ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Τεχνικές βελτιστοποίησης για παράλληλο λογισμικό μεγάλης κλίμακας

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ευάγγελος Α. Γεωργανάς

Επιβλέπων: Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής

Αθήνα, Ιούλιος 2011

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Τεχνικές βελτιστοποίησης για παράλληλο λογισμικό μεγάλης κλίμακας

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ευάγγελος Α. Γεωργανάς

Επιβλέπων: Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 18η Ιουλίου 2011

....	....	....
Ν. Κοζύρης	Ν. Παπασπύρου	Δ. Φωτάκης
Αν. Καθηγητής ΕΜΠ	Επ. Καθηγητής ΕΜΠ	Λέκτορας ΕΜΠ

Αθήνα, Ιούλιος 2011

....
Ευάγγελος Α. Γεωργανάς
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Ένα από τα πιο δύσκολα προβλήματα στα συστήματα παράλληλης επεξεργασίας είναι η ανάπτυξη παράλληλου λογισμικού το οποίο κλιμακώνει αποδοτικά. Αρκετές εφαρμογές δεν κλιμακώνουν έπειτα από έναν αριθμό επεξεργαστών επειδή το κόστος επικοινωνίας γίνεται συγκρίσιμο με την ωφέλιμη υπολογιστική εργασία. Σκοπός αυτής της διπλωματικής εργασίας είναι η υλοποίηση ενός σχήματος παράλληλης εκτέλεσης που επικαλύπτει υπολογισμούς με επικοινωνία ώστε τελικά να μειώνεται ο συνολικός χρόνος εκτέλεσης της εφαρμογής. Χρησιμοποιούμε την επικαλυπτόμενη δρομολόγηση σε τέσσερις αντιπροσωπευτικές εφαρμογές: (α)Τον αλγόριθμο Floyd -Warshall που χρησιμοποιεί συλλογική επικοινωνία, (β)τον αλγόριθμο Smith-Waterman που είναι μία τυπική εφαρμογή δυναμικού προγραμματισμού, (γ)την εξίσωση μεταφοράς τριών διαστάσεων που απαιτεί επικοινωνία των διεργασιών μόνο προς ορισμένη κατεύθυνση και (δ)την εξίσωση διάδοσης θερμότητας τριών διαστάσεων, όπου κάθε διεργασία ανταλλάζει δεδομένα με όλες τις γειτονικές της διεργασίες. Υλοποιούμε αυτήν την τεχνική με δύο τρόπους, αρχικά χρησιμοποιώντας ασύγχρονη επικοινωνία και έπειτα αναθέτοντας το υπολογιστικό κομμάτι και την επικοινωνία σε διαφορετικά νήματα. Η τελευταία υλοποίηση βασίζεται σε υβριδικό προγραμματισμό με MPI και OpenMP. Η πλατφόρμα εκτέλεσης είναι μία συστοιχία από κόμβους πολυεπεξεργαστών με μοιραζόμενη μνήμη (SMP) και τα διαθέσιμα δίκτυα διασύνδεσης είναι Gigabit Ethernet και Myrinet. Εξετάζουμε την βιωσιμότητα αυτής της τεχνικής και στα δύο δίκτυα διασύνδεσης και παρατηρούμε βελτίωση στην επίδοση έως 40% σε σχέση με τις απλές παράλληλες εφαρμογές.

Λέξεις κλειδιά: επικάλυψη επικοινωνίας, υβριδικός προγραμματισμός, SMP συστοιχία, Floyd-Warshall, Smith-Waterman, εξίσωση μεταφοράς τριών διαστάσεων, εξίσωση διάδοσης θερμότητας τριών διαστάσεων

Abstract

One of the most difficult problems in parallel computing is to develop parallel software that has effective scalability. Several applications do not scale further than a number of processors because communication overhead becomes comparable with computational work. The goal of this diploma thesis is to implement a parallel execution scheme that overlaps computation and communication and eventually reduces the overall execution time of the application. We utilize this overlapping schedule in four representative applications: (a)The Floyd-Warshall algorithm that uses collective communication, (b)the Smith-Waterman algorithm which is a typical dynamic programming application, (c)the 3D advection equation that requires the communication between processes only towards a specific direction and (d)the 3D heat equation that has a "halo" communication pattern, i.e. every process exchanges data with all of its neighbors. We develop this optimization technique in two ways, initially we use non-blocking communication functions and then by assigning the computation and the communication workload to separate threads. The latter implementation is based on hybrid programming with MPI and OpenMP. Our execution platform is a cluster of SMP nodes and there are available a commodity interconnect (Gigabit Ethernet) and a high performance network (Myrinet). We investigate the viability of this optimization technique regarding both interconnection networks and we notice improvement in performance up to 40% relative to the baseline parallel applications.

Keywords: communication overlapping, hybrid programming, SMP cluster, Floyd-Warshall, Smith-Waterman, 3D advection equation, 3D heat equation

Ευχαριστίες

Η διπλωματική εργασία αυτή πραγματοποιήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου, υπό την επίβλεψη του Αναπληρωτή Καθηγητή Νεκτάριου Κοζύρη.

Καταρχήν θα ήθελα να ευχαριστήσω τον καθηγητή μου κ. Νεκτάριο Κοζύρη, για την εποπτεία του κατά την εκπόνηση της εργασίας μου, αλλά και για τη συμβολή του στη διαμόρφωσή μου ως μηχανικού από τις διδασκαλίες του.

Θα ήθελα επίσης να εκφράσω τις ευχαριστίες μου σε όλα τα μέλη του εργαστηρίου και ιδιαίτερα στον Μεταδιδακτορικό Ερευνητή Γιώργο Γκούμα για τη συνεχή καθοδήγηση και ενθάρρυνση που μου προσέφερε για την ολοκλήρωση της διπλωματικής μου εργασίας.

Τέλος, θα ήθελα να ευχαριστήσω το οικογενειακό και φιλικό μου περιβάλλον, η υποστήριξη των οποίων με βοήθησε τόσο στην εκπόνηση της εργασίας όσο και στην ολοκλήρωση των προπτυχιακών μου σπουδών.

Contents

Chapter 1

Introduction

1.1 Overview

Since the start of the personal computing revolution one of the things application vendors could count on was that new hardware made applications faster. Disk, memory, bus, processors - always became even more and always faster. Application developers typically did not need to do anything - if the application ran on the operating system and the operating system ran on the new hardware, the application ran faster.

That was the case until now. Single core clock speeds maxed out and consequently multicore is the next evolution in computing performance. Putting more cores on a single processor seems like a straightforward way to deliver big performance increases. With a first sight this evolution is ideal, except for two things: managing multiple cores creates some significant overhead and most applications require non-trivial re-engineering to parallelize the code and take advantage of more than one core. In other words, without some considerable work there are some applications that will run slower on multicore systems.

Moreover, there are scientific applications extremely demanding in computational sources like weather forecasting and protein synthesis. In addition, there are non-scientific applications that are computationally demanding, e.g. search engines, web servers and databases. All these applications may need notable time to execute but the user would like to have the results as soon as possible, either because it could prevent unwelcome situations (see well-timed forecasting of natural disasters) or because long waits bring resentment (for instance imagine a search on the web that would last some minutes).

To deal effectively with this family of applications, parallel processing has become a promising paradigm in computational models and parallel computer architecture has been introduced to support these applications. Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multicore computers having multiple processing elements within a single machine, while clusters use multiple computers to work on the same task.

The tall challenge is to create hardware and software that will make it easy to write correct parallel processing programs[1] that will execute efficiently in performance and power as the number of cores per chip and the number of nodes per cluster scale up. There are rising questions relative to multicore processors, clusters and parallel programs that will run on such systems:

- Why is it difficult to create efficient parallel processing programs?
- Which parallel architectures are available?
- Given the difficulty of writing programs to run well on parallel hardware, which parallel programming models are available in order to simplify the task?

These questions will be answered in the following sections of this chapter.

1.2 The challenge of parallel programming

The availability of multicore processors does not guarantee that applications will execute faster without any effort. In several cases, the software needs to be redesigned in order to run in parallel and take advantage of the available cores. First of all, it is harder to develop a parallel program than a sequential one because eventually the parallel program *must* have better performance and efficiency, otherwise there is no sense in re-writing programs, particularly because sequential programs for uniprocessors are much easier to be written.

So the question is why writing fast parallel processing programs is a hard task especially as the number of cores increases. The first performance-restrictive factor is that the computational load has to be broken into equal-sized pieces and then each piece should be assigned to a core. In case that the pieces are not equal-sized, some cores would be idle while waiting for the other ones with the larger pieces to finish. Another factor that could limit performance is that different processors have to communicate with each other in order to process successfully their workload. Obviously, if the time spent for communication is significant part of the total time then the speedup of the parallel program would be downgraded. In short, scheduling, load balancing, time for synchronization and overhead for communication between the cores are significant challenges and they become even greater as the number of processors increases.

