Evaluating techniques for parallelization tuning in MPI, OmpSs and MPI/OmpSs

Author:
Vladimir SUBOTIĆ

Advisors:
Prof. Jesús LABARTA
Prof. Mateo VALERO
Prof. Eduard AYGUADÉ

A THESIS SUBMITTED IN FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Doctor per la Universitat Politècnica de Catalunya
Departament d'Arquitectura de Computadors
Barcelona, 2013

Abstract

Parallel programming is used to partition a computational problem among multiple processing units and to define how they interact (communicate and synchronize) in order to guarantee the correct result. The performance that is achieved when executing the parallel program on a parallel architecture is usually far from the optimal: computation unbalance and excessive interaction among processing units often cause lost cycles, reducing the efficiency of parallel computation.

In this thesis we propose techniques oriented to better exploit parallelism in parallel applications, with especial emphasis in techniques that increase asynchronism. Theoretically, this type of parallelization tuning promises multiple benefits. First, it should mitigate communication and synchronization delays, thus increasing the overall performance. Furthermore, parallelization tuning should expose additional parallelism and therefore increase the scalability of execution. Finally, increased asynchronism would allow more flexible communication mechanisms, providing higher tolerance to slower networks and external noise.

In the first part of this thesis, we study the potential for tuning MPI parallelism. More specifically, we explore automatic techniques to overlap communication and computation. We propose a speculative messaging technique that increases the overlap and requires no changes of the original MPI application. Our technique automatically identifies the application's MPI activity and reinterprets that activity using optimally placed non-blocking MPI requests. We demonstrate that this overlapping technique increases the asynchronism of MPI messages, maximizing the overlap, and consequently leading to execution speedup and higher tolerance to bandwidth reduction. However, in the case of realistic scientific workloads, we show that the overlapping potential is significantly limited by the pattern by which each MPI process locally operates on MPI messages.

In the second part of this thesis, we study the potential for tuning hybrid MPI/OmpSs applications. We try to gain a better understanding of the parallelism of hybrid MPI/OmpSs applications in order to evaluate how these applications would execute on future machines and to predict the execution bottlenecks that are likely to emerge. We explore how MPI/OmpSs applications could scale on the parallel machine with hundreds of cores per node. Furthermore, we investigate how this high parallelism within each node would reflect on the constraints of the interconnect. We especially focus on identifying critical code sections in MPI/OmpSs. We devised a technique that quickly evaluates, for a given MPI/OmpSs application and the selected target machine, which code section should be optimized in order to gain the highest performance benefits.

Also, this thesis studies techniques to quickly explore the potential OmpSs parallelism inherent in applications. We provide mechanisms for the programmer to easily evaluate potential parallelism of any task decomposition. Furthermore, we describe an iterative trial-and-error approach to search for a task decomposition that will expose sufficient parallelism for a given target machine. Finally, we explore potential of automating the iterative approach by capturing the programmers' experience into an expert system that can autonomously lead the process of finding efficient task decompositions.

Also, throughout the work on this thesis, we designed development tools that can be useful to other researchers in the field. The most advanced of these tools is Tareador – a tool to help porting MPI applications to MPI/OmpSs programming model. Tareador provides a simple interface to propose some decomposition of a code into OmpSs tasks. Then, based on the proposed decomposition, Tareador dynamically calculates data dependencies among the annotated tasks, and automatically estimates the potential OmpSs parallelization. Furthermore, Tareador gives additional hints on how to complete the process of porting the application to OmpSs. Tareador already proved itself useful, by being included in the academic classes on parallel programming at UPC.

Acknowledgement

On January 12th 2007, I started my Ph.D. studies at the UPC. In the first two hours of "sitting in front of computer", I managed to format my Linux partition. Today, I'm about to defend my doctorate in computer science. I'm not the same guy from my first day, both personally and professionally. Many people helped me on my way. I owe them many thanks.

I'm very thankful to Jesus, who taught me to be methodological and direct – I would say a brutal engineer.

I'm thankful to Mateo, whose encouragement and support I knew to recognize and appreciate fully always with a significant delay.

I must thank Edu for "adopting" me close to the end of my studies and helping me finish my Ph.D. in a surprisingly good mood.

I'm thankful to Jose Carlos, who enormously helped me writing papers.

Also, I wish to thank my numerous colleagues from Barcelona Supercomputing Center who shared with me their working and non-working hours and made my Ph.D. time both more productive and enjoyable. I must pick out Saša and Srdjan, who showed me many tricks of the trade. And, I would like to specially mention Uri Prat, a big guy who had amazing patience with me, even in the days when I was a technical idiot.

Finally, this thesis would not be possible without my girlfriend Jelena. She has stuck with me through thick and thin.

Errata

My graduate work has been financially supported by the following projects:

- Computacion de altas prestaciones V (TIN2007-60625).
- TEXT – Towards EXaflop applicaTions. EU Commission, Information Society Technologies, IST-2007-261580.
- BSC-IBM MareIncognito project. IBM Research.
- Intel-BSC Exascale Lab. Intel Corporation.

Also, throughout my Ph.D. studies, I was receiving a grant from Barcelona Supercomputing Center.

Contents

List of Figures

List of Tables

1 Introduction

Technological evolution demands increasing amounts of computational power and causes appearance of brand new large-scale parallel machines. Today, virtually all science and engineering is based on high-accuracy numerical simulations. These simulations are extremely computation-intensive programs. In order to finish the computation in limited time, these programs must execute in parallel. Therefore, parallel processing is becoming an indispensable part of modern science.

However, it is very hard to make a parallel machine work efficiently. As the number of processors in a system grows to hundreds and beyond, organizing inter-processor communication becomes a very important issue. In such large systems, inefficient communication and synchronization can introduce long execution stalls. These stalls prevent the processors from computing useful work and seriously harm performance.

In order to eliminate processor stalls, the computer architect's community tries to accelerate communication and synchronization mechanisms by investing more in interconnects. Hardware vendors constantly deliver more powerful networks. These networks constantly advance technological parameters, providing higher bandwidth and lower latency. As a result of this trend, the cost of the interconnect is becoming a significant share in the total cost of these parallel machines [32]. Moreover, the trend of the Top 500 list [87] forecasts that the share of the interconnect in both power and cost of the whole system will be increasing.

Therefore, the trend of simply improving technological parameters of interconnects, and paying for those improvements, becomes economically unsustainable. Many recent studies show that, in High Performance Computing (HPC), interconnection networks are over-designed and yet underutilized [35]. Also, despite their increasing cost, new high-end interconnects deliver just a slight improvement in the overall performance. Thus, the community must optimize the utilization of network resources. Rather than trying to create new, faster and lately very expensive cycles in the network, the community must learn how to profit more from the already existing cycles.

To tackle the issue of under-utilization of parallel systems, in this thesis we explore techniques for parallelization tuning. We especially target techniques that increase asynchronism in parallel execution. Theoretically, this type of parallelization tuning promises multiple benefits. First, it should mitigate communication and synchronization delays, thus increasing the overall performance. Furthermore, parallelization tuning should expose additional parallelism and therefore increase the scalability of execution. Finally, increased asynchronism would allow more flexible communication mechanisms, providing higher tolerance to slower networks, as well as to external noise.

1.1 Goals

The major goal of this thesis is to explore techniques for parallelization tuning in MPI and MPI/OmpSs. In studying MPI execution, we evaluate automatic techniques for hiding communication delays. We explore techniques that can automatically overlap communication and computation, without the need to refactor the original legacy MPI code. Furthermore, we explore techniques for optimizing MPI/OmpSs execution. We specially focus on pinpointing bottlenecks of MPI/OmpSs execution. Also, we explore how MPI/OmpSs execution can be improved by changing the task decomposition of the code.

Also, throughout the work on this thesis, we strove to produce a useful development infrastructure that can be used by both research and industry. Nowadays, very few people can efficiently program in parallel [38, 49]. However, as the existing hardware becomes increasingly parallel, parallel programming will inevitably become mainstream. Therefore, this thesis also aims to produce new development environments that can further increase the overall understanding of parallelism. We hope that our development environments can facilitate the adoption of parallel programming, especially the MPI/OmpSs programming model.

1.2 Approach

The experimental part of this thesis is based on trace-driven simulation. Initially, our study targets parallelization tuning of MPI execution. However, simulating MPI execution is a very hard problem. The simulation must process high number of separate MPI processes that among themselves communicate, synchronize and share resources (e.g. interconnection network). Thus, simulators of MPI execution are very computation-intensive and hard to parallelize. Our experience tells us that the mainstream solution for simulating MPI execution is a trace-driven simulation at a high level of abstraction.

We customize the conventional trace-driven methodology, by migrating the feature modeling effort from the simulation phase to the tracing phase. In the conventional trace-driven simulation, modeling a new feature is done in the simulation phase. The tracer instruments the application and collects the trace of the run. Then, the simulator replays the collected trace to reconstruct the time-behavior of the studied application on a parallel machine. When testing a new feature, the developer must extend the simulator in order to incorporate the effects of the inspected feature into the resulting time-behavior. On the other hand, in this thesis, we design a novel simulation methodology in which modeling a new feature is done in the tracing phase. In our simulation approach, we extend the tracer in order to introduce the effects of the inspected feature already into the collected trace. Then, the unchanged simulator can replay the collected trace and propagate the effect of the modeled feature throughout the simulated parallel execution.

Our methodology allowed us to simulate effects of low-level features in large-scale systems. Our approach stresses the tracing part of the simulation, by moving the feature modeling effort into this phase. Since each MPI process is traced concurrently, the intensive computation for feature modeling is naturally parallelized across all MPI processes. This parallelization of feature modeling computation allowed us to build very powerful environments. In many developed tools, we implement tracers based on binary translation environments. These tracers instrument execution at the level of a single instruction and model very low-level features. On the other hand, an unchanged MPI simulator propagates the effects of the modeled feature throughout a very large-scale parallel machine.

Based on the presented methodology, we created new useful development environments. More specifically, we designed three development environments. The first environment automatically evaluates the potential of communication/computation overlap in MPI applications. The second environment extends the legacy MPI replaying toolchain (mpitrace, Dimemas) into a new MPI/OmpSs replaying toolchain (mpisstrace, Dimemas). Finally, the last environment is Tareador – a tool that guides parallelization of sequential applications. Tareador showed to be especially useful for teaching parallel programming, so it was included in the academic program of UPC.

1.3 Contributions

The first contribution of the thesis consists of exploring techniques for parallelization tuning. The goal of these techniques is to hide communication delays and increase parallelism in scientific parallel applications:

1. In our study of techniques for tuning parallelism in MPI execution, we focus on exploring the potential of communication-computation overlap. We introduce *speculative dataflow* – a technique that automatically overlaps communication and computation in MPI applications, without the need to restructure the target legacy code. We describe the protocol of speculative dataflow, prove its feasibility and demonstrate its potential benefits in characteristic MPI applications. Furthermore, we show that in real scientific applications, overlap can achieve significant execution speedup, as well as higher tolerance to bandwidth reduction. However, we point out that the potential overlap is often seriously limited by the applications intrinsic computation pattern. In the case of one application, we illustrate how that computation pattern can be changed by refactoring the ap-

plication. However, we conclude that it is impractical to do such manual refactoring in each application. In a search for a dynamic technique for restructuring computation patterns, we start our study of task-based dataflow programming models (OmpSs and MPI/OmpSs).

2. In our study of techniques for tuning parallelism in MPI/OmpSs execution, we first identify parallelization bottlenecks in existing unmodified MPI/OmpSs applications. The years of practice in optimizing applications points that the major issue is *focus* – identifying the code section whose optimization would yield the highest overall applications speedup. We illustrate that, due to the irregular parallelism of MPI/OmpSs, the programmer can hardly identify the critical code section. Furthermore, we demonstrate that in many applications, the choice of the critical section decisively depends on the configuration of the target machine. For instance, in HP Linpack, optimizing a task that takes 0.49% of the total computation time yields the overall speedup of less than 0.25% on one machine and at the same time yields the overall speedup of more than 24% on a different machine. To tackle this issue, we devised an automatic approach that, for a given target parallel machines, identifies the critical code sections of an MPI/OmpSs application. Compared to the state-of-the-art research, our approach accounts for more influences, and estimates the potential benefits of the optimization in advance, before incurring into any coding efforts.

3. We further explore techniques to introduce OmpSs parallelism into existing MPI applications. OmpSs potentially extracts very irregular parallelism, parallelism that the programmer cannot identify himself. Thus, for a programmer without any development support, it is very hard to anticipate whether some task decomposition exposes parallelism or not. To that end, we provide mechanisms for the programmer to evaluate quickly the potential parallelism of any OmpSs task decomposition. Furthermore, we describe an iterative trial-and-error approach to search for a task decomposition that will expose sufficient parallelism for a given target machine. Finally, we explore the potential of automating the iterative approach by capturing the programmers' experience into an expert system that can autonomously lead the process of finding efficient task decompositions.

The second contribution of the thesis consists in creating an infrastructure that can be used for future research, as well as for educating programmers about parallel programming. To that end, we developed two environments:

1. **mpisstrace** – an environment for replaying MPI/OmpSs parallel execution. Already existing BSC tools allow tracing MPI execution with mpitrace and replaying that execution with Dimemas. Based on this infrastructure, we extended MPI tracing library in order to instrument MPI/OmpSs codes and Dimemas in order to support task-based dataflow execution. As the result, we obtained a framework that is fully compatible with the legacy BSC tool-chain, but also supports simulating MPI/OmpSs execution.

2. **Tareador** – a tool to assist porting MPI applications to MPI/OmpSs. Having an MPI application, the programmer can very simply propose some decomposition of the code into tasks. Then, Tareador dynamically identifies data-dependencies among the annotated tasks and reconstructs potential parallel time-behavior. If the programmer is satisfied with the obtained parallelization, Tareador can further assist the process of porting the application by identifying input and output parameters of each task. Tareador already proved to be so useful in exploring parallelism, that it was included in the academic program of UPC.

1.4 Document structure

The document is organized as follows. Chapter 2 presents the state-of-the-art technology related to the topic of this thesis. It provides a survey on the hardware architecture, programming models and development tools related to this thesis. Chapter 3 illustrates the performance issues in parallel computing and introduces the techniques for parallelization tuning addressed in this thesis. In Chapter 4, we describe the development environments designed during the work on this thesis. Furthermore, Chapter 5 presents our work in the field of tuning MPI parallelism. More specifically, it describes our work in the field of overlapping communication and computation in MPI applications. Furthermore, Chapter 6 presents our work in the field of tuning MPI/OmpSs parallelism. Namely, it describes our techniques for identifying parallelization bottlenecks in MPI/OmpSs and exploring potential MPI/OmpSs parallelism inherent in applications. Chapter 7 presents the previous work related to the research covered in this thesis while Chapter 8 draws the conclusion of this thesis and presents the direction of the future research in this field. Finally, Chapter 9 lists the papers that we published throughout the work on this thesis.

2 Background

This Chapter presents the background and state-of-the-art relevant to this thesis Section 2.1 presents background on the architecture of parallel machines. It focuses on the classification of parallel machines on shared-memory and distributed-memory, especially analyzing the cost of communication among separate processing units. Section 2.2 describes the mainstream parallel programming models. It revisits the most widely used programming models for distributed-memory parallel machines (MPI) and shared-memory parallel machines (OpenMP). Also, we describe the OmpSs programming model that extends OpenMP providing semantics for expressing dataflow parallelism. Finally, in Section 2.3 we present the legacy tools that are used throughout this thesis. These tools are the starting point of the thesis – our initial extensions finally graduated into fully independent tools.

2.1 Parallel machines

The constant need for higher performance caused the appearance of machines with parallel architecture. A conventional sequential computer consists of the processor connected to the memory via a datapath. Intensive computation makes each of these three parts a bottleneck. As the direct acceleration on these parts showed unfeasible, the solution was found in multiplying resources. However, in order to take advantage of this multiplicity, the execution must exhibit parallelism. The parallelism can be implicit (hidden from the programmer) or explicit (directly expressed by the programmer). In the rest of this Section, we present architectural concepts related to parallel processing.

2.1.1 Processor architecture trends

The initial trend in manufacturing more powerful computers was building processors with higher clock frequency. Processor chips are the key components of computers. The most straight-forward approach for making faster computers is making a computer that operates on a higher clock frequency. For a long time, the vendors provided computers with increased frequency, making that property the selling point for all processor chips. Frequency scaling provides an automatic (free) speedup for any software – due to frequency scaling, an unchanged software automatically achieves the speedup proportional to the frequency improvement (gray region in Figure 2.1). However, increased frequency dramatically increases chip's power consumption. Therefore, computing platforms have hit the so called "power wall"[68], with frequency scaling no longer appealing over the 3 GHz.

As the trend of increasing clock frequency became unsustainable, the vendors started investing more effort in designing architectural improvements on the chip. In 1965, Gordon Moore made an empirical observation that the number of transistors on a processor chip doubles every 18-24 months. This observation, called Moore's law [77], still holds. The increased number of transistors in the chip is used for architectural improvements, such as additional functional units, additional registers, wider paths ... This resource abundance increases chip's floating point peak performance (light-red region in Figure 2.1). However, unlike frequency scaling, the resource abundance pro-

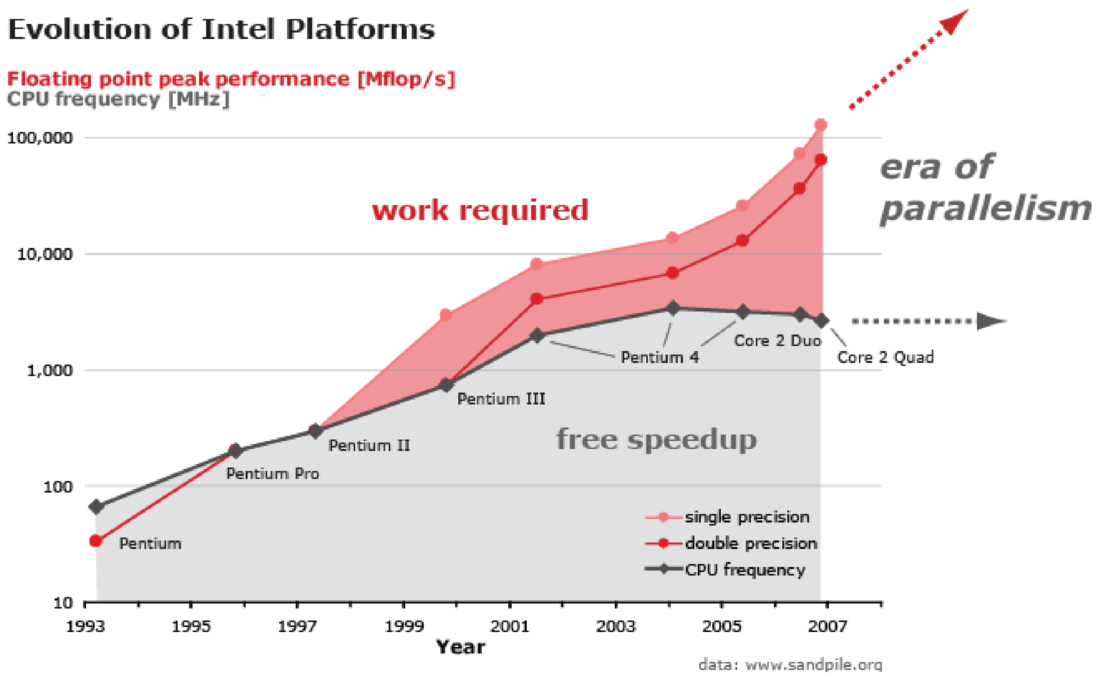


Figure 2.1: The evolution of computing platforms. Chart originally from SPIRAL project website [72]

vides no automatic speedup of execution. Therefore, in order to achieve execution speedup, either hardware or software must include additional logic that leverages the additional resources.

Increased number of transistors in the cores lead to architectural improvements that increase the internal processors use of parallelism. This type of parallelism is extracted in the hardware and entirely hidden from the programmer. There are two types of internal processor's parallelism:

1. **Pipelining [47]:** The hardware breaks each instruction into pipeline stages, so different stages of different instructions can execute concurrently. Typically, an instruction is broken into stages of fetch, decode, execute and write-back. The hardware itself checks for data-dependencies among different pipelines, and allows the independent pipeline stages to executed concurrently. This type of parallelism is called instruction-level parallelism (ILP). Available degree of parallelism theoretically increases with the number of pipeline stages. Current processes have between 2 and 26 pipeline stages.

2. **Superscalar execution [47]:** The hardware consists of multiple independent functional units. The processor is multi-issue – it issues multiple instructions at the same time in order to utilize multiple ALUs, FPUs, load/store units ... These multiplied resources can be utilized in parallel execution of different instructions, as long as data-dependencies are satisfied. Depending on how are the data-dependencies among instructions calculated, these processors can be classified into superscalar processors and very long instruction word (VLIW) processors. In superscalar processors, the hardware dynamically detects data-dependencies, which significantly increases the architectural complexity of the chip. On the other hand, in VLIW processors, the compiler resolves dependencies, generating long instructions that explicitly specify which operations can execute concurrently.

However, since the techniques of implicit parallelism showed only limited potential, new computer architectures targeted higher performance by allowing the programmer to explicitly expose parallelism. The presented two techniques exposed parallelism without any involvement of the programmer – the user programmed a sequential control flow but the underlying system automatically extracted parallelism. However, the implicit parallelism showed to be insufficient. An alternative approach puts multiple independent cores in one chip and allows the programmer to have a different control flow in each of the cores. These parallel architectures provided more flexibility in using the computation resources, but also increased the complexity of programming.

2.1.2 Memory Organization

Parallel computers can have shared or distributed-memory organization. Here, a clear distinction should be made between how the memory is physically organized in hardware and how the memory is perceived by the programmer. In respect to the physical organization of the memory, a parallel machine can have shared memory or distributed memory. Moreover, the machine can also have a hybrid organization where at the level of one node the machine has shared memory, while different nodes among themselves operate on distributed memory. However, all these physical systems, from the programmers point of view can be systems with shared or distributed address space. Further text in this Section describes the differences between shared-memory and distributed-memory systems.

Ditributed-Memory Organization

Distributed-memory machines (DMM) are computers with physically distributed memory. Each node is an independent unit that consists of a processor and a local memory. An interconnection network connects all the nodes and allows communication among them. Each node can only access its local memory. If a node needs data that is not in its local memory, the data needs to be transferred by sending messages through the network.

Distributed-memory machines improve efficiency by investing in faster and more intelligent networks. In DMMs, the nodes are usually connected by point-to-point interconnection links. Each node connects to a finite number of neighboring nodes. The network topology is regular, often a hypercube or a tree. Since each node can send the message only to its neighboring nodes, limited connectivity significantly restricts programming. Initially, communication between nodes that have no direct connection had to be controlled by software of the intermediate nodes. However, new intelligent network adapters enabled data transfers to or from the local memory without participation of the host processor. This allowed that the host processor can be efficiently computing while, in background, there is a transfer to/from it. Furthermore, the state-of-the-art networks optimize communications by dedicating special links for executing multicast transfers.

A distributed-memory machine consists of loosely coupled processing units, making it easy to assemble but difficult to program. DMMs can be assembled using off-the-shelf desktop computers. However, to achieve high performance, the nodes must be interconnected using a fast network. On the other hand, DMMs are very difficult to program. The programmer must explicitly specify the data decomposition of the problem across processing units with separate address spaces. Also, the programmer must explicitly organize inter-processor communication, making both the sender and the receiver aware of the transfer. Moreover, the programmer must take special care about data partitioning among the nodes, because delivering some data from one node to another may be very expensive. Thus, the data layout must be selected carefully to minimize the amount of the exchanged data.

Shared-Memory Organization

Shared-memory machines (SMM) are computers with a physically shared memory. A SMM consists of more processors, a shared physical memory (global memory), and the network that connects the processors and the memory. The processors communicate by reading and writing shared variables. The global memory usually provides a common address space over a set of memory modules. The programming model allows coordination of processors through the accesses to the common address space. However, concurrent accesses to the shared data must be coordinated in order to avoid race conditions with unpredictable effects.

The users perception of shared memory facilitates programming, at the cost of difficult implementation of the machine. Communication through shared variables allows easy parallelization. The programmer is less concerned about data locality, expecting from the machine to serve all his requests for data. However, providing a fast global memory access to multiple processors makes the technical realization of a SMM a serious effort. Increasing the number of processors in the system further stresses performance of the global memory. Thus, a shared resource of global memory is an impediment for high scalability of these machines. Scaling of these machines beyond tens of cores is very hard.

The most common parallel programming model for SMMs is threaded execution. Multiple threads have separate control flows but can access shared global memory. A programmer expresses parallelism using parallel constructs offered by the programming model. The programming model maps user threads on system kernel threads, while the operating system maps kernel threads to processors. These mapping algorithms are partly or entirely hidden from the programmer. Also, the operating system can take advantage of hardware parallelism by starting concurrent execution of different sequential programs on different processors.

Initial implementations of SMMs usually relied on uniform memory access (UMA) architectures. UMA architecture provides a uniform access time from any processor to the single shared memory. Thus, UMA platforms are often called symmetric multiprocessors (SMP). The interconnect is typically a central bus that connects small number of processors to the global memory. In this architecture, the interconnect becomes a bottleneck that limits the number of processors in the system. Besides caches, there is no other memory private to the processors. Caches allow faster access time, and avoid that all the data requests go to the interconnect.

Today many implementations of SMMs rely on NUMA (non-uniform memory access) architecture. NUMA architectures offer to the programmer perception of a shared address space, although the underlying architecture is based on physically distributed memory. Thus, NUMA platforms are often called distributed shared-memory (DSM) machines. Perception of shared address space is achieved using the cache coherence protocol. The coherence protocol guarantees that the memory access goes to the last version of the variable, independent of where the variable is physically stored. However, coherence causes the access time to memory to depend on the physical location of the accessed data. An access to the data that is locally stored is faster than an access to the data that is stored in the local memory of some other node.

2.1.3 Message Passing Cost

This thesis focuses on achieving more efficient inter-processor communication. The efficiency of communication can significantly determine the overall performance of the parallel system. In this subsection, we explore in more details the factors that determine the cost of message passing among nodes. Also, we explore the possible techniques to reduce the transfer time for messages.

The message transfer time is a sum of the time needed to prepare the message for transmitting, and the time needed for the message to traverse the network to its destination. The system parameters that determine the transfer time are:

- **Startup time:** Startup time is the processing time local to the node that is needed before sending and after reception of the message. This time accounts for the time needed to pack the data into the message (adding message header, tail and error correction information). Also, this time accounts for executing routing algorithms and interfacing of the processor with the router.

- **Latency:** On its way to the destination, the message has to do a finite number of hops via links that directly connect routers. Latency time is the multiply of the latency of the router and the length of the routing path (number of hops to reach the destination).

- **Bandwidth:** Bandwidth determines the speed by which the data traverses the network. Bandwidth of the network depends of the clock rate and the width of the links as well as of the speed of the routing and buffering within the routers.

One way of improving message passing among nodes is to reduce the transfer time. Network vendors constantly deliver more expensive networks that have more resources. First, networks with faster clock rate can provide higher bandwidth and lower latency. Latency can further be improved by the increased connectivity among nodes. Furthermore, new network interfaces can offload message processing from the node, thus reducing the startup time. Also, message transfer time can be significantly reduced by the programmer. Organizing a good data layout, the programmers can reduce the total amount of data that needs to be communicated. Moreover, the programmer can amortize the startup overhead by communicating in bulk – accumulating more data and communicating in larger messages.

However, this thesis focuses on different techniques that improve message passing – instead of reducing time of the transfer, our goal is to hide the transfer time by overlapping it with useful computation. As already mentioned, state-of-the-art parallel machines can transfer messages without direct involvement of the processors. Thus, while the message is in transfer, the processor can be busy doing some useful computation. This approach potentially allows to execute long transfers, without consequently stalling the involved processors.

2.2 Parallel programming models

The search for the "holy grail" of parallel computing has failed – there is no efficient and highly applicable automatic parallelization. The biggest idea in the field of parallel computation was to find techniques that would automatically expose parallelism in the applications. However, despite decades of work [78], automatic parallelization showed very limited potential. Today, the only viable solution is to rely on the programmer to expose parallelism.

For distributed-memory machines, the mainstream programming models are message passing (MP) and partitioned global address space (PGAS). The most popular implementation of MP model is message passing interface (MPI) [81]. In MPI, the programmer must partition the workload among processes with separate address spaces. Also, the programmer must explicitly define how the processes communicate and synchronize in order to solve the problem. Conversely, PGAS model is implemented in languages such as UPC [20], X10 [25] and Chapel [24]. PGAS model provides a global view for expressing both data structures and the control flow. Thus, as opposed to message passing, the programmer writes the code as if a single process is running across many processors.

On the other hand, OpenMP [31] is a de-facto standard for programming shared-memory machines. OpenMP extends the sequential programming model with a set of directives to express shared-memory parallelism. These directive allow exposing fork-join parallelism, often targeting independent loop iterations. Nevertheless, OmpSs programming model [37] extends OpenMP offering semantics to express dataflow parallelism. Compared to fork-join parallelism exposed by OpenMP, the parallelism of OmpSs can be much more irregular. Throughout this thesis, we focus mainly on OmpSs as a programming model for shared-memory machines. We also explore the potential of MPI/OmpSs, a hybrid programming model that integrates OmpSs and MPI.

2.2.1 MPI

Message Passing Interface (MPI) [81] is the most widely used programming model for programming distributed parallel machines. To facilitate writing message-passing programs, MPI standard defines the syntax and semantics of useful library routines. Today, MPI is the dominant programming model in high-performance computing.

In the MPI programming model, multiple MPI processes compute in parallel and communicate by calling MPI library routines. At the initialization of the program, a fixed set of processes is created. Typically, the optimal performance is achieved when each MPI process is mapped on a separate core. For easier coordination among processes, MPI interface provides functionality for communication, synchronization and virtual topology.

The most essential functionality of MPI is point-to-point communication. The most popular library routines are: MPI_Send to send a message to some specified process; and MPI_Recv to receive a message from some specified process. Point-to-point operations are especially useful for implementing irregular communication patterns. A point-to-point operation can be in synchronous, asynchronous, buffered, and ready form, providing stronger and weaker synchronization among communicating processes. The ability to probe for messages allows MPI to support asynchronous communication. In asynchronous mode, the programmer can issue many outstanding MPI operations.

Collective operations allow communication of all processes in a process group. The process group may consist of the entire process pool, or it may be user defined subset of the entire pool. A typical operation is MPI_Bcast (broadcast), in which the specified root process sends the same message to all the processes in the specified group. A reverse operation is MPI_Gather, in which the specified root process receives one message from all the processes in the specified group. Other collective operations implement more sophisticated communication patterns.

Throughout its evolution, MPI standard has introduced new features to facilitate easier and more efficient parallel programming. The initial MPI-1 specification focused on message passing within a static runtime environment. Additionally, MPI-2 includes new features such as parallel I/O, dynamic process management, one-sided communication, etc. Furthermore, recent research studies [50] motivated MPI community to include non-blocking collective operations [50] as a part of MPI-3 standard.

A simple program

In this Section, we present a simple program and explain the runtime properties of MPI execution. The example shows a simple code with only two sections of useful work (function $compute$) and one Section that exchanges data (function $MPI\_Sendrecv$). Each MPI process executes function $compute$ on local buffer $buff1$, then sends the calculated buffer $buff1$ to its neighbor. At the same time, each process receives buffer $buff2$ from another neighbor, and again executes function $compute$ on the received buffer. The processes communicate in a one-sided ring pattern – each process receives the buffer from the process with rank for 1 lower, and sends the calculated buffer to the process with rank for 1 higher.

All the processes start independently, learning about the parallel environment by calling MPI_Init. MPI execution starts by calling the MPI agent ($mpirun{-}nx./binary.exe$)

```c
#include <mpi.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
   float buff1[BUFSIZE], buff2[BUFSIZE];
   int numprocs;
   int myid;
   int tag = 1;

   MPI_Status stat;
   MPI_Init(&argc,&argv);

   /* find out how big the SPMD world is */
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);

   /* and this processes' rank is */
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   /* At this point, all programs are running equivalently,
   the rank distinguishes the roles of the programs in the SPMD model */

   /* compute on the local data (buff1) */
   compute(buff1);

   /* exchange data (send buff1 and receive buff2 )*/
   my_dest = (myid + 1) % numprocs;
   my_src = (myid + numprocs - 1) % numprocs;
   MPI_Sendrecv(   /* sending buffer */ buff1, BUFSIZE, MPI_FLOAT,
                   /* destination MPI process */ my_dest, tag,
                   /* receiving buffer */ buff2, BUFSIZE, MPI_FLOAT,
                   /* source MPI process */ my_src, tag,
                   MPI_COMM_WORLD, &stat);

   /* compute on the received data (buff2) */
   compute(buff2);

   /* MPI programs end with MPI Finalize; this is a weak synchronization point */
   MPI_Finalize();

   return 0;
}

```

Figure 2.2: Example MPI code

that spawns the specified number (x) of MPI processes. In the studied case (Figure 2.3), the MPI execution starts with 2 independent MPI processes. By calling *MPI_Init*, each process learns about the MPI parallel environment. All the MPI processes are grouped into the universal communicator (*MPI_COMM_WORLD*). Using the universal com-

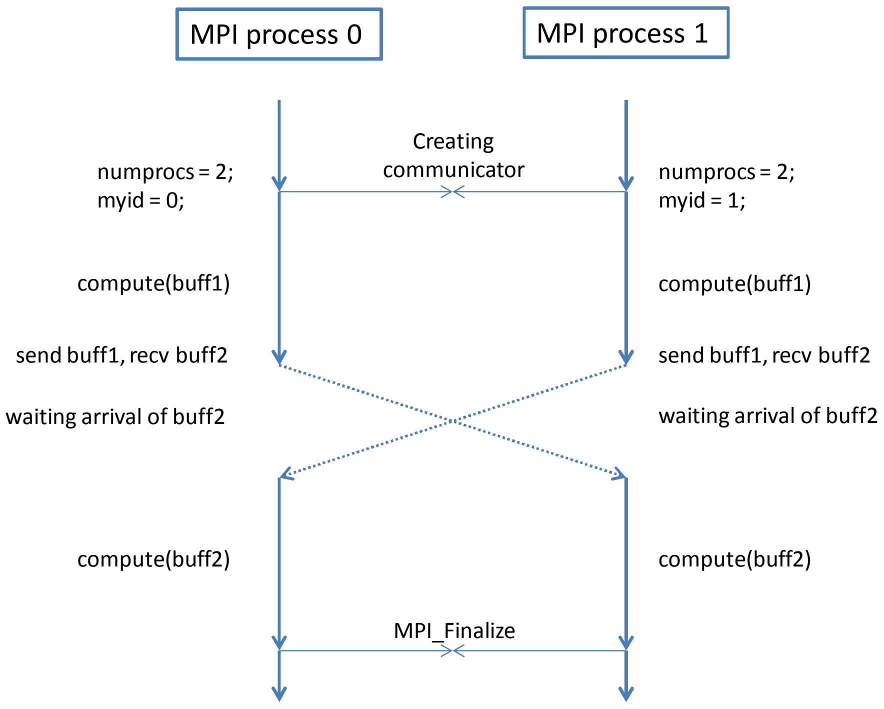


Figure 2.3: Execution of the example MPI code

municator, each process identifies the total number of MPI processes in the system (*MPI_comm_size*) and gets the unique rank of the process (*MPI_comm_rank*).

Each MPI process, knowing its rank and the size of the universal communicator, identifies its role in the execution of the parallel program. Each process identifies the part of the total workload that is assigned to it. Also, each process identifies the ranks of the neighboring processes with which it should communicate in order to get the job done. In the presented example, based on *myid* and *numprocs*, each process calculates ranks of its neighboring processes (*my_dest* and *my_src*) to generate the one-sided ring communication pattern.

When the computation on the local data finishes, the processes communicate to exchange the data and start the next phase of computing on the local data. Each process calculates the buffer $buff1$ in the function *compute*. Then, the process sends the processed $buff1$ to the neighboring process. At the same time, the process receives a message from some other neighboring process and stores the content of that message into local buffer $buff2$. Then, the process locally computes on $buff2$ in another instantiation of function *compute*. Thus, MPI process 0 calculated on its local $buff1$, and then after the $MPI\_Sendrecv$ call, it calculated on its local $buff2$ (that was $buff1$ local to MPI process 1).

When the useful work finishes, all the processes call $MPI\_Finalize$ to announce the end of the parallel section. $MPI\_Finalize$ implicitly calls a barrier, waiting for all the MPI processes from MPI_COMM_WORLD to come to this point. When all the processes reach the barrier, the joint work is guaranteed to be finished, and all the processes can exit the parallel execution independently. The parallel execution finishes.

This simple example also illustrates one of the main topics of this thesis – communication delays that are characteristic for MPI execution. When both processes finish executing $compute(buff1)$, they initiate their transfers in the same moment. While the messages are in transit, both processes are stalled without doing any useful work. Chapter 3 further illustrates this problem and presents some of the possible solutions.

Different MPI implementations

MPI standard defines a high-level user interface, while low-level protocols may vary significantly depending of the implementation. MPI provides a simple-to-use portable interface for a basic user, setting a standard for hardware vendors of what they need to provide. This opens space for various MPI implementations that have different features and performances.

Depending on the implementation, MPI messaging may use different messaging protocols. An MPI message passing protocol describes the internal methods and policies employed to accomplish message delivery. Two common protocols are

- **eager** – an asynchronous protocol in which a send operation can execute without an acknowledgement from the matching receive; and
- **rendezvous** – a synchronous protocol in which a send operation can execute only upon the acknowledgement from the matching receive.

Eager protocol is faster, as it requires no "handshaking" synchronization. However, this relaxation of synchronization comes at the cost of the increased memory usage for message buffering. Thus, a common implementation uses eager protocol only for messages that are shorter than the specified threshold value. On the other hand, messages larger than the threshold are transferred using rendezvous protocol. Also, it is common that a very long message is partitioned into chunks, with each chunk being transferred using separate rendezvous protocol.

Also, MPI implementations provide different interpretation of independent progress of transfers. Independent progress defines whether the network interface is responsible for assuring progress on communications, independent of making MPI library calls. This feature is especially important for the messages that use rendezvous protocol. For example, rank 0 sends a non-blocking transfer to rank 1 using rendezvous protocol. If rank 0 comes to its MPI_Isend before rank 1 comes to the matching receive, rank 0 issues handshaking request and leaves the non-blocking send routine. Later, when rank 1 enters its corresponding MPI_Recv, it acknowledges the handshake, allowing rank 0 to send the message. The strict interpretation of the independent progress mandates that rank 0 sends the actual message as soon as it receives the acknowledgement from rank 1. Conversely, the weak interpretation mandates that rank 0 must enter some MPI routine in order to process the acknowledgement and prepare for the actual message transfer. Here, the weak interpretation of independent progress will be very performance degrading if after the non-blocking send, rank 0 enters a very long computation with no MPI routine calls. Most of the state-of-the-art networks provide the strict implementation of progress by introducing interrupt-driven functionality in the network interface.

Depending on the computation power of the network interface, an MPI implementation can provide different ability for communication/computation overlap. MPI standard specifies semantic for asynchronous communication that offers significant performance opportunities. However, in some machines, the processors are entirely responsible for assuring that the message reaches its destination. On the other hand, many state-of-the-art networks provide intelligent network adapters that take care of delivering the message, allowing the processor to dedicate to useful computation. This way it is possible to achieve overlap of communication and computation – a feature that is considered of a major importance for high parallel performance.

Also, an MPI implementation can provide different interpretation for offload. Offload is the feature that enables the processor to pass the overhead of MPI routines to the network interface. Offload avoids the host processor involvement in progressing on a non-blocking send on the sender side and storing the received message into the local memory of the receiver side. An implementation that supports offload allows both overlap and independent progress to be achieved without host processor overhead. In applications with frequent calls to MPI library, offload can significantly boost the performance [19].

2.2.2 OpenMP

OpenMP (Open Multi-Processing) [31] is the mainstream programming model for programming shared-memory parallel systems. OpenMP is an application programming interface (API) that supports multi-platform shared-memory programming in C, C++, and Fortran. It uses a portable model that provides to programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer. OpenMP allows integration with MPI, providing a hybrid MPI/OpenMP model for parallel programming at large scale.

OpenMP is a programming model based on fork-join parallelism. On reaching a parallel section, a master thread forks a specified number of slave threads. All the threads run concurrently, with the runtime environment mapping threads to different processors. When the parallel section finishes, the slave threads join back into the master. Finally, the master continues through the sequential section of the program.

OpenMP provides a set of programming language extensions that the programmer can use to expose parallelism in the program. OpenMP extends the sequential programming model with a set of directives (pragmas in C/C++) to express shared-memory parallelism. Also, it provides a set of runtime library routines and environment variables that allow the programmer to dynamically modify parallel execution. The OpenMP language extensions can be classified into: 1) control structures for expressing parallelism; 2) data environment constructs for communicating among threads; and 3) synchronization constructs for coordinating threads. In the following paragraphs, we further explain these three types of extension.

Control structures alter the flow of execution. There are two kinds of constructs for expressing parallelism. The *parallel* directive encapsulates a block of code and creates a set of threads that concurrently execute that block. The multiple concurrent threads execute different execution instances of that block of code. The second control structure is the *do* directive that allows to divide the work among an existing set of threads.

Data environment constructs enable communication among threads. When the master thread reaches a parallel construct, it creates new slave threads with private execution contexts. This enables a new thread to execute without interfering with stack frames of other threads. Data environment constructs allow the programmer to choose whether some variable will be shared among threads. Each variable has one of the three sharing attributes:

- *shared*: A variable with shared scope clause has a single storage location in memory during the parallel construct. Using this unique memory location, the threads can communicate through read/write operations.

- *private*: A variable with private scope clause has a separate storage location for each thread during the parallel construct. All read/write operations on this location are protected from the accesses of other threads.

- *reduction*: The reduction clause is used for variables that are target of reduction operations. This sharing attribute mixes the properties of shared and private attribute, allowing the compiler to optimize accesses to it.

Synchronization constructs coordinate the execution of threads. OpenMP threads communicate via read/write operations to the shared variables. However, without any thread synchronization, concurrent accesses to shared variables may induce race conditions. Two common OpenMP synchronization mechanisms are mutual exclusion and event synchronization. Mutual exclusion guarantees that a shared variable is accessed exclusively by one thread. Exclusivity is achieved with the *critical* directive that encapsulates a block of code that can be accesses by one thread at the time. Event synchronization signals the occurrence of some event across multiple threads. The most common event synchronization construct is the *barrier* directive that defines a point where each thread waits for all the other threads to catch up. Besides *critical* and *barrier*, OpenMP provides other directives that can model more complicated synchronization patterns.

A simple OpenMP example

This Section analyzes a simple loop parallelization using OpenMP. Figure 2.4 shows a simple OpenMP code. The code consists of one loop that calls function *compute* in each iteration. Before the loop, the buffer is initialized (function *initialize*), and after the loop, the buffer is validated (function *validate*). In each iteration of the loop, function *compute* executes on a different element of array *a*, making all the iterations of the loop independent. Thus, the appropriate OpenMP parallelization of this code would be using the *parallel* construct to make all the iterations of the loop execute concurrently. The original sequential code is parallelized by adding only one line of code.

