvements a program can expect when communication statements are correctly placed. Compilers usually apply similar optimizations to sequential codes. In the next Section we propose a method to consider the semantics of communication statements by a compiler. Because this representation is based on a well established framework, i.e., the PM, existing analyses (like data dependence analysis) can be exploited to improve the behaviour of message passing programs.

```
1  for(unsigned iter=0; iter<NUM_ITERS; iter++) {
2  S0 MPI_Sendrecv(&A[ROWS-2][0], COLS, MPI_DOUBLE, top, 0,
3                  &A[0][0], COLS, MPI_DOUBLE, bottom, 0, MPI_COMM_WORLD, &s);
4  S1 MPI_Sendrecv(&A[1][0], COLS, MPI_DOUBLE, bottom, 1,
5                  &A[ROWS-1][0], COLS, MPI_DOUBLE, top, 1, MPI_COMM_WORLD, &s);
6     for(unsigned i = 1; i<ROWS-1; ++i)
7        for(unsigned j = 1; j<COLS-1; ++j)
8  S2       tmp[i][j] = A[i][j] + 1/4*(A[i+1][j]+A[i-1][j]+A[i][j-1]+A[i][j+1]);
9     double** ttemp=A; A=tmp; tmp=ttemp; // swap arrays
10 }
```

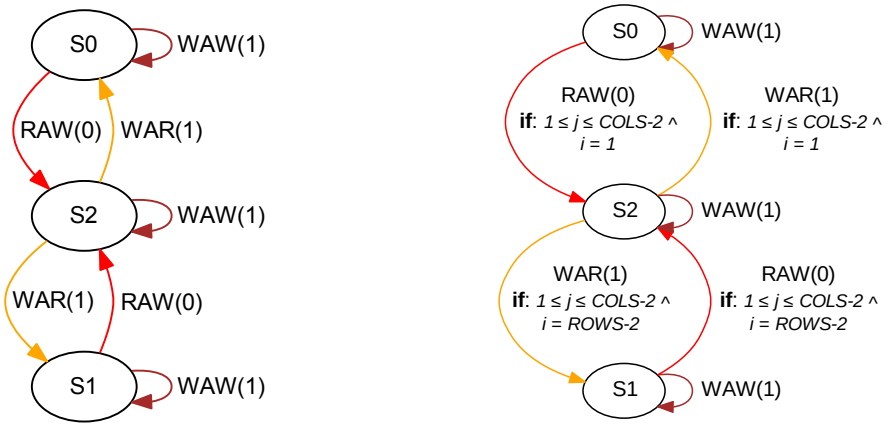
Listing 5.6: 5-points stencil code

5.3 Exact Dependence Analysis for Increased Communication Overlap

In the previous section we introduced a technique which improves cache behaviour for particular sizes of the transmitted data buffer. However, beyond the size of the cache, and in general for large messages, the rule of thumb is to use non-blocking communication routines and use as much as possible computations to hide the communication overhead. In this section we leverage the result of exact static dependence analysis, produced by the PM, to increase the opportunities for hiding communication costs for existing message passing programs.

5.3.1 Motivation and State of the Art

MPI programs often exhibit recurring code patterns which are direct consequences of the programming paradigm. For example, many programs read the data right after receiving it from a peer process by iterating over the received array elements. Similarly, data is usually sent right after the sender process finishes the computation that writes to array elements being transmitted. A relevant example is represented by a standard parallelization of a 5-point stencil code depicted in Listing 5.6. Stencil codes are very important in computational sciences and we show here a common way to parallelize such a code [115]. We have communication statements at the beginning of the loop, statements `S0` and `S1`, which exchange data being computed in the previous iteration. Right after the communication is performed, data is updated by a computational loop, statement `S2`. In both case the compiler determines a *true*, or RAW (see Section 2.3.3), data dependence on the elements of array `A` from statement `S0` to `S2` and between `S1` and `S2`.



(a) Statement-based DDG

(b) Instance-based DDG

FIGURE 5.7: DDG for 5-points stencil code in Listing 5.6

Traditional compiler analyses usually derive dependence information on a per-statement basis. For the 5-point stencil code in Listing 5.6 the DDG built by classical data dependence analysis [116] is represented in Figure 5.7(a). We neglect, in this analysis, the swap statements in line 10 since it introduces data dependencies between successive iterations of the `iter` loop which are irrelevant since our focus is in maximizing the overlap within the loop body.

Each dependence type is associated with a *distance vector* represented in brackets which, in the case of loop-independent dependencies, is zero. We see that there are two, loop-independent, RAW dependencies from statement `S0` to `S2` and between `S1` and `S2`, respectively. This is caused by the receive operation (implicit in the `MPI_Sendrecv` routine) writing elements of the array `A`. More precisely, the receive operation in `S0` writes `A`'s array elements in the range $A[0][0 : COLS)$. Same elements which are going to be read later in the first iteration of the stencil loop – and thus Read-After-Write – by statement `S2`. Although correct,these results are too conservative and coarse grained *inhibiting any kind of automatic optimization.* As a matter of fact, every dependence in the DDG exists for all the dynamic executions, or instances, of interested statements, however this is not the case. For example the dependence between `S0` and `S2` only applies to the first iteration of the stencil loop, all the remaining dynamic executions of statement `S2` are not dependent on `S0`. Similar considerations can be done for statement `S1`, for which the data dependence applies solely to the last iteration of the stencil loop.

The PM enables novel data analysis and transformation techniques by representing dependencies at the finest detail by an instance-based fashion. This technique is also referred to as *exact* data dependence analysis [101]. This allow a compiler to relax some of the constraints and apply more aggressive transformations at the array element level

which would not be supported by a more coarse level of analysis at the object level. An example of the dependence polyhedron for the stencil code is shown in Figure 5.7(b). The graph contains the exact same key dependencies but it carries more information for each of them. An *expression predicate* states which subset of the statement instances are affected by the dependence. When the predicate is missing, then the dependence applies to every instance of that couple of statements. For example, the non loop-carried RAW dependence between statements `S0` and `S2` exists for all the instances of `S2` where iterator `i` is 1 and `j` is between 1 and `COLS-2` inclusive. This means that the remaining instance of the stencil loop are not dependent on the communication statements and therefore can be used to hide communication costs.

5.3.2 MPI Semantics in the PM

In this section we describe how the semantics of many MPI routines can be described within the PM so that code regions containing MPI statements can be understood by the compiler as standard SCoPs. The semantics of MPI functions is fully specified in the MPI standard [3] and, in many cases, the effects of communication routines on the input program can be described within the limitations imposed by the PM.

We divide this section into 2 parts respectively considering: (*i*) point-to-point and (*ii*) collective communications. While presenting point-to-point and collective routines, we assume, the data-type to be of `MPI_BYTE` size. The model can be extended to take into account datatypes, but this aspect is not dealt with in this work.

5.3.2.1 Point-to-Point Communication

In MPI the semantics of point-to-point communications are slightly different from the one presented in Section 2.2.1. In MPI, channels are not *explicit*. However the concept of communication channel (see Definition 15) is implicit within communication routines. In MPI, a message channel can be identified by the quadruple `(SRC,DEST,TAG,COMM)`. Where `SRC` and `DEST` are the ranks of respectively the sender and the receiver process; the `TAG` identifies one of the available channels and the `COMM` defines the context on which this communication occurs. Communications between different contexts are not allowed. Channels exist within a unique communication context, therefore the `TAG` is used only to refer to channels within the current context, i.e., `COMM`. Also the *rank* of a process depends on the context, meaning that a process can have assigned a different rank depending on the `COMM` object on which the channel operates. MPI guarantees that the order of messages transmitted on the same channel is preserved at the receiver side.

We recall the signature of MPI's send operation, see Appendix A for a more detailed description:

```
MPI_Send(buff,n,dtype,dest,tag,comm)
```

The function takes an array starting from *buff* address and sends to the destination process, identified with *dest*, a number of n elements. The type, and thus the size of each element being sent, is given by the *dtype* variable. As previously stated, we assume for simplicity the type to be `MPI_BYTE`. The semantics of the `MPI_Send` statement can be represented as follows:

$$\mathbf{USE}(buff[i]) : \forall i \mid 0 \leq i < n$$

This is equivalent to a `FOR` statement accessing the first n elements of an array *buff*, which perfectly fits within the constraints of the PM. Note that this is a simplification of the actual behaviour of MPI implementation. An MPI library could decide to traverse the elements of the array with a different ordering. In our model an ordering is not fixed, the equation states that after the send operation n elements of the *buff* array will be accessed in *read mode*. No dependencies are imposed between those accesses.

Similar consideration can be done for the `MPI_Recv` routine. Its signature is defined as follows:

```
MPI_Recv(buff,n,dtype,src,tag,comm,stat)
```

In the PM, a receive operation can be represented as a write operation (or definition) to the first n elements of the *buff* array, additionally the update of *stat* object can be captured as follows:

$$\mathbf{DEF}(buff[i]) : \forall i \mid 0 \leq i < n;\ \mathbf{DEF}(stat[0])$$

It is worth noting that we do not capture the uses of the n, *dtype*, *src* and *tag* variables since the C language implements a pass-by-value semantics for function arguments.

5.3.2.2 Capturing Communication Channel Semantics

While the introduced equations describe the input/output behaviour of the two basic MPI routines, they miss to capture the semantics of the underlying communication channel. Send and receive routines are not pure *stateless* functions, the MPI standard *guarantees* that consecutive messages being sent to the same destination process, through the same communicator, using the same message tag will be delivered in the same order

```
1 MPI_Send(&a, 2, MPI_INT, 1, 0, comm);                    S1
2 MPI_Send(&b, 1, MPI_DOUBLE, 1, 0, comm);                 S2
3 ...
4 MPI_Recv(&a, 2, MPI_INT, 0, 0, comm, s);                 S3
5 MPI_Recv(&b, 1, MPI_DOUBLE, 0, 0, comm, s+1);            S4
```

LISTING 5.7: Example demonstrating MPI's channel semantics

$\{\langle Send, r, t_1, c, s_1\rangle\} \sqcap \{\langle Send, r, t_2, c, s_2\rangle\ \}$		
	$t_2 \in \mathbb{C}onst$	$t_2 \in \mathbb{V}ar$
$t_1 \in \mathbb{C}onst$	**if** $t_1 \sqcap_c t_2 = \top \Rightarrow$ $\{\langle Send, r, t_1, c, s_1\rangle, \langle Send, r, t_2, c, s_2\rangle\}$ **else** $\Rightarrow \{\langle Send, r, t_1 \sqcap_c t_2, c, s_1 \cup s_2\rangle\}$	$\{\langle Send, r, \bot, c, s_1 \cup s_2\rangle\}$
$t_1 \in \mathbb{V}ar$	$\{\langle Send, r, \bot, c, s_1 \cup s_2\rangle\}$	$\{\langle Send, r, t_1 \sqcap_v t_2, c, s_1 \cup s_2\rangle\}$

TABLE 5.2: Definition of meet operator, i.e., $\sqcap$, for *channel analysis*.

of issuing. The triplet `(RANK,TAG,COMM)` can be used to refer to an MPI communication *channel* given a process execution context. An example of the importance of statically capturing channel semantics is depicted in Listing 5.7. MPI guarantees that the message sent from statement `S1` is received by `S3` and, at the same way, `S2` and `S4` match. Any static compiler analysis or transformations should keep the semantics unchanged.

Unfortunately, this important dependence is not correctly captured by our PM representation since there is no explicit dependence between statements `S1` and `S2` which states that the receiver is expecting the messages issued with a specific ordering. This is solved by explicitly adding an artificial data dependence (on a ghost variable) between statements depending on the semantics of the underlying communication channel. For example consider the message passing code in Listing 5.7. The two statements `S1` and `S2` have a channel dependence. In the PM representation we can state this dependence by introducing a use immediately followed by a definition of a new variable associated to the communication channel:

$$\mathcal{D}_{S1} := \mathbf{USE}(buff[i]) : \forall i \mid 0 \leq i < n; \mathbf{USE}(chn_0) \prec \mathbf{DEF}(chn_0)$$

A similar representation is used for statement `S2` leading to RAW dependence on variable chn_0 which the transformation module cannot automatically break. A transformation who wants to split and rearrange the schedules of these two statements would need to manually handle the RAW dependence in order to maintain the channel semantics.

Determining MPI statements depending on same channel is trivial when the value of the destination/source rank, message tag and communicator are compile time constants. However, the problem becomes more complex, in the general case, when variables are

utilized and the control flow of the program is not trivial (e.g., no control flow statements within the code). This problem can be partially resolved by employing *constant propagation* to replace constant variables with their assigned value. However, a specific analysis is required to capture dependencies among MPI communication statements due to the channel semantics. We achieve this by using classical data-flow analysis techniques [102] presented in Appendix D. We derive an analysis which states, for each MPI statement, whether the statement has an additional dependence on the communication channel. When this dependence exists, the identifiers of directly dependent statements are returned by the analysis. Because we are interested in capturing such *channel dependencies* within SCoPs boundaries, the analysis is limited to those code regions, therefore there is no need for inter-procedural dataflow analysis as functions have been already inlined before the SCoP analysis.

At a specific program point, the information of dependent communication statements are propagated using a dataflow variable (see Appendix D) *Channel* which is a set of tuples defined as follows:

$$
\begin{aligned}
Channel = \langle \mathbf{x}, \mathbf{rank}, \mathbf{tag}, \mathbf{comm}, \mathbf{DepStmts} \rangle \quad & where: \\
\mathbf{x} &\in \{Send, Recv\} \\
\mathbf{rank}, \mathbf{tag}, \mathbf{comm} &\in \{\bot, c \in \mathbb{C}onst, v \in \mathbb{V}ar\} \\
\mathbf{DepStmts} &\subset \mathbb{N}
\end{aligned}
$$

The value of the destination/source *rank*, message *tag*, and *comm* can be either a compile constant, c, a program variable, v, or the $\bot$ symbol used to indicate that the corresponding *rank*, *tag* and *comm* is not known and can therefore be an arbitrary value. The *DepStmts* set stores the identifiers of MPI communication statements which are reaching at the entry of the CFG block (see Definition 35) containing a particular communication statement. Communication statements are identified by the enclosing CFG block identifier. Since we work with a non-separable dataflow framework (see Appendix D), the CFG is built in such a way that blocks can only contain one statement. We also assume, as pre-condition, the availability of *alias* analysis [102] and therefore the existence of an $isAliasOf(a, b)$ predicate which states whether variable a is an alias for variable b. The dataflow problem for the channel analysis is defined on the power-set of the *Channel* set (i.e., $2^{Channel}$) and the meet (or confluence), $\sqcap$, operator is defined as follows:

$$\{\langle Send, r_s, t_s, c_s, s_1 \rangle\} \sqcap \{\langle Recv, r_r, t_r, c_r, s_2 \rangle\} = \{\langle Send, r_s, t_s, c_s, s_1, \rangle, \langle Recv, r_r, t_r, c_r, s_2 \rangle\}$$

Moreover, the outcome of the confluence operator applied to two send operations targeting the same rank and communicator, but using different message tags, is depicted in Table 5.2. Operators $\sqcap_c$ and $\sqcap_v$ are defined as follows:

$$c_1 \sqcap_c c_2 = \begin{cases} c_1 & \text{if } c_1 = c_2 \\ \bot & \text{if } c_1 = \bot \vee c_2 = \bot \\ \top & \text{otherwise} \end{cases}$$

$$v_1 \sqcap_v v_2 = \begin{cases} v_1 & \text{if } v_1 = v_2 \vee \mathit{isAliasOf}(v_1, v_2) \\ \bot & \text{otherwise} \end{cases}$$

The $\sqcap_c$ and $\sqcap_v$ operators determine whether the constants or variables used to address a channel refer to the same value (or channel instance). In that case the constant value, or the variable addressing the channel is returned. When instead we can statically determine that the values being merged are different, the $\top$ symbol is returned. In the cases where a decision cannot be made statically, the $\bot$ symbol is used. For example two variables which are not in an aliasing relation may or may not refer to the same communication channel, therefore a safe assumption must be taken stating that there might be a channel dependence between the two statements. Note that since $\bot$ is the *greatest lower bound* (glb) of the lattice corresponding to this problem (see Appendix D). In Table 5.2 the $\bot$ value has been neglected as input value for t_1 and t_2 since the resulting value of the merge operation is always $\langle \mathit{Send}, r, \bot, c, s_1 \cup s_2 \rangle$. The same merge operator is utilized to aggregate *rank* and *comm* values.

For simplicity, let us define a boolean predicate, $\mathit{collide}(\mathit{ch}_1, \mathit{ch}_2)$, which given two tuples $\mathit{ch}_1, \mathit{ch}_2 \in \mathit{Channel}$ returns true when the two channels may refer to the same channel:

$$\mathit{collide}(\langle \mathit{Send}, r_1, t_1, c_1, \_ \rangle, \langle \mathit{Send}, r_2, t_2, c_2, \_ \rangle) = \begin{cases} \mathit{true} & \text{if } r_1 \sqcap r_2 \in \{\bot, r_1\}, \\ & \quad t_1 \sqcap t_2 \in \{\bot, t_1\}, \\ & \quad c_1 \sqcap c_2 \in \{\bot, c_1\}, \\ \mathit{false} & \text{otherwise} \end{cases}$$

The four component of the non-separable analysis framework, as described in Appendix D, are defined as follows:

$$ConstGen_n = \emptyset$$

$$DepGen_n(x) = \begin{cases} \{\langle Send, r \sqcap r_1, t \sqcap t_2, c \sqcap c_1, \{n\} \cup s_1 \rangle\} & \text{if } n \text{ is } send(\ldots, r, t, c) \wedge \\ & \exists\, ch := \langle Send, r_1, t_1, c_1, s_1 \rangle \in x \\ & collide(ch, \langle Send, r, t, c, \_ \rangle) \\ \{\langle Send, r, t, c, \{n\} \rangle\} & \text{if } n \text{ is } send(\ldots, r, t, c) \wedge \\ & \forall\, ch := \langle Send, r_1, t_1, c_1, s_1 \rangle \in x \\ & \neg\, collide(ch, \langle Send, r, t, c, \_ \rangle) \\ \emptyset & \text{otherwise} \end{cases}$$

$$ConstKill_n(x) = \emptyset$$

$$DepKill_n(x) = \begin{cases} \{\langle Send, r_1, t_1, c_1, s_1 \rangle\} & \text{if } n \text{ is } send(\ldots, r, t, c) \wedge \\ & \exists\, ch := \langle Send, r_1, t_1, c_1, s_1 \rangle \in x, \\ & collide(ch, \langle Send, r, t, c, \_ \rangle) \\ \emptyset & \text{otherwise} \end{cases}$$

In order to explain the equations, we present an example, in Listing 5.8. The CFG annotated with the value of the dataflow variables generated by the channel analysis is depicted in Figure 5.8. Dataflow values are generated (thanks to the $DepGen_n$ component), for statements `S1`, `S2`, `S3`, `S4`, `S5` and `S6`. At the meet point of the CFG (in line 9) the merge operator, i.e., $\sqcap$ is used to merge the values coming from the two preceding blocks. The channel dependencies are listed as the last component of the generated tuples. Statement `S2` is dependent on `S1` on the *send* operation. At the same way, there is a dependence (on a *recv*) from statement `S4` to `S5`. At the merge point of the CFG a $\perp$ value is generated since $a \sqcap 1 = \perp$. Note that this only applies for the channel with *tag* value 0. Next *send* statement, `S7`, is then recognized to have a channel dependence with statement `S3` since both operations targets the same destination and use same *tag* and *comm* values. It is worth noting that the analysis does not understand the parallel execution of the code, meaning that an ordering is enforced also for communications executing in parallel by distinct processes. This has the effect of generating more dependences than needed which however are semantically correct.