Another fact that should be taken in deep consideration when writing parallel programs is *Amdahl's law* [1]. Let us first define speedup in a given problem as:

$$speedup(n\ processors) \equiv \frac{execution\ time(1\ processor)}{execution\ time(n\ processors)} \tag{1.1}$$

[1]We use the term parallel processing program to refer to a single program that runs on multiple processors simultaneously

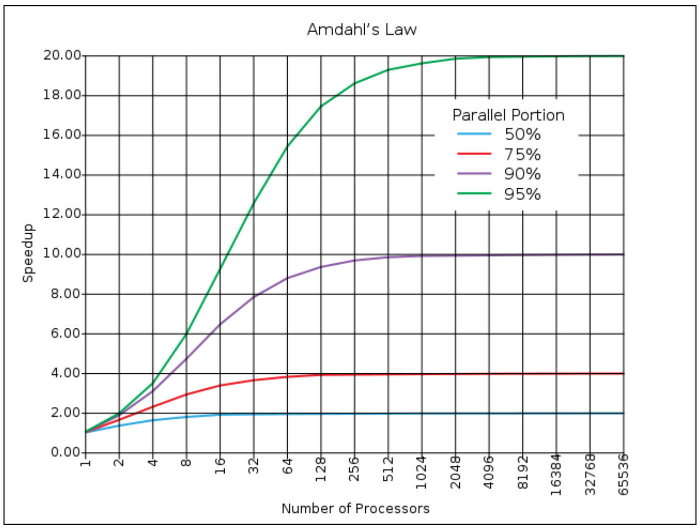


Figure 1.1: Total speedup of a parallel program as parallel portion and number of processors increase

Amdahl's law refers to the increase of the performance that can be achieved by improving a part of the total program, let it be a proportion p. If this improvement makes that part of the program s times faster, i.e. it has a speedup equal to s, then the total speedup of the program is given by the equation:

$$total\ speedup \equiv \frac{1}{(1-p)+\dfrac{p}{s}} \tag{1.2}$$

In our case, Amdahl's law can be rephrased: If p is the proportion of the total program that can be parallelized and $1-p$ is the part of the program that can not be parallelized (i.e. remains sequential), then the maximum total speedup that can be achieved by using N processors is given by the equation:

$$total\ speedup \equiv \frac{1}{(1-p)+\dfrac{p}{N}} \tag{1.3}$$

As N goes to infinity, the maximum speedup goes to $1/(1-p)$. Practically, the analogy between efficiency and cost increases dramatically even if $1-p$ is relatively small. For instance, if $p = 90\%$ then $1-p = 10\%$, so the program can be accelerated by a factor of 10 in the best case, no matter how many processors we use. At Figure 1.1 we can see how total speedup is affected while proportion p and number of processors N change and we can draw some conclusions. First of all, the overall speedup of a program that utilizes many cores in a parallel portion is bounded by the sequential part of the program. Therefore, using even more cores and improving parallel computer architectures is not panacea. Instead, an application should be redesigned so that a larger part of it is parallelized.

1.3 Multiprocessor architectures

According to *Flynn's taxonomy* [2] computer systems are categorized in terms of how they use data, and how that data is controlled or manipulated. It defines two control streams: SI (single instruction) and MI (multiple instruction), and two data streams: SD (single data) and MD (multiple data). Any computer system will have one type of control stream and one type of data stream, so there are four categories:

- **SISD**: A sequential computer
- **SIMD**: Massively parallel processors, same instructions, multiple data (data-level parallelism)
- **MISD**: Unusual – there are few machines in this category, none that have been commercially successful or had any impact on computational science
- **MIMD**: Each processor executes its own instructions and processes its own data. There are multiple threads (thread-level parallelism). Examples of MIMD systems are: clusters (commodity/custom clusters) and multicore systems

We will examine MIMD multiprocessor architectures according to their memory organization. They can be placed in three categories that will be further analyzed in the next subsections: shared memory architectures, distributed memory architectures and hybrid architectures.

1.3.1 Shared memory architecture

A *shared memory multiprocessor (SMP)* offers the programmer a single physical address space across all processors, although a more accurate term would have been shared-address multiprocessor. It is important to clarify that such systems can run independent tasks in their own virtual address spaces, even if they all share a physical address space. Processors communicate through shared variables in memory, with all processors being capable of accessing any memory location via loads and stores. Moreover, each processor has its local cache hierarchy. Figure 1.2 shows the classic organization of an SMP.

Single address space multiprocessors come in two memory organizations: The first takes about the same time to access main memory independently of which processor requests it and independently of which word is requested. These machines are called *uniform memory access (UMA)* multiprocessors. In the second category, some memory accesses are much faster than others, depending on which processor asks for which word. Such machines are called *non-uniform memory access (NUMA)* multiprocessors. As it can be inferred, NUMA multiprocessors conceal harder programming challenges than UMA multiprocessors, but on the

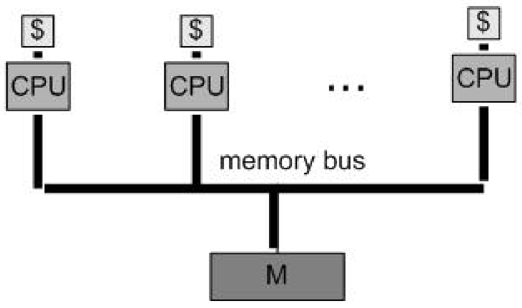


Figure 1.2: Classic organization of an SMP

other hand NUMAs can have lower latency to nearby memory and significantly higher aggregate memory bandwidth.

As processors operating in parallel will normally share data, they also need to coordinate when operating on shared data. Otherwise, a processor could start working on a data before another processor is finished with it. Thus, when sharing is supported with a single address space there must be a mechanism for synchronization. An approach to achieve synchronization is the usage of locks for shared variables. Only one processor at a time can acquire the lock and any other processor interested in shared data must wait until the original processor unlocks the variable.

The ability to access all shared data efficiently from any of the processors using ordinary loads and stores, together with the automatic movement and replication of shared data in the local caches, makes them attractive for parallel programming. These features are also very useful for the operating system, whose different processes share data structures and can easily run on different processors. However, this approach has been used for connecting very small number of processors, typically up to 20 or 30. The scaling limit for these machines comes primarily due to bandwidth limitations of the shared bus and memory system.

1.3.2 Distributed memory architecture

Another approach to sharing an address space is that each processor has its own cache hierarchy and its own private physical address space. Figure 1.3 shows the classic organization of a distributed memory architecture which is also called message-passing architecture. Such a system has routines to send and receive messages and the coordination is built in with message passing, since one processor knows when a message is sent and the receiving processor knows when a message arrives. If the sender needs information that the message has arrived, the receiving processor can then send an acknowledgement message back to the sender. Note that unlike the SMP, the interconnection network is between processor-memory nodes.

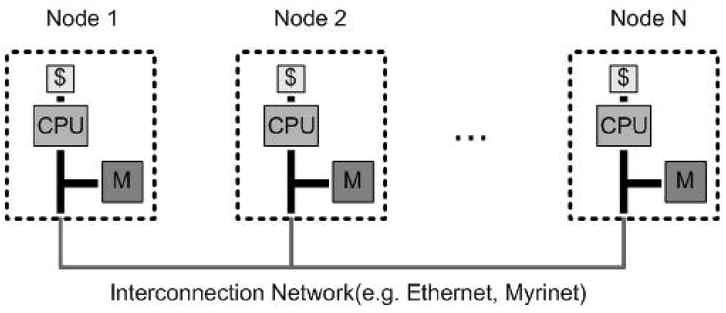


Figure 1.3: Classic organization of a distributed memory architecture

There were several attempts to build high-performance computers based on high performance message-passing networks and they did offer better absolute communication performance than clusters built using commodity networks. Clusters are generally collections of commodity computers that are connected to each other over their I/O interconnect via standard network switches and cables. Each one computer runs a distinct copy of the operating system.

One drawback of clusters has been that the cost of administering a cluster of n machines is about the same as the cost of administering n independent machines, while the cost of administering a shared memory multiprocessor with n processors is about the same as administering a single machine.

Another drawback of clusters is that the processors in a cluster are usually connected using the I/O interconnection of each computer, whereas the cores in a multiprocessor are usually connected via the memory interconnect of the computer. The memory interconnect has higher bandwidth and lower latency, allowing much better communication performance.

The downside for programmers is that it's harder to port a sequential program to a message-passing computer, since every communication must be identified in advance and explicitly implemented. However, the weakness of separate memories for user memory turns into a strength in the availability of the system. The cluster software is a layer that runs on top of local operating systems running on each computer, therefore it is much easier to disconnect and replace a broken machine.