Figure 2.5 illustrates OpenMP parallel execution of the presented code. The program initializes with only one thread active – the main thread. After finishing function

```
int main(int argc, char *argv[]) {
   const int N = 100000;
   int i, a[N];

   initialize(a);

   #pragma omp parallel
   for (i = 0; i < N; i++)
      compute(a[i]);

   validate(a);

   return 0;
}
```

Figure 2.4: simple OpenMP code

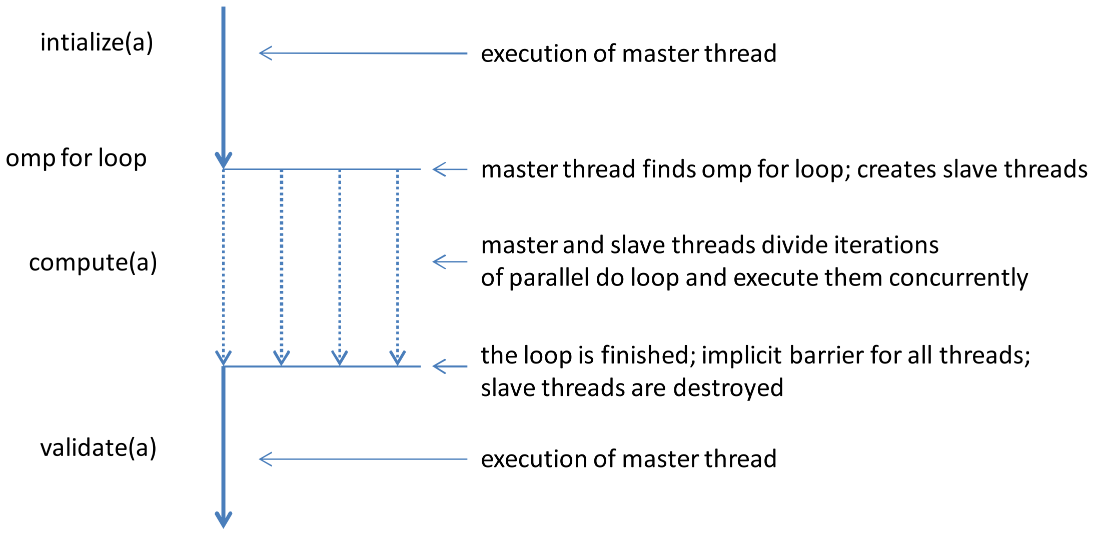

intialize(a)
execution of master thread
omp for loop
master thread finds omp for loop; creates slave threads
compute(a)
master and slave threads divide iterations
of parallel do loop and execute them concurrently
the loop is finished; implicit barrier for all threads;
slave threads are destroyed
validate(a)
execution of master thread

Figure 2.5: simple OpenMP code

initialize, the main thread reaches the parallelized loop and spawns the specified number of slave threads (3 in this case). The four threads partition among themselves the iterations of the loop and start computing concurrently. The parallel for loop ends with an implicit barrier that waits for each thread to finish executing iterations that are assigned to it. When all the threads finish their portion of work, the barrier condition is fulfilled. The main thread destroys all the slave threads and proceeds with the sequential section of the program (function *validate*).

It is important to note that an OpenMP programmer just specifies how to divide the work between threads, but not how to execute that work on the underlying machine. For the programmer, the thread is a separate control flow that operates on the same memory as the main thread. Having that abstraction, the programmer does not care whether OpenMP thread is implemented as an OS thread or a Pthread. Also, the programmer leaves to the compiler (or runtime) to decide how to partition the iterations across the threads. However, to achieve higher data locality, OpenMP provides to the programmer special semantics to specify how the iterations should be distributed across threads.

In the presented simple code, the scope of each variable is determined by the default OpenMP rules. By default, all the variables are shared if not declared differently. Thus, array *a* is shared. Each iteration of the parallelized loop accessed a different element in the array, avoiding any conflict of concurrent accesses to a shared variable. Especially, loop index *i* is an exception in the default OpenMP rules. Within a parallel construct, loop index is automatically declared as private variable. Thus, the conflicts are avoided by making a private copy of the loop index for each thread. After the parallel loop finishes, these private copies are destroyed, and the value of *i* is assumed to be undefined.

Again, our simple example requires no explicit synchronization. Synchronization is primarily used to coordinate accesses to shared variables. Since each iteration of the loop computes on a different element of array *a*, there is no need for explicit synchronization inside parallel loop. However, in order to do the validation of the array (function *validate*), all the threads must finish their computation of loop iterations. Again, explicit synchronization constructs are avoided, because parallel loop implicitly ends with a barrier. The barrier assures that all the concurrent threads finished their work, so the main thread can proceed to *validate*.

```
p = listhead;
while (p) {
   process (p)
   p=next (p);
}
```

(a) sequential code.

```
p = listhead;
num_elements = 0;
while(p) {
   listitem[num_elements++] = p;
   p = next(p);
}
#pragma omp parallel for
for (int i=0; i<num_elements; i++)
   process( listitem[i] );
```

(b) OpenMP without tasks.

```
#pragma omp parallel
{
   #pragma omp single
   {
      p = listhead;
      while (p) {
         #pragma omp task
            process(p);
         p = next(p);
      }
   }
}
```

(c) OpenMP with tasks.

Figure 2.6: Pointer chasing application parallelized with OpenMP

OpenMP tasks

Although the parallelization of the presented code appears easy and elegant, it remains a question whether OpenMP loop parallelization could be widely applicable. OpenMP is tailored for applications with array-based computation. These application have very regular parallelism and regular control structures. Thus, OpenMP can identify all work units in the compile time and statically assign them to multiple threads. However, irregular parallelism is inherent in many applications such as tree data structure traversal, adaptive mesh refinement and dense linear algebra. These applications would be very hard to parallelize using only basic OpenMP syntax.

Let us consider possible OpenMP parallelization of the sequential code from Figure 2.6a. The program consists of a while loop that updates each element of the list. The code cannot be parallelized just by adding a parallel loop construct, because the list traversal would be incorrect. Thus, in order to use a parallel loop construct, the list first has to be translated into an array (Figure 2.6b). However, this translation causes an inadmissible overhead.

In order to tackle this issue, OpenMP introduces support for tasks. Tasks are code segments that may be deferred to a later time. Compared to the already introduced work units, tasks are much more independent from the execution threads. First, a task is not bound to a specific thread – it can be dynamically scheduled on any of the active threads. Also, a task has its own data environment, instead of inheriting the data environment from the thread. Moreover, tasks may be synchronized among themselves, rather than synchronizing only separate threads. This implementation of OpenMP tasks allows much higher expressibility of irregular parallelism.

Figure 2.6c illustrates the possible parallelization of the studied code using OpenMP tasks. The master thread runs and dynamically spawns a task for each instantiation of the function *process*. On each instantiation, the content of pointer *p* is copied into the separate data environment of the task. Since the inputs to the task are saved, the task can execute later in time. Thus, the master thread sequentially traverses the list and dynamically spawns tasks. The spawned tasks are executed by the pool of worker threads. When the main thread finishes spawning all tasks, it joins the workers pool. Therefore, despite of irregular control structures, OpenMP tasks allow elegant parallelization.

Since tasks showed to be very powerful in exposing irregular parallelism, OpenMP community concentrates on developing new tasking constructs that could further facilitate programming. Thus, soon to appear OpenMP API version 4.0 introduces data-dependencies between tasks. Namely, the 4.0 standard introduces a new task clause:

depend(dependence-type : list),

where *dependency-type* can be *in*, *out* or *inout*, while *list* is one or more storage locations. The depend clause allows specifying additional constraints on the scheduling of tasks. Hence, a task may be dependent on a previous sibling task (task spawned from the same parent task), if the *depend* clauses of these tasks result in a RAW, WAW or WAR dependency of some storage location specified in the *list* (the specified storage locations must be either identical or disjoint). Furthermore, the OpenMP 4.0 API offers *taskyield* clause that denotes the execution point at which the current task can be preempted in favor of execution of some other task.

2.2.3 OmpSs

OmpSs [37] is a parallel programming model based on dataflow execution. OmpSs is an effort to extend OpenMP with new directives to support dataflow parallelism. Compared to fork-join parallelism exposed by OpenMP, the parallelism of OmpSs can be much more irregular and distant. Similar to the integration of MPI and OpenMP, OmpSs can integrate with MPI in the MPI/OmpSs hybrid parallel programming model.

OmpSs slightly extends C, C++ and Fortran, providing semantics to express task-based dataflow parallelism. There are two essential annotations needed to port a sequential application to OmpSs (Figure 2.7):

- *task decomposition* – to mark with pragma statements, which functions should be executed as task; and
- *directionality of parameters* – to mark inside pragmas, how are the passed arguments used within the taskified function. The specified directionality can be *input*, *output* and *inout*.

```
#pragma omp task input(A) output(B)
void compute(float *A, float *B) {
   ...
}


int main () {

...

compute(a,b);

...

}
```

Figure 2.7: Porting sequential C to OmpSs

Given the OmpSs annotations, the runtime can schedule all tasks out-of-order, as long as the data dependencies are satisfied. The main thread starts, and, when it reaches a taskified function, it instantiates that function as a task and proceeds. Based on the parameters' directionality, the runtime places the obtained task instance in the dependency graph of all tasks. Then, considering the dependency graph, the runtime is free to dynamically schedule the execution of tasks on multiple worker threads. Also, whenever the main thread reaches a blocking condition (e.g. a barrier), it helps the worker threads by executing tasks.

To increase parallelism, the runtime automatically renames data objects to avoid false dependencies (dependencies caused by buffer reuse). The renaming technique is similar to the ones introduced in superscalar processors [80] or optimizing compilers [54]. Renaming avoids false dependencies by eliminating write-after-read and write-after-write dependencies. Other programming models avoid these dependencies by forcing the programmer to explicitly require per-thread copies of the variables. Conversely, OmpSs resolves this problem using automatic renaming. Whenever an array that is passed to a task has *output* directionality, the runtime automatically allocates a new array and operates on it. The runtime also allows configuring the amount of memory used for double buffering, to avoid the hazard of intensive swapping.

The information passed in pragmas allows efficient scheduling of tasks. The programmer can add the *highpriority* clause to some task, indicating that the task has higher scheduling priority. These tasks are scheduled as soon as possible. On the other hand, while scheduling tasks of regular priority, the runtime strives to exploit data locality. The scheduler favors running a task in the thread that just generated one of the input parameters of the task. Also, the scheduler tries to isolate threads in the data-dependency graph of tasks, thus reducing the possibility of two different threads accessing the same data. Furthermore, in the cases of imbalanced execution, threads can steal work from one another. In the case of work stealing, the thread chooses a task instance that spent most time in the waiting queue, thus increasing the probability that the input data for that instance is already evicted from the cache of the thread it is stealing from.

Also, OmpSs allows simple semantics to be used for programming heterogeneous architectures. By adding only one directive to the pragma construct, OmpSs can declare that instances of some task are to be executed on hardware accelerators. Reading these annotations, the runtime schedules the execution of the specified task on the dedicated hardware and automatically moves all the needed data for that task. This feature significantly facilitates the easy programming of heterogeneous architectures, as it was proven for programming Cell B./E. [11] and Nvidia GPUs [9].

Dataflow parallelism introduced in OmpSs showed to be powerful in extracting parallelism, and it is finding its place in the standards of the mainstream parallel programming models. Perez *at. el.* [69] showed that in many applications, OmpSs significantly outperforms fork-join based programming models such as OpenMP [31] and Cilk [42]. As already mentioned in Section 2.2.2, task-based dataflow parallelism is introduced in the OpenMP 4.0 standard.

Example of irregular parallelism – Cholesky

Figure 2.8 shows OmpSs parallelization of the sequential Cholesky code. In order to parallelize Cholesky code with OmpSs, only four code lines need to be added. All four functions called from *compute* are encapsulated into tasks using *#pragma omp task* directives. For each of these functions, pragma directives also specify the directionality of function parameters. This type of coordinating tasks on the shared variables is much easier for the programmer than determining what variables should be shared or private among the threads. After adding the annotations, the resulting code has the same logical structure as the original sequential code. Also, note that compiling this OmpSs code with a non-OmpSs compiler simply ignores OmpSs pragmas and creates a binary for the corresponding sequential execution.

In parallel execution of this code, the annotated tasks can execute out-of-order, as long as data-dependencies are satisfied. The program initiates with only one active thread – the master thread. When the master thread reaches a taskified function, it instantiates that function into a task and wires in the new task instance into the tasks dependency graph. Considering the dependency graph, the runtime schedules out-of-order execution of tasks.

Compared to OpenMP, OmpSs potentially exposes more distant and irregular parallelism. For example, some of the instances of task *sgemm_tile* are mutually independent, while some are data-dependent (Figure 2.9). This type of irregular concurrency would be very hard to express with OpenMP. However, OmpSs runtime dynamically exposes the potential parallelism, keeping the programmer unaware of the actual dependencies among tasks. Also, in parallelizing instances of *sgemm_tile*, OpenMP would introduce implicit barrier at the end of the loop. On the other hand, OmpSs omits this barrier, allowing instances of *sgemm_time* to execute concurrently with some instances of tasks *ssyrk_tile* (Figure 2.9). Again, the programmer alone could hardly identify and expose this potential concurrency.

Interoperability with MPI

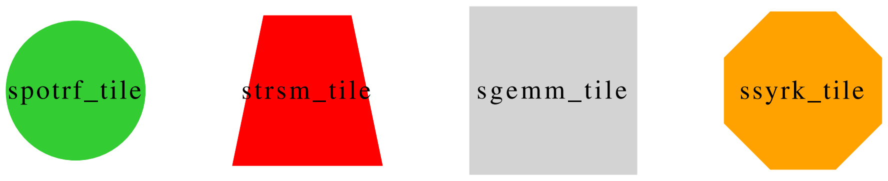
OmpSs integrates with MPI in a manner similar to integration of MPI and OpenMP. The result is a hybrid MPI/OmpSs [63] programming model, in which the work is parallelized across separate address spaces using MPI, while the work of each MPI

```
#pragma omp task input(NB) inout(A)
void spotrf_tile(float *A,unsigned long NB);

#pragma omp task input(A, B, NB) inout(C)
void sgemm_tile(float *A, float *B, float *C, unsigned long NB);

#pragma omp task input(T, NB) inout(B)
void strsm_tile(float *T, float *B, unsigned long NB)

#pragma omp task input(A, NB) inout(C)
void ssyrk_tile( float *A, float *C, long NB)

void compute(long NB, long DIM, float *A[DIM][DIM]) {

   for (long j = 0; j < DIM; j++) {

      for (long k= 0; k< j; k++) {
         for (long i = j+1; i < DIM; i++) {
            sgemm_tile( A[i][k], A[j][k], A[i][j], NB);
         }
      }

      for (long i = 0; i < j; i++) {
         ssyrk_tile( A[j][i], A[j][j], NB);
      }

      spotrf_tile( A[j][j], NB);

      for (long i = j+1; i < DIM; i++) {
         strsm_tile( A[j][j], A[i][j], NB);
      }
   }

}
```

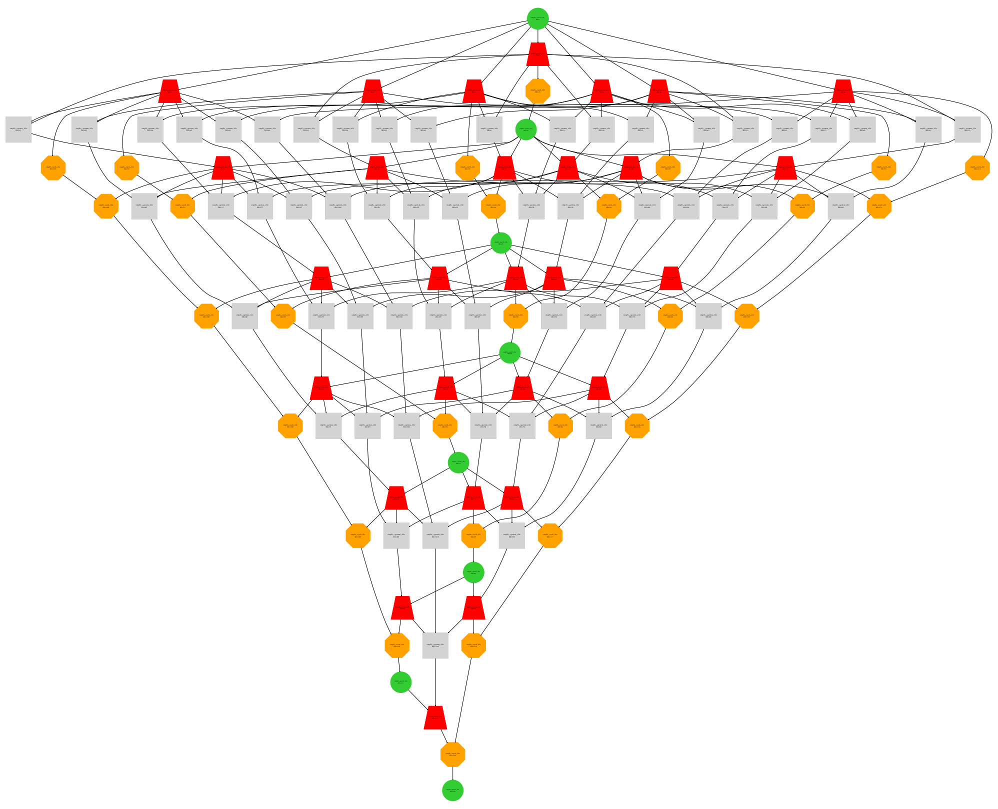
Figure 2.8: OmpSs implementation of Cholesky

process is parallelized using OmpSs.

MPI/OmpSs synergizes dataflow execution with message passing, providing high and robust performance. MPI/OmpSs allows a programmer to taskify functions with MPI transfers and thus relate the messaging events to dataflow dependencies. For example, a task with *MPI_Recv* of some buffer gets that buffer from the network and locally stores (*output* pragma directionality) it to the memory. This way, the arrival of the MPI message is related to data-dependencies among tasks. Then, the runtime can schedule OmpSs tasks in a way that overcomes strong synchronization points of pure MPI execution. Marjanovic *at. el.* [63] showed that apart from better peak GFlop-



(a) legend



(b) dependency graph

Figure 2.9: Dependency graph of Cholesky

s/s performance, compared to MPI, MPI/OmpSs delivers better tolerance to network limitations and external perturbations, such as OS jitters.

In MPI/OmpSs, the runtime dedicates one high-priority thread only for executing tasks with MPI transfers. A task with MPI call may be very inefficient, because it may spend significant time stalled – waiting on some blocking transfer. It would be very beneficial if the runtime could preempt the stalled task, and dedicate the computing resources to some task that is ready to compute useful work. OmpSs solves this problem by instantiating as many working threads as cores in the node, plus one additional thread that dedicates only to executing tasks with MPI calls. Then, whenever the MPI thread gets stalled waiting for the message, one working thread preempts the MPI thread in order to do some useful work. When the blocking MPI call completes, the MPI thread takes over the control of the core and proceeds. Practice shows that it is very beneficial to have communication thread of higher priority than the computation ones. This allows instant preemption as soon as the blocking MPI call finishes, providing a very efficient and flexible mechanism for assuring progress of MPI messages. An alternative implementation of MPI/OmpSs [62] offers a *restart* directive that can restart a task that gets blocked on MPI calls. When a communication task gets blocked on some MPI call, it automatically restarts and goes to the ready queue, offering computation resources to other tasks with useful computation. However, this implementation requires more programming effort.

2.2.4 Example - MPI/OmpSs vs. MPI/OpenMP

Compared to MPI/OpenMP, MPI/OmpSs provides higher potential for lookahead of MPI iterations. Lookahead of depth d means that for sending the message produced in the iteration n, the execution must complete all the computation of the iteration $n - d$. Thus, lookahead of depth 0 means that sending a message of iteration n requires that all the tasks in the iteration n are executed. Similarly, lookahead of depth 1 means that sending a message of iteration n requires that all the tasks from the iteration $n - 1$ are executed. In this Section, based on a simple example, we illustrate the potential lookahead for MPI/OpenMP and MPI/OmpSs programming models.

Figure 2.10 illustrates a simple MPI code. The code consists of sections that produce buffer a (the loop calling $compute1$ and $independent\_work1$), communicate the data (sending buffer a and receiving buffer c in $MPI\_SendRecv$), and consuming the received buffer c (the loop calling $compute2$ and $independent\_work2$). The only data-dependencies in this code are from all instances of $compute1$ to $MPI\_SendRecv$, and from $MPI\_SendRecv$ to all instances of $compute2$.

In this program, MPI/OmpSs can easily achieve lookahead and overlap. An intuitive OmpSs parallelization encapsulates all the functions into OmpSs tasks. Especially, task $MPI\_SendRecv$ is marked to be a communication task. The runtime automatically detects all the data-dependencies – from all instances of $compute1$ to

$MPI\_SendRecv$, and from $MPI\_SendRecv$ to all instances of $compute2$. There are no dependencies between task instances of $independent\_work1$ and $independent\_work2$. The communication task ($MPI\_SendRecv$) gets *outstanding* scheduling priority – the priority that allows it to preempt other tasks. Since they give dependency to the communication task, the instances of $compute1$ get a high scheduling priority. Therefore, the instances of $compute1$ are scheduled first for parallel execution. As soon as they complete, the runtime schedules $MPI\_SendRecv$. As $MPI\_SendRecv$ blocks on receiving array c, the runtime preempts it and schedules parallel execution of tasks $independent\_work1$ and $independent\_work2$. As soon as the message arrives, task $MPI\_SendRecv$ finishes, and all the tasks that are not executed so far are free to execute concurrently. This type of execution brings two benefits:

- overlap of computation and communication – while an MPI process waits for the incoming message, instances of $independent\_work1$ and $independent\_work2$ do some useful work

- lookahead – an MPI process can perform its communication ($MPI\_SendRecv$) before completing all instances of $independent\_work1$. This allows parallel execution of tasks from different sides of the communication request (instances of $independent\_work1$ can execute concurrently with instances of $independent\_work2$).

```
int main(int argc, char *argv[]) {
   const int N = 100000;
   int i, a[N], b[N], c[N];

   for (i = 0; i < N; i++) {
   compute1(a[i]);
   independent_work1(b[i]);
   }

   MPI_SendRecv ( /* send buf */ a,
                  /* recv buf */ c);

   for (i = 0; i < N; i++) {
   compute2(c[i]);
   independent_work2(b[i]);
   }

   return 0;
}
```

Figure 2.10: Simple MPI code to study potential lookahead

On the other hand, in this example, MPI/OpenMP cannot achieve neither lookahead nor overlap. Using an OpenMP parallel for construct, the programmer can parallelize the loops in the code. However, this construct has an implicit barrier at the end of the loop, thus executing $MPI\_SendRecv$ must wait all the instances of $compute1$ and $independent\_work1$ from the first loop to finish. Also, since each iteration of the second loop calls $compute2$, none of the iterations can start before the transfer is finished. The resulting parallel execution has:

- no overlap – during $MPI\_SendRecv$ the processors are idle
- no lookahead – there is no concurrency between some work before the communication and some work after the communication.

In order to achieve lookahead using OpenMP, the programmer must use advanced synchronization techniques or severe restructuring of the code. Since functions $independent\_work1$ and $independent\_work2$ are independent from other tasks, the programmer can restructure the code and put these tasks into separate loops. Then the programmer must break the $MPI\_SendRecv$ call into the non-blocking send and the non-blocking receive call. Thus, the programmer must explicitly assure that the communication will be overlapped with computation. Furthermore to achieve lookahead, the programmer must avoid the simple $parallelfor$ construct and therefore avoid the implicit barrier it forces. Thus, the programmer must implement a customized synchronization that will assure correctness of the execution. The complexity of this approach makes it impractical for implementation in complex applications.

2.3 Tools

Our study is based on the infrastructure for trace driven simulation developed in Barcelona Supercomputing Center. The work in this thesis also included further development of these tools. Furthermore, we designed new Valgrind-based tools and integrated the into the already existing environment. All development infrastructure designed throughout this thesis is open-source. In this Section, we describe only the tools used to build our ecosystem. We comment other research/commercial tools related to our study in Chapter 7.

2.3.1 mpitrace

mpitrace [55] is a tracing library for instrumenting parallel execution. mpitrace intercepts calls to certain functions and emits to the trace the events that mark these occurrences. By default, it intercepts MPI functions, timestamps the occurrence of the calls and emits that information to the trace. In addition, mpitrace can record information collected from various hardware counters. The output of mpitrace is a tracefile that can be fed to Paraver for visualization or to Dimemas for further simulation of parallel execution.

mpitrace is implemented as a light, easy to use, dynamic library. The instrumentation is activated by dynamically preloading the tracing library, requiring no changes of the original application's code. The intercepting mechanism is very light, providing the overhead of around $200ns$ per intercepted call, and around $6\mu s$ for reading hardware counters [64]. Furthermore, mpitrace uses conventional optimization techniques, such as: sampling (switching on/off instrumentation), filtering (configuring which subset of the events should be tracked), and buffering (collecting events locally and flushing them to the disc in bulk).

2.3.2 Valgrind

Valgrind [66] is a virtual machine that uses just-in-time (JIT) compilation techniques. The original code of an application never runs directly on the host processor. Instead, the code is first translated into a temporary, simpler, processor-neutral form called Intermediate Representation (IR). Then, the developer is free to do any translation of the IR, before Valgrind translates the IR back into machine code and lets the host processor run it.

Valgrind is a very fertile environment for developing new tools based on binary translation. The Valgrind framework is divided into three main areas: core and guest maintenance (coregrind), translation and instrumentation (LibVEX), and user instrumentation libraries. The user can use this modular organization and build a new tool as a plugin instrumentation library. Valgrind tools are most useful for inspecting memory-related issues in user-space programs; they are not suitable for time-specific issues or kernel-space instrumentation/debugging.

2.3.3 Paraver

Paraver [56] is a parallel program visualization and analysis tool. Paraver provides a qualitative perception of the application's time-behavior by visual inspection. Moreover, it provides a quantitative analysis of the run. Paraver reveals bottlenecks in parallel execution and discovers areas of suboptimal performance. This functionality is essential in the process of optimizing parallel applications. Paraver is not tied to any programming model. It provides a flexible three-level model of parallelism on which the parallel execution should be mapped. Currently, Paraver supports MPI, OpenMP, OmpSs, pthreads and CUDA, as well as the hybrid programming models such as MPI/OpenMP and MPI/OmpSs.

Paraver is a flexible data browser that provides a huge analysis power. Paraver's flexibility comes from the fact that the trace format is independent of the programming model. Thus, the visualization requires no changes in order to support some new programming model. In order to visualize time-behavior of some new programming model, the programming model needs to express its performance data in the Paraver independent trace format. On the other hand, Paraver's expressing power comes from the possibility to easily program different metrics. The tool provides filter and arithmetic operations on the essential metrics in order to generate new derived and more complex metrics. This way, from one trace, the tool can provide different views with different performance metrics. To capture the experts' knowledge, any view can be saved as a Paraver configuration file. Recomputing the view on new data is as simple as loading a file.

2.3.4 Dimemas

Dimemas [45] is an open-source tracefile-based simulator for analysis of message-passing applications on a configurable parallel platform. The Dimemas simulator reconstructs the time behavior of a parallel application on a machine modeled by a set of performance parameters. Dimemas also allows to model aspects of the MPI library and the operating system. As the input, Dimemas takes the trace produced by mpitrace and the configuration file of the modeled parallel machines. As the output, Dimemas generates trace files processable by two performance analysis tools: Paraver and Vampir [65]. These visualization tools enable the user to qualitatively inspect the simulated parallel time-behavior.

The initial Dimemas architecture model considered networks of Shared-Memory Processors (SMP). Dimemas configuration file defines the modeled target machine, specifying the number of SMP nodes, the number of cores per node, the relative CPU speed, the memory bandwidth, the memory latency, etc. The simulation also allows different task to node mappings. The communication model consists of a linear model and nonlinear effects, such as network congestion. The configuration file parametrizes the network specifying the bandwidth, the latency, and the number of global buses (modeling how many messages can concurrently travel throughout the network). Also, each processor is characterized by the number of input/output ports that determine its injection rate to the network.

Dimemas simulates the trace by replaying computation records and re-evaluating communication records. Dimemas trace consists basically on the two types of records: computation records specifying the duration of the computation burst; and communication records specifying transfer parameters such as the sender, the receiver, the message size, the tag, whether the operation was blocking or non-blocking, etc. During the simulation, the computation records are simply replayed – each computation burst lasts as it is specified in the trace. Conversely, each communication record is reevaluated – each transfer request is evaluated considering the specified configuration of the target machine and the newly calculated transfer time is incorporated in the simulated run.

3 Motivation

Nowadays, programmers struggle to deliver high-performance parallel software. In order to solve some problem in parallel, the programmer must organize a coordinated execution of multiple processing units. Parallel programming models facilitate this process by offering various parallelization constructs. High-level parallelization constructs allow easy parallelization, but often result in unsatisfactory performance. On the other hand, low-level parallelization constructs potentially provide high performance, but demand a lot of programming effort. Consequently, the practice shows that parallel executions often experience performance that is far from optimal. In the following Sections we further explain the inefficiencies of parallel programming. We also introduce some techniques for parallelization tuning.

3.1 MPI programming

MPI programming especially suffers from suboptimal performance, because the programmer must directly control various performance related details, e.g. algorithm complexity, load balance, cache conflicts, noise. MPI provides a very low level mechanisms of explicit parallelization. The programmer must explicitly partition the workload among processes with separate address spaces. Also, the programmer must explicitly orchestrate communication among processes. Furthermore, in order to improve the performance, the programmer must optimize the application by-hand. These optimizations may be very complex and by rule significantly reduce programmers productivity and clarity of the code.

3.1.1 Bulk-synchronous programming

Pressured by the high overhead of MPI routines, MPI programmers tried to minimize the number of transfers. Initially, MPI routines executed on the host processor (without offload). In order to reduce this overhead, programmers strove to generate fewer calls to MPI routines. This reasoning led to establishing bulk-synchronous programming [48] as a practical standard for MPI programming. Bulk-synchronous programming has one clear goal – maximizing the size of each MPI transfers and therefore reducing the overhead per sent byte ratio.

However, bulk-synchronous programming causes significant communication delays. Bulk-synchronous programming tries to maximize the uninterrupted computation in order maximize the size of each MPI transfer. However, during the long computation phase, the network is idle. On the other hand, during the communication phase, the processors are idle, waiting for the communication to finish. These communication delays can significantly harm the overall performance, especially at large-scale.

To mitigate communication delays, the community makes an effort to accelerate interconnection networks. Faster networks reduce message transfer time and consequently shorten communication delays. However, modern networks achieve high bandwidth and low latency more often in benchmarks than in realistic workloads. Latest studies [35] show that for scientific codes, the networks are over-designed and underutilized. Therefore, there is a need for a new approach to solve problems of communication delays. The goal of a new approach should be not to create new and fast cycles on the network, but rather to profit more from the already existing cycles.

3.1.2 Communication Computation Overlap

Communication-computation overlap is already recognized as a promising technique to reduce communication delays. Overlap enables processors to be at the same time busy computing and communicating. Thus, it allows message transfer time to be overlapped with useful computation. Overlap provides two potential benefits to parallel execution: 1) execution speedup; and 2) relaxation of network constraints without consequent performance penalty.

Overlap is usually achieved by software optimization techniques that leverage non-blocking MPI communication. The needed code refactoring decouples each blocking transfer into non-blocking initialization and the transfer's wait-request. Computation between the initialization and the wait-request is useful computation that is independent of the ongoing transfer. This computation can overlap and hide message transfer time. Therefore, the more independent computation the optimization exposes, the higher is the probability of overlap actually occurring. In the following Section, we describe this mechanism in more details by applying the needed refactoring on a simple code.

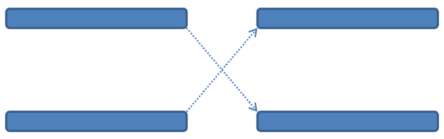
The mechanism of overlap

Figure 3.1 illustrates the case of bulk-synchronous programming that suffers from lack of overlap. The source code (Figure 3.1a) consists of one loop. In each iteration of the loop, the program locally calculates the message (function *work*), and then communicates the message using a blocking transfer (function $blocking\_MPI\_Sendrecv$). Figure 3.1b illustrates the execution of this code on two MPI processes. The sections of computation and communication are disjunctive, providing no overlap.

Conversely, Figure 3.2 illustrates the overlapped version of the previous code. Figure 3.2a shows the required code refactoring. Function *work* splits into three smaller functions ($pre\_independent\_work$, $transfer\_dependent\_work$ and $post\_independent\_work$). Function $transfer\_dependent\_work$ is the only one that actually uses the message received in the previous iteration of the loop and produces the message that is sent in the current iteration of the loop. Functions $pre\_independent\_work$ and $post\_independent\_work$ are independent of the transfered buffers. Also, the blocking transfer from Figure 3.1a splits into the initialization of the transfer ($nonblocking\_MPI\_Sendrecv$) and the wait-

```
for ( ... ) {
   work();
   blocking_MPI_Sendrecv();
}
```

(a) code

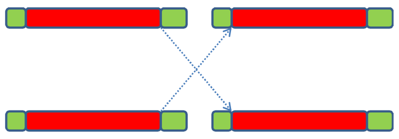


(b) execution

Figure 3.1: The case of nonoverlapped MPI

```
for ( ... ) {
   pre_independent_work();
   if (!first_iteration)
      wait_for_previous_MPI_Sendrecv();
   transfer_dependent_work();
   nonblocking_MPI_Sendrecv();
   post_independent_work();
}
wait_for_last_MPI_Sendrecv();
```

(a) code



(b) execution

Figure 3.2: The case of overlapped MPI

request (*wait_for_last_MPI_Sendrecv*). Figure 3.2b shows that the sections of independent work (the green sections) now overlap with the message transfer, thus reducing the communication delay.

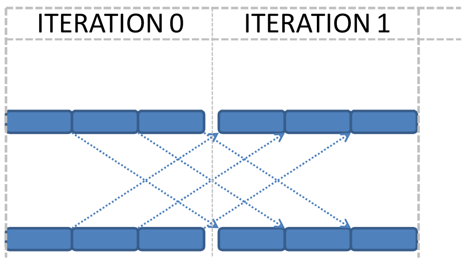
However, this code restructuring significantly reduces the clarity and maintainability of the code. Two lines in the non-overlapped code now expand into seven lines that expose and overlap independent work. As a result, the obtained structure of the code significantly obscures its usage. The obtained overlapped code has serious maintainability issues, causing very low programming productivity.

3.2 Our effort in tuning MPI parallelism

Our effort in tuning MPI parallelism targets eliminating communication and synchronization delays. Our goal is to mitigate these stalls by introducing an automatic technique that achieves maximal potential overlap and requires no code refactoring of the targeted application.

```
for ( ... ) {
   wait_for_previous_MPI_Sendrecv( 1/3 );
   work( 1/3 );
   nonblocking_MPI_Sendrecv( 1/3 );
   wait_for_previous_MPI_Sendrecv( 2/3 );
   work( 2/3 );
   nonblocking_MPI_Sendrecv( 2/3 );
   wait_for_previous_MPI_Sendrecv( 3/3 );
   work( 3/3 );
   nonblocking_MPI_Sendrecv( 3/3 );
}
```

(a) code



(b) execution

Figure 3.3: Overlap with chunks

3.2.1 Automatic overlap

The state-of-the-art overlapping techniques achieve limited overlap and require significant code restructuring. The presented by-hand optimization provides limited overlap, because it overlaps transfer time only with transfer independent work. Moreover, the required code intervention requires significant refactoring that seriously harms programming productivity.

Our technique of automatic overlap

In this thesis, we design a technique that maximizes the potential overlap in an application. The technique consists of four mechanisms:

- **Message chunking**: Each original MPI message is partitioned into independent *chunks* consisting of one or more data elements.
- **Advancing sends**: Each chunk is sent as soon as it is produced.
- **Double buffering**: Two different buffers are used to differentiate the chunks being consumed at the current iteration and the incoming chunks for consuming at the next iteration.
- **Postponing receptions**: Each chunk is waited at the moment when it is really needed for consumption.

Compared to the overlapping technique illustrated in Figure 3.2, our technique additionally breaks the original message into chunks and then overlaps the communication time of each chunk independently. Figure 3.3 illustrates overlap with chunks in the case when the original message is broken into three chunks. The idea is that, ideally, each third of the computation in some iteration produces one third of the message to be sent in that iteration. Similarly, each third of the received message in some iteration enables execution of one third of the computation in the next iteration. Then, the first third of the message can be sent after 33% of the first iteration, and it must arrive before the start of the second iteration. Thus, the transfer of this first chunk can be overlapped with the resting 66% of the computation in the first iteration. Therefore, in our technique there is no need for extracting independent work – work that does not produce/consume the message. Conversely, this technique overlaps the transfer of one part of the original message with the production/consumption of the rest of that message.

Also, it is important to note that the patterns by which each MPI process locally computes on transferred data can seriously limit the potential for overlap. The overlap of chunks comes from the potential of the execution to advance partial sends and postpone partial receptions. In the presented example, the first chunk is produced after 33% of the iteration, allowing it to be overlapped with the resting 66% of the computation in that iteration. However, if that first chunk was finally produced at 80% of the iteration, its transfer could overlap with only 20% of the computation. Thus, the production/consumption patterns can seriously limit the potential for advancing/postponing message chunks. Therefore, our study of automatic overlap will pay a special attention to internal computation patterns of applications.

The goal of this thesis it to explore techniques that achieve chunked overlap from Figure 3.3b without consequently degrading maintainability of the code. The code restructuring that is needed to achieve the chunked overlapped execution seriously hurts the maintainability of the code (Figure 3.3a). Thus, our goal is to explore the techniques that will achieve chunked overlapped execution (Figure 3.3b) but still preserve the clarity of the code from Figure 3.1a.

Our contribution in exploring automatic overlap

This thesis tries to design and evaluate an automatic technique that, using a specialized hardware support, extracts the maximal potential overlap without the need to restructure the source code of an application (Chapter 5). In Section 5.2, we further describe our technique of automatic overlap. Furthermore, we introduce a new overlapping technique that we named *speculative dataflow*. Speculative dataflow is a speculative technique that, using a specialized hardware support, implements automatic overlap without any restructuring of the original MPI application. We demonstrate the feasibility of this speculative dataflow (Section 5.3) and evaluate its potential in the case of real scientific MPI applications (Section 5.4).

Also, throughout our study, we designed a development environment that can be very useful in further studies of overlap (Section 4.2). Given only the executable of a legacy MPI application, the environment automatically evaluates the potential overlap. Moreover, the visualization support allows comparison between the original and the overlapped execution of the targeted application. Using our environment, a programmer can quickly evaluate the potential benefits of overlap in any application. Thus, prior to any implementation effort, the programmer can estimate the potential benefits of intended overlapping technique and decide whether the implementation is worth the effort. Furthermore, using the visualization support, the programmer can identify bottlenecks of the intended implementation and explore solutions to overcome them.

3.3 MPI/OmpSs programming

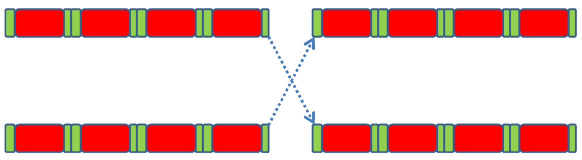
As already mentioned in Section 2.2.3, MPI/OmpSs generates very efficient and flexible execution. OmpSs enables out-of-order execution of tasks within each MPI process. Moreover, OmpSs runtime can dynamically schedule MPI transfers in order to efficiently hide communication delays. Also, OmpSs can expose additional parallelism within each MPI process, therefore increasing the overall scalability of the parallel execution.

3.3.1 Hiding communication delays

Figure 3.4 illustrates another case of non-overlapped pure MPI execution. In each iteration of the outer loop, the program computes the message and then sends the message using a blocking MPI call ($blocking\_MPI\_Sendrecv$). The program produces the MPI message in four independent iterations of the inner loop (four calls to function *work*).

```
for ( ... ) {
   for ( 0:4 ) {
      work();
   }
   blocking_MPI_Sendrecv();
}
```

(a) code

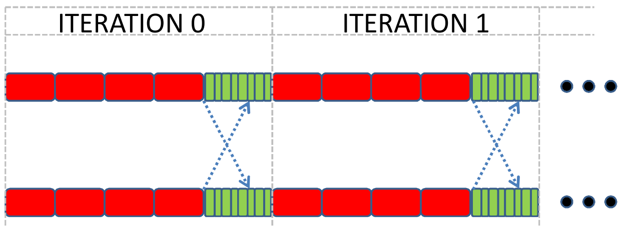


(b) execution

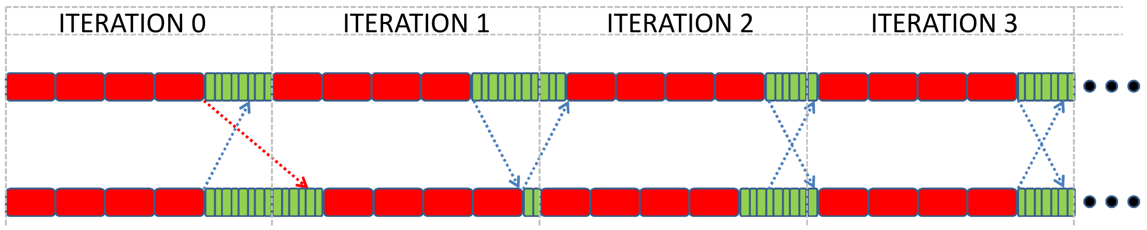
Figure 3.4: Example of nonoverlapped MPI

```
#pragma omp task ...
pre_independent_work();
#pragma omp task ...
transfer_dependent_work();
#pragma omp task ...
post_independent_work();
#pragma omp task ...
blocking_MPI_Sendrecv();

for ( ... ) {
   for ( 0:4 ) {
      pre_independent_work();
      transfer_dependent_work();
      post_independent_work();
   }
   blocking_MPI_Sendrecv();
}
```

(a) code



(b) execution



(c) execution with noise affecting traffic

Figure 3.5: Example of overlapped MPI/OmpSs

Without any code restructuring, this code achieves no overlap (Figure 3.4b).

In order to port this MPI code to MPI/OmpSs (Figure 3.5), the programmer must do a minor refactoring. The programmer must separate the code that actually computes the message (*transfer_dependent_work*) from the independent work (*pre_independent_work* and *post_independent_work*). No refactoring of the transferring routine (*blocking_MPI_Sendrecv*) is needed. Finally, the programmer must encapsulate all the presented functions into tasks (Figure 3.5a).

Then, the OmpSs runtime can dynamically schedule the instantiated tasks in order to achieve overlap (Figure 3.5b). The runtime identifies the task with an MPI call ($blocking\_MPI\_Sendrecv$) and sets it as the task with outstanding priority. Furthermore, the runtime identifies all tasks that give dependency to the communication task and sets their priority to high. Thus, the OmpSs runtime first executes the tasks that generate the message (*red* tasks). After these tasks finish, the runtime schedules the task with the MPI call. As the communication task blocks waiting for the message, the runtime preempts it and schedules the tasks with independent work (*green* tasks). This way, the runtime dynamically overlaps the message transfer time with the computation that executes independent work.

Furthermore, MPI/OmpSs execution is more tolerant to external noise. Figure 3.5c shows the case when one of the messages (the red transfer) becomes late (e.g. due to some contention of the network). Since the message is late, the second process cannot start the second iteration with instances of transfer-dependent tasks (red tasks). However, the runtime can schedule the execution of independent work at the beginning of the iteration 1. Thus, the lower process starts its iteration 1 with independent work, and as soon as the message arrives, it switches to computing its transfer-dependent work. This causes the lower process in the iteration 1 to be late sending the message needed by the upper process in the iteration 2. However, this latency is smaller than the originally introduced latency. In the following iterations, the message latency additionally relaxes, so both MPI processes enter the stable state at the beginning of iteration 4.

3.3.2 Additional parallelism within an MPI process

Moreover, OmpSs provides additional parallelism within each MPI process. If the target machine provides multiple cores per MPI process, tasks within each MPI process can execute concurrently. Figure 3.6 shows how the presented MPI/OmpSs code can execute if each MPI process is parallelized across two cores. For example, let us consider the case when all red tasks are mutually independent, and all green tasks are mutually dependent. Thus, instances of red tasks can execute in parallel, significantly reducing the total execution time. On the other hand, due to data dependencies, instances of green task must be serialized.

This example also illustrates how hard it is to anticipate how will an MPI/OmpSs

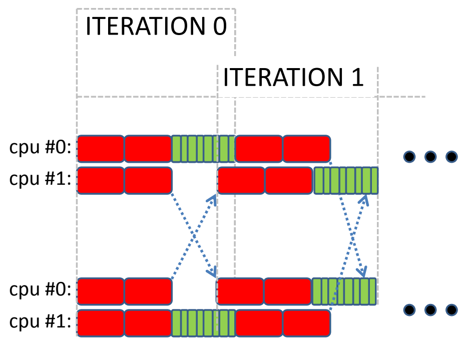


Figure 3.6: MPI/OmpSs execution with multiple cores per MPI process

application execute on a different target machine. Concurrency of red tasks accelerates computation local to each MPI process and consequently reduces the total computation time. However, accelerated MPI processes put more pressure on the interconnect, whose speed did not scale accordingly. Transfer independent work (green tasks) can overlap communication delays. Still, since green tasks cannot execute in parallel, they can overlap communication delays only for one core per MPI process. All these interrelated execution properties make it hard to predict the performance of MPI/OmpSs application on a parallel platform.

3.4 Our effort in tuning MPI/OmpSs parallelism

This thesis also explores techniques for tuning MPI/OmpSs parallelism. We want to investigate techniques to hide stalls in MPI/OmpSs execution, especially in the case of highly parallel target platform. First, we explore techniques to identify parallelization bottlenecks in MPI/OmpSs execution. Instrumenting an MPI/OmpSs code, our technique pinpoints critical code section – the code section whose optimization yields the highest overall speedup. Second, we explore strategies for exposing and incrementing OmpSs parallelism in MPI applications. We describe an iterative approach to test various task decompositions of the code and select the one that exposes the highest parallelism for the selected target machine.