5.3.2.3 Collective Routines

Until now, collective routines have been largely overlooked for compiler optimizations. However they are widely used in production codes because of their performance. We describe how the semantics of a subset of the collective routines present in MPI can be

```
1  if (rank==0) {
2      MPI_Send(&a, 2, MPI_INT, 1, 0, c);
3      MPI_Send(&b, 1, MPI_DOUBLE, 1, 0, c);
4      MPI_Send(&a, 1, MPI_INT, 2, 1, c);
5  } else if (rank==1) {
6      MPI_Recv(&a, 2, MPI_INT, 0, 0, c);
7      MPI_Recv(&b, 1, MPI_DOUBLE, 0, 0, c);
8      MPI_Send(&a, 1, MPI_INT, a, 0, c);
9  }
10 if (rank==0)
11     MPI_Send(&b, 1, MPI_DOUBLE, 2, 1, c);
12 ...
```

LISTING 5.8: Example demonstrating MPI's channel semantics

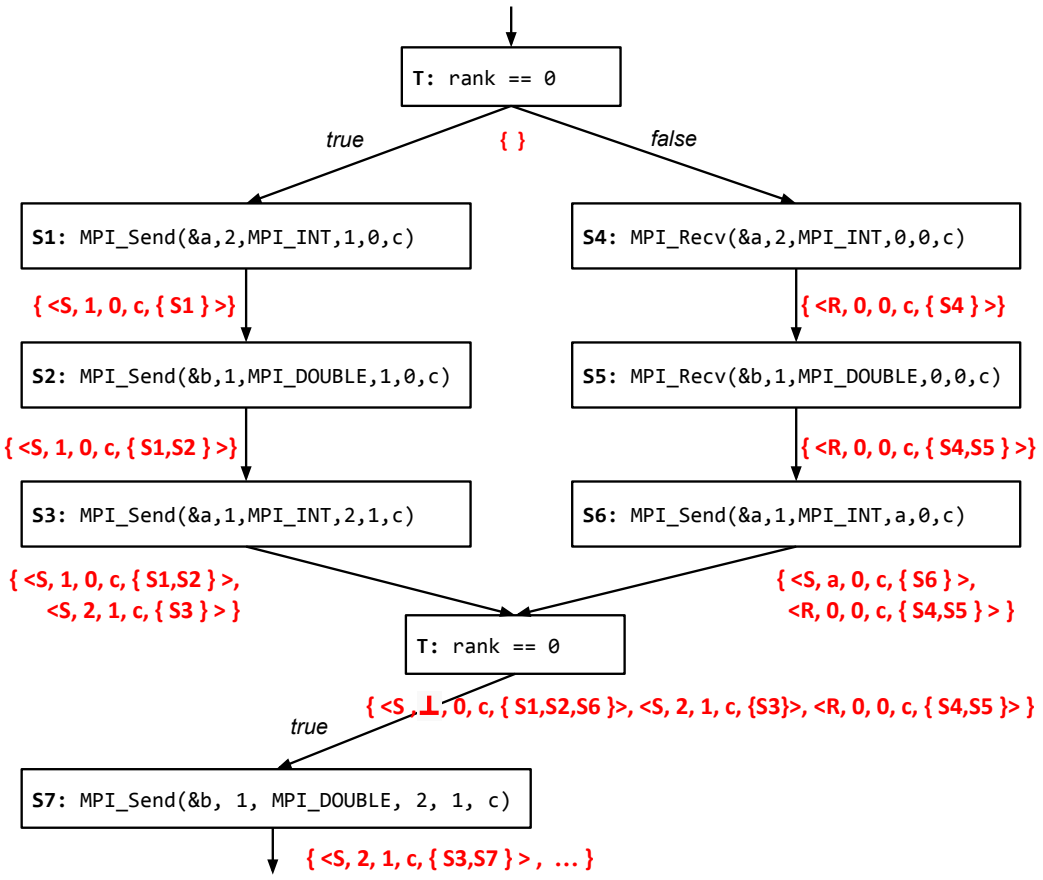


FIGURE 5.8: CFG of Listing 5.8 with annotated, at each CFG block's exit point, the values of the dataflow variables generated by the channel analysis.

described in the PM highlighting the limitations of the approach. Similar to what we did for the point-to-point routines, we consider message types of bytes.

It is important to note that compared to collective routines described in Section 2.2.2, in the MPI standard, these operations work within a communication context (or *communicator*). As described in Appendix A, all processes belonging to the same communication

```
1 if (rank==0) {
2    for (unsigned i=0; i<N; ++i) { def(buff[i]); }                    S1
3    MPI_Bcast(buff,N,MPI_BYTE,0,comm);                                S2
4 } else {
5    MPI_Bcast(buff,N,MPI_BYTE,0,comm);                                S3
6 }
7 ...
8 for (unsigned i=0; i<N; ++i) { use(buff[i]); }                       S4
```

LISTING 5.9: Example of MPI code using broadcast operation

context must participate to a collective routine, and moreover, the operations are always blocking. However, for the sake of this thesis, we largely consider programs using a unique communication context (i.e., `MPI_COMM_WORLD`). The semantics of the collective operations is therefore the same as described in Section 2.2.2.

The *bcast* operation (see Definition 23) is a 1-to-N collective routine where a *root* process sends to all other participating processes a number of data elements. Its signature follows:

```
MPI_Bcast(buff,n,dtype,root,comm)
```

The main issue with most of MPI's collective routines is that their behaviour depend on the process *id* (or *rank*) value which is known only at runtime. For example, in the broadcast operation, the process whose *rank* is equal to *root* reads n consecutive elements of array *buff*. All the remaining processes, i.e., $\mathcal{P} - p_{root}$, write the same range of elements in the *buff* array (see Definition 23). In general, the *bcast*'s semantics can be encoded in the PM as follows:

$$\mathbf{USE}(buff[i]): \ \forall i \mid 0 \leq i < n \wedge rank = root$$
$$\mathbf{DEF}(buff[i]): \ \forall i \mid 0 \leq i < n \wedge rank \neq root$$

In order for the formula to be evaluated at compile time, we assume that a variable *rank*, containing the runtime value of the process identifier (within the communicator *comm*), is available at the entry point of the SCoP. If this is not the case a call to `MPI_Comm_rank` routine shall be automatically inserted by the compiler. It is also important that eventual control flow statements in the SCoP, which depend on the process rank value, are based on the same *rank* variable used to describe semantics of the MPI routines. Also aliases to that variable must be rewritten in terms of the *rank* variable. We assume for now that this is true for the codes we are interested in. A compiler pass which is designed to deal with such problem, called *rank propagation* is presented later in Section 5.4.3.1.

Let us consider the code in Listing 5.9. Process $rank := 0$ initializes the *buff* array before broadcasting the value to its peers. By intersecting of the iteration domains to the access functions, $\mathcal{A}$, associated to each statement of the SCoP, we obtain the following equations:

$$
\begin{aligned}
\mathcal{A}_{S_1} =& \{\mathbf{DEF}(buff[i]) : \ i \mid 0 \leq i < N \wedge rank = \mathbf{0}\} \\
\mathcal{A}_{S_2} =& \{\mathbf{USE}(buff[i]) : \ i \mid 0 \leq i < N \wedge rank = \mathbf{0} \wedge rank = \mathbf{0}\}; \\
& \{\mathbf{DEF}(buff[i]) : \ i \mid 0 \leq i < N \wedge rank \neq \mathbf{0} \wedge rank = \mathbf{0}\} \\
=& \{\mathbf{USE}(buff[i]) : \ i \mid 0 \leq i < N \wedge rank = \mathbf{0}\} \\
\mathcal{A}_{S_3} =& \{\mathbf{USE}(buff[i]) : \ i \mid 0 \leq i < N \wedge rank = \mathbf{0} \wedge rank \neq \mathbf{0}\}; \\
& \{\mathbf{DEF}(buff[i]) : \ i \mid 0 \leq i < N \wedge rank \neq \mathbf{0} \wedge rank \neq \mathbf{0}\} \\
=& \{\mathbf{DEF}(buff[i]) : \ i \mid 0 \leq i < N \wedge rank \neq \mathbf{0}\} \\
\mathcal{A}_{S_4} =& \{\mathbf{USE}(buff[i]) : \ i \mid 0 \leq i < N\}
\end{aligned}
$$

After the conversion, the analysis modules which computes data dependencies based on the PM representation can determine that a RAW data dependence exists between statements `S1` and `S4`, but no RAW dependence exists between `S2` and `S4` (since both operations use of the *buff* array). This means that the transformation modules of a compiler can allow the minimization of the dependence distance by for example fusing `S1` and `S4` as shown in Listing 5.10.

It is worth noting that the transformation would not be valid if the variable used to drive the control flow of the program is not the same utilized in the PM representation. This knowledge allows us to statically remove the write access to the *buff* array (by `S1`) within the $rank = 0$ CFG branch. For most MPI programs we can rely on the fact that the variable used to store the current process rank is consistent within the program. However, in the general case aliasing can occur and known dataflow analysis techniques must be employed to rewrite all control flow expressions which takes into account the process rank on the base of the same rank variable. Such analysis is presented in Section 5.4.3.1.

scatter (see Definition 24) and *gather* (see Definition 25) routines have respectively a 1-to-N and N-to-1 collective semantics. Unlike the *bcast* operation, a *scatter* splits the given buffer into chunks of equal size and distributes chunk p to process rank p. The signature of `MPI_Scatter` is the following:

```
MPI_Scatter(sbuff,scount,stype,rbuff,rcount,rtype,root,comm)
```

```
1  if (rank==0) {
2     for (unsigned i=0; i<N; ++i) {
3        def(buff[i]);                                                    S1
4        use(buff[i]);                                                    S4
5     }
6     MPI_Bcast(buff,N,MPI_BYTE,0,comm);                                  S2
7  } else {
8     MPI_Bcast(buff,N,MPI_BYTE,0,comm);                                  S3
9     for (unsigned i=0; i<N; ++i) { use(buff[i]); }                      S4
10 }
```

LISTING 5.10: Optimized MPI code with minimized distance between RAW dependencies

Similar to the *bcast* operation, the behavior of this routine depends on the value of the process rank invoking the operation. The *root* process reads $scount * nprocs$ element from the *sbuf* array. For all (including the *root*), *rcount* elements are written to the *rbuff* array. Note that the behaviour of MPI's scatter routine is slightly different from the one described in Definition 24. In fact, in MPI two arrays are provided as send and receive buffers and the *root* process writes element to the receive buffer (i.e., *rbuff*). In the PM the access functions for this operation is represented with the following constraints:

$$\mathbf{USE}(sbuff[p][i]) : \forall\ p, i \mid 0 \leq p < np \wedge 0 \leq i < scount \wedge rank = root$$
$$\mathbf{DEF}(rbuff[i]) : \forall\ i \mid 0 \leq i < rcount$$

Beside *rank*, the polyhedral equations require the existence of another variable which must be available at compile time, $np := |\mathcal{P}|$ which represents the number of processes belonging to the communicator object *comm* utilized by the analyzed collective operation. Similar to the *rank* variable, a call to `MPI_Comm_size` shall be inserted at the entry point of a SCoP to obtain a reference to its value.

One additional step is required to guarantee that analysis and transformations on such representation are semantically correct. The *scatter*/*gather* functions accept the *sbuff* array as a one dimensional array but internally we use a 2-dimensional accessing scheme to address its elements. This might be non consistent with the way the buffer is utilized outside the MPI routine and we need to keep the access in a consistent form in order for the PM to understand whether two accesses (e.g., `A[5], A[2][1]`) address the same memory cell and therefore could lead to a data dependence. In our approach, the multi-dimensional accesses are linearized before being feed into the PM. Therefore access $sbuff[p][i]$ is rewritten as $sbuff[p * np + i]$. Because of the limitation of the PM, i.e., non-affine access function are not supported, *np* must be a compiler time constant. This is a limitation imposed by the current state of art of the libraries supporting the

Algorithm 4 Transformation flow for maximizing communication/computation overlap

```
1: Input: P = Syntax Tree of the input program; MOD = modified AST
2: Output: T = Syntax Tree of the transformed program
3: procedure MAXIMIZE_OVERLAPP(P : input, T : output)
4:     T = P
5:     repeat
6:         MOD = false;
7:         G = extractDDG(T)
8:         for all dep ∈ G do
9:             if dist(dep) = 0 ∧ src(dep) is MPI routine ∧ sink(dep) is loop body ∧ dep applies
   to a subset of stmt instances then
10:                T = applyLoopFission(T, sink(dep), findCut(dep));
11:                MOD = true
12:            end if
13:        end for
14:    until MOD is false
15:    for all dep ∈ G do
16:        if dist(dep) = 0 ∧ src(dep) is MPI routine ∧ sink(dep) is loop body then
17:            {COMM_STMT, WAIT_STMT} = toAsynchronous(src(dep))
18:            T = removeStmt(T, src(dep))
19:            T = moveToEarliestSchedule(T, COMM_STMT)
20:            T = moveToLatestSchedule(T, {WAIT_STMT, sink(dep)})
21:        end if
22:    end for
23: end procedure
```

PM representation. However simplifications exists which could allow handling of such expressions (for example forcing the access through the all $sbuf$ array) [117].

In the same way the support for several other MPI collective routines such as `MPI_Gather`, `MPI_Reduce` can be encoded.

5.3.3 Implementation and Evaluation

In this section we propose a compiler transformation which based on the result of the instance based data dependence analysis obtained by the PM, maximizes the communication/computation overlap by accordingly transforming the input program. In this work we focus on point-to-point communication routines.

5.3.3.1 Implementation

The approach is implemented in the *Insieme Compiler and Runtime* infrastructure presented in Appendix C. The Insieme Compiler fully integrates the PM analysis and transformations and provides a foundation for source-to-source program optimization.

5.3.3.2 Normal Form

Before applying any transformation, the input code is pre-processed into a normal form. In this form, an MPI program only contains `MPI_Send` and `MPI_Recv` statements so that successive steps of the analysis process are simplified. It is worth noting that the normalized program could have different buffering requirements and therefore may lead to deadlocks if executed. For example rewriting a non-blocking send as `MPI_Send` may cause, if sends are present in every parallel branch of the CFG, a deadlock for message size which are larger than the eager limit threshold. However, the program is kept in this normalized form only for the sake of performing static analysis. The shape of an MPI program in normal form is described by the following rules:

- Non-blocking point-to-point operations are rewritten to use the corresponding blocking version. This is obtained by replacing every asynchronous routine with the synchronous counterpart and by removing every `MPI_Wait` statement in the input code. Optimized, non-blocking code is generated by the code transformation described in Algorithm 4.
- `MPI_Sendrecv` operations are split into the corresponding `MPI_Send` and `MPI_Recv` operations.
- `MPI_Ssend`, `MPI_Rsend` or `MPI_Bsend` are rewritten to plain `MPI_Send`.

Once the program is in normal form, we perform SCoP analysis to obtain a semantically equivalent representation of the code region based on the PM. From this representation we proceed with the extraction of the data dependence polyhedron.

5.3.3.3 Code transformation

Once the instance-based DDG is generated, we apply a sequence of transformations as described in Algorithm 4. The idea is to iterate through all the loop-independent dependencies which have an MPI communication statement as the source and a loop body as sink. If the dependence applies to a subset of the instances of the sink then we split the loop, applying the loop fission transformation [116], at the range provided by the dependence analysis. In this way the iterations which are dependent on the MPI communication statement are isolated into a new loop statement. Note that fission is possible as long as there are no dependencies in the loop body that conflict with the transformation being applied. The transformation framework in the Insieme Compiler implements a pre-condition analysis which determine whether a transformation can be safely applied.

```
1  for(unsigned iter=0; iter<NUM_ITERS; iter++) {
2     MPI_Request __req0, __req1;
3     MPI_Irecv(&A[0][0],COLS,MPI_DOUBLE,bottom,0,com,&__req0);
4     MPI_Irecv(&A[ROWS-1][0],COLS,MPI_DOUBLE,top,1,com,&__req1);
5     MPI_Send(&A[1][0],COLS,MPI_DOUBLE,bottom,1,com);
6     MPI_Send(&A[ROWS-2][0],COLS,MPI_DOUBLE,top,0,com);
7     // stencil loop after fission
8     for(unsigned i = 2; i<ROWS-2; ++i)
9        for(unsigned j = 1; j<COLS-1; ++j)
10          tmp[i][j] = A[i][j] + 1/4*(A[i+1][j]+A[i-1][j]+A[i][j-1]+A[i][j+1]);
11    MPI_Wait(&__req0, MPI_STATUS_IGNORE);
12    // first iteration of stencil
13    for(unsigned j = 1; j<COLS-1; ++j)
14          tmp[1][j] = A[1][j] + 1/4*(A[2][j]+A[0][j]+A[1][j-1]+A[1][j+1]);
15    MPI_Wait(&__req1, MPI_STATUS_IGNORE);
16    // last iteration of stencil loop
17    for(unsigned j = 1; j<COLS-1; ++j)
18          tmp[ROWS-2][j] = A[ROWS-2][j] + 1/4*(A[ROWS-1][j]+
19                        A[ROWS-3][j]+A[ROWS-2][j-1]+A[ROWS-2][j+1]);
20    double** ttemp=A; A=tmp; tmp=ttemp; // swap arrays
21 }
```

Listing 5.11: 5-points stencil code after code optimization

The procedure repeats until a fix-point is reached where every dependence in the DDG applies to all the instances of the source and sink statement. The next step is to consider all dependencies between communication statements and computational loops based on the transformed code. For each of them, the source of the dependence – the communication statement – is removed from the code and the corresponding asynchronous version of the routine is scheduled in its earliest position (which is determined by constraints in the DDG). Listing 5.11 shows the transformed stencil code from Listing 5.6. The receive is scheduled at the beginning of the loop body as shown in lines 3 and 4. The loop depending on the communication statement, i.e., the sink, is scheduled lazily prepending to it an `MPI_Wait` operation placed to preserve the semantics of the program, lines 11 – 19 of Listing 5.11. The remaining non-dependent loop iterations will be, by the end of the transformation, confined between the issuing of the asynchronous communication operations and the consumption of the received data (lines 8–10). Therefore maximizing the overlap window.