Finally, a great advantage of distributed memory systems has to do with their scalability. Given that clusters are constructed from whole computers and independent, scalable networks, this architecture scales up to some thousands of nodes.

1.3.3 Hybrid architecture

The hybrid architecture combines the previous two architectures: nodes with shared memory architecture are connected via an interconnection network using the distributed memory architecture. Figure 1.4 illustrates the organization of a hybrid architecture. Obviously, this architecture shows up the advantages of

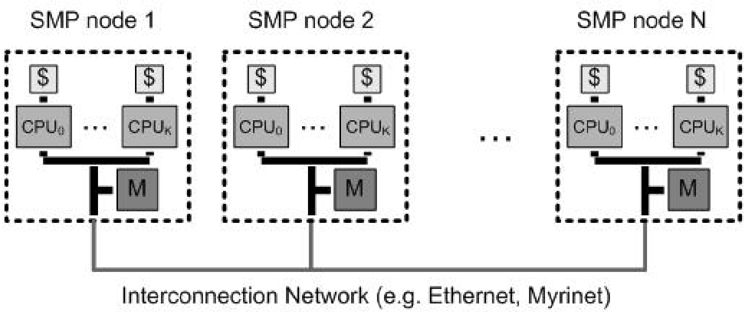


Figure 1.4: Example of a hybrid architecture

the previous two types and therefore it is the typical architecture of the modern clusters and supercomputers.

1.4 Parallel programming models

A parallel programming model is a concept that enables the expression of parallel programs which can be compiled and executed. The value of a programming model is usually judged on its generality: how well a range of different problems can be expressed and how well they execute on a range of different architectures. The implementation of a programming model can take several forms such as libraries invoked from traditional sequential languages, language extensions, or complete new execution models.

One way to classify parallel programming models is according to the process interaction that takes place in a parallel processing program. Process interaction relates to the mechanisms by which parallel processes are able to communicate with each other. The most common forms of interaction are shared memory and message passing. Furthermore, there is a mixed-type interaction between processes and this represents the hybrid parallel programming model.

Parallel programming models exist as an abstraction above hardware and memory architectures. Although it may not seem apparent, these models are *not* specific to a particular type of machine or memory architecture. In fact, any of those models can, theoretically, be implemented on any underlying hardware. For example, a shared memory model may apply on a distributed memory machine if the physically distributed machine memory is appeared to the user as a single shared memory (global address space). Which model to use is often a combination of what is available and personal choice. There is no “best” model, although there certainly are better implementations of some models over others.

1.4.1 Shared address space programming model

In a shared address space model, parallel tasks share a address space which they read and write to asynchronously. This requires protection mechanisms such as locks and semaphores to control concurrent access to the shared memory. Shared memory can be also emulated on distributed-memory systems but non-uniform memory access (NUMA) times can come into play.

An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. As a consequence, program development can often be simplified. On the other hand, an important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality. Keeping data local to the processor that works on it implies memory accesses, cache refreshes and bus traffic when multiple processors use the same data. Unfortunately, controlling data locality is hard to understand and beyond the control of the average user.

On shared memory platforms, the native compilers translate user program variables into actual memory addresses, which are global. An implementation that supports shared memory programming is OpenMP and will be discussed in following chapter.

1.4.2 Message–passing programming model

In a message passing programming model, a set of tasks use their own local memory during computation and multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines. Tasks exchange data through communication by sending and receiving messages. In this programming model there is a distance between itself and the real operations of the hardware, since the communication on the user's level is executed by the operating system or by function calls from a library that in turn execute many low-level operations, including the communication. At the user level, the most common communication operations are variants of simple send and receive operations. In the naive version, a send specifies a local buffer of data that will be sent and a receiving process. A receive operation specifies a sending process and a local buffer of data in which the received data will be stored. So, each send has a matching receive and together they accomplish data transfer from a process to another, as it is illustrated in Figure 1.5.

In most message passing systems, the operation of send is allowed to have attached to a message an identifier or a tag, and the operation of receive could specify a matching rule for the identifiers and the tags. In other words, the user's program renames local addresses and records to an abstract place of process-tags. Each combination of a send and the corresponding receive achieves copy from memory to memory, where each endpoint of the communication determines the address of its local data.

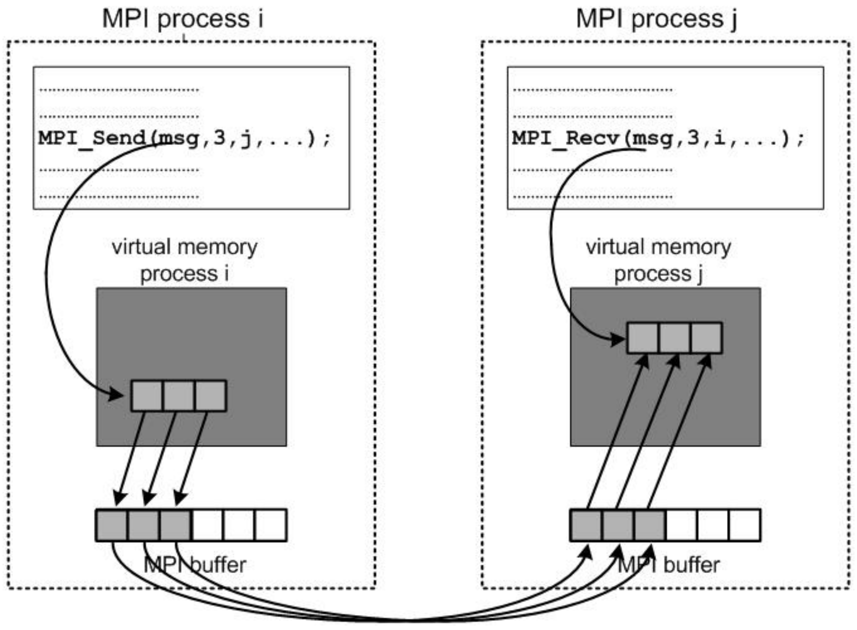


Figure 1.5: The operations of send/receive from a user's level viewpoint

As previously mentioned, there are many versions of send and receive operations. *Synchronous* communications require some type of "handshaking" between tasks that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer. Synchronous communications are often referred to as *blocking* communications since other work must wait until the communications have completed. *Asynchronous* communications allow tasks to transfer data independently from one another. For instance, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data does not matter. Asynchronous communications are often referred to as *non-blocking* communications since other work can be done while the communications are taking place.

Another classification of communication in a parallel program has to do with the number of tasks that take part in it. *Point-to-point* communication involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer. On the other hand, *collective* communication involves data sharing between more than two tasks, which are often specified as being members in a common group. For example, imagine the scenario where a task has to send some data possessed by it to a group of tasks. The collective way of sending that data is to *broadcast* it to the desired group of tasks.

From a programming perspective, message passing implementations commonly embody a library of subroutines that are used in source code. The programmer is responsible for determining all parallelism. Historically, a variety of message passing libraries have been available since the 1980's. These implementations differed substantially from each other making it difficult for programmers to develop portable applications. In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations. MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. For shared memory architectures, MPI implementations usually do not use a network for task communications. Instead, they use shared memory (memory copies) for performance reasons. MPI will be discussed in more detail at a following chapter.

1.4.3 Hybrid programming model

In this model, the previous parallel programming models are combined. A common example of a hybrid model is the combination of the message passing model (MPI) with the shared memory model (OpenMP). This hybrid model fits well to the increasingly common hardware environment of networked SMP machines. OpenMP is used to parallelize a program interior a node of a SMP cluster and MPI is used for the communication between processes that reside in different nodes of a cluster. More details in implementing such a programming model will be described in following chapter.

Chapter 2

Execution environment

In the previous chapter, parallel architectures and programming models were discussed in general. Now we will analyze an execution platform which is part of our case study, as our parallel processing programs will run there.

2.1 Architecture of the system

Our system embraces the hybrid parallel architecture, i.e. it is a cluster consisting of SMP nodes where each node has two Intel® Xeon® E5335 processors (eight cores in total). The clock frequency of each core is 2.00 GHz and the cores share in a pairwise way the same level two cache, which is 4 MB. Additionally, each SMP node has one memory slot, whose size varies from node to node. Finally, the SMP nodes are connected via an interconnection network and form a hybrid system. Totally, there are 16 SMP nodes available in our cluster and also two different types of interconnection network may be used: Gigabit Ethernet and Myrinet.