3.4.1 Identifying parallelization bottlenecks

MPI/OmpSs exposes very irregular parallelism, making it hard to identify the best way to optimize the execution. In the example from Figure 3.6, there are many ways to accelerate the parallel execution. One way is to increase the number of cores per MPI process and increase the concurrency of execution of red tasks. On the other hand, using faster cores would accelerate both red and green tasks. A faster network would reduce transfer time. Manual code refactoring could target red tasks (because they are computation intensive) or green tasks (because they are serialized). Dedicated accelerators may execute some sections of the code. However, even in this simple code, each of these optimizations would significantly affect the execution and change tasks scheduling in a very unpredictable way. Thus, it is very hard to estimate the potential benefits of these optimization techniques and identify which technique is the most cost-effective. We further illustrate this issue in the motivating example from Section 6.1.1.

This thesis tries to gain a better understanding and control of the parallelism exposed by MPI/OmpSs, to evaluate how MPI/OmpSs applications would execute on future machines and to predict the execution bottlenecks that are likely to emerge. We explore how MPI/OmpSs applications could scale if the target parallel machine offers hundreds of cores per node. Furthermore, we investigate how this high parallelism within the node would reflect on the constraints of the interconnect. We especially focus on identifying critical code sections in MPI/OmpSs. We explore techniques to quickly evaluate, for a given MPI/OmpSs application and the selected target machine, which code section should be optimized in order to gain highest performance benefits.

Throughout our study of parallelization bottlenecks, we developed **mpisstrace** (Section 4.3) – a development environment that enables trace-based simulation of MPI/OmpSs execution. The environment simulates MPI/OmpSs execution on a configurable parallel platform, allowing detailed analysis of various influences. Using our environment, a programmer can easily explore the scalability of an MPI/OmpSs code. Furthermore, using the visualization support, a programmer can qualitatively inspect parallelization bottlenecks. Finally, in Section 6.1 we demonstrate how a programmer can use the environment to automatically identify the **critical code section** – the code section whose optimization would yield the highest benefit to the overall execution time.

3.4.2 Searching for the optimal task decomposition

Another way to tune parallelism in MPI/OmpSs is to select a different task decomposition. Figure 3.6 points that executing each MPI process on two cores already generates execution stalls. One way to reduce these stalls could be to refine the decomposition in order to avoid serialization of the green tasks. Other way would be to refine red tasks in order to extract more computation that is independent of the message transfer. However, testing different decomposition is very hard. For testing some decomposition, a programmer must generate correct MPI/OmpSs code that implements that decomposition and then measure the obtained parallelism. We further illustrate this issue in the motivating example from Section 6.2.1.

This thesis studies techniques to quickly explore the potential parallelism in applications. We provide mechanisms for the programmer to easily evaluate potential parallelism of any task decomposition. Furthermore, we describe an iterative trial-and-error approach to search for a task decomposition that will expose sufficient parallelism for a given target machine (Section 6.2). Finally, we explore potential of automating the iterative approach by capturing the programmers' experience into an expert system that can autonomously lead the process of finding efficient task decompositions (Section 6.3).

In our study of potential parallelism in applications, we designed **Tareador** (Section 4.4) – a tool to help porting MPI applications to MPI/OmpSs programming model. Tareador provides a simple interface to propose some decomposition of a code into OmpSs tasks. Then, based on the proposed decomposition, Tareador dynamically identifies data dependencies among the annotated tasks, and automatically estimates the potential OmpSs parallelization. Furthermore, Tareador gives additional hints on how to complete the process of porting the application to OmpSs. Using Tareador, throughout trial-and-error top-to-bottom iterative approach, a programmer can test various task decompositions and find one that exposes sufficient parallelism to efficiently use the target parallel machine. Also, we designed an autonomous driver that runs Tareador to automatically explore potential task decompositions of an application.

4 Infrastructure

The research in this thesis is based on the trace-driven simulation using the tools that are developed in BSC (Section 2.3). The initial trace-driven infrastructure includes mpitrace, Dimemas and Paraver. mpitrace library is a dynamic library that intercepts MPI related events and emits them to the trace. Paraver is a performance analysis tool that can visualize the traces obtained with mpitrace. Furthermore, Dimemas can replay the traces obtained with mpitrace to simulate the execution with the changed configuration of the target parallel machine.

In the conventional trace-driven simulation, simulating a new execution feature is done completely in the simulator. The tracing library instruments the execution of the code and inserts to the trace the events that describe that execution. Then, the simulator consumes the obtained trace, replaying the collected events and calculating new time-stamps according to the specified configuration of the target parallel machine. Thus, accounting for a new execution feature is done entirely in the simulator – the simulator takes the new feature into account when calculating the time-stamps of the simulated execution.

However, the conventional methodology makes it impossible to simulate low level architectural features in the highly parallel target machine. The problem arises from the need of simulators to be sequential. For instance, when simulating MPI execution, any change in the local time-stamps of one MPI process may change the order of messages on the network. Since the network is a shared resource, the simulator must explicitly synchronize all MPI processes on every network access. Thus, the simulator can hardly be parallelized. On the other hand, simulating numerous MPI processes causes a very large simulation. Thus, the simulation time impedes simulating a low level architectural feature on a big parallel system. To overcome this problem, we had to design a different simulation methodology that is based on modifying the trace in such a way that it models the new feature to be explored.

4.1 Simulation aware tracing

Using the state-of-the-art tools for trace-based simulation, we develop a new simulation technique that models the simulated feature in the earlier part of methodology – during the tracing of the application. As in conventional trace-driven simulation, our tracer instruments the execution and generates the trace of the real run. We will call this trace the *authentic* trace. However, apart from the records needed for the *authentic* trace, the tracer emits to the trace additional records related to the studied feature. Then, from the *authentic* trace and the additional events, we derive the $artificial$ trace – what would be the trace of the potential execution of the same application if it would include the studied feature. Then, the unchanged replay simulator replays both traces, providing a comparison between the actually executed run and the potential run that includes the studied feature.

Our methodology allows simulating the effect of a low-level feature on a highly parallel machine. By making the tracing process responsible for modeling a new execution feature, our methodology shifts the major computation effort from simulation phase to tracing phase. Since each MPI process in traced concurrently, the feature modeling computation is naturally parallelized across MPI processes. The parallelization of the feature modeling effort allowed us to make very complex simulations – simulations that model very low-level features on very large-scale parallel machines. In order to instrument low-level execution properties of execution (such as memory

```
PROCESS 0:                  PROCESS 1:                  PROCESS 2:                  PROCESS 3:
CPU_burst (0.112)           CPU_burst (0.111)           CPU_burst (0.112)           CPU_burst (0.121)
Broadcast (/* root */ 0)    Broadcast (/* root */ 0)    Broadcast (/* root */ 0)    Broadcast (/* root */ 0)
CPU_burst (0.121)           CPU_burst (0.122)           CPU_burst (0.132)           CPU_burst (0.112)
```

(a) original trace – broadcast implemented with collective call

```
PROCESS 0:            PROCESS 1:            PROCESS 2:            PROCESS 3:
CPU_burst (0.112)     CPU_burst (0.111)     CPU_burst (0.112)     CPU_burst (0.121)
Send ( -> 1 )         Recv ( 0 -> )         Recv ( 0 -> )         Recv ( 0 -> )
Send ( -> 2 )         CPU_burst (0.122)     CPU_burst (0.132)     CPU_burst (0.112)
Send ( -> 3 )
CPU_burst (0.121)
```

(b) changed trace – broadcast implemented with point-to-point calls (one-to-many)

```
PROCESS 0:            PROCESS 1:            PROCESS 2:            PROCESS 3:
CPU_burst (0.112)     CPU_burst (0.111)     CPU_burst (0.112)     CPU_burst (0.121)
Send ( -> 1 )         Recv ( 0 -> )         Recv ( 1 -> )         Recv ( 2 -> )
CPU_burst (0.121)     Send ( -> 2 )         Send ( -> 3 )         CPU_burst (0.112)
                      CPU_burst (0.122)     CPU_burst (0.132)
```

(c) changed trace – broadcast implemented with point-to-point calls (cyclic)

```
PROCESS 0:            PROCESS 1:            PROCESS 2:            PROCESS 3:
CPU_burst (0.112)     CPU_burst (0.111)     CPU_burst (0.112)     CPU_burst (0.121)
Send ( -> 1)          Recv ( 0 -> )         Recv ( 1 -> )         Recv ( 0 -> )
Send ( -> 3)          Send ( -> 2 )         CPU_burst (0.132)     CPU_burst (0.112)
CPU_burst (0.121)     CPU_burst (0.122)
```

(d) changed trace – broadcast implemented with point-to-point calls (logarithmic)

Figure 4.1: Simulating different implementations of broadcast

accesses), we often designed tracers based on binary translation tools (Valgrind tools). These tracers can instrument the execution at the level of a single instruction. The effect of the new feature on every single instruction is calculated and incorporated into the *artificial* trace of each MPI process. Finally, the simulator replays the trace, propagating the effect of the new feature across the whole MPI execution.

4.1.1 Illustration of the methodology

This Section illustrates one simple application of our methodology. We present an example of using this methodology to test various implementations of the collective broadcast call.

A Dimemas trace consists of computation bursts and communication requests. A computation burst specifies only the duration of the computation that an MPI process executes sequentially. A communication request specifies the communication among processes, marking the processes that are involved in the communication and the parameters of the transfer. A communication request can represent a point-to-point or a collective communication.

Figure 4.1 illustrates our methodology on the example that explores the effects of different collective communication patterns. Figure 4.1a illustrates the *authentic* trace collected by the *mpitrace* library. All the processes compute for some time and then communicate using a broadcast call. After the communication, the processes continue computing.

From the *authentic* trace, we offline generate various *artificial* traces that use different implementations of the broadcast. The *artificial* trace is a copy of the *authentic* trace with the broadcast event replaced with some equivalent point-to-point implementation. Thus, the first implementation uses the one-to-many transfer pattern (Figure 4.1b), the second uses the cyclic transfer pattern (Figure 4.1c) and the third uses the logarithmic transfer pattern (Figure 4.1d). Then, all these four traces (one *authentic* and three *artificial* ones) are replayed with the legacy Dimemas simulator, showing the potential performance of all of the implementations.

In the following Sections, we introduce some advanced simulation environments that we developed using the described methodology. Section 4.2 describes the environment for identifying potential communication/computation overlap in applications. Section 4.3 describes the environment that replays MPI/OmpSs execution. Finally, Section 4.4 describes Tareador – the environment that identifies the potential parallelism in sequential codes.

4.2 Framework to identify potential overlap

For a programmer that wants to increase the overlap in his application, it is very important to anticipate the benefits of the targeted technique, and thus decide in advance whether implementing that technique is worth the effort. Refactoring applications to achieve more overlap is an optimization that can be very time consuming. Thus, in order to engage in this effort, the programmer should be convinced that the optimization will yield a significant performance improvement. Therefore, it would be very useful to have an environment that would automatically estimate the performance benefits of the targeted overlapping technique.

To respond to this demand, we designed a simulation framework that automatically quantifies the potential benefits of overlap in scientific MPI applications. To our knowledge, this is the first work that uses a simulation methodology to study overlap. The simulated overlapping technique works at the MPI level, by automatically capturing all MPI messages and trying to overlap those messages with useful computation of the application. The overlapping technique consists of the following mechanisms: message chunking, advancing sends, double buffering, and post-postponing receptions (as described in Section 3.2.1). Moreover, the programmer can propose a custom overlapping technique, and test how much an application would benefit from it.

Our framework is an automatic and easy-to-use approach to obtain a rich simulation output that can significantly increase the overall understanding of communication/computation overlap. The simulation framework allows us to get predictions quicker, and furthermore evaluate the impact of different network properties. The main advantage of this approach is that it automatically predicts the benefits of overlap in scientific MPI applications, without the need to know or understand the application's source code. The second advantage is that our simulation framework can visualize the simulation's output, allowing us to qualitatively inspect differences between the non-overlapped and overlapped executions. Using this feature, a programmer can identify bottlenecks in the overlapping technique and try to avoid them.

The simulation framework (Figure 4.2) is based on the integration of three widely used tools: the binary translation tool *Valgrind*, the network simulator *Dimemas*, and the visualization tool *Paraver*. An MPI application executes in parallel, with each process running on its own Valgrind virtual machine. Each of these virtual machines implements an instance of the designed tracing tool. The tool instruments the original application and extracts the trace of the *authentic* (non-overlapped) execution, while at the same time, it generates what would be the trace of the $artificial$ (overlapped) execution. Then, Dimemas simulator uses the traces obtained from each MPI process and off-line reconstructs the application's time-behavior on a configurable parallel platform. Finally, Paraver visualizes the obtained time-behaviors, allowing to qualitatively study the effects of the communication-computation overlap.

The second output of the framework are the production/consumption patterns of the messages. As explained in Section 3.2.1, production/consumption patterns directly

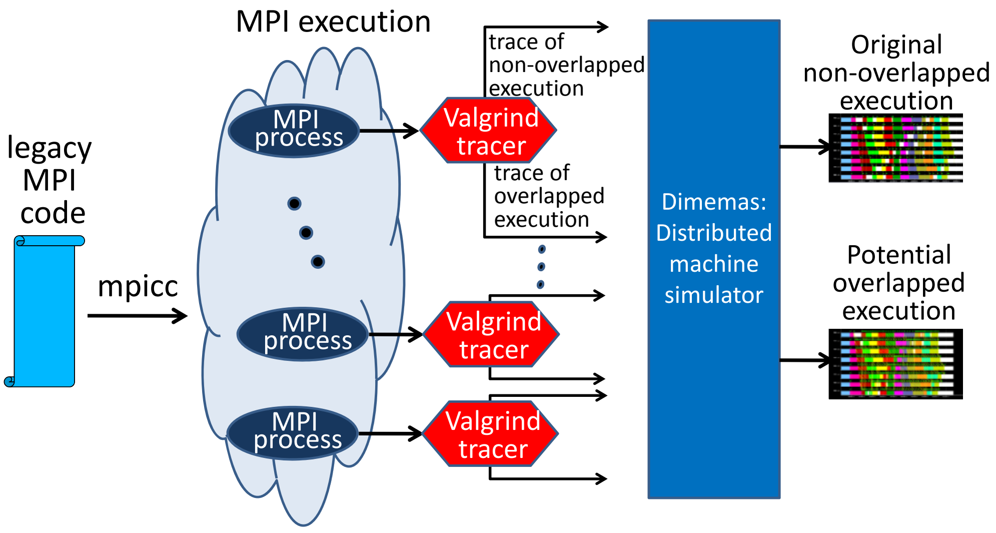

MPI execution
legacy MPI code
mpicc
MPI process
MPI process
MPI process
Valgrind tracer
Valgrind tracer
Valgrind tracer
trace of non-overlapped execution
trace of overlapped execution
Dimemas: Distributed machine simulator
Original non-overlapped execution
Potential overlapped execution

Figure 4.2: The framework for studying overlap integrates Valgrind, Dimemas, and Paraver.

determine the potential for advancing partial sends and postponing partial receives, thus strongly influencing the overall potential for overlap. Hence, our study must bring the potential overlap in the application in relation with the application's computation patterns. Therefore, our framework will also record the patterns by which each process produces the buffers that it sends and consumes the buffers that it receives.

4.2.1 Implementation details

Our major implementation effort consisted on designing a Valgrind tracing tool. The tracer leverages two key Valgrind functionalities for dynamic analysis of applications: wrapping function calls and tracking memory accesses (loads and stores). The tool wraps each MPI call to read the parameters of the transfer and tracks each memory activity to monitor accesses to the transferred data. Furthermore, the tool needs additional data structures to keep track of the transfers' state and production/consumption progress of every chunk. Finally, the tracer obtains time-stamps by scaling the number of executed instruction by the average MIPS rate observed in a real run.

In every run, the tracing tool generates one non-overlapped (*authentic*) and two overlapped (*artificial*) Dimemas traces. The non-overlapped trace describes the original execution of a legacy code by emitting two types of Dimemas trace records: computation records specifying the length of the original computation bursts; and communication records specifying the MPI message parameters. In addition to that tracing methodology, the first overlapped trace identifies within the original computation bursts, the points where partial data can be sent/is needed. Then it automatically splits each original message into various chunks and injects the chunked communication requests after the identified data is fully produced (for sends) and actually first needed (for receptions). Furthermore, in order to stress the influence of production/consumption patterns, the tool generates the second overlapped trace which assumes that the application's production/consumption patterns are ideal. This tracing methodology models an ideal computation pattern by uniformly distributing the chunked communication requests throughout the original computation bursts.

To model overlapped execution, the Valgrind tracer must intercept and process each application's load and store access. For each load in the application, the tool checks whether the requested element belongs to some incoming chunk that is not received so far. If so, the tracer emits a Dimemas *wait-for-receive record* for the corresponding chunk and marks that chunk as already received. Therefore, the tool guarantees that the wait for each incoming chunk is at the point where that chunk is needed for the first time. Similarly, for each store in the application, the tool checks whether the accessed element belongs to some chunk to be sent. If so, it refreshes the time of the chunk's last update. The tool uses that information in the trace generation process described below.

Also, the tracer tool has to intercept each MPI call in order to reinterpret the original communication using new chunks. When the tracer intercepts a receive call, it emits a Dimemas *non-blocking-receive record* for each chunk of the original message. This way, the tracer initiates the transfers of chunks and proceeds, waiting for the chunks to be received as late as possible – when those chunks are actually needed for consumption. On intercepting a MPI send call, the tool consults the time of the last update of every chunk in the message – the information generated during the production tracking. Using this data, the tracer emits a Dimemas *send record* of every chunk at the moment of the last update of that chunk, therefore generating the trace in which every chunk is sent at the exact moment when its final version is produced.

The tracer breaks each collective communication into the corresponding sequence of point-to-point transfers, and then overlaps each obtained transfer independently. Note that collective communication operations are performed in Dimemas without assuming any collective hardware support on the network.

The tracer obtains time-stamps in terms of the number of instructions executed in computation bursts outside of MPI calls. To represent time, the number of executed instructions is scaled by the average MIPS rate observed in a real run. The adopted notion of time obviously ignores many real world factors. The first is that it does not count time inside MPI routines. That is done intentionally, because MPI overheads could significantly exceed the lengths of computation phases that we are interested in. Also, the adopted model excludes effects of memory hierarchy misses (cache, TLB...) and context switches. This design decision keeps the model simple and easily controllable. It captures the application properties of interest and isolates the system from the undesirable and hard to control effects. However, the model can be extended in the future to include additional parameters.

Recording message production/consumption

As already explained, the potential overlap is very dependent on the message production/consumption patterns. To study production patterns, the tracer must record all stores to a buffer that is to be sent. We adopt as one production interval of some buffer the execution between two consecutive send calls of that buffer. During this interval, the tracer records each store to the communicated buffer and marks the store's relative time since the last send call. At the end of the interval, the tracer flushes the collected records, normalizing the relative time of each store with the duration of the interval. The obtained records now contain the information about all the writes into the production buffer and the percentage of production interval progress at which each of those writes occurred.

Similarly, to study consumption patterns, the tracer has to record all loads from the received buffer. We adopt as one consumption interval of some buffer the execution between two consecutive receive calls of that buffer. Now the loads from the buffer are examined, and the list of all time-stamped loads is generated. Finally, the obtained list contains all the loads from the received message and the percentage of the computation phase progress at which each of them occurred.

Predicting future sends

In order to monitor production of data that will be sent, it is crucial to know the address of the sending buffer before the actual send is issued. Without this knowledge, the tracer cannot know the bounds of the buffer that should be instrumented. To tackle this issue, we assume a deterministic execution of the studied applications for determining future transfers. More precisely, we assume that each MPI process has a deterministic sequence of MPI transfers and memory allocation requests. Then, two consecutive tracings of the program are used to obtain the final traces.

The first pass just marks the sequence of MPI transfers that occurred, while the second pass assumes repetition of that sequence. The first pass collects only temporal traces of MPI transfers and all memory allocations. The second pass reads the sequence of transfers from the previous execution. Only these transfers with their parameters are now predicted to occur. Also, during the second pass, memory allocation requests and their returned addresses are intercepted and compared to the allocations of the first pass. Based on a comparison of the returned addresses, the second pass dynamically updates the addresses of the expected transfers. This algorithm guarantees that the addresses of the future transfers will be determined early enough so the tracer can monitor the entire production transferred buffer.

4.3 Framework to replay MPI/OmpSs execution

For novel programming models, such as MPI/OmpSs, it is very important to prove that the programming model can efficiently utilize future parallel machines. MPI/OmpSs mixes message passing and dataflow dynamic scheduling, exposing very irregular parallelism. Thus, the programmer can hardly anticipate how MPI/OmpSs execution would perform on a different parallel machine. Moreover, it would be very hard to estimate the potential benefits of optimizing some section of the code or deploying some part of the application on dedicated accelerators. Therefore, it is very hard to estimate efficiency of MPI/OmpSs when using new heterogeneous parallel machines.

To tackle this problem, we designed a trace-based simulation framework to replay MPI/OmpSs execution on a parallel machine. The legacy MPI simulation environment (mpitrace, Dimemas, Paraver) allow replaying MPI execution varying the properties of the target parallel machine. We extended the legacy environment, enabling it to replay MPI/OmpSs execution. This allowed us to conduct various parametric studies of MPI/OmpSs.

Our environment deals with very complex parallel execution, offers fast and flexible simulation and provides a rich output. Up to our knowledge this is the first simulation environment that can simulate parallel execution that integrates MPI with dataflow programming model. Tracing part of the simulation is very light, reporting to the trace only events related to MPI and OmpSs activity. On the other hand, post-mortem simulation allows to process 4.000 tasks in less than 10 seconds, and 2 million tasks in about 10 hours. Also, the environment provides conventional techniques for speeding up trace driven simulation, such as filtering and sampling. Finally, we believe that the biggest contribution of the environment is its flexibility and rich output. The user can easily change the targeted platform and visually (qualitatively) inspect how will the dataflow parallelism react.

In Section 6.1, we show some of the possible usages of the environment. More specifically, we illustrate three usages of the environment

1. **Educating newcomer programmers**: The programmer can use the environment to easily and quickly simulate MPI/OmpSs execution on a configurable parallel platform, and study in detail various influences on the dataflow execution. Especially, the programmer can use the visualization support to qualitatively inspect the simulated time-behaviors and learn more about the mechanisms of dataflow parallelism.

2. **Identifying potential parallelism**: The programmer can use the environment to automatically simulate how would his application perform on target machines that provide different degrees of parallelism.

3. **Identifying parallelization bottlenecks**: The programmer can use the environment to automatically identify the **critical section** of the code – the section whose optimization would yield the highest benefit to the overall execution time.

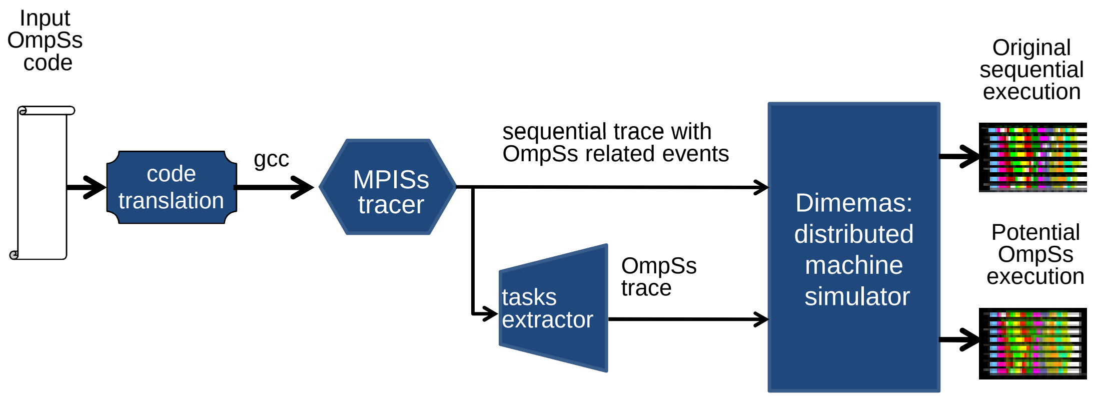

Input OmpSs code
code translation
gcc
MPISs tracer
sequential trace with OmpSs related events
tasks extractor
OmpSs trace
Dimemas: distributed machine simulator
Original sequential execution
Potential OmpSs execution

Figure 4.3: The environment integrates Mercurium code translator, MPISS tracer, tasks extractor, Dimemas simulator and Paraver visualization tool.

4.3.1 Implementation details

The main idea of our methodology is 1) to instrument the execution without OmpSs parallelism; and then 2) to reconstruct what would be the execution with OmpSs parallelism. The input code executes without task parallelism (ignoring the OmpSs pragma annotations). Thus, the *authentic* trace describes the execution without OmpSs parallelism. Still, in runtime, we evaluate pragma annotations and incorporate the evaluation results into the trace of the run. Then, offline, from the collected enriched *authentic* trace, we reconstruct the final $artificial$ trace – what would be the trace of the potential run with OmpSs parallelism. Finally, we replay the reconstructed $artificial$ trace, by scheduling the OmpSs tasks in dataflow manner.

The environment (Figure 4.3) consists of Mercurium based **code translator**, **MPISS tracer**, **tasks extractor**, Dimemas **replay simulator** and Paraver **visualization tool**. A **Mercurium based tool** translates the input OmpSs code into the sequential code with inserted functions that annotate pragma primitives. The obtained code executes sequentially. While tracing that execution, **MPISS tracer** emits to the trace additional user events that signal OmpSs primitives. From the collected trace, the **tasks extractor** reconstructs the trace of the potential OmpSs execution. Then, **Dimemas** replays the reconstructed trace to simulate the potential OmpSs execution. Finally, **Paraver** can visualize the simulated execution.

Note that simulating MPI/OmpSs codes differs only in tracing, where the environment traces the corresponding MPI execution, and then extracts OmpSs tasks. The input code is a correct MPI/OmpSs code. The application executes without OmpSs parallelism (as a pure MPI execution). Therefore, the *authentic* trace describes the corresponding MPI execution. On the other hand, offline processing of the trace extracts OmpSs parallelism and generates the final *artificial* trace that describes MPI/OmpSs execution. Finally, the *artificial* MPI/OmpSs trace is fed to Dimemas in order to reconstruct the potential parallel time-behavior. For reasons of simplicity, in the following subsections, we further describe the methodology steps in simulating OmpSs execution (without MPI parallelism).

Code translator

Our Mercurium based tool forces in-order execution of tasks (Figure 4.4). The translated code is a sequential code with empty functions annotating occurrences of OmpSs primitives. Thus, for each section of the execution, the introduced functions specify: 1) to which task that section belongs; and 2) what are input and output parameters of that section. The translated code is executed sequentially and traced with MPISS tracing library.

```
#pragma omp task input(A) output(B)
void compute(float *A, float *B) {
...
}

int main () {

   ...



   compute(a,b);

   ...

}
```

(a) input code

```

void compute(float *A, float *B) {
...
}

int main () {

   ...
   mpisstrace_start_task("compute");
   mpisstrace_input_par("A", a);
   mpisstrace_output_par("B", b);
   compute(a,b);
   mpisstrace_end_task();
   ...

}
```

(b) translated code

Figure 4.4: The code translation inserts functions that signal OmpSs pragma annotations.

Tracer (MPISS tracing library)

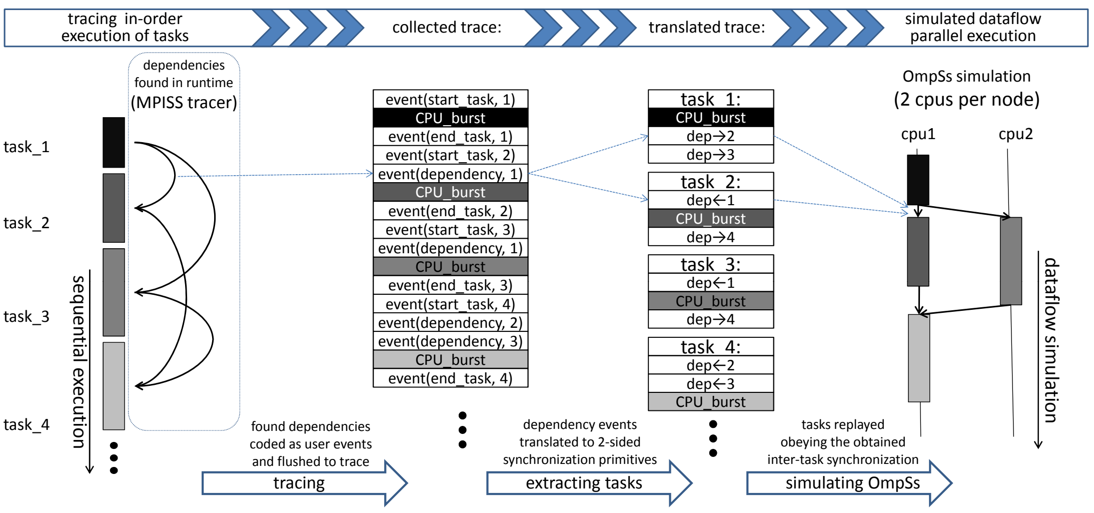
The most important feature of the tracer is to detect dependencies in run-time and mark them into the trace (Figure 4.5). In addition to the functionality of MPI tracing library, MPISS tracer intercepts the functions inserted in code translation. On intercepting *mpisstrace_start_task* (*mpisstrace_end_task*), the tracer emits to the trace an event to mark start (end) of a task. Thus, the collected trace also carries the information of which section of trace belongs to which task. More importantly, on intercepting function *mpisstrace_input_par* and *mpisstrace_output_par*, MPISS tracer calculates data-dependencies among the identified tasks and emits that information into the trace. Thus, the trace obtained with MPISS tracer specifies how is the sequential execution decomposed into OmpSs tasks and what are the data-dependencies among the instantiated tasks.

Tasks extractor (Trace translator)

From the collected trace, tasks extractor reconstructs the potential trace with task-based dataflow parallelism. For each task, tasks extractor crops from the sequential trace the segment that belongs to that task, and instantiates a new task in the OmpSs trace. Moreover, for each dependency user event from the sequential trace, task extractor creates a two-sided synchronization event in the reconstructed OmpSs trace (Figure 4.5). Thus, the trace derived with tasks extractor instantiates a separate trace entity for each task, defining the dependencies among the tasks using semantics that Dimemas simulator can process.

Replay simulator

We extended Dimemas in order to simulate MPI/OmpSs execution (Figure 4.5). We implemented a task synchronization as an intra-node instantaneous MPI transfer that specifies the source and the destination task. This implementation allows Paraver to visualize both MPI communications among processes and data dependencies among tasks. Furthermore, we implemented a new scheduling policy that optimizes MPI/OmpSs execution. The new scheduling policy optimizes parallel execution by favoring tasks that are on the critical path of execution. Furthermore, when simulating MPI/OmpSs execution, the scheduler assigns outstanding priority to tasks that contain MPI calls,


tracing in-order execution of tasks
collected trace:
translated trace:
simulated dataflow parallel execution
dependencies found in runtime (MPISS tracer)
task_1
task_2
task_3
task_4
sequential execution
event(start_task, 1)
CPU_burst
event(end_task, 1)
event(start_task, 2)
event(dependency, 1)
CPU_burst
event(end_task, 2)
event(start_task, 3)
event(dependency, 1)
CPU_burst
event(end_task, 3)
event(start_task, 4)
event(dependency, 2)
event(dependency, 3)
CPU_burst
event(end_task, 4)
task 1: CPU_burst dep→2 dep→3
task 2: dep←1 CPU_burst dep→4
task 3: dep←1 CPU_burst dep→4
task 4: dep←2 dep←3 CPU_burst
OmpSs simulation (2 cpus per node)
cpu1
cpu2
dataflow simulation
found dependencies coded as user events and flushed to trace
tracing
dependency events translated to 2-sided synchronization primitives
extracting tasks
tasks replayed obeying the obtained inter-task synchronization
simulating OmpSs

MPISS tracer intercepts functions inserted by code translation, detects dependencies among tasks and marks these dependencies in the trace (example: at the beginning of *task*_2 the tracer identifies dependency from *task*_1, and emits to the trace *event*(*dependency*, 1)).
Tasks extractor translates dependency user events into two sided synchronizations (example: *event*(*dependency*, 1) is translated into outgoing dependency ($dep \rightarrow 2$) in *task*_1 and incoming dependency ($dep \leftarrow 1$) in *task*_2).
Dimemas replays the tasks obeying the defined dependcies (example: due to events ($dep \rightarrow 2$) in *task*_1 and ($dep \leftarrow 1$) in *task*_2, *task*_2 cannot start before *task*_1 finishes).

Figure 4.5: Environment methodology

and allows these tasks to preempt regular (non-MPI) tasks. This latter scheduling feature significantly improves execution by prioritizing messaging tasks.

4.4 Tareador – Framework to identify potential dataflow parallelism

Dataflow programming models can extract very distant and irregular parallelism, a type of parallelism that a programmer himself can hardly identify. Tareador allows the programmer to start from a sequential application, and using a set of simple constructs propose some decomposition of the sequential code into tasks. Then, Tareador dynamically instruments the annotated code and in run-time detects actual data-dependencies among the proposed tasks. Furthermore, Tareador automatically estimates the potential parallelism of the proposed decomposition, providing to the programmer the data-dependency graph of the tasks, as well as the simulation of potential parallel time-behavior. In a similar manner, the programmer can start from an MPI application, propose some decomposition of MPI processes into OmpSs tasks and observe the parallelism of the potential MPI/OmpSs execution.

4.4.1 Implementation details

The idea of the framework is to: 1) run a sequential code with annotations that mark task the decomposition; 2) dynamically detect memory usage of each annotated task; 3) identify dependencies among all task instances; and 4) simulate the parallel execution of the annotated tasks. First, Tareador instruments all annotated tasks in the order of their instantiation. That way, the instrumentation can monitor accesses to all memory objects and thus identify data dependencies among tasks. Considering the detected dependencies, Tareador creates the dependency graph of all tasks, and finally, simulates the OmpSs execution. Moreover, Tareador can visualize the simulated time-behavior and offer deeper insight into the OmpSs execution.

The framework (Figure 4.6) takes the **input code** and passes it through the tool chain that consists of Mercurium based **code translator**, Valgrind based **tracer**, Dimemas **replay simulator** and Paraver and dependency graph **visualization tools**. Input code is a complete OmpSs code or a sequential code with only light annotations specifying the proposed *taskification*. A Mercurium based tool translates the input code in the sequential code with inserted functions annotating entry and exit from each task. The obtained code is compiled and executed sequentially. The Valgrind tracer dynamically instru-

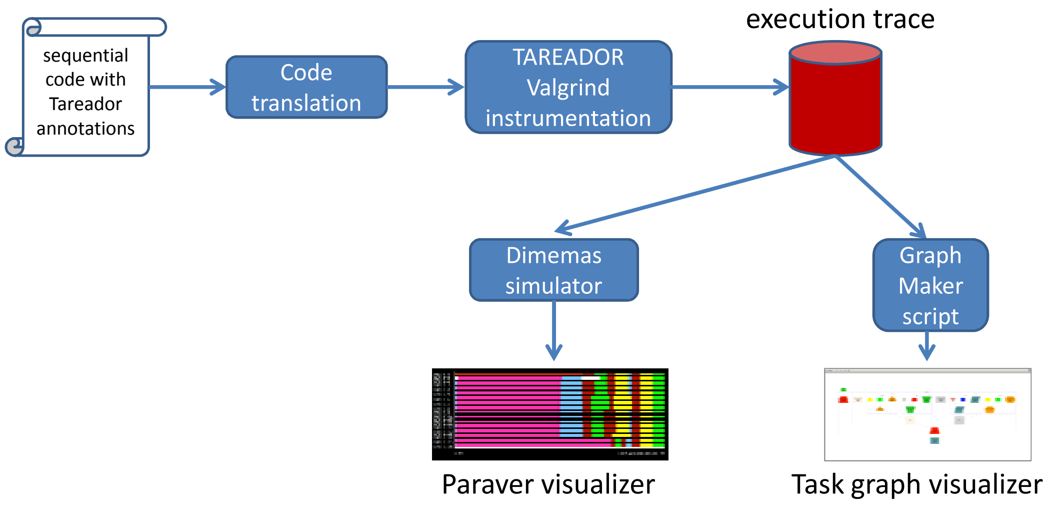


Figure 4.6: The Tareador framework integrates Mercurium code translator, Valgrind tracer, Dimemas simulator, Paraver and dependency graph visualization tool

ments this sequential execution. The tracer makes the *authentic* trace of the (actually executed) sequential execution, while at the same time, it reconstructs what would be the *artificial* trace of the (potential) OmpSs execution. Processing the obtained traces, Dimemas simulator reconstructs parallel time-behavior on a reconfigurable platform. Paraver can visualize the simulated time-behaviors and allow to profoundly study the differences between the (instrumented) sequential and the (corresponding simulated) OmpSs execution. Furthermore, simple offline processing translates the OmpSs trace into the tasks dependency graph that can be visualized through the Graphviz graph visualization environment [1].

Note that instrumenting an MPI code with annotated tasks, Tareador can reconstructs the potential parallel MPI/OmpSs execution. Input code is a complete MPI/OmpSs code or an MPI code with only light annotations specifying the proposed *taskification*. The translated code is executed in pure MPI fashion. Each MPI process runs on top of one instance of Valgrind virtual machine that implements a designed tracer. The tracer makes the *authentic* trace of the MPI execution, and the *artificial* trace of the (potential) MPI/OmpSs execution. Both traces are fed to Dimemas for the simulation of the parallel execution. For reasons of simplicity, in the following subsections, we further describe the methodology steps in instrumenting sequential applications and identifying the potential OmpSs parallelism (without MPI parallelism).

Input code

The input code can be an OmpSs code or a sequential code with light annotations that specify task decomposition. The input code has to specify which functions (parts of code) should be executed as tasks, but not the directionality of the function parameters. Thus, the input code can be a sequential code, only with annotations that specify some task decomposition. Figure 4.7 on the left shows an example of a sequential code with annotated *taskification* choice.

Code translator

Our Mercurium based tool translates the input code into the code with forced serialization of tasks. The obtained code is the sequential code with empty functions (*hooks*) annotating when the execution enters and exits from a task (Figure 4.7). The translated code is then compiled with a native sequential compiler, and the binary of the sequential execution is passed to the Valgrind tracer for further instrumentation. It is important to note that functions *tareador_start_task* and *tareador_end_task* may be injected directly into the input code. This feature allows very flexible specification of task decomposition, even in applications with very irregular code layout. Nesting of tasks is also supported.

Tracer

Leveraging Valgrind functionalities, the tracer instruments the execution and makes two Dimemas traces: the *authentic* trace describing the instrumented sequential execution; and the *artificial* trace describing the potential OmpSs execution. The tracer uses the following Valgrind functionalities: 1) intercepting the inserted *hooks* in order to track which task is currently being executed; 2) intercepting all memory allocations in order to maintain the pool of data objects in the memory; and 3) intercepting memory accesses in order to identify data dependencies among tasks. Using the obtained information, the tracer generates the *authentic* trace of the original (actually executed) sequential execution. Concurrently with that process, the tracer reconstructs the *artificial* trace of the potential (simulated) OmpSs execution.

The tool instruments accesses to all memory objects and derives data dependencies among tasks. By intercepting all dynamic allocations and releases of the memory

```
#pragma omp task
void compute(float *A, float *B) {
...
}

int main () {

   ...

   compute(a,b);

   ...

}
```

(a) input code

```

void compute(float *A, float *B) {
...
}

int main () {

   ...
   tareador_start_task("compute");
   compute(a,b);
   tareador_end_task();
   ...

}
```

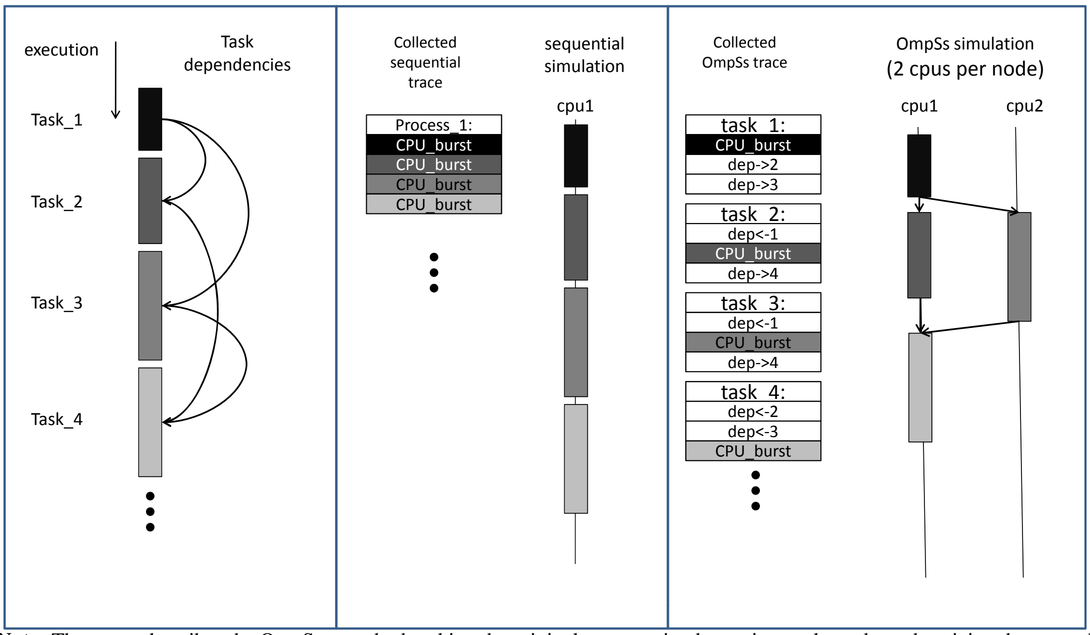
(b) translated code

Note: The input code must specify the entry/exit of each task. Thus, the input code does not have to be a complete OmpSs code, but rather a sequential code only with a specified proposed decomposition.

Figure 4.7: Translation of the input code required by the framework.

(*allocs* and *frees*), the tool maintains the pool of all dynamic memory objects. Similarly, by intercepting all static allocations and releases of the memory (*mmaps* and *munmaps*), and reading the debugging information of the executable, the tool maintains the pool of all the static memory objects. The tracer tracks all memory objects, intercepting and recoding accesses to them at the granularity of one byte. Based on these records, and knowing in which task the execution is at each moment, the tracer detects all read-after-write dependencies and interprets them as dependencies among tasks.

The tool creates the *authentic* trace of the executed sequential run and, considering identified task dependencies, creates the $artificial$ trace of the potential OmpSs run (Figure 4.8). When generating the original trace, the tool describes the actually executed run by putting in the trace the computation record stating the length of computation burst in terms of the number of instructions. Additionally, when reconstructing the trace of the potential OmpSs run, the tracer breaks the original computation bursts into tasks, and then synchronizes the created tasks according to the identified data dependencies.



Note: The tracer describes the OmpSs trace by breaking the original computation bursts into tasks and synchronizing the created tasks according to the identified data dependencies.

Figure 4.8: Collecting trace of the original sequential and the potential OmpSs execution.

4.4.2 Usage of Tareador

Based on Tareador, we design the top-to-bottom trial-and-error approach that can be used to port sequential applications to OmpSs. The approach requires no knowledge or understanding of the target code. The programmer starts by proposing a very coarse-grain task decomposition of the studied sequential code. Then, Tareador estimates the potential parallelism of the proposed decomposition and plots the potential parallel time-behavior. Based on this output, the programmer decides how to refine the decomposition to achieve higher parallelism. The programmer repeats these steps of proposing decomposition until finding a decomposition that exposes satisfactory parallelism. In Section 6.2, we illustrate this iterative process on the examples of sequential code of Cholesky and MPI code of HP Linpack. Furthermore, in Section 6.3 we describe an effort to automate the process of exploring potential decompositions of sequential codes.

Given the final decomposition, the programmer can use Tareador to get hints to complete the process of exposing OmpSs parallelism. Tareador detects for each parameter of the task whether it is used as *input*, *output* or *inout*. Moreover, Tareador warns the programmer about the objects that are accessed in tasks but not passed through the parameters list. Finally, Tareador can be used as a debugging tool – the programmer can run it on the already existing MPI/OmpSs code to automatically detect all the miss-uses of the memory. By doing all these checks with Tareador, the programmer can assure that an MPI/OmpSs code is correct from the point of view of OmpSs parallelization.