The transformation can be easily extended to take into account loop-carried dependencies, in that case the distance of the data dependence, d, defines the number of loop cycles which can be executed between the source and the sink of the dependence. This can be handled by automatically allocating an array of d request objects for each communication routine where the `MPI_Wait` statement of a request generated by a communication statement at iteration i occurs at iteration $i + d$. This transformation, also known as *software pipelining* [116], requires additional control code, therefore overhead,

VSC2				LEO3			
# of MPI	Original (in msecs.)	Transformed (in msecs.)	Improvement (in %)	# of MPI	Original (in msecs.)	Transformed (in msecs.)	Improvement (in %)
16	219	218	0.3	12	264.9	264.5	0.09
32	89.7	89.0	0.8	24	118.7	118.9	-0.01
64	35.1	32.0	9.5	48	37.4	37.0	0.9
128	20.1	17.9	12.6	96	21.0	20.2	4.0
256	13.1	11.5	13.5	192	11.3	9.8	15.3
512	12.0	9.3	27.9	384	7.6	6.4	19.1

TABLE 5.3: Evaluation of the transformed code on the VSC2 and LEO3 cluster, fixed problem size of 4Kx4K and `NUM_ITERS=10`

to be inserted by the compiler to correctly fill and unload the pipeline. A compiler can employ static heuristics in order to determine when software pipelining is beneficial for a given input code.

5.3.4 Evaluation

We tested the transformed 5-point stencil code, depicted in Listing 5.11, on the two production cluster LEO3 and VSC2 illustrated in Table 5.1 and compared its execution time with the original code shown in Listing 5.6.

The code has been executed keeping the problem size constant, 4K by 4K elements, and varying the number of MPI processes, results for both architectures are shown in Table 5.3. We see that, as expected, the transformed code has overall a better performance. Additionally, the improvement increases with the number of cores since the smaller problem slice assigned to each processor is, the more dominant the communication overhead becomes. Since our transformation aims at hiding communication costs, its benefit grows as the computation/communication ratio diminishes.

5.4 Static Matching of Communication Statements with Affine Domains

In the previous section we have seen that changing a massage passing program often goes beyond the manipulation of the single communication routines. The underlying channel semantics must be understood in order to maintain program semantics. In MPI, one of the implementation of the message passing paradigm, channels are not explicit within the point-to-point communication routines. This means that in order to automatically modify a message passing program, a compiler must reconstruct the missing knowledge by analyzing the input code.

In Section 5.3.2.2, we proposed an analysis which determines channel dependencies of communication statements. However, when a compiler decides to change a communication statements, e.g. by applying message coalescing, it must know which of the receive statements within a program needs to be updated in order to keep program semantics. The information of which receive operation matches a send operation (and vice-versa) is generally difficult to determine statically, in MPI, due to the non-deterministic behaviour which exists within the programming model.

In this section we elaborate a novel algorithm which delivers statement matching of MPI programs whose control-flow can be described within the constraints of the PM.

5.4.1 Background and Related Work

Several researchers investigated the matching problem in various forms. Most of those works were tailored for a specific constrained environment rather than a general analysis. Those works can be classified into three different categories: static matching, dynamic runtime matching, and dynamic post-mortem matching.

The majority of the approaches rely on post-mortem analysis of program traces to determine communication patterns (e.g. [118–121]). The main idea behind these tools is an automatic instrumentation of the original code and a tracing run on parallel system at the desired scale. First, any result gathered with this analysis may not be sound since it just represents one possible matching as opposed to all possible matchings that are required for program transformations. A second major drawback is the huge resource requirement because realistic executions often generate terabytes of tracing data which has to be analyzed for patterns. FACT [122] uses *program slicing* [123] to obtain a reduced program. Beside focusing on communication pattern detection, FACT also statically analyses the input program to match communication statements using simple heuristics.

Dynamic matching schemes commonly establish matches during runtime based on inspector-executor principles [9]. The obtained matching is also just one possible matching and it can only be used to transform the communication during the execution. We argue that source-code transformations that can be performed after static matching analyses are generally more powerful and incur less runtime overhead than dynamic matching methods.

A purely static matching algorithm based on Diophantine inequalities is proposed by [17]. However, this approach requires source and destination of messages to be compile time constants in order to correctly extract the system of inequalities. Since nearly all MPI

codes work with different numbers of processes where each process calculates communication partners based on the number of processes and its own process *id*, this method is substantially limited.

Source-code annotations, as described by [124] can be used to specify matches explicitly. However, they put an additional burden on the programmer and add a maintenance risk due to the duplication of semantics (later updates to the code may not update the annotations). Thus, such approaches are less effective in practice and deriving matches automatically remains most desirable.

More recently, symbolic static matching was formalized by [18], who introduces the concept of a parallel CFG (pCFG), defined as the Cartesian product of the CFGs of the processes interacting in a distributed memory program. A dataflow iterative framework which works on such pCFG enabling analysis of message passing programs using an iterative fix-point solver. This framework is generic in the sense that it can be used to discover several properties of distributed programs. As shown in [18], it can be employed to solve some instances of the static matching problem. However, wildcards and asynchronous communications are not handled (even under the constraint of interleaving-obliviousness) and communication statements inside loops pose an efficiency problem.

The matching problem also plays a role in semantic analysis of message-passing programs using *formal verifiers*. SPIN-MPI [125] extracts communication statements from an MPI source code and feeds it to the SPIN model checker. Since not all semantics of MPI communication statements are fully supported by SPIN, some simplifications are necessary (e.g., wildcards are not allowed). Another approach is ISP [126]. ISP builds a model from program traces collected from a single execution of the input code. Successively the verification scheduler plays all possible interleavings. The main drawback of formal verifiers is the exponential explosion of possible interleavings making them impractical to be used for large codes. This problem was investigated recently by [127] who proposed a novel technique based on a formal definition of the *matches-before* relation for MPI communications. By exploiting it, alternative non-deterministic matches can be detected dynamically for large node sets with a modest overhead.

All previous approaches have limitations in various regards. The biggest problems in the area of static matching are non-deterministic communication relations using wildcard (any source) receives and non-blocking static analysis is able to compute static matches in their presence. In this work, we propose a new method that uses the elements of the PM in novel ways to determine static matches for practical message passing programs.

5.4.2 Preconditions and the Compiler Framework

In this work we consider the semantics of MPI's point-to-point communication routines. Sends and receives can be either blocking or non-blocking. Non-blocking communications return a *request* object which can be used by the *wait* primitive to block until the data transfer is completed. Through the rest of the paper we make no distinction between the blocking and non-blocking semantics unless explicitly stated.

Our method relies on well-established analysis frameworks such as the PM and dataflow analysis, available in many mainstream compilers. The approach has been partially implemented within the Insieme compiler, see Appendix C, however it is presented in a general way so it can be reproduced in any compiler framework. In this Section we present the set of features that such compiler framework must provide. It is worth noting that the internal representation of the Insieme compiler, called INSPIRE, is not in Static Single Assignment (SSA) form, therefore the dataflow analyses here presented, which are based on such "close-to-source" program representation, can be simplified if the underlying compiler provides such abstraction.

Real message passing programs span across procedures and source files. In order to precisely reconstruct the semantics, the complete control flow of the program must be available to the compiler and thus to the analysis module. Therefore the underlying compiler must support inter-procedural analysis across translation units. *Whole program analysis* [102] is usually not supported by mainstream compilers for performance reasons. In compilers like GCC and LLVM such code analyses can be *optionally* enabled by means of Link Time Optimizations (LTO). Our source-to-source compiler infrastructure, Insieme, links call expressions to the actual function definitions as part of the compiler frontend which loads all translation units into memory and builds a single AST.

We require an iterative dataflow solver and a collection of classical dataflow analysis, e.g., *constant propagation*, *points-to* and *alias* analysis and *reaching definitions*. Since SCoP analysis is limited to functions, a *code inliner* is responsible of removing any function call from an input program. Recursion is therefore not supported. We propose the use of inlining since it is implemented in many mainstream compilers and it is an easily way to prepare the code for a successive *legacy* SCoP analysis. Additionally it is worth noting that this process is only done for the sake of collecting cardinality information associated to the communication statements, once the matching is established the inlined code can be discarded. In Insieme the analysis module builds SCoPs for every procedure, treating function calls as simple statements. Inlining is then performed by merging and manipulating the polytopes within the call graph.

A common practice which could inhibit the inliner is the use of static storage variables within functions. This problem is however solved in our compiler by a pre-processing phase which performs the *erasure* of global and static variables which are redeclared as regular variables in the main function.

We assume that PM support is available as part of the compiler. A SCoP analysis determines the largest non-overlapping regions subject to the constraints imposed by the PM. All statements in these regions are associated with information of their corresponding iteration domains. If the entire program cannot be covered by a single SCoP, then *reaching definitions* analysis is used to ensure the following: all read accesses of a parameter p in a set of SCoP regions $\{R1, \ldots, Rn\}$ are all reached by $def(p)$ - a unique definition of p dominating those regions. Moreover, we use the ISL library [128] for polytope analysis and manipulation and the Barvinok library [129] for computing the cardinality of the iteration domains.

5.4.3 The Message Matching Algorithm

The matching algorithm we introduce in this paper is composed of two major phases which are discussed in this section.

Phase 1 : The first phase, called *rank propagation*, makes sure that any variable v depending on the process *id*, or any other input parameter, is replaced by an expression returning the value of v at that specific program point. This transformation prepares the code to be analyzed for SCoPs and makes sure that the symbolic cardinality expressions associated to communication statements are functions of the process *id* and program input parameters only.

Phase 2 : The second phase of the analysis scans the input program for send and receive statements to generate a bipartite graph such that sends and receives are in different partitions. Each node of the graph is labeled with its cardinality and the source and target domains associated to the communication operation. Communication statements are then partitioned into independent program *regions*, this is done using the following steps:

1. Statements are selected on the basis of their position within the program control flow, a *tentative* region is formed.
2. Within this region a *preliminary* matching between communication statements is established based on iteration domain and source/target expressions of the communication statements.

3. Region boundaries are then verified by computing the maximum network flow of the graph. If there exist at least one flow which allows every instance of send statements to be matched by a receive operation, then we mark the statements as belonging to this region and repeat the procedure (from step 1) with successive statements.
4. Otherwise additional communication statements are added to the region and continue from step 2.

In general, a region may contain several valid flows, therefore they must be all considered in successive transformation phases. The result is an over approximation of the actual matches, however it guarantees that none of the existing matches are excluded.

5.4.3.1 Phase 1: Rank Propagation

In the first phase the value of every program variable v, which directly or indirectly depends on the process id, is replaced at the program point p by an expression, i.e., $exp(id, \ldots)$. We call v a *dependent variable*. This transformation is a prerequisite to recognize the control flow statements of an SPMD program which split the execution flow according to the process id. Moreover the dependence to the process id of variables utilized as source and target of communication statements is made explicit. As result, the SCoP analysis determines the *parallel control-flow* as part of the polytope's constraints. Beside the process id, the analysis also covers any variable which is recognized to be *invariant* with respect to the entire program, for example, the total number of processes, np, and any input argument of the application. In Listing 5.12, we show a non-trivial 1-D mesh neighbour communication pattern, implemented in MPI, used to explain the steps of the matching algorithm presented in this paper. To illustrate the concept of dependent variables, let us consider *even* which is used in the `IF` expression in line 7. Without considering the context, the SCoP analysis tags *even* as a parameter of the model meaning that its relation with the process id is not explicitly shown in the extracted inequalities for the communication statements. Moreover since the value of *even* is being written in line 3 and 4, the SCoP analysis fails to recognize the entire code segment as a single SCoP.

The proposed technique is similar to *program slicing* [123], but instead of being expressed as a backward analysis, which has to be repeated for every program variable within control expressions and communication statements, we define it as a *forward analysis* which focuses only on the subset of dependent variables. Moreover, if the program is already in SSA form, the dataflow analysis can be simplified. We present the analysis equations as implemented in the Insieme compiler.

```
1  MPI_Comm_rank(comm, &id);
2  MPI_Comm_size(comm, &np);
3  even = 0;                 // RankProp(n)={v2:<true,even,0>}
4  if (id%2==0) even = 1; // RankProp(n)={v3:<id%2==0,even,1>}
5  // meet: RankProp(n)={v4:<true,even,(id%2==0?1:0)>}
6  if (id>0 && id<np-1) {
7     int left = id-1, right = id+1;
8     if (even) {
9        MPI_Irecv(b, 1, MPI_INT, f(id), 0, MPI_COMM_WORLD, &req);          S0
10       MPI_Send(b, 1,  MPI_INT, g(id), 0, MPI_COMM_WORLD);                S1
11    } else {
12       MPI_Irecv(b, 1, MPI_INT, left, 0, MPI_COMM_WORLD, &req);           S2
13       if (right!=np-1)
14             MPI_Send(b, 1, MPI_INT, right, 0, MPI_COMM_WORLD);           S3
15    }
16    MPI_Wait(&req, MPI_STATUS_IGNORE);
17 } else
18    if (np>1 && id==0)
19          MPI_Send(b, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);                  S4
20    else if (np%2==0)
21          MPI_Recv(b, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD);     S5
```

LISTING 5.12: MPI running example

The *rank propagation* analysis is formally defined on top of the classic iterative dataflow framework [102]. Each block n of the CFG is associated with a *dataflow variable* structured in the following way $RankProp_n = \langle n, \mathcal{B}, v, e \rangle$. The first element, n, is the address (or the identifier) of the CFG block producing this dataflow variable. The second element is the domain $\mathcal{B}$, a generic a boolean conjunction of the control flow conditions encountered within the *program path* connecting the main entry point to block n. The third element of the tuple, v, is the variable identifier (i.e., $v \in \mathbb{V}ar$, where $\mathbb{V}ar$ is the set of program variables which are not loop iterators, i.e., $\mathbb{I}ter$) which is defined within block n. The last element, $e \in \{\mathbb{E}xpr, \bot, \top\}$ is an expression (with $\mathbb{E}xpr$ being the set of valid expressions) which computes the value of variable v at that specific program point, i.e., $v = e(id, \ldots)$. Two auxiliary symbols are introduced to represent additional states of the analysis. The $\top$ symbol represents the fact that the variable v is *undefined*. This means that the initial value for that variable is generating an *external function* which we represent in the analysis with $read()$. When instead an expression which solely relies on the parameters of the input program (e.g., no loop iterators, i.e., $\mathbb{I}ter$, are involved) cannot be formed, the $\bot$ value is utilized.

The dataflow problem for the rank propagation uses a semi-lattice $L = (2^{RankProp}, \sqcap)$ defined by the power-set of the $RankProp$ and a *meet* (or confluence), $\sqcap$, operator which aggregates dataflow values at meet points of the CFG. The dataflow value of block n is computed based on the dataflow information available at the exit point of the predecessor

nodes, $pred(n)$, using the following formula:

$$In_n = \begin{cases} \emptyset & \text{if } n \text{ is Start} \\ \bigsqcap_{p \in pred(n)} Out_p & \text{otherwise} \end{cases}$$
$$Out_n = f_n(In_n)$$

Where $In_n, Out_n \in L$ correspond to the dataflow information which are associated, respectively, to the entry and the exit of CFG block n. The framework iteratively solves the dataflow equations until the *maximum fix-point assignment* (MFP) is found. $f_n(In_n)$ is the transfer function which specifies which dataflow information is generated or killed by a generic CFG block n. Because the definition of the transfer function for our problem also depends on the value of the dataflow information at the entry of the block, we define the analysis on top of a *non-separable* dataflow framework [102]. Therefore $f_n(In_n)$ is defined as follows:

$$f_n(\mathsf{x}) = (\mathsf{x} - (ConstKill_n \cup DepKill_n(\mathsf{x}))) \cup (ConstGen_n \cup DepGen_n(\mathsf{x}))$$

Dependent parts in the Gen and $Kill$ sets make it difficult to summarize the effect of multiple statements in a flow function. Hence, basic blocks for non-separable analyses consist of *single statements*. For our rank propagation analysis the components are defined as follows:

$$ConstGen_n = \begin{cases} \{ \langle n, true, v, id \rangle \} & n \text{ is } \texttt{MPI\_Comm\_rank(...,\&v)} \\ \{ \langle n, true, v, np \rangle \} & n \text{ is } \texttt{MPI\_Comm\_size(...,\&v)} \\ \emptyset & \text{otherwise} \end{cases}$$
$$DepGen_n(\mathsf{x}) = \begin{cases} \{ \langle n, \mathcal{B}, v, eval(e, \mathsf{x}) \rangle \} & n \text{ is assignment } v = e \\ \{ \langle n, \mathcal{B}, v, \top \rangle \} & n \text{ is } v = read() \\ \emptyset & \text{otherwise} \end{cases}$$
$$ConstKill_n = \emptyset$$
$$DepKill_n(\mathsf{x}) = \begin{cases} \{ \langle m, \mathcal{B}, v, d \rangle \} & n \text{ is assign. } v = e,\ \langle m, \mathcal{B}, v, d \rangle \in \mathsf{x} \\ \{ \langle m, \mathcal{B}, v, d \rangle \} & n \text{ is } read(v),\ \langle m, \mathcal{B}, v, d \rangle \in \mathsf{x} \\ \emptyset & \text{otherwise} \end{cases}$$

For each visited block the analysis has to build the expression $\mathcal{B}$ by logically combining the control flow expressions within the dominator nodes. Since those expressions may contain dependent variables, the *eval* function is applied to the condition of a node p (where p dominates n) to replace every dependent variable with the dataflow information computed for node p. Hence, $\mathcal{B}$ may contain $\bot$.