2.2 Interconnection Network

The overall performance of a cluster system can be determined by the speed of its processors and the interconnection network. Regardless of how fast the processors are, communication among processors, and hence scalability of applications, is in several cases bounded by the network bandwidth and latency. The *bandwidth* is an indication of how fast a data transfer may occur from a sender to a receiver, while *latency* is the time needed to send a minimal size message from a sender to a receiver. In the early days of clusters, Ethernet was the main interconnection network used to connect nodes. Many solutions have been introduced to achieve high-speed networks. Key solutions in high-speed interconnects include Gigabit Ethernet and Myrinet.

2.2.1 Gigabit Ethernet

Ethernet [3], in general, is a packet-switched LAN technology introduced by Xerox PARC in the early 1970's. Ethernet was designed to be a shared bus technology where multiple hosts are connected to a shared communication medium. All hosts connected to an Ethernet receive every transmission, making it possible to broadcast a packet to all hosts at the same time. Ethernet uses a distributed access control scheme called Carrier Sense Multiple Access with Collision Detect (CSMA/CD). Multiple machines can access an Ethernet at the same time. Each machine senses whether a carrier wave is present to determine whether the network is idle before it sends a packet. Only when the network is not busy sending another message can transmission start. Each transmission is limited in duration and there is a minimum idle time between two consecutive transmissions by the same sender. In order to achieve an acceptable level of performance and to eliminate any potential bottleneck, there must be some balance between the Ethernet speed and the processor speed. Essentially, Gigabit Ethernet is the technology for transmitting Ethernet frames at a rate of a gigabit per second.

In our case, when Gigabit Ethernet is selected to be the interconnection network, the message passing of MPI uses the TCP protocol. TCP supports message communication over Ethernet using the socket interface to the operating system's network protocol stack. Thus, when using TCP over Ethernet, memory copies are required to move data between the application and the kernel. This procedure is analyzed in [4]. We should have this issue in mind as it will be discussed again in later chapter.

2.2.2 Myrinet

Myrinet [5] is a high-speed local area networking system designed by Myricom to be used as an interconnect between multiple machines to form computer clusters. Myrinet has much lower protocol overhead than standards such as Ethernet, and therefore provides better throughput, less interference, and lower latency while using the host CPU. Although it can be used as a traditional networking system, Myrinet is often used directly by programs that "know" about it, thereby bypassing a call into the operating system. Myrinet physically consists of two fibre optic cables, upstream and downstream, connected to the host computers with a single connector. Machines are connected via low-overhead routers and switches, as opposed to connecting one machine directly to another. Myrinet includes a number of fault-tolerance features, mostly backed by the switches. These include flow control, error control, and "heartbeat" monitoring on every link. The newest, "fourth-generation" Myrinet, called Myri-10G, supports a 10 Gbit/s data rate and is interoperable with 10 Gigabit Ethernet on the physical layer (cables, connectors, distances, signaling).

Myrinet is a lightweight protocol with little overhead that allows it to operate with throughput close to the basic signaling speed of the physical layer. However,

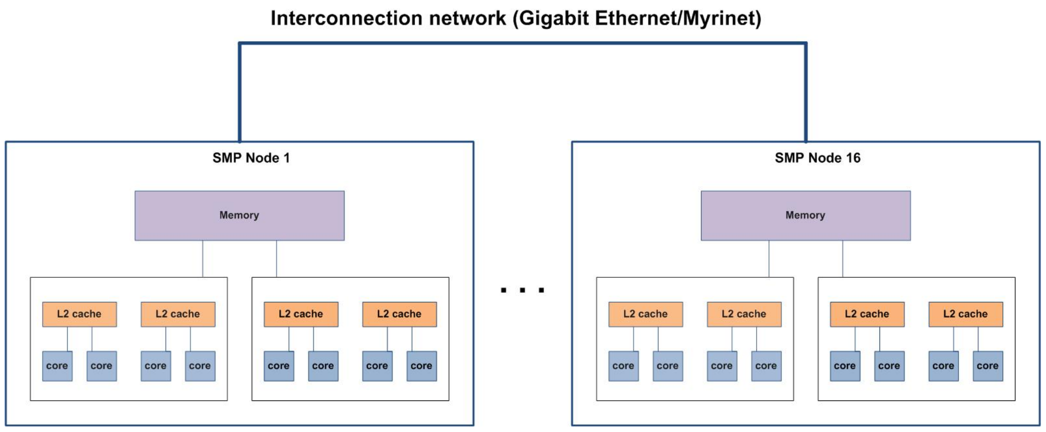


Figure 2.1: An overview of the architecture of the cluster

for supercomputing, the low latency of Myrinet is even more important than its throughput performance, since, according to Amdahl's law, a high-performance parallel system tends to be bottlenecked by its slowest sequential process, which is often the latency of message transmission across the network.

In contrary to Ethernet, user-level communication protocols employed by Myrinet avoid memory copies to move data between the application and the kernel. This is achieved by directly transferring data between the network interface and application buffers, resulting finally in lower communication latencies.

In Figure 2.1 we illustrate an overview of the architecture of the cluster.

2.3 Implementation of a hybrid programming model

In this section we discuss issues relevant to shared address space programming, message passing and, finally, how hybrid programming is achieved by using OpenMP and MPI.

2.3.1 OpenMP

In OpenMP's execution model, programs execute serially until they encounter the *parallel* directive. This directive is responsible for creating a group of threads, where each thread has a unique thread id. The exact number of threads can be specified by setting appropriately an environment variable, or at runtime using OpenMP functions. The main thread that encounters the parallel directive becomes the master of this group of threads and is assigned the thread id 0 within the group. Then, each thread created by this directive executes the structured block specified by the parallel directive. Finally, at the end of a parallel region the

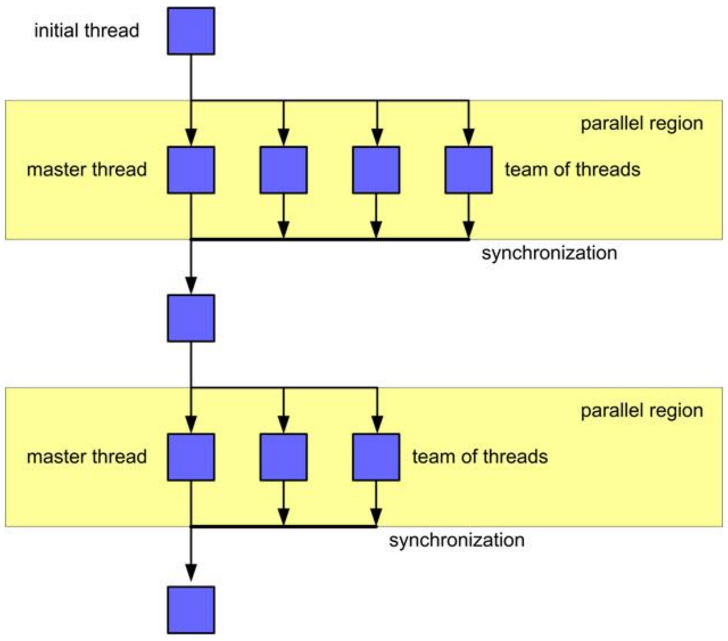


Figure 2.2: OpenMP execution model

threads are synchronized. This execution model is illustrated in Figure 2.2.

The parallel directive includes a clause list which is used to specify conditional parallelization, number of threads and data handling. In regard with data handling, a variable used in a parallel region can be private, firstprivate, lastprivate and shared. A private variable is local to each thread, this is to say that each thread has its own copy for that variable. The difference between a private and a firstprivate variable is that the latter is initialized to a corresponding value before the parallel region. Also, a lastprivate variable is updated after the parallel construct. Finally, a shared variable is shared across all the threads, i.e. there is only one copy for this variable. Special care must be taken while handling shared variables by threads to ensure serializability, since race conditions might be concealed. This model of data is illustrated in Figure 2.3.

The work is shared among threads in a parallel region using the directives *for*, *sections* and *single*. Specifically, the *for* directive distributes iterations of a for-loop to a group of threads, the *sections* directive specifies one or more independent code sections and they are assigned to the threads, while the *single* directive declares that a code section should be executed by a sole thread of the group. We should note here that work sharing could be also carried out by assigning separate code sections to threads according to their thread ids.

