The MPI/OmpSs Parallel Programming Model

Vladimir Marjanović

Department of Computer Architecture

Universitat Politècnica de Catalunya

Supervisor: Jesùs Labarta
Co-supervisor: Josè Gracia

Doctor of Philosophy in Computer Architecture

11th of November, 2015

This page is intentionally left empty.

Abstract

Even today supercomputing systems have already reached millions of cores for a single machine, which are connected by using a complex network interconnection. Reducing communication time across processes becomes the most important issue in order to achieve the highest possible performance. The Message Passing Interface (MPI), which is the most widely used programming model for large distributed memory, supports asynchronous communication primitives for overlapping communication and computation. However, these primitives are difficult to use and increase code complexity. which then requiring more development effort and making less readable programs.

This thesis presents a new programming model, which allows the programmer to easily introduce the asynchrony necessary to overlap communication and computation. The proposed programming model is based on MPI and tasked based shared memory framework, namely OmpSs.

The thesis further describes implementation details which in order to allow efficient inter-operation of the OmpSs runtime and MPI. The thesis demonstrates the hybrid use of MPI/OmpSs with several applications of which the HPL benchmark is the most important case study. The hybrid MPI/OmpSs versions significantly improve the performance of the applications compared with their pure MPI counterparts. For the HPL we get close to the asymptotic performance at relatively small problem sizes and still get significant benefits at large problem sizes. In addition, the hybrid MPI/OmpSs approach substantially reduces code complexity and is less sensitive to network bandwidth and operating system noise than the pure MPI versions.

In addition, the thesis analyzes and compares current techniques for overlapping computation and collective communication, including approaches using point-to-point communications and additional communication threads, respectively. The thesis stresses the importance of understanding the characteristic of a computational kernel that runs concurrently with communication. Experimental evaluations is done using the Communication Computation Concurrent (CCUBE) synthetic benchmark, developed in this thesis, as well as the HPL.

Contents

List of Figures

List of Tables

1 Introduction

Scientists and engineers recognize that computers can help solving problems that would be too complex and expensive with other methods. Computational modeling and simulation of these problems became a new methodology and standard tool in academia and industry. Dealing with these applications requires powerful systems in terms of raw cpu speed, main memory and data store. A single workstation as a desktop machine would never fulfill these requirements due to size of problem. Demand for high performance parallel machines is a logical step in order to address these issues. High-Performance Computing (HPC) systems delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business. The HPC systems with a very high-level computational capacity, also called supercomputer, could efficiently deal with a large non-embarrassingly parallel problems of earth and life since.

In contrast to large data center, an HPC machine contains thousands of processors and complex interconnection networks between these processors. Raw computational power of large data centers, e.g. Amazon data center [23] cannot processes large non-embarrassingly problems within reasonable time-frame, due to low performance of interconnection network. This lack of high bandwidth and low latency interconnection network is the main difference between computational data centers and HPC systems. Efficient hardware and software approaches of communication between processors are the biggest challenges for computer scientist in HPC world.

Hardware of HPC system is very robust, expensive and complex and requires a software stack for users with specialized expertise to use. Regarding interconnection network, hardware designers try to minimize latency and maximize bandwidth, while software designers try to exploit hardware properties as efficiently as possible and minimize communication and synchronization cost between processors. Complexity of HPC systems grows constantly and programming effort become expensive in terms of time to solution, pushing users to understand hardware and software of machines. Software layer should provide easy to use interface for programmers and keep high efficiency of HPC systems.

To tackle productivity, in this thesis we explore software techniques for efficient communication between processors and propose a hybrid programming model based task based programming model OmpSs and the distributed programming model MPI. The programming model offers high performance and high productivity at the same time. Finding well balanced ratio between programability/performance, porting time for already existing codes, and portability drive the work of the thesis to the MPI/OmpSs programming approach as a future tool for scientist.

1.1 Goals

HPC aims a high performance, thus the main goal of this thesis is to propose a programming model that achieves high efficiency. While a singe thread performance execution leans on compiler optimization and a single thread optimization techniques, parallel programming focuses on efficiency of parallel execution. The following challenges are addressed by this work:

- *Increase Parallelism* - making all available computing resources busy is the first step to high performance. Parallel programming models should offer a flexible execution model, where idle states of computing units are minimize. Introducing a term *'degree of resource utilization'* as the ratio between the number of independent task and a number of computing units. A degree of resource utilization greater than 1 leads to the perfect parallelism. The goal is to keep the degree as high as possible in every moment of execution.

- *Scalability* - The goal of programming models is customization and a scalable execution model, that matches complex node architecture and schedules task in a fashion that N, cores speed up N times sequential execution

- *Hiding communication overhead* - Tasks need communication operations, accelerating and hiding communication operation is the key. The communication overhead is the biggest challenge for parallel programming paradigm and capability of programming model to minimize communication costs determines its efficiency. The main goal of the thesis is to reduce communication cost during execution.

- *Tolerance to network contention* - HPC systems contain powerful interconnection network, in order to move data across distributed nodes and fulfill task dependencies. Nevertheless, nonuniform utilization of network stresses interconnection throughput and slows down execution. The programming model should distribute communication over time, relax the interconnection network while overall performance does not suffer.

- *Tolerance to OS jitters* - driving applications across large number of cores increase possibility that unpredictable performance drops might appear on any core caused by system noise. The performance drop of one thread propagates to overall performance and lowers efficiency. The goal of parallel execution modeling is to filter non-deterministic noise and smoothly muffle performance degradation.

- *Code Complexity* - once the parallel programming model shows a successful improvement in term of performance, programming effort stays a big challenge. There is no clear metric that evaluates programming productivity and ranks parallel programming approaches. Programmers usually focus on expressing an algorithm, once the code gives correct results tuning optimization phase take place. Improving performance increases code complexity while new radical programming approaches discharge programmers to learn them.The goal of the programming model is a friendly, easy-to-use interface, where codes stays as close as possible to an initial version where the algorithm has been described. The syntax of a programming model should be as natural as possible.

The overall goal of the thesis aims at a future way of thinking regarding parallel programming on HPC systems,both for, programmers and programming model developers.

1.2 Methodolgy

Proposing a future programming model requires review of state-of-the art programming model. De facto, Message Passing Interface (MPI), is the most accepted programming model for distributed memory systems. Almost all applications have been written using the MPI paradigm. MPI is a library that can be used with C/C++ or FORTRAN. MPI forms a basis for the thesis and can be considered as a one of the low level programming layer for the proposed programming model.

On shared memory, level the most used parallel programming approach is OpenMP annotations of C/C++ or FORTRAN. The OmpSs interface extends OpenMP features and introduces flexibility and task based view. Nevertheless, OmpSs is not as widely accepted as OpenMP. It presents the state of the art and a new way of approaching parallelism on shared memory level. OmpSs together with MPI are future of programming in HPC world and are the starting point of the thesis research.

Selecting MPI and OmpSs programming models, would help programmer productivity and smooth transition of existing codes. Improving performance and exploring capabilities of combined MPI and OmpSs approach present a real challenge. In order to understand optimization technique of MPI and reducing communication overhead by using MPI. The thesis does detailed analysis of already existing overlapping technique of communication and computation. Understating both computation and communication lead to an optimal time to solution execution.

Once performance techniques of the chosen programming model has been understood, the thesis defines easy to use interface that combines MPI and OmpSs. A clear metric for friendly use interface does not exist, and feedback of potential users and already experience developers play the key role.

In order to address performance goals, the thesis shows a novel idea that merges features of MPI and OmpSs and software optimization techniques.

Portability across different HPC systems, requires customization of the run-time. Understanding hardware architecture, OS system implementation and implementation of MPI libraries show potential capabilities of MPI/OmpSs programming approach where the run-time implementation proves ideas.

Experiments and proof of concept of programming model, require well written highly optimized and widely known HPC applications. In order to compare results the thesis uses several HPC applications on different platforms and focuses on the High Performance Linpack (HPL) benchmark [15]. The fact that the HPL is used to order TOP500 [48] supercomputer places it to the most important HPC benchmark.

1.3 Contributions

This work makes the following contributions:

1. Investigation of techniques to overlap communication and computations in MPI applications.
2. Analysis and demonstration of the impact of a computational kernel on overlapping techniques and time to solution.
3. Demonstration that granularity of computational routines effects overlapping efficiency and thus needs to be tuned.
4. Proposing a hybrid MPI and OmpSs programming model, define user interface for MPI/OmpSs.
5. Propose overlapping techniques and their implementation by using MPI/OmpSs programming model.
6. Address implementation challenges of overlapping techniques on various HPC architectures.

7. Propose OmpSs_MPI library, a message passing library for MPI/OmpSs programming model.

8. Demonstration of porting by using MPI/OmpSs programming model on different HPC applications.

9. Evaluate of MPI/OmpSs on different HPC applications and their basic comparison with original MPI versions.

10. Comment a high efficiency of MPI/OmpSs for small problem sizes.

11. Investigate a tolerance to low bandwidth and OS jitters by using MPI/OmpSs.

This work makes the three main contributions:

1. Investigation of techniques to overlap communication and computations in MPI applications. Analysis and demonstration of the impact of a computational kernel on overlapping techniques and time to solution.Demonstration that granularity of computational routines effects overlapping efficiency and thus needs to be tuned.

2. Proposing a hybrid MPI and OmpSs programming model, define user interface for MPI/OmpSs. Propose overlapping techniques and their implementation by using MPI/OmpSs programming model. Address implementation challenges of overlapping techniques on various HPC architectures. Propose OmpSs_MPI library, a message passing library for MPI/OmpSs programming model.

3. Demonstration of porting by using MPI/OmpSs programming model on different HPC applications. Evaluate of MPI/OmpSs on different HPC applications and their basic comparison with original MPI versions. Comment a high efficiency of MPI/OmpSs for small problem sizes. Investigate a tolerance to low bandwidth and OS jitters by using MPI/OmpSs.

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 and parallel programming models. First, it introduces parallel computers and their benefits over single thread computing approaches. Then it reviews the software stack which is used on parallel computers.

- Chapter 3 illustrates the performance issues in current parallel programming environment and criticizes inefficient approaches that solves the performance issues. The Chapter 3 criticizes a poor programming productivity of proposed software optimization.

- In Chapter 4, the thesis analyses and compares overlapping technique of collective communication and computation based nonbloking interface, point-to-point approach and additional thread approach. It also emphasizes characteristics of computational kernels which overlap communication. The Chapter 4 outputs results that give clear guidelines for the MPI/OmpSs programming model.

- Chapter 5 is the kernel of the thesis. It proposes the MPI/OmpSs programming model, defines an user interface of the model, adds overlapping techniques to the OmpSs run-time, and explains implementation of the OmpSs run-time on different architectures.

- Furthermore Chapter 6 evaluates the proposed programming model in terms of performance. It introduces applications used for experiments and their characteristics. The Chapter presents and discusses results of ported HPC applications on different platforms and shows performance advantages of MPI/OmpSs programming model.

- In addition Chapter 7 proofs and demonstrates the high tolerance to low bandwidth networks and OS jitters of the MPI/OmpSs programming model. It describes a method to mimic low bandwidth network and simulates OS noise. It comments performance results of the MPI/OmpSs application affected by mentioned phenomenon.

- Chapter 8 presents previous work related to the research covered in this thesis, while Chapter 9 draws conclusion of this thesis and presents the direction of the future research in this field.

- Finally, Chapter 10 lists the papers published as the results of this thesis.

2
Background

This Chapter presents the state-of-the-art technology related to the topic of this thesis. Section 2.1 gives background on the architecture of parallel machines in HPC. It explains parallel machines on shared-memory and distributed-memory level, the section also reviews architectures specially build for HPC systems. Section 2.2 describes the parallel programming approaches for HPC systems. It focuses on the Message Passing Interfaace(MPI) library as the most widely accepted approach for distributed-memory parallel machines and briefly reviews the OpenMP programming model for shared-memory parallel machines. The section introduces task based programming model called the OmpSs that extends OpenMP.

2.1 Parallel machines

Modern science stimulates the rapid growth of high performance computing. When experimental and theoretical science join to computational science, large and powerful machines became a basic tool for them. The constant need for higher performance caused the appearance of machines with parallel architecture. Single core machines have not been able to fulfill scientific requirements. Multiplying single compute units, efficiently solved embarrassingly parallel problems but more likely real problems requires interconnection network between compute nodes due to dependent work across parallel machines. Building parallel machines brought challenges in hardware and software design. In the rest of this section, we present concepts related to HPC parallel computing for hardware and software.

2.1.1 Processor architecture trends

In the early 1960 there ware sever factors where experimental science evolved into computation science, and required large computer resources to solve their problems. The price of the advantage single-processor computer increases faster than its computational power. So, the price/performance ratio points to the direction of parallel computers. During the ’80 PC (personal computers) has increased the performance and price of single-processor computers has fallen. Also the price of interconnection network have fallen , thus a PC workstation has provided a significant computer power on a small budget.

Manufacturing more powerful PC computers was building processors with higher operational frequency. The straight-forward approach for making faster computers is making a computer that operates at a higher clock frequency. During the ’90 the vendors increased frequency, which makes all software running faster. The HPC systems followed the same trend by building the system out of commercial processors made for servers. However, increased frequency dramatically increases chip’s power consumption. Therefore, computing platforms have hit the “power wall”, so frequency scaling stopped at around 3 GHz.

From then on, improving single core performance introduces more computational unit additional functional units, additional registers, wider path, cache size etc. Exploring new resources required new chip design and lead to instruction level parallelism (ILP).

The following hardware approches appeared:

1. **Instruction pipelining** (ILP) where the execution of multiple instructions can be partially overlapped.

2. **Superscalar execution**, and the closely related explicitly parallel instruction computing concepts, in which multiple execution units are used to execute multiple instructions in parallel.

3. **Out-of-order execution** where instructions execute in any order that does not violate data dependencies. Note that this technique is independent of both pipelining and super-scalar. Current implementations of out-of-order execution dynamically (i.e., while the program is executing and without any help from the compiler) extract ILP from ordinary programs. An alternative is to extract this parallelism at compile time and somehow convey this information to the hardware. Due to the complexity of scaling the out-of-order execution technique, the industry has re-examined instruction sets which explicitly encode multiple independent operations per instruction.

4. **Register renaming** which refers to a technique used to avoid unnecessary serialization of program operations imposed by the reuse of registers by those operations, used to enable out-of-order execution.

5. **Speculative execution** which allow the execution of complete instructions or parts of instructions before being certain whether this execution should take place. A commonly used form of speculative execution is control flow speculation where instructions past a control flow instruction (e.g., a branch) are executed before the target of the control flow instruction is determined. Several other forms of speculative execution have been proposed and are in use including speculative execution driven by value prediction, memory dependence prediction and cache latency prediction.

6. **Branch prediction** which is used to avoid stalling for control dependencies to be resolved. Branch prediction is used with speculative execution.

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 techniques exposed parallelism without any involvement of the programmer – the user programmed a sequential control flow but the underlying system automatically exposed parallelism. However, the implicit parallelism turned on 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 Types in HPC Systems

HPC systems tend to be multi-core systems that access memory in two different ways: first shared-memory systems where cores are sharing physical memory and a single instance of operating systems control threads and processes, second distributed memory systems where several instances of operating system control local threads and processes while communication protocol allows communication between distributed memory system. Machines can also contain both approach where locally shared memory system are connected between each other as a big distributed memory system.

Shared-Memory Systems

Shared memory system appears as a solution for parallel computing where a single operation system and its adress space work on top of multicore hardware. The shared memory hardware connects all processing units and main memory, each processing unit can access the whole memory directly. Processors read and write to shared variables and communication throught them. Programming model and operation system provide a framework and programmer uses the shared variables in order to enforce correct exectution.

Parallel machines through shared variables is the easiest way to build parallel program because inter-process communication is implicit. Programmer should not be concerned with data locality. Shared variables create easy perspective on the global memory space. These machines is very difficult to build in terms of fast access to all addresses of memory from all processes. Reaching low latency in shared memory machines is the biggest challenge for hardware designers. Programmer does not need to provide any hints about data locality and thus a hardware cannot predict and overlap communication between memory and processing unit.

The Multi-threading execution model is widely used for shared memory machines. A single process launches several threads within the same address space. Threaded programming approach introduces a shared variable concept. Threads read and write to shared variable and a system preforms implicit communication. Valid shared variable is stored in the single address location within memory.

Accessing time from processing units to memory can be uniform or nonuniform, so there are two type of memory architecture: uniform memory access (UMA) architecture and nonuniform memory access (NUMA) architecture.

- UMA access time to a memory location is independent of which processor makes the request or which memory chip contains the shared variable. Thus, UMA platforms are often called symmetric multiprocessors (SMP). The main disadvantage of UMA architecture is scalability. Bandwidth of a single memory bank becomes the bottleneck for large number of processors.
- NUMA: Noways, shared memory systems are based on NUMA memory architecture. Instead of a single memory bank, NUMA contains several memory banks where the memory access time depends on the memory location relative to the processor. Accessing to its local memory is faster than non-local memory. NUMA attempts to address UMA single memory bank issue by providing separate memory for each processor, a voiding the performance penalty when several processors attempt to address the same memory. NUMA improves scalability compared with UMA architecture.

Distributed Memory Systems

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 using message-passing.

Distributed memory machines evolve by improving the interconnection network and decoupling the network from the nodes. 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.

2.2 Parallel programming models

There is no efficient and highly applicable automatic parallelization. The biggest idea of parallel computation was to find techniques that would automatically expose parallelism in the applications. However, despite decades of work, 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) [44]. 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 [10], X10 [12] and Chapel [11]. 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 [14] is a de-facto starndard 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 [17] 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 and distant. Throughout this thesis, we focus mainly on OmpSs as a programming model for shared memory machines.

2.2.1 MPI

Message Passing Interface (MPI) [44] 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 calls 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_Reduce, in which the specified root process receives one message from all the processes in the specified group. Additionally, the root performs an operation on all the received messages. MPI_Alltoall is the most expensive routine where all processes send and receive message from all processes. Other collective operations implement more sophisticated communication patterns.

Throughout its evolution, MPI standard introduces 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. While MPI-3 introduces non-blocking collective operations [24].

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 other neighbor, and again executes function $compute$ on the received buffer. The processes communicate in 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 more about the parallel environment by calling MPI_Init. MPI execution starts by calling MPI agent ($mpirun - nx./binary.exe$) that spawns the specified number (x) of MPI processes. In the studied case (Figure 2.2), 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 communicator, each process identifies the total number of MPI pro-

```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 2.1: Example MPI code

cesses 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

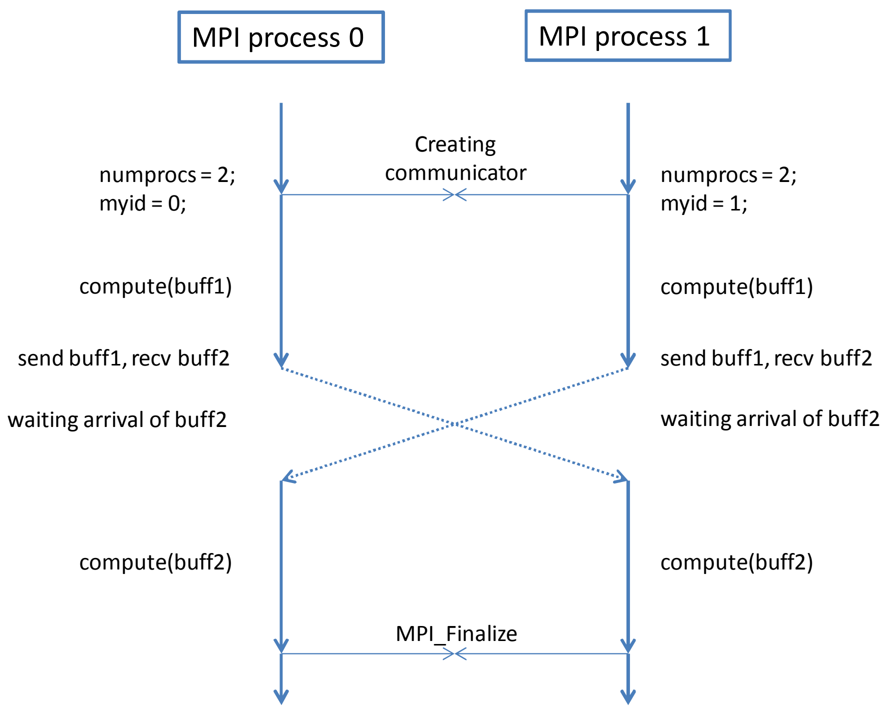


Figure 2.2: Execution of the example MPI code

ranks of the neighboring processes with which it should communicate in order to make 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 caused by 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.

Relevant features of MPI implementation

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 what they need to provide. This opens space for various MPI implementations that have different features and performances.

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

Eager protocol is faster, as it requires no "handshaking" synchronization. However, this relaxation of synchronization compensates with 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 outstanding 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. Still, most of the 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.

2.2.2 OpenMP

OpenMP (Open Multi-Processing) [14] 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.

```
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 2.3: Pointer chasing application parallelized with OpenMP

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.

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 2.3a. The program consists of a while loop that updates each element of the list. The code cannot be parallelized just by adding parallel loop construct, because the list traversal would be incorrect. Thus, in order to use parallel loop construct, the list first has to be translated into an array (Figure 2.3b). However, this translation causes the 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 2.3c illustrates the possible parallelization of the studied code using OpenMP tasks. The master thread runs and dynamically gives a raise to each instantiation of the task *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.

2.2.3 OmpSs

Omp Superscalar (OmpSs) [17] 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.

The OmpSs programming model extends the standard C, C++ and Fortran programming languages with a set of pragmas/directives to declare functions that are potential tasks and the intended use of the arguments of these functions.

There are two essential annotations needed to port a sequential application to OmpSs function-header | function-definition

With the following possible clauses:

- in(data-reference-list)
- out(data-reference-list)
- inout(data-reference-list)

The name, Superscalar, came from the same idea as a superscalar CPU where out-of-order execution of instructions mapping out-of-order execution of annotated functions. Superscalar programming family uses concepts mentioned in the section

2.1 which explains the ILP techniques. What is an instruction for ILP, it is an annotated function for OmpSs. Hardware implementation allows instruction level parallelism while OmpSs runtime as a software implementation provides function level parallelism.

Based on the in/out specifications and the actual arguments in function invocations, the runtime system is able to determine the actual data dependencies between tasks and schedule their parallel execution such that these dependencies are satisfied. The dependencies derived at runtime replace the use of barriers in most of the cases, allowing the exploitation of higher degrees of distant parallelism. Given the OmpSs annotations, the runtime can schedule all tasks out-of-order, as long as the data dependencies are satisfied.

The OmpSs environment consists of a source-to-source compiler that substitutes the original invocations of the annotated functions with calls to an add_task runtime call, specifying the function to be executed and its arguments. The resulting source code is compiled using the platform native compiler and linked to the OmpSs runtime library. The add_task runtime call uses the memory address, size and directionlity of each parameter at each function invocation to build a dependence task graph. A node in the task graph is added to represent the newly created task and it is linked to previous task on whose output it depends. Once a task is finished, the runtime looks in the graph for tasks that depend on this one and if they have no other pending dependencies they are interted into the ready queue. Concurrently with this main thread, a set of worker threads, started at initialization time, traverse this list looking for tasks ready for execution. In the case that the main thread encounters a synchronization (barrier, wait on specific data or end of the program) it cooperates with the worker threads to execute pending tasks.

In order to eliminate false dependencies (i.e. dependencies caused by data reuse), the OmpSs runtime is capable of dynamically renaming data objects, leaving only true dependencies. This is the same technique used in current superscalar processors and optimizing compilers to remove false dependencies due to the reuse of registers. In SMPSs the renaming may apply to whole regions of memory passed as arguments to a task. Such renaming is implemented by the runtime, allocating new data regions and passing the appropriate pointers to the tasks, which themselves do not care about the actual storage positions passed as arguments. The runtime is responsible for properly handling the actual object instance passed to successive tasks. Also if necessary, it copies back the data to its original position.

This renaming mechanism has the potential to use available memory to increase the actual amount of parallelism in the node. An excessive use of renaming may result in swapping and introduce a high performance penalty. A parameter in a configuration file limits the size of memory that can be used for renaming.

The priority clause gives a hint to the runtime system about the “urgency” when scheduling the task. The runtime has two ready queues and tasks from the high priority queue are selected before tasks in the low priority queue. This mechanism allows a programmer with global understanding of the critical computations to influence the actual scheduling.

One of the main goals of scheduling in OmpSs is to exploit data locality. In that regard the scheduler takes advantage of the graph information in order to schedule dependent tasks to the same core so that output data is reused immediately.

The programmer can add the *priority* clause to some task, indicating that the task has higher scheduling priority. These tasks are scheduled as soon as possible.

The OmpSs runtime is also aware about NUMA memory architecture and schedules a task according to NUMA locality of task arguments. The OmpSs scheduler contains a queue for NUMA domain and work stealing is supported. Applications do not have the same behavior, so the scheduling options are configurable from the configuration file where a user could tune scheduler for targeted application.

OmpSs programming model supports also programming heterogeneous architectures. By adding *target* clause 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. [6] and Nvidia GPUs[9].

OmpSs is a most promising programming approach in HPC world. It makes porting of sequential applications natural and straight-forward approach. A programmer could focus on expressing an algorithm, while the OmpSs runtime cares about performance and parallel execution. This thesis is based on the OmpSs approach and merge MPI and OmpSs.

The OpenMP 4.0 standard introduced OpenMP Task Dependency Support, thus it could be used as an implementation platform for the work of this thesis.