The confluence operator, $\sqcap$, on elements in L is defined to propagate (as they are) the

dataflow values incoming from every CFG edge of a meet point. However, if the same variable is assigned within multiple paths, the meet operation is defined as follows:

$$\langle n_1, \mathcal{B}_1, v, e_1 \rangle \sqcap \langle n_2, \mathcal{B}_2, v, e_2 \rangle = \langle m, \mathcal{B}_1 \vee \mathcal{B}_2, v, e_1 \widehat{\sqcap} e_2 \rangle$$

Where $\widehat{\sqcap}$ is:

$$e_1 \widehat{\sqcap} e_2 = \begin{cases} e_1 & \text{if } \mathcal{B}_1 = \mathcal{B}_2 \wedge e_1 = e_2 \\ (\mathcal{B}_1 ?\ e_1 : (\mathcal{B}_2 ?\ e_2 : \top)) & \text{if } dom(n_1) \cap pdom(n_2) \neq \{\,\} \\ (\mathcal{B}_2 ?\ e_2 : (\mathcal{B}_1 ?\ e_1 : \top)) & \text{if } dom(n_1) \cap pdom(n_2) = \{\,\} \\ \bot & \text{if either } e_1, e_2, \mathcal{B}_1, \mathcal{B}_2 = \bot \end{cases}$$

This states that the value of v after the meet point is computed by a conditional expression (i.e., `?:`) combining contributions from incoming edges. The expression is built taking into account the positions at which the two definitions of v appeared. This is important since the order in which the conditions $\mathcal{B}_1$ and $\mathcal{B}_2$ should be evaluated changes accordingly. If the set of dominators of block n_1 (i.e., $dom(n_1)$) intersects the set of post-dominators of n_2 (i.e., $pdom(n_2)$) then block n_2 precedes n_1. In the other case either n_1 precedes n_2 or variable v is defined in both conditional branches of an `IF` statement, the latter makes the evaluation order of the boolean conditions not important.

An example of how the dataflow equations work for an actual code is shown, in Figure 5.9, for the variables *even*, $left$ and *right*. In line 2 of Listing 5.12, the dataflow analysis generates the variable $v2 : \langle true, even, 0 \rangle$. In line 3, *even* gets assigned the value 1 under the constraint that the value of *id* is even. The dataflow analysis stores this information in a new dataflow variable, $v3$, which contains the control flow condition for which this block is executed, e.g., `id%2==0`. $v2$ is killed within this control flow path. In line 4, at the meet point, the confluence operator is applied between $v2$ – coming from the $false$ path of the `IF` statement – and $v3$. The outcome is a new dataflow variable, $v4$, in which the value of *even* is expressed as a conditional expression which is a function of the process *id*. Since the value of *even* is not modified again, the variable $v3$ is propagated to the succeeding blocks. Any use of *even* will be replaced by the generated expression.

The **eval**(**e**, x) function returns the expression which computes the value of v based on other dependent variables (and constant values). A generic expression $e = exp(\ldots, e_i, \ldots)$ is an abstraction for any right-hand side of an assignment statement in the input code such that $e_i \in Opd(e) \subseteq (\mathbb{V}ar, \mathbb{I}ter)$. The function $Opd(e)$ extracts variables from any

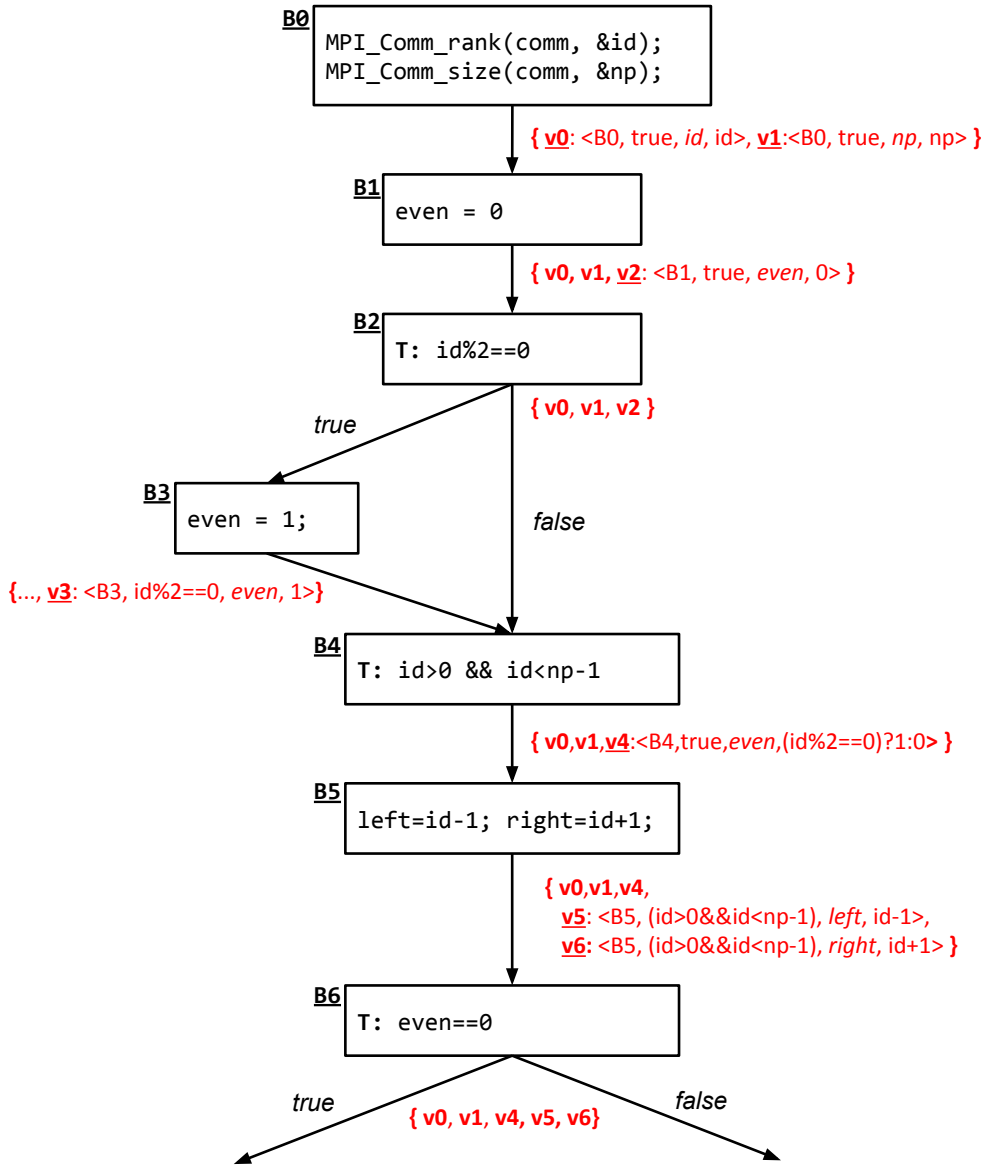


FIGURE 5.9: CFG and dataflow variables generated by the rank propagation analysis of Listing 5.12.

given expression e.

$$eval(e, \mathsf{x}) = \begin{cases} \bot & \text{if } val(e_i, \mathsf{x}) = \bot, e_i \in Opd(e) \\ exp(\ldots, e_i, \ldots) & \text{if } val(e_i, \mathsf{x}) = \top, e_i \in Opd(e) \\ exp(\ldots, val(e_i, \mathsf{x}), \ldots) & \text{otherwise} \end{cases}$$

$eval$ uses $\mathbf{val}(\mathbf{e_i}, \mathsf{x})$ to denote the value of a variable e_i (consisting of either a variable, $\mathbb{V}ar$, or a loop iterator, $\mathbb{I}ter$) in the context of the given dataflow information x:

$$val(e_i, \mathsf{x}) = \begin{cases} \bot & \text{if } e_i \in \mathbb{I}ter \\ d & \text{if } e_i \in \mathbb{V}ar, \langle m, \mathcal{B}, e_i, d \rangle \in \mathsf{x}, d \in (\mathbb{E}xpr, \top) \end{cases}$$

Succeeding to this analysis, a transformation pass replaces every dependent variable, which has been marked with neither $\top$ nor a $\bot$ with the computed expression at that program point. If any communication statement is within a control flow expression marked as $\bot$, then the analysis terminates since it will not be able to correctly compute iteration domains for the enclosed statements. This also invalidates the next phase of the matching analysis since a SCoP cannot be extracted. The effect of this phase to the code in Listing 5.12 is depicted by the arrows forwarding the value of dependent variables *even*, $left$, and *right* (omitted for readability reasons) through the various uses.

5.4.3.2 Phase 2: Bipartite Graph and Network Flow

After the first phase, SCoP analysis labels each communication statement of the input program with information of its iteration domain. If SCoP analysis is not able to encapsulate all of the communication statements of the given program within affine regions, then the matching algorithm gives up and stops here. Otherwise, we continue with the second phase during which we extract the communication statements and reorganize them into a *bipartite graph* $MG = (S, R, E)$ where S is the set of send statements, R is the set of receive statements and E is the set of edges, $e = (s, r)$, such as $\forall\ e \in E \mid s \in S \wedge r \in R$.

Each statement `S` has therefore three pieces of information attached:

- The iteration domain associated to `S`, i.e., $\mathcal{D}_{\mathtt{S}}$.
- The expression used to address source process *ids* in a receive statement (i.e., src) or target process *ids* in a send operation (i.e., trg). src and trg can be both non-affine expressions.
- The cardinality of statement `S`, i.e., $|\mathcal{D}_{\mathtt{S}}|$.

Since we want to represent data transfers in an homogeneous way, we use *polyhedral relations* [128, 130] to express the exchange of data between two process groups. A relation is a mapping from points of an input polytope to points in a target polytope. Relations can be described by the conjunction of a set of affine constraints. For example, a relation which maps every process $id \in [0 \dots np)$ to a target process 0 can be expressed as follows:

$$\mathcal{R} : [\, np \,] \Rightarrow \{id \rightarrow 0 \mid 0 \leq id < np\}$$

where np is a parameter (or free variable) of the relation. Relations are often used to represent data dependencies within the PM [128]. Two operations are available: $dom(\mathcal{R})$

returning the input domain to which the relation is applied (e.g., $\{\, id \,|\, 0 \leq id < np \,\}$), and $range(\mathcal{R})$ returning the points in the target domain (e.g., $\{0\}$). Elements which satisfy the relation are tuples mapping elements of the domain to the range, (e.g., $\{(0 \rightarrow 0), (1 \rightarrow 0), \ldots, (np-1 \rightarrow 0)\}$. We continue by showing how send and receive statements can be represented within this formalism.

Let us consider a number of send statements, S, surrounded by the following iteration domain $\mathcal{D}_S = \{\, id \,|\, 0 \leq id < 5 \,\}$ (i.e., the $dom(\mathcal{R})$), meaning that S is executed by all processes whose id is strictly smaller then 5. These data transfers can be represented in our formalism as follows:

$$
\begin{aligned}
send(buff, 5) &: \{\, id_s \rightarrow id_t \,|\, (0 \leq id_s < 5) \wedge id_t = 5 \,\} \\
send(buff, p) &: [\, p \,] \Rightarrow \{\, id_s \rightarrow id_t \,|\, 0 \leq id_s < 5 \wedge id_t = p \,\} \\
send(buff, id + 1) &: \{\, id_s \rightarrow id_t \,|\, 0 \leq id_s < 5 \wedge id_t = id_s + 1 \,\} \\
send(buff, i{:}lb..ub) &: [\, lb, ub \,] \Rightarrow \{\, id_s \rightarrow i \,|\, 0 \leq id_s < 5 \wedge lb \leq id_t < ub\} \\
send(buff, f_{na}(\ldots)) &: [\, np \,] \Rightarrow \{\, id_s \rightarrow id_t \,|\, 0 \leq id_s < 5 \wedge 0 \leq id_t < np \,\}
\end{aligned}
$$

We identify with $id_s \in dom(\mathcal{R})$ the set of processes which issue the send operation. $id_t \in range(\mathcal{R})$ is instead an iterator through the possible targets. Cases 1 and 2 are quite straightforward. Case 3 represents a typical neighbour communication. Case 4 describes the situation in which a loop iterator is used to define the target process. The semantics of the loop iterator is defined so that $lb \leq i < ub$, a *constant* stride is also allowed since it can be represented within the constraints of the PM. Case 5 shows the use of a non-affine function for selecting the destination of a message, (e.g., `A[B[i]]` or `id*np`).

Receives can can be modeled similarly. The main difference here is that receives can also accept wildcards (i.e., *any*) enabling messages from any source. Let us consider receive statements within an iteration domain (i.e., the $dom(\mathcal{R})$), such that $\mathcal{D}_S = \{\, id \,|\, 5 \leq id < np \,\}$. The relation for receives is defined in a way that id_s refers to the process id of the sender (or source) process whereas id_t is the target of the data transfer.

$$
\begin{aligned}
recv(buff, 3) &: [\, np \,] \rightarrow \{\, id_s \rightarrow id_t \,|\, id_s = 3 \wedge 5 \leq id_t < np \,\} \\
recv(buff, p) &: [\, np, p \,] \rightarrow \{\, id_s \rightarrow id_t \,|\, id_s = p \wedge 5 \leq id_t < np \,\} \\
recv(buff, id - 1) &: [\, np \,] \rightarrow \{\, id_s \rightarrow id_t \,|\, id_s = id_t - 1 \wedge 5 \leq id_t < np \,\} \\
recv(buff, i{:}lb..ub) &: [\, np, lb, ub \,] \rightarrow \{\, id_s \rightarrow id_t \,|\, lb \leq id_s < ub \wedge 5 \leq id_t < np \,\} \\
recv(buff, any \,|\, f_{na}(\ldots))) &: [\, np \,] \rightarrow \{\, id_s \rightarrow id_t \,|\, 0 \leq id_s < np \wedge 5 \leq id_t < np \,\}
\end{aligned}
$$

The considerations are similar to the send operation. We treat the wildcard *any* as a possible match from any of the processes in the communicator (i.e., $id \in [0 \dots np)$).

For any pair of communication statements $(s, r) \in S \times R$ a relation $\mathcal{R}_m$ can be built by logically combining the *constraints* of relations $\mathcal{R}_s$ and $\mathcal{R}_r$. This can be easily done since the mapping associated to the communication statements is always from id_s to id_t (both for sends and receives). When the new relation $\mathcal{R}_m$ is not empty then it means that there are values of id_s and id_t which satisfy the inequalities and therefore matching between the two statements is possible.

$$\mathcal{R}_m(s, r) = \{\, id_s \rightarrow id_t \,|\, \mathcal{R}_s \wedge \mathcal{R}_r \,\} \neq \emptyset$$

This relation is conservative in the sense that no existing match can be excluded. However, many *spurious* matches will be established making any result produced by the algorithm useless. For example by the equations given above, any send statement, issued by process p, targeting process id 0 will produce a match towards every receive operation issued by process 0 whose source expression includes process p.

5.4.3.3 Detection of Communication Phases

The number of spurious matchings can be reduced based on the observation that many real codes have a regular structure and they are usually divided into different *independent regions (*or *phases)*. The idea is that in each program phase, all communication statements are only matched with others belonging to the same region. We define, for this scope, a novel static analysis to determine the subset of sends and receives semantically belonging to the same program phase and therefore restrict the preliminary matches only within regions.

Scheduling functions are assigned to every program statement on the basis of their position within the AST as explained in [23]. However, since our approach focuses on communication statements only, a different strategy is necessary. We compose the scattering functions in a way that statements belonging to different branches of conditional statements (e.g., `IF` and `SWITCH`) are set to be executed in parallel, and thus the same execution time is assigned. The Algorithm 5 is used to assign scheduling functions to a generic MPI program. While this is correct for blocking statements, the schedule for non-blocking communications is slightly different. Indeed, we assign the schedule for an asynchronous communication statement based on the location of the corresponding *wait* statement which can be statically determined on the basis of dataflow *def-use chains* analysis. A limitation of this approach is represented by the semantics of `MPI_Waitany`,

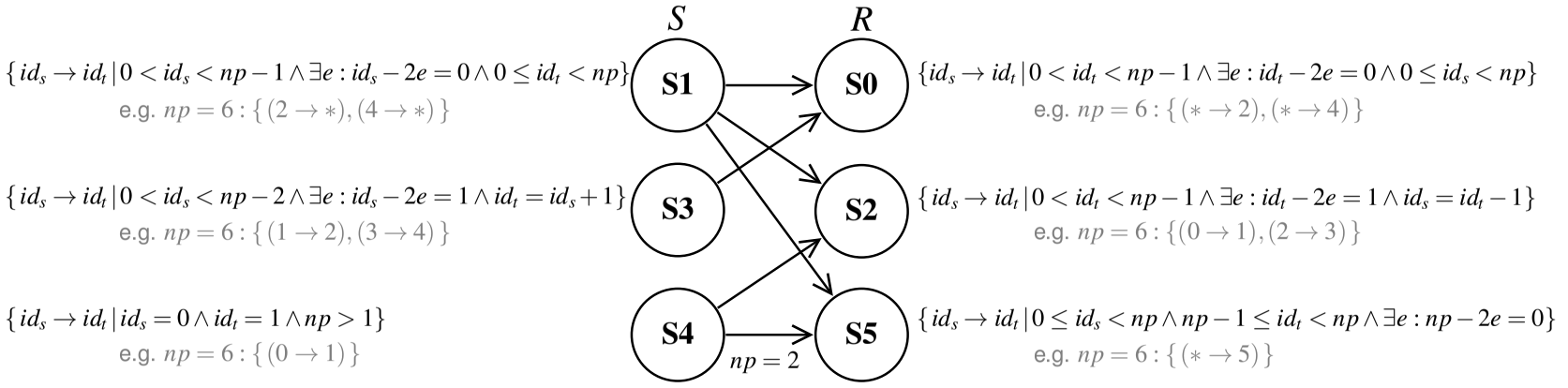


FIGURE 5.10: Preliminary matches for the MPI code in Listing 5.12 determined on the basis of polyhedral relations.