5 Overlapping communication and computation in MPI scientific applications

Overlapping communication and computation is believed to be a very promising technique to optimize MPI execution. As already mentioned in Section 3.1.2, overlap is the concurrency of computation and communication that results in hiding transfer delays. The overlap leads to two clear benefits: 1) overall speedup of the parallel execution; and 2) relaxation of network demand without the consequent degradation of the parallel performance. However, state-of-the-art techniques that increase overlap require serious refactoring of a target application and significantly reduce the maintainability of the code. Therefore, the goal of our research presented in this Chapter is to achieve higher overlap without any refactoring of the legacy code.

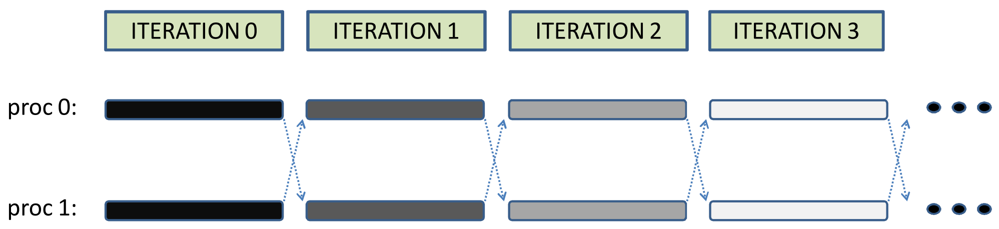
The rest of the Chapter is organized as follows. In Section 5.1, we present three characteristic behaviors in MPI bulk-synchronous programming. We show that these behaviors seriously suffer from lack of overlap. Section 5.2 introduces automatic overlap at the level of MPI calls. The technique increases overlap by breaking each message into independent chunks and maximizing the overlap of each chunk. In Section 5.3, we present speculative dataflow – a hardware assisted technique that achieves automatic overlap at the level of MPI calls without any need to restructure the original legacy code. We prove the feasibility of the technique and demonstrate that it improves execution of the three characteristic behaviors. Moreover, Section 5.4 quantifies the potential benefits of automatic overlap in real-world scientific applications. The study especially focuses on different patterns of production and consumption that seriously influence the overall potential of overlap.

5.1 Characteristic application behaviors

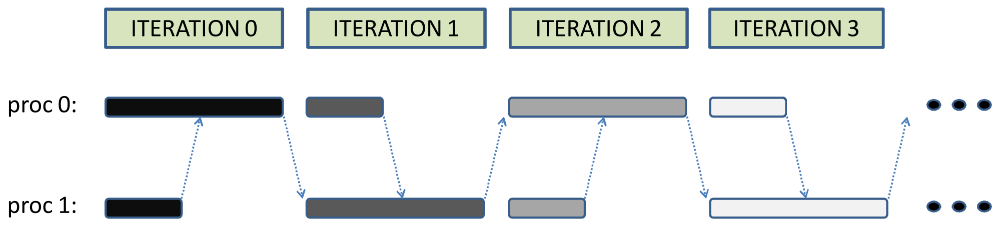
The mainstream methodology for MPI programming is bulk-synchronous programming that suffers from lack of overlap of communication and computation. Bulk synchronous programming tends to maximize the size of each MPI transfers and therefore reduce the overhead per sent byte ratio. We identify three characteristic application behaviors for applications written using bulk-synchronous programming. These three behaviors (Figure 5.1) express a significant lack of overlap that causes performance losses.

The first characteristic behavior is balanced execution (Figure 5.1a). In balanced execution, in each iteration, all MPI processes have the same amount of computation. During this computation phase, the network is completely idle. After the computation phase finishes, all MPI processes start communicating. During this communication phase, all the processors are idle, waiting for the communication to finish. Moreover, the second problem appears from the fact that all the processes demand network service at the same time. This type of "bursty" traffic causes a lot of network contention and puts a serious pressure on the interconnect. The length of the communication phase depends only on the technological parameters of the network, such as bandwidth and latency. Therefore, relaxing network constraints (decreasing bandwidth or increasing latency) directly increases the duration of the communication phase, and the overall length of the parallel execution.

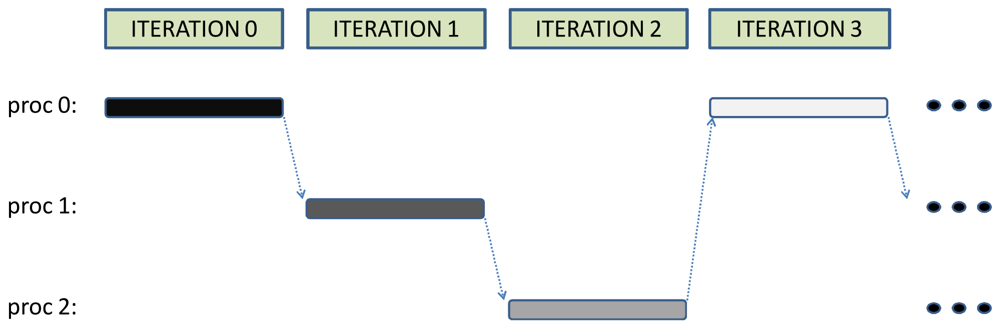
The second characteristic behavior is micro-imbalanced execution (Figure 5.1b).



(a) balanced execution



(b) micro-imbalanced execution



(c) pipeline execution

Figure 5.1: Three characteristic MPI behaviors that suffer from lack of overlap

Micro-imbalanced execution is globally balanced – at the level of the whole execution, all MPI processes compute for the same time. At the same time, micro-imbalanced execution is locally imbalanced – at the level of one iteration, different MPI processes compute for different time. Although the application is globally balanced, imbalance at the level of one iteration causes significant communication delays. Whenever the sending process has more computation than the corresponding receiving processes, the receiver stays idle until the sender finishes the computation. Moreover, this transfer is completely non-overlapped and causes communication delays.

The third characteristic behavior is pipeline execution (Figure 5.1c). In pipeline execution, the end of computation of one process provides data for the other process to start. As data is sent at the end of the computation, the second processor cannot start until the previous one finishes. Moreover, apart from waiting the sending process to finish the computation, receiving process must also wait for the complete transfer to arrive.

5.2 Automatic Communication-Computation Overlap at the MPI Level

Overlapping communication and computation at the MPI level consists of overlapping MPI transfers with the computation in which the data elements of these MPI transfers are produced and consumed. This can be achieved using the following four techniques.

- **Message chunking**: Each original MPI message is partitioned into independent *chunks* consisting of one or more data elements.
- **Advancing sends**: Each chunk is sent as soon as it is produced.
- **Double buffering**: Two different buffers are used to differentiate the chunks being consumed at the current iteration from the incoming chunks for consuming at the next iteration.
- **Postponing receptions**: Each chunk is waited at the moment when it is really needed for consumption.

Figure 5.2 shows the traditional case of non-overlapped MPI communications. Here, process A must wait until the MPI message that consist of four data elements is fully produced during the iteration i. Then, process A sends the MPI message to process B, so process B can execute its iteration $i + 1$. Thus, there is no overlap of the communication of the MPI message with any of the computation phases in the iterations i and $i+1$. Consequently, both processes suffer the corresponding communication delays.

Conversely, Figure 5.3 shows the case of using the four mentioned techniques to overlap communication and computation. Using automatic overlap, the application can overlap the communication of a chunk with the computation that produces succeeding chunks at the sender side and with the computation that consumes preceding chunks at the receivers side. For example, the communication delays of the chunk p_1 can be

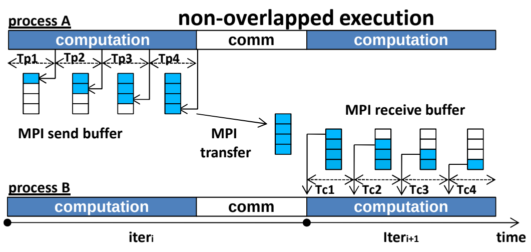


Figure 5.2: Non-overlapped execution

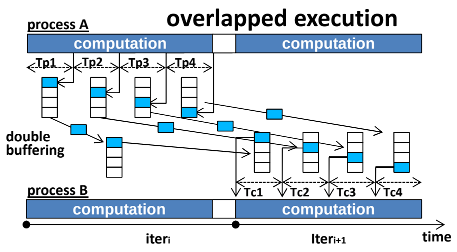


Figure 5.3: Overlapped execution

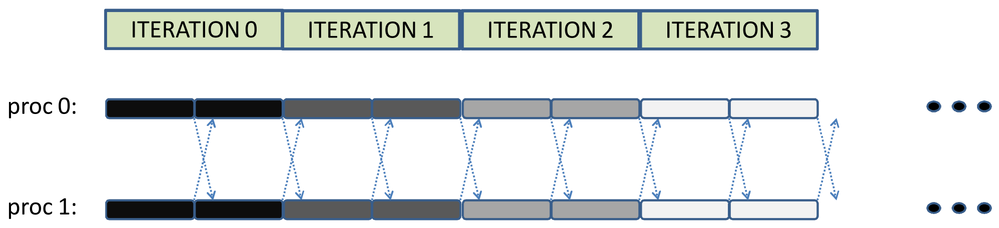
overlapped with the computational time to produce the following chunks p_2 and p_3 ($Tp_2 + Tp_3$) and also with the computational time to consume the chunk c_0 (Tc_0). In general, the transfer of a chunk i can be overlapped with the following computation times:

$$\sum_{i+1}^{n-1} Tp_j + \sum_{0}^{i-1} Tc_j, 0 \leq i \leq n-1 \tag{5.1}$$

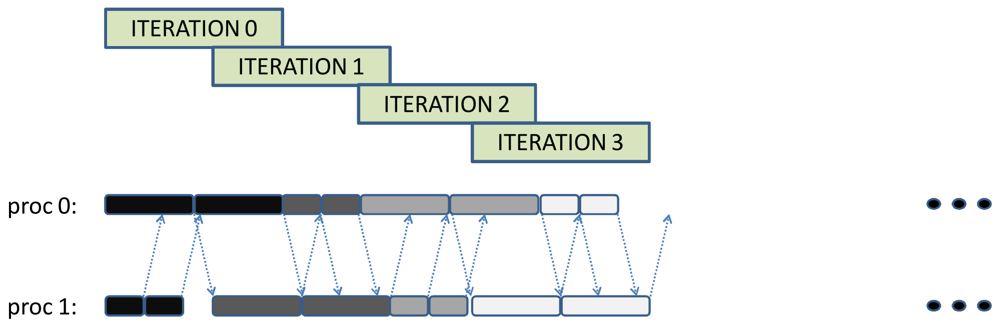
where Tp_j and Tc_j are the production and consumption time intervals to process the chunk j, and n is the total number of chunks in a MPI message.

Additionally, the double buffering technique is used to prevent overwriting of the communicated data at the receiver side. As illustrated in Figure 5.3, the chunk p_0 might arrive to process B during the iteration i, instead of at the next iteration $i+1$. Therefore, it can conflict with the previous value p_0 that is already there. The double buffering technique solves this anti-dependency by storing the message for the iteration $i + 1$ in a different buffer from the values used in the current iteration i.

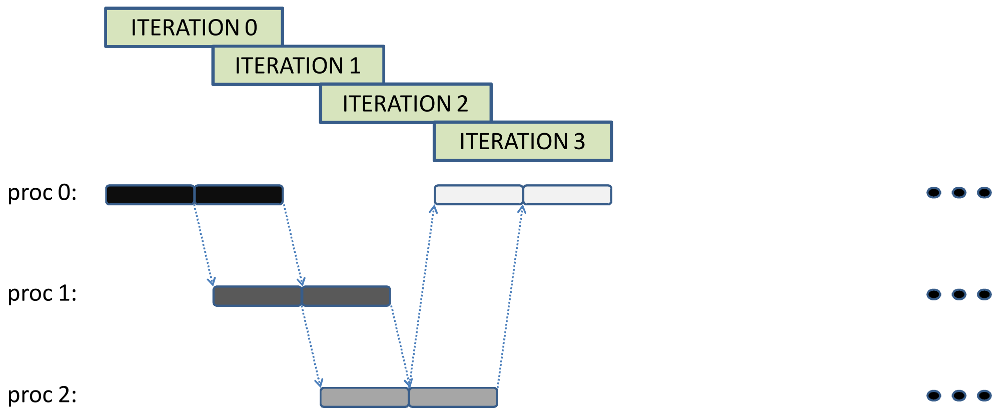
It is important to note that the equation above describes the ideal case where the computation time available to overlap the transfers is the highest possible for all chunks. However, an application can use a different production/consumption pattern that might be less favorable for overlap than the pattern above, and thus the total amount of computational time available for overlap may be drastically reduced. For example, if an application first consumes the last produced chunk, there is no computational time available to overlap this particular chunk. Even worse, if an application produces and consumes all chunks at the same time, there is no computational time available to overlap any of the chunks. The diversity of production/consumption patterns and their influence on the overlapping potential will be analyzed in detail by our simulation framework.



(a) balanced execution



(b) micro-imbalanced execution



(c) pipeline execution

Figure 5.4: Three characteristic behaviors with chunked overlap

5.2.1 Automatic overlap applied on the three characteristic behaviors

This Section illustrates the effect of automatic overlap at MPI level in the described characteristic MPI behaviors. For the sake of ease of illustrating, we assume that the overlapping technique breaks each original computation into two independent chunks. We also assume that the production/consumption patterns are *ideal*. Assuming an ideal pattern means that, after finishing $n\%$ of the iteration, the sender produces $n\%$ of the whole message. Similarly, on the receiver's side, after receiving $n\%$ of the original message, the receiver can process $n\%$ of the corresponding computation interval.

In balanced execution, automatic overlap can bring the maximal speedup of 2 and a significant relaxation of network constraints (Figure 5.4a). In the balanced execution without overlap (Figure 5.1a), the phases of computation and communication are disjunctive, and the total execution time is the sum of the two. With automatic overlap, each process sends a half of the message at the half of the iteration 0. Then the iteration 1 can start as soon as this half of the message arrives. Therefore, if the transfer time of a half of the message takes less than a half of the computation in one iteration, the parallel execution can proceed with no communication delays. Therefore, if in the non-overlapped balanced execution, the communication phase takes equal time as the computation phase, automatic overlap can hide all the communication delays and achieve the maximum speedup of $2x$. On the other hand, automatic overlap can also significantly relax the network constraints. As long as the transfer time of the original message takes less time than the computation in one iteration, automatic overlap can overlap this transfer time and hide all communication delays. Also, the traffic becomes less "bursty", since the automatic overlap with n chunks breaks one big transfer burst into n smaller bursts.

Automatic overlap can bring a significant execution speedup to the micro-imbalanced execution. In non-overlapped execution (Figure 5.1b), process 1 cannot start its iteration 1 until the process 0 finishes its iteration 0. Conversely, with automatic overlap (Figure 5.4b), process 1 can start its iteration 1 as soon as the process 0 finishes the first half of its iteration 0. This way, start of the iteration 1 of the process 1 overlaps with the end of the iteration 0 of the process 0. This overlap of iterations leads to smoother execution that has less communication delays and achieves better performance.

Finally, in pipeline executions, automatic overlap achieves better parallelization and leads to high speedup. In the presented non-overlapped pipeline execution (Figure 5.1c), process 1 can start its iteration 1 only after process 0 finishes its whole iteration 0. Thus, the iterations 0 and 1 are completely disjunctive. On the other hand, when using automatic overlap (Figure 5.4c), process 1 can start its iteration 1 after process 0 finishes only one half of its iteration 0. This way, iteration 1 partially overlaps with iteration 0. This overlap leads to better hiding of communication delays and the overall speedup of the applications. Theoretically, for p number of processes and the automatic overlap that breaks the original message into c chunks, the maximal speedup is $min\{p, c\}$.

5.3 Speculative Dataflow – A proposal to achieve automatic overlap

In this Section, we propose speculative dataflow – a speculative technique that implements automatic overlap. Speculative dataflow introduces mechanisms that achieve automatic overlap without restructuring the code of the application. The technique allows each MPI process to advance sends by speculatively sending parts of the original message. Furthermore, the technique allows each process to postpone receives by waiting parts of the incoming message only when these parts are needed. Finally, speculative dataflow must ensure correctness by guaranteeing that it can recover from miss-speculation.

Our main goal is to show that the protocol is feasible and that it is beneficial for the three targeted behaviors from Figure 5.1. Also, we discuss in more details the possible implementation of the mechanism, specially focusing on the recovery mechanism and the potential hardware support. Furthermore, in Section 5.4, we investigate in more depth the potential benefits of this mechanism in real-world scientific applications.

5.3.1 Protocol of speculative dataflow

The protocol uses speculation to advance partial sends. Speculative dataflow can advance sends using two mechanisms: 1) a mechanism to identify which buffer will be sent in the future; and 2) a mechanism to identify when some part of the message is produced and ready to be sent. In a general case, these two mechanisms cannot be both efficient and always correct. Therefore, our technique implements these mechanisms as speculative, allowing them to be aggressive, but providing that the errors can be recovered.

The protocol also postpones partial receives. Automatic overlap breaks the whole message into chunks and identifies the moment when each of the chunks is needed for consumption. Therefore, speculative dataflow can postpone partial receives by waiting for each part of the message to be received exactly when that part is needed for consumption.

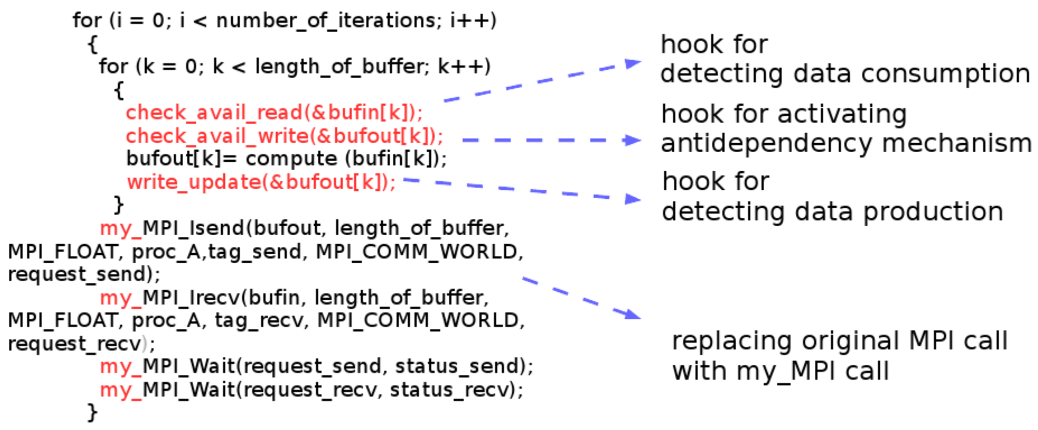
To complete the speculation protocol, the receiving process has to control speculation. Receiver is in speculative phase if it computes using the data that was sent speculatively, and still is unconfirmed by the sender. While in the speculative phase, the receiver produces temporal data, data that is not committed into memory. If the sender confirms the transfer, the receiver commits the temporal data to the memory. If instead of confirmation, the receiver gets notification of miss-speculation, the receiver activates its recovery mechanism and discards the temporal data.

Conditions and assumptions

In order to keep the complexity of the initial implementation low, we adopt a set of simplifying assumptions. Our speculation technique relies on the highly repetitive pattern of MPI codes [41]. For each intercepted MPI_Send call, we predict that this MPI call will repeat – that during the execution, the same buffer will be again sent to the same destination. We implemented the mechanism for detecting the end of a data chunk production assuming sequential order of data production. In other words, we consider that the production of some element of the buffer signifies that all previous elements in the buffer are already produced and ready to be sent. We model the recovery mechanism as a re-computation of the miss-speculated phase. In a future implementation of speculative dataflow, we expect to include the hardware mechanisms for checkpoint and recovery proposed in the scope of multi-threaded speculation [73].

5.3.2 Emulation

In order to gain experience, identify issues, and spot the possible problems, we designed an emulation framework for the proposed solution. This emulation environment consists of a modified MPI library and a set of probes used to intercept every memory access. These software hooks detect memory usage of the data that is involved in communication. To make the initial evaluation of the potential benefit of our proposal, we imported a set of benchmarks into the emulation framework and executed them on MareNostrum.



```
    for (i = 0; i < number_of_iterations; i++)
     {
      for (k = 0; k < length_of_buffer; k++)
       {
        check_avail_read(&bufin[k]);
        check_avail_write(&bufout[k]);
        bufout[k]= compute (bufin[k]);
        write_update(&bufout[k]);
       }
      my_MPI_Isend(bufout, length_of_buffer,
MPI_FLOAT, proc_A,tag_send, MPI_COMM_WORLD,
request_send);
      my_MPI_Irecv(bufin, length_of_buffer,
MPI_FLOAT, proc_A, tag_recv, MPI_COMM_WORLD,
request_recv);
      my_MPI_Wait(request_send, status_send);
      my_MPI_Wait(request_recv, status_recv);
     }
```

hook for
detecting data consumption
hook for activating
antidependency mechanism
hook for
detecting data production
replacing original MPI call
with my_MPI call

Figure 5.5: Software mock-up for the evaluation

The emulation environment consists of wrappers for MPI calls and manually inserted software hooks for instrumenting memory accesses. Code lines in black color belong to the original MPI code while lines in red represent the injected calls. The role of every new procedure is marked on Figure 5.5. Each MPI routine is intercepted and replaced with a new customized MPI routine that enforces speculative dataflow. Function *write_update* instruments message production and signals if the process finished producing some chunk of the message. Function *check_avail_read* instruments message consumption and signals if the process wants to access some chunk that is not received yet. Finally, function *check_avail_write* instruments message production and signals the anti-dependency hazard – the process wants to write into some chunk although that chunk is not sent in the previous iteration.

The described mock-up environment requires manual insertion of software hooks. Our target is to achieve the same functionality without modifying even the binary of the target application. Thus, we need a hardware mechanism that detects the production and consumption events and fires exception handling routines that drive our speculative protocol. In Section 5.3.3, we describe hardware support that can drastically reduce the overhead of our pure software solution.

Emulation results

We used the three characteristic MPI behaviors to test the potential of speculative dataflow. For each of the characteristic behaviors, we wrote a micro-benchmark and

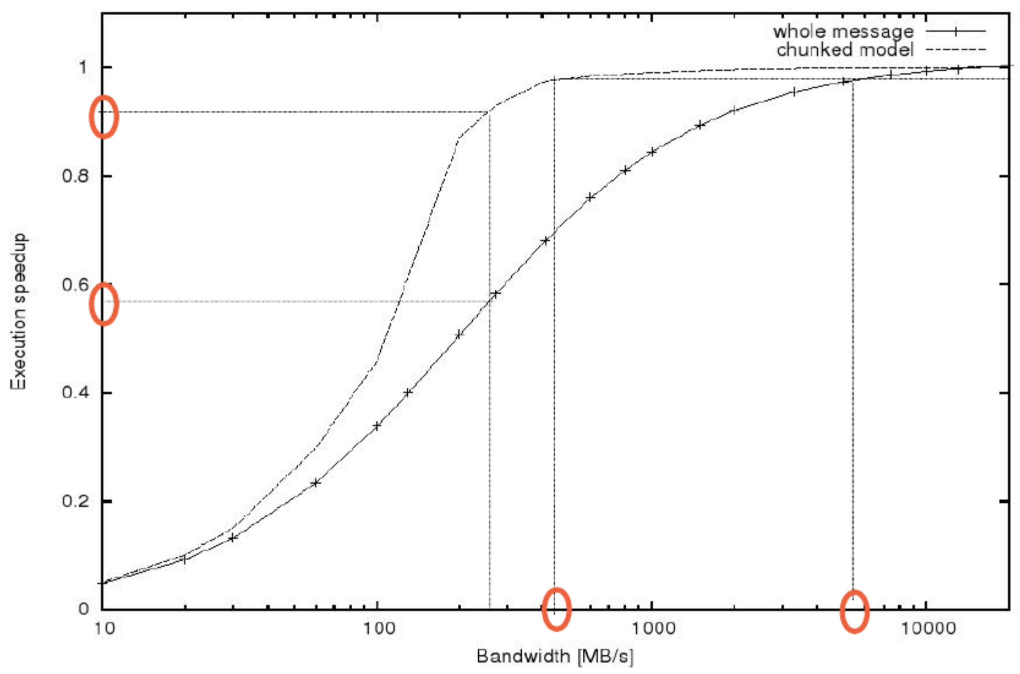


Figure 5.6: Influence of the network bandwidth on execution time

adapted it for instrumenting inside our emulation environment.

First we tested speculative dataflow on a balanced application with 8 processes. The application consists of computation and a two-way ring communication pattern. We execute the application with both non-overlapped and overlapped model. Then we processed the obtained traces using Dimemas simulator, to plot how execution time depends on the network bandwidth. Figure 5.6 shows the relative performance normalized to the performance of the non-overlapped execution with unlimited bandwidth. The results show that speculative dataflow can achieve significant speedup. For example, for the network bandwidth of $250MB/s$ (bandwidth of MareNostrum) speculative dataflow achieves the speedup of 1.72 over the non-overlapped execution. The results also show that speculative dataflow achieves significant bandwidth relaxation without consequent performance penalty. More specifically, to achieve the performance of the overlapped execution with a bandwidth of $450MB/s$, the non-overlapped execution requires a bandwidth of $4200MB/s$.

As mentioned above, speculative dataflow optimizes bandwidth usage by eliminating “bursty” traffic patterns. Figures 5.7 and 5.8 illustrate five iterations of non-

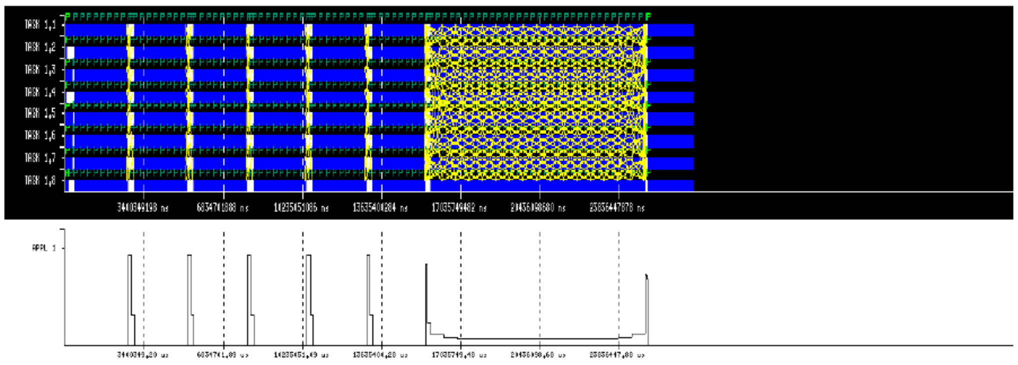


Figure 5.7: Bandwidth usage for the link bandwidth of 2.5GB/s

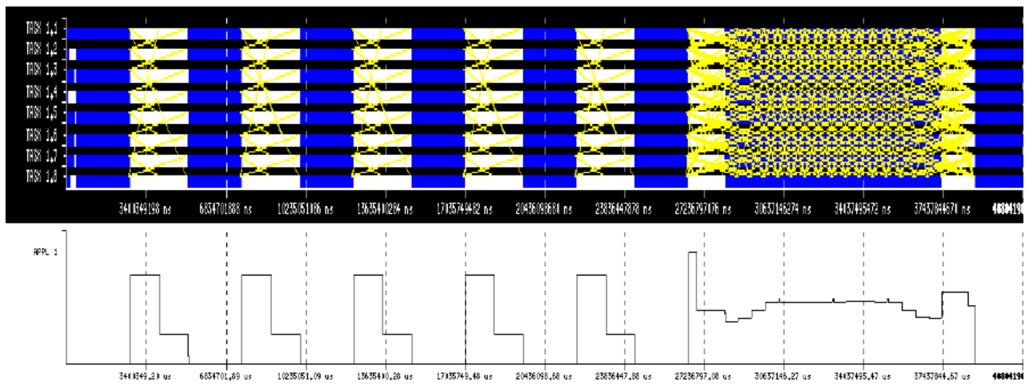


Figure 5.8: Bandwidth usage for the link bandwidth of 250MB/s

overlapped execution, followed by five iterations of overlapped execution. The upper pictures show execution timeline, where blue regions represent computational phases and white regions denote communication delays. The lower plots show the system bandwidth usage throughout the execution (the time scale is the same as for the upper plots). Figure 5.7 illustrates the execution with link bandwidth of $2.5GB/s$, while Figure 5.8 is for link bandwidth of $250MB/s$. It is interesting to note that speculative dataflow ”flattens“ bandwidth usage peaks, causing the reduction of needed bandwidth and the execution speedup.

The second experiment uses the same set of processes and communication pattern, but now modeling a micro-imbalanced application. Figure 5.9 illustrates 20 iterations of original execution followed by the same number of iterations in the “chunked mode”. The vertical line separates the two modes. White color represents processor stalls and other colors represent different iterations. The same color in two different

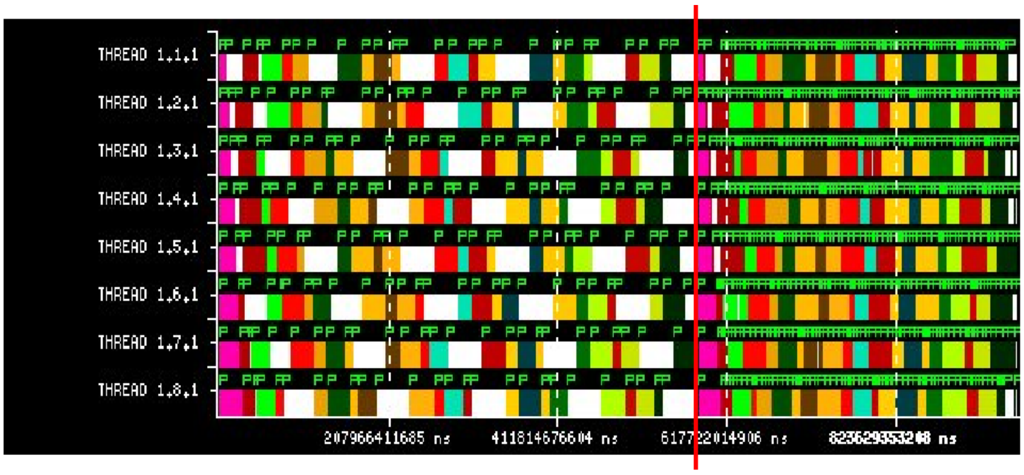


Figure 5.9: Trace of a micro-imbalanced application

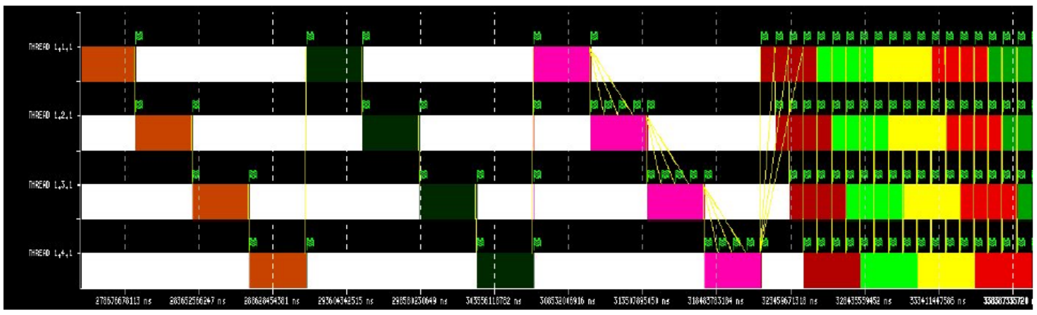


Figure 5.10: Overlapping in pipeline executions

processes indicates matching iterations. The results show that speculative dataflow reduces the time spent in stalls from 35% to 2.3%. Furthermore, in spite of added overhead, speculative dataflow achieves the speedup is 1.51.

Figure 5.10 illustrates the potential benefits of speculative dataflow in pipeline applications. The figure shows the execution of the non-overlapped run, followed by few iterations that include speculative dataflow. Again, white color represents stalls and the same colors represent matching iterations. Speculative dataflow eliminates almost all stalls from the non-overlapped execution and achieves the speedup of 3.32.

5.3.3 Hardware support

Profiling of the implementation indicated the functions for intercepting memory accesses to be the ones that represent the highest overhead. The emulation environment calls these functions for each memory access of the application, while the actual chunk transfer happens only few times per computation burst. In order to hide this overhead and try the proposed technique on legacy binaries, we propose a possible hardware device to take over the hooks' detection functionality. Hardware support that detects such events and raises a signal would avoid large fraction of the overhead and allow a very flexible implementation in the software signal handlers.

Hardware support should be the activation mechanism for the protocol of speculative dataflow. The hardware records the actual state of ongoing transfers, monitors the accesses to the data involved in communication and detects the events when some chunk should be sent/received. Upon detection of such event, the hardware stalls the processor and raises a signal. The protocol of speculative dataflow handles the signal, executes the necessary communication operations and updates the hardware records about the state of ongoing transfers. Upon termination of the signal handler, the host processor regains the control of execution. Similar hardware structures have already been proposed in the field of speculative multithreading [93].

The proposed hardware is similar to TLB table. The difference is that TLB table raises the signal if the address is not found in the table, while our table should trigger the signal when the match occurs. The other properties of these two matching mechanisms are the same. Hence, dualism with the already implemented mechanism of TLB proves the feasibility of our proposal.

5.3.4 Conclusions and future research directions

We showed that speculative dataflow could be an efficient technique to achieve automatic overlap at the level of MPI calls. We designed a speculative technique that can increase overlap in applications, without any refactoring of the target code. We especially explored how speculative dataflow would affect the three characteristic behaviors of bulk-synchronous programming. We proved that speculative dataflow can bring the two expected benefits of overlap: 1) execution speedup; and 2) bandwidth relaxation without consequent performance penalty.

Throughout our study of speculative dataflow, we assumed very regular application behaviors. To continue researching overlap, we must test these assumptions against realistic workloads. Therefore, in the following Section, we use the environment described in Section 4.2 to measure the potential overlap in a set of real scientific codes.

5.4 Quantifying the potential benefits of automatic overlap

State-of-the-art scientific applications achieve very little overlap. Most of techniques to increase overlap are based on code refactoring. Theses code optimizations are cumbersome and demand a lot of programming time. On the other hand, other techniques, such as speculative dataflow (Section 5.3), achieve overlap relying on additional hardware support. In both cases, it is very hard to anticipate how much would a real-world application benefit from these overlapping techniques. The anticipation of the potential benefits is very important, because the programmer can evaluate the potential of the technique and decide whether the implementation of the technique is worth the effort.

So far, evaluating the potential benefits of overlapping communication and computation has not been sufficiently addressed. There have been attempts to treat this issue [76], but they mostly rely on theoretical quantification of overlap, by trying to make representative mathematical models of the applications studied. However, these studies in their modeling of scientific codes tend to suppress some very important execution properties, and later omit their influence on the final result. More specifically, these studies assume that applications have the ideal computation patterns – that each MPI process linearly produces/consumes buffers that are involved in communication. Our study invalidates this assumption, and furthermore proves that computation patterns of the data that is communicated, decisively determines the potential overlap.

The major goal in this work is to offer to the community a fast and precise framework that estimates how much an MPI application can benefit from increased overlap. We designed a framework (Section 4.2) that takes a binary of the legacy MPI application and directly determines how much the application can profit from overlap. Our technique requires no intervention on the studied application. Using our framework, prior to any implementation effort, a programmer can instrument the targeted application and identify the potential of the planned optimizations. Moreover, the framework can increase the understanding of overlap by providing a visualization support that can compare non-overlapped and overlapped executions. All these features make the framework a very useful tool for studying overlap in the real scientific applications.

5.4.1 Experimental Setup

We conducted our measurements on two popular benchmarks and four well-known scientific applications. The choice of the application suite provides a wide range of different program behaviors to be studied. For all the codes we use their original version, as directly obtained from the distribution. The application pool consists of Sweep3D [4], POP [3], Alya [51], SPECFEM3D [22] and NAS [2] benchmarks BT and CG. Sweep3D is a wavefront application that solves a three-dimensional neutron transport problem. The problem size used is $50 \times 50 \times 50$ with *mk=10*. POP, Parallel Ocean Program, simulates oceans and their influence on climate. The input deck used is *test* with grid size of $192 \times 128 \times 20$. Alya is a multi-physics application that solves a variety of physics problems such as Convection-Diffusion-Reaction, Incompressible Flows, Compressible Flows, Turbulence, Bi-Phasic Flows and so on. We used the NASTIN module that solves the Incompressible Navier-Stokes. SPECFEM3D simulates earthquakes in complex three-dimensional geological models. The input deck used is *test* with 80 cells. And finally, BT and CG are two NAS parallel benchmarks. The problem used is class B.

Our test-bed system consist of 64 PowerPC 970 2.3 GHz processors interconnected with a Myrinet network that provides a unidirectional bandwidth of 250 MB/s. This basic system configuration corresponds to the Marenostrum supercomputer (as of 10^{th} of June of 2010).

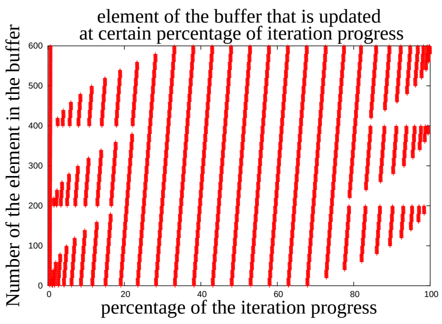
5.4.2 Patterns of production and consumption

For studying the potential of automatic overlap, it is very important to explore how are the transferred messages produced at the sender side and consumed at the receiver side. Each MPI program consists of computation bursts separated by MPI communication calls. A computation burst consumes the data received in the preceding MPI call and produces the data for the forthcoming MPI call. Ideally, if the production of the buffer to be sent is linear and uniform across the burst, the process can send the first half of the message half way through the computation burst. Similarly, if the consumption of the received buffer is linear and uniform across the burst, the process can postpone the blocking wait for the second half. Thus, computation patterns determine the potential for advancing sends and postponing receives and strongly influence the potential for overlap. Therefore, our first study focuses on characterizing patterns of consumption and production of the communicated data. To that end we developed an instrumentation tool based on Valgrind (Section 4.2.1). For each MPI transfer, the tool identifies the address of a communicated buffer and instruments all memory accesses to that buffer. This information allows us to determine when each element of a communicated buffer is produced or consumed.

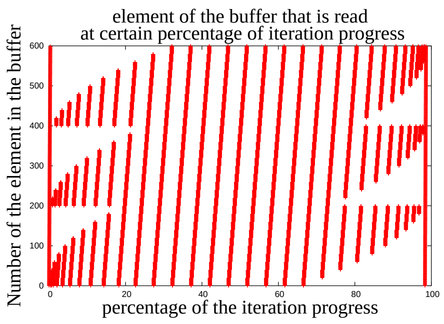
Results of tracing patterns

Due to complex computational algorithms, scientific applications tend to have diverse patterns of production and consumption (Figure 5.11). Figures on the left show production patterns, and figures on the right show consumption patterns. For each plot, the x axis represents the normalized time within the corresponding computation interval (from start to end), while the y axis represents an element's address within the accessed buffer. Points represent when each element was written for production patterns and read for consumption patterns. Many such patterns have been identified within each application.

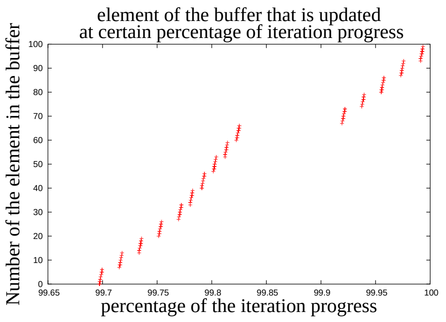
Figures 5.11a and 5.11b present the production and consumption patterns that are present in 50% of all transfers in Sweep3D. As presented on the y-axis, the communicated buffer has 600 elements and all of them are revisited and accessed many times during one computation phase. This causes very late production of the final version of the elements and decreases the potential for advancing partial sends. As shown in Figure 5.11a, the final version of the first element is produced at 66.3% of the production interval, while the final version of the first quarter of the message is produced at 94.8% of the production interval. Figure 5.11b indicates that the potential for postponing partial receives is even lower. This is because the application reads all the elements of the received message very early in the consumption phase. Compared to the ideal



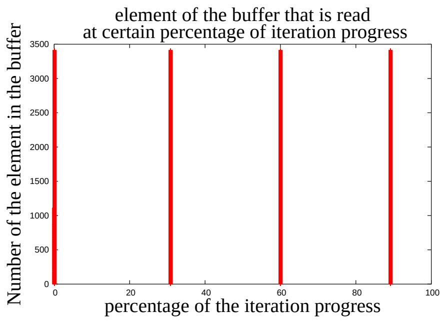
(a) SWEEP3D production pattern



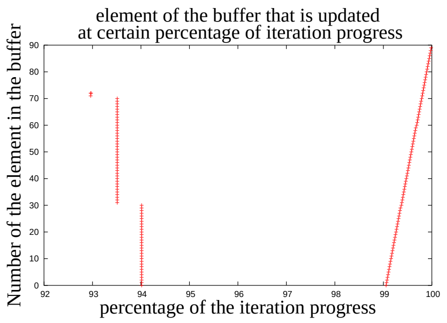
(b) SWEEP3D consumption pattern



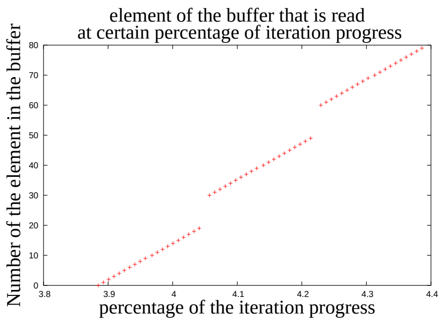
(c) NAS-BT production pattern



(d) NAS-BT consumption pattern



(e) POP production pattern



(f) POP consumption pattern

Figure 5.11: Production and consumption patterns of various applications

patterns, where with a received quarter of the message the program can execute 25% of the consumption interval, in Sweep3D a quarter of the message allows only 0.03% of the interval to be passed.

NAS-BT represents a separate class of applications with respect to the patterns of production and consumption. Figures 5.11c and 5.11d show that the communicated buffers are effectively accessed in a very short bursts of time. Figure 5.11d shows that all of the elements of the received buffer are loaded four times, each time in an extremely short interval, implying that the data is copied to some other location from where it is consumed during the computation. The production pattern (Figure 5.11c) suggests that the only activity in the buffer is packing of the message at the very end of the production phase (note the change in horizontal scale). As can be seen, in an application that uses out-of-place computation, the potential for advancing sends and postponing reception of independent chunks is negligible.

Some characteristic patterns of POP are presented in Figures 5.11e and 5.11f. The production pattern shows presence of data rewriting and late production of the final versions of the elements, starting from 99% of the production interval. Figure 5.11f shows that the received message is accessed only in a short portion of computation time, but allowing 3.9% of the interval to be passed before waiting for any part of the incoming message. Patterns like these provide low potential for automatic overlap, but still more than in the case of the applications that use buffer copying techniques.

Characterization of real patterns

Due to many different transfers in real codes it is very hard to come up with the concise characterization of the production/consumption patterns of an application. In order to give at least a rough estimate of the potential for advancing sends and postponing receptions, we provide the average values describing at what percentage of the computation phase progress certain portions of the message are produced and consumed. This data, presented in Table 5.1, can further be used to coarsely estimate the expected benefits of automatic overlap in the application. Since the critical transfers in Alya are of length of one element and no chunking is possible, only the data for producing the whole message and consuming the first element is presented.

Averaged patterns seem very unfavorable for overlap. The table on the left shows when in the production phase the final version of the message or its part is produced. Most of the applications produce the first element very late in the execution, many of them after 95% of the production interval, thus providing negligible potential for

	Percent of production phase needed to produce a part of message			
	1 elem	quarter	half	whole
ideal	0%	25%	50%	100%
NAS-BT	99.1%	99.37%	99.56%	99.98%
NAS-CG	3.98%	27.98%	51.99%	99.97%
Sweep3D	66.3%	94.8%	98.2%	99.8%
POP	95.5%	96.62%	97.75%	99.99%
Specfem	95.3%	96.48%	97.65%	99.99%
Alya				98.8%

(a) Potential for advancing sends

	Percent of consumption phase that can be passed upon reception of a part of message		
	nothing	quarter	half
ideal	0%	25%	50%
NAS-BT	13.68%	13.71%	13.74%
NAS-CG	2.175%	18.35%	34.53%
Sweep3D	0.02%	0.003%	0.004%
POP	3.525%	3.53%	3.534%
SPECFEM	0.032%	0.034%	0.036%
Alya	0.4%		

(b) Potential for postponing receptions

Table 5.1: Average patterns of production and consumption

advancing sends. Average patterns of consumption are even less favorable for overlap. For some applications, the reception of half of the incoming message allows less than 1% of the consumption phase to be executed. On the other hand, in NAS-BT, the averages indicate that a noticeable part (13%) of the consumption interval can be executed before waiting for any part of the incoming message, i.e., does not depend on the incoming message. This transfer independent computation provides significant potential for overlapping communication and computation.