Although, synchronization of the threads is implied at the end of parallel regions, sometimes it may be necessary for a programmer to force synchronization explicitly. Therefore, synchronization constructs are available and provide many functionalities, e.g. the *barrier* construct results in the synchronization of the

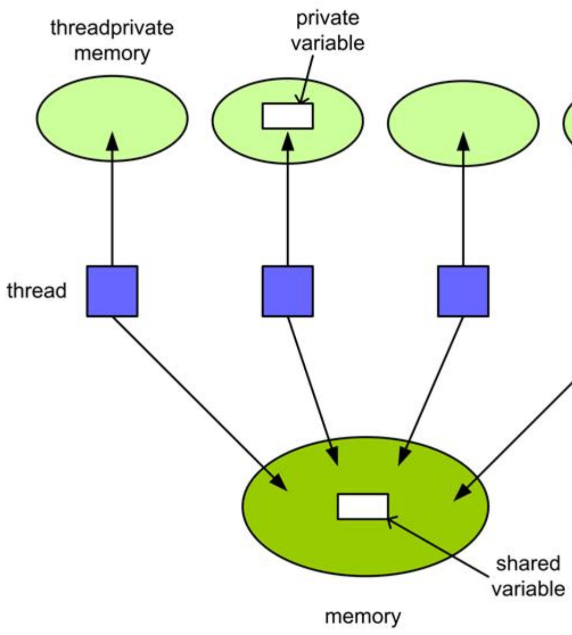


Figure 2.3: OpenMP data model

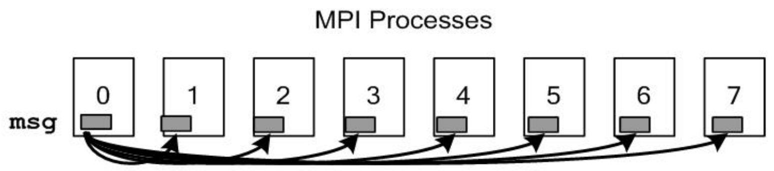
threads of the group, the *master* construct specifies code that is executed only by the master thread etc. Moreover, the OpenMP run-time library supports a set of simple and nestable lock routines.

Finally, OpenMP supports reduction operations and in its last version (OpenMP 3.0) the *task* construct appeared, which is useful for parallelization when an application produces work dynamically. For more details about OpenMP we refer the reader to [6]. An example of code using OpenMP directives is illustrated in Code 2.1, where functions $f1()$ and $f2()$ are executed in parallel.

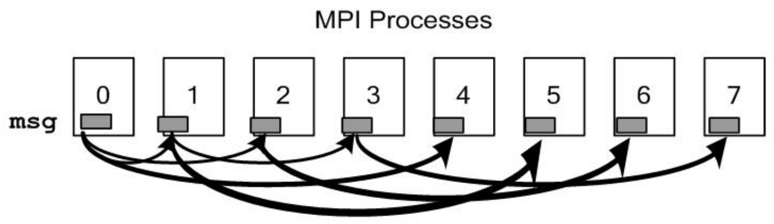
Code 2.1 OpenMP code where $f1()$ and $f2()$ are executed in parallel

```
#pragma omp parallel sections
{
   #pragma omp section
       f1();

   #pragma omp section
       f2();
}
```



(*a*)



(*b*)

Figure 2.4: (*a*) Point to point communication (*b*) Collective communication

2.3.2 MPI

As mentioned in section 1.4.2, MPI is the "de facto" industry standard for message passing and most message-passing programs are written using the single program multiple data (SPMD) approach. In SPMD programs the code executed by different processes is identical. Moreover, each process has a unique rank in a group of processes (such a group of processes is called *communicator*), and each process works on a subset of the total data or differentiates its execution flow according to its rank.

We focus on the communication functions, which are the heart of the message passing paradigm. The point-to-point blocking communication is realized by the functions *MPI_Send* and *MPI_Recv*. Their arguments specify in general the sender, the receiver, the amount of transferred data and its datatype. At this point we remind that each send must have a matching receive. The respective non-blocking functions for point-to-point communication are *MPI_Isend* and *MPI_Irecv*. Additionally, *MPI_Test* and *MPI_Wait* functions are used to check whether or not a non-blocking send or receive operation has finished.

Regarding collective communication, *MPI_Bcast* is the function that makes a source process broadcast data to a communicator specified in the arguments. We emphasize that collective communication is usually very efficient, e.g. sending a message in a communicator that has p processes will take $\log p$ steps instead of *p-1* steps that would be needed if we used point-to-point communication, as it can be shown at Figure 2.4. However, *MPI_Bcast* and all collective communication functions are blocking which means that collective operations must be completed before continuing the execution of code. Another significant collective function is *MPI_Reduce*, which allows the process to perform an operation like min, max, sum, product etc on data possessed by every process in a communicator. The final result after applying the operation is stored at *root* process which is an argument of the function. Moreover, *MPI_Scatter* and *MPI_Gather* functions are used to scatter and gather data respectively. The latter functions become very useful when initial data, like an array, has to be partitioned and distributed by a source process to the others, or when messages from every process have to be gathered and merged to a specific process. At this point, we should note that collective communication *must involve all processes* in the scope of a communicator.

Finally, *MPI_Barrier* function is used to synchronize the processes of a communicator and if it is not used carefully the parallelism is limited. We refer the reader to [7] for an extensive description of the aforementioned MPI functions. An example of MPI code that computes in parallel the expression $f(0)+f(1)$ is shown in Code 2.2.

Code 2.2 MPI code where the expression $f(0)+f(1)$ is computed in parallel

```
#include <mpi.h>

int main(int argc, char** argv) {
    int v0, v1, sum, rank;
    MPI_Status stat;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank==1) {
        v1=f(1);
        MPI_Send(&v1,1,0,50,MPI_INT,MPI_COMM_WORLD);
    } else if (rank==0) {
        v0=f(0);
        MPI_Recv(&v1,1,1,50,MPI_INT,MPI_COMM_WORLD,&stat);
        sum=v0+v1;
    }
    MPI_Finalize();
    return 0;
}
```

2.3.3 Combining MPI and OpenMP

So far we have discussed basic attributes of MPI and OpenMP, but the ultimate goal was to implement a hybrid programming model. Since each MPI process can call OpenMP routines, then it is straightforward how to create hybrid programs. So, when a MPI process creates threads by using a *parallel* directive, then these threads are assigned (if possible) to different processors on the SMP node that the MPI process runs. This scheme is very flexible and provides the programmer with many parallelization opportunities. An example of hybrid code that computes in parallel the expression $f(0)+f(1)+f(2)+f(3)$ is shown in Code 2.3. In this code we create two MPI processes and each process has available two threads. So the first MPI process computes in parallel the subexpression $f(0)+f(1)$ by using a OpenMP sections directive (and utilizing the two available threads) and so does the second MPI process regarding subexpression $f(2)+f(3)$.

Code 2.3 Hybrid code where the expression $f(0)+f(1)+f(2)+f(3)$ is computed in parallel

```
int main(int argc, char** argv) {
    int v0, v1, v2, v3, subexpr, sum, rank;
    MPI_Status stat;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    omp_set_num_threads(2);
    if (rank==1) {
        #pragma omp parallel sections
        {
            #pragma omp section
                v0=f(0);
            #pragma omp section
                v1=f(1);
        }
        subexpr=v0+v1;
        MPI_Send(&subexpr,1,0,50,MPI_INT,MPI_COMM_WORLD);
    } else if (rank==0) {
        #pragma omp parallel sections
        {
            #pragma omp section
                v2=f(2);
            #pragma omp section
                v3=f(3);
        }
        sum=v2+v3;
        MPI_Recv(&subexpr,1,1,50,MPI_INT,MPI_COMM_WORLD,&stat);
        sum+=subexpr;
    }
    MPI_Finalize();
    return 0;
}
```

Chapter 3

Overlapping computation with communication

3.1 Motivation

In section 1.2 we discussed the difficulty of creating efficient parallel programs. Load-balancing is one of the factors that can reduce the performance of a parallel application dramatically. Another issue that a programmer should have in mind when creating parallel applications is *granularity* of parallelism, i.e. a qualitative measure of the ratio of computation to communication. In a typical parallel program, periods of computation are separated from periods of communication by synchronization events. Therefore according to the ratio computation/communication the parallelism could be *fine-grain* or *coarse-grain*.

In the first case, relatively small amounts of computational work are carried out between communication events and this leads to a low computation to communication ratio, since fine-grain parallelism implies high communication overhead and less opportunity for performance enhancement. In an extreme scenario, if granularity is too fine, it is possible that the communication overhead between tasks takes longer than the computation. However, successful load balancing is easier to be achieved by following a fine-grain policy. Many methods have been proposed in order to merge consecutive iterations of computation and we refer the reader to [8] and [9]. On the other hand, in the case of coarse-grain parallelism, large amounts of computational work are carried out between communication/synchronization events and consequently the computation to communication ratio is high. Although this fact provides more opportunity for performance increase, it is harder to load balance efficiently. Figure 3.1 illustrates the difference between fine-grain and coarse-grain parallelism. In general, the most efficient granularity is dependent on the algorithm and the execution environment (hardware, run-time system etc).