`MPI_Waitsome` and `MPI_Test` loops which are difficult to capture statically. Scheduling for the communication statements of Listing 5.12 are the following:

$$\theta_{S0} = \{1,0\} \quad \theta_{S1} = \{0,0\} \quad \theta_{S2} = \{1,0\} \quad \theta_{S3} = \{0,0\} \quad \theta_{S4} = \{0,0\} \quad \theta_{S5} = \{0,0\}$$

It is important to note that the statements $S0$ and $S2$, which are both non-blocking receive operations, have the same schedule. The scattering refers to the position of the wait statement in line 16. By ordering the statements based on the execution date we obtain the following sequence: $(S1 \,||\, S3 \,||\, S4 \,||\, S5) \prec (S0 \,||\, S2)$. This means that $S1, S3, S4$ and $S5$ are executed in parallel and before $S0$ and $S2$ which are also independent on each others.

Once statements are labeled with a schedule, a second algorithm finds partitions of statements which logically belong to the same program phase. The process is described in Algorithm 6. Communication statements are ordered by their schedule. The *phase* variable (defined as a *set* of statements) is initialized by statements with the same earliest execution time (i.e., '0'). On these statements we perform the preliminary matching returning a bipartite graph $MG(S, R, E)$. In order for MG to be recognized as a region, several properties have to be verified.

The idea is to guarantee that every instance of send statements in S is matched, in a bijective way, to a receive operation in R. This problem is similar to computing the maximum flow of a network [104]. Indeed, if we think of send statements as generators (or *sources*) of the flow and receives as consumers (or *sinks*), a matching is *valid* only if the maximum flow possible through the network equals the generated capacity. If just one of the sends or receive instances is unmatched, then this set of statements cannot be considered a program phase, hence a larger region has to be considered. In our context, the flow is represented by the cardinality information available at the sources, sinks and the edges of the bipartite graph MG.

In order to perform this check, the bipartite graph is transformed by introducing two *artificial* vertices: s and t representing, respectively, the source and the sink of the network. Edges are inserted into the graph MG connecting the source with every send statement, $\forall\, v \in S \,|\, (s, v) \in E$, and each receive with the target vertex $\forall\, v \in R \,|\, (v, t) \in E$. The capacity of these edges is the value of $|\, \mathcal{D}_S \,|$ associated to send and receive statements. We use $c(u, v)$ to represent the capacity of a generic edge (u, v); $f(u, v)$ for its flow. Moreover, for matching edges $c(u, v)$ is the cardinality of the relation $\mathcal{R}_m(u, v)$.

The first property is that every process *id* is contributing to a region. Since the scheduling time is always associated to blocking calls, a synchronization point is found if every

Algorithm 5 Scheduling for blocking comm. statements

```
1:  Input: stmt                                          ▷ Root node of the program AST
2:  Outpt: stack sched                                   ▷ Generated schedule for this program
3:  procedure VISIT(stmt : input, stmt : output)         ▷ AST Visitor
4:      switch kind(stmt)
5:          case IF:                                     ▷ Similar to SWTICH (omitted for space reasons)
6:              proc_id_dep ← contains(cond(stmt), id)
7:              sched.push(0)
8:              Visit(child(stmt, 0))                    ▷ Recur on the then body
9:              sched.pop()
10:             sched.push(proc_id_dep ? 0 : 1)
11:             Visit(child(stmt, 1))                    ▷ Recur on the else body
12:             sched.pop()
13:         case FOR:
14:             sched.push(iter(stmt))
15:             Visit(child(stmt, 0))                    ▷ Recur on the for body
16:             sched.pop()
17:         case COMPOUND:                               ▷ Any block of code: { ... }
18:             cur ← 0
19:             for each child ∈ stmt do
20:                 sched.push(cur++)
21:                 Visit(child)                         ▷ Recur on every child stmt
22:                 sched.pop()
23:             end for
24:         default:                                     ▷ Any other stmt which is not a control-flow stmt
25:             if isCommunication(stmt) then
26:                 stmt.schedule ← to_vect(sched)       ▷ Assign schedule
27:             end if
28: end procedure
```

process is in a communication routine.

$$\bigcup_{v \in S} dom(\mathcal{R}_v) \cup \bigcup_{v \in R} range(\mathcal{R}_v) = \{id \,|\, 0 \leq id < np\} \tag{5.1}$$

This rule restricts our region detection scheme to program phases actively involving all processes of a communicator; this is however the usual pattern for SPMD programs. The second property is that the sum of the capacities leaving node s has to be equal to the sum of capacities reaching node t:

$$TotalFlow(\, MG(S, R, E)\,) = \sum_{v \in S} c(s, v) = \sum_{v \in R} c(v, t) \tag{5.2}$$

We call this quantity the $TotalFlow$. When this property holds, we proceed by computing the maximum flow of the network. This condition is necessary but not sufficient since within the network there might be not enough edges (or capacity associated to them) to transfer the generated flow toward the sink. Unfortunately, in the generic case,

Algorithm 6 Detection of Communication Phases

```
1: Input: stmt                          ▷ List of communication statements in the program
2: Output: set phases ← { }             ▷ Detected standalone program regions
3: set phase ← { }; int cur ← 0, idx ← 0
4: procedure DetectPhases(stmts : input, phases : output)
5:     sort(stmts, key = stmt.schedule)                    ▷ Sort stmts by schedule
6:     while cur < max(stmts.schedule) do
7:         while idx < stmts.size ∧ stmts[idx].schedule = cur do
8:             phase.insert(stmt[idx++])
9:         end while
10:        MG(S, R, E) ← PreliminaryMatch(phase)
11:        if CheckMaxNetworkFlow(MG) then
12:            phases.insert(MG)
13:            phase ← { }                                 ▷ start new phase
14:        end if
15:        cur++
16:    end while
17:    return phases
18: end procedure
```

cardinality information are parametric on the value of np and possibly any input of a program. Therefore solving the parametric maximum network flow problem, in general, may be difficult.

For our running example (in Listing 5.12), Algorithm 6 performs the following steps. Initially statements with execution time equals to '0' (i.e., $S1$, $S3$, $S4$, and $S5$) are inserted into the set *phase*. It can be verified that condition (5.1) is met. Condition (5.2) requires the cardinalities associated to the communication statements:

$$|\mathcal{D}_{S0}| = |\mathcal{D}_{S1}| = |\mathcal{D}_{S3}| = \left\lfloor \frac{np-2}{2} \right\rfloor \qquad |\mathcal{D}_{S2}| = \left\lfloor \frac{np-1}{2} \right\rfloor$$

$$|\mathcal{D}_{S4}| = \begin{cases} 0 & \text{if } np = 1 \\ 1 & \text{otherwise} \end{cases} \qquad |\mathcal{D}_{S5}| = \begin{cases} 1 & \text{if } np \text{ is even} \\ 0 & \text{otherwise} \end{cases}$$

Applying condition (5.2) yields that:

$$|recvs| = |sends| \Rightarrow \begin{cases} 2\left\lfloor \frac{np-2}{2} \right\rfloor + 1 = 1 & \text{if } np \text{ is even} \\ 2\left\lfloor \frac{np-2}{2} \right\rfloor + 1 = 0 & \text{if } np \text{ is odd} \end{cases}$$

The system has a solution only for $np = 2$ since $S4$ and $S5$ matches and the cardinalities of the other statements are zeros. However, a solution which holds for any value of np cannot be found. This can be easily seen by the fact that statement $S3$, in this configuration of the MG, has no outgoing edges meaning that none of its instances

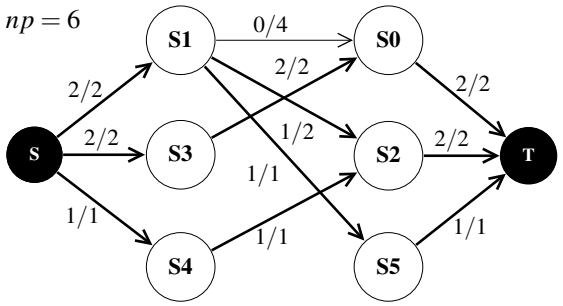


FIGURE 5.11: Maximum network flow for the example of Listing 5.12 for $np = 6$.

can be matched. Therefore the algorithm continues by adding to the *phase* variable additional statements with a schedule equal to '1'. Both non-blocking receive statements $S0$ and $S2$ are included. We again check whether (5.2) is verified:

$$\begin{cases} 2\left\lfloor \frac{np-2}{2} \right\rfloor + 1 = \left\lfloor \frac{np-2}{2} \right\rfloor + \left\lfloor \frac{np-1}{2} \right\rfloor + 1 & \text{if } np \text{ is even} \\ 2\left\lfloor \frac{np-2}{2} \right\rfloor + 1 = \left\lfloor \frac{np-2}{2} \right\rfloor + \left\lfloor \frac{np-1}{2} \right\rfloor & if np \text{ is odd} \end{cases}$$

When np is even the relation $\left\lfloor \frac{np-2}{2} \right\rfloor = \left\lfloor \frac{np-1}{2} \right\rfloor$ holds, thus the first equality is confirmed for every even value of np. For odd values $\left\lfloor \frac{np-1}{2} \right\rfloor = \left\lfloor \frac{np-2}{2} \right\rfloor + 1$, hence (5.2) is verified for any np. Since the checks are satisfied, we proceed on computing the value of the maximum flow. Solving the parametric maximum network flow problem is beyond the scope of this paper we therefore consider, for simplicity, a value of $np = 6$ and solve this particular instantiation of the network. It can be proved, by induction, that the findings hold for any value of np. The outcome is depicted in Figure 5.11.

Every edge (u, v) of the network in Figure 5.11 has been labeled with $f(u, v)/c(u, v)$ stating, respectively, the passing flow and the total capacity of an edge. The computed maximum flow is 5 which equals the value of $TotalFlow = 2\left\lfloor \frac{6-2}{2} \right\rfloor + 1$.

Beside validating MG as a standalone phase, the maximum flow excludes $(S1, S0)$ from the matches. This is an important property, since the cardinality of $S3$ and $S0$ are the same, the only way for dynamic instances of $S3$ to find a match is through instances of statement $S0$. For the code in Listing 5.12, only one solution is produced by the maximum flow algorithm, it can be verified that the same matching is established for any $np > 3$. Precisely, for odd values of np edge $(S1, S5)$ is removed as explained before. In general, there might be several solutions, involving a different configuration of the edges; in such cases the algorithm accepts only those with a flow equal to the $TotalFlow$. If MG is marked as a region, all found valid matches must be considered.

	dependent variables	# of comm. stmts	# of regions	valid matches
Jacobi	2	4	1	1
ADI Solver	4	12	4	4
Particle [132]	0	10	5	5
Bitonic [133]	1	11	4	4
NPB-IS	0	2	1	1
NPB-CG	0	20	10	10
NPB-MG	0	16	2	**30**
NPB-BT	2	36	13	13
NPB-SP	0	24	7	7
NPB-LU*	12	24	9	9
NPB-DT	–	–	–	–

TABLE 5.4: Evaluation of the static matching algorithm for real MPI codes.

5.4.4 Experimental Evaluation

We manually evaluated our approach on a number of MPI codes and benchmarks from the NAS Parallel Benchmark (NPB) suite [131]. We only considered codes using point-to-point communications and excluded the ones which solely rely on collective communications (i.e., `EP`, `FT`). For each of them we determined the number of variables being detected as dependent, the total number of point-to-point communication statements (after inlining), the number of regions recognized by our detection algorithm, and at last the number of valid matchings found by our analysis. Results of the analysis are collected in Table 5.4. Tools from related work could not be considered for a comparison since the majority of the chosen codes contain non-blocking communications which are not supported.

The `Jacobi` and the `ADI Solver` are structured as neighbour communication. As we examined in this paper, this kind of pattern is well handled by our approach. For both cases we determine exactly one valid matching for every identified region. The `Particle` Simulator [132] code is interesting since this code has not been extensively optimized like the NPB. Also for this case our phase detection algorithm finds regions which are composed by a pair of send/receive statements making the matching trivial. Beside the matchings, we can determine the actual communication patterns within those regions since the available information both at sender and receiver sides is complete. Moreover, many of the phases identified in the `Particle` code contain collective communication patterns similar to scatter and gather operations. A successive optimization phase could, based on our analysis results, replace the original code with a call to more efficient collective operations.

Another interesting example is `NPB-MG`. In this code send and receive communication statements are within two distinct function calls which are being invoked multiple times. In the order, a function containing a non-blocking receive is invoked twice followed by two invocations of the function with blocking sends. These should be seen as two logically distinct phases, but since they are overlapped, our algorithm recognizes them as a single one. Within this phase we therefore obtain a number of spurious matchings (i.e., 15 instead of 1). Since this pattern is repeated twice in the program, we end up with 30 possible matchings instead of 2. However, this result is a cause of the inlining pass which produces multiple instances for every communication statement inside functions. By projecting the result on the original statement identifiers, the spurious matchings can be eliminated.

Benchmarks `NPB-BT` and `NPB-SP` have a similar communication structure. One of the issues in these codes is that target and source of communication statements are often calls to non-affine functions. Interestingly in both benchmarks there is one particular region, which is composed by 12 statements (6 sends and 6 receives), for which we are able to determine the region boundaries thanks to the flow check. However for this region, the amount of spurious matchings makes the number of valid matchings large (i.e., 6!). Fortunately, by including the message *tag* in the inequalities our method can eliminate all spurious matchings and validate a single match for that region.

`NPB-LU` represents one of the most complex codes. Many dependent variables are used to guide the control flow. They are correctly propagated by our rank propagation analysis however generated expressions are not always affine. This is due to the fact that communication is arranged as a grid and neighbors are computed using non affine expressions. Differently from other benchmarks, these expressions are used also within control flow statements and therefore this inhibits our approach from being applied since cardinality information cannot be computed. However since the `NPB` are usually compiled for a specific communicator size, constant folding can be used to compute the value of all those expressions statically and thereby making the code suitable for our approach. Under this constraint we recognize the 9 phases all of them containing a single valid match.

The `NPB-DT` code cannot be handled by our approach because of the control flow structure non respecting the constraints of SCoPs. Many for loops contain `continues`, furthermore several control-flow conditions surrounding communication statements are based on comparisons of string literals. This prevents the iteration domains from being computed therefore inhibiting the rest of the approach.

5.5 Summary

In this chapter we exploited compiler technology for optimizing distributed memory programs. We proposed a transformation which can be delivered by a compiler to hide communication costs by increasing the overlap between computation and communication (see Section 5.2). Experiments shown this technique shows its benefits when a large number of presses are being deployed since communication costs become predominant.

A second opportunity for optimization has been uncovered by analyzing cache behaviour of point-to-point communication routines (see Section 5.3). We shown that for particular sizes of the buffer being exchanged, code can be transformed to achieve a 40% performance improvement. We proposed a set of simple rules to be considered while writing a distributed memory code which is aware of cache behaviour. Beside being used by developers, we believe those rules are simple enough to be automatically exploited by compilers.

At last, we proposed a static analysis with the purpose of determining a match of point-to-point communication routines (see Section 5.4). This analysis is of paramount importance for allowing any compiler to safely manipulate a distributed memory program.

This work, we believe, will be the basis for future compiler analyses and transformations targeting distributed memory system. The work pursued in this thesis mostly focuses on point-to-point communication statements but extension to collective routines is possible.

Chapter 6

Conclusion and Future Work

Message passing nowadays is wide-spread in the HPC community. However, the continuous growth of computing units within workstations, made possible by multi-core chips and the advent of General Purpouse Graphics Processing Unit (GPGPU) computing, calls for a redesign of the model. Lightweight threads should be made first-class citizens within the model and whether a thread is allocated on a CPU or a GPU, the way it is addressed by the model should be the same. With *libWater*, presented, in Section 3.4, we achieved that. A lightweight interface coupled by a powerful runtime system allows programmers to address heterogeneous cluster systems hiding low level synchronization characteristic of the message passing model. Thanks to the use of the OpenCL computational model, access to *su*s are completely homogeneous. The distributed runtime system realized within this thesis showed very good scalability on large production systems. The infrastructure allows for pluggable dynamic DAG-based optimizations which make *libWater* easy to extend in order to address future challenges.

Runtime systems for message passing are quite complex and highly configurable. In Chapter 3 we showed the measure of the performance improvement which can be achieved by tuning a subset of the runtime parameters provided by the Open MPI library. Our methods can improve performance up to 30% if parameters are tuned to match a particular application code. In order to deliver parameter settings which are suited for a larger range of codes we used a statistical approach based on ANOVA. Results show that applications running with these parameter see a 20% performance improvement with respect to Open MPI's default parameter setting.

Finally we showed how compilers can help in rewriting input programs to better match the underlying architecture. We showed how the placement of message passing statements impact on the cache utilization and we elaborated few intuitive rules, that can be coded into a compiler, which can significantly improve program performance.

We proposed in this thesis an analysis which is able to deliver the matching of communication statements. This is the corner stone for any compiler based transformation of a message passing code. We expressed our analysis on top of well-established analysis techniques employed in several mainstream compilers. However the costs of the proposed analysis, although we use heuristics to reduce the complexity, may be too expensive for production compilers. This is nevertheless an improvement over current techniques which can handle only a small subset of the message passing programming model (i.e., no wildcards and no asynchronous communications).

At last we presented a method to encode the semantics of message passing routines within the PM. This allows common loop transformations, such as strip mining, distribution and fusion, to be applied to communication statements in an automatic way. In this thesis, we use such method to extract fine grained dependence analysis information which allows the compiler to better hide communication costs.

6.1 Contributions

We here list the peer-reviewed papers representing the contributions for each chapter of the thesis.

6.1.1 Chapter 3

PDP12 : Simone Pellegrini, Radu Prodan, Thomas Fahringer, *A Lightweight C++ Interface to MPI*, Proceedings of the 20th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), Garching, Germany, 15-17 February, 2012.

EuroMPI11 : Simone Pellegrini, Radu Prodan, Thomas Fahringer, *Leveraging C++ Meta-Programming Capabilities to Simplify the Message Passing Programming Model*, Proceedings of the 18th EuroMPI Conference, Santorini, Greece, 18-21 September, 2011.