Also, it is important to note that, due to many different transfers and their patterns, this averaging may also hide or misrepresent some of the results of interest. An example of this is NAS-BT, where in 63% of the consumption patterns, all the elements of the received message are loaded in the first 1% of the consumption interval. Still, due to averaging with the other 37% of the transfers, the average value shows that 13.68% of the work in the consumption phase can execute before the reception of any part of the message. For this reason, a study of all of the patterns independently in such cases may be beneficial.

The obtained results raise two issues. The first is that the measured patterns seem to be less beneficial than initially thought. The question is whether a code restructuring can rearrange computation pattern to make it similar to the ideal linear patterns (this issue is tackled in the following Section). The second question is that, even if the pattern may show some potential, it is important to be able to quantify the actual impact of the overlap on performance (Section 5.4.3 covers this issue). Furthermore, it is important anticipate the benefit that pattern linearization can achieve in terms of absolute performance. This information will be very useful to determine whether the restructuring effort is worthwhile.

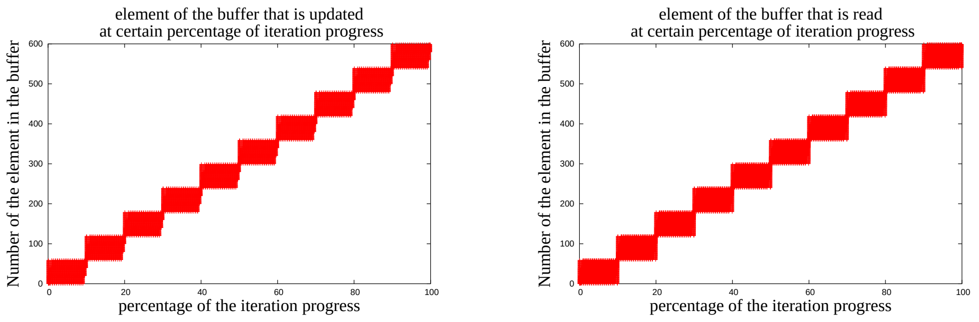
Achieving more profitable patterns

We propose code restructuring to change the applications' computational patterns. Due to complex algorithms and high level optimizations, restructuring a computational kernel in order to obtain sequential patterns is a difficult task. Therefore, our approach is to use a blocking technique that requires less knowledge about the computational kernel.

The idea is to make coarser granularity for the transfers than for computation, so the buffer could be transferred at once, while the computation with the buffer could be done by several independent computational iterations. Then, the computations accessing the same buffer are set in the desired order. This preserves the original computation patterns within the small computational iteration, while at the level of the whole buffer, the computations could be arranged in such way that the obtained patterns are ”quasi-linear“. Theoretically, increasing the ratio of the granularities of transfers and computations leads to patterns that are closer to the ideal.

Our code restructuring technique to obtain more favorable computation patterns is tested using Sweep3D. This choice of the application is made because Sweep3D has very interesting patterns, as shown in Figure 5.11. Furthermore, it represents a real scientific application and its non-trivial structure of more than 3500 lines of code proves the applicability of the approach in real scientific codes.

One iteration in Sweep3D consists of a reception of the buffer, a computation that consumes the received data and produces the buffer that will be sent, and a sending of the produced data. As in many scientific codes, one input parameter determines the granularity of the iteration, thus influencing the size of the buffer and the length of the computation. In the case of Sweep3D, this parameter is called *k-plane*. Our blocking technique converts this parameter into two independent parameters that define the granularity of the execution. One parameter represents the granularity of the communication, thus influencing the length of transferred messages. The second parameter denotes the granularity of the computation, hence specifying the length of workset of a computational iteration.



(a) production pattern

(b) consumption pattern

Figure 5.12: Production and consumption patterns of the restructured Sweep3D

Figure 5.12 represents the patterns for the case when the granularity of transfers is ten times coarser than the granularity of computations. Compared to the patterns of the original Sweep3D code presented in Figure 5.11, the new results show a significantly increased potential for advancing sends and postponing receives of the independent chunks. Now, the first final version of some of the elements is produced at 6.8% through the computation burst. When partitioning messages into 4 chunks, the final version of the first chunk can be produced and sent at around 30% through the computation interval. The consuming plots also show similar "quasi-sequential" pattern, hence allowing the receptions of the independent chunks to be delayed.

The code restructuring technique showed to be easily applicable in the case of Sweep3D. The proposed modifications affected less than 2% of the total code lines and required only a superficial knowledge of the application's algorithm. The obtained patterns seem much more favorable for overlap than the original patterns. However, in order to confirm the usefulness of this refactoring, we must prove that the refactoring increased the potential overlap in the application.

5.4.3 Simulating potential overlap

This Section presents the simulation results that evaluate the overlapping potential in scientific MPI applications. The simulated overlapping technique works at the MPI level, by automatically capturing all MPI messages, and trying to overlap these messages with useful computation of the application. The overlapping technique consists of the following mechanisms: message chunking, advancing sends, double buffering, and post-postponing receptions. These mechanisms are broadly described in Section 5.2. Our methodology takes into account a wide range of application behaviors, without requiring the knowledge of the application's algorithmic structure. Additionally, the results of our simulation can be visualized, thereby allowing qualitative comparison between non-overlapped and overlapped execution. Moreover, our framework allows us to simulate various network configurations and evaluate the impact of overlap on future networks.

For illustration purposes, Figure 5.13 presents Paraver visualization of overlap in NAS-CG executed with four MPI processes. The overlapped execution achieves 8% performance improvement with respect to the non-overlapped execution. Using Paraver visualization, we can easily investigate the cause of this improvement. Figure 5.13 shows that the overlapping technique can significantly advance partial sends, allowing to partially overlap chunk transfers with the production of the succeeding chunks. However, the overlapping technique cannot postpone partial receives. Thus, to increase the application's overlapping potential, the programmer should rearrange the consumption patterns and expose potential for postponing receives.

Simulation results

The results of our simulation show and compare execution of the same program with and without automatic overlap applied. The simulation of the overlapped execution is made assuming that each original message is partitioned into four chunks. Both Sancho's work [76] and our study show that finer-grain chunking brings no significant improvement compared to the granularity used in our evaluation. Furthermore, in order to study the influence of the computation pattern on the potential of overlap, we extract two different simulations of the overlapped execution. The first overlapped execution takes into account the real production/consumption patterns of the code, while the second assumes the ideal patterns. Reconfigurability of the Dimemas simulator is used to change parameters of the targeted system and to examine the influence of various network properties.

All the results are obtained for runs on 64 processors. The Dimemas configuration for a homogeneous distributed multiprocessor is selected for the target parallel ma-

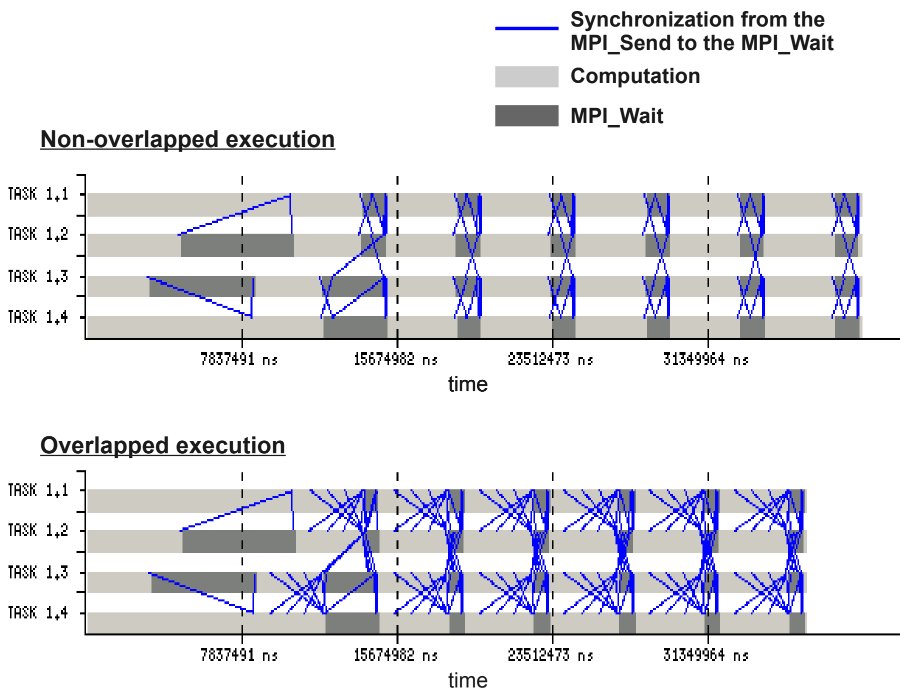


Figure 5.13: Paraver visualization for the non-overlapped and overlapped executions of NAS-CG.

chine. Every processor uses one input and one output port to connect to the network, which is modeled as an unlimited number of buses. That means that the network can support an unlimited number of simultaneous transfers and the only limitation comes from the processors' injection rate to the network.

Figure 5.14 shows the overlapping speedup both for real and ideal computation patterns. Due to different values of speedup in the applications, the plots for Sweep3D and SPECFEM3D are made on a different scale. Figure 5.14 shows that the overlap provides a speedup for a wide range of bandwidths. At high bandwidths, automatic overlap never causes performance degradation. On the other hand, Figure 5.14c demonstrates that in the range of very low bandwidths, automatic overlap can cause a slowdown. Also, Figures 5.14f and 5.14a show that the idealized patterns provide much higher overlap than the realistic ones. The following sections further discuss the results and reveal more about the mechanism of overlapping communication and computation.

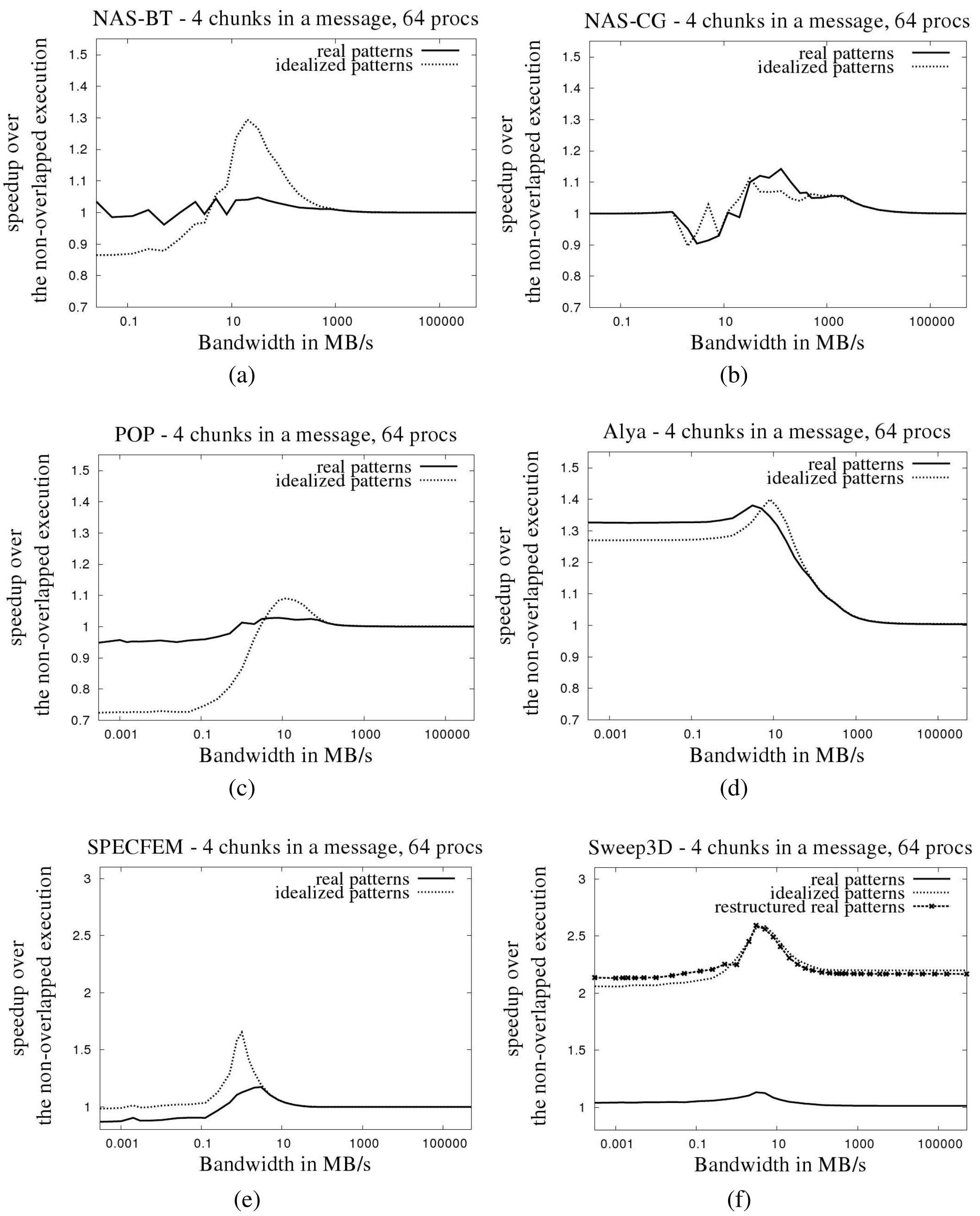


Figure 5.14: Speedup of overlapped execution over original execution

Sources of the potential overlap

The presented plots show three characteristic regions of system's bandwidth in which different sources of the potential overlap dominate the performance. For extremely high bandwidths, greater than one hundred times the bandwidth needed for the peak overlap speedup, time spent in computation dominates the total execution time. In these cases, automatic overlap establishes finer-grain communication and dependencies among processors, thus achieving a higher degree of overlap of computations. Advancing sends and postponing receives allows the destination processor to start before the source processor finishes its computation and sends the complete message, therefore allowing concurrency among these two computation phases of different processors. This source of the potential overlap is especially expressed in applications with microscopic imbalance of computation. In these applications, finer-grain transfers and dependencies lead to significant hiding of communication delays caused by the imbalance at the level of one iteration.

In the range of low bandwidths, lower than one hundred times the bandwidth needed for the peak overlap speedup, execution time is dominated by the time spent in communication. The mechanism of automatic overlap breaks all the messages into chunks, thus allowing higher concurrency of the created fine-grain partial transfers to be achieved. By postponing receives, the destination processor can start its iteration before the complete message is received. Moreover, the receiver can start advancing its partial sends, therefore creating concurrency among the sending and receiving transfers. This source of the potential overlap is especially expressed in applications in which at the level of one iteration processes communicate different amounts of data, while at the level of the whole application the amount of the communicated data is similar for all of the processes. We will refer to this type of codes as codes with microscopic imbalanced communication.

For intermediate bandwidths, in which the time spent in message transferring is comparable to the time spent in computation, the dominant source of the potential overlap lies in hiding the in-flight time of transfers. Automatic overlap increases the time gap between the initialization of the transfer and the wait for its completion, thus relaxing the time constraints for the transfer's completion. Therefore, automatic overlap increases the portion of the computation that can be used to overlap and hide communication stalls. Since the mechanism is based on hiding the transfer time by literally overlapping it with computation, it provides a maximal speedup of two. This source of overlap is present in all types of codes and it is especially expressed in the case of balanced execution.

Figure 5.14 shows that all the applications experience the peak value of the speedup in the middle range of bandwidths, since that source of the overlap is present in all applications. In addition, due to its pipeline behavior, Sweep3D shows a significant speedup on both high and low ranges of bandwidths (Figure 5.14f). This is because the pipeline application could be considered as an execution that expresses microscopic imbalance of both computation and communication.

The proposed technique of automatic overlap is based on the mechanism for advancing sends and postponing receptions of the independent chunks. A linear communication model would guarantee that the advanced send arrives earlier to the destination, thus providing better performance of the overlapped execution for any network bandwidth. However, due to a limited network resources, the simulation allows the appearance of non-linear effects, such as network contention. At extremely high network bandwidths, contention of the network is very unlikely to happen, thus practically eliminating the non-linear effects from the Dimemas communication model. Consequently, in these ranges, overlapped execution always performs better than the original execution, as evidenced by Figure 5.14. However, for low bandwidths, non-linear effects are more likely to occur. The partial sends cause different serialization of the messages in the shared network, therefore not guaranteeing any more that a chunk that is sent in advance will arrive earlier to its destination. This behavior can cause performance degradation, as evidenced in Figures 5.14a and 5.14c. Fortunately, this behavior is significant only in the range of very low network bandwidths. In state-of-the-art networks, the bandwidth is high enough to mitigate this effect.

Reduction of bandwidth requirements

Figure 5.15 compares the absolute execution time for both the non-overlapped and the two overlapped runs. The results for SPECFEM (Figure 5.15a) show that the overlapped execution is much more tolerant to bandwidth reduction. Thus, going from the range of very high to the range of low bandwidths, the non-overlapped execution loses performance much faster than the overlapped execution. This delayed drop of performance is the source of the peak speedup that appears at the middle range of bandwidths, as evidenced by Figure 5.14e. The results also show that sometimes the non-overlapped execution on extremely high bandwidths, cannot reach the perfor-

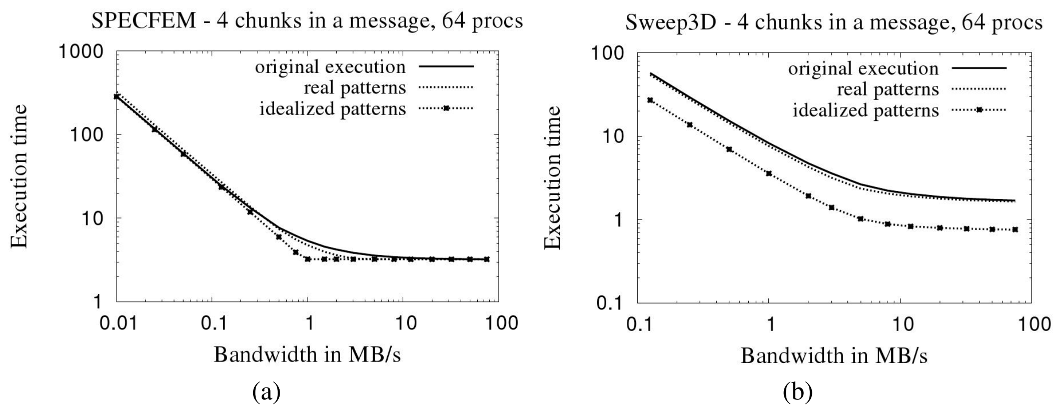
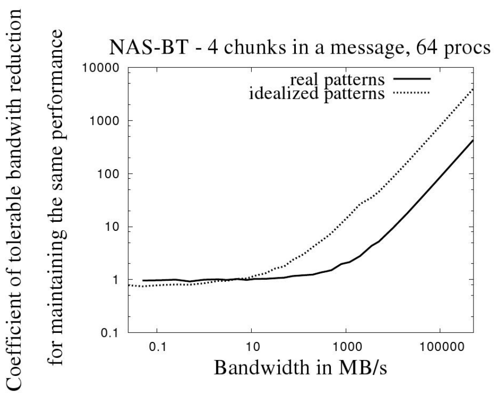


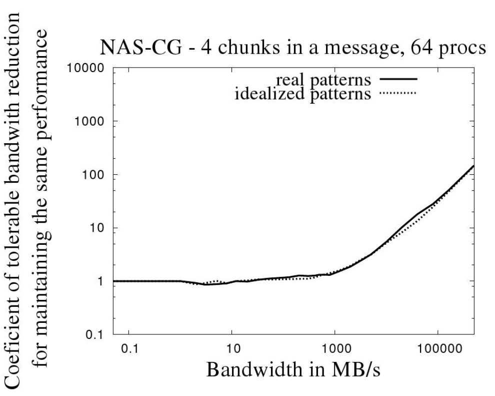
Figure 5.15: Execution time for original and overlapped (real and ideal patterns) execution

mance of the overlapped run on moderate bandwidths. For example, in the case of Sweep3D (Figure 5.15b), performance of the overlapped execution with ideal patterns on bandwidth of 5 MB/s, cannot be achieved by non-overlapped execution even with the infinite bandwidth.

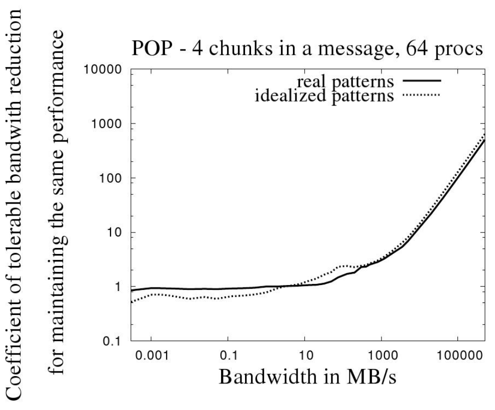
Figure 5.16 shows the factor of bandwidth reduction that would still allow the overlapped execution to have the same performance as the original execution that uses the full bandwidth. The plot is obtained by measuring the bandwidths at which the original and the overlapped execution deliver the same performance. Then the x axis marks the bandwidth of the original execution and the y axis shows how many times lower bandwidth can be used with the overlapped execution to achieve the same performance. For example, the performance of the non-overlapped Sweep3D at $1GB/s$ could be achieved with overlapped execution with the bandwidth which is 11.7 times lower when the real patterns of production and consumption are considered. On the other hand, the overlapped execution needs 142 times lower bandwidth if the ideal patterns are assumed. The value marked on y axis is referred to as the coefficient of tolerable bandwidth reduction because it describes how much the bandwidth can be reduced when applying the automatic overlap, in order to preserve the performance of the original execution. Overall, the overlapped run requires many times lower bandwidth for achieving the same performance as the original run on the state-of-art bandwidths. Also, the constantly increasing plot in all of the figures proves that in the range of high bandwidths the overlapped run always has better performance than the original run.



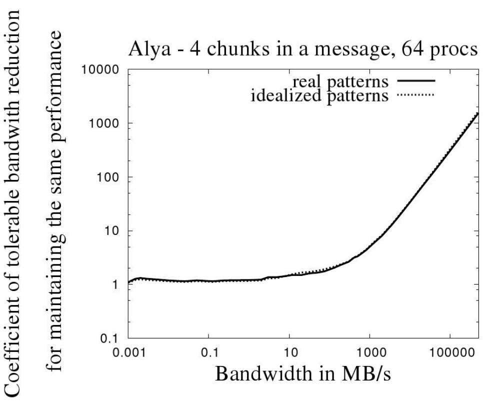
(a)



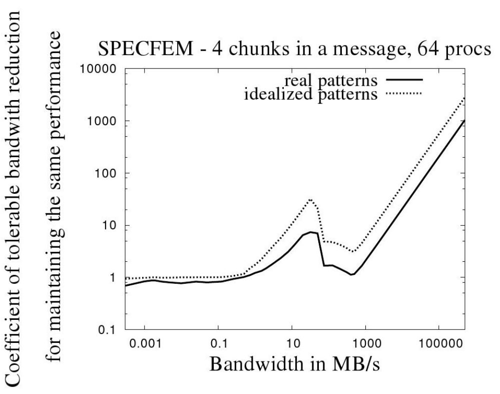
(b)



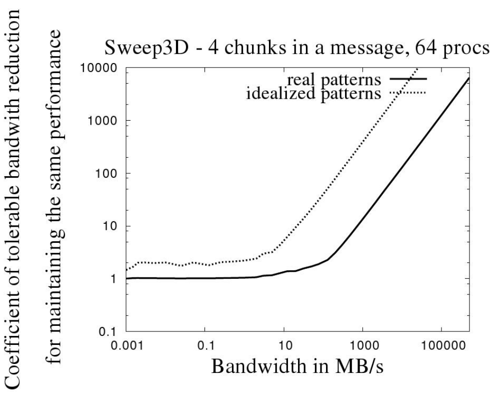
(c)



(d)



(e)



(f)

Figure 5.16: Factor of bandwidth reduction for which the overlapped execution maintains the performance of the original execution on full bandwidth

Correlation of the potential overlap with the patterns of production and consumption

Figure 5.14 shows that there is a significant performance difference in the runs that have real computation patterns and the runs which assume the ideal patterns. In order to study these differences and to determine the influence of the computation patterns on the speedup induced by overlap, we revisit Table 5.1.

Computation patterns determine the applications potential for overlap. As mentioned, NAS-BT has a very late production of chunks in all the computation bursts and a very early consumption in 63% of the bursts. Thus, the real patterns are unfavorable for overlap, providing much lower speedup than in the case of the ideal patterns. On the other hand, POP has no potential for advancing sends in 75% of the bursts. However, POP has at least 3% of the computation phase before any part of the message is consumed in all of the consumption intervals, thus giving a maximum speedup of about 3% for real patterns and a higher speedup when the patterns are idealized.

However, profitable patterns cannot guarantee high potential overlap. NAS-CG has profitable patterns, but still the speedup obtained by overlap is quite low, for both the real and the assumed ideal patterns. This is because the application contains computation bursts of very different lengths, causing the potential for advancing sends and postponing receives in extremely short bursts to be very low in terms of absolute time. Consequently, in these short bursts, the time constraints for the transfers completion are kept fixed, thereby presenting strong synchronization points and causing a lot of communication delays. Also, it is interesting to note that these short computation intervals make the code sensitive to changes of the serialization of chunks, as evidenced by the ”jagged“ plot in Figure 5.14b.

Sometimes, the speedup achieved by automatic overlap is attributed to changed ordering of messages. In Alya, the speedup in the low range of bandwidths comes from breaking collectives into point-to-point communications. Also, a small peak for the middle range of bandwidths is due to a small overlap of computation and transferring time. Furthermore, the overlap in SPECFEM provides higher speedup than what can be predicted considering the data from Table 5.1. This is because the processes that in one computation burst send more messages have earlier production of chunks than the processes that send fewer messages, thus favoring processes with more intensive

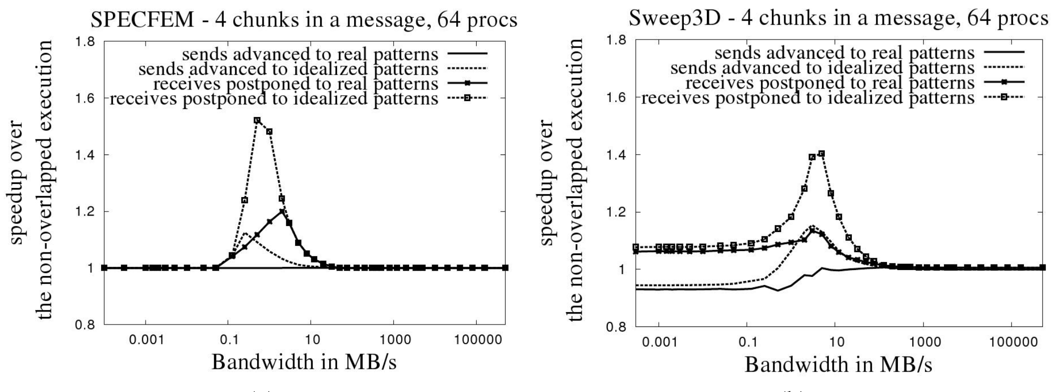


Figure 5.17: Benefit of only advancing sends or only postponing receives

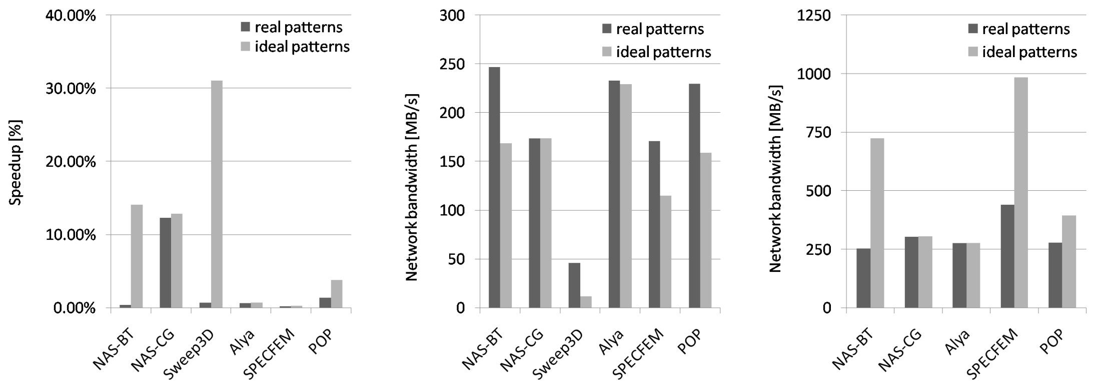
communication. For the idealized patterns, due to significantly faster production and slower consumption of chunks the speedup is much higher.

Figure 5.14f also illustrates the effect of the code restructuring for pattern linearization applied to Sweep3D. The real patterns provide little potential for advancing sends and postponing receives, thus offering very low overlapping speedup. On the other hand, the ideal patterns provide high speedup in all ranges of bandwidth. Figure 5.14f also shows the speedup achieved after the code restructuring for pattern linearization. The obtained ”quasi-linear“ patterns provide almost as much potential for the overlap as the ideal patterns.

Figure 5.17 shows the potential benefits of overlap if only advancing send or only postponing receives is used. Figure 5.17a shows that applying only one of the mechanisms drastically decreases the overlapping potential. Furthermore, as explained above, sources of overlap in the ranges of low and high bandwidths achieve speedup only when the technique both advances sends and postpones receives. Since this is not the case here, the obtained speedup in these ranges is very low, as evidenced by Figure 5.17b.

MareNostrum case study

This Section estimates the potential benefits of overlap in scientific applications executed on MareNostrum parallel platform. For that purpose, target machine configuration in Dimemas is set to faithfully model MareNostrum. Table 5.2 shows the number



(a) Overlapping speedup
Speedup of the overlapped execution
over the non-overlapped execution

(b) Relaxation of network bandwidth:
Network bandwidth at which
the overlapped execution
gives the same performance as
the non-overlapped at 250MB/s

(c) Equivalent bandwidth advancement:
Network bandwidth at which
the non-overlapped execution
gives the same performance as
the overlapped at 250MB/s

Note: In Figure (c), the equivalent bandwidth improvement for Sweep3D for both real and ideal patterns tends to infinity, and therefore it is not shown.

Figure 5.18: Simulation of the overlapped executions on the real and ideal production/consumption patterns.

of buses used in our experiments for each application. The number of buses is determined empirically, by comparing Dimemas simulation with the real execution on MareNostrum parallel computer.

Table 5.2: Number of network buses used in Dimemas for each application.

Sweep3D	POP	Alya	SPECFEM3D	BT	CG
12	12	11	8	22	6

Overlap provides a small speedup for the real patterns and a decent speedup for the ideal patterns (Figure 5.18(a)). The unfavorable production/consumption patterns seriously limit the applications' overlapping potential. As anticipated from Table 5.1, the real patterns allow speedup only in the case of NAS-CG. On the other hand, for modeled ideal patterns, some applications achieve a significant speedup. The highest speedup is reached for Sweep3D due to the wavefront behavior of the application. For this type of applications (concurrent, pipeline), the chunking mechanisms of overlap causes finer-grain dependencies among processes and potentially increases parallelism among the processes.

The biggest benefit of overlap is that it allows to significantly relax network bandwidth without consequently degrading the performance. Figure 5.18(b) shows that in order to achieve the performance of the non-overlapped execution on $250MB/s$, the overlapped execution needs much less bandwidth. Again, Sweep3D benefits from overlap the most and allows to reduce the network bandwidth to $11.75MB/s$ and maintain the performance of the original execution. Relaxation of network bandwidth is very important because it means that in order to achieve the performance of the original execution on a state-of-the-art network, the overlapped execution requires a much cheaper network.

Finally, the benefits achieved by applying automatic overlap sometimes cannot be reached by simply increasing the network bandwidth. Figure 5.18(c) shows the bandwidth required by the non-overlapped execution in order to achieve the performance of the overlapped execution at $250MB/s$. In other words, it presents what is the overlap's equivalent in increased network bandwidth. The result of Sweep3D shows that for some applications, the performance of the overlapped execution cannot be achieved with non-overlapped execution on any bandwidth. Also, it is interesting to note that overlap brings little speedup in SPECFEM3D (Figure 5.18(a)), but the benefits achieved by overlap are equivalent to benefits that could be achieved by increasing the network bandwidth almost four times.

5.4.4 Conclusions and future research directions

Our study confirms that overlapping communication and computation is a very promising solution for achieving more efficient communication. We show that overlap brings significant execution improvements, especially in the case of favorable production/-consumption patterns. We showed that real scientific applications have diverse computation patterns that are often unfavorable for overlap. We confirm that for favorable patterns, overlap achieves two benefits: execution speedup and relaxation of bandwidth without consequent degradation of performance. Moreover, we show that in applications with micro-imbalance of computation, the benefits achieved by overlap cannot be reached by simple increasing network bandwidth.

Our study can be useful for researchers in the field to understand better the potential and the mechanism of overlap. We designed a simulation framework that can simulate applications' overlapped execution automatically, without the need to know or change the application's legacy code. Compared to the previous studies in the field, our framework accounts for a wider range of application properties and allows to study overlap on diverse network configurations. Finally, the framework provides useful visualization of the simulated time behaviors, so we can qualitatively compare the non-overlapped and the overlapped execution.

Also, our environment can be very useful for a programmer that intends to increase the overlap in his application. The programmer can use our environment to anticipate the potential impact of overlapping technique that he wants to implement. Furthermore, the programmer can visually inspect the potential overlapped execution and conclude how to customize the planed implementation in order to increase the overlap.

The results of this study showed us that the overlap at the level of MPI calls is very limited by the application's production/consumption patterns. Unfavorable computation patterns decrease the potential for advancing sends and postponing receives, thus limiting the potential for automatic overlap. We also showed that code refactoring can rearrange these patterns in order to increase the potential overlap. However, this code refactoring requires a deep understanding of the targeted code, so applying it on many applications is unfeasible. Therefore, we believe that it is of major importance to find a way to rearrange computation patterns without code refactoring. The following Chapter explores OmpSs programming model. OmpSs introduces dynamic task scheduling that can rearrange computation patterns into ones that are more profitable for overlap.

6
Task-based dataflow parallelism

Our research of automatic overlap showed that there is a need for a mechanism to change the internal computation pattern by which each MPI process locally computes on the data that is involved in communication. We also illustrated that manual code restructuring can obtain patterns that are more favorable for overlap. However, this manual code restructuring requires significant programming effort. Thus, we want to explore the potential of hybrid programming models that integrate MPI with some shared memory parallel programming model. The shared memory programming model should rearrange the execution inside each MPI process and dynamically obtain computation patterns more suitable for overlap.

Our choice is to explore the integration of MPI and OmpSs programming model, a programming model developed in BSC. OmpSs is a dataflow task-based programming model that can execute tasks out-of-order, as long as data dependencies are satisfied. In addition, OmpSs integrates with MPI into a hybrid MPI/OmpSs programming model in which the workload is parallelized across distributed address spaces using MPI, while the workload of each MPI process is parallelized using OmpSs. This integration leads to higher performance, as well as to execution more tolerant to network contention and OS noise [63]. More detailed description of the MPI/OmpSs programming model can be found in Sections 2.2.3 and 2.2.4. Furthermore, the potential for overlap in MPI/OmpSs execution is further explained in Section 3.3.1.

In this Chapter, we explore two approaches for tuning MPI/OmpSs parallel execution – first by optimizing some sections of the code, and second by changing the task decomposition of the code. The rest of this Chapter is organized as follows. In Section 6.1, we explore the mechanisms to find the critical code sections in MPI/OmpSs applications. Furthermore, Section 6.2 describes how using Tareador, a programmer can iteratively explore the potential task decompositions for a given application. Finally, in Section 6.3, we present the autonomous driver that iteratively runs Tareador to find the optimal task decomposition of an application.

6.1 Identifying critical code sections in dataflow parallel execution

Task-based dataflow programming models showed to be very powerful in exposing high level of parallelism. Dataflow can extract very irregular parallelism. Also, dataflow introduces significant asynchronism in the execution, providing performance that is more tolerant to external contention and OS noise [63]. Due to all these potential benefits, dataflow became the parallelization strategy for various programming models [69][40][53], as well as for application specific backends [83].

However, due to irregularity of dataflow parallelism, it is very hard to anticipate and control the parallelism exposed in the application. By varying granularity of execution or task decomposition of the code, the programmer generates different number of tasks, thus changing the potential parallelism of the execution [84]. If there is too few tasks, the application may expose insufficient parallelism to keep the target machine utilized. However, if there is too many tasks, the runtime overhead may seriously harm the performance [63]. Thus, to optimize the performance, the programmer must control the parallelism released in the application and suit it for the parallelism offered by the target machine.

Moreover, to fine-tune the execution of his application, the programmer may try to optimize some section of the code, either by rewriting the code of that section, or by executing that section on an accelerator. The years of practice in optimizing applications points that the major issue is focus – identifying the code section whose optimization would yield the highest overall applications speedup. In other words, prior to any optimization effort, the programmer must identify **the critical section** – the code section whose optimization would bring the highest reduction of the total execution time.

To address this issue, we designed an environment that automatically estimates the potential parallelism of an application and furthermore identifies critical code sections. The programmer can use this environment to estimate the potential optimization speedup. This is very important, because it allows the programmer to anticipate the benefits of the intended optimization and decide in advance whether the optimization is worth the effort. We show that in many applications, the choice of the critical section of the code decisively depends on the configuration of the target machine. For instance, in HP Linpack, optimizing a task that takes 0.49% of the total computation time yields the overall speedup of less than 0.25% on one machine, and at the same time, yields the overall speedup of more than 24% on a different machine.

6.1.1 Motivation

In this Section, we present a simple OmpSs application (Figure 6.1) and explore the potential benefits of optimizing different sections of the code. There are three functions ($comp1 - comp3$) of equal duration. These functions are encapsulated into tasks. The code consists of one loop that calls each of the functions in every iteration. Depending on the loop index, each function selects on which buffers to compute. The only difference among the functions is the step ($step1 - step3$) based on which they determine which buffers to use.

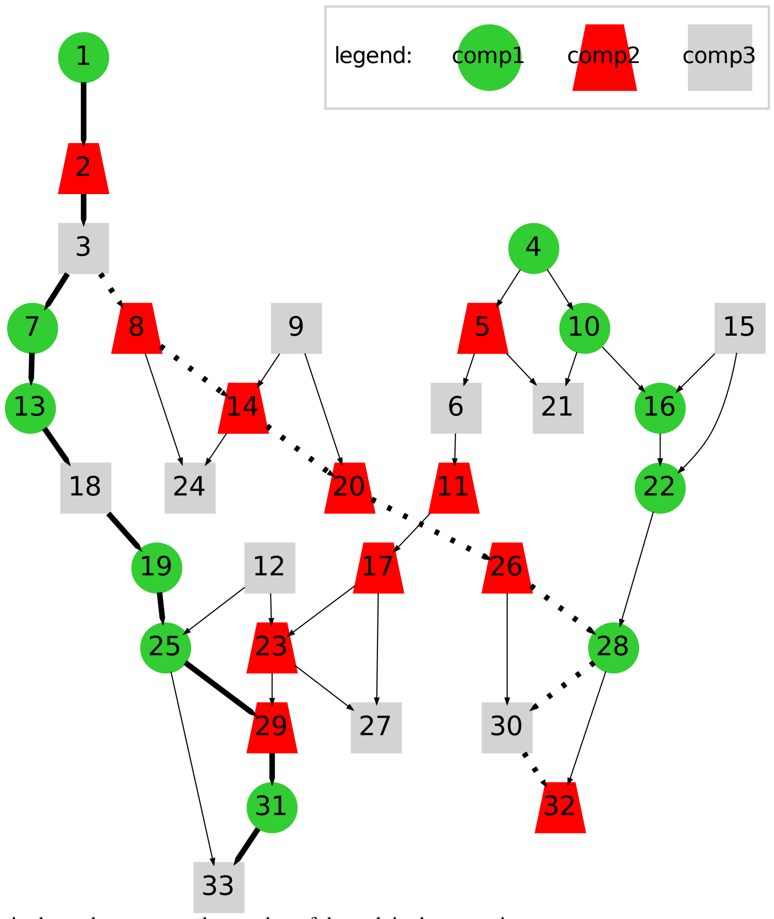
Although the application is simple, the resulting parallelism is very irregular. The task dependency graph (Figure 6.2) has sections of low concurrency, as well as sections of high concurrency (for example, tasks number 7, 8, 9, 5, 10, 15 can all execute concurrently). This irregular parallelism leads to a scalability that is hard to anticipate. Thus, for executing on 2 cores the speedup is almost ideal. On the other hand, for 3 cores the speedup is ideal, while, surprisingly, adding one more core gives no

```
#pragma omp task inout(A,B)
void comp1(float *A, float *B);
#pragma omp task inout(A,B)
void comp2(float *A, float *B);
#pragma omp task inout(A,B)
void comp3(float *A, float *B);

#define buf_cnt           19
#define iterations        11
#define step1             9
#define step2             10
#define step3             11
#define buffer_length     400

int main () {
   static float a[buf_cnt][buffer_length];

   for (int i=0; i<iterations; i++) {
      comp1( a[(i*step1)%buf_cnt], a[(i*step1+1)%buf_cnt] );
      comp2( a[(i*step2)%buf_cnt], a[(i*step2+1)%buf_cnt] );
      comp3( a[(i*step3)%buf_cnt], a[(i*step3+1)%buf_cnt] );
   }

   return 0;
}
```

Figure 6.1: Code of the motivating example.

additional speedup.

Although the tasks in the code (Figure 6.1) appear to be of the same importance, accelerating different tasks yields very different overall speedup (Figure 6.3). Figure 6.3a shows the resulting speedup when some of the tasks is accelerated by a factor of 2. On the machine with 1 or 2 cores all tasks are equally critical. However, on the machine with 3 or 4 cores, $comp1$ is significantly more critical than the other two tasks. Thus, the choice of the critical task depends on the configuration of the target machine. Moreover, if the tasks are to be accelerated by a factor of 100 (Figure 6.3b), on the machine with 4 cores, tasks $comp1$ and $comp3$ are equally critical. Thus, the choice of the critical task also depends on the potential acceleration factor. Therefore, in order to identify the critical code section, one has to take into account both the targeted machine and the acceleration factor of the possible optimization.



Note: The numbers in the nodes annotate the number of the task in the execution.

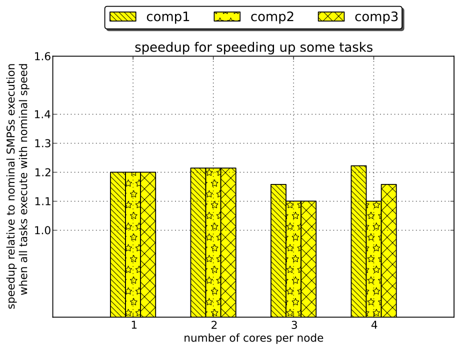
Figure 6.2: Data-dependency graph of the motivating example.

Table 6.1: Speedup for different number of cores.

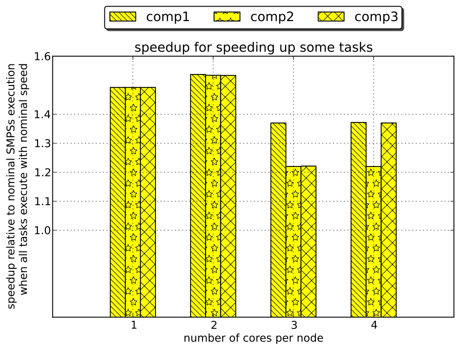
cores	1	2	3	4
speedup	1.00	1.94	3.00	3.00

6.1.2 Motivation example interpreted by the state-of-the-art techniques

A very common technique for performance analysis is tracing [55, 65, 67, 91]. Tracing environments instrument the execution and provide visualization of the obtained data. However, tracing presents to the programmer a vast amount of data, so the programmer



(a) Scaling all tasks by a factor of 2



(b) Scaling all tasks by a factor of 100

Note: The figure shows the resulting execution speedup when some of the tasks is speeded up. The overall speedup is calculated over the original execution – execution in which each task has its original duration.

Figure 6.3: The resulting speedup when accelerating different tasks.

can identify bottlenecks on his own. In other words, tracing does not directly focus the programmers attention to the bottleneck.