Example of irregular parallelism – Cholesky

Figure 2.4 shows OmpSs parallelization of a 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 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,see 2.5. 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 2.5). 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 2.5). Again, the programmer alone could hardly identify and expose this potential concurrency.

```c
#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(struct timeval *start, struct timeval *stop,
             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][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);
      }
   }

}
```

Figure 2.4: OmpSs implementation of Cholesky

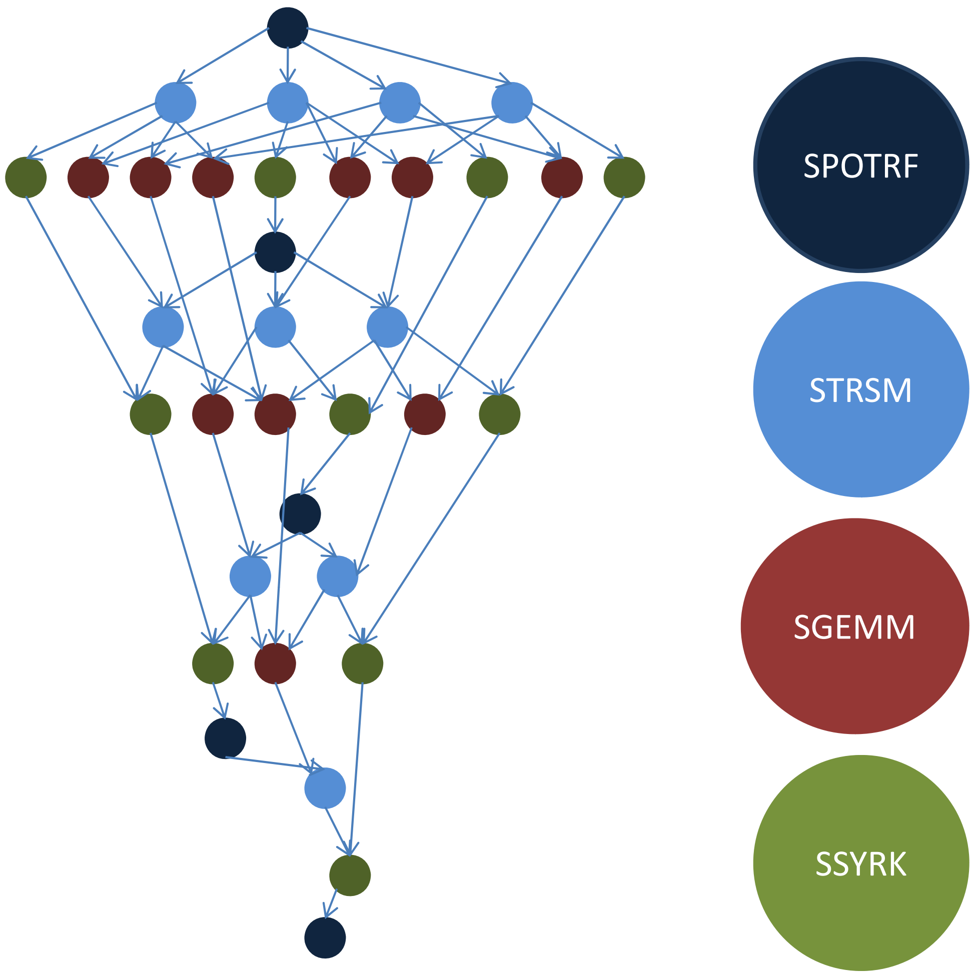


Figure 2.5: Dependency graph of Cholesky

3 Motivation

Delivering effective and portable parallel code to the complex HPC system is a big challenge. On the one side, we have scientific algorithms that model and simulate natural phenomena, and on the other side we have huge parallel machines. There are several layers between users and hardware. Compilers and operating systems give access to machines, while programming models give a view of machines to users. Nevertheless, in order to use machines efficiently, users require knowledge of all layers. Complexity of these layers is a nightmare for users. Very often, we see result with unsatisfactory performance.

High level programming models offer a nice environment and a good productivity, but codes often run poorly in terms of scalability, balancing and single thread performance.

Programmers start optimizing their codes by using lower level programming paradigms and tuning techniques. Productivity decreases, while the cost of development becomes very high. Finding balance between productivity and performance is a main goal.

In the following section, we further explain the challenges of parallel programming on HPC systems. We point to the tuning techniques used in MPI and MPI/OpenMP programming model and how these tuning techniques affect productivity.

3.1 Development Cost

There is no clear metric to assess programming models, thus evaluating the programming environment is difficult. Endless discussion usually finish with statements "it depends on a problem and machine", which is true for all programming environment across different platforms and algorithms.

The cost of running applications depends on total execution time, power consumption and frequency of execution. On the other hand, programmers try to deliver code before a deadline or within a reasonable time. In order to evaluate programming model, two properties are important: first, defining development time as the time necessary to deliver a code that would fulfill the requirements and solve the problem, and second a time of single execution. Finally, we can draw a curve that models a programming environment. The Figure 3.1 shows these curves.

Looking at Figure 3.1, on the x-axis is the development time, while y-axis presents the execution time of a single run. Longer developing time increases performance and makes the execution time shorter, and opposite, quick time to solution shows sub-optimal performance. Mathematically speaking, the development time and the execution time are inversely proportional. Different curves present different programming environment where analytic function of programming environments contain different constants.

- **Programming environment A**: $executionTime = ConstA * f(developingTime)$
- **Programming environment B**: $executionTime = ConstB * f(developingTime)$

The Figure 3.1 shows that a programming environment A is better than a programming environment B because $ConstA < ConstB$. Minimal $Const$ means maximizing success with a minimal effort. The thesis tries to propose the programming model where $Const$ is minimal.

This thesis offers the HPC programming model that increases performance without degradation of productivity. The thesis claims that a very first version of parallel program could be easily converted to the efficient productive version.

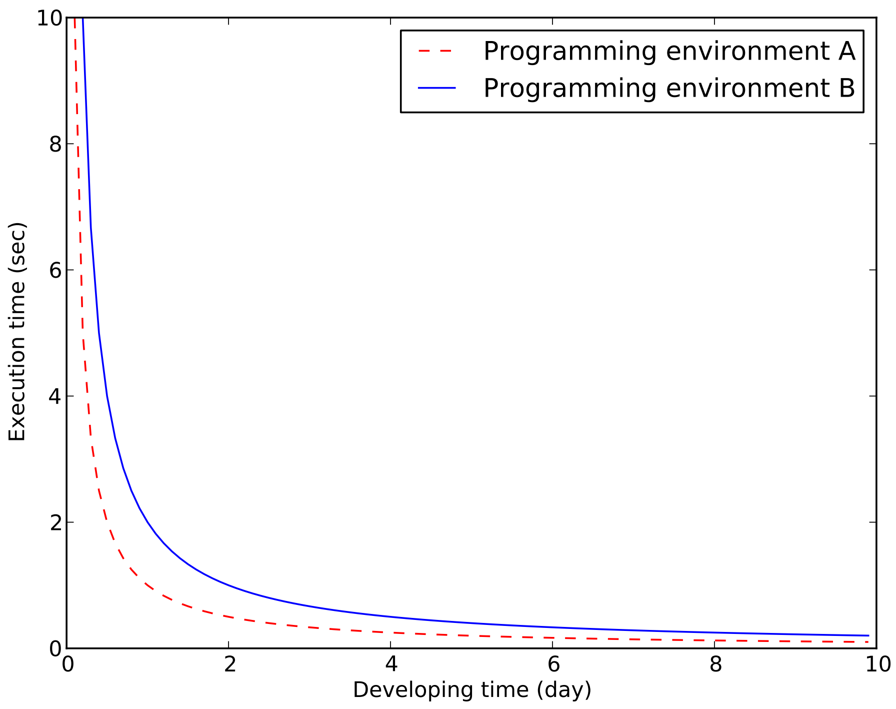


Figure 3.1: Modeling programming enviroments. A curve presents the properties of the programming enviroment.

3.2 MPI Tuning and Its Challanges

MPI programming paradigm is ubiquitous in scientific computing. Nowadays, we cannot find a production code that runs on thousands of cores without using the MPI paradigm. Explicit communication, low level mechanism and static distribution give full control to programmers. In the very first phase of the code design, a programmer cares about explicitly partitioning the workload among processes with separate address spaces. Data distribution and number of instruction per process should be equally distributed, while invoking communication calls should be as synchronized in order to balance execution of processes. This requires optimization on the MPI level, where the programmer introduces tuning techniques by hand. Optimization of MPI codes may be very complex and requires a restructure of codes which decreases readability. In this section, we address the synchronization between processes and an overhead of transferring data, named transfer issue. Finnally, the Section 3.2 discusses the scalability issue and how MPI/OpenMP addresses the scalability.

3.2.1 Synchronization Issue

In order to reach perfect synchronization between a group of MPI processes, MPI processes should arrive at the same time to a blocking MPI call that performs communication between them. Fulfilling these requirement, depends on various hardware and software properties. MPI needs hardware with identical nodes and interconnection network that makes access to all nodes identical. Placing MPI processes across identical cores and binding processes helps to the symmetric execution. From the software point of view, an algorithm should distribute load and instruction equally across the nodes. Perfect distribution of algorithm and symmetric hardware still cannot guarantee a perfect synchronization because of the unpredictable behavior of OS noise, cache locality, CPU temperature could disturb runs and introduce load imbalance.

Non-synchronized processes cause communication delays and idle states of CPUs, a local slowdown of a single core propagates across all processes and can significantly harm the overall performance, especially at the large-scale.

Programming practice suggests code restructuring in order to avoid synchronization delays. Understanding hardware and computational routines, programmers estimate execution time of certain parts of code and predicts synchronization point of communication calls. This is not straight-forward programming practice and it is very difficult to predict a synchronization point of MPI calls in the development phase.

The Figure 3.2 shows tuning steps from an original MPI code where processes are not synchronized to the code where different order of computational chucks makes the execution balanced. The process P0 arrives earlier at the communication point then P1, as has to wait, which leads to the *synchronization issue*.

Arrows present a communication direction between process while a red burst shows an idle state of core during the communication. We can see that the synchronization issue and the transfer issue cause the idle state of one processes. Restructuring computational chucks improves synchronization and minimizes the idle time on P0.

Therefore, there is a need for a new approach to solve problems of synchronization where programmer should not care about order of computational chucks. The goal of

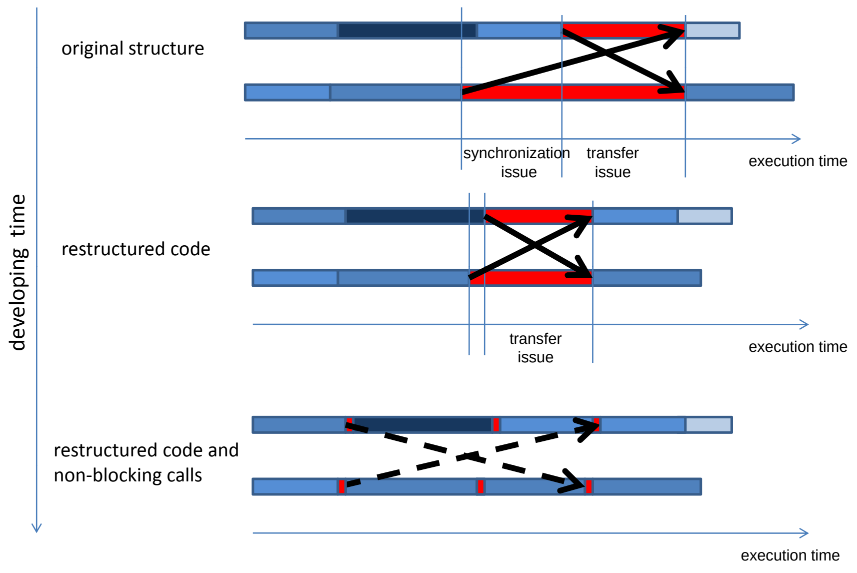


Figure 3.2: Tuning challenges: Synchronization Issue and Transfer Issue and how to address them. X axis is the exuction time, Y axis is the development time. Red chunks are idle state. Blue chunks are running state.

a new approach is automatic, out-of-order, scheduling of computational chucks that leads to a good synchronization across MPI processes.

3.2.2 Transfer Issue

The Figure 3.2 shows, that solving synchronization problem does not reduce the transfer time. The red bursts are still present. Two processes start communication at the same time, they hold cores busy while interconnection network perform communication operation, named the *transfer issue*. The overhead of communication transfer depends on hardware properties of interconnection network (bandwidth and latency), routing, size of messages and implementation of MPI libraries.

On the other hand, application developers have no control over these factors and they use optimization techniques on the application level in order to reduce the transfer time. Overlapping communication and computation is the most promising technique to reduce communication delays. By using overlapping techniques, processors start communication operation and switches to computation that does not require completed communication operation. This approach speeds up execution and increases overall efficiency.

Nevertheless, this tuning technique requires again code restructuring and the use of non-blocking MPI call. Programmers place non-blocking call and request a communication operations. The core instantly executes non-blocking call and continue further with computation, while the communication hardware transfers data in the background. Programmers also need to place probing calls which would frequently check for the end of communication operation and release a communication buffer for new writes.

The Figure 3.2 shows the final step in the development phase where code restructuring and non-blocking communication calls tune executions. Small red bursts present probing calls that intercept computation. Sometimes probing calls have to split an external single library call, this also increases code complexity because programmers should divide the library call in several calls of the same routine.

Our approach also targets the transfer issue and aims automatic overlap of communication and computation without code restructuring or explicit use of non-blocking and probing calls.

3.2.3 Scalability Issue and Fork/Join Issue

Overlap and code restructuring show solution for synchronization and transfer issues of MPI. Nevertheless, writing scalable program for huge number of processes might introduce the *scalability issue* due to nature of algorithm. Some algorithms limit a maximum number of MPI processes and do not scale beyond this maximum number. Static data distribution across MPI makes load balancing difficult and increases number of MPI communication calls.

In order to address the scalability issue, programmers mix MPI and OpenMP programming models, e.g. a single MPI process contains N OpenMP threads. Thus total number of MPI processes decreases by N which implies N times less communication

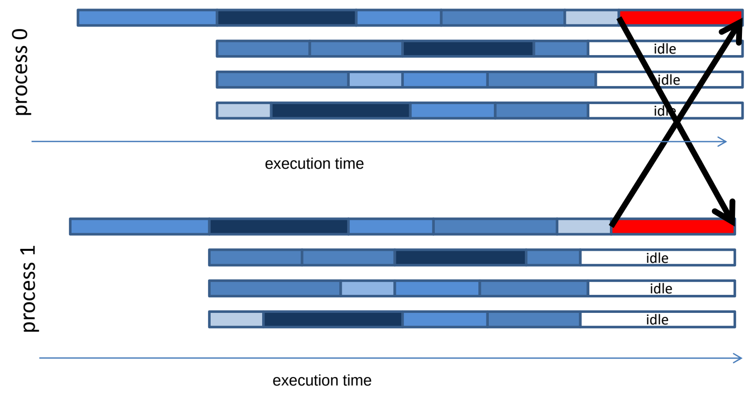


Figure 3.3: Scalability Issue: MPI/OpenMP improves the scalability issue but introduces fork/join issue. First four lines present a process 0 with for 4 threads, second four lines present the process 1 with 4 threads. Red chunks are idle state. Blue chunks are running state. White chunks are idle states due to fork/join issue.

between nodes and better scalability. Dynamic load balancing of OpenMP approach improves overall performance.

Programming practice places communication calls in a main thread, while N-1 threads wait for the end of communication operations. The Figure 3.3 shows the scenario where the main threads of processes 0 and 1 exchange data while other threads are idle. The cost of the performance drop due to the fork/join issue depends on number of OpenMP threads. Larger N leads to better load balancing but an expensive fork/join. This is the reason why MPI often outperforms MPI/OpenMP implementation of a given algorithm.

Our programming model would contain benefits of a good dynamic load balancing within node as OpenMP but it avoids the fork/join issue by allowing communication concurrently with computation in any thread. We also propose a hybrid programming approach as MPI/ OpenMP that automatically addresses the communication overhead issue based on OmpSs data flow programming model.

3.3 Automatic overlap

The thesis proposes the hybrid programming model MPI/OmpSs. The approach is based on data-flow programming, where programmers divide computations in tasks and define data dependencies between them. Instead of code restructuring in traditional optimization technique, programmers keep the same structure of the code and only give hints to the system. The system does out-of-order execution of computational tasks, and makes decisions during runs. OpenMP also addresses dynamic load balancing, but OpenMP can deal only with an embarrassingly parallel workload where little or no effort is required to separate the problem into parallel tasks. OmpSs automatically creates a data dependency graph and deals with any kind of workload.

The approach goes one step further and encapsulates MPI calls within OmpSs tasks. Thus the system could call them in out-of-order fashion,furthermore named a communication task. Out-of-order execution of computation tasks and communication tasks minimize the synchronization issue because the OmpSs task scheduler dynamically schedules tasks and accelerate a critical path of execution. MPI blocking calls still hold CPUs that execute them. This makes blocked cpus unusable for OmpSs threads.

The approach introduces various techniques for treating communication tasks and does not allow communication tasks to block CPUs. As soon as a synchronization issue has been detected, a thread that blocks a core yields the core which makes the core available. Immediately, another thread takes the core and starts execution of a ready-to-run task.

The MPI/OmpSs execution model aims at the execution model of the Figure 3.4. Processes with several threads improve scalability, the system automatically reorders chunks and overlaps communication where all threads can execute communication in parallel with computation.

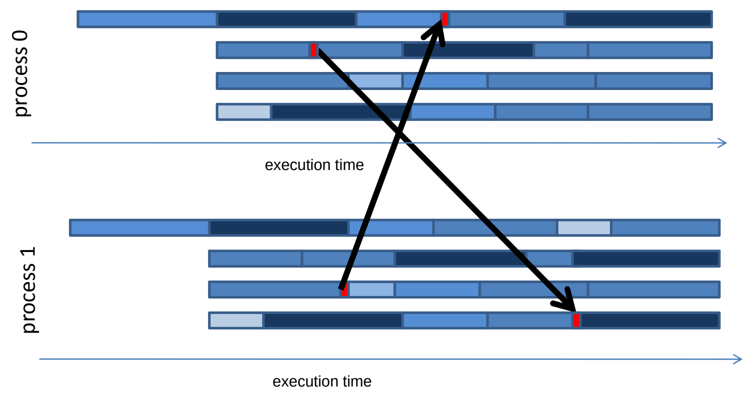


Figure 3.4: The execution with automatic overlapping and reordering. Blue chunks are running states.

Our contribution in exploring automatic overlap

- Propose HPC programming model that keeps the same structure as a pure non-optimized MPI code and introduces automatically all tuning techniques to reduce communication overhead.
- Build a programming model that is more tolerant to low bandwidth network and non deterministic perturbations.
- Analyze different tuning techniques for overlapping computation and communication on various machines.
- Explain porting strategy and compare results of well know kernels and applications.

4 Analysis and Evaluation of Existing Overlapping Techniques

The Chapter 4 reviews existing overlapping techniques and analyzes a correlation between the characteristics of computational kernel and overlapping techniques. Conclusions of the Chapter 4 provides a basis and an essential knowledge required for the MPI/Ompss programming model.

4.1 Introduction and Motivation

The main issues to achieve good, performance is reducing the communication overhead between MPI processes. MPI contains asynchronous communication routines that offer the possibility for overlapping communication and computation. Programmers use the MPI's asynchronous (non-blocking) communication calls to: (i) issue communication requests as soon as the data is ready, (ii) perform another computation independent on this data, and (iii) then wait for the end of the communication. The

MPI-1 and MPI-2 standards support asynchronous communication calls for the most essential point-to-point operations (MPI_Isend(), MPI_Irecv(), ...). However, a regular communication pattern easily can be expressed by calling collective routines, where MPI-1 and MPI-2 provide only synchronous interface.

An exascale machine executes a huge number of MPI processes, which uses expensive collective operations for communication between MPI processes. The two standard approaches for achieving better communication performance are (i) improving network infrastructure, and (ii) improving software communication patterns. Improving the network infrastructure, provides a better interconnect network with higher bandwidth and lower latency. Increasing bandwidth is technically relatively simple, but latency is already close to the physical limit an cannot be increased. At the same time improving the network infrastructure causes high network hardware cost. On the other hand, the software improvements provide a cheaper approach for achieving better performance. The improvement of software communication patterns forces the programmers to overlap collective communication with computation. The programmer have two standard approaches to overlap computation and communication: (i) by implementing collectives with *asynchronous point-to-point* routines or (ii) by implementing a *multithreading* approach e.i. placing a blocking collective to an additional thread. These approaches have not been popular in MPI programming community due to increase of the code complexity. The code complexity arises from the interaction of the MPI communication patterns with the routing protocols and underlying interconnection networks. In the case of multithreading approach, the code has to take into account the presence of operating system scheduler which increases the complexity on the MPI applications.

The both software approaches (*asynchronous point-to-point* and *multithreading* approaches) interrupt the computational kernel. Point-to-point collectives explicitly probe for a message while the OS controls sleeping and running phases for the additional thread. The interruption of computation introduces an overhead in overall performance.

The overhead depends on the characteristics of the kernel. This work evaluates overall performance degradation introduced by overlapping techniques on the computational kernel and analyzes performance of collective communications.

For the evaluation, the thesis uses synthetic benchmarks and the High Performance

LINPACK (HPL). We implement the synthetic benchmark, Concurrent Communication Computational (C-CUBE) synthetic benchmark, which uses three overlapping techniques (point-to-point, mutithreading with communication thread, and MPI-3's Ibcast) and we perform broadcasting in parallel with three different type of kernels (compute-bound, memory-bound and compute/memory bound (hybrid) kernels). We also evaluate overlapping techniques on the High Performance Linapck as a case study for real application.

The Chapter 4 makes the following contributions:

- We show that in order to reduce the application's communication overhead the programmer has to understand the characteristics of the computational kernel and the implementation of overlapping techniques. The lack of understanding can reduce the performance and make the overlap useless.

- We show that the MPI-3 standard introduces easy-to-use concept with non-blocking collectives which increases programming productivity. At the same time, the optimized MPI non-blocking collectives with a asynchronous communication thread achieve the desired computation/communication overlap without additional overhead.

- Finally, it is shown that a hand-tuned collective operation shows better progress behavior than the MPI broadcast collective.

The rest of the the chapter is organized as follows. Section 4.2 explains the overlapping technique for collective operations and defines three types of computational kernels. Sections 4.3 gives implementation details of the C-CUBE benchmark and explains a modified look-ahead technique for the HPL. Section 4.4 evaluates the overhead of overlapping techniques. Finally, Section 4.5 concludes the chapter.

4.2 Overlapping Techniques and The Nature of Computational Kernels

In this section, we review known overlapping techniques and classify computation kernels depending on their memory/CPU usage.

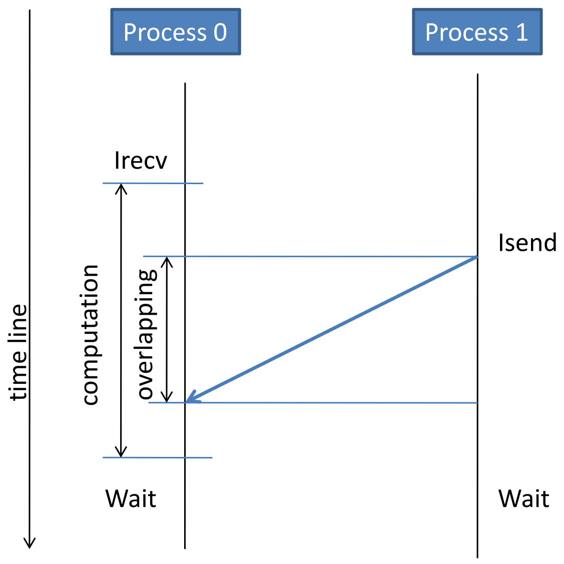


Figure 4.1: Overlapping communication and computation by using nonblocking point-to-point calls.

4.2.1 Overlapping Collectives

From the very beginning, the MPI-1 standard offers an interface for overlapping communication and computation, e.g.of nonblocking sends (MPI_Isend) and nonblocking receives (MPI_Irecv) to improve performance. The nonblocking calls initiate operations, place requests and immediately return the execution flow to the main thread that executes the computation, while the communication operation progresses in the background. The overlap of computation and communication finishes with a waiting call, where the main thread verifies that the data has been copied out, see Figure 4.1.

The MPI-1 and MPI-2 standard contain nonblocking point-to-point operations, however collective operations can only be expressed with a blocking interface. Collective operations have high demands on IC in terms of high bandwidth and lower latency. Some system, for instance IBM Blue Gene dedicate a special network just for a collective communication.

Programmers use two software approaches to try to reduce communication cost: (i) the first approach requires implementation of collective communication by using non-blocking point-to-point calls [49] and (ii) the second approach tends to overlap computation with an additional communication thread[29] and thus exploits a multi-core and hyper-threading architecture of modern nodes.

Algorithms - with potential overlap of collectives - motivate programmers to avoid easy-to-use concept of collective routines and implement their own collective operation that hides the communication overhead. This was the first approach that addresses the communication overhead of collectives.

The manual implementation of collective operation (by using nonblocking collectives) significantly increases code complexity This technique requires code restructuring and the placing of probing calls in the code. The performance of collective operations also depends on a routing of interconnection network and makes the manual implementation less portable from performance point of view. The programmers are forced to understand several layers of software stack (algorithms, MPI implementations, mapping of MPI processes and interconnection networks). A complex software stack reduces programmer's productivity and as a consequence the programmers rarely use nonblocking hand-tuned collectives.

The second approach exploits SMP nodes and multi-threading. The programmers overlap the communication and computation by changing the applications to run collective operations in the communication thread (a.k.a. communication thread) while the application's main thread executes the computation. This approach requires forking and joining of the communication thread for the synchronization, see Figure 5.4.

The communication thread approach is easier to use than point-to-point implementation. However, the thread-safety of the MPI library and the interference of the OS scheduler limit the usage of this approach.

The MPI-3 standard supports non-blocking collectives[26] and introduces easy-to-use concept for overlapping collectives. At the same time, the MPI library developers can improve the performance of MPI collectives by targeting specific architectures and specific HPC systems.

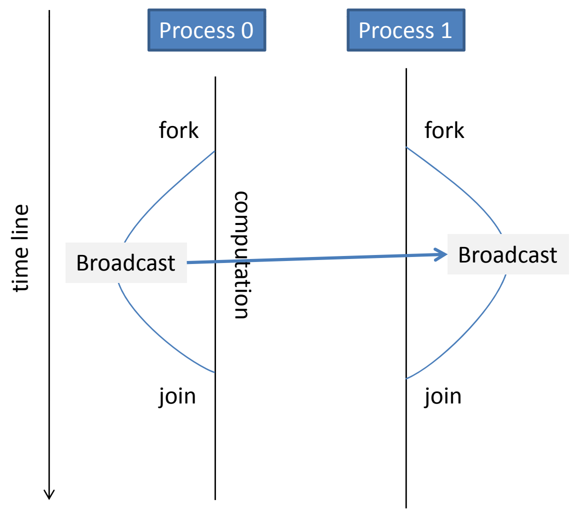


Figure 4.2: Overlapping communication and computation by using blocking thread.

4.2.2 Computational Kernels

In general, MPI nodes execute same application code with different data sets and use MPI's synchronization/communication primitives to coordinate their execution. The programmers' main focus is:

(i) to identify the computational kernels of the applications, and

(ii) to write synchronization/communication code to coordinate the communication between the nodes.

In other words, programmers have to understand the computational kernels and communication patterns present in applications, in order to efficiently map the HPC applications to MPI programming model.

In HPC applications, there are 3 types of kernels:

(i) memory intensive kernels,

(ii) computationally intensive kernels, and

(iii) memory and computationally intensive kernels.

- **Memory intensive kernels** spend most of their execution time reading and writing memory. One typical example of memory intensive kernel is a kernel that copies the data from one memory area to a different memory area using the `memcpy()` function.

- **Computationally intensive kernels** spend most of their execution time doing computation. One example of computationally intensive kernel is a kernel that calculates prime factors of a large positive integer number. The prime number calculation is trivially parallelizable algorithm that does not require communication between MPI nodes.

- **Memory and computationally intensive kernels** spend their execution time accessing memory and doing computing equally. One example of memory and computationally intensive kernel is a kernel that executes the matrix multiplication algorithm. The matrix multiplication algorithm stresses the memory by reading large matrices and stresses the CPU's ALUs by executing several multiply and add instructions every cycle.