ICS13 : Ivan Grasso, Simone Pellegrini, Biagio Cosenza, Thomas Fahringer. *libWater: Heterogeneous Distributed Computing Made Easy.* In 27th ACM International Conference on Supercomputing (ICS), Eugene, Oregon, 10-14 June.

6.1.2 Chapter 4

Euro PVM/MPI09 : Simone Pellegrini, Jie Wang, Thomas Fahringer, and Hans Moritsch, *Optimizing MPI Runtime Parameter Settings by Using Machine Learning.* Proceedings of the 16th Euro PVM/MPI Conference (EuroPVM/MPI 2009), September 7–10, 2009, Espoo, Finland.

CF10 : Simone Pellegrini, Thomas Fahringer, Herbert Jordan, Hans Moritsch, *Automatic tuning of MPI runtime parameter settings by using machine learning*, Proceedings of the 7th ACM international conference on Computing frontiers, Bertinoro, Italy, May 17–19, 2010. [poster]

IWPAPS12 : Simone Pellegrini, Radu Prodan, Thomas Fahringer, *Tuning MPI Runtime Parameter Setting for High Performance Computing.* Proceedings of the IEEE Cluster Workshops IWPAPS 2012, Beijing, China, 24–28 September 2012.

6.1.3 Chapter 5

EuroMPI12 : Simone Pellegrini, Torsten Hoefler, Thomas Fahringer, *Exact Dependence Analysis for Increased Communication Overlap.* In Proceedings of the 19th EuroMPI Conference, Vienna, 23-26 September, 2012.

Cluster12 : Simone Pellegrini, Torsten Hoefler, Thomas Fahringer, *On the Effects of CPU Caches on MPI Point-to-Point Communications.* In the IEEE International Conference on Cluster Computing, Beijing, September 2012.

6.1.4 Other Contributions

The work behind this thesis also inspired and enabled other publications on topics on not strictly related to the ones covered in this thesis.

In [134], the Insieme compiler has been used for optimizing runtime and efficiency of OpenMP cache-sensitive parallel programs. The compiler relies on the polyhedral model to combine *valid* loop transformations and to produce many semantically equivalent versions from every OpenMP code region. A novel optimizer is then employed which determines the set of *pareto-front* solutions – for a combination of number of threads and input data – by running a small subset of generated code versions.

In [135], polyhedral model iteration domain information have been used to generate *effort estimation functions* used by the Insieme runtime to derive the optimal loop schedule for a given loop, workgroup size, iteration range and system state.

In [136], the IR employed within the Insieme compiler, called INSPIRE, is described. The paper describes the basic infrastructure with a focus on the representation of the parallel control flow of a program which is natively supported by the proposed IR. The paper describes how several parallel paradigms (i.e., OpenMP, OpenCL and MPI) can be represented by INSPIRE.

6.2 Future Work

There is certainly an effort that the research community has to make to improve the programmability of distributed memory systems. OpenCL seems the right way to address this problem. However distributed memory system should be made a first class citizen within the model itself. We think in the future OpenCL is going to be used as the backend of more sophisticated tools which will have to purpose to hide the underlying complexities. This is similar to what we proposed with *libWater*, however more focus on performance is needed in order to breach the HPC community.

Tuning of runtime parameters is an easy way of squeezing out performance of existing codes for which the source code may be not available. Although this is a well-known effect, MPI implementations expose different parameters with slightly similar semantics. We believe that an effort should be made within the MPI forum to bring standardization to those parameters. In this way, by having a common interface and a precisely defined semantics, it would be possible to make optimal parameter settings portable across implementations. Another interesting research aspect would be to open-up tuning interfaces to make those runtime parameters adjustable at runtime. This will allow a program to adapt to optimally address each of the program phases which may have different requirements for what concerns optimal runtime parameters settings.

Compiler-enabled optimization of message passing programs is a field which we believe has not been completely exhausted yet. In this thesis we propose a way of representing the semantics of MPI programs within the PM. This enables many transformations (e.g. message splitting, coalescing) to be applied automatically to a code. Furthermore, since the PM allows for composability of transformations, iterative feedback-driven techniques may be used in the future to determine the best transformation sequence that optimizes a message passing programs.

In this thesis we focused primarily in improving the performance of a program, however many of the methods presented here may be suitable to optimize other application non-functional parameters like energy consumption and efficiency.

Appendices

Appendix A

The Message Passing Interface

The Message Passing Interface or MPI is the *de-facto* standard for programming distributed memory machines. Its development started in 1991 by an open, international forum consisting of representatives from industry, academia, and government laboratories. It has received widespread acceptance because it has been carefully designed to permit maximum performance on a wide variety of systems. The standard [3] defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in Fortran or the C programming language.

The MPI interface is meant to provide essential *virtual topology*, *synchronization*, and *communication* functionality between a set of processes in a language-independent way. Typically, for maximum performance, each CPU (or core in a multi-core machine) will be assigned just a single process. This assignment happens at runtime through the agent that starts the MPI program, normally called `mpirun` or `mpiexec`.

MPI library functions include, but are not limited to, point-to-point rendezvous-type send/receive operations, choosing between a Cartesian or graph-like logical process topology, exchanging data between process pairs (*send/recv* operations), combining partial results of computations (*gather* and *reduce* operations), synchronizing nodes (*barrier* operation) as well as obtaining network-related information such as the number of processes in the computing session and the current processor identifier. Point-to-point operations come in synchronous, asynchronous, buffered, and ready forms, to allow both relatively stronger and weaker semantics for the synchronization aspects of a rendezvous-send. Since MPI version 3.0 [3], collective operations can also have non-blocking semantics for better communication over computation overlap.

A.1 Structure

MPI programs must follow a precise structure. Before any communication is performed the `MPI_Init` routine must be invoked which initializes the communication context (the communicator). Within an MPI program, the `MPI_Init` routine must be invoked only once by each process, successive invocations are either a no-operation or may cause the program to abort. After the initialization of the communication context, MPI routines can be invoked. Before the end of a program, in order to release resources, the function `MPI_Finalize` must be invoked. No communication routines can be invoked after the release of the communication context. Additionally, a call to `MPI_Init` is not allowed after the `MPI_Finalize` call.

A.2 Concepts

In this Appendix we show the mapping between the generic message passing programming model presented in Section 2.2 and the MPI interface with is used throughout this paper.

A.2.1 Communicator

A communicator object groups processes together within a *communication context*. Each communicator assigns to each contained process an independent identifier and arranges its contained processes in an ordered topology. Messages exchanged within a context cannot be captured by others, therefore communicators are often used to encapsulate the communication context used within parallel libraries.

Beside communicators, MPI also offers the concept of *groups* which are mainly used for reorganizing a group of processes before a new communicator is created. This gives the ability to split and merge communicators creating many communication contexts within a program.

The message passing model presented in Section 2.2 does not explicitly define communicators since in many of the work presented a single communication context is needed. In all of the code example used throughout this thesis a single communicator is often used, i.e., `MPI_COMM_WORLD`, which is the communicator created after the initialization of an MPI program which contains all the processes being spawned by the `mpirun` agent.

A.2.2 Point-to-Point routines

Semantics of point-to-point routines is covered by Section 2.2. In this section, we map the semantics of the basic operations in our model with the concrete implementation provided by the MPI standard.

The semantics of the *send* operation (see Definition 16) is implemented by:

```
int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm)
```

This routine sends `count` elements of type `datatype` (we will discuss datatypes in Section A.2.4) starting from local address `addr` to process *id* `dest` through channel `tag` within communicator `comm`. The routine is blocking waiting from a compatible receive operation being posted by the `dest` process. It then issue the transfer and returns the control to the caller.

The non-blocking semantics, i.e., *isend* (see Section 18), is implemented by the MPI routine:

```
int MPI_Issend(void *buf, int count, MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm, MPI_Request* req)
```

Differently from `MPI_Ssend` this routine returns the control immediately to the caller and its competition can be checked by testing the `req` handler using the *wait* routine, see Definition 20, which in MPI corresponds to:

```
int MPI_Wait(MPI_Request* req, MPI_Status* status)
```

This function receive as input the handler object `req` and blocks until the corresponding operation is completed. The result of the underlying transfer is then written to the `status` object which contains information on the destination or source process, the tag being used, amount of data exchanged and in case an error occurred, the corresponding error code.

The *recv* and *irecv* primitives, see Definition 17 and Definition 19, have also corresponding routines in MPI, one which implements the blocking semantics, i.e., `MPI_Recv`, and one which is non-blocking, i.e., `MPI_Irecv`. Like for the send, the signature of the two functions are similar:

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int src,
             int tag, MPI_Comm comm, MPI_Status* status)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int src,
              int tag, MPI_Comm comm, MPI_Request* req)
```

The blocking version returns information associated with the received message. The `status` object contains information described above concerning the source and the amount of data being received. If the programmer is not interested in this information, because it is already explicit in the call to the receive, then the value `MPI_STATUS_IGNORE` can be used. As a matter of fact querying the `status` object is only meaningful when the receive statement uses wildcards for selecting message sources, i.e., `MPI_ANY_SOURCE`, and tags, i.e., `MPI_ANY_TAG`.

A.2.2.1 Send and Buffering

The MPI standard offers several versions of the *send* routine with different behaviours. One of the most used, which is usually also utilized within code examples throughout this thesis is the `MPI_Send`. Differently from the `MPI_Ssend`, presented above, `MPI_Send` directly pushes the content of the sent buffer to the receiver process, by storing it into an *internal* (or *kernel*) buffer, without waiting for the posting of a matching receive. In this way, when the `MPI_Recv` is posted the data will be read from the kernel buffer.

This way of transferring data is called *eager*. However, since the amount of buffering provided at the receiver side is limited, this mechanism alone is not sufficient. The presence of implicit buffering can also cause portability problems since programs which work fine on one architecture stops working (usually deadlocking) on a different setup.

For larger message sizes the behaviour of the `MPI_Send` switches to a different protocol called *rendezvous*. In this mode the sender sends to the receiver a request for transferring the data. When the receiver is ready, which means the receive routine is being executed, it sends back to the sender an acknowledgment. After that the transfer begins and the content flows directly from the sender buffer to the receiver buffer without any additional copy.

A.2.3 Collective Routines

Collective calls in MPI involve communication among all processes in a process group or communicator, see A.2.1. The basic collective operations introduced in Section 2.2.2 have a concrete implementation in MPI. The main difference stays in the fact that while

out abstract operations always involve the whole set of processes in a parallel program, the one provided by MPI can be invoked on a subset of processes.

The *barrier* defined in Definition 22, is implemented by the routine:

```
MPI_Barrier(MPI_Comm comm)
```

When the routine is invoked using the `MPI_COMM_WORLD` communicator its semantics is to block the caller process until all processes have reached the synchronization point. Immediately after the control is returned to the caller.

The *bcast* operation is implemented in MPI by the following routine:

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

The signature and semantics is the same as the one presented in the model section, see Definition 23. The difference is that here `count` number of `datatype` objects are sent instead of simple bytes.

The *scatter* and *gather* operations, on the other hand, are slightly different from the one presented in the Section 2.2.2.

```
int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
                void *recvbuf, int recvcnt, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype,
               void *recvbuf, int recvcnt, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

In both cases two buffers are provided, one send buffer and a receive buffer. For the `MPI_Scatter` the `sendbuff` is only meaningful for the `root` process, other processes can just provide a `NULL` pointer. All the processes, `root` included, must allocate a `recvbuff` (every process is collaborating with data). The `sendcnt` and `recvcnt` state how many objects of the corresponding type are sent by the `root` process to each other process and received by every process. If the same type is used, `sendcnt` must equal `recvcnt`, however the library allows for more complex behaviours.

Similar discussion can be done for the `MPI_Gather` routine. In this case the semantics is inverted.

Last function defined in our model is the reduction primitive. For this, MPI provides an implementation which is similar to our formal definition:

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm)
int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)
```

As for preceding collective operations, also the `root` process must contribute to the `sendbuf` and provided operator `op` must be a binary function which is associative. Non-commutative user-define functions are allowed and defined using the `MPI_Op_create` function. The flag `commute` is utilized to provide the information whether the defined function is commutative.

A.2.4 Derived Datatypes

MPI routines must specify the type of the data which is sent between processes. Primitive types have special type definitions, e.g., the C/C++ `int` type is implemented by the `MPI_INT` datatype. These definitions have two main objectives, on the one hand they are portable across programming languages (an `MPI_INT` type uses the same amount of bytes independently on the underlying programming language). On the other hand, they automatically perform data conversions so that machines with different representations, i.e., little-endian versus big-endian, can transparently interact with each other.

Beside contiguous blocks of data, MPI's datatypes can be used to transfer noncontiguous data. Instead of offload the burden to the programmer of *packing* noncontinuous data into a contiguous buffer at the sender side and *unpack* it at the receiver side; MPI provides mechanisms to specify more general, mixed and noncontiguous communication buffers. It is up to the implementation to decide whether data should be first packed in a contiguous buffer before being transmitted, or whether it can be collected directly from where it resides. The general mechanisms provided by the MPI standard allow one to transfer directly, without copying, objects of various shapes and sizes. Two datatypes will be mostly used throughout this work.

The `MPI_Type_vector` datatype is a general constructor that allows replication of a datatype into locations that consist of equally spaced blocks.

```
int MPI_Type_vector(int count, int blocklength, int stride,
                    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Each block is obtained by concatenating the `blocklenght` number of copies of the old datatype (i.e., `oldtype`). The spacing between blocks, i.e., `stride`, is a multiple of the extent of the old datatype. The number of blocks constituting this datatype is `count`.

The second datatype used is the defined by the function `MPI_Type_create_struct`. This is the most general type constructor. The function signature is the following:

```
int MPI_Type_create_struct(int count, const int array_of_blocklengths[],
                           const MPI_Aint array_of_displacements[], const
                           MPI_Datatype array_of_types[],
                           MPI_Datatype *newtype)
```

It creates a type composed of `count` blocks (or number of entries). The number of elements in block `i` is given by the value of `array_of_blocklengths[i]`. The type of the elements in block `i` is provided by the value of `array_of_types[i]`. The displacement of block `i`, in bytes, from the first block of the struct is given by `array_of_displacements[i]`.

Appendix B

Open MPI's Runtime Parameters

A description of the 27 Open MPI's runtime parameters used in Chapter 4 as returned by the `ompi_info --all` command.

`mpi_yield_when_idle`	Yield the processor when waiting for MPI communication (for MPI processes, will default to 1 when oversubscribing nodes)
`mpi_paffinity_alone`	If nonzero, assume that this job is the only (set of) process(es) running on each node and bind processes to processors, starting with processor ID 0
`mpi_preconnect_mpi`	Whether to force MPI processes to fully wire-up the MPI connections between MPI processes during `MPI_INIT` (vs. making connections lazily – upon the first MPI traffic between each process peer pair)
`mpi_leave_pinned`	Whether to use the "leave pinned" protocol or not. Enabling this setting can help bandwidth performance when repeatedly sending and receiving large messages with the same buffers over RDMA-based networks (0 = do not use "leave pinned" protocol, 1 = use "leave pinned" protocol, -1 = allow network to choose at runtime)
`COLL`: Collective operation tuning (`coll_*`)	
`sm_tree_degree`	Degree of the tree for tree-based operations (must be ≥ 1 and $\leq min(control\_size, 255)$)

sm_control_size	Length of the control data – should usually be either the length of a cache line on most SMPs, or the size of a page on machines that support direct memory affinity page placement (in bytes)
sm_fragment_size	Fragment size (in bytes) used for passing data through shared memory (will be rounded up to the nearest control_size size)
sync_barrier_after	Do a synchronization after each Nth collective
sync_barrier_before	Do a synchronization before each Nth collective
tuned_init_tree_fanout	Initial fanout used in the tree topologies for each communicator. This is only an initial guess, if a tuned collective needs a different fanout for an operation, it build it dynamically. This parameter is only for the first guess and might save a little time
tuned_init_chain_fanout	Initial fanout used in the chain (fanout followed by pipeline) topologies for each communicator. This is only an initial guess, if a tuned collective needs a different fanout for an operation, it build it dynamically. This parameter is only for the first guess and might save a little time
SM: Shared memory communication tuning (btl_sm_*)	
eager_limit	Maximum size (in bytes) of "short" messages (must be ≥ 1)
max_send_size	Maximum size (in bytes) of a single "phase 2" fragment of a long message when using the pipeline protocol (must be ≥ 1)
rndv_eager_limit	Size (in bytes) of "phase 1" fragment sent for all large messages (must be ≥ 0 and *le* eager_limit)
fifo_size	–
num_fifos	–
OpenIB: InfiniBand communication tuning (btl_openib_*)	
eager_limit	Maximum size (in bytes) of "short" messages (must be ≥ 1)
max_send_size	Maximum size (in bytes) of a single "phase 2" fragment of a long message when using the pipeline protocol (must be ≥ 1)

`rndv_eager_limit`	Size (in bytes) of "phase 1" fragment sent for all large messages (must be ≥ 0 and *le* eager_limit)
`use_message_coalescing`	If nonzero, use message coalescing
`use_eager_rdma`	Use RDMA for eager messages (-1 = use device default, 0 = do not use eager RDMA, 1 = use eager RDMA)
`eager_rdma_num`	Number of RDMA buffers to allocate for small messages (must be ≥ 1)
`use_async_event_thread`	If nonzero, use the thread that will handle InfiniBand asynchronous events
`ib_max_rdma_dst_ops`	InfiniBand maximum pending RDMA destination operations (must be ≥ 0)
`rdma_pipeline_send_lenght`	Length of the "phase 2" portion of a large message (in bytes) when using the pipeline protocol. This part of the message will be split into fragments of size `max_send_size` and sent using send/receive semantics (must be ≥ 0; only relevant when the PUT flag is set)
`rdma_pipeline_frag_size`	Maximum size (in bytes) of a single "phase 3" fragment from a long message when using the pipeline protocol. These fragments will be sent using RDMA semantics (must be ≥ 1; only relevant when the PUT flag is set)
`min_rdma_pipeline_size`	Messages smaller than this size (in bytes) will not use the RDMA pipeline protocol. Instead, they will be split into fragments of `max_send_size` and sent using send/receive semantics (must be ≥ 0, and is automatically adjusted up to at least (`eager_limit+btl_rdma_pipeline_send_length`); only relevant when the PUT flag is set)

Appendix C

The Insieme Compiler Toolset

The Insieme project [137] comprises two major components – the *Insieme Compiler* and the *Insieme Runtime* System (RS). The compiler's task is to convert input codes based on various languages (C, C++, OpenCL) into a C-based representation which, combined with the RS, can be compiled into a binary for a target architecture. During this process, the compiler analyzes, manipulates and restructures the code as long as the *intended* semantic of the program is not lost.