Profiling is usually more efficient, since it reduces the collected information into a short report. The traditional profilers report the percentage of computation time spent in each function. Thus for a sequential application, gprof [46] identifies the most time consuming code section, and that section is automatically the critical section. However, the same information derived from the parallel profilers [44, 57, 75, 79, 88] cannot directly identify the critical code section. As shown in the motivation example from Section 6.1.1, all functions take the same portion of the total computation time, but not all functions are equally critical.

Other approaches identify the critical code section as the code section that contributes the most to the critical path of execution. The representatives of this technique are Vtune [30] from Intel and Spartan [5] from University of Illinois. In the motivation example, these techniques would identify only one critical path of length 11 (the bold path marked in Figure 6.2). Then, they would identify *comp*1 as the critical function, since it has 6 task instances in the critical path. However, these tools would overlook the path of length 10 (marked with dotted bold lines in Figure 6.2). This path is slightly shorter, but it has only 2 instances of task *comp*1. Thus, accelerating *comp*1 would faster reduce the first path than the second one, and the second path would prevail. One of the illustrations of this behavior is that in Figure 6.3b, accelerating functions *comp*1 and *comp*3 often gives the same overall speedup. The tools like VTune and Spartan would fail to detect this effect.

The newest approaches identify the sections of execution with low parallelism and try to find which sections of code to blame. Quartz [6] and Intel Thread Profiler[17][29] try to quantify the potential parallelism of each section of the code. Similarly, Tallent *at. el.* [85][86] try to identify threads that are responsible for synchronizations that introduce stalls. Furthermore, they consider different policies of spreading the blame for low parallelization, from contexts in which spin-waiting occurs (victims), to directly blaming a lock holder (perpetrators). However, in the example from Section 6.1.1, seeing the execution on 3 cores (Table 6.1), these approaches would see no stalls, so they would not be able to identify any part of code as more critical than some other.

We show that the previous approaches cannot capture some influences that showed to be very decisive in the motivation example. First, that the choice of the critical section depends on the parallel target machine on which the application executes. Second, that depending on the factor of acceleration different sections may be the most beneficial to accelerate. Finally, all the previous techniques work with "measure-modify" approach. In this approach, the technique profiles the execution and points to the critical section. Then, the programmer optimizes that section, and then again runs the application "hoping" that the optimization will bring some overall speedup. Conversely, our approach anticipates the benefits prior to any optimization effort by providing a "what if" approach. This approach in advance evaluates the overall speedup of the application executed on a specified parallel machine if a specified code section is accelerated by a specified optimization factor.

6.1.3 Experiments

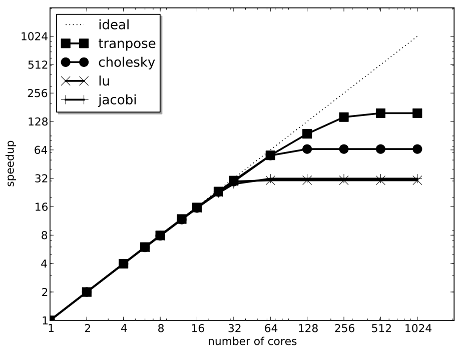
Based on the infrastructure described in Section 4.3, we designed an automatic approach to pinpoint the critical code section of MPI/OmpSs execution. First, we trace an MPI/OmpSs code and derive the trace of that execution. Then, an automatic script drives the environment to make multiple simulations of MPI/OmpSs execution, each of them with a different code section accelerated by some factor. Based on the collected results, the script identifies the code section whose acceleration yields the highest overall speedup. Finally, the script delivers to the user a single information – identification of the critical code section.

We partition the experiments Section into two parts: one studying OmpSs codes and other studying MPI/OmpSs codes. For OmpSs applications we experiment with four well-known computation kernels, while for MPI/OmpSs codes we experiment with two large MPI/OmpSs applications. For each application, we identify sections of code that should be optimized in order to increase parallelism.

OmpSs codes

In the study of OmpSs applications, we experimented with four representative kernels: Jacobi, Cholesky, LU and HM transpose. Jacobi executes on a problem size of 8192, using block size of 128. Cholesky executes on a problem size of 8192, using a block of size 128. LU factorization executes on a problem size of 4096, with the block size of 128. Finally, HM transpose executes on a problem size of 4096, with the block size of 256. For Cholesky and HM transpose, we simulate OmpSs execution of the first four iterations of the main loop. For Jacobi, we simulate OmpSs execution of the first 32 iteration of the main loop, because the major source of parallelism comes from concurrency among iterations of the main loop. Finally, for LU factorization, we simulate the entire execution.

All codes expose a substantial amount of parallelism that a many-core node can efficiently exploit (Figure 6.4). It is important to note that, with up to 32 cores in the target machine, all the applications achieve speedup that is close to the ideal. However, beyond 32 cores, the level of parallelism is different for all the applications. In par-



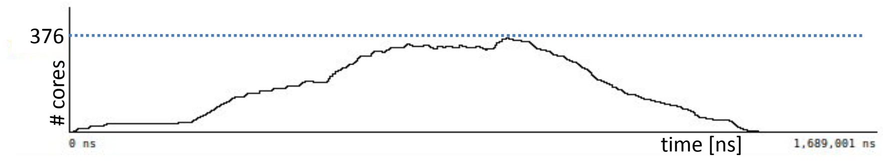
Note: Plotted speedup is the speedup of OmpSs execution (using different number of cores), over the sequential execution (using always one core). Ideal speedup is of value *N*, if OmpSs executes on a machine with *N* cores.

Figure 6.4: Parallelism of OmpSs applications.

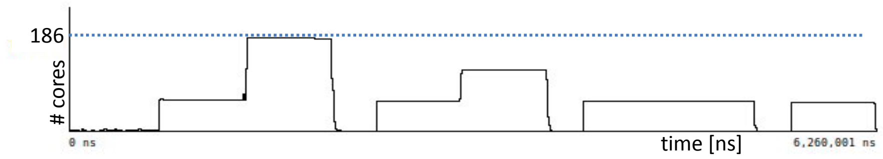
ticular, HM transpose, Cholesky, LU, and Jacobi achieve maximal speedup of 156.79, 65.36, 31.77, and 30.50, respectively (for 256 cores).

Figure 6.5 illustrates machine utilization throughout the execution of the studied applications. For each application, we show the number of cores that are active in each moment of execution. The plots are made for unlimited number of cores in the target machine. Thus, these plots illustrate the application's inherent parallelism (unbounded by the target parallel machine). Also, note that the total length of the plot represents the application's total execution time, while the surface below the plot represents the application's total computation time.

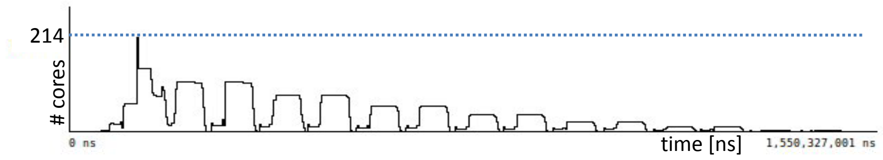
The aforementioned Figure points to the presence of two parallelization patterns. The first pattern characterizes applications HM transpose (Figure 6.5a) and Jacobi



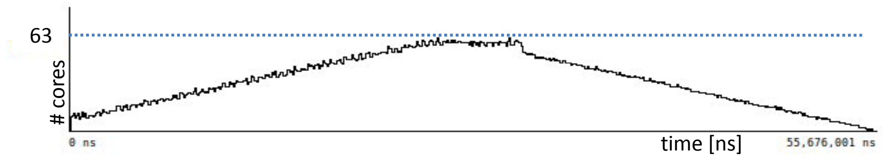
(a) HM transpose



(b) Cholesky



(c) LU factorization



(d) Jacobi

Note: These Paraver views show the number of cores that are active in each moment of execution. All the applications execute with the unlimited number of cores on the target machine. It is interesting to note that the surface below the plot represents the application's computation time, while the total length of the plot represents the application's total execution time.

Figure 6.5: Number of cores active during execution.

HM transpose	
task name	percentage [%]
main_task	0.61
transpose	2.77
transpose_exchange	96.63

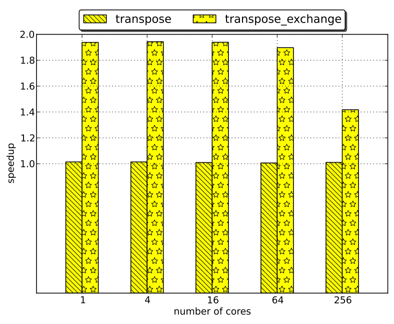
Cholesky	
task name	percentage [%]
main_task	0.33
sgemm_tile	59.29
spotrf_tile	0.52
ssyrk_tile	0.75
strsm_tile	39.12

LU factorization	
task name	percentage [%]
main_task	0.28
bdiv	2.76
block_mpy_add	54.44
bmod	38.29
fwd	3.49
lu0	0.53

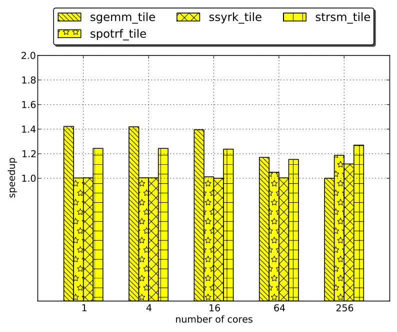
Jacobi	
task name	percentage [%]
main_task	2.35
getfirstcol	3.18
getfirstrow	0.98
getlastcol	1.48
Getlastrow	1.46
jacobi	90.54

Note: The tables suppress functions that take less 0.2% of the total computation time of the application.

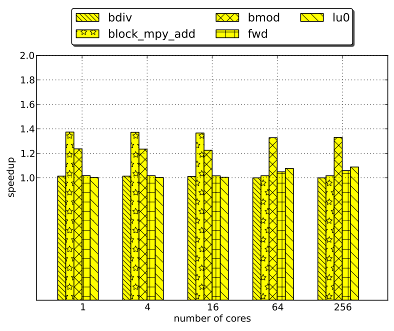
Figure 6.6: Tasks profile



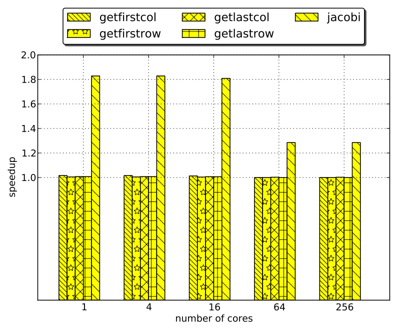
(a) HM transpose



(b) Cholesky



(c) LU factorization



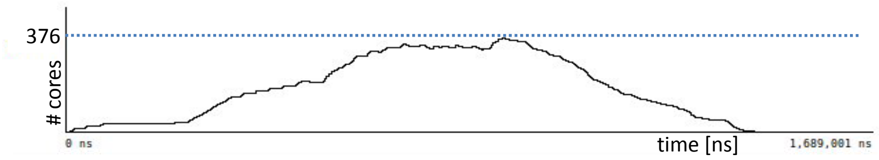
(d) Jacobi

Figure 6.7: Speedup when one task is speeded up by 2x (OmpSs codes).

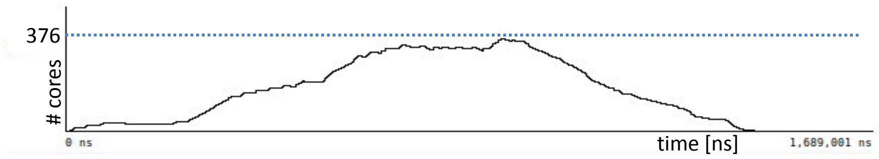
(Figure 6.5d). In these applications, as the execution approaches the middle of the run, increases the number of cores that can be utilized concurrently. After reaching the peak concurrency, the parallelism drops until the end of the run. This way, HM transpose and Jacobi achieve the peak parallelism of 376 and 63, respectively.

The other parallelization pattern characterizes applications Cholesky (Figure 6.5b) and LU factorization (Figure 6.5c). In these applications, throughout the run, the execution passes through interleaved sections of high and low parallelism. Usually, each following section of high parallelism has lower peak parallelism than the previous one. Thus, in Cholesky, peak speedups of the first four iterations are 186, 123, 61 and 60, respectively, while in LU factorization, peak speedups of the first four iterations are 214, 113, 113 and 84, respectively.

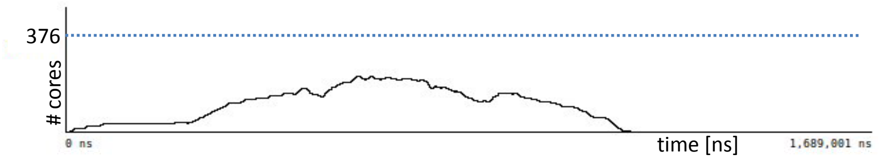
Figure 6.6 shows the distribution of the computation time on different tasks. Applications HM transpose and Jacobi have one very dominant task – HM transpose spends 96.63% of time in *transpose_exchange*, while Jacobi spends 90.54% of time in *jacobi*. Other two applications have two dominant tasks – Cholesky spends 59.29% of



(a) all tasks with the original duration

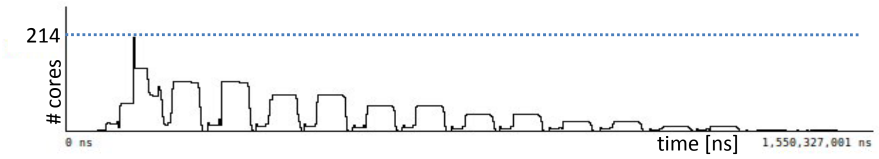


(b) *transpose* speeded up two times

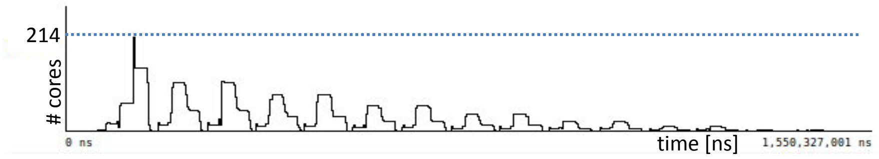


(c) *transpose_exchange* speeded up two times

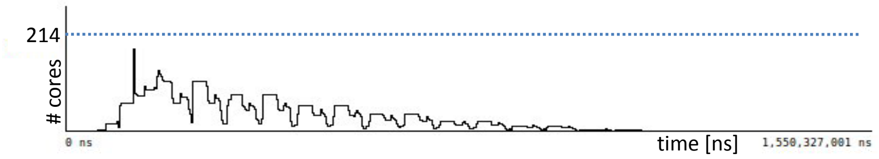
Figure 6.8: HM transpose: number of active cores



(a) all tasks with the original duration



(b) *block_mpy_add* speeded up two times



(c) *bmod* speeded up two times

Figure 6.9: LU factorization: number of active cores

time in *sgemm_tile* and 39.12% in *strsm_tile*, while LU factorization spends 54.44% of time in *block_mpy_add* and 38.29% in *bmod*. In a sequential execution of all the four applications, these dominant tasks would be the best candidates for optimization.

However, in the parallel execution, the function that is dominant in the total computation time is not necessarily the function that is critical for the execution time. Figure 6.7 shows the speedup obtained by speeding up some of the functions by a factor of 2. In HM transpose and Jacobi, for a target machine with a low number of cores, optimizing the identified dominant function brings the highest benefits. However, as the number of cores in the target machine grows, these benefits significantly drop. Nevertheless, Cholesky and LU show even more drastic behavior. For a machine with a small number of cores, the critical function is the most dominant one – *sgemm_tile* for Cholesky, and *block_mpy_add* for LU factorization. Nevertheless, as the target machine gets more cores, other tasks are becoming critical. In fact, in both applications when executing on 256 cores, accelerating the dominant function yield the overall speedup of less than 2%.

Identification of the critical tasks can be brought in relation with the parallelization pattern of the application. In HM transpose and Jacobi, the dominant task is the critical one. In HM transpose, speeding up the task that takes a small portion of computation time has no detectable effect on the execution of the application (Figure 6.8b). On the other hand, speeding up *transpose_exchange* by a factor of 2 (Figure 6.8c), significantly reduces both the computation and the execution time. It is interesting to note that the peak parallelism drops, as well. This happens because, as *transpose_exchange* takes shorter time, the probability of concurrent execution of different instances of that task drops.

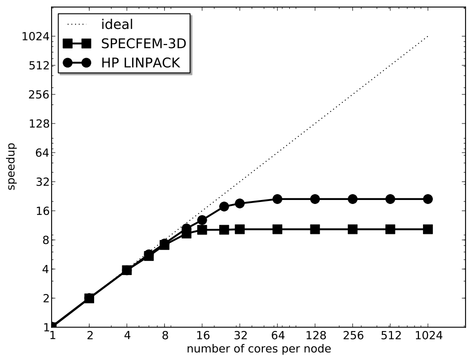
On the other hand, in Cholesky and LU factorization, some task that takes a small share of the total computation time may become a task that is critical for the total execution time. In LU factorization (Figure 6.9a), the total computation time is $47.29s$, while the total execution time is $1.55s$. When speeding up *block_mpy_add* by a factor of 2, the computation time drops for $\frac{54.44\%}{2} = 27.22\%$, while the total execution time drops for only 1.97% (Figure 6.9b). On the other hand, when speeding up *bmod* by a factor of 2, the computation time drops for $\frac{38.29\%}{2} = 19.14\%$, while the total parallel execution time drops for 32.48% (Figure 6.9c). The figures show that speeding up *block_mpy_add* shrinks the sectors with high parallelism. On the other hand, speeding up *bmod* shrinks the sectors with low parallelism, thus bringing the sectors with high parallelism closer together. Therefore, a small reduction in the total computation time can cause a significant reduction in the total execution time.

MPI/OmpSs codes

In the study of MPI/OmpSs applications, we use the codes of HP Linpack [36] and SPECFEM-3D [22]. Linpack is one of the most famous parallel applications used to rank parallel machines on the top 500 list [87]. We run it for the problem size of 65536 and block size of 128. SPECFEM-3D simulates earthquakes in complex three-dimensional geological models. We run it with the configuration $NSPEC = 198352$ and $NGLOB = 12912201$. We simulate only four iterations of the main loop for each application. Both applications execute with 16 MPI process with each MPI process running on a separate node in the machine.

Figure 6.10 shows the potential parallelism of the studied MPI/OmpSs codes. If the parallel machine has up to 8 cores per node, both application express enough parallelism to achieve the parallelization close to the ideal. However, for machines with more cores per node, the speedup grows until saturation. In particular, SPECFEM-3D saturates on 32, and Linpack on 64 cores per node. The resulting maximum speedups are of 10.31 for SPECFEM-3D and 21.18 for Linpack.

Finally, MPI/OmpSs applications especially emphasize the phenomenon where a task that takes a small portion of the computation time becomes the critical task in the execution. In SPECFEM-3D, *scatter* takes only 6.37% of the total computation time



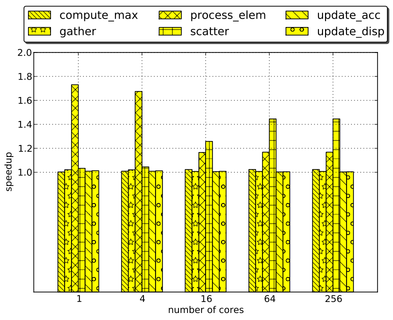
Note: Plotted speedup is the speedup of MPI/OmpSs execution (using different number of cores per node), over the MPI execution (using always one core per node). Ideal speedup is of value N, if MPI/OmpSs executes on a machine with N cores per node.

Figure 6.10: Parallelism of MPI/OmpSs applications.

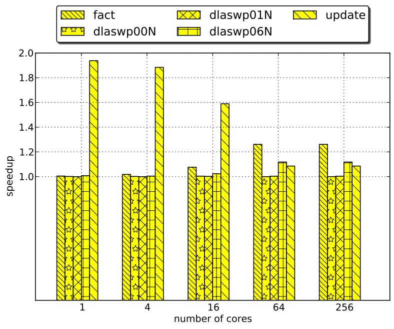
SPECFEM-3D	
task name	percentage [%]
compute_max	0.46
gather	4.15
process_elem	84.60
scatter	6.37
update_acc	1.75
update_disp	2.55

HP Linpack	
task name	percentage [%]
fact	0.49
dlaswp00N	0.34
dlaswp01N	0.63
dlaswp06N	1.29
update	96.99

Figure 6.11: Tasks profile



(a) SPECFEM-3D



(b) HP Linpack

Figure 6.12: Speedup when one task is speeded up by 2x (MPI/OmpSs codes).

(Figure 6.11). Thus, speeding up *scatter* by a factor of 2 reduces the total computation time for $\frac{6.37\%}{2} = 3.14\%$. However, when the application executes with 64 cores per node, this small reduction of total computation time results in the overall execution speedup of 44.16% (Figure 6.12a).

HP Linpack shows even more extreme behavior. The dominant task in HP Linpack (*update*), takes 96.99% of the total execution time. Speeding up *update* by a factor of 2, reduces the computation time for $\frac{96.99\%}{2} = 48.50\%$. When HP Linpack executes with only 1 core per node, this reduction of computation time results in the overall speedup of 95.46%. However, when HP Linpack executes with 64 cores per node, the same reduction of the computation time results in less than 10% of overall speedup. On the other hand, $fact$ is the fourth most dominant function, taking only 0.49% of the total computation time (Figure 6.11). Speeding up $fact$ by a factor of 2 reduces the computation time for $\frac{0.49\%}{2} = 0.24\%$. Still, when HP Linpack executes with 64 cores per node, this small reduction in computation time brings the overall application speedup of 24.54% (Figure 6.12b).

This effect happens because task instances of *update* are highly parallelizable, while task instances of $fact$ are extremely non-parallelizable. Although *update* takes a big portion of the computation time, instances of *update* parallelize across available cores without throttling the execution. On the other hand, $fact$ takes a small portion of computation time, but instances of $fact$ cannot execute concurrently on different cores. Furthermore, fact has numerous MPI communications so execution of some instance of $fact$ in one MPI process also may condition some other instance of $fact$ in other

MPI processes.

6.1.4 Conclusion and future research directions

Task based dataflow programming models potentially extract very irregular parallelism, making it hard to estimate the potential parallelism in applications. For a real application with thousands of tasks, this estimation exceeds the prediction ability of any human programmer. To tackle this problem, we design an environment that quickly estimates the possible parallelization of an application on a parallel platform. Furthermore, the environment pinpoints the causes that limit the dataflow parallelism. These two features of the environment can significantly facilitate the development and optimization of applications based on dataflow programming models.

We showed how, using the designed environment, a programmer can easily identify the critical task – a task that should be optimized in order to increase the scalability of the code. We identified that many parallel applications exhibit the phenomenon in which a task that takes a small portion of the total computation time may decisively contribute to the total parallel execution time. In particular, in the case of HP Linpack, function $fact$ takes only 0.49% of the total computation time. However, when the application executes with 16 MPI processes, with 64 cores for each MPI process, speeding up task $fact$ by a factor of 2 results in the overall execution speedup of 24.54%. Also, we observed that this phenomenon is especially significant at large-scale.

Finally, it is our great hope, that our environment can be very useful to help a programmer understand the mechanisms of dataflow parallelism. The environment allows fast and flexible simulation, so various influences on the dataflow can be studied quickly. Moreover, the environment provides visualization support so the programmer can qualitatively inspect the simulated dataflow execution. We believe that our environment can be of great help to all newcomers to the dataflow programming models.

To investigate other approaches for tuning MPI/OmpSs parallelism, in following Sections we explore the techniques for identifying optimal task decomposition of the code.

6.2 Tareador: exploring parallelism inherit in applications

New proposals for large-scale programming models are persistently spawned, but most of these initiatives fail because they attract little interest of the community. It takes a giant leap of faith for a programmer to take the already working parallel application and to port it to a novel programming model. This is especially problematic because the programmer cannot anticipate how would the application perform if it was ported to the new programming model, so he may doubt whether the porting is worth the effort. Moreover, the programmer usually lacks developing tools that would make the process of porting easier.

In order to port the application to a task-based programming language, the programmer must partition the sequential code into tasks and take advantage of the existing dataflow parallelism inherent in the application. However, obtaining the partitioning that achieves optimal parallelism is not trivial because it depends on many parameters such as the underlying data dependencies and global problem partitioning. To assist the process of finding a partitioning that achieves high parallelism, we designed Tareador (Section 4.4) – a framework that a programmer can use to find the best strategy to expose dataflow parallelism in his application. Furthermore, we present an iterative approach that uses Tareador to find the optimal task decomposition of the code. The presented approach requires only superficial knowledge of the studied code and iteratively leads to the optimal partitioning strategy.

The main objective of Tareador is to get a wider community involved with OmpSs by encouraging MPI programmers to port their applications to MPI/OmpSs. This encouragement is strictly related to assuring the programmer that he can benefit from this porting and that the porting would be easy. Therefore, our goal in this study is to illustrate how Tareador can be used to:

- Help an MPI programmer estimate how much parallelism he can achieve using MPI/OmpSs
- Help an MPI programmer find the optimal strategy to port his MPI application to MPI/OmpSs

6.2.1 Motivating example

Even if the sequential application is trivial, finding the optimal task decomposition can be a difficult job. Dataflow parallelism allows exploiting very irregular and distant parallelism, parallelism among sections of code that are mutually far from each other. This type of parallelism is very hard for the programmer to identify and expose without any development support. The programmer must know the application in depth in order to identify all the data dependencies among tasks. Furthermore, even knowing all dependencies, the programmer must anticipate how will all the tasks execute in parallel, and what is the possible parallelism that these tasks can achieve.

Figure 6.13 shows a simple sequential application composed of four computational parts (A, B, C and D), the data dependencies among those parts, and some of the possible taskification strategies. Although the application is very simple, it allows many possible decompositions that expose different amount of parallelism. $T0$ puts all the code in one task and, in fact, presents a sequential code. $T1$ and $T2$ both break the application into two tasks but fail to expose any parallelism. On the other hand, $T3$ and $T4$ both break the application into 3 tasks, but while $T3$ achieves no parallelism, $T4$ exposes parallelism between C and D. Finally, $T5$ breaks the application into 4 tasks but achieves the same amount of parallelism as $T4$. Considering that increasing the number of tasks increases the runtime overhead, one can conclude that the optimal taskification is $T4$, because it gives the highest speedup with the lowest cost of the increased number of tasks.

Nevertheless, compared to the presented trivial execution, a real-world application would be more complex in various aspects. A real application would have hundreds of thousands of task instances, causing complex and well populated dependency graphs. The large dependency graph would allow unpredictable scheduling decisions that would exploit very distant parallelism. Also, with the task instances of different

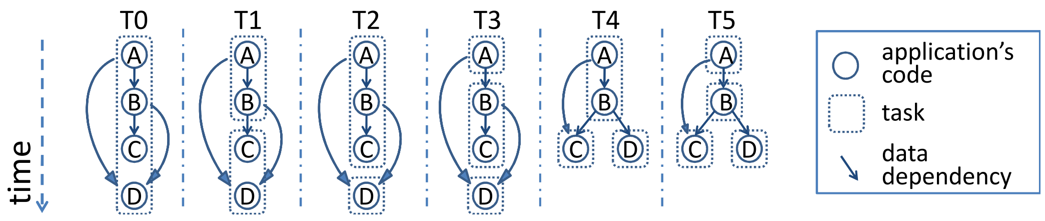


Figure 6.13: Execution of different possible taskifications for a code composed of four parts.

length, evaluating the potential parallelism would be even harder. Moreover, in an MPI/OmpSs application, the total number of task increases by the factor of the number of MPI processes. Furthermore, taskifying MPI calls spreads inter-task dependencies across MPI processes. Finally, the MPI transfers cause communication delays that also affect overall parallelism. Due to all this complexity, it is unfeasible for a programmer to do the described analysis and estimate the potential parallelism of a certain task decomposition.

We believe that it would be very useful to have an environment that quickly anticipates the potential parallelism of a particular taskification. We designed such environment and we show how it should be used to find the optimal taskification. Furthermore, we present two cases studies that use a black-box approach to explore potential task decomposition of applications.

6.2.2 Experiments

Our experiments demonstrate how the programmer can use Tareador to explore the potential parallelization of an application. We present two case studies that explore the potential parallelization strategies in the sequential code of Cholesky and the MPI code of HP Linpack. Our experiments demonstrate the top-down trial-and-error approach that requires no knowledge of the studied code, and ultimately leads to exposing dataflow parallelism in the code. The approach uses the following algorithm:

1. Propose a coarse-grained taskification of the code;
2. Given the taskification, estimate the potential parallelism and obtain the visualization of the parallel execution.
3. Based on the estimation of the potential parallelism, choose a finer-grained taskification that should expose more parallelism.
4. Return to step 2.

We start from the most coarse-grain taskification ($T0$) that puts whole execution into one task. Then, using $T0$ as the baseline decomposition, we determine the *potential parallelism of Ti normalized to $T0$* as the speedup of Ti over $T0$ when both taskifications execute on an idealized machine with unlimited number of cores.

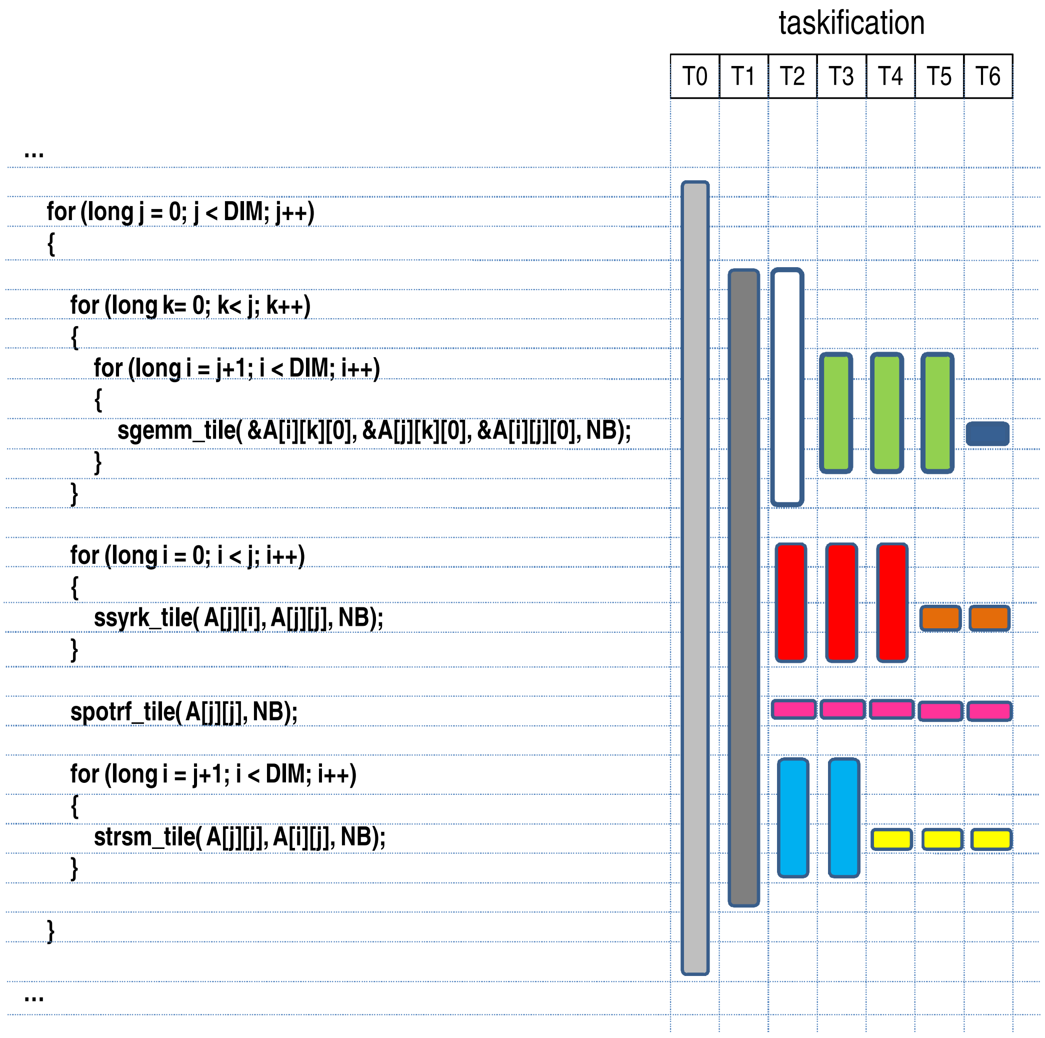
Due to the huge overhead introduced by Tareador, we conduct the experiments on small input sets. Tareador counts executed instructions and translates that information to get a notion of time. Thus, the time spent in each code section should be considered not as the absolute value, but rather as relative comparison of the time spent in different code sections.

Case study 1: Using Tareador on Cholesky

Our first case study explores the potential parallelization of the sequential code of Cholesky. We used Cholesky with the problem size of 1024 and various granularities of execution (block sizes of 8, 16, 32, and 64). We are primarily interested in the potential parallelism inherent in the application. Thus, we often simulate the parallel execution on the idealized target machine – a machine with infinite number of cores. These results represent the upper bound of achievable parallelism. Finally, we show how this inherent parallelism results in speedup when the application executes on a realistic target machine.

First, the framework instruments the application to obtain the profile that guides the taskification process. Table 6.14b shows the accumulated time spent in each function of the application. This information identifies functions that need to execute concurrently in order to achieve significant parallelism. In this case, this is *sgemm_tile*, because it is the most time consuming function. On the other hand, Table 6.14c shows the average duration of each function. This information identifies which function is a good candidate to be decomposed into smaller tasks. It is important to note that decreasing BS reduces execution time of most of the functions, thus making a finer granularity of execution.

Considering the data showed in the described tables, we start the process of exploring parallelism by iteratively refining decomposition. Figure 6.14a illustrates the code of Cholesky and the tested taskifications. On the other hand, Figure 6.15 shows the Tareador output for the selected taskifications – the number of created tasks and the obtained parallelism. The top-down trial-and-error iterative approach starts with baseline taskification $T0$ that puts the whole sequential execution into one task. The next taskification $T1$ puts each iteration of the main loop in one task. This strategy gives no additional parallelism compared to $T0$. Furthermore, $T2$ breaks each iteration of the main loop into three separate loops and the call to function *spotrf_time*. This decomposition exposes very limited parallelism. $T3$ additionally refines the decompo-



```
...
for (long j = 0; j < DIM; j++)
{
    for (long k= 0; k< j; k++)
    {
      for (long i = j+1; i < DIM; i++)
      {
        sgemm_tile( &A[i][k][0], &A[j][k][0], &A[i][j][0], NB);
      }
    }

    for (long i = 0; i < j; i++)
    {
      ssyrk_tile( A[j][i], A[j][j], NB);
    }

    spotrf_tile( A[j][j], NB);

    for (long i = j+1; i < DIM; i++)
    {
      strsm_tile( A[j][j], A[i][j], NB);
    }

}
...
```

(a) Potential decompositions of Cholesky.

		granularity			
		BS-64	BS- 32	BS-16	BS-8
task name	spotrf_tile	2.54	0.89	0.32	0.11
	strsm_tile	18.80	9.90	5.01	2.66
	sgemm_tile	59.06	76.67	86.63	90.18
	ssyrk_tile	19.57	12.40	7.30	3.81
	main_task	0.03	0.14	0.74	3.23

(b) Distribution of time spent in tasks (%).

		granularity			
		BS-64	BS- 32	BS-16	BS-8
task name	spotrf_tile	0.259	0.054	0.012	0.004
	strsm_tile	0.547	0.081	0.014	0.003
	sgemm_tile	0.859	0.134	0.024	0.005
	ssyrk_tile	0.569	0.101	0.020	0.005

(c) Average function duration (ms).

Figure 6.14: Exploring potential decomposition of Cholesky code

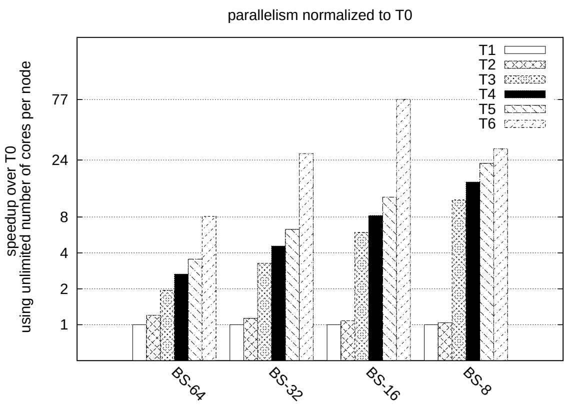
sition, by breaking the uppermost loop into its iterations. This decomposition achieves a significant parallelism, reaching the parallelism of 11.09 for block size of $BS = 8$. Further refining of taskifications passes through $T4$, $T5$ and $T6$, reaching the maximal parallelism of 77.20 for decomposition $T6$ and block size $BS = 16$.

Refining decomposition increases the number of generated task instances (Table 6.15a). Also, reducing block size further augments the number of instances. Higher number of tasks *virtually always* provides higher potential parallelism (Figure 6.15b). However, in decomposition $T6$, transition from block size 16 to block size 8 causes the drop in parallelism. This can be explained by very fine granularity, that causes the increase of the loop control overhead. The overhead increases the percentage of time spent in the main task. Consequently, for block size of 8, the execution spends 3.23% of total computation time in the main task (Table 6.14b). Since the main task is not parallelized, Amdahl's law limits the potential parallelism to $\frac{100}{3.23} = 30.96$. Therefore, in this configuration, the achieved parallelism is 29.78 (Figure 6.15b).

Figure 6.16 shows the speedup and the parallel efficiency of $T6$ for different number of cores. For all tested block sizes, $T6$ performs similar for machines with up to four cores(Figure 6.16a). However, on machines with more cores, different block sizes achieve different speedups. Thus, on 128 cores, block size of 64, 32, 16 and 8 achieve speedups of 8.08, 27.18, 77.20 and 29.78, respectively. Furthermore, Figure

		granularity			
		BS-64	BS-32	BS-16	BS-8
taskification	T1	8	16	32	64
	T2	32	64	128	256
	T3	52	168	592	2.208
	T4	72	272	1.056	4.126
	T5	92	376	1.520	6.112
	T6	120	816	5.984	45.760

(a) Total number of tasks created.



(b) Speedup normalized to $T0$.

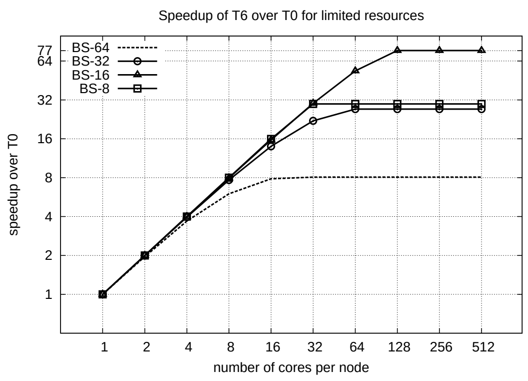
Note: In Figure 6.15b, all taskifications ($T0$-$T6$) execute in OmpSs fashion on an ideal target machine. Then, the parallelism of taskification Ti normalized to taskification $T0$ represents the speedup of taskification Ti over taskification $T0$.

Figure 6.15: Number of task instances and the potential parallelism for various taskifications of Cholesky.

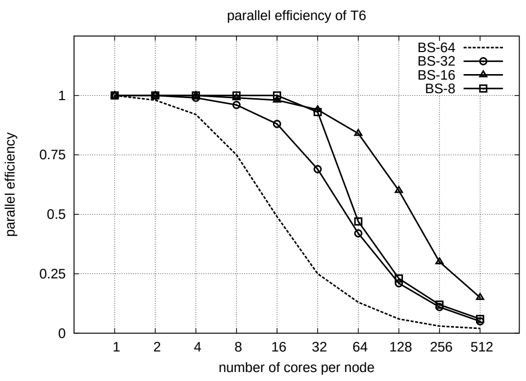
6.16b shows parallel efficiency (core utilization) – the ratio between the application's speedup achieved on some parallel machine and the number of cores in that machine. The higher the parallelism of some configuration (Figure 6.15b), the bigger machine that configuration can utilize efficiently. Assuming that utilization is **efficient** if the parallel efficiency is higher than 75%, the block size of 64, 32, 16 and 8 can efficiently utilize machines of approximately 8, 28, 88 and 41 cores, respectively. Also, it is interesting to note that for 16 cores, *BS-8* achieves slightly higher efficiency than *BS-16*. However, on machines with more cores, *BS-16* becomes by far more efficient than *BS-8*. Also, although the *BS-64* achieves the maximum speedup of 8.08 (Figure 6.16a), executing on a machine with 8 cores, *BS-64* achieves the efficiency of only 75%.

Case study – Using Tareador on HP Linpack

Our second case study explores MPI/OmpSs execution of HP Linpack on a cluster of many-core nodes. We used HPL with the problem size of 8192 and with 2*x*2 (PxQ) data decomposition. Also, we test various granularities of execution by running HPL with block sizes (*BS*) of 32, 64, 128, and 256. Our target machine consists of four many-core nodes, with one MPI process running on each node. Each node has an infinite number of cores while the network connecting nodes is ideal (infinite bandwidth and zero latency).



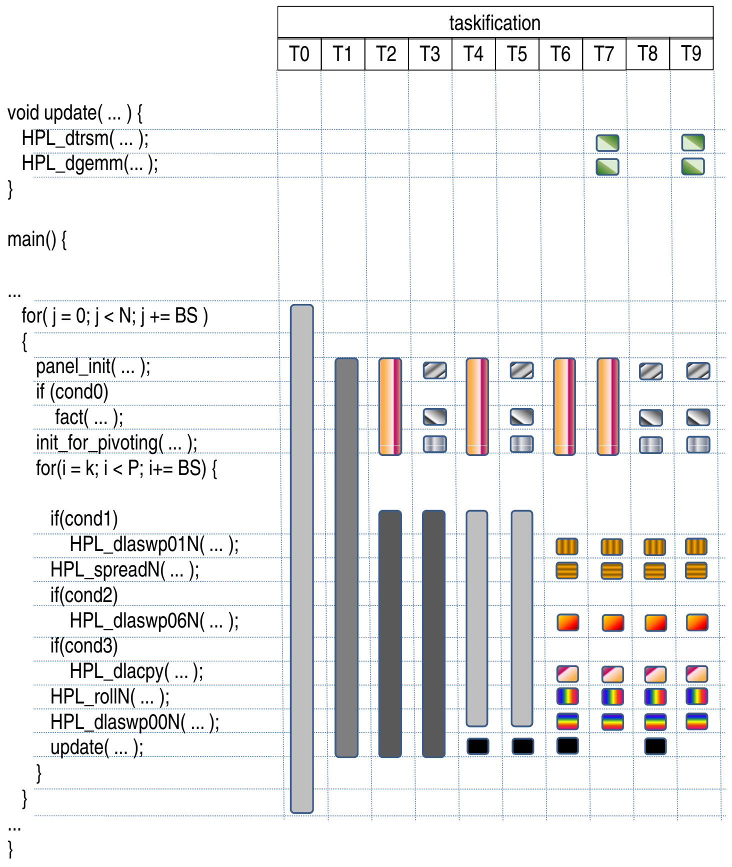
(a) Speedup of T6 for different granularities.



(b) Parallel efficiency for T6 for different granularities.

Note: Parallel efficiency denotes the ratio between the application's speedup achieved on some parallel machine and the number of cores of that parallel machine. In fact, the metric presents average core utilization in the whole machine.