4.3 A Case Studies: Synthetic benchmark and HPL

In this section, we describe applications used in the evaluation phase. First, we explain the implementation of the Concurant Communication Computation (C-CUBE) synthetic benchmark and later, we review a look-ahead communication optimization used in HPL.

4.3.1 The C-CUBE benchmark

In order to evaluate the influence of the computational kernels to the overlapping techniques, we developed the C-CUBE synthetic benchmark. The C-CUBE contains three different overlapping techniques and three different computational kernels. The synthetic benchmark isolates the MPI overlap approaches from the side effects (data distribution, load balancing, MPI synchronization, an algorithm specific execution) present in MPI applications.

The C-CUBE uses a broadcast as a collective communication operation. The broadcast is the one of the most used collective operation in scientific applications. A broadcast strategy is a basic operation in distributed linear algebra algorithms[20] and is the most demanding communication pattern that used a special network interconnection. On the other hand, the all-to-all communication is the most expensive communication that uses 2D and 3D tours on the machines with a special interconnect.

We apply three overlap techniques to a broadcast operation:

- **Asynchronous Hyper-cube Broadcast** - For the point-to-point version of broadcast, we use a hypercube communication pattern. The hypercube algorithm preforms broadcast among N processes in log(N) steps. The algorithm broadcasts information in the lowest number of necessary steps and show the highest efficiency in regular network topology[18].

  Each process calls MPI_Recv once, while the number of sends depends on the rank of a process. We add a probing call before MPI_Recv to make the broadcast asynchronous. Pseudo code is written below:

```
for(i=0; i<log(N); i++){
  update(mask, mask2)
  if ((rank & mask2) == 0){
   partner = rank ^ mask;
   partner =  MModAdd(partner,root,size);

   if (rank & mask){
     MPI_Iprobe(..., &flag ,...);
     if ( !flag ) return 1;
       MPI_Recv(...);
   }else
     MPI_Send(...);
   }
}
```

- **Communication Thread** - In order to overlap communication and computation using multi-threaded model, we choose the Pthreads programming model. The Pthreads implementation is a low-level and light programming approach for implementing a communication thread. It is the standard library for implementation of shared memory programming models due to high performance. Pthreads offers full control over creating, joining and destroying the communication thread.

  In the communication thread technique, we place the blocking MPI_Bcast in the communication thread. The main thread creates the communication thread, starts a computation, and sets a synchronization point after the computation.

- **Asynchronous Broadcast Operation** - The MPI-3 interface contains the MPI_Ibcast function as a non-blocking broadcast operations. The MPI_Ibcast starts a asynchronous broadcast operation, while probing for a message preforms interruption of a computation kernel. The MPI_Test first checks whether data has already arrived or not. If so,the MPI_Wait can be done and data is available. In the case data has not arrived, the computation kernel runs again.

  We introduce probing for messages in order to measure the progress of communication. Programming practice with a pair MPI_Ibcast and MPI_Wait is enough to preform an overlap.

The synthetic benchmark contains three computational kernels: check prime number as a compute-bound, memory copy as a memory-bound kernel, and matrix multiplication as the hybrid kernel.

- **Check Prime Number** - We use primality test function as a compute-bound kernel. The primailty test function uses the trial division algorithm to determine if a number is prime. The algorithm tests if the number is divisible by 2 or any odd integer greater or equal than 3 and less then the square root of the number. Essentially, the algorithm is brute force test that does not access memory and only stresses the ALU. It is important to mention that primality test function does computation accessing only the CPU registers because compiler is able to put all the variables (that are used in the main loop of the kernel) in registers and as a consequence the main loop does not access memory.

- **Memory Copy** - We use the `memcpy()` function from the C standard library as a memory-bound kernel. The `memcpy()` function is a good proxy for memory-bound applications because it stresses the memory subsystem by executing memory move instructions in a tight loop. We can assume that `memcpy()` function will be implemented as `rep mov` instruction on x86 processors because on the current Intel Nehalem and Sandy Bridge processors, this is the fastest method for moving large blocks of data, even if the data are unaligned[19].

- **Matrix Multiplication** - We use the matrix multiplication kernel DGEMM as a hybrid kernel. DGEMM is the most widely used routine to compute matrix multiplication using double precision floating point numbers. The performance of DGEMM is expressed in FLOPS (floating point operations per second) and is often used as used metric for HPC systems. There are several very well optimized implementations [22].

  In this chapter, we use Cray DGEMM implementation for our benchmark. In our evaluation environment, DGEMM reaches the efficiency of 103% of peak performance due to Intel "Turbo Boost" technology that enables the CPUs to run at higher frequencies than specified. The high efficiency is a clear sign message of the high optimization levels of DGEMM, and any interference with this kernel could easily lead to a performance drop.

4.4 Results

In this section, we describe the platform used for the experiments and evalute the overlapping techniques in term of performance for the C-CUBE benchmark and the HPL benchmark respectively.

4.4.1 Platform

We give a brief overview of the cluster describing computing nodes and interconnection network [4].

64 nodes with 2 chips of the Intel SandyBridge 2,6 GHz E5-2670 per node and 64 GB of shared memory per node. Each chip has 8 cores (16 hardware threads) with 20 MB of shared cache memory per chip. The cores use "Turbo Boost" technology that enables the cores to run up to 3.3 GHz when the thermal budget is not exceeded. All of the core can execute 8 floating point operations per cycle.

We use Cray Programming Environment: C Cray Compiler, MPICH-6.0.1 Cray MPI library that uses the MPICH2 distribution from Argonne, the MPI library provides a full support for the MPI-3 standard. The Cray Scientific Libraries package, and LibSci BLAS (Basic Linear Algebra Subroutines, including routines from the University of Texas 64-bit libGoto library).

The Cray MPI implementation uses an asynchronous progress engine to improve overlapping of communication and computation. Each MPI process launches a communication thread during the initialization phase. These threads help progress the MPI engine. Cray modifies the OS Linux kernel to treat communication threads differently from application threads. Enabling and disabling the asynchronous progress engine options has a huge impact on the overall performance. The usage of asynchronous communication thread can be disabled with environment variable.

All experiments are done on 1024 cores. The number of MPI processes per node is equal to the number of real cores per node and we bind MPI processes to the cores. Communication thread approach requires two threads per process, so we used hyper-threading over the logic cores. This process mapping isolates NUMA effect and maximizes utilization of the nodes.

4.4.2 C-CUBE Evaluation

We run the C-CUBE benchmark with small (1KB) and large (128KB) message sizes. The small messages use MPI with eager protocol, and the large messages use MPI with reandezvous protocol. We run the benchmark with and without asynchronous progress thread by setting MPICH_NEMESIS_ASYNC_PROGRESS environment variable.

We set different problem sizes for different computation kernels. Thus kernels execution take roughly the same amount of time. For matrix multiplication, matrix dimensions are A = 16384x128, B = 128x16384 and C = 16384x16384. Size of matrix bigger than 64 and power of two gives the best performance of the Cray DGEMM. We split matrix multiplication in 128 iterations in oder to interleave matrix multiplication with communication. Similarly, we split mem-copy kernel and compute-bound kernel in 128 iterations. For mem-bound kernel, we copy 80KB per iteration and compute-bound kernel we test primality of the number 10963707205259 [43].

First, we measure the execution time of computational kernels and broadcast implementation without overlapping and then we combine a broadcast and a kernel in round robin manner. Table 4.1 shows the execution time of all combination of computation kernel and communication implementation for different problems sizes and mode of progress engine.

Figure 4.3 presents the second column of the results from Table 4.1. We enable the progress thread engine and set the broadcast message size to 128KB. The execution times are clustered in three groups (compute-bound, memory-bound, and hybrid-kernel). OrgComputation and OrgCommunication bars represent the execution time without overlapping and hatched stack-bar on top of the bars represent an overhead introduced by overlapping techniques: (point-to-point hypercube)P2P, communication thread, and non-blocking collectives respectively. The blocking broadcast with communication thread and P2P hypercube implementation have the same execution time, while non blocking broadcast has 4.57x higher execution time. The non-optimal implementation of the MPICH-6.0.1 could limit a purpose of asynchronous collective for short computations.

Comp kernels	Broadcast	Comm./ Comp	Progres thread ON		Progres thread OFF	
			Small	Large	Small	Large
Comp bound	P2P	Comm.	0.25388	0.40533	0.25295	4.88044
		Comp.	4.69775	5.00175	4.72262	5.00169
	Comm. thread	Comm.	0.00151	1.03605	0.00071	1.03033
		Comp.	4.68298	4.84199	4.70082	4.84628
	Ibcast	Comm.	0.29041	1.10040	0.21946	5.52053
		Comp.	4.69291	4.85831	4.70131	4.72761
Memory bound	P2P	Comm.	0.25340	0.50630	0.25427	4.82266
		Comp.	4.68523	4.84253	4.68656	4.85381
	Comm. thread	Comm.	0.00148	1.85891	0.00086	1.88939
		Comp.	4.68345	4.68526	4.68557	4.68546
	Ibcast	Comm.	0.21856	1.59165	0.25449	5.47348
		Comp.	4.68550	4.71227	4.68673	4.63563
Hybrid	P2P	Comm.	0.23620	0.45799	0.23565	4.25181
		Comp.	4.34192	4.34291	4.26505	4.31262
	Comm. thread	Comm.	0.00120	1.33824	0.00078	1.33916
		Comp.	4.11927	4.44465	4.17536	4.46042
	Ibcast	Comm.	0.23602	1.24101	0.23654	4.84270
		Comp.	4.13161	4.27067	4.19874	4.12929

Table 4.1: The exection time of computation and communication for various overlapping techniques, computational kernels, mode of the progress thread and message size.

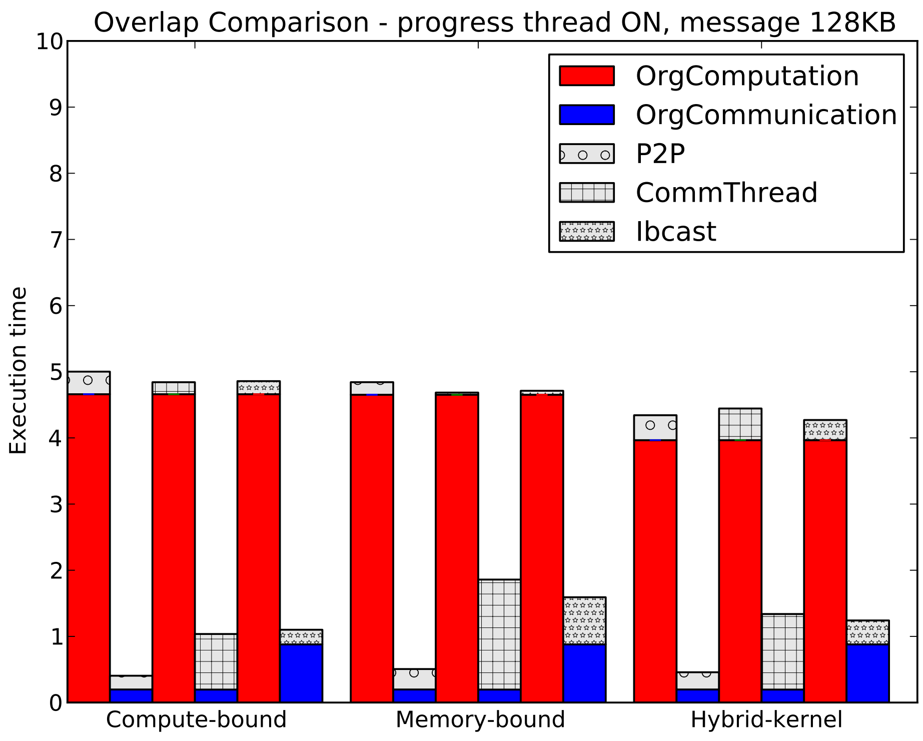


Figure 4.3: The execution time of communication and computation for various computation kernel and overlapping techniques when the progress thread is disabled and the message size is 128KB.

Broadcasting performance of P2P approach shows the best results. For the P2P, the execution time depends on probing granularity, while the communication thread and asynchronous collective approaches depend on core utilization due to hyperthreading. Broadcasting overhead is the most significant for overlapping memory-bound kernel, due to concurrently memory copy operation of both threads.

Send calls of the P2P case dominates the computation overhead. Communication thread and non-blocking collectives overlapping techniques introduce an overhead to optimized compute-bound kernel and almost no overhead to memory bound kernel due to a good OS scheduling.

The performance of hybrid-kernel (optimized DGEMM) is very sensitive to an interrupt. For the hybrid-kernel combined with the P2P and the communication thread approaches, the computation overheads are bigger than the broadcasting without overlap, which makes overall performance lower. Only non-blocking broadcast gains performance improvement against non-overlap execution because the progress thread performs the forwarding message with the lowest overhead.

Figure 4.4 presents the second column of the results from Table 4.1. We disable the progress thread engine and set the broadcast message size to 128KB (this implies that we use randezvous protocol). Asynchronous calls of P2P approach and nonblocking collectives use the progress thread.

Asynchronous routines from the P2P and non-blocking broadcasts get benfit from the progress thread. In Figure 4.4, communication bars of the P2P and non-blocking broadcasts are higher than computation bars. Initial ratio of OrgComputation and OrgCommunication without overlap is 19.84 and after overlapping computation/communication is to less than 1. The probing for message do not help progress MPI engine stage. Broadcasting starts at the end of computation phase while asynchronous calls start before computations.

In the P2P and non-blocking broadcasts, probing calls, MPI_Test or MPI_Iprobe, intercept computations. The number of probing calls necessary to complete communication operation, we also call the polling frequency. Table 4.2 compares the polling frequency of the P2P and non-blocking broadcasts for massage size and thread progress mode.

Table 4.2 also shows the same conclusion, where MPI_Ibcast needs up to 3 times more probing messages than the hypercube implementation in order to complete com-

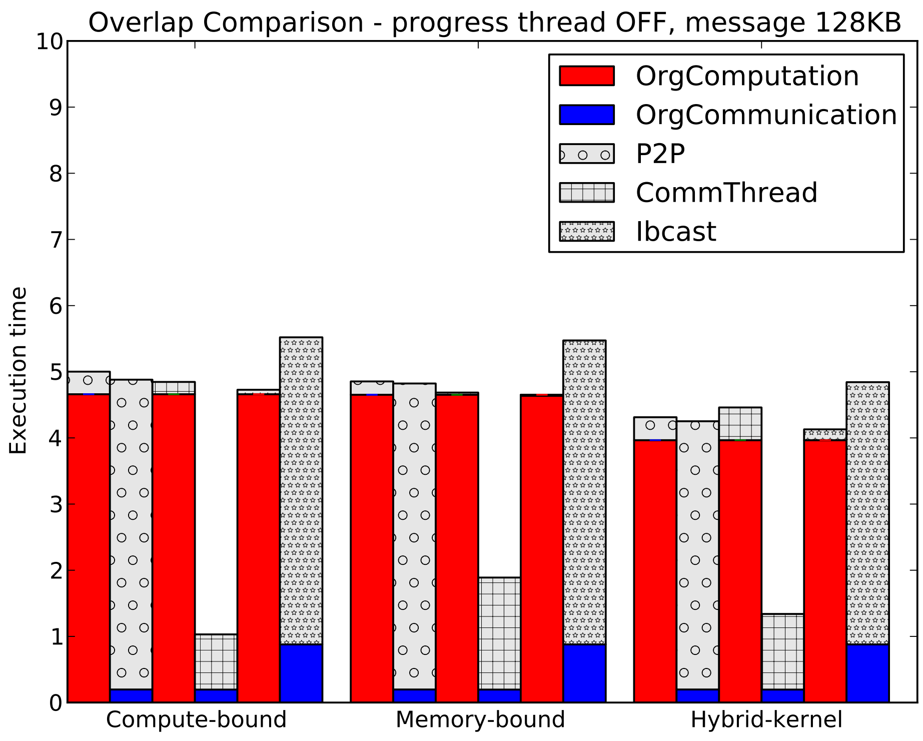


Figure 4.4: The execution time of communication and computation for various computation kernel and overlapping techniques when the progress thread is disabled and the message size is 128KB.

Comp Kernels	Broadcast	Progres thread ON		Progres thread OFF	
		Small	Large	Small	Large
Comp - bound	P2P	7	10	7	128
	Ibcast	8	27	6	128
Memory - bound	P2P	7	14	7	128
	Ibcast	6	42	7	128
Hybrid	P2P	7	13	7	128
	Ibcast	7	36	7	128

Table 4.2: Frequency polling of asynchronous broadcasts for dirrent problem sizes and the thread progress mode.

munications.

For the small messages, P2P approach performs the communication in 7 probing points. This is expected result because hypercube needs n = $\log_2 1024$ = 1= hops to finish broadcast operation. There are runs where nonblocking collectives completes the operation in 6 steps and outperform hypercube for a small messages. Nonblocking operation should always outperform hand-tuned version because it does not need explicit probing point to speedup a progress.

4.4.3 HPL Evaluation

For the HPL described in Section 6.1, we evaluate the performance using 1024 cores. We compare the performance of these four versions of HPL for different problem sizes (Figure 4.5): (i) no overlapping, (ii) with lookahead depth 1 and P2P broadcasting(2modified ring), (iii) with lookahead depth and 1 Communication thread, and (iv) with lookahead depth and 1 non blocking broadcast.

HPL uses (P,Q)=(16, 64) 2D decomposition. Note that decomposition (16, 64) may not be optimal for these number of cores and (32, 32) decomposition make better overall performance. Our work is focused on overlapping evaluation and Q = 64 means broadcasting needs 64 processes. We could use 1D decomposition (1, 1024) where a broadcasting is done across 1024 process but load imbalance of the HPL would be dominant factor. So we decide to make a trade-off between a good load balancing and expensive broadcast.

Figure 4.5 shows the performance rate (Gflop/s) of overlapping versions with asyn-

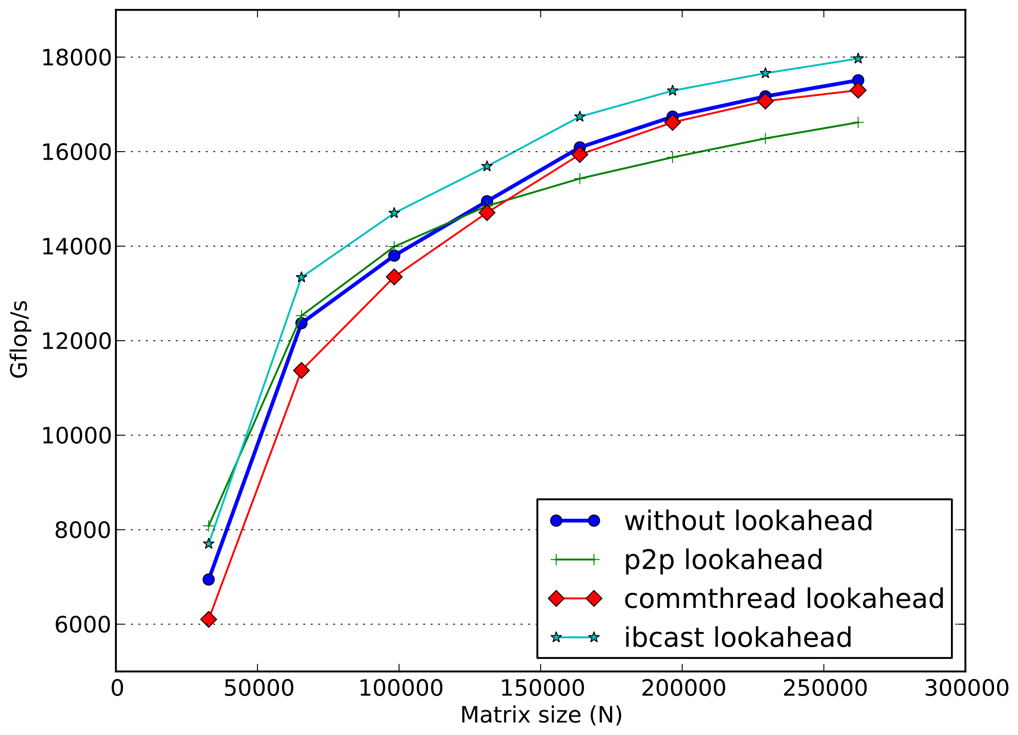


Figure 4.5: Performance comparison of look-ahead techniques for different problem sizes.

chronous progress thread. Complexity of the HPL computation O(N^3) vs. communication O(N^2). So, we differentiate two regions:

1. For small problem size, the ratio communication computation is significant and overlapping improves overall performance. The P2P broadcasting shows that the best performance improvement is due to non optimized broadcast operation in MPI library.

2. For large problem size, the P2P and communication thread approaches perform worse than non overlap version, because explicit polling and implicit OS interrupt interrupt large DGEMM kernel. DGEMM's update phase results in lower IPC and lookahead is useless. Ibcast successfully overlaps the update phase and improves results for up to 1.2%.

4.5 Conclusion

The chapter compares different techniques for overlapping collective communication and computation using MPI-3 programming model. The C-CUBE synthetic benchmark contains three different overlapping techniques of broadcast operation (hand tuned point-to-point, communication thread, and non-blocking collective) for small and large messages. C-CUBE defines three types of kernels: compute-bound, memory-bound, and hybrid-kernel. The chapter analyzes the use of the asynchronous progress thread on Cray MPICH2 and shows benefits of using it. We show that presence of the progress thread improves the progress of asynchronous broadcast without performance degradation of computation, while the absence of the progress thread degrades the performance and the communication operation starts when the MPI_Wait starts.

The new non-blocking collectives in MPI-3 standard simplify development of MPI applications and enable higher programming productivity, and at the same time provide better performance than manual overlapping approaches. We show that asynchronous broadcast collective combined with hybrid-kernel outperforms P2P and communication thread overlapping techniques.

In the case of HPL benchmark, the use of MPI_Ibcast simplifies the look-ahead code because the update phase of the benchmark is not interleaved with message probing and improves overall performance up to 13%.

The execution time of MPI_Ibcast indicates non-optimal implementation of the Cray library for a large message because the hybercube broadcast implementation and MPI_Bcast runs 4.57 times faster than MPI_Ibcast.

The Chapter 4 presents overlapping techniques and their pitfalls. MPI/OmpSs programming model should allow to use all overlapping techniques and try to merge them with the OmpSs runtime where programmers would not explicitly deal with issues presented in this Chapter.

5 Design and Implementation of the MPI/OmpSs Programming Model

This Chapter describes the design and implementation of the MPI/OmpSs programming model. The Chapter 5 is the kernel of the thesis and presents a new parallelism approach by combining MPI and OmpSs programming paradigm. The MPI/OmpSs programming model introduces an extension of existing programming interface and the OmpSs run-time. Portability of the new MPI/OmpSs programming models is an important feature, thus the Chapter also discusses implementations details/issues of the OmpSs run-time that supports overlapping techniques on various hardware platforms.

5.1 Taskified MPI calls

The dataflow execution model in OmpSs can be effectively used to exploit the distant parallelism that may exist between tasks in different regions separated by MPI calls. In order to achieve this, MPI calls need to be encapsulated in OmpSs tasks.

In order to allow a pure dataflow execution model, the first step consists on considering MPI calls as OmpSs tasks that consume data (MPI_Send) or produce data (MPI_Recv) in the task graph. We can encapsulate these communication requests as OmpSs tasks by specifying their inputs (for sends) and outputs (for receives). By doing so, we may rely on the general OmpSs scheduling mechanism to reorder the execution of such tasks relative to the computational tasks just guaranteeing that the dependencies are fulfilled. Assuming a sufficient number of processors for each MPI process this would have the effect of propagating the asynchronous dataflow execution supported by OmpSs within each node to the whole MPI program.

For example, Figure 5.3 shows a possible implementation of the broadcast operation, in which the original sends and receives are replaced by tasks with the appropriate input and output clauses.

With this encapsulation, the OmpSs scheduler is able to reorder the execution of communication tasks relative to the computational tasks, just guaranteeing that the dependencies are fulfilled. In this way, the programmer is relieved from the responsibility to schedule the communication requests. At the global application level, MPI will impose synchronization between matching communication tasks. The fact that each of these tasks can be reordered with respect to the computation tasks enables the propagation of the asynchronous dataflow execution within each node to the whole MPI program.

5.2 Overlapping Approaches

5.2.1 Restart mechanism

As opposed to standard computation tasks, communications tasks have an undetermined execution time, depending on when (or whether) the communication partner invokes the matching call. In addition, blocking communication calls could lead to deadlock situations [24] in an architecture where the number of threads per node is limited.

Usually to appropriately handle blocking communication calls, the programmer needs to split a blocking call into a non-blocking call to issue the communication request and a wait call to wait for the data. This separation just moved the deadlock risk

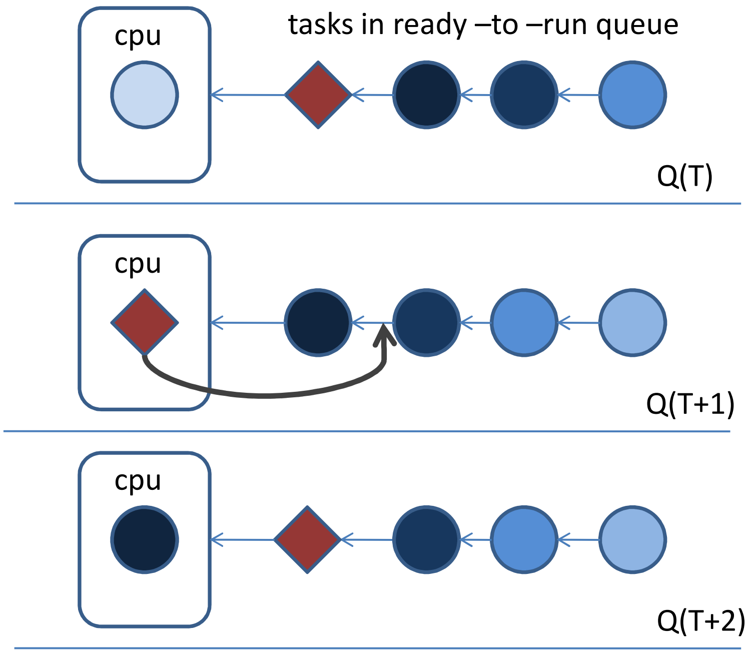


Figure 5.1: Execution stages of the restart mechanism.

mentioned above from the blocking communication call to the wait call. To solve the problem we added a new pragma in the OmpSs programming model:

```
#pragma css restart
```

See Figure 5.1,the effect of this pragma is to abort the execution of the current task and put it again in the ready queue. With this new pragma, the wait can be implemented with 1) a MPI_Test to check whether data has already arrived or not; 2) if so, the MPI_Wait can be done and data is available for OmpSs task depending on it; 3) if not, the restart pragma is executed, aborting the wait task and queuing it again in the ready queue for later consideration. The code fragments in Figure 5.2 and 5.3 show the code transformation done for a blocking receive call and for a broadcast operation.