The source-to-source compiler provides a set of libraries including algorithms and data structures to parse, analyze, manipulate and synthesize programs. In order to render compiler analyses and transformations independent on the input language, the Insieme Compiler relies on a program representation which is independent on the input language and paradigm used. The Insieme parallel intermediate representation (INSPIRE) [136], or just IR, is realizing this central component. This formal language has been designed to express both sequential and parallel concepts encountered within programming languages like OpenMP, OpenCL and MPI in a uniform way. As a consequence, analyses and transformations designed to work on this IR can be reused among several formalisms.

The RS provides an abstract interface to the underlying hardware infrastructure. Further, the RS offers interfaces for influencing the execution of parallel codes (runtime-tuning) as well as for monitoring the execution of applications. The latter allows the use of collected data for post-mortem analyses as well as for real-time decision making processes during the execution.

C.1 Compiler Infrastructure

This thesis focuses prevalently on the Insieme Compiler which is composed of four main building blocks:

Core: It contains the data structures and utilities for representing, building and manipulating the IR;

Frontend: It converts a generic input program into INSPIRE. Currently only C is supported. The C frontend, realized as part of this thesis work, builds the INSPIRE representation starting from the AST produced by the LLVM/Clang parser [138]. Within the frontend also the parallel semantics of the input codes is re-arranged into INSPIRE constructs (e.g., *pfor* are generated out of OpenMP data sharing loops and channels from MPI point-to-point communications).

Analyses: It realizes several modules used to extract static knowledge from an INSPIRE representation. In particular, an analysis is provided which analyzes the IR for SCoPs. Polyhedral representation extracted from these SCoPs provides the core data structure for loop dependencies analysis and transformations. Furthermore, this module implements the construction of a CFG from INSPIRE and a basic dataflow framework was implemented to support the research presented throughout this thesis.

Transformations: It implements support for program transformations based on INSPIRE. It provides utilities for INSPIRE manipulation and navigation and loop transformations based on the polyhedral representation. Loop transformations currently supported in Insieme are: interchange, strip mining, tiling, unrolling, fusion and distribution.

Backend: It generates C code from INSPIRE. The backend has the ability to produce standalone code, or programs which rely on the RS. Throughout this thesis, we used the former mechanism since the runtime still lacks the ability to run on a distributed memory system.

C.2 INSPIRE Overview

INSPIRE provides the means for representing the structure and the relevant semantic aspects of a program in a way which is independent on the input language [136]. INSPIRE is composed by a minimal type system for representing *parameterized abstract types* (e.g.,$int\langle 8\rangle$ or $type\langle bool\rangle$), *structs*, *unions*, *functions* and *closures*. Expressions can

be variables, literals (e.g., string, integer and floating point values) and call expressions. Special type of semantics is implemented through the use of built-in functions (or operators). For example, *mutable state* is modeled through the *ref.assign* operator which is invoked providing an R-value as right-hand side argument and an L-value (represented in INSPIRE by the $ref\langle\alpha\rangle$ extension) as left-hand side argument. The sequential control flow is represented by 9 statements: *if*, *for*, *while*, *switch*, *compound*, *declaration*, *return*, *break*, and *continue*.

One of the main contributions of INSPIRE is the explicit representation of the parallel control flow done via constructs at the language level and not by means of external library routine calls (as commonly done within mainstream compilers). The basic mean to represent parallelism is represented by the *job* construct which represent computation which is executed cooperatively by a group of threads (i.e., a *thread_group*). A *job* is spawned by any thread using the *spawn* function. Joining of the spawned group is done using the *merge* function.

Information within a *thread_group* is exchanged using 3 primitives. The *pfor* enables work-distribution, it resembles the semantics of the OpenMP parallel for. The *redistribute* routine allows data to be rearranged among the threads using an arbitrary functor. This function is useful to implement collective operations like *bcast*, *scatter*, *gather* and *reduction* operations. The third and last communication mean is represented by *channels* which provide a mechanism for point-to-point communication.

C.2.1 MPI Semantics in INSPIRE

Of particular importance for this thesis is the way the semantics of MPI is represented within INSPIRE. In MPI, the entire application is processed by multiple cooperating processes, typically running on different nodes of a cluster. The whole-program parallelism of an MPI application can be modeled using a top-level $merge/spawn/job$ combination. Within the executed job, an array of communication channels of type $channel\langle msg, 1\rangle$ is created (COM-array). The utilized *msg* type is a struct modeling MPI messages and associated envelope, see Definition 14. The basic structure of an MPI program in the IR is the following

```
1 merge(spawn(job[1-inf] {
2       array<channel<msg,1>> com = redistribute(channel.create(msg,1), id);
3       /* ... MPI program body ... */
4       channel.release(com[getThreadID(0)]);
5   }));
```

where *id* is the identity function of type $(\alpha) \to \alpha$. This listing covers the obligatory `MPI_Init` and `MPI_Finalize` calls (see Appendix A.1). In particular, in line 2 channels

are created to allow communication between each pair of processes (i.e., `MPI_Init`). In line 4, channels are released and spawned tasks are merged together (i.e., `MPI_Finalize`).

Each channel within the COM-array is associated to one of the participating workers. If one MPI worker is sending data to a peer using `MPI_Send` or similar primitives, the data is encapsulated into a message and submitted to the channel associated to the receiver using the *channel.send* primitive. Consequently, MPI instructions receiving messages (e.g.,`MPI_Recv`) are based on the *channel.recv* operator accessing the channel associated to the local thread. Only one channel is instantiated between each pair of processes, this represents a simplification of the *channel* primitive presented in Definition 15, i.e., $|\mathcal{CH}(p_i, p_j)| = 1$. A channel is created through the *channel.create* primitive. It accepts a generic message type, i.e., *msg*, and the amount of buffering provided (in terms of number of messages) by the channel. This allows the definition of both blocking and non-blocking semantics for the *send* operation. However, differently from the point-to-point primitives presented in Section 2.2, a channel cannot be queried for the completion of a data transfer. In INSPIRE, it is assumed that a message is safely copied into the internal buffer once the *channel.send* primitive completes; therefore there is no need for a *wait* primitive.

Beside *send*/*recv* routines, MPI covers a rich set of collective operations. These kind of operations have been generalized by the IR's *redistribute* operator which can be specialized for the particular cases. For instance, an MPI reduction operation summing up the distributed values x and reporting the result exclusively to worker 0 can be modeled by

```
1 redistribute(x,
2    (array<int<4>> data) {
3       if (getThreadID(0) != 0) { return 0; }
4       return sum(data);
5    });
```

where *sum* is an ordinary sequential function summing up the elements of an array. Similar derived constructs can be used to represent *broadcast*, *scatter* and *gather* routines. The redistribute only needs to preserve the I/O semantics of the collective operation. The runtime, instructed by the compiler backend, is then responsible of choosing an appropriate implementation optimized for the target architecture.

INSPIRE implementation of MPI covers a small, but relevant, portion of the standard. Some of the unsupported features can be rewritten in terms of supported ones (e.g., non-blocking semantics of point-to-point routines can be replaced by blocking behaviour). However, some functionalities do not fit with the execution model imposed by the RS. One example is represented by process groups which in MPI are used to define communicators, see Appendix A. Since the parallel model underlying INSPIRE is based on

a *fork-join* paradigm, no primitives are yet provided to regroup threads belonging to a *thread_group*. Another important feature of MPI which has not yet been integrated in INSPIRE is the concept of user data-types. In MPI data-types are used to avoid expensive packing/unpacking operations in the use code. In order to introduce a similar mechanism in INSPIRE the concepts of blocking communication calls and buffering need to be relaxed. Non-blocking semantics can be easily integrated though future objects which resemble the semantics of handler objects returned by MPI non-blocking routines. Furthermore, a mechanism to map MPI data-types to IR types is necessary.

Appendix D

The Iterative Dataflow Analysis Framework

The CFG of a program is an important data structure which allows the compiler to gather information on its dataflow. In particular, dataflow analysis attempts to obtain particular information about a program at each point. For the sake of this thesis, a *program point* corresponds to a basic block. An example of the type of information which can be discovered by dataflow analysis are: the set of definitions reaching a particular use of a variable (i.e., *reaching definitions*), or the set of variables that may be potentially read before their next write (i.e., *live variables*).

Such information can be generally computed from the AST, however the CFG provides a much easier data structure for this purpose. The analysis defines a so called *dataflow equation* for each node of the CFG which is then solved by repeatedly calculating the output values from the inputs locally at each node. Computed outputs are then propagated through the CFG edges to be the input of the successor (in the case of *forward* analyses) or the output of predecessor (for *backward* analyses) nodes. Since the CFG can have loops, dataflow equations have to be solved iteratively until the whole system stabilizes, i.e., it reaches a so called *fixpoint*.

Dataflow variables are associated with the entry and exit points of each basic block in the CFG. Solving of the dataflow equations yields the new dataflow variable associated with either the entry or the exit (depending on the direction of the analysis). In general, in order for the iterative approach to converge, the value of a dataflow variable during an iteration is always as *subset* of the value in the preceding iteration. In fact, an order can be imposed on the entire space of dataflow values which can be assigned to variables associated with basic blocks. This is often modeled using a mathematical abstraction called *lattice*.

Definition 39 – *Poset*

Let us define a *partial order*, $\sqsubseteq$ on a set B as a relation over $B \times B$ that has the following properties:

1. *Reflexive.* For all elements $x \in B, x \sqsubseteq x$.
2. *Transitive.* For all elements, $x, y, z \in B, x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$.
3. *Anti-symmetric.* For all elements $x, y \in B, x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$.

A *partially ordered set*, or *poset*, is denoted by $(B, \sqsubseteq)$, is a set B and a partial order $\sqsubseteq$.

In the context of dataflow analysis, the relation $x \sqsubseteq y$ can be read as "x is a conservative (safe) approximation of y". If $x \sqsubseteq y$, then the dataflow value x can be used in place of y without effecting the correctness of the program. We use the $\sqsubseteq$ symbol as an abstract relation which is then replaced by a concrete mathematical operation depending on the analysis being performed (e.g., $\sqsubseteq$ is $\supseteq$ for live variable analysis). Posets which are defined by dataflow problems often have an element which is safer than any other element in the poset. If exists, such element is called the *least element*, or *bottom*, and it is denoted as the symbol $\bot$. The *greatest element*, or *top*, defined similarly, is denoted by $\top$.

In a poset two important elements may exist:

Definition 40 – *join*

Let us define the *join*, or the *lower upper bound* (lub) of a set B as an element x such that (i) x is an upper bound of B and (ii) for all other upper bounds $y \in B$, $x \sqsubseteq y$. This element is denoted as $\bigsqcap B$. The lub of two elements can be also expressed as $x \sqcap y$.

Definition 41 – *meet*

Let us define the *meet*, or the glb of a set B as an element x such that (i) x is a lower bound of B and (ii) for all other lower bounds $y \in B$, $y \sqsubseteq x$. This element is denoted as $\bigsqcup B$. The glb of two elements can be also expressed as $x \sqcup y$.

Both the *join* and *meet* of a set, if they exist, are unique. In the context of data flow analysis, the meet operator is used to merge data flow values along different paths and reaching a join node of the underlying CFG. The result of the meet operation is the most exhaustive safe approximation of data flow values along each of the paths.

Definition 42 – *Meet semilattice*

A poset $(L, \sqsubseteq)$ is a *meet semilattice*, if and only if for each non-empty finite $B \subseteq L$, $\bigsqcap B \in L$.

Since possible dataflow values are element of a meet semilattice several considerations can be made on the convergence to a fixpoint solution. A fixpoint of a function $f : L \to L$ is a value $v \in L$ that satisfies $f(v) = v$. In particular, if the function being applied to the dataflow variables iteratively (also called *flow function*) is *monotonic* it can be proved that the fixpoint solution will be reached [102].

A basic block, n, of the CFG has two variables associated: $In_n, Out_n \in L$. The former is the dataflow value available at the entry point of the block while the latter is the value at the exit point. The dataflow value associated with In_n is computed based on the dataflow information generated by preceding nodes, $Pred(n)$, using the following formula:

$$In_n = \begin{cases} BI & \text{if } n \text{ is } entry \text{ block} \\ \bigsqcap_{p \in Pred(n)} f_p(In_p) & \text{otherwise} \end{cases} \tag{D.1}$$

Where $f_n(In_n)$ is the *flow (or transfer) function* which specifies which dataflow information is generated or destroyed by a generic CFG block. The equation presented above is characteristic of forward dataflow problems where information are propagated accordingly to the direction of the CFG's edges. However backward analysis can be defined by replacing, in the equation, In_n with Out_n.

In general, definition of the transfer function may depend on the value of the dataflow information at the entry of the block. In literature this is known as a *non-separable* dataflow framework [102]. Transfer functions of a non-separable dataflow analysis is characterized by four components, a generic $f_n(In_n)$ for a non-separable framework is defined as follows:

$$f_n(x) = (x - (ConstKill_n \cup DepKill_n(x))) \cup (ConstGen_n \cup DepGen_n(x))$$

Where:

$ConstGen_n$**:** dataflow information generated by the basic block n;

$ConstKill_n$**:** dataflow information killed (or destroyed) by the basic block n;

$DepGen_n(x)$**:** dataflow information generated by the basic block n which depends on the dataflow value in input to block n;

$DepKill_n(x)$**:** dataflow information killed by the basic block n which depends on the dataflow value in input to block n.

An example of forward dataflow analysis is the reaching definitions:

> A definition $d_i \in \mathbb{D}efs$ of variable $x \in \mathbb{V}ar$ reaches a program point u if d_i occurs on some path from *entry* to u and is not followed by any other definition of x on this path.

The lattice L for this problem is composed by the powerset of the set $\mathbb{D}efs$. The initial value associated to the *entry* block, BI, and as well to the other blocks in the CFG is the empty set, $\emptyset$. The confluence, or meet operator $\bigsqcup$, is the set union operation $\bigcup$. This captures the "any path" nature of the dataflow since join point of the CFG should propagate information coming from every incoming edge. Gen_n contains downwards exposed definitions in n whereas $Kill_n$ contains all definitions of all variables modified in n. Both $DepGen_n(x)$ and $DepKill_n(x)$ are not defined since there are no dependent components. An instance of non-separable dataflow analysis will be presented in chapter 5 of this thesis.

The dataflow iterative framework allows the definition of new analysis in a declarative way. Its importance is well understood and almost every compiler integrates a dataflow solver. As part of this thesis work, a dataflow solver has been integrated in the Insieme compiler, presented in Appendix C.

Appendix E

Acronyms

ANN Artificial Neural Network ... xii

ANOVA Analysis of Variance ... xii

API Application Programming Interface ... iv

AST Abstract Syntax Tree ... xi

bps bits per second ... 13

BSC Barcelona Supercomputing Center ... xi

BTL Byte Transfer Layer ... 86

ccNUMA Cache coherent NUMA ... 11

CFG Control Flow Graph ... xi

CICO copyin/copyout semantics ... 127

CPU Central Processing Unit ... 9

CUDA Compute Unified Device Architecture ... 62

DAG Directed Acyclic Graph ... xi

DCR Dynamic Collective Replacement ... xi

DDG Data Dependence Graph ... xii

DMA Direct Memory Access ... 20

DSL Domain Specific Language ... 82

FIFO First In, First Out ... 17

FLOPS FLoating-point Operations Per Second ... 10

glb *greatest lower bound* ... 145

GPU Graphics Processing Unit ... xi

GPGPU General Purpouse Graphics Processing Unit . . . 175
HPC High-Performance Computing . . . iv
IC Iterative Compilation . . . 85
IFT Iterative Feedback-driven Tuning . . . 85
ILP Instruction Level Parallelism . . . 9
IPS instructions per second . . . 10
IR Intermediate Representation . . . 27
LOOCV Leave-One-Out Cross Validation . . . 105
LTO Link Time Optimizations . . . 157
lub *lower upper bound* . . . 200
MCA Modular Component Architecture . . . 84
ML Machine Learning . . . xii
MLP Multi-Layer Perceptron . . . 105
MPI Message Passing Interface . . . iv
MPL Meta-Programming Library . . . 57
MPMD Multiple Program Multiple Data . . . xiii
MPP MPI CPP Interface . . . xi
NiC Network interface Controller . . . 12
NPB NAS Parallel Benchmarks . . . xi
NUMA Non-Uniform Memory Access . . . 11
OOMPI Object-Oriented MPI . . . 3
OOP Object-Oriented Programming . . . 4
OpenCL Open Computing Language . . . iv
OpenMP Open Multi-Processing . . . 38
OPTO Open Tool for Parameter Optimization . . . 85
OPUB Observed Performance Upper-Bound . . . xii
OS operating system . . . 13
pCFG parallel CFG . . . 156
PGAS Partitioned Global Address Space . . . 3
PM Polyhedral Model . . . iv
RAW Read-After-Write . . . 32

Bibliography

[1] Matthias S. Müller, G. Matthijs van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C. Brantley, Chris Parrott, Tom Elken, Huiyu Feng, and Carl Ponder. SPEC MPI2007 – an application benchmark suite for parallel systems using MPI. *Concurrency and Computation: Practice and Experience*, 22(2):191–205, 2010.

[2] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., 2006.

[3] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3, 2012. URL http://www.mpi-forum.org.

[4] Jeff Squyres Bill, Bill Saphir Y, and Andrew Lumsdaine Z. The Design and Evolution of the MPI-2 C++ Interface. In *In Proceedings, 1997 Internantional Conference on Scientific Computing in Object-Oriented Parallel Computing, Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[5] Prabhanjan Kambadur, Douglas Gregor, Andrew Lumsdaine, and Amey Dharurkar. Modernizing the C++ Interface to MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Lecture Notes in Computer Science, pages 266–274. Springer Berlin / Heidelberg, 2006.

[6] B. C. McCandless, J. M. Squyres, and A. Lumsdaine. Object-Oriented MPI (OOMPI): A class library for the message passing interface. In *Proceedings of the Second MPI Developers Conference*, pages 87–. IEEE Computer Society, 1996.