An equally important issue when building parallel programs especially for the

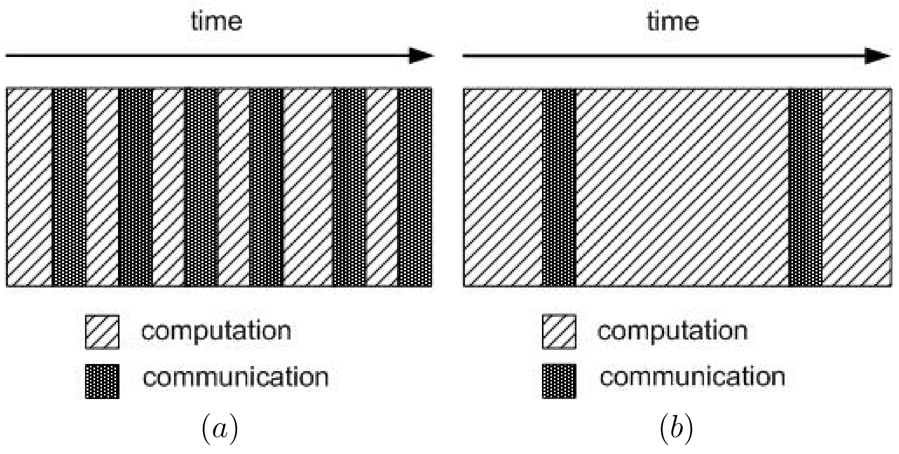


Figure 3.1: (a) Fine-grain parallelism (b) Coarse-grain parallelism

large scale is the scalability of the program, i.e. how the execution behavior alters as more processors are added in the execution. We expect that the pure computational time will decrease linearly as the number of cores increases linearly as well. Furthermore, when the working set assigned to each processor is too small and fits in its cache, then the decrease in the computational time could be super-linear. On the other hand, the communication time of a program depends on the application and the execution platform. For example, applications which are contingent on point-to-point communication pattern possibly have larger communication overhead than parallel programs that exploit collective communication. In addition, when the number of processors increases it is much more difficult to have efficient communication due to contention to network access. In [10], the constraints on the network latency and bandwidth that cause insufficient scalability of three scientific application classes (classical molecular dynamics, cosmological situations and unstructured grid computations) are analyzed.

Let us pick the most favorable scenario, where the communication time decreases as the number of cores increases. Since the number of cores could scale up to hundreds and even millions in modern platforms, the computational time would be reduced drastically. Even if we assume that the parallelism has been implemented in such a way so that the amount of communication is optimal, the time required for each communication and synchronization overhead would be comparable with the tiny computation time, forming a state similar to Figure 3.1(a). This is true in real world, because when even more processors try to communicate using a commodity interconnection network, then communication bottlenecks appear and the communication time is not reduced linearly as computation time does. Such a usual behavior of applications is depicted in Figure 3.2(a). First of all we observe that the computation time halves as the number of cores participating in execution doubles. The communication time (vertical distance of the compact

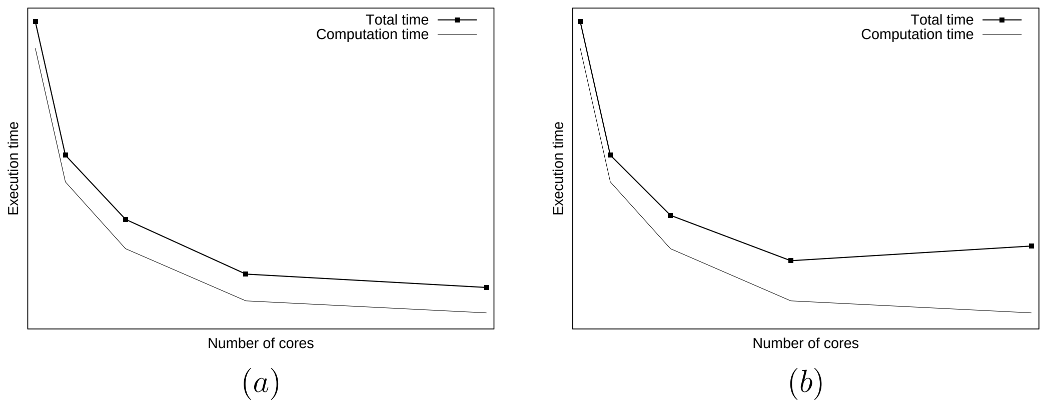


Figure 3.2: Typical behavior of applications as the number of cores increases and the communication time: (a)remains constant, (b)increases.

lines) decreases with a such a slow rate that it could be regarded as constant. Consequently, there is a point that communication and computation time become equal and after that point the communication becomes the dominating factor. Another typical behavior for parallel applications is illustrated in Figure 3.2(b). In this case, the communication time increases as long as we add cores to the execution and thus it becomes rapidly the dominating factor.

After all, we drew the conclusion that communication is the dominating factor in several parallel applications, especially when the number of processors increases significantly. The most common cost model used in algorithm design for large scale multiprocessors assumes that the program alternates between computation and communication phases and that communication requires time linear in the size of the message, plus a start-up cost [11]. This cost model governs Algorithm 3.1 and the total execution time of the parallel program is:

$$T = T_{compute} + T_{communicate} \tag{3.1}$$

and $T_{communicate} = N_c(T_s + L_cT_b)$, where T_s is the start-up cost, T_b is the time per byte-transfer, L_c is the message length, and N_c is the number of communications. To achieve a satisfactory performance, the program should yield a sufficiently high ratio of computation to computation, for example there should be $T_{compute} \geq 9 \cdot T_{communicate}$.

Here comes the idea of overlapping computation with communication ([12], [13]). If we can execute in parallel the communication and the computation part of the algorithm then the execution time would be:

$$T = max(T_{compute} + N_cT_s, N_cL_cT_b) \tag{3.2}$$

Thus, to achieve high processor efficiency, the communication and compute times need only balance, and the compute time needs to overweight the communication

Algorithm 3.1 Simplified model of a parallel algorithm

```
while $work$ ≠done do
    communicate()
    compute()
end while
```

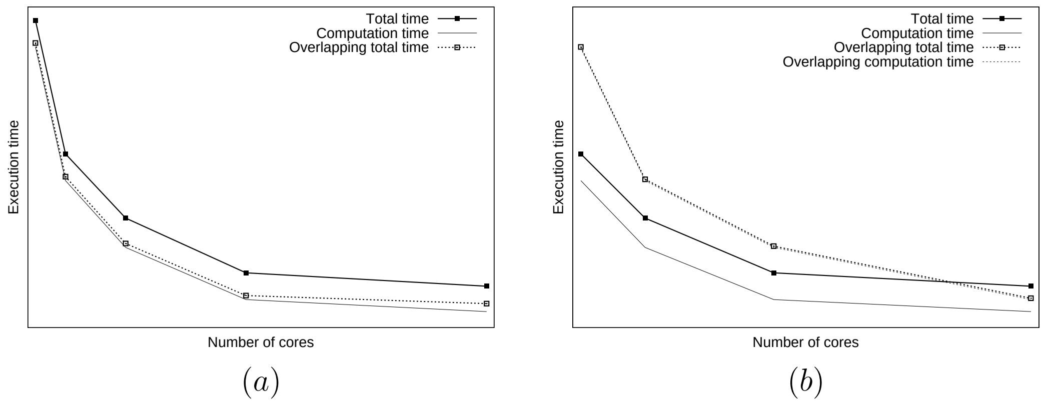


Figure 3.3: Expected behavior of the application after overlapping computation with communication by using: (a)non-blocking functions, (b)helper threads

overhead, i.e. $T_{compute} \gg N_c T_s$. The algorithmic changes that must be made in order to execute in parallel computation with communication are not always straightforward and usually they imply some computational and communication overhead. For the moment, we neglect these issues. In the next section we discuss two alternatives to implement overlapping.

3.2 Implementation of the technique

We consider two distinct approaches to implement applications that overlap computation with communication. The "traditional" one involves non-blocking communication and the "proposed" method uses separate threads for the computation and the communication part of the program.

3.2.1 Utilizing non-blocking communication functions

So far it is clear that blocking communication functions wait for the communication process to complete before continuing to the next program instruction. Therefore, in the Algorithm 3.1 the communication and the computation process are serialized and consequently the expression 3.1 implies the summation of the individual times. However, as we mentioned in section 1.4.2, non-blocking communication functions allow other work to be done while communication is in progress. Assuming that all necessary algorithmic changes have been made in order to overlap the computation with communication, algorithm 3.2 is a version that implements overlapping. We refer the reader to [13], where this method is analyzed extensively.