This approach requires the explicit separation of blocking MPI calls into the appropriate sequence of their corresponding non-blocking calls. Both tasks are invoked in

```
#pragma css task output(buf, req)
void recv (<type> buf[count], MPI_Request *req){
  MPI_Irecv(buf,…,req);
}

#pragma css task input(req)
void wait (MPI_Request *req){
  int go;
  MPI_Test (req, &go, ...);
  if (go==0) #pragma css restart;
    MPI_Wait (req_recv, …);
}
void application_receive(){
  recv ();
  wait ();
}
```

Figure 5.2: Receive operation implemented by using the restart mechanism.

```
#pragma css task input(buf) output(req)
void send (<type> buf[N*nb], MPI_Request *req);

#pragma css task input(req)
void wait (MPI_Request *req);

#pragma css task output(buf, req)
void recv (<type> buf[size], MPI_Request *req);

void broadcast (int root, <type> buf){
  if (root){
    send (buf, req_send);
    wait (req_send);
  } else {
    recv (buf, req_recv);
    wait (req_recv);
  if (necessary) {
    send (buf, req_forward);
    wait (req_forward);
  }
}
```

Figure 5.3: Broadcast operation implemented by using the restart mechanismm.

sequence in the source code although if data take some time to arrive, the scheduler will launch the execution of another computational tasks. With the proposed approach, the programmer does not need to think about the relative placement of both asynchronous calls, which would force a specific schedule which may or may not be the most appropriate. Notice that the transformation described above could be even hidden inside the implementation of the MPI library or in stubs calling it, making the use of the hybrid MPI/OmpSs even more simpler and productive.

The possibility to abort and resubmit a task has several implications. First, the task must be free of any side effect on the state of the program or environment, as the whole task could be repeated a number of times that is outside the control of the programmer. Second, the runtime should not immediately selected the aborted task for execution if there are other tasks in the ready queue, as this may result in the same resource starvation and associated deadlock we tried to avoid. And third, the runtime should give these aborted tasks an opportunity to execute relatively frequently as this will result in better application responsiveness to incoming messages and may result in faster propagation of data along the critical path.

In the current implementation a task that invokes a restart primitive is inserted back in the to ready queue behind the first ready task, leaving at least a normal ready task between two restarted tasks in the list. This is done to avoid a potential deadlock in the case of two concurrent wait tasks. If the task that is restarted is marked as highpriority, it looses this condition and goes into the low priority list. Because this mechanism re-injects restarted tasks towards the head or the low priority ready queue, the net effect is that the restarted task still goes before the many possibly ready tasks in the low priority queue.

5.2.2 Communication thread

Tasks that encapsulate blocking MPI calls have an unpredictable execution time (depending on the MPI synchronization with the matching call in the remote process). This may cause deadlock if we actually devote processors to these tasks and not to advance computational tasks. In order to solve this problem, we need to ensure that every process can always devote resources to the computational task such that local progress is guaranteed. A second effect of communication tasks is that they do not make an efficient use of processor time, wasting resources while they are blocked. The aim is to maximize the amount of actual computation performed while the data transfer activities are overlapped with it. Our approach instantiates as many threads as cores in the node to execute computational tasks plus one additional thread that only executes tasks that encapsulate MPI calls. When the MPI call blocks, the thread releases the CPU and thus as many computation threads as cores can be active during most of the time (if the applications has sufficient parallelism at the node level). When the blocking MPI call completes, the blocked thread will wake up and thus contend for a core with the other threads. We aim to minimize such contention and accelerate the execution of the communication thread as this would free local dependencies, progress to the next communication task and block again. The sooner these activities are done, the faster the application will be able to progress globally.

We overload the cpu resources, where number of threads is larger than number of cores for 1, see the Figures 5.4. A simple way to achieve this is to reduce the priority of the computing threads (through a setpriority call at initialization time) and leave the communication thread at a higher priority. In this way, when the communication thread blocks, all computation threads can proceed. When the communication thread unblocks it gets to execute rapidly. Note that task priorities (as specified in the task pragma) apply to individual tasks and are used by threads when selecting tasks from

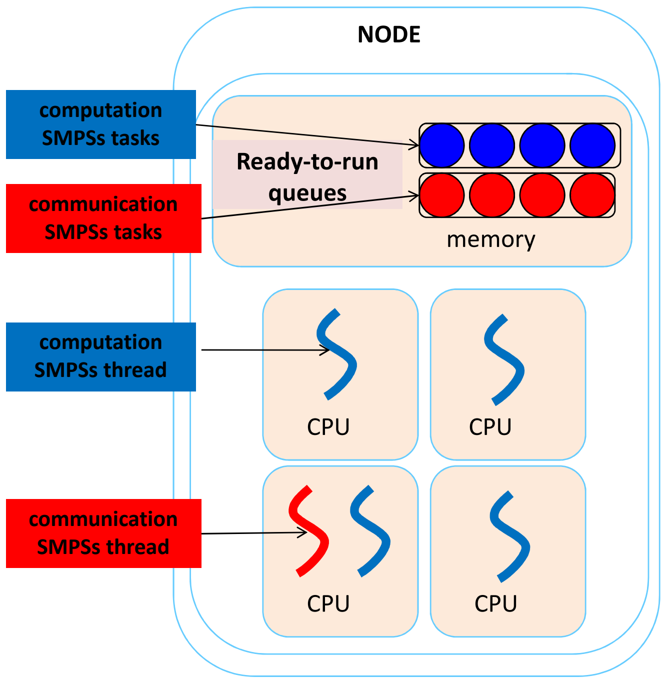


Figure 5.4: Distribution of the OmpSs threads across cores.

the user level ready queues while thread priorities determine the scheduling policy by the OS kernel. The Figure 5.5 shows a code example of the broadcast operation by using commucanition thread. A programmer just needs to mark communication task with “target” clause.

With this approach a computation thread is certainly preempted for a while but it gets its core back very soon, see Figure 5.6. Although this may be seen as a problem it is actually beneficial. The main argument is that being based on preemptions, we avoid the need to periodically poll for the completion of communication requests, decoupling the granularity of tasks from the need to ensure progress in the communication activity. This results in two benefits: first the progress of the communication activity takes place immediately, initiating the next transfer right after a message arrives or leaves a node; second, the granularity of the computation tasks can be tuned considering only its algorithmic needs thus allowing to use coarser grain tasks (e.g. invoke BLAS routines

```
#pragma css task input(buf, count) target(comm_thread)
void send (double buf[count], int count){
  MPI_Send(buf, count, MPI_DOUBLE, next_proc,...);
}

#pragma css task input(count) output(buf) target(comm_thread)
void recv (double buf[count], int count){
  MPI_Recv (buf, count, MPI_DOUBLE, prev_proc,....)
}

void broadcast (int root, double buf, int count){
  if (root)
    send (buf, count);
  else{
    recv (buf, count);
  if (necessary)
    send (buf, count);
  }
}
```

Figure 5.5: Broadcast operation implemented by using commucanition thread.

on larger matrices) potentially achieving higher computation performance (i.e. IPC).

Up to this point our proposal consisted in encapsulating individual MPI calls as tasks, assuming that the MPI calls would use a blocking mode (i.e. the MPI thread releases the CPU in the case of a blocking call). Applications may have phases where sequences of small communication requests may be intermixed with some small computations. Generating one task for each such communication and computation would produce a very large overhead. It is also quite possible that at the algorithmic conceptual level it might be more appropriate to encapsulate such series of communications and computations as a single task. Programming these tasks in our proposed model is perfectly possible as the MPI thread is a general thread and can perform also the computations intermixed within the communications. The main problem, if the granularity of communication and computation is very fine, would be the overhead of blocking, unblocking and preemptions. Waking up a sleeping thread causes OS latency and leads

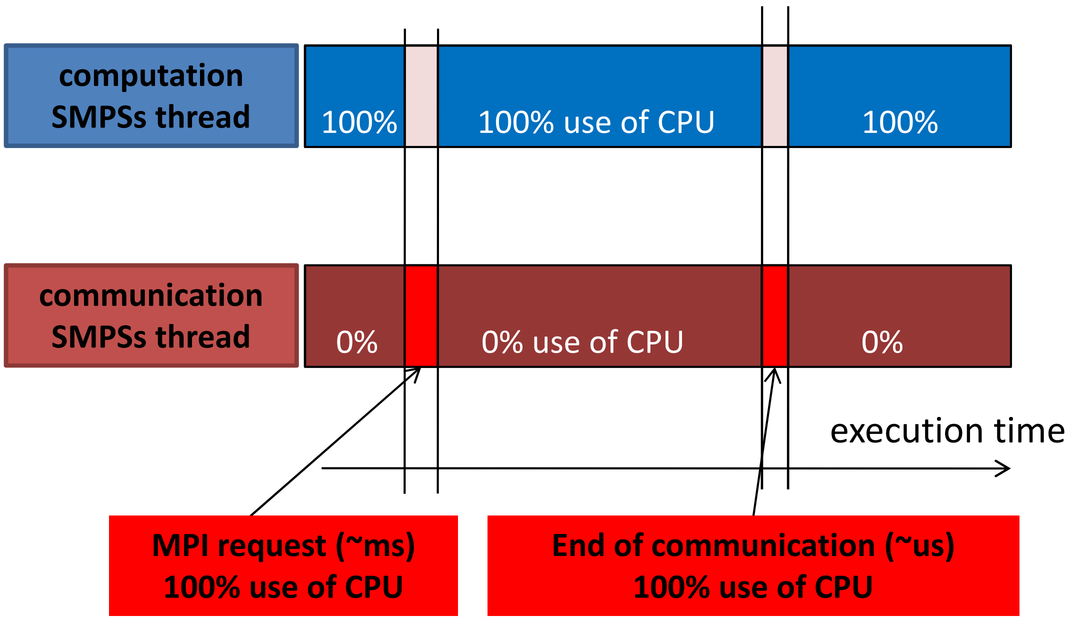

computation SMPSs thread
100%
100% use of CPU
100%
communication SMPSs thread
0%
0% use of CPU
0%
execution time
MPI request (~ms) 100% use of CPU
End of communication (~us) 100% use of CPU

Figure 5.6: Usage of cpu cycles by using commucanition thread.

to a performance drop. Our approach is to dynamically change between blocking and polling waiting mode during the execution. We have implemented in MPI a call that allows the programmer to switch the default blocking mode to polling mode within one task:

MPI_set_waiting_mode(<polling/blocking>);

This call should not be used in communication tasks that just perform a single MPI call potentially blocking for a long time; however, it can be used in tasks where a whole bunch of MPI calls are invoked in sequence.

In fact, this raises a new issue as the switch to polling mode (and back to blocking) can be done at any time within the task. Ideally, the polling mode should be used when there is a certain guarantee that the duration of the successive MPI calls will be short. When two tasks in two different processes exchange such sequences of messages, the first message in the sequence plays a synchronizing role between the tasks in the two processes. The recommended practice is then that the programmer switches from blocking to polling mode after the first blocking call. If a task actually interacts with several other tasks in different processes, the best point to switch to polling mode would be after having completed a blocking call to each other process.

5.3 MPI Serialization and Its Challanges

The restart mechanism offers automatic overlap of communication and computation and there is no need for restructuring the code. A programmer keeps focus on the algorithm while the runtime cares about performance. Still, the restart mechanism shows some performance weaknesses, because a task granularity determines message progress and delays communication operation. The approach that involves communication thread successfully avoids influence of a task granularity and increases time response of message progress. Both techniques encapsulate MPI calls inside OmpSs task and hide data dependency information from the main thread. The programmer extracts data dependencies thought direction clause of data flow programming model while a run-time does serialization of task that contain MPI calls because MPI guarantees that messages will not overtake each other.

All MPI routines are encapsulated inside the task that allow acceleration on the critical path and avoid fork/join approach by using MPI/OpenMP programming model. To avoid deadlocks and ensure correctness of the result, all task that contain MPI calls are serialized in the runtime. The OmpSs runtime recognizes task that has been marked as communication task and serialized them in order of their concurrent execution. A programmer is responsible to place communication calls in correct order assuming in order execution of the function(task) and guarantees matching, while the runtime serializes them. In the current implementation, a compiler/runtime introduces automatically serialization to the execution of communication tasks.

Serialization of communication task creates artificial dependencies in data dependency graph, while dependency between task in data flow programming model should be only real data dependency. These artificial dependencies potentially reduces parallelism and slows down an execution of the critical path.

In particular, porting codes to the MPI/OmpSs framework shows that linear algebra algorithms contain several independent execution paths, each with its own communication operations.

The following example discusses the widely used HPL is a numerical linear algebra algorithm for solving LU factorization. The data dependency graph version of the HPL is shown in the Figure 5.7. Here we are going to discuss data flow graph of one MPI process, where nodes present tasks inside the process.

Figure 5.7 shows the data dependencies graph of the HPL benchmark. Nodes are instances of OmpSs task created at the very beginning of execution by a master thread. Numbers inside the nodes are unique task IDs of the OmpSs runtime.

Nodes filled with black color are OmpSs tasks that preform a computation operation, while red nodes contain MPI routines and execute communication operations. Solid filled arrow represent data dependencies between task inside a shared address space and thus define an execution order. While solid arrow show real data dependencies, dashed arrows represent execution order of communication tasks. Artificial dependencies between communication tasks are not visible in the source code by looking at the arguments of function, the runtime automatically introduces them. If we look at the graph, clearly we can recognize the critical path in the current example. In order to progress as deep as possible to the graph and free more task which implies more parallelism, the execution order should be 1, 2, 5, 8, 9 , 12 etc. Serialization of commutation task does not allow natural execution order of the critical path and forces the following order : 1, 2, 5, 8, 3, 4 etc. It means that the serialization delays the critical path for the execution time of non-urgent communication tasks. Thus, a lack of parallelism does not allow an effect overlapping of communication and computation.

The serialization shows bottelnecks and introduces new challanges. The thesis sets goals that a new approach has to satisfied in order to deal with the bottlenecks of the serialization:

1. First, it has to deal with out-of-order execution of data independent communication tasks. Dependency between commucanition task should only be due to natural data dependencies expressed thought arguments of functions in the source code.

2. Second, programming productivity has to stay high. This avoids code restructuring on the top level that is outside of task, introducing extra logic inside a communication task are not acceptable as well. Programmer still needs to be

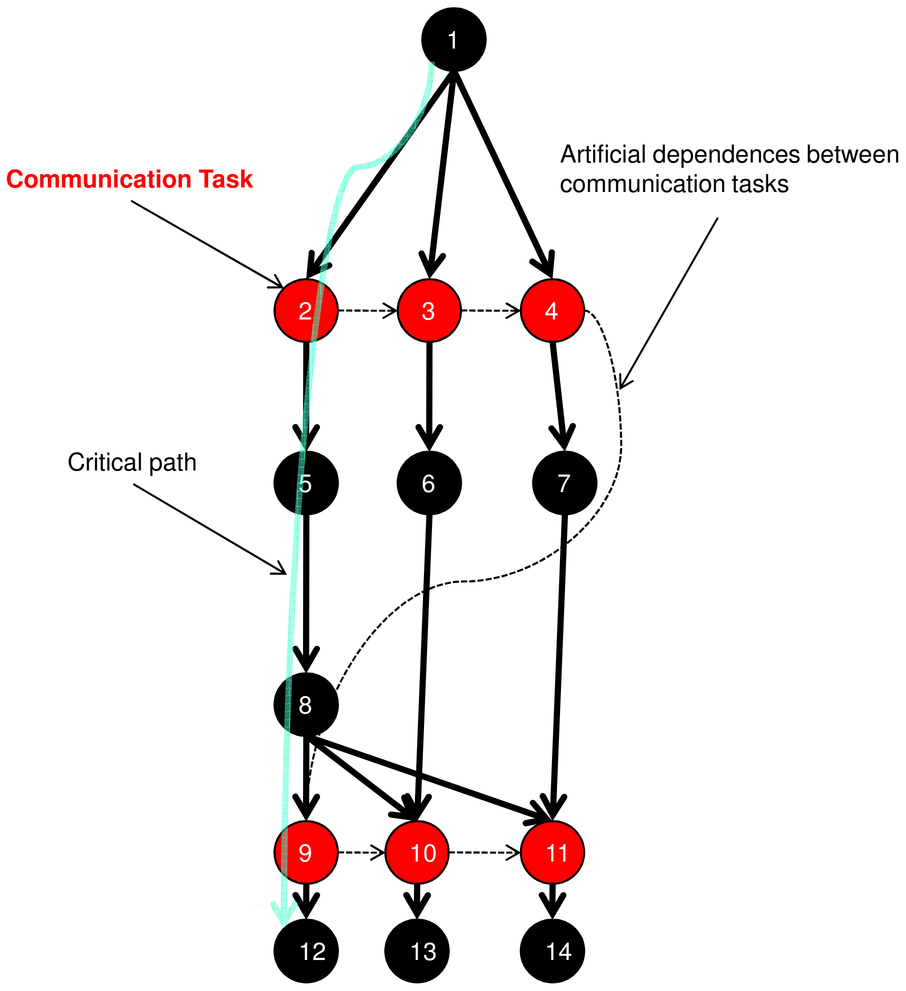


Figure 5.7: Data Flow Graph of the HPL; MPI task serialized

only focus on expressing the algorithm where the runtime cares about scheduling of these tasks, execution order and overlapping communication and computation. A new approach should be implemented below application level and hide complexity from the programmer.

3. Third, high performance stays as a main goal of high performance programming model, the benefits from out-of-order execution of communication task and increase of parallelism has to be bigger than the implementation overhead of the

```c
int OmpSs_MPI_Recv(void *buf,int cnt,MPI_Datatype type,int src,int tag,MPI_Comm comm,MPI_Status *s)
{

  int ierr;
  int done = 0;

  ierr = MPI_Irecv(buf, cnt, type, src, tag, comm, &request);
  ierr = MPI_Test(&request, &done, status);

  while(!done){

    nanos_yield();
    ierr = MPI_Test(&request, &done, status);

  }
  return ierr;
}
```

Figure 5.8: Implementation of OmpSs_MPI_Recv.

new approach.

5.4 OmpSsMPI Library

The thesis proposes the OmpSsMPI library which build on top of the MPI library. OmpSsMPI keeps the MPI interface e.i. names and arguments of OmpSsMPI calls are identical to the corresponding MPI calls. Programmer just needs to add a prefix of encapsulated MPI calls within OmpSs tasks to OmpSs_MPI. This is the only effort done by programmers which makes this approach easy-to-use. Tagging a task with the special clause is not necessary. The OmpSs library informs the OmpSs runtime about the presence of MPI call and the runtime internally marks the task as a communication task.

The Figure 5.8 shows the pseudo code of OmpSs_MPI_Recv. The implementations of all blocking calls follow the same structure. The OmpSs_MPI_Recv routine calls the ompss_communication_task routine that detects the parent task of the OmpSs_MPI and marks as the communication task. Therefore appropriate non-blocking call issues communication test and the loop with MPI_Test and nanos_yield() follow the non-blocking call. The loop structure is very similar to the structure used in the restart mechanism. Instead of the restart clause, the OmpSs library call nanos_yield().

Once nanos_yield() is called, the OmpSs scheduler plays the key role. The OmpSs scheduler contains a special ready-to-run queue for communication tasks. The communication task which calls nanos_yield() imminently releases the core and goes to the end of the queue and another communication task takes computing resource. The

OmpSs scheduler considers priorities of task and round-robin policy in order to select a next task.

From the execution point of view, this approach has two advantages: first, communication tasks can be executed out-of-other because there is no internal blocking call that would lead to a deadlock. Thus, the OmpSs scheduler selects the natural critical path of application and accelerates the execution. Second, the wrong ordering od MPI calls does not lead to a deadlock. The programming model becomes deadlock free. Instances of the OmpSs runtime across different MPI changes the order of MPI calls and match them.

5.5 Implementation Issues on Various HPC platforms

In order to prove the potential of the MPI/OmpSs programming model, the deployment of MPI/OmpSs to various plaforms is crucial. The MPI/OmpSs has also been a basic programming platform of the EU project,called TEXT, where a main goal is evaulating real application on various clusters. Conceptly proposed overlapping techniques should increase performance, in reality system a single implentation of the overlapping technique do not fit to all machines. This section describes implementation challanges across different platforms and explains implementation modification.

5.5.1 Single threaded SMP nodes

AA node with four physical cores was the first platform where the OmpSs environment has been developed. The node uses a standard Linux system and MPI library that is not the thread-safe. Hardware and software environment pushes to the implementation of the restart mechanism as the overlapping technique. The OmpSs runtime loads four threads per load and bind to the cores. The restart mechanism works fine for a small messages but large messages did not perform as suppose to do due to MPI implementation of the system. MPI implementations sending the large messages as a sequence of the small messages, chunk-by-chuck technique, and a single call of MPI progress engine triggers a single chuck. All MPI calls internally call once the MPI progress engine. It means a large message that contains 10 chucks need 10 MPI_Test calls to locally finalize the send operation.

The MPI implantation issue forces the usage of the communication thread and overload the node with one thread more than the number of cores. Dedicating a single core for the communication thread is very expensive and reduces the maximum theoretical efficiency to 75% because only 3 out of 4 cores do useful computation.

Placing the communication thread and one of computational thread to the single core introduces new challenges as a time sharing policy between threads. Default setting of the OS thread scheduler treat all threads equally and does not prioritize commutation operation that lies on the critical path. Thus the OmpSs runtimes increases the priority of the communication thread among computational threads. High priority of the communication thread threaten in over-usage of the core and cycle wasting. Setting the MPI waiting mode to the blocking mode avoids starvation in the scheduling and the communication yields the core when it waits for the message.

HPC system usually uses a standard Linux scheduler with a complex scheduling policies. The OmpSs runtime implementation gives hints to the OS thread scheduler regarding OmpSs threads and the OS scheduler can make optimal decisions and increase efficiency of the cycle usage.

5.5.2 Hyper-Threading SMP nodes

Nowadays it also is very rare to find HPC machines without Hyper-Threading(HT) support for each logical core. The core usually contains two or more HT with its own general-purpose, control, and machine state registers. Two threads that require different architectural states could share the single logical core, where the core enables the HT support, without performance degradation. Nevertheless, enabling HT for HPC applications and launching number of threads equal with number of hardware threads lead to performance degradation because logical threads compute for access of shared resources, as caches. HT utilization depends on characteristics of the application. Runtimes and OS schedulers are not aware about characteristics of the application and a lack of information stops them to make a smart decision and to benefit of HT.

The MPI/OmpSs runtime cannot characterize computational tasks and stick to the strategy where the number of computational threads are equal to the number of logical threads. On the other hand, the OmpSs communication thread benefits of HT technology. Most of the time the communication thread stays in sleeping stage and when it wakes up it always performs memory intensive operation. Asynchronous data transfer from memory to NIC would not disturb compute intensive task and does not intercept execution of computational thread. As we saw the single threaded SMP nodes requires the control of the OS thread scheduler, while HT-SMP nodes keep the same priority of all threads. Computation threads are bound to logical core and the communication threads flies between logical cores. The OS thread scheduler never blocks the communication thread which avoids the waking-up overhead faced in the single thread SMP nodes.

Described rutime configuration for HT-SMP nodes is used in the evaluation section of the paper [TODO cite TEXT paper] and showed the best results among tested configurations.

5.5.3 Blue Gene nodes

The more challenging issues have been found on the Blue Gene/P platform, due to the fact that the compute nodes do not have a regular full Linux system, but a lighter one, known as the Compute Node Kernel (CNK). The main issues were related to memory management, kernel thread scheduling and to specifics of the MPI library.

This platform presented major difficulties given that the compute nodes do not have standard Linux kernel. In particular, the topics that we had to address were:

Memory management: the mmap functionality that was used for general purpose Linux could not be used. In order to optimize the usage of memory, we had to modify the memory manager of the OmpSs runtime.

- Kernel thread scheduling: The kernel in the compute nodes does not support general-purpose multiprogramming within the node, although it does support several kernel threads per processor. In this case, the context switch has to be an explicit request from the user code.

- On Blue Gene, the MPI library does not support blocking mode for receiving a message. We implemented our MPI blocking calls that mimic two different modes (polling and blocking). Switching between these modes, a programmer could increase efficiency of MPI/OmpSs applications.

The Blue Gene implementation requires following three modification:

1. Blue Gene/P system has 2GB or 4GB of physical memory per node and there is no virtual paging. A memory management that increases the memory usage in standard Linux machines could quickly consume the memory available for applications. In order to increase efficiency of the use of memory, the MPI/OmpSs runtime implements a memory manager that renames parameters trying to reuse memory already allocated while its amount is below the mmap threshold. In this environment, a Linux machine does not use the physical memory requested by mmap if there is not a real need, while the CNK1 Blue Gene allocates physical memory requested by mmap, whether it is used or not.

   In the MPI/OmpSs runtime for Blue Gene the memory manager was disabled and a pair of calls to malloc/free take care of allocating and deallocating memory.

2. CNK can be configured to allow multiple application threads per core. There are limitations regarding CNK support for multiple application threads per core that do not apply when running on a regular Linux kernel. A user configures the number of application threads per core using the environment variable which can be between one and three. The CNK pins the thread to the core that has less threads running on it. Once the thread is bound, it can not switch between cores. The system does not automatically switch threads between cores, but the user code may control thread scheduling through a sched_yield() system call, signal delivery, or futex wakeup.

   MPI/OmpSs applications use as many computation threads as the node has cores, plus one extra thread for communication tasks. Therefore, MPI/OmpSs runtime requires support for time sharing between the communication thread and one of the computation threads. Linux provides support through the standard Linux scheduler. Besides, the runtime contributes to improve the thread scheduling by using system calls for setting thread priorities, and POSIX semaphores for switching threads between blocking and ready-to-run queues. For a Blue Gene system, the MPI/OmpSs runtime is only responsible for the thread scheduling control. The CNK reacts on sched_yield calls from the threads. Each thread checks the ready-to-run queues for tasks, if there is no runable task, the thread invokes sched_yield(). In order to mimic thread priorities, the computation threads call sched_yield() after the execution of each task while the communication thread only calls the sched_yield() when the ready-to-run queue for communication tasks is empty. This gives higher priority to the communication threads and accelerates execution of communication tasks that usually lie on the critical path.

3. On Blue Gene, the MPI implementation does not support blocking mode. In order to mimic blocking mode, we used a pair of non-blocking MPI call/MPI_Test call and a call to sched_yield between them. The non-blocking MPI call issues the communication request, while the MPI_Test checks whether the data has arrived or not: 1) if so, the MPI_Wait can be called and the data is available for OmpSs tasks waiting for them; 2) if not, the sched_yield call yields the CPU to a computation thread and the communication thread sleeps while the computation is done. The experience shows that often calling sched_yield after each MPI_Test check is not good for small messages. We decided to repeat the MPI_Test call several times before calling the sched_yield. MPI/OmpSs runtime uses a self-tuning technique to define the number of MPI_Test repetitions and this number depends on the application.

6 Evaluation of automatic overlaping and reordering through MPI/OmpSs

The Chapter 6 describes HPC applications used for experiments, evaluates and compares the MPI/OmpSs programming model withing a node and a large-scale machine with MPI and OpenMP.