Figure 6.16: Speedup and parallel efficiency for $T6$ for various number of cores.



```
void update( ... ) {
  HPL_dtrsm( ... );
  HPL_dgemm(... );
}

main() {

...
  for( j = 0; j < N; j += BS )
  {
    panel_init( ... );
    if (cond0)
      fact( ... );
    init_for_pivoting( ... );
    for(i = k; i < P; i+= BS) {

      if(cond1)
        HPL_dlaswp01N( ... );
      HPL_spreadN( ... );
      if(cond2)
        HPL_dlaswp06N( ... );
      if(cond3)
        HPL_dlacpy( ... );
      HPL_rollN( ... );
      HPL_dlaswp00N( ... );
      update( ... );
    }
  }
...
}
```

(a) HPL and the evaluated taskifications.

			granularity			
			BS-32	BS- 64	BS-128	BS-256
task name	outer	panel_init	0.0003	0.0002	0.0001	0.0000
		fact	0.7525	1.2071	1.8795	3.2077
		init_for_pivoting	0.0246	0.0487	0.0925	0.1795
	inner	HPL_dlaswp01N	0.2583	0.2917	0.2906	0.2815
		HPL_spreadN	0.1599	0.0800	0.0378	0.0181
		HPL_dlaswp06N	0.1222	0.1359	0.1367	0.1274
		HPL_rollN	0.3267	0.1619	0.0762	0.0363
		HPL_dlacpy	0.0857	0.0932	0.0929	0.0912
		HPL_dlaswp00N	0.3706	0.4485	0.4736	0.4837
	update	HPL_dtrsm	0.8269	1.6674	2.8347	5.0772
		HPL_dgemm	97.0683	95.8614	94.0813	90.4935

(b) Distribution of total execution time spent in tasks (%).

			granularity			
			BS-32	BS- 64	BS-128	BS-256
task name	outer	panel_init	0.0003	0.0003	0.0003	0.0003
		fact	1.3468	3.7670	11.3572	37.8463
		init_for_pivoting	0.0221	0.0761	0.2797	1.0594
	inner	HPL_dlaswp01N	0.0073	0.0289	0.1130	0.4392
		HPL_spreadN	0.0022	0.0040	0.0073	0.0141
		HPL_dlaswp06N	0.0034	0.0135	0.0532	0.1987
		HPL_rollN	0.0046	0.0080	0.0148	0.0283
		HPL_dlacpy	0.0024	0.0092	0.0361	0.1423
		HPL_dlaswp00N	0.0052	0.0222	0.0921	0.3773
	update	HPL_dtrsm	0.0117	0.0826	0.5514	3.9605
		HPL_dgemm	1.3677	4.7492	18.3016	70.5900

(c) Average function duration (ms).

Note: In Tables 6.17b and 6.17c, apart from statistic for each function of the code, we present the statistics for two logical sections: **outer** – consisting of *panel_init*, *fact* and *init_for_pivoting*; and **inner** consisting of *HPL_dlaswp01N*, *HPL_spreadN*, *HPL_dlaswp06N*, *HPL_rollN*, *HPL_dlacpy* and *HPL_dlaswp00N*.

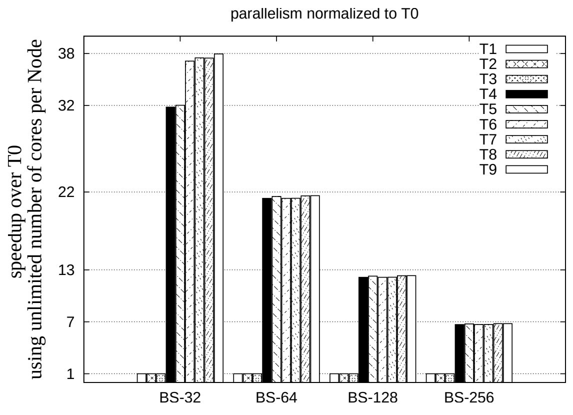
Figure 6.17: Exploring task decompositions of HP Linpack.

Again, the framework instruments the application to obtain the profile that guides the taskification process. Table 6.17b shows the accumulated time spent in each function of the application. In this example, a good taskification should provide concurrency of instances of functions *update*, because the application spends in that function at least 95.57% of the computation time. On the other hand, Figure 6.17c shows the average duration of each function. This information identifies which function is a good candidate to be decomposed into smaller tasks. In this example, function *panel_init* is very short so breaking it into smaller tasks makes little sense. Also, it is important to note that decreasing *BS* reduces execution time of most of the functions, thus generating finer-grained execution.

We pass the HPL code through the iterative approach for exploring potential taskifications (Figure 6.17a). We adopt the baseline taskification $T0$ that makes only one task per MPI process. $T1$ puts each iteration of the outer loop in one task, but this strategy exposes no additional parallelism. Furthermore, $T2$ breaks down the code into section *outer* and separate iterations of the inner loop, still providing no improvement in speedup. $T3$ additionally breaks down section *outer*, but with no increases in speedup. Finally, $T4$ compared to $T2$ separates section *inner* from function *update* and releases the significant amount of parallelism. Namely, it achieves the speedup of 6.76, 12.28, 21.48 and 32.02 for block sizes of 256, 128, 64 and 32, respectively (Figure 6.18b). Also, $T4$ significantly increases the number of tasks in the application to 2.128, 8.336,

		granularity			
		BS-32	BS-64	BS-128	BS-256
taskification	T1	1024	516	260	132
	T2	66.314	16.778	4.298	1.130
	T3	69.898	18.570	5.194	1.578
	T4	131.600	33.040	8.336	2.128
	T5	135.184	34.832	9.232	2.576
	T6	327.458	81.826	20.450	5.122
	T7	425.382	106.216	26.502	6.614
	T8	331.042	83.618	21.346	5.570
	T9	428.966	108.006	27.398	7.062

(a) Total number of tasks created.



(b) Speedup normalized to $T0$.

Note: In Figure 6.18b, all taskifications ($T0$-$T9$) execute in MPI/OmpSs fashion on an ideal target machine. Then, the speedup of taskification Ti over taskification $T0$ represents the parallelism of taskification Ti normalized to taskification $T0$.

Figure 6.18: Number of task instances and the potential parallelism of each taskification.

33.040 and 131.600 for block sizes of 256, 128, 64 and 32, respectively (Figure 6.18a).

Using Paraver visualization, the programmer can visually inspect the tested taskifications and understand the nature of the exposed parallelism. Taskification $T2$ joins sections *inner* and *update* into one task (Figure 6.17a). Since all these tasks are mutually dependent, $T2$ provides no parallelism. On the other hand, $T4$ breaks that task into separate tasks of *inner* and *update*. Paraver view from Figure 6.19 reveals that each task *inner* depends on the task *inner* from the previous iteration of the loop; and each task *update* depends on the task *inner* from the same iteration of the loop. Since *inner* is much shorter than *update*, all dependent instances of *inner* can serialize quickly, and then independent instances of *update* can execute concurrently.

Further decomposition of *outer*, *inner* and *update* contributes little to the potential speedup (Figure 6.18b). Breaking *outer*, for block sizes of 256, 128 and 64, causes slightly higher parallelism of $T5$, $T8$ and $T9$, compared to $T4$, $T6$ and $T7$. On the other hand, breaking *inner*, for block size of 32, causes significantly higher parallelism of $T6$, $T7$, $T8$ and $T9$, compared to $T4$, $T5$. This effect happens because for very high concurrency of *update* (speedup is higher than 30), the critical path of the execution moves and starts passing through section *inner*. In these circumstances, breaking *inner* significantly increases parallelism by allowing concurrency of functions *HPL_dlaswp00N*, *HPL_dlaswp01N* and *HPL_dlaswp06N*. Finally, breaking of *update*, for block size 32, causes slightly higher parallelism of $T9$ compared to $T8$.

Figure 6.20 shows the speedup and parallel efficiency of $T9$ for different number

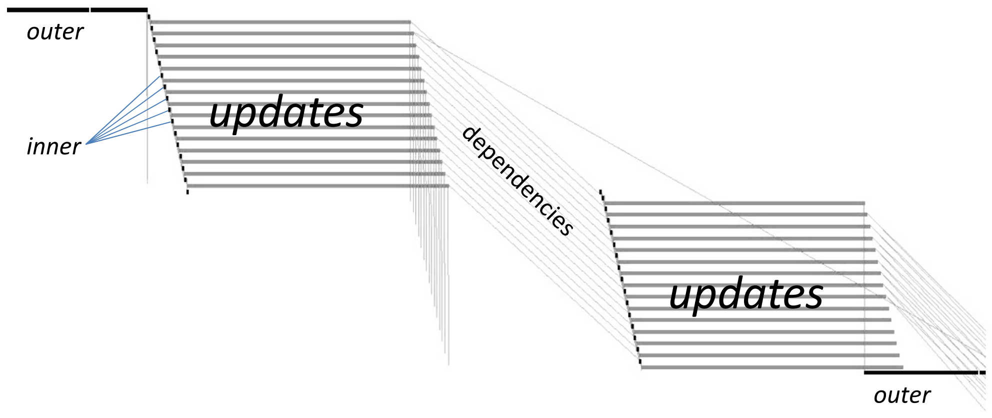
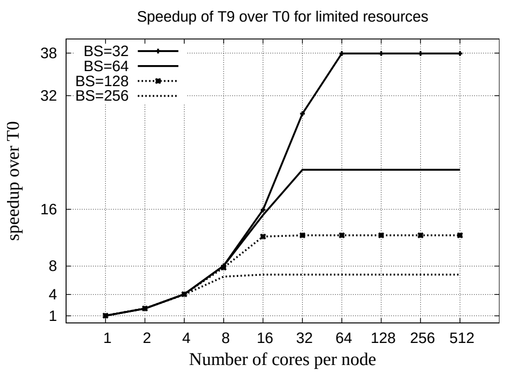


Figure 6.19: Paraver visualization of the first 63 tasks and the dependencies among them (taskification T4, BS=256).

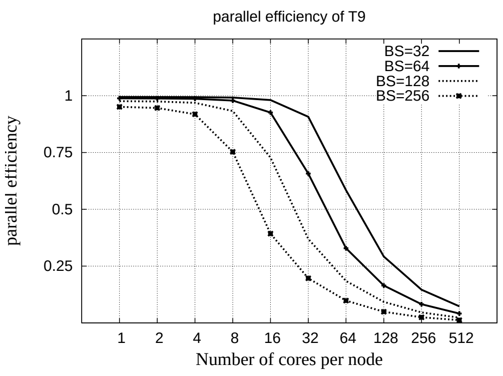
of cores per node. The results show that high parallelism in the application is useful not to achieve high speedup on a small parallel machine, but rather to utilize efficiently a large parallel machine. Figure 6.20a shows that for a machine with 4 cores per node, $T9$ with all block sizes achieves the speedup of around 4, with difference between the highest and the lowest of less than 2%. However, for a machine with 32 cores per node, $T9$ with block sizes of 256, 128, 64 and 32, achieves the speedup of 6.80, 12.34, 21.57 and 29.47, respectively. Furthermore, Figure 6.20b shows parallel efficiency (core utilization) – the ratio between the application's speedup achieved on some parallel machine and the number of cores in that machine. Adopting that an application efficiently utilizes a machine if the parallel efficiency is higher than 75%, the results show that $T9$ with block sizes of 256, 128, 64 and 32, can efficiently utilize the machine of 8, 15, 26 and 47 cores per node, respectively. Therefore, to efficiently employ many-core machine with hundreds of cores per node, HPL has to expose even more parallelism, for instance, by making finer-grain taskification with further reduction of block size.

6.2.3 Conclusion and future research directions

Tasks-based parallel programming languages are promising in exploiting additional parallelism inherent in MPI parallel programs. However, the complexity of this type of execution impedes an MPI programmer from anticipating how much dataflow par-



(a) Speedup.



(b) Parallel efficiency.

Note: Parallel efficiency denotes the ratio between the application's speedup achieved on some parallel machine and the number of cores of that parallel machine. Infact, the metric presents the overall average core utilization in the whole machine.

Figure 6.20: Speedup and parallel efficiency for T9 for various number of cores.

allelism can be extracted from the application. Moreover, it is nontrivial to determine which parts of code should be encapsulated into tasks in order to expose the parallelism and still avoid creating unnecessary tasks that increase runtime overhead. To address this issue, we have developed Tareador – a framework that automatically estimates the potential dataflow parallelization in applications. We show how, using Tareador, one can find optimal taskification choice for any application through a trial-and-error iterative approach that requires no knowledge of the studied code. We prove the effectiveness of this approach on a case study in which we explore the taskification of Cholesky and High Performance Linpack. We show that the global partitioning significantly impacts parallel efficiency, and thus, in order to efficiently utilize higher number of cores, finer granularity of execution should be used.

Our next goal is to explore the potential for automating iterative approach that uses Tareador. The described iterative approach was successful, but it required significant interaction of the programmer. In the following Section, we try to formalize the programmer's experience into a set of policies that can autonomously lead the process of finding a good task decomposition.

6.3 Automatic exploration of potential parallelism

The previous Section described the iterative top-to-bottom approach that uses Tareador to find a suitable task decomposition of a code. Using very simple annotations, the programmer proposes some task decomposition of the studied sequential code. Then, Tareador automatically identifies potential parallelism of that decomposition. If the programmer is not satisfied with the achieved parallelism, he refines the last task decomposition and repeats Tareador instrumentation. The programmer iteratively tests different task decompositions, until finding one that provides sufficient parallelism.

The described manual iterative approach strongly relies on the programmer. The programmer must have sufficient experience to drive the iterative process of finding the optimal task decomposition. A programmer that has little experience with parallel programming may make wrong decisions in the iterative process of refining task decompositions. This decisions may lead to the final decomposition that is far from the optimal, resulting in the less efficient parallel execution. Also, the manual iterative approach requires frequent programmer interaction.

In this Section we go one step further and explore the possibility of automatic exploration of parallelization strategies. Our goal is to formalize good programmers' experience into simple metrics that can autonomously drive the above iterative top-down process. We believe that automatic exploration of parallelization strategies would bring the following benefits:

1. **Fast and autonomous exploration of parallelization strategies based on task decomposition**, with no major programmer interaction.

2. **Automatic estimation of the potential parallelism** with no major understanding of the sequential code, providing the programmer with hints on how to achieve that parallelism.

3. **Educating programmers about parallelism**, showing them which decisions, and in which order, are taken in order to exploit the inherent parallelism in the application.

6.3.1 The search algorithm

We propose an iterative algorithm to explore different task decomposition strategies and estimate their performance. The inputs are the original unmodified sequential code and the number of cores in the target platform. With that information the algorithm (Figure 6.21) performs of the following steps:

1. Start from the most coarse-grain task decomposition, i.e. the one that considers the whole main function as a single task.

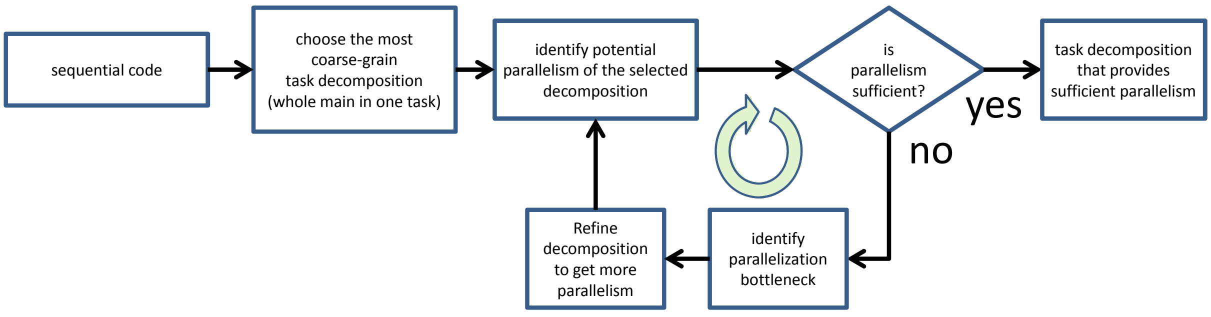

sequential code
choose the most coarse-grain task decomposition (whole main in one task)
identify potential parallelism of the selected decomposition
is parallelism sufficient?
yes
task decomposition that provides sufficient parallelism
no
identify parallelization bottleneck
Refine decomposition to get more parallelism

Figure 6.21: Algorithm for exploring possible task decompositions.

2. Perform an estimation of the potential parallelism of the current task decomposition (the speedup with respect to the sequential execution).

3. If the current task decomposition is satisfactory (heuristic 2), report it as final and finish.

4. Else, identify the parallelization bottleneck (heuristic 1) in the current task decomposition, i.e. the task that should be further decomposed into finer-grain tasks.

5. Refine the current task decomposition in order to avoid the identified bottleneck. Go to step 2.

In the following Sections, we further describe the design choices made in designing these two heuristics and the three metrics used. But before that, we must define more precise terminology. First, we need to make a clear distinction between a **task type** and a **task instance**. If function *compute* is encapsulated into task, we will say that *compute* is a **task type**, or just a **task**. Conversely, each instantiation of *compute* we will call a **task instance**, or just an **instance**. Then, if we can define some metric for each task instance, we can derive a collective metric for the whole task type.

Second, we will often use a term **breaking a task** to refer to the process of transforming one task into more fine-grain tasks. For example, Figure 6.22 illustrates the iterative task decomposition process. The process starts with the most coarse-grain decomposition ($D1$) in which function A is the only task. By breaking task A, we obtain decomposition $D2$ in which A is not a task anymore and instead the direct children (B and C) become tasks. If in the next step we break task B, since B contains no children

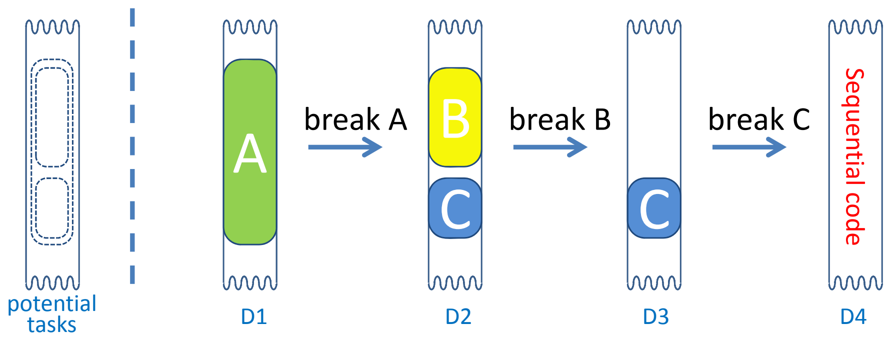


Figure 6.22: Iterative refinement of decompositions.

tasks, B will be serialized (i.e. B is not a task anymore and it becomes a part of the sequential execution). Similarly, the next refinement serializes task C and leads to the starting sequential code. At this point, no further refinement is possible, so the iterative process naturally stops.

Heuristic 1: which task to break

In the manual search for a satisfactory decomposition, the programmer himself decides which task is the parallelization bottleneck. Our goal is to formalize this programmer experience into simple metrics that can lead an autonomous algorithm for exploring potential task decompositions.

Metric 1: task length cost

A task type that has long instances is a potential parallelization bottleneck. Thus, based on the duration of instances, we define a metric called length cost of a task type. Length cost of some task type is proportional to the duration of the longest instance of that task. Therefore, if task i has task instances whose lengths are in the array T_i, the length cost of task i is:

$$l_i = \max(t), t \in T_i \tag{6.1}$$

Furthermore, we define a normalized length cost of task i as:

$$\overline{l_i(p)} = \frac{(l_i)^p}{\sum_{j=1}^{N} (l_j)^p} \tag{6.2}$$

where a control parameter p is used to tune the weight of this metric in the overall cost function.

Metric 2: task dependency cost

A task type that causes many data-dependencies is another potential parallelization bottleneck. Thus, based on the number of dependencies, we define a metric called dependency cost of a task type. Dependency cost of some task is proportional to the maximal number of dependencies caused by some instance of that task. Therefore, if task i has instances whose numbers of dependencies are in the array D_i, the dependencies cost of task i is:

$$d_i = \max(z), z \in D_i \tag{6.3}$$

Furthermore, using a control parameter p, we define the normalized dependency cost of task i as:

$$\overline{d_i(p)} = \frac{(d_i)^p}{\sum_{j=1}^{N} (d_j)^p} \tag{6.4}$$

Metric 3: task concurrency cost

A task type that has low concurrency is another potential parallelization bottleneck. Concurrency of some instance is determined by the number of other instances that execute in parallel with that instance. Thus, we define concurrency cost of some task to be inversely proportional to the number of instances that run concurrently with that task (number of instances that execute on all cores). Therefore, if task i has task instances which run for time $T_{i,j}$ while there are j instances running concurrently, the concurrency cost of task i is:

$$c_i = \sum_i \frac{T_{i,j}}{j} \tag{6.5}$$

Again, using a control parameter p, we define the normalized concurrency cost of task i as:

$$\overline{c_i(p)} = \frac{(c_i)^p}{\sum_{j=1}^{N} (c_j)^p} \tag{6.6}$$

Overall cost

The **cost function** for task type i is defined as the sum of the three previous normalized metrics

$$\overline{l_i(p_1)} + \overline{d_i(p_2)} + \overline{c_i(p_3)} \tag{6.7}$$

with different weights p_1, p_2 and p_3 for each metric.

Control parameter p

In all the defined metrics, the normalized cost is calculated using a control parameter p. For each metric separately, the sum of the normalized costs across all tasks is equal to 1. The parameter p additionally controls the mutual distance of the normalized costs for different tasks. For instance, let us assume that the applications consists of task instances A and B and that A is two times longer than B. If the control parameter p is equal to 1, task A has the length cost of 0.67, while task B has the length cost of 0.33. However, if the selected control parameter p is equal to 2, task A has the length cost of 0.8, while task B has the length cost of 0.2.

Therefore, the control parameter of some metric determines the impact of that metric on the overall cost. For instance, if we select the control parameter for length cost to be 0, all tasks will have the same normalized length cost, independent of the duration of instances of these tasks. Thus, the length of task instances would have no impact on the overall cost. On the other hand, if we select the control parameter for length cost to be infinite, the task with the longest instance will have the normalized length cost of 1, while all other tasks will have the normalized length cost of 0. In this case, the length of task instances would have a huge impact on the overall cost.

6.3.2 Heuristic 2: When to stop refining the decomposition

At this stage of our study, the goal of the automatic search is not to find the optimal decomposition, but rather to explore the broad space of possible decompositions. Claiming that some decomposition is optimal is dubious in itself, because there is no straightforward and complete comparison of decompositions. For instance, although some decomposition provides the highest parallelism on one machine, it does not necessarily provide the highest parallelism on a different machine. Thus, the goal of our algorithm is to explore various decompositions and find some that provide satisfactory parallelism. As already mentioned, the iterative refinement of decompositions naturally ends in the starting sequential code with no tasks. Having made the complete cycle, the optimal decomposition may be found by comparing all the tested decompositions.

6.3.3 Working environment

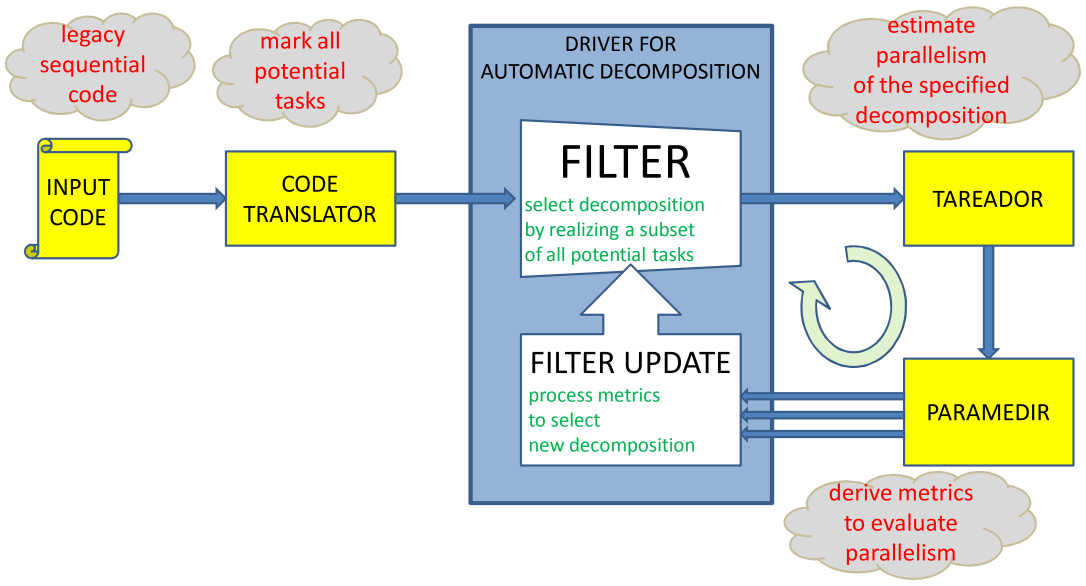

legacy sequential code
mark all potential tasks
DRIVER FOR AUTOMATIC DECOMPOSITION
estimate parallelism of the specified decomposition
INPUT CODE
CODE TRANSLATOR
FILTER
select decomposition by realizing a subset of all potential tasks
TAREADOR
FILTER UPDATE
process metrics to select new decomposition
PARAMEDIR
derive metrics to evaluate parallelism

Figure 6.23: The environment that automatically explores possible task decompositionss.

The environment for exploring potential decompositions consists of Mercurium code translator, Tareador, Paramedir and the search Driver. The Mercurium code translator marks in the code all the potential tasks – code sections that are suitable for tasks. The Driver sets a filter that decides which of the potential tasks should result into real OmpSs tasks, thus specifying one task decomposition of the code. Tareador evaluates the potential parallelism of the specified task decomposition, generating a Paraver trace of the potential parallel execution. Paramedir processes the obtained Paraver trace to derive metrics that describe simulated parallel execution. Based on the derived metrics, the Driver decides how to refine the task decomposition, updates the filter and starts new Tareador instrumentation. The following paragraphs further explain the role of each tool in the environment.

Code translation

Code translator processes a legacy sequential code and marks all potential tasks – all code sections that may represent a task (Figure 6.24). First, code translation marks the start and the end of the main function with calls to functions *tareador_start_main* and *tareador_end_main*. Furthermore, each function call is wrapped with calls to *tareador_start_function* and *tareador_end_function*. Finally, each loop in the original code is wrapped with calls to *tareador_start_loop* and *tareador_end_loop*. This way, the translation marks each code sections that potentially takes significant time to be a potential task in parallel execution.

Paramedir

Paramedir is a non-graphical user interface to the Paraver analysis module. Paramedir accepts the same trace and configuration files as Paraver. This way the same information can be captured in both systems. Still, while Paraver processes input and visualizes the results in graphical user interface, Paramedir reports the same results in textual for-

```c
 1 #define N 50
 2 #define M 10
 3
 4 float summarize (float a) {
 5    total = 0;
 6
 7
 8    for (j=0; j<N; j++){
 9       total += a[j]
10    }
11
12 }
13
14 int main () {
15
16
17
18    float A[M][N];
19    float sum = 0;
20
21
22    for (i=0; i<M; i++) {
23
24       sum += summarize (A[i]);
25
26    }
27
28
29
30 }
```

(a) original sequential code

```c
 1 #define N 50
 2 #define M 10
 3
 4 float summarize (float a) {
 5    total = 0;
 6
 7    tareador_start_loop("main.c:8");
 8    for (j=0; j<N; j++){
 9       total += a[j]
10    }
11    tareador_end_loop();
12 }
13
14 int main () {
15
16    tareador_start_main();
17
18    float A[M][N];
19    float sum = 0;
20
21    tareador_start_loop("main.c:22");
22    for (i=0; i<M; i++) {
23       tareador_start_function("summarize");
24       sum += summarize (A[i]);
25       tareador_end_function();
26    }
27    tareador_end_loop();
28
29    tareador_end_main();
30 }
```

(b) translated code

Figure 6.24: Code translation for automatic task decomposition.

mat. Avoiding the graphical user interface allows to translate the detailed human driven analysis into rules suitable for processing by an expert system.

Driver

The Driver is a python script that integrates all the previously described tools in a common environment. It receives as input the binary code and a list with the tasks that compose the current task decomposition. Initially the list just contains the main function of the program. The Driver automates the process of exploring potential decompositions by guiding the environment through the following steps:

1. Run Tareador to estimate the potential parallelism of the current decomposition.
2. If the current decomposition is satisfactory (Heuristic 2), finish the process and report the current decomposition as the final one.
3. Else, run Paramedir to derive the metrics and identify the bottleneck task (Heuristic 1).
4. Update the list of potential tasks in the Filter Update, by breaking the bottleneck task into its children task, if any.
5. Go to step 1.

6.3.4 Experiments

Our experiments explore possible parallelization strategies for four well-know computation kernels (Jacobi, HM transpose, Cholesky and LU factorization). We start from OmpSs parallel versions, removing all parallelization directives to generate a sequential version. The OmpSs reference task decomposition will be used to compare the task decompositions found by the proposed environment. We select a homogeneous multicore processor as the target platform. The goal of our experiment is to show that the proposed search algorithm, metrics and heuristics can find decompositions that provide sufficient parallelism.

In our experiments, we calculate the cost function as a sum of duration, dependency and concurrency cost (Section 6.3.1), tuning the control parameters to increase the weight of concurrency metrics. More specifically, we determine the cost of task type i as:

$$\overline{t_i} = \overline{l_i(1)} + \overline{d_i(1)} + \overline{c_i(3)} \tag{6.8}$$

The selected control parameters are not result of an extensive parametric study. Rather than that, our initial experiments showed that concurrency criteria prevails very rarely, thus we increased the concurrency control parameter as a positive discrimination measure.

For each application, we present four plots that illustrate the process of automatic task decomposition. The first plot presents the parallelism of all tested decompositions – the speedup over the sequential execution of the application. The second plot shows the number of task instances generated by each decompositions. Also, the first two plots show the parallelism and the number of instances in the default decomposition of the original OmpSs code. The third plot presents the cost distribution for the bottleneck task of each iteration. Finally, the fourth plot shows the most dominant cost for the bottleneck task.

Results

The proposed search algorithm finds decompositions with very high parallelism, often finding the reference decomposition from the original OmpSs code. The algorithm finds the reference decomposition for Jacobi and HM transpose in iterations 4 and 5, respectively (Figures 6.25 and 6.26). In these two applications, the algorithm bases its decisions strictly on the duration criteria. The algorithm also finds the reference decomposition for Cholesky in iteration 7 (Figure 6.27). However, in order to get to this decomposition, the algorithm refines decompositions based on the concurrency metric in iterations 3 and 5. In all three applications, after finding the reference decomposition, the algorithm passes through finer-grain decompositions that provide significantly lower parallelism.

Sparse LU (Figure 6.28), as the most complex of the studied applications, demonstrates the power of our search. Compared to the previous codes, Sparse LU forces the algorithm to use various bottleneck criteria through the exploration of decompositions. It is interesting to note that the search finds a wide range of decompositions (iterations 18 – 36) that provide higher parallelism than the default decomposition. In this case,

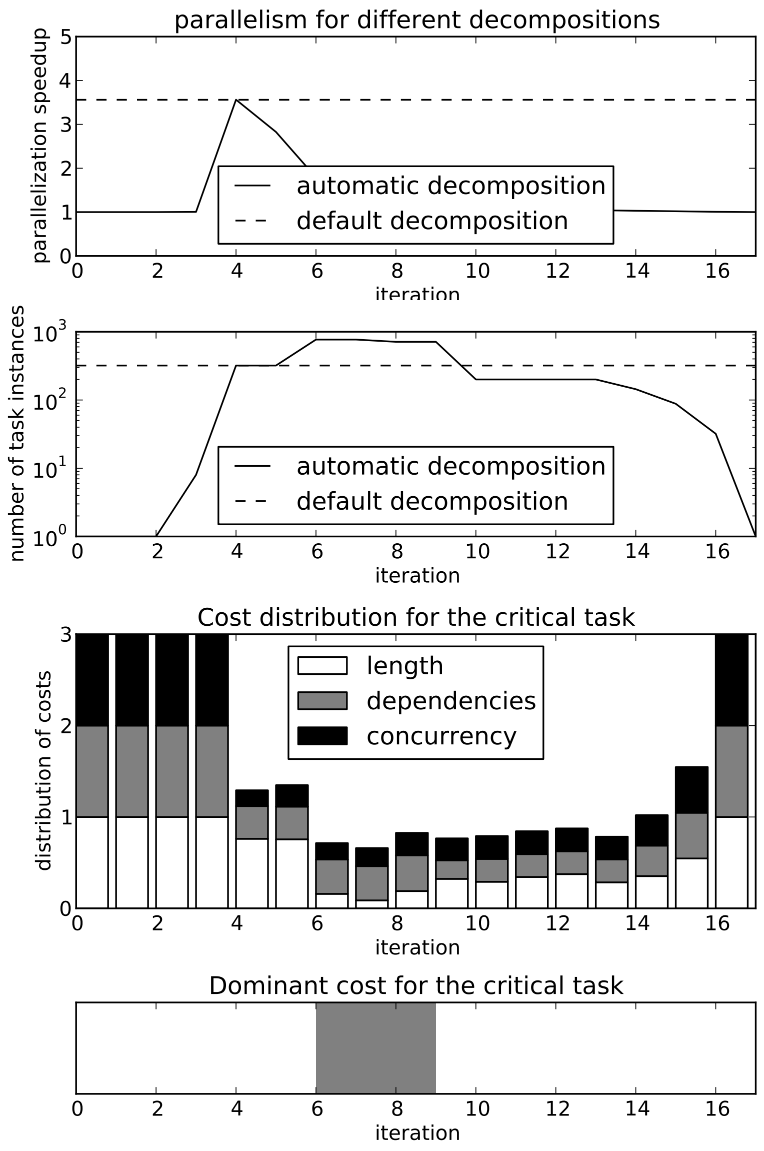


Figure 6.25: Jacobi on 4 cores

it is unclear which of these decompositions is the optimal one. Quantitative reasoning suggests that the optimal task decomposition is the one that provides highest parallelism with the lowest number of created task instances. Following this reasoning, the optimal decomposition (iteration 22) achieves the speedup of 3.98 with the cost of 301 instantiated tasks (note the sudden drop in the number of task instances). On the other hand, qualitative reasoning suggests that, within a set of decompositions that provide a

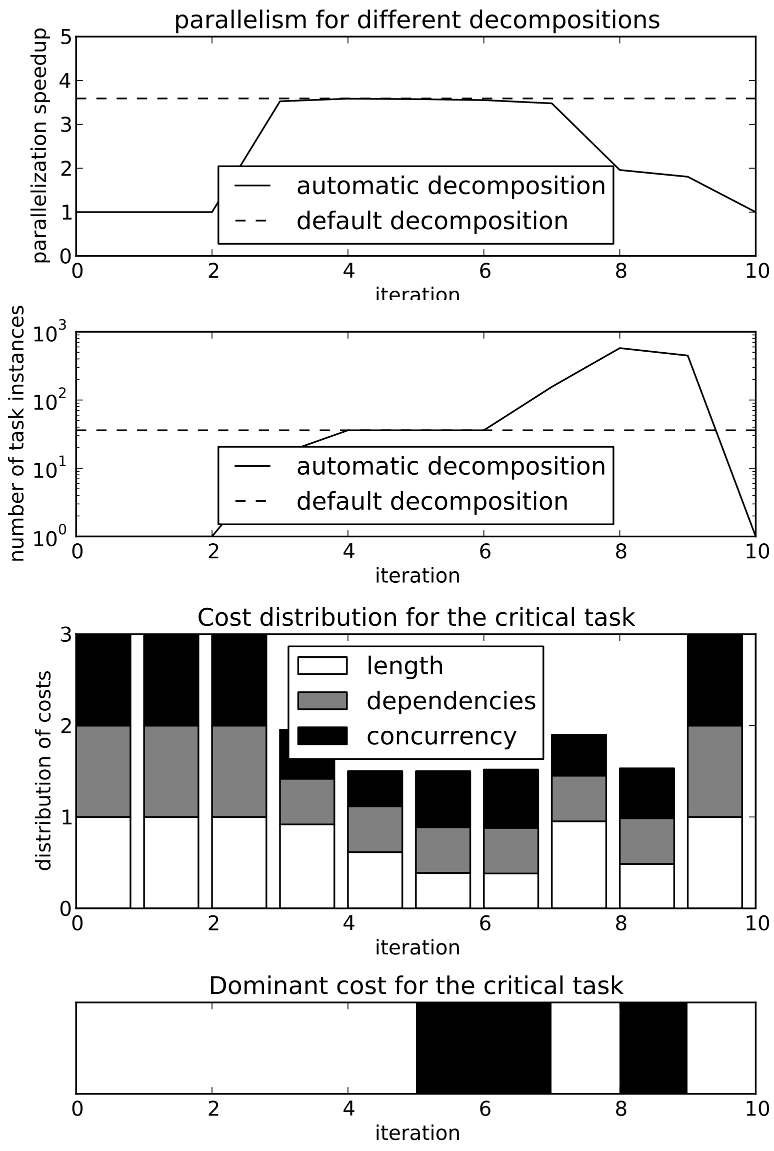


Figure 6.26: HM transpose on 4 cores

similar parallelism generating a similar number of instances, the optimal decomposition is the one that is the easiest to express using semantics offered by the target parallel programming model. At this stage of development, our algorithm is not capable of estimating how easy a decomposition can be expressed using some programming model. Our future research in this field should especially tackle this aspect of the algorithm.

It is also interesting to study how the algorithm adapts to the target parallel ma-

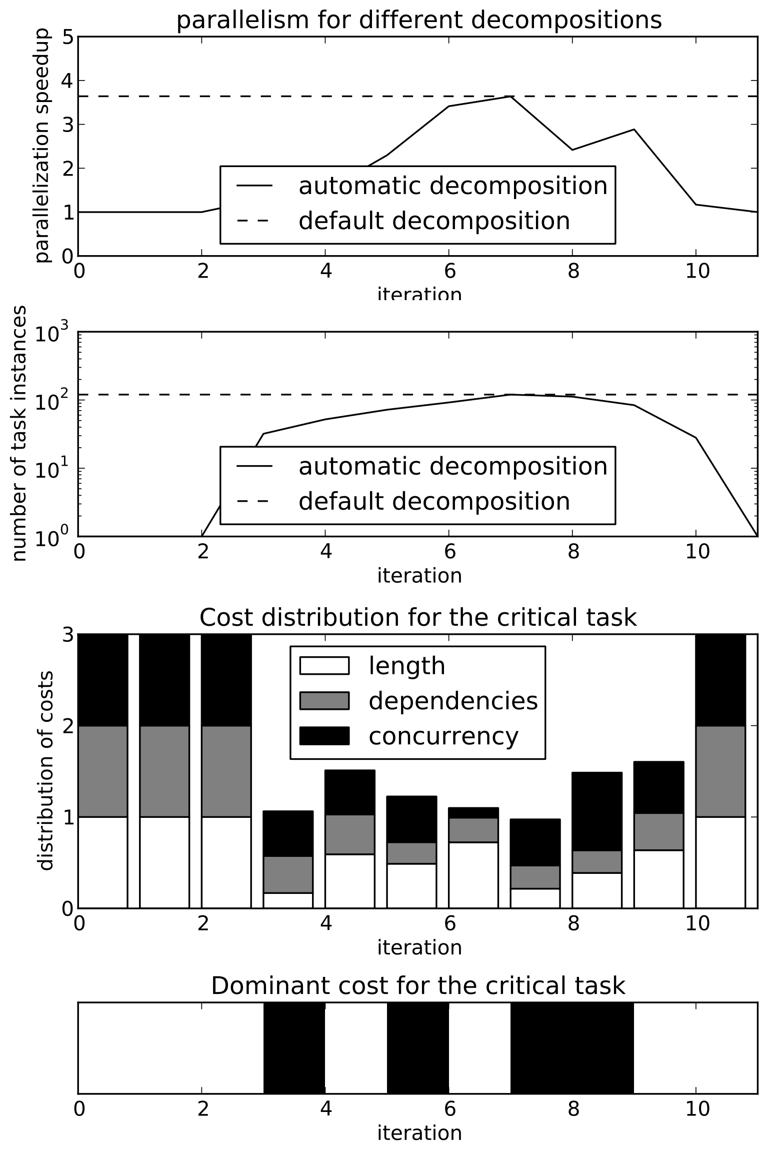


Figure 6.27: Cholesky on 4 cores

chine. Figures 6.29 and 6.30 illustrate potential decompositions for Sparse LU for executing on machines with 8 and 16 cores. Apparently, changing configuration of the target machine changes the algorithm decisions. In the experiments with 8-core target machine (Figure 6.29), the default decomposition achieves the speedup of 7.1 at the cost of generating 316 task instances. The automatic search finds a wide range of decompositions (iterations 20 – 44) that provide slightly higher parallelism than the

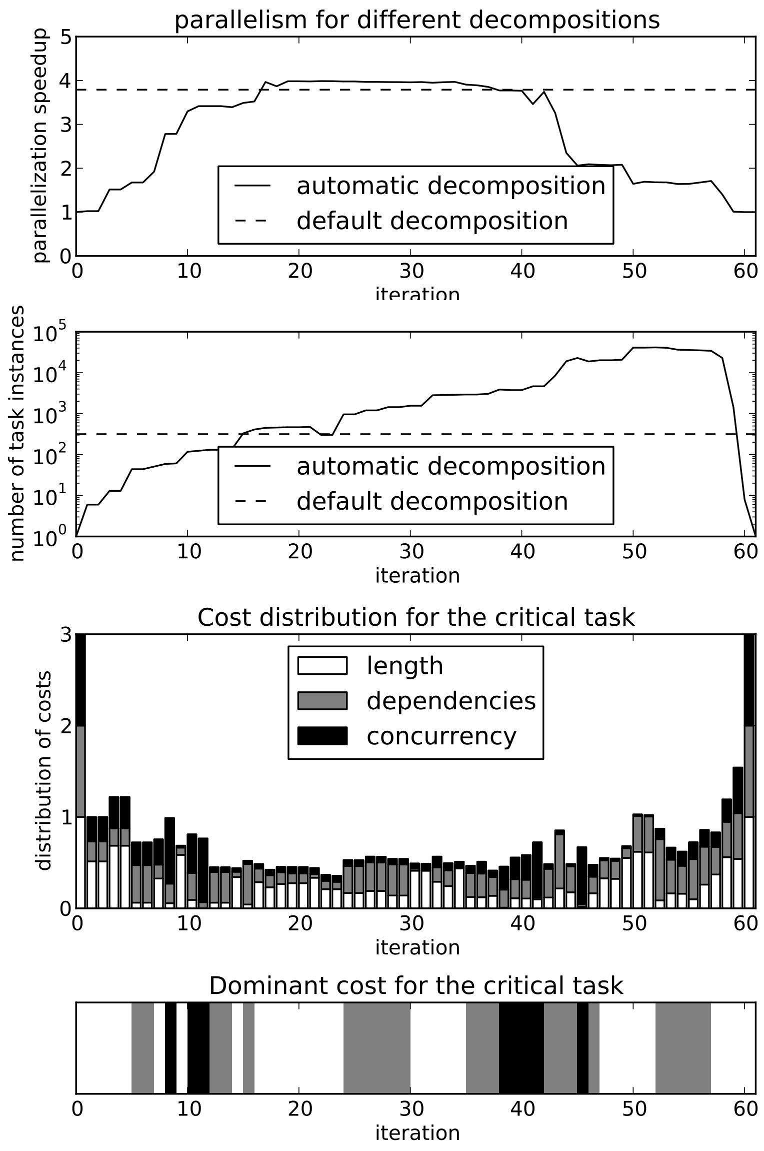


Figure 6.28: Sparse LU on 4 cores

default decomposition. Again, a sudden drop in number of task instances causes that the quantitatively optimal decomposition is in iteration 25 (parallelism 7.73, 325 task instances). On the other hand, in the experiments with 16-core target machine (Figure 6.30), the default decomposition achieves the speedup of 8.85 (316 instances). The algorithm finds only five decompositions (iterations 21 – 25) that provide higher parallelism than the default decomposition. The quantitatively optimal decomposition is

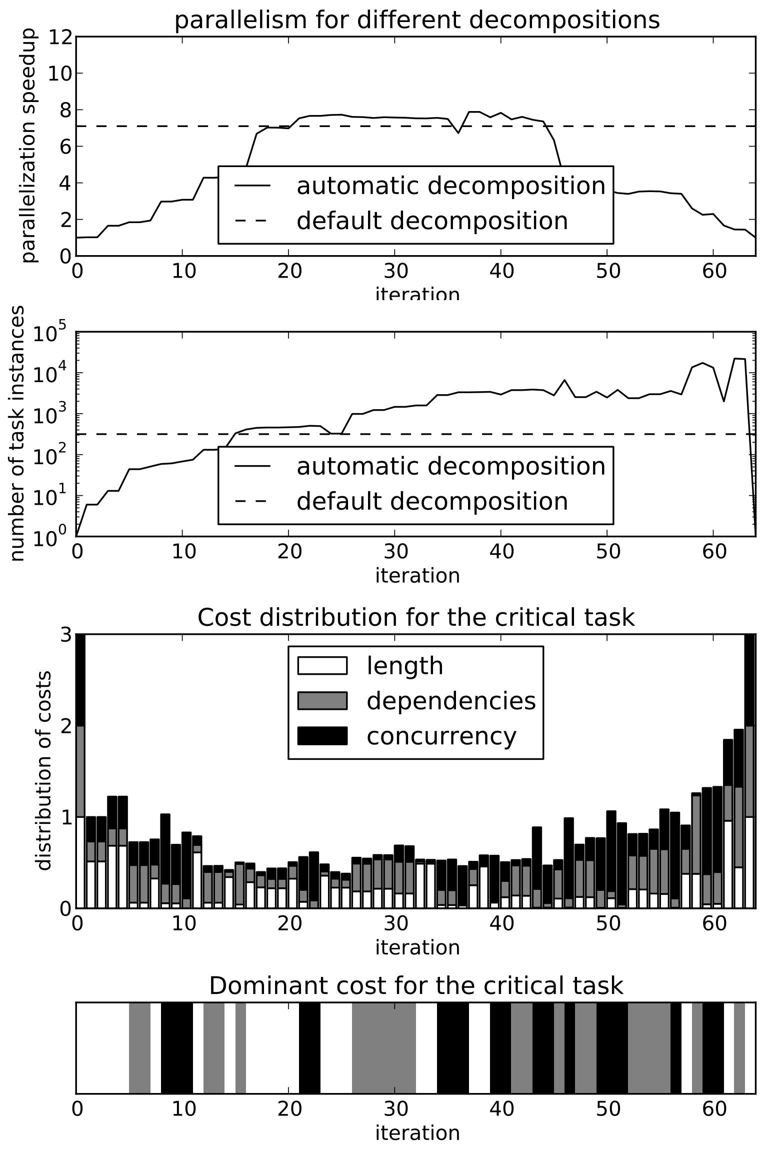


Figure 6.29: Sparse LU on 8 cores

found in iteration 25 (speedup 9.92, 453 task instances). It is also interesting to note a wide valley of low parallelism caused by decompositions from iterations 26 – 43. This happens because the algorithm, after finding the peak parallelism in iteration 25, aggressively tries to expose additional parallelism needed by the target 16-core machine. However, the selected code with the selected execution granularity offers no additional parallelism. Still, the algorithm identifies tasks that are culprits for low parallelism and

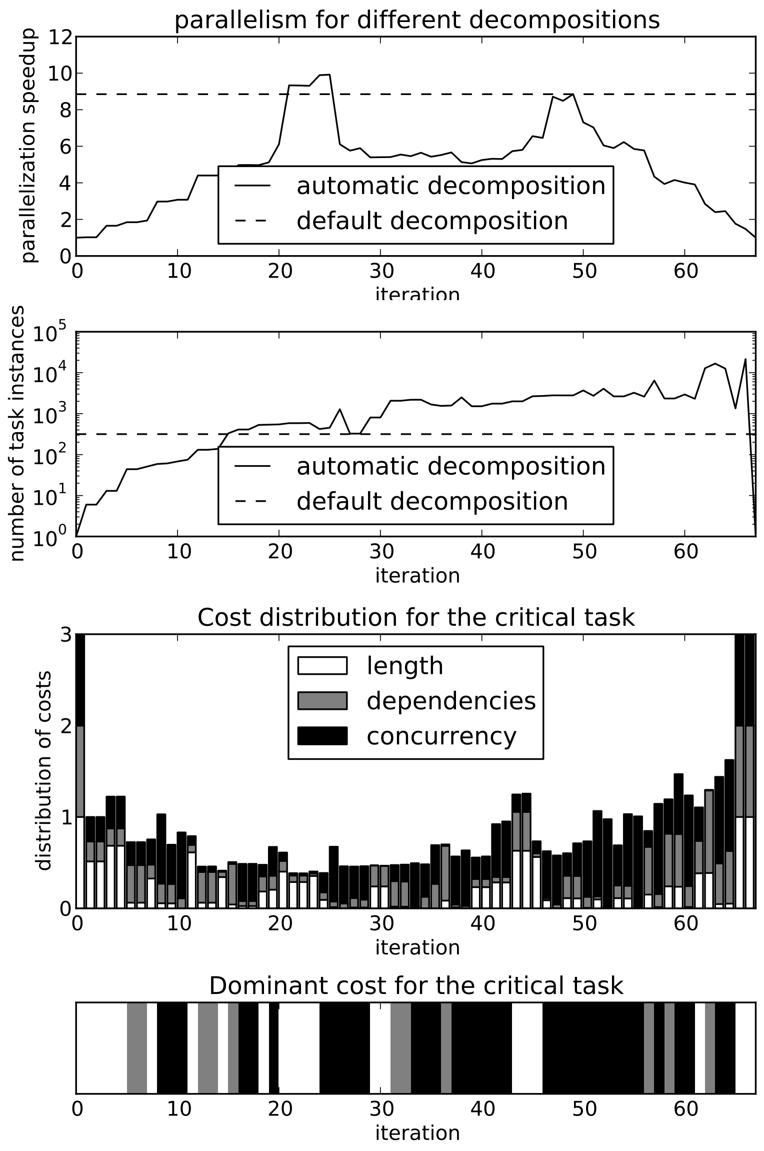


Figure 6.30: Sparse LU on 16 cores

continues refining them (in iterations 24 – 28, the refining is based on the concurrency criteria). The refining finds no additional parallelism, and furthermore diminishes the existing one. The refining also results in a significant drop in number of task instances (transition on iterations 25 – 26 causes the drop from 1285 to 453 instances).