Algorithm 3.2 Algorithm that overlaps computation with communication by using non-blocking functions

```
while $work \neq$done do
  non-blocking_communicate()
  compute()
end while
```

The expected behavior of the application of Figure 3.2(a) after overlapping computation and computation by using non-blocking functions is illustrated in Figure 3.3(a). We assume that the overlapping is ideal, i.e. for a given number of cores, the total execution time of the overlapping version is approximately $max(computation\ time, communication\ time)$ and consequently it is reduced notably. This "traditional" approach has been widely followed in the case of clusters with uniprocessor nodes and high performance computing interconnects and such a cluster is classic paradigm of late 90's and early 00's. In these clusters, each node has just one physical core so we can not use separate thread to undertake the communication part of the program. Therefore, a high performance interconnection is needed so that the non-blocking functions do not waste time on the uniprocessor and they can be executed in parallel with the computational part.

The non-blocking MPI routines that are available to implement the latter algorithm are namely referred to section 2.3.2. Although Algorithm 3.2 is elegant, there are some implementation reasons that could make such kind of overlapping inefficient. These reasons will be discussed later in the chapter.

3.2.2 Dedicating separate threads

The "proposed" implementation of overlapping computation with communication is to utilize separate threads for the computational part and the communication part (see [14]). Again we assume that all necessary algorithmic changes have been made in order to overlap the computation with communication. Pseudocode 3.1, which uses OpenMP directives, illustrates this different implementation.

Pseudocode 3.1 Implementation of overlapping by using separate OpenMP threads

```
# omp parallel {
   while (work != done) {
      # omp sections {

         # omp section {
            communicate();
         }

         # omp section {
            compute();
         }
      }
   }
}
```

In this case, we suppose that OpenMP has been properly set and there are two available threads in the parallel region. As explained in section 2.3.1, each section is assigned to a separate thread, thus communication and computation are executed simultaneously. The expected behavior of the application is depicted in Figure 3.3(b). Notice that, e.g. on the final state, the overlapping version of the program uses the half cores for computational reasons and the rest cores in order to accomplish communication. Therefore, the computation time at that point equals to the computation time the simple program has when it uses the half cores of the final state. However, since communication is the dominating factor for the simple program, its total execution time is higher than the total time of the program which overlaps computation with communication.

This "proposed" implementation of overlapping fits well to clusters of modern multicore nodes, *possibly* without high performance interconnects. In these clusters there are many cores available in every node and the whole communication part of the program can be assigned to a separate thread. Therefore, in several applications there is not need for a high performance interconnection since the overhead of the communication functions (implied by a commodity interconnection network) is not serialized with the the computation part.

3.2.3 Efficiency issues

Although the first way of implementing overlapping is quite elegant, there are some reasons that could downgrade the performance of this implementation.

First of all, in section 3.1 we implied that overlapping computation with communication demands some overhead. Usually, this overhead involves copies to buffers before and after using the communication routines[1]. Provided we use the non-blocking functions to implement overlapping, these extra computations will be serialized and consequently they could increase the total execution time. On the contrary, if there are separate threads, the communication thread could undertake this overhead.

Another considerable issue has to do with the interconnection network and the protocols used to implement message passing. In subsections 2.2.1 and 2.2.2 we emphasized that the protocols used by Ethernet to support message passing require data copies between the application and the kernel. These copies are time demanding and finally a core takes them up. As long as we do not use helper threading to implement overlapping, then these copies are executed by the core and essentially computation and communication are not completely overlapped (before continuing to the computation the *same* core performs these copies). However, in the case of separate threads, this extra copies are completed in parallel, since the communication thread is responsible for them while the other thread makes computations. This fact may not be of great importance granted that the available interconnection network is Myrinet, but mostly commodity networks like Ethernet are used for the interconnection of nodes in a cluster.

Finally, the communication operation of a program could be a collective one, e.g. a broadcast. As mentioned in subsection 2.3.2 all collective operations are blocking, hence the first approach of implementing overlapping can not be followed when communication is collective. Although implementing the latter method of overlapping seems to be advantageous, we should note that it is more resource consuming as half cores should be used to accomplish communication and just the half cores execute computations.

In the next chapters we examine how overlapping is applied in four applications (Floyd–Warshall, Smith–Waterman, 3D advection equation, 3D heat equation) and we study the behavior of both overlapping approaches.

[1]These copies are called *packing* and *unpacking* respectively

Chapter 4

Applications

In the previous chapter we described the main idea of overlapping computation with communication. Now we investigate the way this technique can be applied on four applications: Floyd-Warshall, Sequence alignment, 3D advection equation and 3D heat equation.

4.1 Finding shortest paths: Floyd–Warshall algorithm

4.1.1 Algorithm

The *Floyd–Warshall algorithm*([15], [16]) is a graph analysis algorithm for finding shortest paths in a weighted graph (with positive or negative edge weights). A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices though it does not return details of the paths themselves. The algorithm is an example of dynamic programming. In our case we will examine graphs with positive edge weights.

The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with $\Theta(|V|^3)$ comparisons in a graph. This is remarkable considering that there may be up to $\Omega(|V|^2)$ edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal.

Consider a graph G with vertices V, each numbered 1 through N. Further consider a function $shortestPath(i, j, k)$ that returns the shortest possible path from i to j using vertices only from the set $1, 2, ..., k$ as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each i to each j using only vertices 1 to $k + 1$. There are two candidates for each of these paths: either the true shortest path only uses vertices in the set $1, ..., k$ or there exists some path that goes from i to $k + 1$, then from $k + 1$ to j that is better. We know that the best path from i to j that only uses vertices 1 through k is defined by $shortestPath(i, j, k)$, and it is clear that if there were a better path from i to $k+1$ to j, then the length of this path would be the concatenation of the shortest path from i to $k+1$ (using vertices in $1, ..., k$) and the shortest path from $k+1$ to j (also using vertices in $1, ..., k$). We can define $shortestPath(i, j, k)$ in terms of the following recursive formula, granted that $w(i, j)$ is the weight of the edge between vertices i and j:

$$shortestPath(i, j, 0) = w(i, j)$$

$$shortestPath(i, j, k) = min \begin{cases} shortestPath(i, j, k-1) \\ shortestPath(i, k, k-1) + shortestPath(k, j, k-1) \end{cases}$$

This formula is the heart of the Floyd–Warshall algorithm. The algorithm works by first computing $shortestPath(i, j, k)$ for all (i, j) pairs for $k = 1$, then $k = 2$, etc. This process continues until $k = n$, and we have found the shortest path for all (i, j) pairs using any intermediate vertices.

4.1.2 Serial implementation

Conveniently, when calculating the $k-th$ case, one can overwrite the information saved from the computation of $k-1$. This means the algorithm uses quadratic memory. Pseudocode 4.1 describes the serial implementation of the algorithm, assuming that there are n vertices and the graph is initialized properly.

Pseudocode 4.1 Serial implementation of Floyd–Warshall algorithm

```
for (k=1 to n)
    for (i=1 to n)
        for (j=1 to n)
            path[i][j]=min(path[i][j], path[i][k]+path[k][j]);
```

4.1.3 Baseline parallel implementation

There are many approaches to parallelize Floyd–Warshall algorithm. In our case we will follow a *domain decomposition* policy, i.e. the data associated with our problem is decomposed and each parallel process then works on a portion of the data.

Specifically, matrix *path* is partitioned in chunks that contain equal number of lines and each process is assigned such a chunk, as it is illustrated at Figure 4.1. All processes must synchronize before moving to a next k iteration, because at the $k+1$ iteration the line $k+1$ of the initial matrix is necessary for the computations and this line *must* be computed up to the $k-th$ iteration. The latter requirement

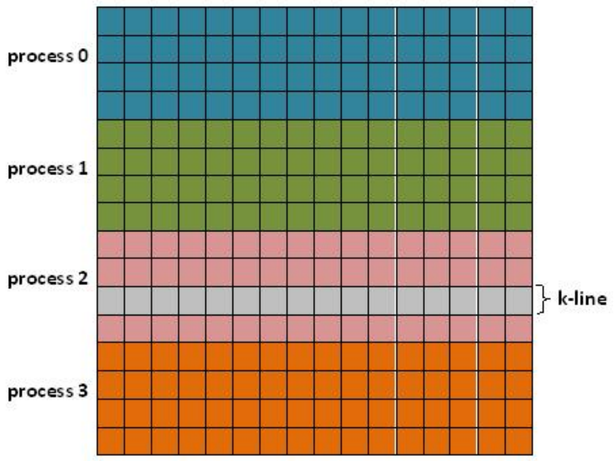


Figure 4.1: Partitioning of the matrix ($n = 16$) into 4 chunks. The gray line indicates the one to be broadcast by process 2 when $k = 11$.

is satisfied by synchronizing the processes at the end of each k iteration. The communication needed between the processes is a broadcast of the $k-th$ line of the initial matrix at each k iteration. This $k-th$ line is crucial in order to make computations because essentially it is the term $path[k][j]$ of Pseudocode 4.1. As it can be shown at Figure 4.1, the owner-process of the $k-th$ line depends on the value of k. Pseudocode 4.2 describes this simple parallel version of the algorithm.