6.1 High Performance Linpack

The HPL [16] is the most widely used benchmark to measure the floating-point execution rate of a computer and the basis to rank the fastest supercomputers in the TOP500 list [48]. Although the kernel simply solves a system of linear equations, it can be considered a good representative of a significant set of applications. The parameters used to configure its execution lead to changes in terms of: global computation and communication ratio, load balance, amount of fine grain (small frequent messages) communications, performance of inner sequential computation, etc. This section describes the techniques used in the parallelization of the HPL benchmark and shows their impact in the code structure and readability, as a motivation for the proposed hybrid MPI/OmpSs approach.

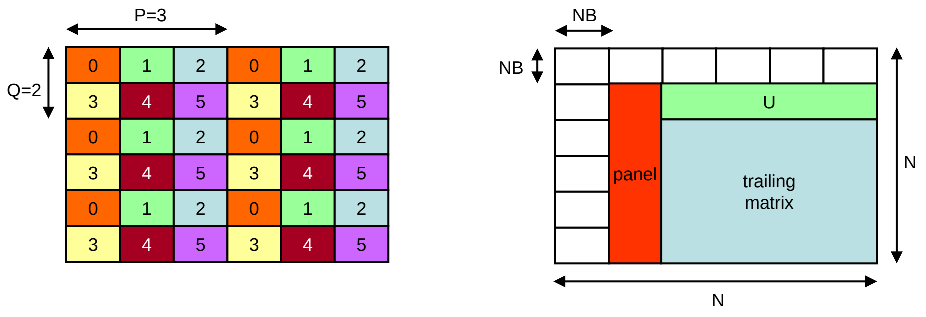


Figure 6.1: P by Q partitioning of the matrix in 6 processes (2x3 decomposition) (a) and one step in the LU factorization (panel, U and trailing matrix) (b).

6.1.1 HPL Parallelization

The HPL benchmark implements a LU decomposition with partial pivoting. The elements of the coefficient matrix are double precision floats initialized with a random distribution. The matrix to be factored has NxN elements and it is decomposed into blocks of size NBxNB, which are distributed onto a grid of PxQ processes. Due to the triangular nature of the algorithm the blocks are distributed among processes in a cyclic way, as shown in Figure 6.1, to achieve load balance. In a typical PxQ partition, every process will have a set of blocks regularly spaced over the original matrix. These blocks are stored contiguously in a local matrix, which can then be operated on with standard BLAS routines. Of course, highly optimized versions are used in order to achieve a high percentage of processor peak performance. An iteration of the main loop of the overall algorithm consists of three main steps: panel factorization, broadcast, and the update of the trailing submatrix.

When the computation of the panel factorization is finished, the panel needs to be broadcast to the other processes along the Q dimension so that they can perform the update of the trailing submatrix. This broadcast can be implemented using the

```
#define NPANELS N/NB*Q)
#define root (j%Q==my_rank)

double A[N/P][NPANELS*NB];
double tmp_panel[N/P][NB];
int k=0;

for( j = 0; j < N; j += nb ){
  if (root){
    factorization (&A[k*NB][k*NB], tmp_panel, k);
    k++;
  }
  broadcast (root, tmp_panel);
  for(i = k; i < NPANELS; i++ )
    update (tmp_panel, &A[k*NB][i*NB],k);
}
```

Figure 6.2: Simplified version of the main loop in HPL.

MPI_Bcast call if the machine provides an efficient implementation of this primitive (as for instance in Blue Gene [3]). Alternatively, several methods are provided in the HPL distribution to perform the broadcast by circulating the data in one or several rings of point-to-point communications. The LU factorization is done by iteratively applying these two steps on the trailing submatrix. Figure 6.2 shows the pseudo-code for a simplified version of the main loop in HPL.

The main objective of the look-ahead technique is to accelerate the execution of the critical path in the computation and to overlap communication with computation. The panel factorization process lies in the critical path of the application. When the panel in iteration j has been factored by processes in column q=j%Q and broadcasted, the globally next urgent job to perform is the factorization and communication of the panel in iteration j+1 by processes in column (q+1)%Q. The HPL code includes a lookahead option that performs this optimization. As soon as a column of processes q receives a panel factored by its previous column, they update, factor and send the next panel before updating the rest of panels also owned by this column of processes. In

```
double tmp_panel[2][N/P][NB];
double *p[2];

p[0] = tmp_panel[0][0][0];
p[1] = tmp_panel[1][0][0];
k = 0; j = 0;

if (root){
  factorization(&A[k*NB][k*NB], p[0], k);
  k++;
}
broadcast_start(root, p[0]);

for (j = nb; j < N; j += nb){
  broadcast_wait(p[0]);
  if (root){
    update (p[0], &A[k*NB][k*NB], k);
    factorization (&A[k*NB][k*NB,], &p[1], k);
    k++;
  }

  broadcast_start(root, p[1]);
  for (i = k; i < NPANELS; i++)
    update_and_broadcast_progress (p[0], &A[k*NB][i*NB], k, root, p[1]);
  p[0] = p[1];
}

broadcast_wait(p[0]);
for (i = k; i < NPANELS; i++)
  update (p[0], &A[k*NB][i*NB], k);
```

Figure 6.3: Version of the HPL pseudocode with look-ahead equals to one.

this way, the transmission of the data can be advanced and the global critical path is accelerated. Figure 6.3 shows the pseudo-code for a simplified version of the main loop using a look-ahead degree of 1 iteration. Notice that introducing this optimization requires significant changes in the source code, not only in the main iterative loop, but also in the different routines called inside this loop (not shown in the figure). For example, functions such as update_and_broadcast_progress should include periodical probing calls and message retransmission if needed, increasing internal code complexity. In addition to that, the programmer has to explicitly allocate several data structures to temporarily hold the broadcasted factorized panels. Higher degrees of look-ahead require further modifications in the code and allocation of data structures.

If look-ahead is turned off the whole update loop can be executed as a single BLAS invocation that would result in a better execution performance (i.e. IPC, instructions per cycle) and might partially compensate for the lack of overlap. It would be very important to improve the overlap without penalizing the IPC of the sequential parts of the code.

The rationale for a two-dimensional data distribution originates from the actual amount of data to be transmitted at every step and the potential concurrency of such transmissions. A value of P larger than 1 implies that different blocks of the panel can be sent concurrently as each of the P processes has one part of the panel. However, this also introduces additional communication in the factorization step. These communications are of much finer grain than those in the panel broadcast phase. The value of P introduces a clear trade-off between communication and synchronization overhead in this phase and the parallelism to execute this phase which lies in the critical path. The case of P=1 is a special situation: it avoids all communications in the factorization phase as well as the need to broadcast the U submatrix (see Figure 6.2) in the update phase, but has to pay for a long sequential time of the factorization phase and long communication chain for the panel broadcast.

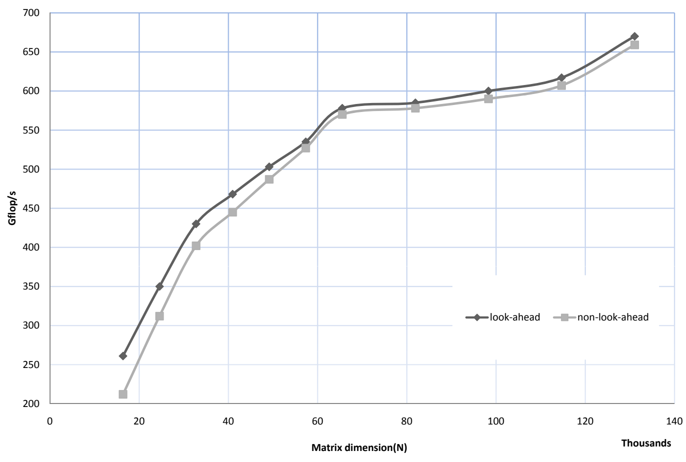
6.1.2 HPL Performance Analysis

Performance of a system depends on a large variety of factors. Achieving the best performance requires well done analysis of these factors. HPL offers the list of 31 tuning parameters that defines how the problem is to be solved. Varying these parameters HPL stresses some parts of the system more than others and also gives a good representation of some MPI scientific and technical applications. We did analysis on 128 processors assigning the most important tuning parameters: problem size (N), block size (NB), data decomposition (P and Q), overlapping communication and computation by using look-ahead technique.

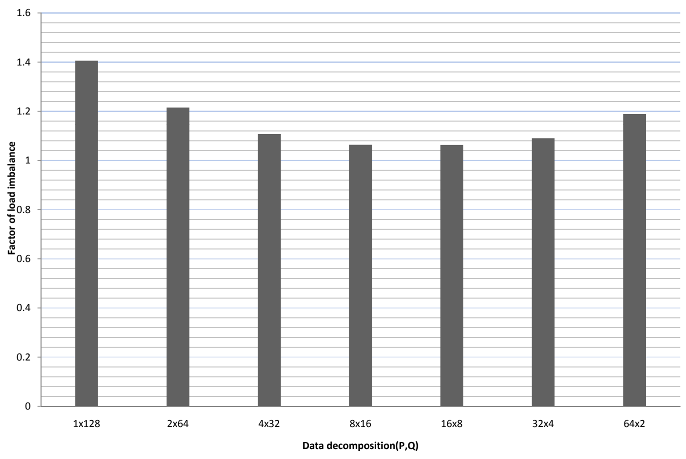
The largest problem size (N) that fits in memory gives the best performance of the system. In effect, matrix dimension (N) defines the ratio between communication and computation. For small problem size, HPL is very sensitive to network performance, increasing the problem size communication and computation increases as well, but computation increases much faster and the communication overhead decreases. For very large matrix, the influence of network performance drops significantly. Figure 6.4 (a) shows performance results for various problem sizes using look-ahead technique and HPL version without using look-ahead technique. The HPL version with look-ahead decreases the communication overhead by overlapping communication and computations and gives better performance results for small problem sizes, while both versions give almost the same performance for large problem size.

Proper block size (NB) determines to data distribution, computation granularity (probing granularity for look-ahead techniques) and performance of BLAS routines. Large block sizes tend to cause load imbalance and limits probing for message, while small block sizes increase internal blocking factor of BLAS routines and thus decreases efficiency of matrix multiplication. Figure 6.4 (b) presents sensitivity to various NB. For this experiment we used N=65536 and P=8 and Q=16, as such NB=128 gives optimal interaction between data distribution and computation granularity. Variables P and Q determine the data distribution. For 128 processors, possible grids are (P,Q)=(1,128), (2,64), (4,32), (8,16), (16,8), (32,4), (128,1) , these respond for load balance and scalability of the algorithm. In order to analyze load balance, we measure the total execution time of the BLAS GEMM routine for matrix multiplication in the update phase, as the most expensive computations in the application, Figure 6.4 (b) shows that a good load balance prefers square grids. A factor for load imbalance is ratio between the longest and the shortest execution time of computation obtained from MPI processes.

Processes do the panel broadcast operation over Q-processes, so large value Q may limits scalability of the algorithm. Look-ahead technique addresses this issue and tries

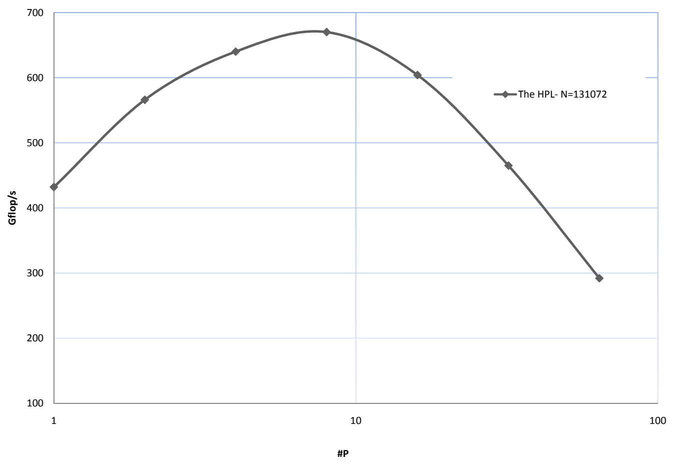


(a) Look-ahead turned ON and OFF

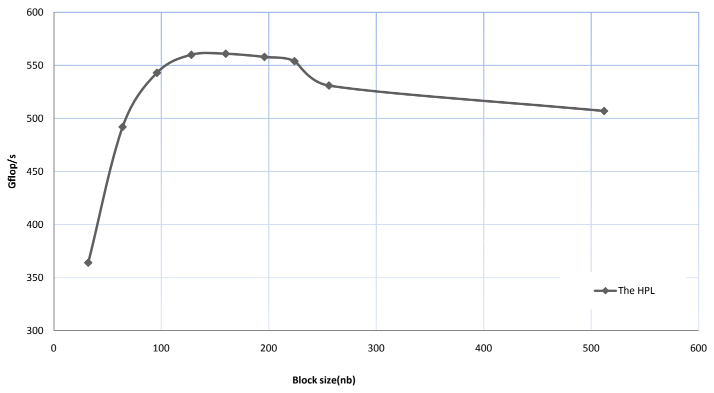


(b) Load Balancing in HPL

Figure 6.4: Performance analysis of HPL of look-ahead and load balancing.



(a) Peformance of HPL for different data decomposition



(b) Peformance of HPL for NB size

Figure 6.5: Performance analysis of HPL for different problem size and block size.

to hide the cost of the broadcast operation. We have already seen how increasing the size of problem reduces communication overhead. Larger P value increases number of fine grain communications and communication latency causes performance degradation.

In order to test our approach, we analyzed two interesting cases: first (1,128) decomposition where coarse grain communications do not appear and the communication overhead only comes from the broadcast operation; second (8,16) decomposition that gives the best performance and contains a good ratio between large messages for coarse grain communication(broadcast operation) and small messages for fine grain communication (panel factorization). These cases represent behavior of some MPI scientific and technical applications offering challenge to our approach.

6.1.3 HPL Single Node Performance Evaluation

In this section, the thesis compares the performance of pure MPI, OpenMP and OmpSs versions on a single node. The programming model comparison uses the HPL as the case study.

Parallelization of the HPL by using MPI has been already explained. According to the performance references of the HPL, an evaluation run loads the same number of threads as the number of physical cores, choose an optimal block size for the BLAS library and make 2D data decomposition where P is less than or equal to Q.

On the other hand, the OpenMP HPL version keeps the structure of the serial code without any explicit communication calls. The OpenMP version of BLAS routines performs embarrassing parallel operatio. The serial HPL becomes the OpenMP HPL version, the dynamic load balancing shows advantages against MPI static distribution but the MPI HPL performs better than OpenMP HPL. The reason lies in the fact that the fork-and-join execution model of OpenMP is not well suited to express the algorithm of the HPL and causes lack of parallelism. Some parts of the algorithm cannot be parallelized, in that moment the main thread is busy while others are idle.

6.1.4 OmpSs HPL

In oder to introduce the OmpSs HPL, we comment the sequential HPL and then describe how the HPL code can be easily restructured in the data flow manner using the

OmpSs programming paradigm.

Instead of deleting MPI calls from the source code, we use the original HPL for (P,Q)=(1,1), which makes the HPL is a single MPI process application. By setting input parameters to (1,1), the main loop only contains the factorization phase and the update phase, see the Figure 6.7 . THe internal complexity of these phases is much lower than the MPI HPL. The update phase is made of two steps: the first step pivots the upper panel of NB rows and updates these NB rows. The second step needs to calculate the trailing submatrix based on the matrix product of the factorized panel,as well as the upper panel and the old value of the trailing submatrix.

```
#pragma css task input(A) output(tmp_panel)highpriority
void factorization (double A[N/P][NB],double tmp_panel[N/P][NB]);

#pragma css task input(tmp_panel) inout(A)
void update (double tmp_panel[N/P][NB], double A[N/(P*NUM_OF_SMP_CPUS)][NB]);
```

Figure 6.6: Taskifying the factorization and the update.

Programmer only needs to inserts the pragma specification for factorization and the update phases as shown in Figure 6.6. The factorization is performed by a single task whose input is the updated panel of a previous iteration and whose output is the factorized panel for the current iteration.

Figure 6.7 shows task-dependency graph of the OmpSs HPL. The update phase is partitioned in a set of tasks, each of them taking as input the factored panel (either produced locally or received) and a subset of the panels to update. Thus each update task contains the pivoting and the update of trailing submatrix of one panel. Since the factorization lies on the critical path of the overall algorithm, we use the priority qualifier as a hint for the runtime. The update of the first panel also lies on the critical path and we renamed this update call to the urgent_update with highpriority flag. This accelerates the critical path of the overall algorithm. A simple sequential kernel of the HPL with mentioned pragma specification supported by the OmpSs runtime becomes the OmpSs HPL. Nodes correspond to the different function invocations and arrows to dependence relationships between them. The update tasks are independent with one iteration of the main loop. So, the update phase is easy to parallelize using any

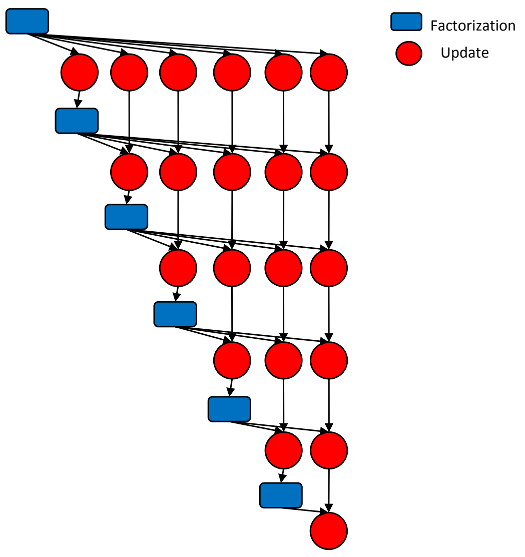


Figure 6.7: Data dependency graph of the OmpSs HPL.

shared memory programming model, like OpenMP. While OpenMP is based on fork-and-join concept and has to pay long sequential execution of the factorization phase. The OmpSs programming paradigm supports out-of-order execution of tasks which results overlapping the factorization task within update tasks. The urgent_update and factorization tasks are executed as soon as possible. In the OmpSs HPL, only the execution of the first factorization task can not be overlapped with non-urgent update tasks, while OpenMP will pay this penalty in every iteration.

6.1.5 Results

We compare the programming paradigms on three different SMP nodes: Intel Nehalem and AMD Istambul based on UMA architecture and ALTIX machine are based on NUMA architecture.

- Intel Nehalem - a node contains two 4-core Nehalem Xeon E5540 core 64-bit processors with 8 MB shared cache per socket and a frequency of 2.53 GHz. The node contains 24 GBytes of main memory. Four FLOP per cycle. Theoretical peak performance is 80.96 Gflop/s. SUSE Linux the kernel 2.6.16 and page size 4Kbytes. As a native compiler OmpSs uses Intel C compiler version 11.1 as well as a native compiler for the MPI version. MPI applications uses the SGI Message Passing Toolkit (MPT) as an MPI implementation. As a BLAS library we used Intel MKL version 10.1.

- AMD Istanbul - A node contains two 6-core Opteron 2435 Istanbul 64-bit processors with 12 MB shared cache per socket and a frequency of 2.6 GHz. The node contains 48 GBytes of main memory. Four FLOP per cycle. Theoretical peak performance is 124.8 Gflop/s. SUSE Linux the kernel 2.6.16 and page size 4Kbytes. As a native compiler OmpSs uses GCC complier version 4.3 as well as a native complier for the MPI version. MPI applications uses the OpenMPI as an MPI implementation. As a BLAS library we used GotoBLAS 1.24.

- ALTIX 4700 - SMP cluster contains 16 blades each blade contains two Itanium2 9030 Montecito 64-bit dual-core processors with 8MB shared cache 1.6 GHz. Four FLOP per cycle. The blade contains 24 GBytes of main memory. Theoretical peak performance is 409.6 Gflop/s. SUSE Linux the kernel 2.6.16 and page size 16Kbytes. As a native compiler OmpSs uses Intel C complier version 11.1 as well as a native compiler for the MPI version. MPI applications uses the SGI Message Passing Toolkit (MPT) 1.23 as an MPI implementation. As a BLAS library, we used GotoBLAS 1.24.

The work compares the performance results of OmpSs and MPI HPL in terms of Gflops for different problem sizes and comments the scalability of different programming models.

In order to discuss the results, we introduced the following two concepts :

- **theoretical peak performance** - theoretical peak performance of one core is the product of CPU speed and number of floating point operation that can be execute in one cycle; for N cores theoretical peak performance is N times theoretical peak performance of the single core.

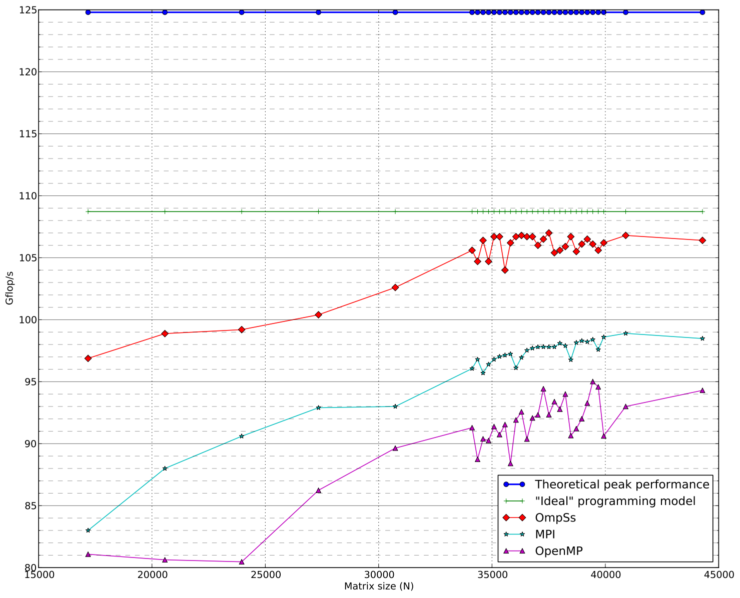
- **ideal programming model** - theoretical peak performance is unreachable for the HPL due to the nature of the algorithm; the ideal programming model is the programming model where the performance of parallel version is the product of number of cores and the performance of the serial implementation obtained from the single core. It means that the ideal programming model makes a serial code parallel and perfectly balanced.

OpenMP HPL shows lower performance than MPI HPL because of its fork/join concept. In order to prove it, we checked the performance of sequential HPL with multithread BLAS library, where the number of threads is equal with number of cores. Thus the HPL behaves as a well written OpenMP program where all parallel section are parallelized in optimal way. The plot in the Figure 6.8 shows how the performance of OpenMP HPL falls behind the MPI HPL. The plot illustates why MPI HPL has been used for evaluation the SMP clusters. Furthermore, the thesis only focuses on the basic comparison between MPI HPL (the best known performance version of HPL) and the OmpSs HPL.

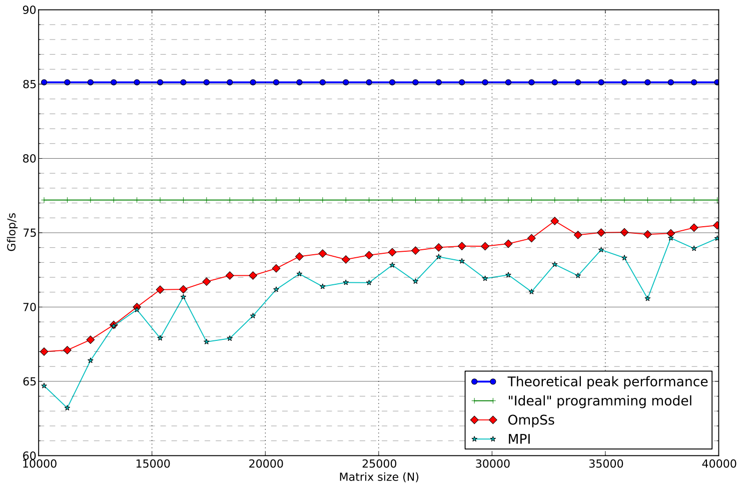
Regarding efficiency of parallel programming model, studies take the theoretical peak as a reference. This approach is not fair due to the sub-optimal efficiency of the serial version. A fair evaluation of a programming model considers ideal programming model as the reference. Ideal programming model makes an assumption that all algorithms could be 100% paralleized which is a optimistic assumption. Unfortunately,it is impossible to compute the potential of parallelism for all algorithms.

Figures 6.8 show the performance rate (Gflop/s) for the MPI HPL and OmpSs HPL running on two different UMA machines (Nehalem and Istanbul). Y-axis presents different problem sizes. Considering (P,Q) decomposition, the MPI HPL uses (2,4) for Nehalem and (3,4) for Istanbul. Experiments use the optimal parameters for NB and N depend on BLAS library. The GotoBLAS library shows the best performance for NB=242, while Intel MKL shows the best performance for NB=256. Ripples appear in the curves when the problem size is not a multiple of NB and performance drops slightly. OmpSs performs better than MPI for all problem sizes and shows the same efficiency on both machines. On the other hand, MPI version shows better efficiency on Nehalem than on Istanbul node, this is not limitation of the hardware as it was suggested [1]. Insted, the decomposition (2,4) suits better than (3,4) and a programming model limitation make the performance difference.

Figures 6.9 shows scalabilty of programming models on Nehalem and Instabul in terms of GFlop/s for different number of cores. The OmpSs curve almost overlaps the

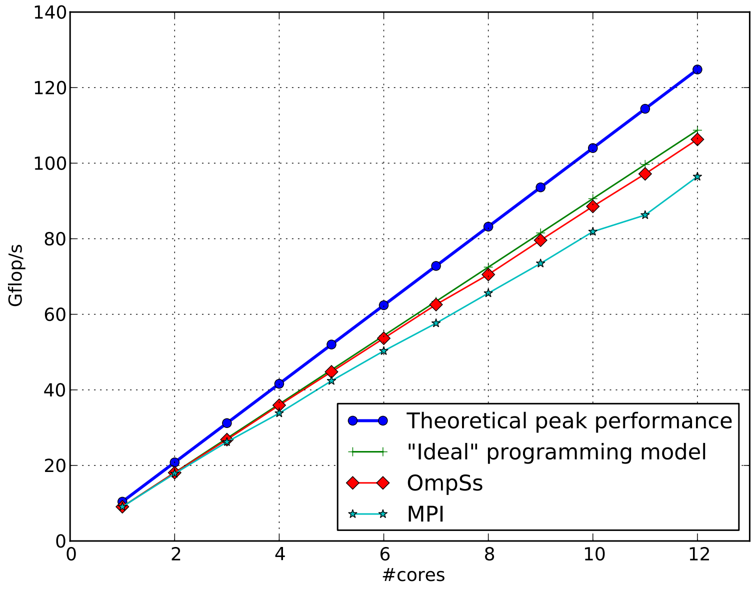


(a) Istanbul

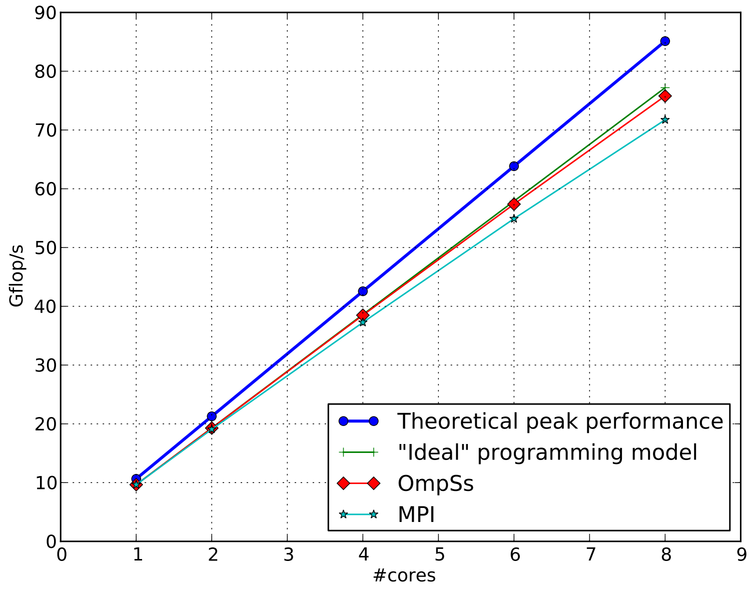


(b) Nehalem

Figure 6.8: Istanbul 12 cores and Nehalem 8 cores. Performance results of the HPL for different problem size NB=242. Theoretical peak performance, "Ideal" programming model, MPI, OmpSs and OpenMP.