6.3.5 Conclusion

We have explored the potential for the automatic exploration of task decomposition strategies in a sequential code. We have presented an effective search algorithm based on three simple metrics, a parametrizable cost function and a couple of heuristics. The cost function and metrics take into account the length (duration) of the tasks, the dependences among tasks and tasks' concurrency level. The search algorithm has been implemented leveraging a tool (Tareador) previously developed in the group. In our experiments, we demonstrate that our search algorithm is able to find task decompositions that provide sufficient parallelism, often higher than the parallelism of the decompositions specified by an experienced programmer.

As future work, we have identified the need to include a new metric that evaluates the cost of expressing the task decomposition using the syntax (and constraints) offered by the target parallel programming model (for example, traditional fork-join, data flow, ...). The automatic search should be able to quantify how expressible (or viable) a decomposition could be and to use this information to guide the iterative exploration process. The result would be the best task decomposition that can be expressed in the target programming model.

Our next step in this research is to try our methodology on real-world applications. So far, we worked in the fashion of reversed engineering – we start from a legacy OmpSs code and remove all OmpSs pragma annotations to obtain the sequential code. However, a sequential application obtained this way inherits structure that is very favorable for parallelization. Thus, starting from a legacy sequential code, automatic search for a good task decomposition would be much harder. Still, proving that our environment can explore parallelism in legacy sequential applications is the only proof of our concept. Therefore, that is our definite future work.

7 Related Work

In relation to the topics covered in this thesis, we present the related work in four distinct fields of research:

- Methodologies for simulating parallel execution
- Overlapping communication and computation
- Identifying bottlenecks in parallel execution
- Tools for assisted parallelization

7.1 Simulation methodologies for parallel computing

The simulation of parallel computing systems is still an unsolved issue. The simulation can be very computation intensive, since the target machine may consist of numerous processing units. Moreover, the simulation is hard to parallelize, because the separate processing units may have very complex interactions. Thus, simulating low level details in a large-scale parallel machine generates a very computation intensive sequential execution. Consequently, these simulations are often unfeasible, due to time or memory constraints.

The conventional trace-driven simulators successfully simulate MPI executions, but they fail to simulate multicore systems. Trace driven simulators, such as Dimemas [45] or MPI-SIM [71], simulate MPI parallel execution. They replay the collected traces and reconstruct the potential parallel time-behavior. However, the conventional traces fail to capture time-dependent executions – executions with dynamic thread scheduling and inter-thread synchronizations. Therefore, with the appearance of multicore systems, the trace-driven simulators failed to provide satisfactory simulations.

In order to simulate time-dependent execution, many recent studies turned to execution-driven simulators. These simulators provide cycle accurate simulations, capturing all possible time-dependent influences. Most of the current execution-driven simulators are based on the off-the-shelf simulation infrastructures such as M5 [14], Simics [60], Simplescalar [8], PTLsim [90], etc. These simulators introduce extremely high overhead. Therefore, for simulating a very large system, the execution-driven approach becomes unfeasible.

Finally, the newest simulation proposals tend to find a sweet spot between execution-driven and trace-driven approaches. COTSon [7] uses AMD functional emulator together with timing models, to achieve a proper combined timing. Compared to execution-driven simulators, COTSON reduces the simulation time, but also reduces simulation flexibility. On the other hand, TaskSim [74] tries to differentiate the applications intrinsic computation from the parallelism related computation. Then, the application intrinsic computation is replayed as in the conventional trace-driven simulation, while the parallelism related computation is recomputed during the simulation.

In this thesis, we introduced a novel simulation methodology called simulation aware tracing. Our methodology allows simulating very low-level architectural features in a large-scale parallel machine. This is enabled by modeling the introduced low-level feature already in the process of tracing. Since each MPI process is traced independently, the computation related to modeling a new feature is naturally parallelized across all MPI processes in the execution. The tracer includes the effects of the new feature into the trace, while the regular replay simulator replays the trace and spreads the effect of the modeled feature across the whole parallel MPI execution.

However, a drawback of our techniques is that the influence of the new feature can spread only bottom-up – a change at low-level can change the performance of parallel execution of MPI processes, but a change in parallel execution of MPI processes cannot change the performance at low-level. For instance, a change in cache performance can affect the scheduling of tasks. On the other hand, change in the scheduling of tasks, cannot affect the cache performance.

Our methodology deals with a very complicated parallel execution, offers fast and flexible simulation and provides a rich output. Up to our knowledge this is the first simulation methodology that can simulate parallel execution that integrates MPI with task-based programming model. Although our methodology originally targets dataflow parallelism, it can be easily adapted to simulate other fork-join based programming models such as OpenMP or Cilk. Finally, we believe that the biggest contribution of the environment is its flexibility and rich output. The user can easily change the target platform and visually (qualitatively) inspect the effects on the parallel execution.

7.2 Overlapping communication and computation

Previous research in the field of overlapping communication and computation could roughly be categorized in three directions. These are:

- exploring state-of-the-art support for exploiting overlap;
- exploring overlapping techniques; and
- measuring the potential for overlap that is present in applications.

First, several studies evaluated the overlapping capability of different processors, networks and programming languages. Sohn *et al.* [82] tested various multiprocessors and compared their overlapping efficiency. Furthermore, Brightwell *et al.* [19] quantified in detail the potential influence of overlap, offload and independent progress. Later work studied many MPI implementations and showed that their overlapping abilities are different [18, 58]. Further research [26, 27] explored the potential of PGAS languages to decouple communication and synchronization and achieve higher overlap. Furthermore, Bell *at al.* [10] showed overlapping advantages of light one-sided transfers implemented in UPC. Throughout our study, we assume that the used underlying communication layer is fully capable of overlapping communication and computation. Namely, our major goal is to identify the potential overlap inherent in the application.

Second, many research efforts explored implementation issues of overlapping techniques. In an effort to hide communication delays, Leu *et al.* [59] identified overlapping as a technique that can achieve the maximum application speedup of two. Later, Danalis *et al.* [33] defined general code restructuring approaches that lead to better overlap in applications that exhibit limited dependencies among iterations. Hoefler *et al.* [50] proved overlapping potential of non-blocking collective communications in MPI. Furthermore, Das *et al.* [34] introduced compiler features that postpone receptions (sink waits) in MPI applications, while Iancu *et al.* [52] extended UPC runtime library to implement demand-driven synchronization, automatic message strip mining and message scheduling. However, the mentioned efforts fail to clearly determine the potential benefits of their overlapping techniques, because they fail to isolate the overlapping effect from the implementations' side-effects such as changed locality (cache and TLB misses) and non-deterministic events (OS daemons, preemptions, interferences in a shared resource). On the other hand, our simulation can measure isolated impact of overlap, since the simulation framework introduces overlapping mechanisms without impacting other execution properties.

Third, there was little effort to identify the potential for overlap in applications. Sancho *et al.* [76] provided a theoretical estimation of the overlapping potential in scientific codes by modeling an application with one iterative loop and parameters that roughly describe the computation pattern. Our study continues Sancho's work by designing a simulation framework that automatically estimates the potential overlap, without the need to understand the studied application. Moreover, our methodology allows studying overlap on diverse network configuration and provides visualization support to qualitatively inspect the simulated execution. Our framework allowed us to analyze how overlap depends on application properties, the overlapping technique and network resources.

7.3 Identifying parallelization bottlenecks

Tracing is a common technique for identifying performance bottlenecks. In tracing, the environment instruments the parallel execution and collects a vast amount of performance data. Usually, the environment provides a visualization support that facilitates the programmer to explore the performance bottlenecks. Still, tracing rarely directly focuses the programmer's attention to the problem. The most wide-spread environments for tracing parallel executions are Jumpshot [91] from Argonne National Laboratory, Paraver [55] from Barcelona Supercomputing Center, Vampir [65] from TU Dresden and ScalaTrace project [67] from North Carolina State University.

Profiling is more efficient as it collects information throughout the execution and then reduces that information into a short report. The traditional profilers showed to be very useful for analyzing sequential execution. The most popular profiler, gprof [46], reports the percentage of computation time spent in each function. Identifying the most time consuming code section, gprof automatically finds the critical code section. However, in parallel executions, the most time consuming code section is not necessarily the critical code section. The most well-known parallel profilers are FPMPI-2 [57] from Argonne National Laboratory, mpiP [88] from Lawrence Livermore National Laboratory , HPCT [23] from IBM, TAU [79] from University of Oregon and Scalasca [44] from Julich Supercomputing Center.

Other techniques explore critical code sections by analyzing the critical path of execution. These techniques identify the critical code section as the code section that contributes the most to the overall critical path. The most popular representatives are Vtune [30] from Intel and Spartan [5] from University of Illinois. However, the analysis of the critical path omits the influence of the target parallel machine on which the application executes. Moreover, optimizing the code may shorten the starting critical path, and some other execution path may prevail, becoming the new critical path.

The newest approaches identify execution phases of low parallelism and then find culprit code sections. Quartz [6] and Intel Thread Profiler [17] [29] try to quantify the potential parallelism of each section of the code. These tools identify critical code sections as the ones with lowest parallelism. Similarly, Tallent *at. el.* [86] try to explore which code sections are responsible for processor stalls. Furthermore, this study considers different policies of spreading the blame for lost performance, from contexts in which spin-waiting occurs (victims), to directly blaming a lock holder (perpetrators). However, the drawback of all these techniques is that the conclusions obtained for one target platform can hardly be applied to a target platform with a different amount of parallelism.

Using our environment (Section 4.3) we show that the previous approaches cannot capture some influences that are very decisive in identifying critical code sections. First, that the choice of the critical section depends on the parallel target machine on which the application executes. Second, that depending on the factor of acceleration different sections may be the most beneficial to accelerate. Finally, all the previous techniques work with "measure-modify" approach – from the profile of one run the technique points to the critical section, the programmer optimizes the section, and then runs again the application "hoping" that the optimization resulted in overall speedup. Conversely, our approach provides anticipation of benefits – prior to any optimization effort, the programmer can estimate the potential benefits of the planned optimization.

7.4 Parallelization development tools

The multicore era created a rising interest for tools that help parallelizing applications. Multicores introduced the need to re-design (parallelize) applications in order to utilize the increased number of available cores. Despite decades of research efforts [12, 16, 89] on auto-parallelization, and the inclusion of auto-parallelization features in some commercial compilers [13], the experience witnesses very limited applicability. In the current scenario, in which systems (from mobile to desktop/laptop and servers) are based mostly on parallel architectures, programmers must use explicit parallel programming to reach the efficiency and scalability demands of future generations of software. However, the years of experience show manual parallelization without any development support has inadmissible cost. Therefore, the community started searching for the tools that assist the parallelization process.

Various academic research efforts tried to make tools that will help parallelization. For example, Alchemist tool [92] identifies parts of code that are suitable for thread-level speculation. Embla [61] is a Valgrind based tool that estimates the potential speed-up of Cilk parallelization. Kremlin [43] identifies regions of the serial program that need to be parallelized. Kremlin uses hierarchical critical path analysis to detect parallelism across nested regions of the program. Then, the parallelism planner evaluates various potential parallelizations to find the best way for the user to parallelize the target program. Starsscheck [21] checks correctness of pragma annotations for STARSs family of programming models. The biggest drawback of the mentioned tools is that they offer very little qualitative information about the target program. These tools are either checkers for different race conditions, or profiling tools that identify code sections that are good candidates for parallelization. None of the mentioned tools provides a visualization support. We believe that this is a major drawback, because in practice, parallelization often requires the programmer to restructure the sequential code in order to enable it for parallelization. However, the mentioned tools provide no guidelines that suggest to the programmer what restructuring is needed.

Also, various vendors started producing their solutions for assisted parallelization. These tools are available for trial usage, so we had an opportunity to test them. Intel's Parallel Advisor [28] helps parallelizing applications using Thread Building Blocks (TBB) [70]. Parallel Advisor provides the profile timing information based on which the programmer can choose which loops should be parallelized. Then, the tool runs an extensive instrumentation to provide additional information of how to complete TBB code, so the program can run in parallel. The drawback of Parallel Advisor is that the tool initially chooses loops only based on the timing profile, and not based on which loops are easy to parallelize. The Scottish company Critical Blue delivers a parallelization tool called Prism [15]. Prism allows the programmer to do "what if" analysis with the sequential program – to test the potential parallelism of any specified decomposition of the code. However, Prism does not target any specific parallel programming model and gives no hints of how the sequential code can be refactored into the parallel code. The Dutch company Vector Fabrics delivers a parallelization tool called Pareon [39]. Pareon also allows "what if" analysis but only for parallelizing loops. Pareon provides to the programmer additional hints that help parallelize the code using vfTasks library (a library developed by Vector Fabrics that is a wrapper for both pthread and Win32 threads). All the three mentioned tools provide very rich GUI and visualization of the potential parallelizations.

Our work tries to make a tool that will be a superset of the previously described efforts. In addition, we believe that a strategic decision to use OmpSs as the targeted programming model gives us a certain advantage, because the semantics of OmpSs matches the type of parallelism identified during the instrumentation. Currently, the main features of Tareador are:

- unbounded "what if" analysis (not limited only to loop iterations)

- visualization of the simulated execution
- estimation of the parallelism on a configurable target platform
- support for MPI applications
- automatic exploration of the potential decompositions

Fruthermore, we hope that in future we will provide the following new Tareador features:

- automatic generation of correct OmpSs (MPI/OmpSs) code
- visualization of objects usage
- integration of Tareador outputs (dependency graph, Paraver time-plots, objects visualization, Tareador logs)
- highly assisted parallelization (more parallelization hints to the programmer)

8 Conclusion

This thesis proposes techniques to improve performance in parallel applications without increasing the complexity of the parallel code. The starting point is the bulk-synchronous MPI programming model. Although widely used to program for very large-scale systems, the execution of bulk-synchronous programs introduces significant stalls in their parallel execution, mainly caused by load imbalance and excessive communication among processes. The causes for that performance degradation can be attacked at the expense of reducing programming productivity. The main objective of this thesis is to explore solutions that provide both high parallel performance and low programming complexity.

In the first part of this thesis, we target techniques for tuning MPI execution. More specifically, we explore the potential of communication/computation overlap. We proposed speculative dataflow – a hardware assisted technique that increases the overlap in applications and requires no intervention on the parallel code. Using a set of micro benchmarks, we demonstrated the feasibility and the effectiveness of this speculative technique. Furthermore, we designed an environment that automatically estimates the potential overlap in an MPI application. We showed that real-world scientific applications have a significant potential for overlap. However, the potential overlap is very limited by the application internal computation patterns. If the computation patterns are unprofitable, the overlap achieves only diminishing benefits. For one of the applications we show that code refactoring can be used to rearrange the computation patterns and increase the application's overlapping potential. However, we conclude that it is not productive, in terms of programming, to manually change the computation patterns for each application. Therefore, to increase the overlap, there is a need for an automatic way to restructure these patterns in order to increase potential overlap.

Since pure MPI execution showed significant lack of overlap, we turned to exploring hybrid parallel programming models. We decided to proceed with MPI/OmpSs programming model, because the potential asynchrony introduced by OmpSs could result in high overlap and deep lookahead. Therefore, the second part of this thesis explores two techniques for tuning MPI/OmpSs parallelism. Namely, we explore how MPI/OmpSs execution can be improved by optimizing some section of the code or by refining the task decomposition of the code.

In exploring the optimization opportunities in MPI/OmpSs execution, our goal was to identify the application's critical code section – the code section whose optimization would bring the highest benefit to the overall execution time. Identifying the critical section in sequential applications is trivial, because the critical section is always the most time consuming section. However, in parallel applications different sections contribute differently to the total parallel execution time. We showed that the choice of the critical section decisively depends on the target machine on which the application executes. For instance, we demonstrate that in HP Linpack, optimizing a task that takes 0.49% of the total computation time yields the overall speedup of less than 1% on a machine with 4 cores per node, and at the same time yields the overall speedup of more than 24% on a machine with 64 cores per node. Moreover, the choice of the critical section also depends on the optimization factor that will be applied. Finally, we designed a tool that predicts, for the targeted parallel machine and the given input, which section of the code should be optimized in order to get the highest overall execution speedup.

Second, we explored how the MPI/OmpSs parallelism could be tuned by selecting a different task decomposition of the code. Different task decompositions provide very different parallelization potential. For a programmer without any development support, it is very hard to anticipate which decomposition exposes parallelism and which not. To that end, we designed Tareador – a tool that estimates the potential parallelism of a task decomposition. Using Tareador, the programmer can easily test various task decompositions and quickly find one that exposes sufficient parallelism to keep the selected target machine efficiently utilized. Furthermore, we designed an autonomous driver that iteratively runs Tareador in the search for a good task decomposition. We show that by using very simple heuristics, the automatic search can find decompositions that provide very high parallelism.

Also, throughout the work on this thesis, we designed two development environments that can be useful to other researchers in the field. These environments are:

1. **mpisstrace** – an environment for replaying MPI/OmpSs parallel execution. The already existing BSC tool-chain allowed replaying MPI execution. We extended this infrastructure, in order to provide support for replaying MPI/OmpSs execution. Our changes are already included in the official distribution of BSC tools.

2. **Tareador** – a tool to help porting MPI applications to MPI/OmpSs. Tareador provides to the programmer a very simple and flexible interface to propose any task decomposition of the code. Then, Tareador dynamically instruments the target code, identifies data dependencies among the annotated tasks and reconstructs the potential parallel time-behavior. If the programmer is satisfied with the obtained parallelization, Tareador can provide further guidelines on how to complete the parallelization process (fill the pragma annotations). Tareador already proved itself useful by entering the undergraduate academic program at UPC. Our ongoing work is concentrated on developing computer logic that can automatically guide Tareador, so the required programmer's interaction could be further reduced.

8.1 Future work

For decades, microprocessors have been improving their performance following Moore's law without requiring major changes in the applications. The performance improvements relied on architectural techniques that improve ILP (instruction-level parallelism) exploitation and compilers that optimize the code for each target architecture. However, due to severe technological constraints, ILP performance gains entered stagnation. Consequently, multicore architectures appeared as the high-performance “promise land”. Multicores introduced the need to re-design (parallelize) applications in order to utilize the increasing number of available cores. As the software community struggles to fulfill this demand, the gap between parallel hardware and sequential software keeps growing. In the following subsection, I give my personal view on bridging this gap.

8.1.1 Parallelism for everyone: my view

The problem of multicore era is not in the fact that the existing applications are sequential, but rather in the fact that the existing programmers are sequential. The major issue in crossing the chasm between sequential software and parallel hardware is enabling existing programmers to write applications that can efficiently execute in parallel.

To that end, it is crucial to make parallel programming similar to sequential programming. Some mainstream parallel programming models require the programmer to change the structure of the sequential application. Thus, for MPI or pthreads, the programmer must make an explicit parallel structure of the code. On the other hand, OpenMP and OmpSs enable parallel execution, but seemingly maintain the structure of the sequential application. Maintaining the code structure is very important, because eases parallel programming for sequential programmers.

Maintaining the code structure also eases the task of automatic parallelization. In OpenMP and OmpSs, parallelizing the application consists of only adding pragmas that define the rules of parallel execution. Tareador can automatically search for a decomposition that exposes parallelism. Furthermore, we need to extend Tareador to continue searching for a decomposition that is easy to express with the semantics offered by the targeted parallel programming model. Then, after finding the optimal decomposition, Tareador should output the content of pragmas needed to generate the correct parallel code. It is important to note that during parallelization, Tareador maintains the structure of the sequential code, and just suggests where to put pragmas.

However, this type of automatic parallelization would often provide very limited parallelism. The search for a decomposition that provides parallelism could go very deep – resulting in a very fine-grain decomposition. This decomposition would be useless for the real run, because the execution would be dominated by the runtime overhead. Thus, the structure of the original sequential code must be changed in order to get more coarse-grain tasks that can still execute in parallel.

At this point, Tareador should advise the programmer how to change the access patterns of the original sequential code, in order for automatic parallelization to be more efficient. Tareador should provide visualization of the memory usage from the objects perspective. Using this visualization, the programmer could see how different tasks access different objects. Then, the programmer could learn how to restructure the sequential application, so the access patterns allow a more coarse-grain task decomposition of the sequential code.

In summary, the result of the described approach is that the programmer only writes the sequential code, while Tareador assures parallel execution of that code. The programmer only writes the sequential code and passes that code to Tareador for automatic parallelization. Tareador reports to the programmer the achieved parallelism. If the programmer is satisfied with the achieved parallelism, the parallelization process is finished. However, if the obtained parallelism is insufficient, Tareador provides to the programmer a set of hints how to rearrange the sequential application. After rearranging the sequential application, the programmer passes the updated code for new automatic parallelization. This process iterative repeats until the programmer is satisfied with the achieved parallelism. Once the code is parallelized, throughout further development of the code, the code maintaining responsibilities are again decoupled. The programmer guarantees the correct execution of the sequential execution – making sure that code updates did not harm the correctness of sequential execution. On the other hand, Tareador guarantees the correctness of parallel execution – making sure that code updates did not harm the concurrency rules among tasks. In conclusion, Tareador should teach the programmer how to write the sequential codes that have more potential for automatic parallelization.

This type of parallelization is not new. In vector processing, the programmer writes a sequential code while the compiler automatically extracts parallelism. The compiler identifies particular parts of the sequential program and transforms these parts into equivalent vectorized code. The compiler especially targets loops vectorization. However, due to possible dependencies among the loop iterations, the vectorization may lead to low performance gains. In this case, the compiler suggests the programmer how to restructure the loop in order to remove these dependencies. After the programmer eliminates dependencies among loop iteration, the compiler can parallelize the code more efficiently.

9 Publications

Conference papers:

Vladimir Subotić, Jesús Labarta, Mateo Valero.
Overlapping MPI Computation and Communication by Enforcing Speculative Dataflow.
INA-OCMC-08. Workshop on Interconnection Network Architectures On-Chip, Multi-Chip. Held in conjuction with HiPEAC-2008. Göteborg, Sweden, January 27-29, 2008

Vladimir Subotić, José Carlos Sancho, Jesús Labarta, Mateo Valero.
A Simulation Framework to Automatically Analyze the Communication-Computation Overlap in Scientific Applications.
Proceedings of the 2010 IEEE International Conference on Cluster Computing, Heraklion, Crete, Greece, 20-24 September, 2010

Vladimir Subotić, José Carlos Sancho, Jesús Labarta, Mateo Valero.

The Impact of Application's Micro-Imbalance on the Communication-Computation Overlap.
Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2011, Ayia Napa, Cyprus, 9-11 February 2011

Vladimir Subotić, Roger Ferrer, José Carlos Sancho, Jesús Labarta, Mateo Valero.
Quantifying the Potential Task-Based Dataflow Parallelism in MPI Applications.
Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, 29 August - 2 September, 2011, Proceedings, Part I

Vladimir Subotić, José Carlos Sancho, Jesús Labarta, Mateo Valero.
Identifying Critical Code Sections in Dataflow Programming Models.
Proceedings of the 21th International Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2013, Belfast, Northern Ireland, 27 February - 1 March 2011

Vladimir Subotić, José Carlos Sancho, Eduard Ayguadé, Jesús Labarta, Mateo Valero.
Automatic exploration of potential parallelism in sequential applications.
Submitted to Euro-Par 2013 Parallel Processing - 19th International Conference, Euro-Par 2013, Aachen, Germany, 2013

Journal papers:

Vladimir Subotić, Steffen Brinkmann, Vladimir Marjanović, Rosa M. Badia, Jose Gracia, Christoph Niethammer, Eduard Ayguade, Jesus Labarta, Mateo Valero
Programmability and portability for Exascale: Top Down Programming Methodology and Tools with StarSs.
Journal of Computational Science, Available online 11 February 2013, ISSN 1877-7503

Tutorials:

Eduard Ayguadé, Rosa M. Badia, Daniel Jiménez, Jesús Labarta, **Vladimir Subotić**.
SC12 HPC Educators session: Unveiling parallelization strategies at undergraduate level.
ACM/IEEE Conference on High Performance Computing, SC 2012, November 10-16, 2012, Salt Lake City, Utah, USA

Eduard Ayguadé, Rosa M. Badia and **Vladimir Subotić**.
SC13 HPC Educators session: Exploring parallelization strategies at undergraduate level.
Submitted to ACM/IEEE Conference on High Performance Computing, SC 2013, November 17-22, 2013, Denver, Colorado, USA

Posters:

Vladimir Subotić, Jesús Labarta, Mateo Valero.
Simulation environment for studying overlap of communication and computation.
IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, www.ispass.org, 28-30 March 2010, White Plains, NY, USA

Bibliography

[1] Graphviz - Graph Visualization Software, web-site confirmed active on 15.03.2013. URL http://www.graphviz.org/. 66

[2] NAS PARALLEL BENCHMARKS, web-site confirmed active on 15.03.2013. URL http://www.nas.nasa.gov/Resources/Software/npb.html. 85

[3] POP: Parallel Ocean Program, web-site confirmed active on 15.03.2013. URL http://climate.lanl.gov/Models/POP/. 85

[4] SWEEP3D: 3D Discrete Ordinates Neutron Transport , web-site confirmed active on 15.03.2013. URL http://wwwc3.lanl.gov/pal/software/sweep3d. 85

[5] Mayank Agarwal and Matthew I. Frank. SPARTAN: A software tool for Parallelization Bottleneck Analysis. In *Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, IWMSE '09, pages 56–63, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3718-4. doi: 10.1109/IWMSE.2009.5071384. URL http://dx.doi.org/10.1109/IWMSE.2009.5071384. 109, 151

[6] Thomas E. Anderson and Edward D. Lazowska. Quartz: a tool for tuning parallel program performance. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '90, pages 115–125, New York, NY, USA, 1990. ACM. ISBN 0-89791-359-0. doi: 10.1145/98457.98518. URL http://doi.acm.org/10.1145/98457.98518. 110, 151

[7] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. COTSon: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43:52–61, January 2009. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1496909.1496921. URL http://doi.acm.org/10.1145/1496909.1496921. 148

[8] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35:59–67, February 2002. ISSN 0018-9162. doi: 10.1109/2.982917. URL http://dl.acm.org/citation.cfm?id=619072.621910. 148

[9] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03868-6. doi: 10.1007/978-3-642-03869-3_79. URL http://dx.doi.org/10.1007/978-3-642-03869-3_79. 29

[10] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine A. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *IPDPS*. 2006. 149

[11] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a programming model for the cell BE architecture. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188546. URL http://doi.acm.org/10.1145/1188455.1188546. 29

[12] Siegfried Benkner. VFC: The Vienna Fortran Compiler. *Scientific Programming*, 7(1):67–81, 1999. 152

[13] Aart Bik, Milind Girkar, Paul Grey, and X. Tian. Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems, 2001. 152

[14] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 Simulator: Modeling Networked Systems.

IEEE Micro, 26:52–60, July 2006. ISSN 0272-1732. doi: 10.1109/MM.2006.82. URL http://dl.acm.org/citation.cfm?id=1158826.1159085. 148

[15] Critical Blue. Prism, web-site confirmed active on 15.03.2013. URL http://www.criticalblue.com/criticalblue_products/prism.shtml. 153

[16] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David A. Padua, Yunheung Paek, William M. Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel Programming with Polaris. *IEEE Computer*, 29(12):87–81, 1996. 152

[17] Clay Breshears. Using Intel Thread Profiler for Win32 threads: Philosophy and theory. web-site confirmed active on 15.03.2013. URL http://software.intel.com/en-us/articles/using-intel-thread-profiler-for-win32-threads-philosophy-and-theory/. 110, 151

[18] Ron Brightwell, Keith D. Underwood, and Rolf Riesen. An Initial Analysis of the Impact of Overlap and Independent Progress for MPI. In *PVM/MPI*, pages 370–377. 2004. 149

[19] Ron Brightwell, Rolf Riesen, and Keith D. Underwood. Analyzing the Impact of Overlap, Offload, and Independent Progress for Message Passing Interface Applications. *IJHPCA*, 19(2):103–117, 2005. 22, 149

[20] William W. Carlson, Jesse M. Draper, and David E. Culler. S-246, 187 Introduction to UPC and Language Specification. 16

[21] Paul M. Carpenter, Alex Ramírez, and Eduard Ayguadé. Starsscheck: A Tool to Find Errors in Task-Based Parallel Programs. In *Euro-Par (1)*, pages 2–13, 2010. 152

[22] Laura Carrington, Dimitri Komatitsch, Michael Laurenzano, Mustafa M. Tikir, David Michéa, Nicolas Le Goff, Allan Snavely, and Jeroen Tromp. High-frequency simulations of global seismic wave propagation using SPECFEM3D_GLOBE on 62K processors. In *SC*, page 60. 2008. 85, 116

[23] IBM Advanced Computing Technology Center. High Performance Computing Toolkit., web-site confirmed active on 15.03.2013. URL http://www.ibm.com. 151

[24] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel Programmability and the Chapel Language. *IJHPCA*, 21(3):291–312, 2007. 16

[25] 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, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094852. URL http://doi.acm.org/10.1145/1094811.1094852. 16

[26] Wei-Yu Chen, Costin Iancu, and Katherine A. Yelick. Communication Optimizations for Fine-Grained UPC Applications. In *IEEE PACT*, pages 267–278. 2005. 149

[27] Cristian Coarfa, Yuri Dotsenko, Jason Eckhardt, and John M. Mellor-Crummey. Co-array Fortran Performance and Potential: An NPB Experimental Study. In *LCPC*, pages 177–193. 2003. 149

[28] Intel Corporation. Intel Parallel Advisor, web-site confirmed active on 15.03.2013. URL http://software.intel.com/en-us/intel-advisor-xe. 153

[29] Intel Corporation. Intel thread profiler, web-site confirmed active on 15.03.2013. URL http://software.intel.com/file/17321. 110, 151

[30] Intel Corporation. Intel VTune Amplifier XE, web-site confirmed active on 15.03.2013. URL http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/. 109, 151

[31] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computing in Science and Engineering*, 5:46–

55, 1998. ISSN 1070-9924. doi: http://doi.ieeecomputersociety.org/10.1109/99.660313. 16, 22, 29

[32] William James Dally. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2004. 2

[33] Anthony Danalis, Ki-Yong Kim, Lori L. Pollock, and D. Martin Swany. Transformations to Parallel Codes for Communication-Computation Overlap. In *SC*, page 58. 2005. 150

[34] Dibyendu Das, Manish Gupta, Rajan Ravindran, W. Shivani, P. Sivakeshava, and Rishabh Uppal. Compiler-controlled extraction of computation-communication overlap in MPI applications. In *IPDPS*, pages 1–8, 2008. 150

[35] Narayan Desai, Pavan Balaji, P. Sadayappan, and Mohammad Islam. Are nonblocking networks really needed for high-end-computing workloads? In *CLUSTER*, pages 152–159, 2008. 2, 40

[36] Jack Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003. 116

[37] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21 (2):173–193, 2011. 16, 27

[38] Ryan Eccles, Blair Nonneck, and Deborah A. Stacey. Exploring Parallel Programming Knowledge in the Novice. In *HPCS*, pages 97–102, 2005. 3

[39] Vector Fabrics. Pareon, web-site confirmed active on 15.03.2013. URL `http://www.vectorfabrics.com/products`. 153

[40] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Memory - Sequoia: programming the memory hierarchy. In *SC*, page 83, 2006. 105

[41] Felix Freitag, Jordi Caubet, Montse Farreras, Toni Cortes, and Jesús Labarta. Exploring the Predictability of MPI Messages. In *IPDPS*, page 69. 2003. 78

[42] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*, pages 212–223, 1998. 29

[43] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *PLDI*, pages 458–469, 2011. 152

[44] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 22(6):702–719, April 2010. ISSN 1532-0626. doi: 10.1002/cpe.v22:6. URL http://dx.doi.org/10.1002/cpe.v22:6. 109, 151

[45] Sergi Girona, Jesús Labarta, and Rosa M. Badia. Validation of Dimemas Communication Model for MPI Collective Operations. In *PVM/MPI*, pages 39–46. 2000. 37, 148

[46] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM. ISBN 0-89791-074-5. doi: 10.1145/800230.806987. URL http://doi.acm.org/10.1145/800230.806987. 109, 151

[47] 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. 10, 11

[48] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin J. Lang, Satish Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. BSPlib: The BSP programming library. volume 24, pages 1947–1980. 1998. 40

[49] Lorin Hochstein, Jeffrey Carver, Forrest Shull, Sima Asgari, and Victor R. Basili. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. In *SC*, page 35, 2005. 3

[50] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *SC*, page 52, 2007. 17, 150

[51] Guillaume Houzeaux, Beatriz Eguzkitza, and Mariano Vazquez. A variational multiscale model for the advection-diffusion-reaction equation. *Communications in Numerical Methods in Engineering*, 2007. 85

[52] Costin Iancu, Parry Husbands, and Paul Hargrove. HUNTing the Overlap. In *IEEE PACT*, pages 279–290. 2005. 150

[53] James Christopher Jenista, Yong Hun Eom, and Brian Demsky. OoOJava: software out-of-order execution. In *PPOPP*, pages 57–68, 2011. 105

[54] David J. Kuck, Robert Henry Kuhn, David Padua, Bruce Leasure, and Michael Joseph Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 207–218, New York, NY, USA, 1981. ACM. ISBN 0-89791-029-X. doi: 10.1145/567532.567555. URL `http://doi.acm.org/10.1145/567532.567555`. 28

[55] Jesús Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, and Luis Gregoris. DiP: A Parallel Program Development Environment. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, Euro-Par '96, pages 665–674, London, UK, UK, 1996. Springer-Verlag. ISBN 3-540-61627-6. URL `http://dl.acm.org/citation.cfm?id=646669.701233`. 36, 108, 151

[56] Vincent Pillet Jesus Labarta, Todi Cortes, and Sergi Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. In *WoTUG-18*. 1995. 37

[57] Argonne National Laboratory. The FPMPI-2 MPI profiling library., web-site confirmed active on 15.03.2013. URL `http://www-unix.mcs.anl.gov/fpmpi/`. 109, 151

[58] William Lawry, Christopher Wilson, Arthur B. Maccabe, and Ron Brightwell. COMB: A Portable Benchmark Suite for Assessing MPI Overlap. In *CLUSTER*, pages 472–475. 2002. 149

[59] JaSong Leu, Dharma Prakash Agrawal, and Jon Mauney. Modeling of parallel software for efficient computation-communication overlap. In *Fall Joiny Computer Conference*. IEEE Press, 1987. 150

[60] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2): 50–58, 2002. 148

[61] Jonathan Mak, Karl-Filip Faxén, Sverker Janson, and Alan Mycroft. Estimating and Exploiting Potential Parallelism by Source-Level Dependence Profiling. In *Euro-Par (1)*, pages 26–37, 2010. 152

[62] Vladimir Marjanovic, Jose Maria Perez, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach. In *UPC-DAC-RR-2009-35*, Research Report, Technical University of Catalunya, 2009. 33

[63] Vladimir Marjanovic, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *ICS*, pages 5–16, 2010. 30, 31, 105

[64] Shirley Moore, David Cronk, Felix Wolf, Avi Purkayastha, Patricia Teller, Robert Araiza, Maria Gabriela Aguilera, and Jamie Nava. Performance Profiling and Analysis of DoD Applications Using PAPI and TAU. In *Proceedings of the 2005 Users Group Conference on 2005 Users Group Conference*, DOD_UGC '05, pages 394–, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2496-6. 36

[65] Wolfgang E. Nagel. VAMPIR: Visualization and Analysis of MPI Resources. 1996. URL http://books.google.es/books?id=LnCmPgAACAAJ. 37, 108, 151

[66] Nicholas Nethercote and Julian Seward. Valgrind, web-site confirmed active on 15.03.2013. URL http://valgrind.org/. 36

[67] Michael Noeth, Jaydeep Marathe, Frank Mueller, Martin Schulz, and Bronis de Supinski. Scalable compression and replay of communication traces in massively parallel environments. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188605. URL http://doi.acm.org/10.1145/1188455.1188605. 108, 151

[68] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008. ISBN 0123744938, 9780123744937. 9

[69] Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *CLUSTER*, pages 142–151. 2008. 29, 105

[70] Chuck Pheatt. Intel threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, apr 2008. ISSN 1937-4771. URL http://dl.acm.org/citation.cfm?id=1352079.1352134. 153

[71] Sundeep Prakash and Rajive L. Bagrodia. MPI-SIM: Using Parallel Simulation To Evaluate MPI Programs, 1998. 148

[72] SPIRAL project. Software/Hardware Generation for DSP Algorithms, web-site confirmed active on 15.03.2013. URL http://www.spiral.net/problem.html. x, 10

[73] Milos Prvulovic, Josep Torrellas, and Zheng Zhang. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *ISCA*, pages 111–122. 2002. 78

[74] Alejandro Rico, Alejandro Duran, Felipe Cabarcas, Yoav Etsion, Alex Ramírez, and Mateo Valero. Trace-driven simulation of multithreaded applications. In *ISPASS*, pages 87–96, 2011. 148

[75] Luiz De Rose, Bill Homer, Dean Johnson, Steve Kaufmann, and Heidi Poxon. Cray Performance Analysis Tools. In *Parallel Tools Workshop*, pages 191–199, 2008. 109

[76] José Carlos Sancho, Kevin J. Barker, Darren J. Kerbyson, and Kei Davis. MPI tools and performance studies - Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *SC*, page 125. 2006. 84, 92, 150

[77] Robert R. Schaller. Moore's law: past, present, and future. *IEEE Spectr.*, 34 (6):52–59, June 1997. ISSN 0018-9235. doi: 10.1109/6.591665. URL `http://dx.doi.org/10.1109/6.591665`. 9

[78] John Paul Shen and Mikko H. Lipasti. *Modern processor design : fundamentals of superscalar processors*. McGraw-Hill Higher Education, Boston, 2005. ISBN 0-07-057064-7. URL `http://opac.inria.fr/record=b1129703`. Index. 15

[79] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006. ISSN 1094-3420. doi: 10.1177/1094342006064482. URL `http://dx.doi.org/10.1177/1094342006064482`. 109, 151

[80] James Smith and Gurindar S. Sohi. The Microarchitecture of Superscalar Processors. 1995. URL `citeseer.ist.psu.edu/35243.html`. 28

[81] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. MPI: The Complete Reference, web-site confirmed active on 15.03.2013. URL `http://www.netlib.org/utk/papers/mpi-book/mpi-book.html`. 15, 16

[82] Andrew Sohn, Jui Ku, Yuetsu Kodama, Mitsuhisa Sato, Hirofumi Sakane, Hayato Yamana, Shuichi Sakai, and Yoshinori Yamaguchi. Identifying the capability of overlapping computation with communication, 1996. 149

[83] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *SC*, 2009. 105

[84] Vladimir Subotic, Roger Ferrer, José Carlos Sancho, Jesús Labarta, and Mateo Valero. Quantifying the Potential Task-Based Dataflow Parallelism in MPI Applications. In *Euro-Par (1)*, pages 39–51, 2011. 105

[85] Nathan R. Tallent and John M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *PPOPP*, pages 229–240, 2009. 110

[86] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *PPOPP*, pages 269–280, 2010. 110, 151

[87] top500. Top500 List: List of top 500 supercomputers., web-site confirmed active on 15.03.2013. URL `http://www.top500.org/`. 2, 116

[88] J. Vetter and C Chambreau. The mpiP MPI profiling library., web-site confirmed active on 15.03.2013. URL `http://mpip.sourceforge.net/`. 109, 151

[89] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Notices*, 29(12):31–37, 1994. 152

[90] Matt T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS 07*. 2007. 148

[91] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward Scalable Performance Visualization with Jumpshot. *Int. J. High*

Perform. Comput. Appl., 13(3):277–288, August 1999. ISSN 1094-3420. doi: 10.1177/109434209901300310. URL http://dx.doi.org/10.1177/109434209901300310. 108, 151

[92] Xiangyu Zhang, Armand Navabi, and Suresh Jagannathan. Alchemist: A Transparent Dependence Distance Profiling Infrastructure. In *CGO*, pages 47–58, 2009. 152

[93] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *ISCA*, pages 224–237. 2004. 83