[7] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[8] Matthew J. Koop, Terry Jones, and Dhabaleswar K. Panda. Reducing Connection Memory Requirements of MPI for InfiniBand Clusters: A Message Coalescing Approach. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '07, pages 495–504, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2833-3.

[9] Torsten Hoefler and Timo Schneider. Runtime Detection and Optimization of Collective Communication Patterns. In *PACT*, pages 263–272, 2012.

[10] Andreas Knüpfer, Dieter Kranzlmüller, and Wolfgang E. Nagel. Detection of Collective MPI Operation Patterns. In *PVM/MPI*, pages 259–267, 2004.

[11] Mohamad Chaarawi, Je M Squyres, Edgar Gabriel, and Saber Feki. A Tool for Optimizing Runtime Parameters of Open MPI. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 210–217, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, LCTES '99, pages 1–9, 1999.

[13] Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332, Washington, DC, USA, 2006. IEEE Computer Society.

[14] R. P. J. Pinkers, P. M. W. Knijnenburg, M. Haneda, and H. A. G. Wijshoff. Statistical selection of compiler options. In *MASCOTS '04: Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 494–501, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Anthony Danalis, Lori Pollock, Martin Swany, and John Cavazos. MPI-aware compiler optimizations for improving communication-computation overlap. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 316–325, 2009. ISBN 978-1-60558-498-0.

[16] Thomas Fahringer and Eduard Mehofer. Buffer-safe communication optimization based on data flow analysis and performance prediction. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*, pages 189–200. IEEE Computer Society Press, 1997.

[17] Beniamino Di Martino, Antonino Mazzeo, Nicola Mazzocca, and Umberto Villano. Parallel program analysis and restructuring by detection of point-to-point interaction patterns and their transformation into collective communication constructs. *Science of Comp. Prog.*, 40, 2001.

[18] Greg Bronevetsky. Communication-Sensitive Static Dataflow for Parallel Message Passing Applications. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3576-0.

[19] Erich Strohmaier. TOP500 - TOP500 supercomputer. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA*, page 18. ACM Press, 2006.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0123704901.

[21] Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, October 1985.

[22] Clark Robert G. Wilson, Leslie B. *Comparative programming languages.* Addison-Wesley, 2001.

[23] Cédric Bastoul. *Improving Data Locality in Static Control Programs.* PhD thesis, University Paris 6, Pierre et Marie Curie, France, December 2004.

[24] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *ETAPS CC*, March 2010.

[25] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, June 2006. Classement CORE : A.

[26] Alexander I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Math. Oper. Res.*, 19(4):769–779, November 1994.

[27] Nicolas Vasilache, Cedric Bastoul, Albert Cohen, and Sylvain Girbal. Violated dependence analysis. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 335–344, New York, NY, USA, 2006. ACM.

[28] Michael Joseph Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, MA, USA, 1990. ISBN 0262730820.

[29] Utpal Banerjee. *Dependence analysis for supercomputing.* Kluwer Academic, Boston, MA, USA, 1988.

[30] MPI Forum. The MPI-1 Specification, . URL http://www.mpi-forum.org/docs/docs.html.

[31] Anthony Skjellum, Diane G. Wooley, Andrew Lumsdaine, Ziyang Lu, Michael Wolf, Jeffrey M. Squyres, Brian Mccandless, and Purushotham V. Bangalore. Object-oriented analysis and design of the message passing interface, 1998.

[32] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59. ACM, 1977.

[33] Jarek Nieplocha, Bruce Palmer, Manojkumar Krishnan, Harold Trease, Edoardo Aprá, Jarek Nieplocha, Bruce Palmer, Manojkumar Krishnan, Harold Trease, and Edoardo Aprá. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *Intern. J. High Perf. Comp. Applications*, 20, 2005.

[34] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17:1–31, August 1998.

[35] UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab Tech Report, 2005.

[36] Ivan Grasso, Simone Pellegrini, Biagio Cosenza, and Thomas Fahringer. libwater: Heterogeneous distributed computing made easy. In *ICS*, pages 161–172, 2013.

[37] C99 standard. URL www.open-std.org/JTC1/SC22/wg14/www/docs/n1124.pdf.

[38] Robert Ramsey. Boost Serialization Library. URL www.boost.org/doc/libs/release/libs/serialization/.

[39] Simone Pellegrini. MPI C++ Interface, . URL https://github.com/motonacciu/mpp.

[40] John Burkardt. QUAD_MPI. URL http://people.sc.fsu.edu/~jburkardt/c_src/quad_mpi/quad_mpi.html.

[41] Andrei Alexandrescu. Traits: The else-if-then of Types. In *C++ Report*, pages 22–25, 2000. URL http://erdani.com/publications/traits.html.

[42] David Abrahamsi and Aleksey Gurtovoy. *C++ Template Metaprogramming*. Addison-Wesley, 2006.

[43] Robert W. Numrich and John Reid. Co-arrays in the next fortran standard. *SIGPLAN Fortran Forum*, 24(2):4–17, August 2005.

[44] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3), August 2007.

[45] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, 2005.

[46] D. Bonachea. GASNet Specification, v1.1. Technical Report UCB/CSD-02-1207, U.C. Berkeley, 2002.

[47] M.M. Strout, B. Kreaseck, and P.D. Hovland. Data-Flow Analysis for MPI Programs. In *Parallel Processing, 2006. ICPP 2006. International Conference on Parallel Processing*, pages 175 –184, aug. 2006.

[48] PAPI: Performance Application Programming Interface. URL http://icl.cs.utk.edu/papi/.

[49] Aleksey Gurtovoy and David Abrahams. The Boost Metaprogramming Library. URL http://www.boost.org/doc/libs/1_46_1/libs/mpl/doc/index.html.

[50] Dan Bonachea Paul H. Hargrove Steven Hofmeyr Costin Iancu Seung-Jai Min Katherine Yelick Yili Zheng, Filip Blagojevic. Getting Multicore Performance with UPC. In *SIAM Conference on Parallel Processing for Scientific Computing*, Seattle, Washington, February 2010.

[51] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *ICS*, pages 341–352, 2012.

[52] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A Package for OpenCL Based Heterogeneous Computing on Clusters with Many GPU Devices. In *Workshop PPAC*, pages 224–231, 2010.

[53] Ryo Aoki, Shuichi Oikawa, Takashi Nakamura, and Satoshi Miki. Hybrid OpenCL: Enhancing OpenCL for Distributed Processing. In *ISPA*, pages 149–154, 2011.

[54] Alves Albano, Rufino Jose, Pina Antonio, and Santos Luis Paulo. clOpenCL - Supporting Distributed Heterogeneous Computing in HPC Clusters. In *10th International Workshop HeteroPar*, 2012.

[55] Magnus Strengert, Christoph Müller, Carsten Dachsbacher, and Thomas Ertl. CUDASA: Compute Unified Device and Systems Architecture. In *EGPGV*, pages 49–56, 2008.

[56] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *HPCS*, pages 224–231, 2010.

[57] Orion S. Lawlor. Message passing for GPGPU clusters: CudaMPI. In *CLUSTER*, pages 1–8, 2009.

[58] Sameer Kumar, Gabor Dozsa, Gheorghe Almasi, Philip Heidelberger, Dong Chen, Mark E. Giampapa, Michael Blocksome, Ahmad Faraj, Jeff Parker, Joseph Ratterman, Brian Smith, and Charles J. Archer. The deep computing messaging framework: generalized scalable message passing on the Blue Gene/P supercomputer. In *ICS*, pages 94–103, 2008.

[59] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, and Derrick Weathersby. Efficient Algorithms for All-to-All Communications in Multi-Port Message-Passing Systems. In *SPAA*, pages 298–309, 1994.

[60] Jelena Pjesivac-Grbovic, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[61] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, 35(12):581–594, 2009.

[62] The Vienna Scientific Cluster 2, 2012. URL http://www.vsc.ac.at.

[63] The MinoTauro GPU Cluster, 2013. URL http://www.bsc.es/marenostrum-support-services/\other-hpc-facilities/nvidia-gpu-cluster.

[64] Wei Wang, Hanli Wang, Dong Guo, Haoyang Wei, and Guosun Zeng. Parallel time-space processing model based fast N-body simulation on GPUs. In *PMAM*, 2013.

[65] Jeroen Bédorf, Evghenii Gaburov, and Simon Portegies Zwart. A sparse octree gravitational N-body code that runs entirely on the GPU processor. *J. Comput. Physics*, 231(7):2825–2839, 2012.

[66] Tsuyoshi Hamada and Keigo Nitadori. 190 TFlops Astrophysical N-body Simulation on a Cluster of GPUs. In *SC*, 2010.

[67] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. Scaling Hierarchical N-body Simulations on GPU Clusters. In *SC*, 2010.

[68] Ivan Grasso, Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. Automatic problem size sensitive task partitioning on heterogeneous parallel systems. In *PPoPP*, 2013.

[69] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An automatic input-sensitive approach for heterogeneous task partitioning. In *ICS*, 2013.

[70] Ma Kai, Li Xue, Chen Wei, Zhang Chi, and Wang Xiaorui. GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU Heterogeneous Architectures. In *ICPP*, 2012.

[71] Dominik Grewe and Michael F.P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *CC*, 2011.

[72] H. J. C. Berendsen, David van der Spoel, and R. van Drunen. GROMACS: a message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 91(1/3), September 1995.

[73] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology (KTH), Stockholm, December 2003.

[74] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[75] MVAPICH Team. MVAPICH 1.0 User and Tuning Guide, 2008.

[76] Open MPI. Modular Component Architecture, 2000. URL http://www.open-mpi.org/faq/?category=tuning.

[77] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. Iterative compilation. pages 171–187, 2002.

[78] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0.

[79] Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. ISBN 1-55860-510-X.

[80] Douglas C. Montgomery. *Design and Analysis of Experiments, Student Solutions Manual*. Wiley, August 2005. ISBN 0471733040.

[81] Rob F. Van der Wijngaart. Nas Parallel Benchmarks Version 3.3. Technical Report NAS-02-007, Computer Science Corporation NASA Advanced Supercomputing (NAS) Division, October 2002.

[82] Parallel Benchmarks Ahmad, Ahmad Faraj, and Xin Yuan. Communication characteristics in the nas. In *In Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002*, pages 729–734, 2002.

[83] H. S. Hall and S. R. Knight. *Higher algebra*. Reem Publications, 2011.

[84] Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the atlas project. *PARALLEL COMPUTING*, 27:2001, 2000.

[85] M. Frigo and S.G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384 vol.3, 1998.

[86] Graham E. Fagg, Thara Angskun, George Bosilca, and Jack J. Dongarra. Decision trees and mpi collective algorithm selection problem. 2006.

[87] K. D. Cooper, T.J. Grosul, A.and Harvey, S. Reeves, D. Subramanian, L. Torzon, and T. Waterman. Exploring the structure of the space of compilation sequences using randomized search algorithms. In *In the Proceedings of the 2004 LACSI Symposium*, pages 295–305. IEEE Computer Society, 2004.

[88] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991.

[89] T. W. Anderson and Theodore W. Anderson. *An Introduction to Multivariate Statistical Analysis, 2nd Edition*. Wiley, 2 edition, September 1984.

[90] Ethem Alpaydin. *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2004. ISBN 0262012111.

[91] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, January 1996. ISBN 978-0198538646.

[92] Peter Dalgaard. *Introductory Statistics with R (Statistics and Computing)*. Springer, 2nd edition, August 2008.

[93] MPI Forum. Message Passing Interface (MPI) Forum Home Page. `http://www.mpi-forum.org/` (Dec. 2009), .

[94] Galen Mark Shipman, Tim S. Woodall, George Bosilca andRich L. Graham, and Arthur B. Maccabe. High performance RDMA protocols in HPC. In *Proceedings, 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Bonn, Germany, September 2006. Springer-Verlag.

[95] Wei-Yu Chen, Costin Iancu, and Katherine Yelick. Communication optimizations for fine-grained upc applications. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 267–278, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2429-X.

[96] Kirk W. Cameron and Xian-He Sun. Quantifying Locality Effect in Data Access Delay: Memory logP. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1926-1.

[97] A. Danalis, L. Pollock, and M. Swany. Automatic MPI application transformation with ASPhALT. In *Par. and Distr. Proc. Symp., IPDPS 2007*, pages 1 –8, Mar. 2007.

[98] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany. Transformations to parallel codes for communication-computation overlap. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 58–, Washington, DC, USA, 2005. ISBN 1-59593-061-2.

[99] Andreas Knüpfer et al. The vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. 2008. ISBN 978-3-540-68564-7.

[100] M.-W. Benabderrahmane et al. The polyhedral model is more widely applicable than you think. In *Proc. of the Intl. Conf. on Compiler Constr.*, LNCS, March 2010.

[101] Nicolas Vasilache, Albert Cohen, Cedric Bastoul, and Sylvain Girbal. Violated dependence analysis. In *In ACM ICS*, 2006.

[102] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. 2009.

[103] Thomas Fahringer and Eduard Mehofer. Buffer-safe and cost-driven communication optimization. *Journal of Parallel and Distributed Computing, Academic Press*, 57, 1999.

[104] L. R. Ford and D. R. Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[105] Kevin London, Shirley Moore, Philip Mucci, Keith Seymour, and Richard Luczak. The PAPI Cross-Platform Interface to Hardware Performance Counters. In *Department of Defense Users Group Conference Proceedings*, pages 18–21, 2001.

[106] Ralf Reussner, Peter Sanders, and Jesper Larsson Träff. SKaMPI: a comprehensive benchmark for public benchmarking of MPI. *Sci. Program.*, 10(1):55–65, January 2002. ISSN 1058-9244.

[107] Duncan Grove and Paul Coddington. Precise MPI Performance Measurement Using MPIBench. In *In Proceedings of HPC Asia*, 2001.

[108] P Coddington. Comparison of MPI Benchmark Programs on Shared Memory and Distributed Memory Machines (Point-to-Point Communication). *International Journal of High Performance Computing Applications*, 24(4):469–483, 2010.

[109] William Gropp and Ewing L. Lusk. Reproducible measurements of mpi performance characteristics. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 11–18, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66549-8.

[110] Simone Pellegrini. The MPI Cache Benchmark, . URL https://github.com/motonacciu/mpi-cache-bench.

[111] Hyun-Wook Jin, Sayantan Sur, Lei Chai, and Dhabaleswar K. Panda. Lightweight kernel-level primitives for high-performance mpi intra-node communication over multi-core systems. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, CLUSTER '07, pages 446–451, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 978-1-4244-1387-4.

[112] Open MPI. URL http://www.open-mpi.org.

[113] T.S. Woodall, R.L. Graham, R.H. Castain, D.J. Daniel, M.W. Sukalski, G.E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. TEG: A high-performance, scalable, multi-network point-to-point communications methodology. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 303–310, Budapest, Hungary, September 2004.

[114] Darius Buntinas and Guillaume Mercier. Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In *Proceedings*

of the International Symposium on Cluster Computing and the Grid, pages 521–530. IEEE Computer Society, 2006.

[115] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI (2nd ed.): portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-57132-3.

[116] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. San Francisco, CA, USA, 2002. ISBN 1-55860-286-0.

[117] Armin Grlinger. *The Challenges of Non-linear Parameters and Variables in Automatic Loop Parallelisation*. Lulu Enterprises, UK Ltd, 2010. ISBN 1445254212, 9781445254210.

[118] R. Preissl, T. Kockerbauer, M. Schulz, D. Kranzlmuller, B. Supinski, and D.J. Quinlan. Detecting Patterns in MPI Communication Traces. In *ICPP*, pages 230 –237, 2008.

[119] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In *ICS*, pages 353–360, 2006.

[120] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20:287–311, May 2006.

[121] Robert Preissl, Martin Schulz, Dieter Kranzlmüller, Bronis R. Supinski, and Daniel J. Quinlan. Using MPI Communication Patterns to Guide Source Code Transformations. In *ICCS*, pages 253–260, 2008.

[122] Jidong Zhai, Tianwei Sheng, Jiangzhou He, Wenguang Chen, and Weimin Zheng. FACT: fast communication trace collection for parallel applications through program slicing. In *SC*, pages 27:1–27:12, 2009.

[123] Mark Weiser. Program slicing. In *ICSE*, pages 439–449, 1981.

[124] Valérie Pascual and Laurent Hascoët. Native Handling of Message-Passing Communication in Data-Flow Analysis. In *Recent Advances in Algorithmic Differentiation*, pages 83–92. 2012.

[125] Stephen F. Siegel and George S. Avrunin. Modeling wildcard-free MPI programs for verification. In *PPoPP*, pages 95–106, 2005.

[126] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. Formal verification of practical MPI programs. In *PPoPP*, pages 261–270, 2009.

[127] Anh Vo, Ganesh Gopalakrishnan, Robert M. Kirby, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. Large Scale Verification of MPI Programs Using Lamport Clocks with Lazy Update. In *PACT*, pages 330–339, 2011.

[128] Sven Verdoolaege. isl: an integer set library for the polyhedral model. In *ICMS*, pages 299–302, 2010.

[129] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. In *Algorithmica*, 2007.

[130] Wayne Kelly and William Pugh. A unifying framework for iteration reordering transformations. In *ICA3PP*, 1995.

[131] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. Technical report, 1991.

[132] John Tramm. MPI Particle Simulator, 2012. URL https://github.com/jtramm/particle-mpi.

[133] Dimitrios Vitsios. MPI Bitonic Sort, 2009. URL https://github.com/dvitsios/biSort_mpi.

[134] Herbert Jordan, Peter Thoman, Juan J. Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 10:1–10:12, 2012.

[135] Peter Thoman, Herbert Jordan, Simone Pellegrini, and Thomas Fahringer. Automatic openmp loop scheduling: A combined compiler and runtime approach. In *OpenMP in a Heterogeneous World*, volume 7312 of *Lecture Notes in Computer Science*, pages 88–101. 2012.

[136] Jordan Herbert, Pellegrini Simone, Thoman Peter, Kofler Klaus, and Thomas Fahringer. INSPIRE: The Insieme Parallel Intermediate Representation. In *22th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Edinburgh, Scotland, sept. 2013.

[137] DPS University of Innsbruck. The INSIEME Compiler System. URL http://www.dps.uibk.ac.at/projects/insieme.

[138] LLVM Team. Clang: A C language family frontend for LLVM, 2009. URL http://clang.llvm.org/.