(a) Istanbul



(b) Nehalem

Figure 6.9: Istanbul and Nehalem Evaluation. Scalability N=16384 and NB=242. Theoretical peak performance, "Ideal" programming model, MPI, OmpSs and OpenMP.

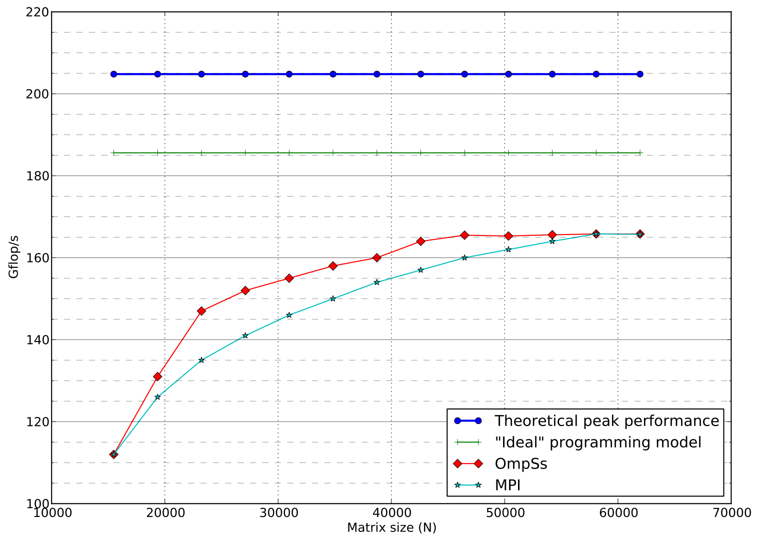


Figure 6.10: Altix 32 cores. Performance results of the HPL for different problem size NB=242. Theoretical peak performance, "Ideal" programming model, MPI and OmpSs.

curve of the ideal programming model. It is interesting to notice the performance drop of the MPI version for 11 cores in the Figure 6.9. 11 as a prime number only allows (1,11) decomposition of MPI and shows the limitation of the static distribution of the MPI programming model.

Figures 6.10 presents results obtained on the NUMA machine (Altix). Considering (P,Q) decomposition, the MPI HPL uses (4,8) for Altix. The OmpSs version is aware of the NUMA architecture and implicitly exchange messages between NUMA nodes. For small problem sizes communication is very costly for both version. When problem size grows the OmpSs version accelerates the critical path and creates parallelization. This allows the runtime to hide implicit data transfer for tasks. For N=45000 the OmpSs reaches asymptotic behavior while the MPI version reaches the same performance for N=62000. For very large problem sizes data transfer is not important and both version perform the same. Figure 6.11 show the scalability up to 64 cores on the Altix machines, and the MPI version again suffers from unsuitable data decomposition (1,61)(2,31).

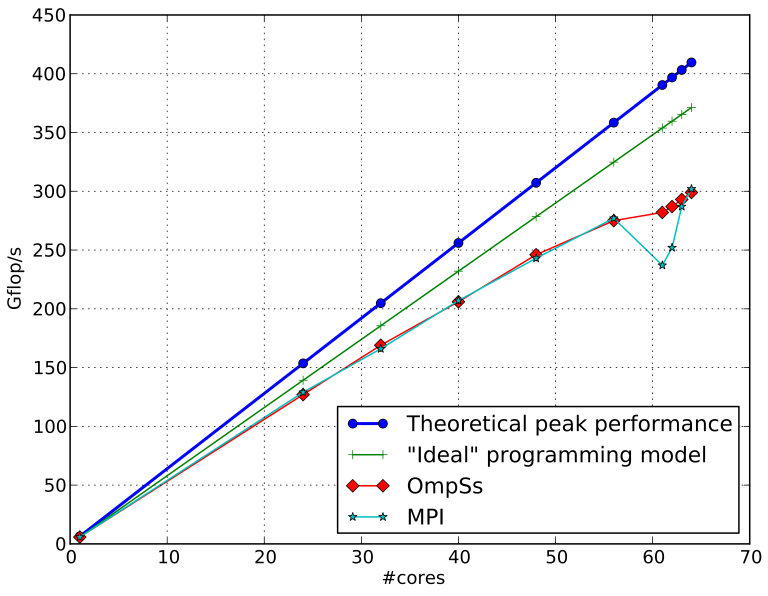


Figure 6.11: Altix. Scalability. Performance results of the HPL for different problem size NB=242. Theoretical peak performance, "Ideal" programming model, MPI and OmpSs.

Summarizing all experiments, the comparison of the MPI and OmpSs HPL shows the limits of MPI HPL benchmark for evaluation of SMP clusters. Evaluation proves that the low efficiency for small problem size of the HPL does not only come from nature of its algorithm, bit the MPI implementation introduces additional limitations. In the case of the MPI HPL, increasing the problem size does not solve issues for the low efficiency, actually the large problem size just hides issues of programming model. Prime numbers of MPI processes limit two-dimensional data decomposition of MPI HPL to keep P and Q values approximately equal, which leads to low scalability. While the OmpSs HPL performs well for numbers of cores that are critical for the MPI version. The OmpSs HPL shows the simple usage of the OmpSs programming paradigm and minimal programming effort to move from sequential to parallel code. We believe that the efficiency, the performance results and the simplicity make the OmpSs HPL a version of the HPL, which will be used for evaluation of a new SMP cluster.

6.2 MPI/OmpSs LINPACK

In this section, we will describe how the HPL code can be restructured to use the proposed hybrid MPI/OmpSs model. First we describe the transformation assuming P=1 (one-dimensional data decomposition) and later describe the differences for a two-dimensional decomposition.

6.2.1 1D Decomposition

Figure 6.6 and Figure 5.5 show the pragma specifications required to convert the pure MPI code in Figure 6.2 to a hybrid MPI/OmpSs. The factorization is performed by a single task whose input is the updated panel of a previous iteration and whose output is the factorized panel for the current iteration. The update of the trailing submatrix is partitioned in a set of tasks, each of them taking as input the factored panel (either produced locally or received) and a subset of the local panels to update. Since the factorization lies on the critical path of the overall algorithm, we use the highpriority qualifier as a hint for the runtime.

Figure 6.12 shows a partial task graph generated during the execution of this hybrid version. In the original HPL with no look-ahead one process executes all tasks in one iteration j before proceeding to the execution of the next iteration j+NB, precluding the overlapping of communication and computation. The original HPL with look-ahead tries to follow the critical path executing tasks that are a certain number of iterations in advance (degree of look-ahead). The control flow in the HPL code achieves this execution. In contrast the hybrid MPI/OmpSs naturally follows the critical path of the execution by executing the task graph in a dataflow way. So for example, process p in Figure 6.12 would execute recv(j), send(j), first instance of update(j), fact(j+NB), send(j+NB), . . . With no look-ahead or dataflow execution, fact(j+NB) would not start until all instances of update(j) were finished, delaying the critical path of the application. This global critical path proceeds along the panel factorization, communication to the next process, update of the first uncompleted panel in this process, factorization of this panel and so on. In order to speedup the computation along this path, the send and receive tasks are labeled as highpriority.

Notice that the renaming mechanism in the OmpSs runtime is dynamically doing

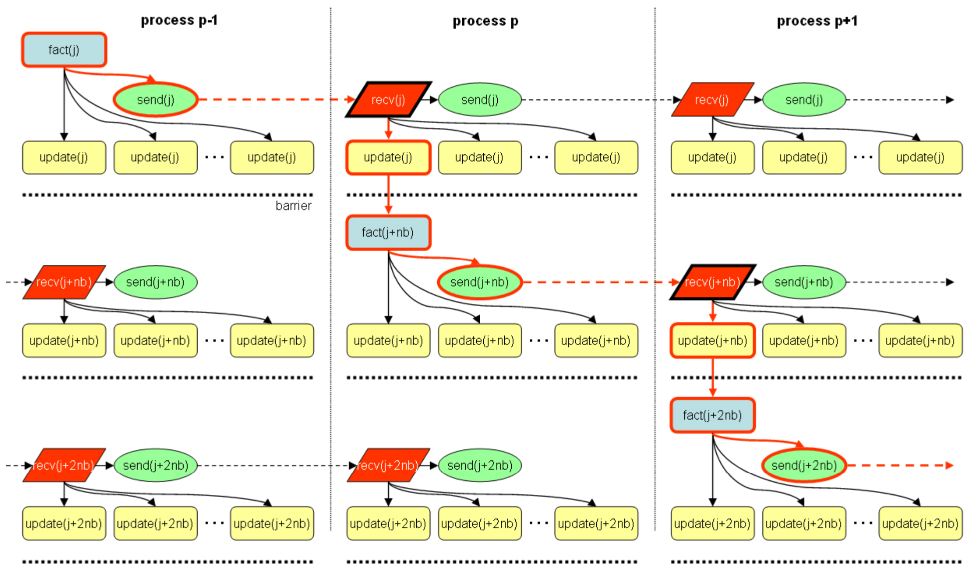


Figure 6.12: Partial dataflow graph in the execution of HPL: MPI process execution in vertical and iteration j of main loop in horizontal. Nodes correspond to the different tasks: fact (panel factorization), send (panel send), recv (panel receive) and update (panel update).

the replication of panels that is necessary to execute the tasks in a dataflow way and whose explicit management added part of the complexity to exploit the look-ahead code which is necessary for other programming models.

6.2.2 2D Decomposition

In order to achieve good load balance and scalability of the algorithm, the HPL distributes data onto two dimensions. As noted in Section 6.1.1, this data distribution adds new communications in the factorization and update phases. In the factorization phase very fine grain communications are needed to exchange rows of size NB doubles for each matrix column when computing the pivot values. At the beginning of the update phase the pivoting has to be applied to the trailing submatrix, requiring the exchange of messages of size NB doubles between the groups of processes among which each panel has been partitioned.

We explored the two possibilities described in Chapter 5 to parallelize with our hybrid approach. The first one consists on taskifying all fine-grain communication operations in panel factorization and pivoting. The overhead introduced to dynamically create and manage these tasks is too large to compensate any benefit. The second alternative consists on defining the pivoting function as a new task. As explained in the Section 5.2.2, we use the polling mode for receiving the first few fine-grain messages while we continue using blocking for the rest. This second alternative results in the best performance results for the hybrid MPI/OmpSs HPL code.

6.2.3 Results

The experimental evaluation in this section is done on a cluster made of IBM JS21 blades (4-way SMP nodes) and Myrinet- 2000 interconnection network . Comparison is done for pure MPI and hybrid MPI/OmpSs versions. We used the space of parameters explored in the Section 6.1.2 and choose the ones that optimized HPL. We performed all the experiments in a normal production environment. As a BLAS kernel we used the Goto library version 1.24. In all experiments we use NB=128, Q larger than P, and look-ahead enabled.

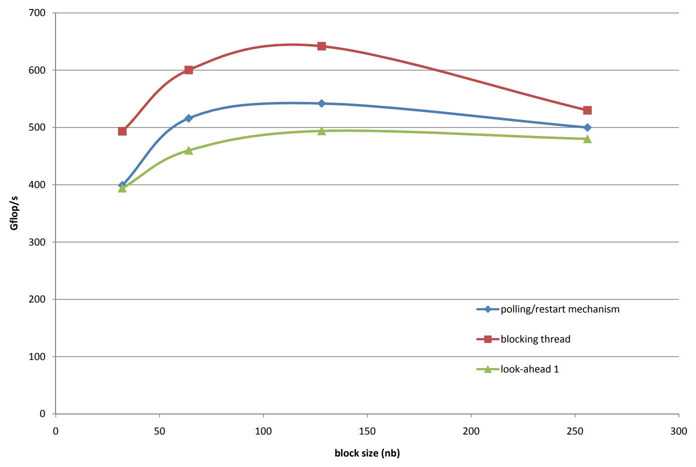
For the HPL, we evaluate the performance using 128, 512 and 1000 cores.First we analyze pure MPI and hybrid MPI/OpenMP versions with look-ahead in the production environment of our machine. We compare the performance achieved by the hybrid MPI/OmpSs with the pure MPI version. We then compare the performance of these two versions of HPL with the hybrid MPI/OmpSs version for different problem sizes and core counts.

Note that using the largest problem sizes that fit into the memory, a machine achieves more than 65% of the theoretical peak (computation O(N3) vs. communication O(N2)). Our work is focused on the smaller problem size, where HPL equally stresses different parts of a machine (CPU, memory and network) as many applications do.

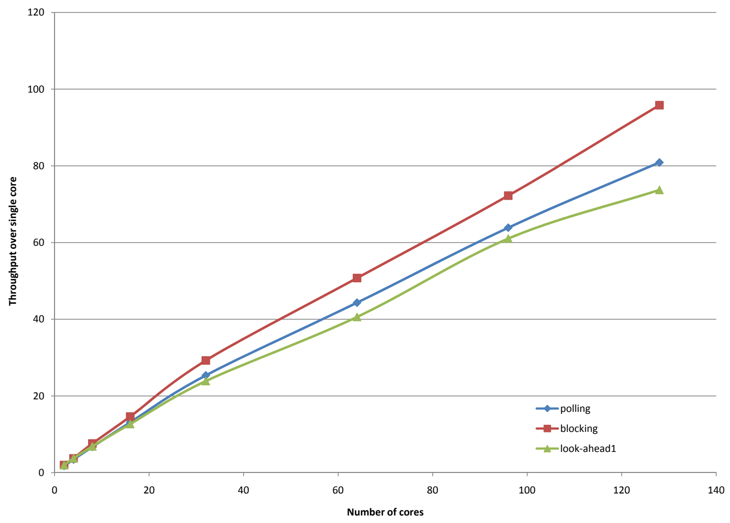
MPI/OpenMP version should have better load balancing and smaller communication overhead. However, OpenMP parallelizes computations between communications but all OpenMP threads synchronize before the next communication is issued. MPI calls are executed by the main thread while other threads are in the idle state, this limits MPI/OpenMP approach and shows lower performance results than pure MPI. The plot in Figure 6.14 for 128 processors shows how the performance of the hybrid MPI/OpenMP version falls behind pure MPI. Tasking in the new OpenMP 3.0 would not make any difference over parallel loops in HPL, because as it is in the current language specification does not allow the anachronism and dataflow execution of OmpSs. From now on we only focus on the basic comparison between the pure MPI and hybrid MPI/OmpSs versions for HPL.

We also compare different overlapping techniques proposed in the Chapter 5: the restart mechanism and the communication thread. The Figures 6.13 show the performance results of the HPL running on 128 processors by using MPI/OmpSs version with restart mechanism, communication thread and the original HPL with look-ahead enabled. All three techniques successfully hide the communication overhead but efficiency of the overlap is not the same. Conceptually, the look-ahead technique and the restart mechanism overlap communication and computation by inserting MPI_Test calls within the update phase. Thus they decrease granularity of the BLAS calls which leads to lower IPC of the DGEMM call. The communication thread does not need to intercept the DGEMM in the update phase, so the IPC stay high and makes the communication thread technique the most efficient. The restart mechanism wins against the look-ahead due to lower number of MPI processes (4,8) compared with (8,16) and the MPI/OmpSs version with restart mechanism uses 4 threads per process that supports the progress of MPI operations. All OmpSs threads call MPI_Test and MPI process can progress quickly when a message arrives. For further experiments, the work considers only the MPI/OmpSs version with communication thread because it delivers the best performance of the proposed techniques on the JS21 platform.

The plots in Figure 6.14 show the performance rate (Gflop/s) for the original MPI version with look-ahead and for the hybrid MPI/OmpSs. The pure MPI version uses a single CPU per MPI process. The hybrid MPI/OmpSs uses one node with 4 CPUs per MPI process, running 4 computation threads and 1 communication thread per node. The pure MPI version uses (8,16),(16,32) and (20,50) decomposition for 128, 512 and 1000 processors, respectively. The hybrid MPI/OmpSs uses (4,8), (8,16) and (10,25) for 128, 512 and 1000 processors, respectively.

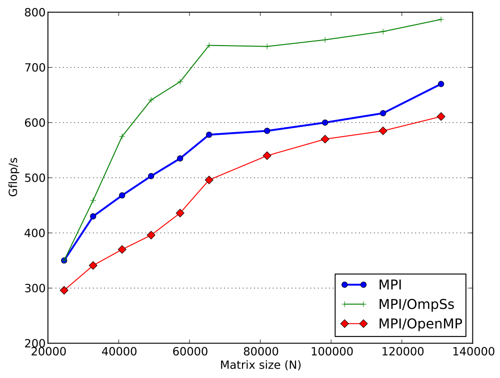


(a) Basic Comparison

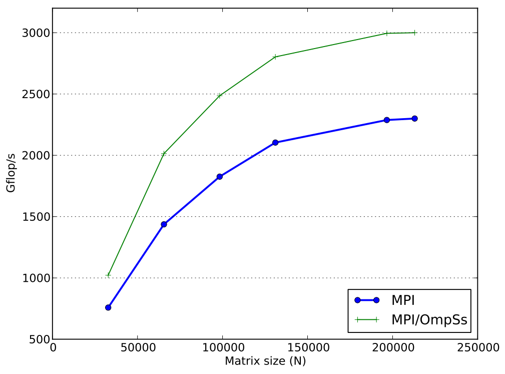


(b) Scalability

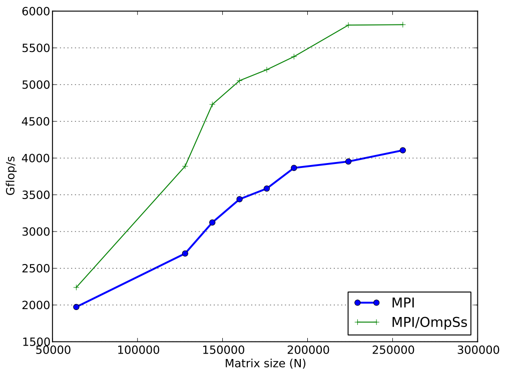
Figure 6.13: Performance comparison of overlapping technique: the restart mechanism, the communication thread and the look-ahead technique.



(a) 128



(b) 512



(c) 1000

Figure 6.14: HPL: performance in Gflops for original MPI version with look-ahead and for the hybrid MPI/OmpSs. Results are presented for 128, 512 and 1000 processors

Besides a more aggressive overlap of communication and computation, the hybrid MPI/OmpSs uses less MPI processes, which reduces the number of messages and gives better load balancing. A further potential performance improvement in the MPI/OmpSs version comes from the possibility of merging the updates of several panels into a single task. This would result in higher IPC for the DGEMM routines, while the communication/computation overlap is still in progress using the communication thread.

We differentiate three regions in the performance plots in Figure 6.14:

- For very small matrices, the computation part of the application is small and there is not much possibilities to overlap communication and computation, which makes the network parameters (bandwidth and latency) the dominant factors. For example, for 512 processors the hybrid MPI/OmpSs gets 5% better performance than the original HPL version. The efficiency of the HPL is 17% while the efficiency of the MPI/OmpSs version is 21,6%.

- By increasing the problem size the hybrid MPI/OmpSs version exhibits its full strength against the original MPI version with look-ahead. For example, for 512 the hybrid version increases performance by 40% when N=131072. The efficiency of the HPL is 43% while the efficiency of the MPI/OmpSs version is 61%.

- For the largest problem sizes we tried, the communication overhead is less dominant and as a consequence the gain of the hybrid MPI/OmpSs version goes down to 30% for 512 processors at N=212992. The efficiency of the HPL is 48,8% while the efficiency of the MPI/OmpSs version is 63,7%.

It is important to note that the execution of the hybrid MPI/OmpSs achieves the same performance as the pure MPI version with much smaller problem sizes. Thus our approach can significantly reduce the time and energy required to report a given HPL performance number.

A different way of presenting the benefits of the hybrid version is to present the Gflops per core as a function of the memory size per core, as shown in Figure 6.15. Notice that the hybrid MPI/OmpSs gets a much better performance at small memory

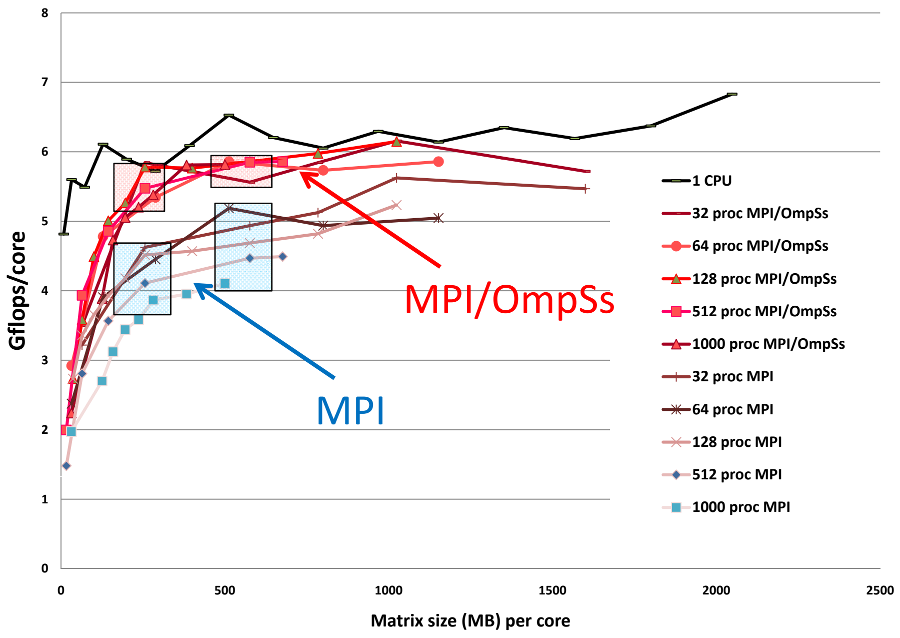


Figure 6.15: Gflops/core.

footprints and that the performance per core does not decrease when the number of cores increases.

6.3 Matrix Multiplication

```
for (i=0; i < N; i++)
  for (j=0; i < N; j++)
    for (k=0; i < N; k++)
      c[i][j] += a[k][j]+b[i,k];
```

Figure 6.16: Serial implementation of the matrix multiplation.

The matrix multiplication is de facto one of the most explored kernels in the HPC area. A basic serial implementation of matrix multiplication contains triply nested loop. The matrix multiplication operation is C(N,M) = A(K,M) x B(N,K), where the matrix multiplication of matrices A and B updates the matrix C. The Figure 6.16 shows a pseudocode that performs the matrix multinational on a single thread.

The Figure 6.17 shows a distributed matrix multiplication kernel (C=AxB) used in this thesis. In this kernel, matrices C and B are distributed by columns and matrix A by rows. Therefore, each process needs to exchange with his neighbors the set of rows of A in order to complete the computation of the set of columns of C assigned to it. For this purpose a temporary buffer rbuff is used together with the exchange communication SendRecv.

Programmers could use non-blocking communication primitives and more complex buffering schemes to obtain a more effective computation/communication overlap. Although this is a simple academic exercise, it would exemplify the reduced programming productivity that incurs. Figure 6.18 shows the result of the taskification process using the proposed OmpSs extensions. Figure 6.19 shows the task graph that would be generated assuming the data distribution in Figure 6.20.

Figure 6.21 shows the speedup with respect to execution with 1 MPI process of two different versions of the matrix multiplication kernel: pure MPI and hybrid

```
double A[BS][N], B[N][BS], C[N][BS];

void mxm (double A[BS][N], double B[N][BS], double C[BS][BS]);

void SendRecv (double src[BS][N], double dest[BS][N]);
...
indx=me*BS;
...
for (i=0; i<P; i++)
{
...
  if(i%2==0) {
    mxm (A, B, &C[indx][0]);
    SendRecv (A, rbuff);
  } else {
    mxm (rbuf, B, &C[indx][0]);
    SendRecv (rbuf, A);
  }
  indx=(indx+BS)%N;
...
}
```

Figure 6.17: MPI implementation of the matrix multiplation.

```
#pragma css task input(A, B) inout(C)
void mxm (double A[BS][N], double B[N][BS], double C[BS][BS]);

#pragma css task input(src) output(dest) target(comm_thread)
void SendRecv (double src[BS][N], double dest[BS][N]);
```

Figure 6.18: Data distribution for matrix multiplation acrooss 4 MPI process. Each process contains a part of matrix A and matrix B.

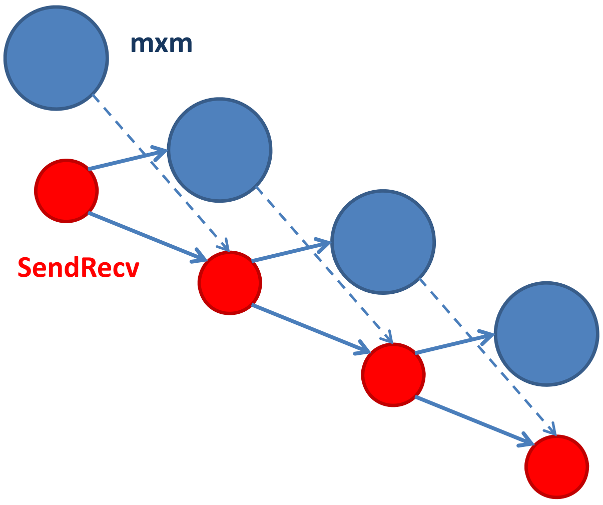


Figure 6.19: Task Dependence Graph of matrix multiplation. White nodes correspond to mxm instances, blue nodes correspond to SendRecv instances, solid edges correspond to true data dependences and dashed edges correspond to antidependences (due to the reuse of data storage for communication). The data renaming done at runtime eliminates these antidependences and the execution of successive SendRecv invocations without waiting for the termination of previous mxm invocations.

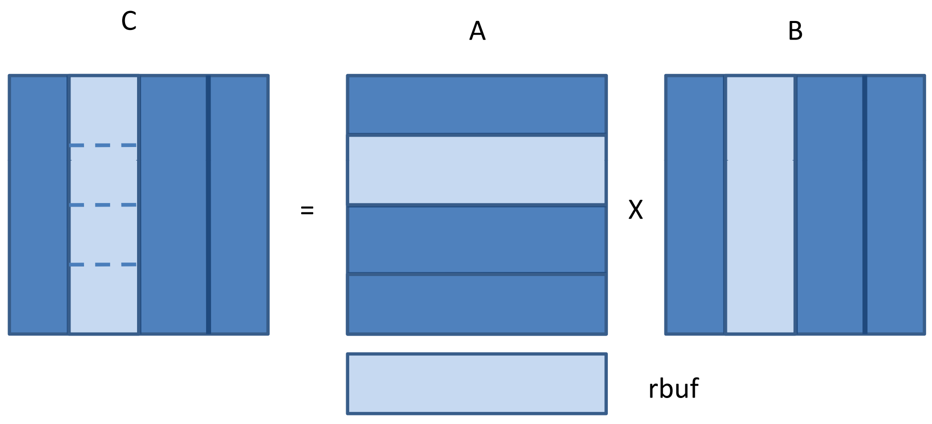


Figure 6.20: data distribution.

MPI/OmpSs, using matrices of size 8192x8192. In order to isolate the benefits of node sharing, for the pure MPI version we also show results when running 1, 2 or 4 MPI processes in the same node. The better overlap in the hybrid MPI/OmpSs approach results improvements in the 14-17% range. Larger improvements from the hybrid code over the pure MPI are observed for smaller matrix sizes (4096x4096), the reason being that the smaller the problem size the more important the communication is compared to the computation.

6.4 CG

In this section, we describe the conjugate gradient solver used in [25] as a case study to show the applicability and usage of nonblocking collective operations to overlap computation and communication. The code uses domain decomposition to distribute a 3D space among the processes and makes use of collectives to communicate border elements with neighbors instead of point-to-point calls.

Figure 6.22 shows the kernel of the main loop using blocking and non-blocking collectives. The kernel contains three steps: send boundaries, compute a volume, and compute boundaries. Sending boundaries is implemented using an all-to-all operation,

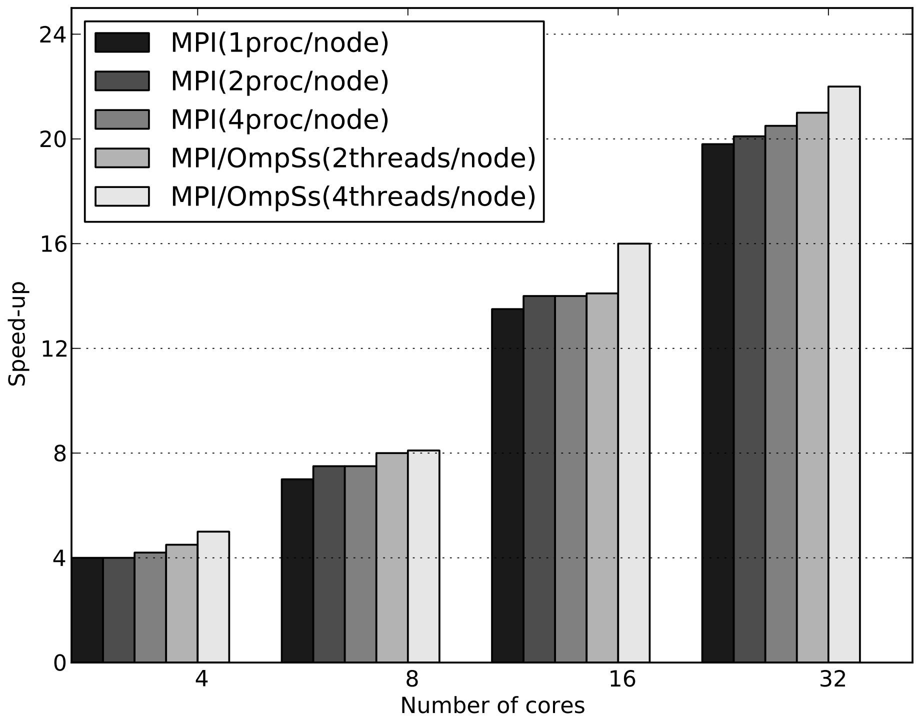


Figure 6.21: Matrix multiply: performance comparison of pure MPI (running different number of MPI processes per node) and hybrid MPIOmpSs (with different number of OmpSs threads per node). Results are for a matrix size of 8192x8192.

where each process communicates with its neighbors. The result of this communication is used to compute the boundaries. The collective operation can either be blocking (MPI_Alltoall) or nonblocking (MPI_Ialltoall). The non-blocking collective reduces the communication overhead provided by MPI_Alltoall, but also increases internal complexity of the mult_volume, where the programmer needs to insert MPI_Test calls in order to progress non-blocking operation. Figure 6.23 shows the MPI/OmpSs version of the code, where the programmer just needs to define tasks and the direction of the arguments.

```
for (...){
  fill_buffers(q,send_buffer);
  if(non_blocking_collectives)
    MPI_Ialltoall(send_buffer,recv_buffer,&req);
  else
    MPI_Alltoall(send_buffer,recv_buffer);
  mult_volume(v,q);
  if(non_blocking_collectives) MPI_Wait(&req);
  mult_boundaries(v,q,recv_buffer);
}
```

Figure 6.22: Main loop in conjugate gradient using blocking and non-blocking collectives.

```
#pragma css task input(q[q_size]) output(send_buffer[buf_size])
void fill_buffers(double *q, double *send_buffer);

#pragma css input(send_buffer[buf_size]) output(recv_buffer[buf_size]) target(comm_thread)
void all_to_all (double *send_buffer, double *recv_buffer);

#pragma css task input(q[q_size]) output(v[q_size])
void mult_volume(double *v, double *q);
#pragma css task input(q[q_size],recv_buffer[buf_size])) output(v[q_size])
void mult_boundaries(double *v, double *q, double *recv_buffer);


for (...){
  fill_buffers(q,send_buffer);
  all_to_all(send_buffer,recv_buffer);
  mult_volume(v,q);
  mult_boundaries(v,q,recv_buffer);
}
```

Figure 6.23: Hybrid MPI/OmpSs version of the CG.

The hybrid MPI/OmpSs version performs up to 15% better than the original MPI version that is not optimized for communication/computation overlap. Figure 6.24 shows results for up to 512 cores, showing how the performance improvement constantly grows with the number of cores. The performance results obtained by the hybrid MPI/OmpSs version are consistent with those shown in [25] using non-blocking collectives.

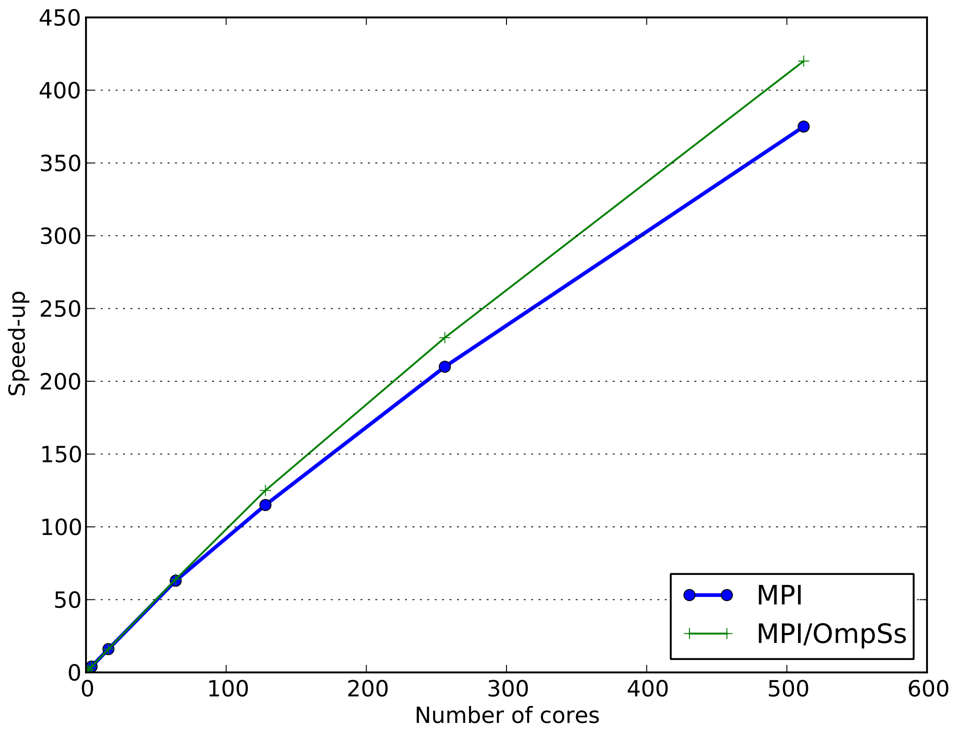


Figure 6.24: Conjugate gradient: performance comparison of pure MPI and hybrid MPI/OmpSs.

7 Evaluation of Tolerance to Low Bandwidth and OS noise by Using the MPI/OmpSs approach

Recent research [27] has shown that operating system (OS) noise and a demand for high bandwidth network [33] limit the application performance in HPC system. While increasing the bandwidth of high capacity interconnection network tries to minimize a communication overhead. The OS interference propagates across the distributed memory nodes due to synchronization between application instances.

The programming model is not specially designed to mitigate these issue, we rather want to analyze how it affects them. The programming model could offer cheap and elegant solution. Compering MPI and MPI/OmpSs approaches, we analyze tolerance to mentioned limiters and show importance of programming model in order to address them.

The rest of the Chapter is organized as follows. In Section 7.1, presents the common platform, software and hardware stack, used for evaluation of the tolerance to OS noise and low bandwidth network. Section 7.2 introduces importance and types of OS noise, describes an implementation of an artificial OS noise used in experiments. Section 7.2 shows how the MPI/OmpSs approach reduces the OS noise and suppresses spreading of the OS noise across distributed nodes. The Section 7.3 analyzes the cost of supercomputer interconnection network, emphasizing a demand for high bandwidth and its cost. The Section 7.3 explains a method for mimicking the low bandwidth network on the real machine. Finally the chapter demonstrates that MPI/OmpSs tolerates the system with low bandwidth interconnects better than pure MPI approach.

7.1 Platform

In this section, we describe the platform used for experiments od the sections 7.2 and 7.3.

32 and 128 nodes with two PowerPC 970MP dual-core processors at 2.3 GHz per node and 8 GB of shared memory per node. Each chip has 2 cores with 1 MB of shared cache memory per chip. All of the core can execute 4 floating point operations per cycle and use uniform memory access (UMA). Myrinet interconnection network links nodes, where bandwidth reaches 256MB/s.

We performed all the experiments in a normal production environment. As a BLAS kernel we used the Goto library version 1.24. The experiments use a standard MPICH implementation of the Myrinet. In order to compare MPI and MPI/OmpSs tolerance to the OS noise and low bandwidth, the thesis uses already described HPL and BT from the Chapter 6

7.2 Tolerance to OS noise

OS noise is any asynchronous interruption of a running program by the operating system. Operating system noise in general and process preemption on massively processing systems in particular have been identified and studied as one of the important potential causes of significant performance degradation. Interference of operation system services and associated daemons slow down an application instance thus local perturbations easily propagate and accumulate through the whole program dependence chains and specially at global synchronization points. Writing to the file-system and a slowdown of a core due to overheating provide an unpredictable behavior of a run and could be simulate as a standard OS noise as well.

Attempts to limit OS noise are focused on operation system scheduling level like timer interrupts and system daemons or by dedicating cores to OS tasks, special lightweight kernel (Blue Gene) where OS cannot do a preemption of the running task and only the running task could explicitly preform the context switch by calling a system call. While a work that addresses a OS noise issue keep focus to understanding the trade-offs and the ratio between OS services for large scale system.

We used the production environment settings on supercomputers for all our experiments and building a kernel-level noise injection into operation system requires special permission and superuser access to the machine. We decided to implement the noise injection on application level.

In order to evaluate the impact of noise on HPC applications, we have modified the source code of the application by generating an additional thread per process that runs periodically during application execution. With parameters for noise generation we control the average duration of each individual noise event and the frequency of the noise. Sleeping and computing phase of the noise are random non-deterministic number. By controlling the average duration of both phases it is possible to simulate different levels of OS noise. This experiment corresponds to a coarse granularity of preemptions where the computation phases take up to the 500ms and the sleeping phases are in the order of seconds.

7.2.1 Results

For experiments, we use the HPL benchmark, explained in the Section 6.1. The HPL algorithm solves different problem size in each iteration. Thus a synchronization point, where a OS noise could potentially harm execution, does not appear at regular intervals of time. At the very beginning of execution, the synchronization point is rare, while after each iteration it becomes more and more frequent. This behavior, various synchronization points, covers all cases where OS noise could reduce performance and makes a good case study for evaluation of programming models regarding OS noise.

We obtain results for different duration of the sleeping phase on 128 and 512 processors. Runs use an average problem size N=65536 and N=131072 for 128 where NB=256. Input parameters aim an efficient execution of HPL in terms of Gflop/s, on the other hand communication cost stay relevant.

Figure 7.1 shows the sensitivity of the two versions of HPL to process preemptions. This experiment corresponds to a coarse granularity of preemptions where the sleeping phases are in the order of seconds, as shown in the horizontal axis. As can be seen in the figure, the hybrid MPI/OmpSs version tolerates preemptions much better. For 128 processors and the period of preemption bursts of 3 seconds, performance of our version does not suffer, while execution time of the HPL is increased by 7 At very high preemption frequencies, both versions suffer the impact of the perturbations. When the duration of the sleeping phase is smaller than 3, the MPI/OmpSs version can not hide the OS noise because the noise becomes very expensive and the OmpSs scheduler is not able to place the noise burst without performance degradation. Figure 7.2 shows the same phenomena for the period of preemtion busts of 30 seconds. Experiments for 512 processors have approximately two times longer execution due to 2 times bigger problem size and 4 time bigger number of processors. For the duration of sleeping phase of 30 seconds, the MPI versions suffers 3% of the performance drop while MPI/OmpSs keeps the same performance. Both figures, 7.1 and 7.2, show a small ripple for the MPI/OmpSs version with the period higher than 3 and 30 seconds respectively, because the OS noise can appear at the end of execution, where synchronization point appear frequently, and the OmpSs scheduler cannot reshuffle bursts.

7.2.2 Conclusion

The high levels of asynchronous introduced by the hybrid MPI/OmpSs model make the applications more tolerant to such perturbations. Out of order execution of communication tasks by using MPI/OmpSs programming model dynamically tolerates CPU frequency variation at the runtime. Slower task of one process cannot disturb other MPI processes and slowdown them. The OmpSs runtime scheduler cares about task selection, accelerates critical path and muffles unwanted OS noise effects. The goal of our programming model is the facing the OS noise issue without introducing any extra work to applications developers.

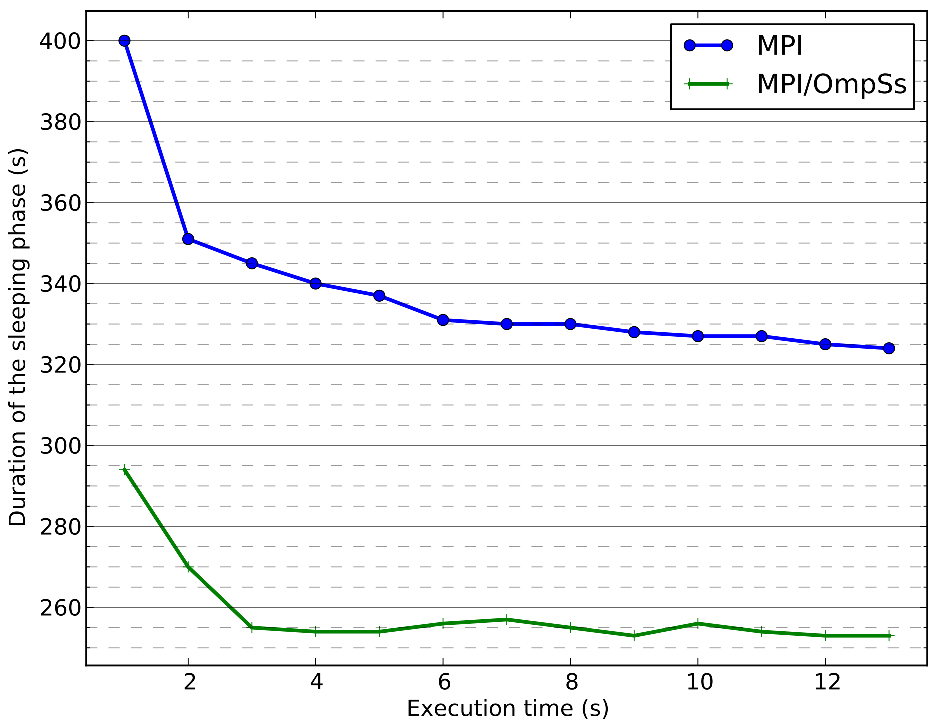


Figure 7.1: HPL: sensitivity to process preemptions. Results are for 128 processors. Problem size N=65536, NB=256. Decomposition 8x16.

We introduce OS noise at user level,using the common abstraction of OS noise with frequency and period. Data flow execution of tasks reschedules a OS noise burst and processes the communication from on the critical path. This scenario makes the MPI/OmpSs more tolerant to OS noise than a pure MPI. Porting an application from MPI to MPI/OmpSs initially may not bring the performance improvement but a real execution suffers from non-deterministic behavior due to OS noise. A programming model that handles unpredictable runs saves a cost of long execution and a rerun of execution.

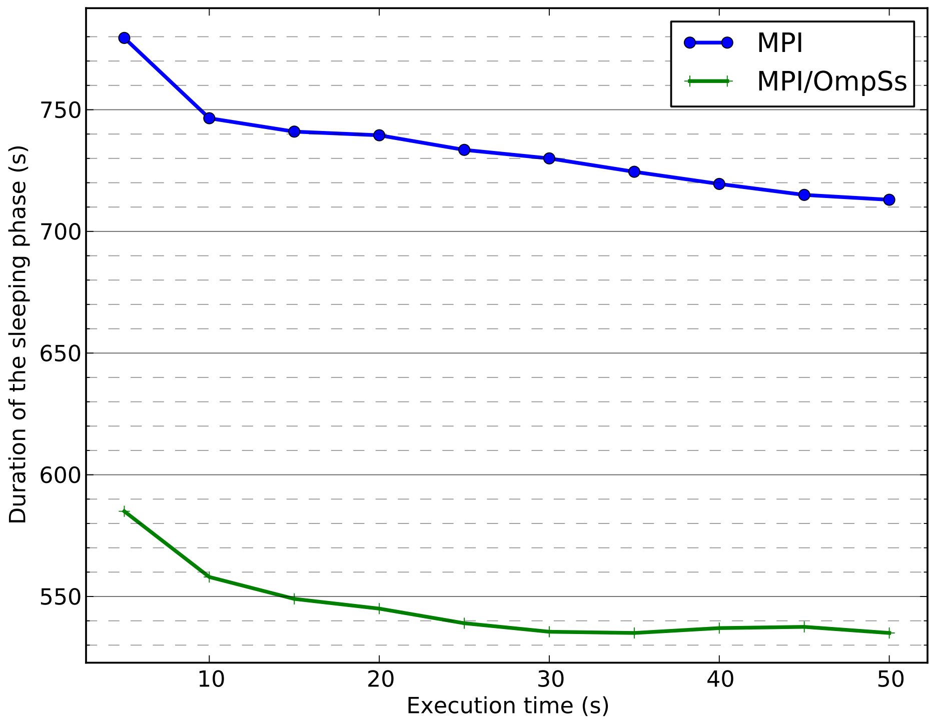


Figure 7.2: HPL: sensitivity to process preemptions. Results are for 512 processors. Problem size N=131072, NB=256. Decomposition 16x32.

7.3 Tolerance to low bandwidth

In order to explore the impact of lower bandwidth we used a dilation technique by modifying the source code such that for each message of size S an additional message of size f*S is transferred between two dummy buffers at sender and receiver. For example, a value of f=1 would mimic the availability of half the original bandwidth.

7.3.1 Results

The section introduces two applications that have been ported to MPI/OmpSs programming model: Jacobi Solver and BT NAS. The communication operations are not costly for these two applications but their algorithms show potential to overlap communication and computation. We found these applications a good case study for tolerance to low bandwidth experiments. The evaluation also uses the HPL with the settings used in the section 7.2.

We describe a Jacobi solution used for solving Poisson equations in a rectangular domain. The original MPI code partitions the data domain among the MPI processes along the xaxis. Each MPI process has a slice of size nx by ny grid points to work on stored in two matrices a[nx+2][ny] and b[nx+2][ny] (which include the boundaries to perform communication).

Figure 7.3 shows the body of the iterative loop in the program performing the usual Jacobi computation: b[i,j] = f(a[i-1,j], a[i,j], a[i,j+1], a[i+1,j]) Each iteration contains two steps: 1) computation of matrix a from matrix b and 2) exchange of the boundaries. Processes exchange the boundaries with their neighbors, which are necessary for computing the first and the last row of the matrix, while the other rows do not have dependencies with this communication operation. The programmer could use nonblocking MPI_Isend and MPI_Irecv calls to achieve an overlap between the exchange phase and computation phase (with the appropriate communication probe calls), at the expenses of increased code complexity. Figure 7.3 also shows the OmpSs annotations that allow the OmpSs runtime system to effectively overlap the two phases in the loop body using blocking MPI calls and without changing the structure of the code.

The core of the NAS BT benchmark [2] is a block tridiagonal solver representing computational patterns found in fluid dynamics codes at NASA. The problem domain is a cube of grid points on which four major steps per iteration are done: the computation of a right hand side matrix (compute_rhs) followed by three sweeps in the x, y and z directions. Each of these sweeps consists of two successive dependence carrying passes: a solve pass in the forward direction and a back-substitute pass in the backward direction.

The MPI implementation requires the number of processors to be a perfect squared number. The cube of grid points is partitioned in P blocks (called cells) in each direction, distributing P cells to each process. The distribution is such that every process has one cell in each plane in each of the three directions. Also for all cells in a given processor, all neighbor cells in a given direction are in the same target processor. The property of such distribution is that on every sweep direction there is always one cell on each processor that can be computed and that the communication pattern is regular.

```
#pragma css task input(send_up, send_down) output(recv_up, recv_down) target(comm_thread)
void exch_boundaries(double send_up[ny],
  double *recv_up[ny],
  double *send_down[ny],
  double *recv_down[ny])
{
  MPI_Sendrecv(send_up, recv_up);
  MPI_Sendrecv(send_down ,recv_up);
}

#pragma css task input(a) output(b)
void compute_row(double a[3][ny], double b[ny]);

#pragma css task input(a) output(b)
void compute_block(double a[nx][ny], double b[nx-2][ny]);

for (...) {
  compute_row(&a[0][0],&b[1][0]);
  compute_row(&a[nx-1][0],&b[nx][0]);
  compute_block(&a[1][0],&b[2][0]);
  exch_boundaries(&b[1][0],&b[0][0],
  &b[nx][0],&b[nx+1][0]);
  tmp=a; a=b; b=tmp;
}
```

Figure 7.3: Loop body for the iterative jacobi in the Poisson equation solver. Annotated functions using OmpSs.

A copy_faces step where boundary data is exchanged for all neighboring cells is also required before the compute_rhs routine updates all the cells within each processor. Although the source code is implemented with non-blocking MPI calls, the wait calls are invoked immediately after issuing the requests for communication. The behavior is thus as if blocking calls had been used and there is no overlap between communication and computation.

With such data distribution, potential sources of overlap of computation and communication appear in two points: first it is possible to perform the compute_rhs and x_solve on the cell free of dependences in the x direction one after the other such that the communication in that direction can be overlapped with the compute_rhs on the other cells in the process; second when performing the first backsubstitute in a given direction, the solve in the next direction can be immediately executed so that a pipeline in the new direction can be started. The actual amount of overlap depends on the ratio of duration of the different computations involved (compute_rhs, solve and backsubstitute) and on the propagation of the complicated dependence chains through the whole computation space.

The main issue in this example is to exploit such potential overlap without drastically restructuring the code and in a way that dynamically adapts to potentially different ratios of computation duration. By taskifying the computations (compute_rhs, solve and backsubstitute) as well as the communication (pack, communicate, unpack) the code keeps the same structure as the original one but the run time has the potential to reschedule computation and communication. Through high priority hints in the declaration of the task (i.e. solve and backsubstitute) can help the runtime to dynamically exploit the above-described potential.

Figure 7.4 shows the execution time of NAS BT, class B on 64 processors, for different values of effective network bandwidth. The rightmost point in the plot corresponds to the actual bandwidth of the target platform; points to the left correspond to runs where the actual bandwidth has been multiplied by the factor indicated in the horizontal axis.

A full overlap of communication and computation within the NAS BT benchmark is not possible due to nature of algorithm and data dependency between computation memory area and communication buffer. Thus, low bandwidth affects both implementation, MPI and MPI/OmpSs version. While MPI/OmpSs approach partially overlaps communications, improves the performance of MPI version shows two interesting regions: for the original bandwidth of machine, the MPI implementation. The Figure 7.4 the bandwidth decreases from right to left and , the MPI/OmpSs tolerates lower bandwidth better than MPI implementation up to the 10 times smaller bandwidth than the initial bandwidth. At very low bandwidth, more than 10 times smaller than initial one, computations are not long enough for overlapping expensive communication burst and both implementations suffer from the low bandwidth.

Figure 7.5 show the execution time of Jacobi on 32 processors for different values of effective network bandwidth. The rightmost point in the plot corresponds to the actual bandwidth of the target platform; points to the left correspond to runs where the actual bandwidth has been multiplied by the factor indicated in the horizontal axis. Notice that the execution time for the hybrid MPI/OmpSs version has more flat increase that the pure MPI version. The tolerance to bandwidth shown by Jacobi is better (flat plot) due to the higher computation/communication overlap achieved.

Figures 7.6 shows the execution time of a HPL run for different effective bandwidth of the network on 128 processors. The plot shows that even if starting at a smaller execution time, the hybrid MPI/OmpSs version is not affected by a reduction to 60%

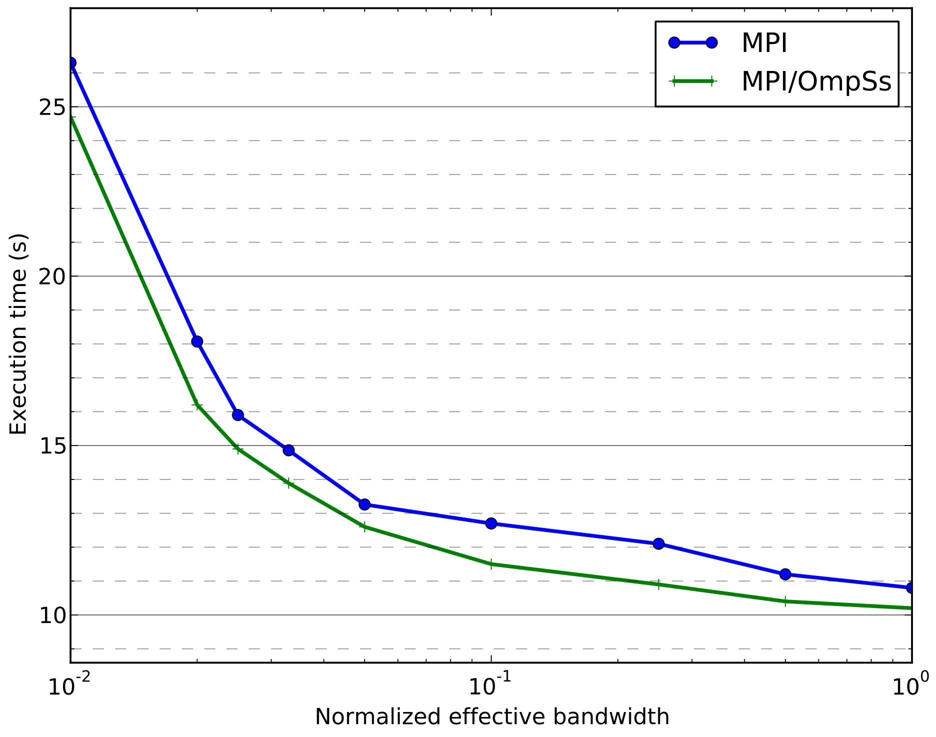


Figure 7.4: NAS BT: sensitivity to low network bandwidth. Results are for class B and 64 processors.

of the actual bandwidth. The HPL version is much more sensitive to such reduction, resulting in an increase of 23,4% in the execution time. Results for 512 processors, see Figure 7.7 show that the hybrid MPI/OmpSs version is slightly affected (increases by 22,8%) by having five times less bandwidth. In the case of HPL,the execution time increases by 91,8% for the same reduction of bandwidth.

7.3.2 Conclusion

The interconnection network is the most expensive part of the modern supercomputers. Applications requires high bandwidth and thus the cost of network rises. The MPI/OmpSs programming model overlaps communication and computation and executes tasks in out-of-order manner, which makes runs more tolerant to low bandwidth networks than traditional MPI approach. Results show that a smart data flow oriented

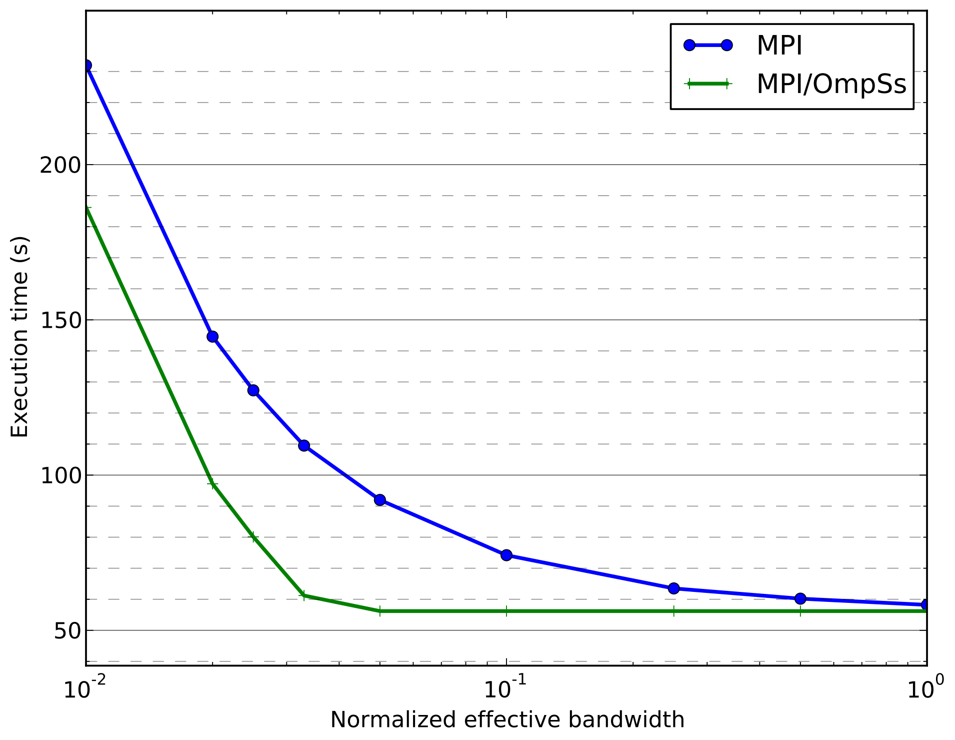


Figure 7.5: Jacobi Solver: sensitivity to low network bandwidth. Results are for 64 processors.

runtime and low bandwidth interconnection network offer the same results as costly high bandwidth interconnects and pure message passing approach. Nowadays, a non-optimized pure MPI codes drive a demand for high bandwidth interconetion networks, which is wrong. The software runtime approach should prefetch and overlap all visible communication and then profiling of these executions define high bandwidth requirements and cost of interconnects.

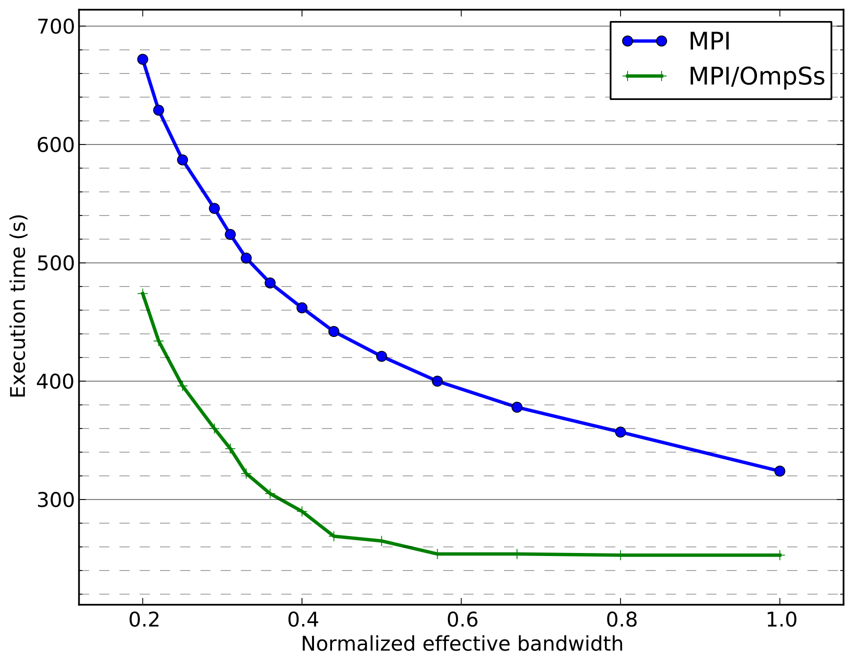


Figure 7.6: HPL: sensitivity to low network bandwidth. Results are for: 128 processors. Problem size N=65536, NB=256. Decomposition 8x16.

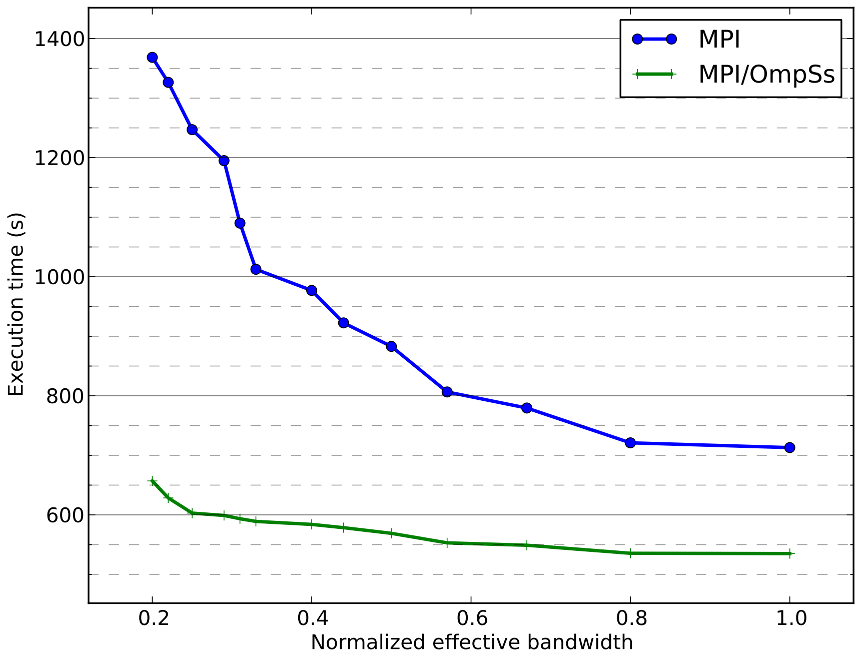


Figure 7.7: HPL: sensitivity to low network bandwidth. Results are for: 512 processors. Problem size N=131072, NB=256. Decomposition 16x32.

8 Related Work

The communication overhead of the MPI has been a hot topic since 1995 [45]. First MPI applications reduce the communication overhead by using asynchronous point-to-point routines. There are several works [3] [37] [31] where an implementation of optimized collective operations uses point-to-point calls.

With SMP nodes, a hybrid MPI and shared memory programming model has become popular [47]. While thread base model as OpenMP have not brought new approaches due to explicit fork/join model [40]. Communication thread based approaches strangle with time sharing on OS scheduling level. Optimization of the communication thread becomes research target, several approaches address this issue and try to improve communication progress [7] [41].

HT cores seems to be a perfect solution for helper thread approach, but HT has not found its place in the HPC system due to unpredictable behavior. On the most HPC environment, this feature is disabled. Lack of non blocking collective force hand-tuned implementation of collective communication. Broadcasting especially has been interested as common collective operation in liner algebra algorithms with potential for overlap. There are plenty broadcast solutions [16] [18]. Finally MPI-3 standard introduced non-blocking collectives, where easy-to-use concept has been presented. Non-blocking collectives already shows application usage[25].

While all works have focused on communication part, there are not any work that classify and analyse overlapped computation kernel and its impact on overall performance, while the Chapter 4 covers this impact. Computation part has been study for GPU executions [54] , but an analysis of interleaving computational kernels with probing messages and interrupt strategy depend on kernel type has not been a topic.

Clusters comprised of a distributed collection of SMP nodes are becoming common for parallel computing. The hybrid use of MPI with shared-memory paradigms, such as OpenMP, has been subject of research and performance evaluation [39] [38] [30]. The explicit fork/join paradigm in these shared-memory programming models and the restrictive barrier synchronization precludes more advance or aggressive overlapping of communication and computation (i.e. across iterations of an outer sequential time step loop).

In order to address the programmer productivity wall in distributed memory architectures, some languages that are based on the partitioned global address-space abstraction (PGAS), such as UPC or CAF, rely on the compiler to perform the appropriate optimizations to overlap communication and computation. The use of pure shared-memory approaches to program these architectures, relying on the compiler to translate from OpenMP to MPI [5] or on the use of a distributed-shared memory (DSM) layer also need to worry about this optimizations at the appropriate level (language extensions to express data distributions and communication [13], compiler optimization [34] or runtime library [36] [35]).

Recognizing the popularity and influence in the research area of the HPL as a benchmark, a lot of previous research has focused on improving its behavior. For example, using hybrid MPI/OpenMP for SMP clusters [51], using optimized BLAS routines [28], or using an asynchronous MPI programming model [50] to explicitly code the overlap of communication and computation. In order to address the programmer productivity issue, some implementations of the HPL benchmark using PGAS languages have appeared [42] [52], focusing on programming productivity and not in achieving big performance improvements.

The hybrid MPI/OmpSs approach presented in this thesis exploits the use of asynchronous MPI calls without increasing complexity of code, which leads to better performance. Overlapping computation and communication is automatically done by the runtime system by appropriately schedule communication and computation tasks in a dataflow way.

9 Conclusion

This thesis studies the programmability aspects of HPC systems, where performance and productivity of the programming model are the main goals, and balances these goals by proposing a novel programming environment. In the HPC area, widely used parallel programming paradigm are the MPI model across distributed nodes and the OpenMP model within a node. Most scientific and industrial applications have been written by using MPI or hybrid MPI/OpenMP. In order to improve performance, programmers need to reduce communication overhead which requires code restructuring and overlapping communication with computation. Thus codes become complex and programming productivity decreases. The goal in this thesis was to introduce programming model that shows the best performance without increasing the complexity of the code.

In the first part of this thesis, Chapter 4, we analysed already known overlapping techniques on kernels that are compute, memory or computememory bound. The thesis demonstrated that the nature of computational kernels is important when choosing overlapping techniques. Full overlap of communication could still lead to performance degradation because the overlapping techniques decrease efficiency of computation and overall performance drops down. The most sensitive kernels are those that are highly optimized and compute/memory bound in the same time as for instance the DGEMM implementation of the BLAS library. The experiments showed that the specialized light-wait thread for MPI communication of OS system minimizes performance degradation of computation.

Furthermore, the thesis presented the hybrid use of MPI and OmpSs (OMP superscalar, a task-based shared-memory programming model), allowing the programmer to easily introduce the asynchrony necessary to overlap communication and computation. The Chapter 5 described implementation issues in the OmpSs runtime that support its efficient inter-operation with MPI. It presented a design of the OmpSs programming environment by defining programming interface and introducing support for overlapping techniques on different platforms.

In order to evaluate the MPI/OmpSs programming model, we ported well known kernels and compared performance within node as well as across large number of nodes. The experimental evaluation on a real supercomputer reveals promising performance improvements. For example, the hybrid approach improves HPL performance up to 40% when compared to the original pure MPI version with look-ahead turned on for the same input data. An important advantage of our approach is, that we can achieve the same performance of the regular HPL benchmark with smaller problem sizes, thus requiring much shorter execution times. Also, the resulting program is less sensitive to network bandwidth and to operating system noise, such as process preemptions.

The proposed programming model has been used in many research projects by users and showed potential to replace already accepted programming approaches. The work of thesis became a motivation for tool developers to provide better support for porting [46] and debugging [8] hybrid MPI/OmpSs programming model.

9.1 Future Work

Since 2008/2009 when we have proposed the first time MPI+SMPSs, later called MPI/OmpSs, standard number of cores per node was 4. Using a single core for communication thread took 1/4 of resources, thus was very costly. Nowadays, we face nodes with more than 32 cores and dedicating one core for MPI services is going to be less and less expensive. Thus the MPI/OmpSs will be more relevant in the future. Today there are special purpose chips designed for HPC and supercomputer are based on commercial chips build for servers. In the near future, we can expect chips build on HPC demand like SPARC64 XIfx 32 cores [53] and two assistant cores, where an assistant core is responsible for MPI communication and OS jitters. MPI/OmpSs fits perfectly to co-designed HPC chips and present already software technology for new hardware.

The MPI/OmpSs has already found its place at the HPC programming environment Further development of MPI/OmpSs programming model should support the latest hardware and software technology, particularly a new MPI Standard. MPI Standard 3.0 is already part of many MPI implementation and it offers new asynchronous interface for non-blocking collective communication, one-side communication and I/O operations. These new MPI routines increase overall MPI interface and give the flexibility of MPI but also introduces an additional level of complexity. The OmpSs runtime could consider a usage of new asynchronous interfaces and make them easy-to-use. One sided communication has not been explored in this work and it will be considered in the future as well.

The MPI standard offers a new MPI_T interface that gives a control over behavior of MPI implementation by using control and environment variables of the MPI implementation. For example, the OmpSs runtime could control the MPI protocol (eager/rendezvous) or the waiting mode (polling/blocking) through MPI_T interface. Exploring usage of MPI_T will be the topic of the future, where dynamic modification of MPI behavior will self-tune runs. The OmpSs would provide an optimal configuration of the MPI library during runs.

The working experience of the thesis stresses the importance of the OmpSs task scheduler. Analyzing ready to run tasks and schedule them in an optimal way is the key point. The scheduler already offers several scheduling policies but exploring data dependency graph and path analysis inside the graph is not considered. Smart prefetching of the OmpSs task will reduce latency of data transfer, which provides delay across nodes. In the future, machines become large and distances between nodes are significant, while applications become more and more memory bound thus latency and data locality will be the main issue.

There were several external users that have been porting their application to MPI/OmpSs programming model. While we were giving support to them, we realized that most applications suffered from the load imbalance across MPI nodes. The work [21] has partially solved the problem of load imbalance but the global load imbalance on MPI level due to static distribution and non deterministic algorithm used within MPI processes stays the dominant factor of low efficient application. Addressing this issue considers task stealing mechanism between MPI processes, remote execution of local task and the scheduler that is aware about whole MPI distribution. This would make the MPI/OmpSs close to the GAS programming model but the smart scheduling policy will avoid scalability issue of the GAS approach.

The productivity of the HPC programming model has to be higher we face today. Nevertheless, the MPI/OmpSs improves the productivity of current technology, it has been considered as a low level programming model. However, computational scientist prefers a high level weakly typed programming language as example[32]. Building software packages that will allow a single call of a solver and execute the solver across nodes will drastically increase the productivity. In the future, MPI/OmpSs would be seen as the standard part of scientific libraries. Building another abstraction on top of MPI/OmpSs is a part of further investigation.

10
Publications

Vladimir Marjanovic, Josep M. Perez, Eduard Ayguade , Jesus Labarta and Mateo Valero. Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach. Research Report. April 2009, UPC.

Montse Farreras, Vladimir Marjanovic, Eduard Ayguade, and Jesus Labarta. Gaining Asynchrony by Using Hybrid UPC/SMPSs. In Workshop on Asynchrony in the PGAS Programming Model (June 2009), Yorktown Heights, NY, 2009.

Vladimir Marjanovic, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Effective communication and computation overlap with hybrid MPI/SMPSs. In ACM Sigplan Notices, volume 45, pages 337–338. ACM, 2010.

Vladimir Marjanovic, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping communication and computation by using a hybrid mpi/smpss approach. In Proceedings of the 24th ACM International Conference on Supercomputing, pages 5–16.

ACM, 2010.

Jesus Labarta, Vladimir Marjanovic, Eduard Ayguadé, Rosa M Badia, and Mateo Valero. Hybrid parallel programming with mpi/starss. In PARCO, pages 621–628, 2011.

Vladimir Subotic, Steffen Brinkmann, Vladimir Marjanovic, Rosa M Badia, Jose Gracia, Christoph Niethammer, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Programmability and portability for exascale: Top down programming methodology and tools with starss. Journal of Computational Science, 4(6):450–456, 2013.

Rosa M Badia, Jesus Labarta, Vladimir Marjanovic, Alberto F Martín, Rafael Mayo, Enrique S Quintana-Ortí, and Ruymán Reyes.
Symmetric rank-k update on clusters of multicore processors with SMPSs. In PARCO, pages 657–664, 2011.

Dirk Brömmel, Paul Gibbon, Marta Garcia, Víctor López, Vladimir Marjanovic, and Jesús Labarta. Experience with the MPI/StarSs programming model on a large production code. In PARCO, pages 357–366, 2013.

Bibliography

[1] Comparison Istanbul and Nehalem., 2010. URL www.advancedclustering.com/company-blog/high-performance-linpack-on-xeon-5500-v-opteron-2400.html. 86

[2] NAS PARALLEL BENCHMARKS, web-site confirmed active on 15.03.2013. URL http://www.nas.nasa.gov/Resources/Software/npb.html. 112

[3] George Almási, Philip Heidelberger, Charles J Archer, Xavier Martorell, C Chris Erway, José E Moreira, B Steinmacher-Burow, and Yili Zheng. Optimization of mpi collective communication on bluegene/l systems. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 253–262. ACM, 2005. 76, 119

[4] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. Cray xc® series network. 47

[5] Ayon Basumallik and Rudolf Eigenmann. Towards automatic translation of openmp to mpi. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 189–198. ACM, 2005. 120

[6] 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. 24

[7] Ron Brightwell and Keith D Underwood. An analysis of the impact of mpi overlap and independent progress. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 298–305. ACM, 2004. 119

[8] Steffen Brinkmann, José Gracia, Christoph Niethammer, and Rainer Keller. Temanejo-a debugger for task based parallel programming models. *arXiv preprint arXiv:1112.4604*, 2011. 123

[9] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M Badia, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. Productive programming of gpu clusters with ompss. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 557–568. IEEE, 2012. 24

[10] William W. Carlson, Jesse M. Draper, and David E. Culler. S-246, 187 introduction to upc and language specification. 14

[11] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007. ISSN 1094-3420. doi: 10.1177/1094342007078442. URL http://dx.doi.org/10.1177/1094342007078442. 14

[12] 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. 14

[13] Juan Jose Costa, Toni Cortes, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Running openmp applications efficiently on an everything-shared sdsm. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 35. IEEE, 2004. 120

[14] 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. 15, 20

[15] 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. 5

[16] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003. 74, 120

[17] 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. 15, 22

[18] Silvia M Figueira. Using a hypercube algorithm for broadcasting in internet-based clusters. In *PDPTA*, 2000. 44, 120

[19] Anger Fog. Optimizing subroutines in assembly language an optimization guide for x86 platforms. p. 156 [version of 2011-06-08]. 46

[20] Dennis B Gannon and John Van Rosendale. On the impact of communication complexity on the design of parallel numerical algorithms. *Computers, IEEE Transactions on*, 100(12):1180–1194, 1984. 44

[21] Marta Garcia, Julita Corbalan, Rosa Maria Badia, and Jesus Labarta. A dynamic load balancing approach with smpsuperscalar and mpi. In *Facing the Multicore-Challenge II*, pages 10–23. Springer, 2012. 125

[22] Kazushige Goto and Robert van de Geijn. On reducing tlb misses in matrix multiplication. Technical report, Technical Report TR02-55, Department of Computer Sciences, U. of Texas at Austin, 2002. 46

[23] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *ACM SIGCOMM computer communication review*, 39(1):68–73, 2008. 1

[24] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. In *SC*, page 52, 2007. 16

[25] Torsten Hoefler, Peter Gottschling, Andrew Lumsdaine, and Wolfgang Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. *Parallel Computing*, 33(9):624–633, 2007. 102, 105, 120

[26] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. Implementation and performance analysis of non-blocking collective operations for mpi. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–10. IEEE, 2007. 41

[27] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010. 106

[28] John R Humphrey, Daniel K Price, Kyle E Spagnoli, Aaron L Paolini, and Eric J Kelmelis. Cula: hybrid gpu accelerated linear algebra routines. In *SPIE Defense, Security, and Sensing*, pages 770502–770502. International Society for Optics and Photonics, 2010. 120

[29] Mao Jiayin, Song Bo, Wu Yongwei, and Yang Guangwen. Overlapping communication and computation in mpi by multithreading. In *PDPTA*, pages 52–57, 2006. 41

[30] Gabriele Jost, Haoqiang Jin, Dieter an Mey, and Ferhat F Hatay. Comparing the openmp, mpi, and hybrid programming paradigms on an smp cluster. In *Proceedings of EWOMP*, volume 3, page 2003, 2003. 120

[31] Rainer Keller, Edgar Gabriel, Bettina Krammer, Matthias S Mueller, and Michael M Resch. Towards efficient execution of mpi applications on the grid: porting and optimization issues. *Journal of Grid Computing*, 1(2):133–149, 2003. 119

[32] Jeremy Kepner and Stan Ahalt. Matlabmpi. *Journal of Parallel and Distributed Computing*, 64(8):997–1005, 2004. 125

[33] Avinash Karanth Kodi and Ahmed Louri. Energy-efficient and bandwidth-reconfigurable photonic networks for high-performance computing (hpc) systems. *Selected Topics in Quantum Electronics, IEEE Journal of*, 17(2):384–395, 2011. 106

[34] Seung-Jai Min, Ayon Basumallik, and Rudolf Eigenmann. Optimizing openmp programs on software distributed shared memory systems. *International Journal of Parallel Programming*, 31(3):225–249, 2003. 120

[35] Yoshinori Ojima, Mitsuhisa Sato, Hiroshi Harada, and Yutaka Ishikawa. Performance of cluster-enabled openmp for the scash software distributed shared memory system. In *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pages 450–456. IEEE, 2003. 120

[36] Liang Peng, Mingdong Feng, and Chung-Kwong Yuen. Evaluation of the performance of multithreaded cilk runtime system on smp clusters. In *Cluster Computing, 1999. Proceedings. 1st IEEE Computer Society International Workshop on*, pages 43–51. IEEE, 1999. 120

[37] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E Fagg, Edgar Gabriel, and Jack J Dongarra. Performance analysis of mpi collective operations. *Cluster Computing*, 10(2):127–143, 2007. 119

[38] Rolf Rabenseifner. Hybrid parallel programming: Performance problems and chances. In *Proceedings of the 45th Cray User Group Conference, Ohio*, pages 12–16, 2003. 120

[39] Rolf Rabenseifner and Gerhard Wellein. Comparison of parallel programming models on clusters of smp nodes. In *Modeling, Simulation and Optimization of Complex Processes*, pages 409–425. Springer, 2005. 120

[40] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and*

Network-based Processing, 2009 17th Euromicro International Conference on, pages 427–436. IEEE, 2009. 119

[41] Mohammad J Rashti and Ahmad Afsahi. Improving communication progress and overlap in mpi rendezvous protocol over rdma-enabled interconnects. In *High Performance Computing Systems and Applications, 2008. HPCS 2008. 22nd International Symposium on*, pages 95–101. IEEE, 2008. 119

[42] John K Reid, Jan Marthedal Rasmussen, and Per Christian Hansen. The linpack benchmark in co-array fortran. In *Proceedings of the Sixth European SGI/Cray MPP Workshop, Septemeber*, 2000. 120

[43] Paulo Ribenboim. *The new book of prime number records*, volume 3. Springer New York, 1996. 48

[44] M. Snir, S. Otto, S. Huss-Laderman, D. Walker, and J. 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`. 14, 15

[45] Marc Snir, Steve W Otto, David W Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: the complete reference*. MIT press, 1995. 119

[46] Vladimir Subotic, Arturo Campos, Alejandro Velasco, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Tareador: The unbearable lightness of exploring parallelism. In *Tools for High Performance Computing 2014*, pages 55–79. Springer, 2015. 123

[47] Hong Tang and Tao Yang. Optimizing threaded mpi execution on smp clusters. In *Proceedings of the 15th international conference on Supercomputing*, pages 381–392. ACM, 2001. 119

[48] top500. Top500 List: List of top 500 supercomputers., 2012. URL `http://www.top500.org/`. 5, 74

[49] Sathish S Vadhiyar, Graham E Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 3. IEEE Computer Society, 2000. 41

[50] Ta Quoc Viet and Tsutomu Yoshinaga. Improving linpack performance on smp clusters with asynchronous mpi programming. *IPSJ Digital Courier*, 2:598–606, 2006. 120

[51] Ta Quoc Viet, Tsutomu Yoshinaga, and Masahiro Sowa. Optimization for hybrid mpi-openmp programs with thread-to-thread communication. *Institute of Electronics, Information and Communication Engineers (IEICE) Technical Eeport*, pages 19–24, 2004. 120

[52] Kathy Yelick and UC Berkeley. Performance and productivity opportunities using global address space programming models, 2006. 120

[53] Takafumi Yoshida, Mikio Hondou, Takekazu Tabata, Ryuji Kan, Naohiro Kiyota, Hiroyuki Kojima, Koji Hosoe, and Hiroshi Okano. Sparc64 xifx: Fujitsu's next generation processor for hpc. 2014. 124

[54] Yao Zhang and John D Owens. A quantitative performance analysis model for gpu architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382–393. IEEE, 2011. 120