Pseudocode 4.2 Simple parallel implementation of Floyd–Warshall algorithm

```
for (k=1 to n) {

    if (my_rank == sender)
        pack(k-th line to aux_buffer);

    MPI_Bcast(aux_buffer, root=sender);

    for (i=1 to lines_per_chunk)
        for (j=1 to n)
            my_chunk[i][j]=min(my_chunk[i][j], my_chunk[i][k]+
                aux_buffer[j]);

}
```

4.1.4 Overlapping parallel implementation

The key observation that leads to overlapping computation with communication is the following one: When processes execute the $k-th$ iteration, the process that owns line $k+1$ could compute it and broadcast it while in parallel computes its chunk for the $k-th$ iteration. The other processes could receive that $k+1$

line while in parallel compute their corresponding chunk for the $k-th$ iteration. As a result, when it is time to execute the $(k+1)-th$ iteration, all processes already have available the line $k+1$ and therefore can start immediately their computations.

Pseudocode 4.3 Parallel implementation of Floyd–Warshall algorithm that overlaps communication with computation

```

# omp parallel
for (k=1 to n) {

    /* Communication thread */
    # omp section {

        /* Compute the line k+1 */
        if (my_rank == owner_of_line(k+1) {
            for (j=1 to n)
                my_chunk[the_line][j] = min(my_chunk[the_line][j],
                    my_chunk[the_line][k] + aux_buffer[j]);
            for (j=1 to n)
                aux_buffer[j] = my_chunk[the_line][j];
        }

        /* Broadcast the line k+1 */
        MPI_Bcast(aux_buffer, root=owner_of_line(k+1));
    }

    /* Computation thread */
    # omp section {

        for (i=1 to lines_per_chunk)
            for (j=1 to n)
                my_chunk[i][j]=min(my_chunk[i][j], my_chunk[i][k]+
                    aux_buffer[j]);
    }
}
```

Obviously, two separate threads should be dedicated so that the first one is responsible for the communication (and probably for computing one line if that process owns the line $k+1$) and the second one is responsible for the computation of a chunk. Pseudocode 4.3 describes this parallel version of the algorithm that overlaps communication with computation (we do not include the necessary initializations like computation and broadcasting of line 1).

4.2 Sequence alignment: Smith–Waterman algorithm

4.2.1 Algorithm

In bioinformatics, a sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. Aligned sequences of nucleotide or amino acid residues are typically represented as rows within a matrix. Gaps are inserted between the residues so that identical or similar characters are aligned in successive columns. A variety of computational algorithms have been applied to the sequence alignment problem, including slow but formally optimizing methods like dynamic programming.

The *Smith–Waterman algorithm* [17] is a general local alignment method based on dynamic programming. Instead of looking at the total sequence, the Smith-Waterman algorithm compares segments of all possible lengths and optimizes the similarity measure. So the core of this algorithm is the computation of a similarity matrix H. Let us first define some notation:

- a, b : strings over an alphabet Σ
- $m = \text{length}(a)$
- $n = \text{length}(b)$
- if $a_i = b_j$ then $w(a_i, b_j) = w(match)$
- if $a_i \neq b_j$ then $w(a_i, b_j) = w(mismatch)$
- $-$ is the gap and implies deletion or insertion
- $w(a_i, -) = w(-, b_j) = gapscore$
- $H(i, j)$ is the maximum similarity-score between a suffix of $a[1...i]$ and a suffix of $b[1...j]$

The similarity matrix H is built from the following recursive expression:

$$H(i,0) = 0, \ 0 \leq i \leq m$$

$$H(0,j) = 0, \ 0 \leq j \leq n$$

$$H(i,j) = max \begin{cases} 0 & \\ H(i-1,j-1) + w(a_i,b_j) & Match/Mismatch \\ H(i-1,j) + w(a_i,-) & Deletion \\ H(i,j-1) + w(-,b_j) & Insertion \end{cases}, \ 1 \leq i \leq m, 1 \leq j \leq n$$

Smith-Waterman algorithm is fairly demanding of time and memory resources: in order to align two sequences of lengths m and n, $O(mn)$ time and space are required.

4.2.2 Serial implementation

The serial implementation of the algorithm is quite straightforward and is presented in Pseudocode 4.4.

Pseudocode 4.4 Serial implementation of Smith-Waterman algorithm

```
for (i=1 to m)
    for (j=1 to n)
        H[i][j] = max(0, H[i-1][j-1] + w[a[i]][b[j]]), H[i-1][j] +
              GAP, H[i][j-1] + GAP);
```

4.2.3 Baseline parallel implementation

In order to parallelize the Smith-Waterman algorithm we follow again domain decomposition policy, i.e. each process will be assigned to compute a chunk of the similarity matrix H. Such a partition of the similarity matrix H is illustrated on Figure 4.2. The parallelization is not obvious yet, since every process (except one) in order to compute the elements of its first column needs elements that do not belong to it. To be more accurate, as Pseudocode 4.4 indicates, the computation of $H[i][j]$ demands among others the elements $H[i-1][j-1]$ and $H[i][j-1]$. In case that $H[i][j]$ is element of the first column in a chunk, then $H[i-1][j-1]$ and $H[i][j-1]$ are elements of the last column in a different chunk.

Now it is clear that every process (except one) has to send the last column of its chunk to the "right neighbor". However, if every process waits its "left neighbor" to compute its whole chunk and then receives the wanted column, the computations of the processes are serialized and consequently there is no parallelism. Instead, each process can compute a line of its chunk and then send the last element of that line to the "right neighbor". After that, the process can receive from the "left neighbor" the element needed to compute its next line and proceed to the computation of the new line. Despite the fact that this scheduling enables parallelism it is fine-grained and hence it is inefficient as described in section 3.1. Here comes the idea of *tiling* [18], a method of aggregating a number of loop iterations into tiles where the tiles execute atomically, i.e. a processor executing the iterations belonging to a tile receives all the data it needs before executing any one of the iterations in the tile, executes all the iterations in the tile and then sends the data needed by other processors. So, every chunk is further partitioned in parts that each contains the equal (if possible) number of lines. Let us name these parts *tiles*. So, the most left process can compute a tile of its chunk, then send the last column of that tile to

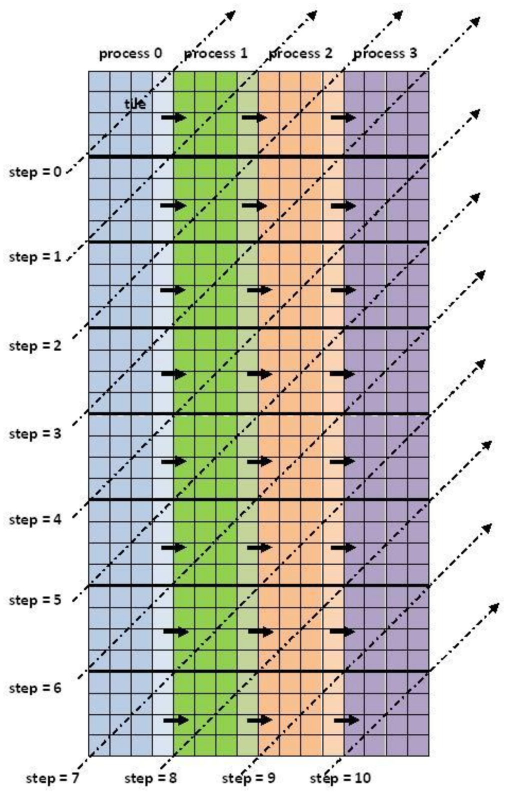


Figure 4.2: Parallel implementation of Smith-Waterman algorithm. Each color represents a chunk of the similarity matrix H and is assigned to a different process. Horizontal bold lines indicate the borders of the *tiles*. The pale columns are the data to be transferred between processes and the bold horizontal arrows demonstrate the flow of data. Finally, the diagonal dashed arrows indicate the step at which each tile is computed.

its "right neighbor", and then proceed with the next tile. Now the "right neighbor" can receive the needed column and continue with the computation of its tile. After computing that tile, the "right neighbor" can send the last column of that tile to its corresponding right process and then receive the next needed column by its left process. Note that the latter column has been computed by the left process as long as the "right neighbor" was computing its tile. This scheduling remains active until all tiles are computed and eventually there is coarse-grain parallelism.

For the moment, assume that there is a large number of tiles at every chunk and that p processes are available. If we define as "step" the computation time of each tile plus the communication time for a single tile-column send, then after $p-1$ steps all the processes will execute in parallel. This parallel scheme is explained in detail on Figure 4.2. In this example, the two sequences have length $m = 32$ and $n = 16$, while the number of processes is $p = 4$. The tile factor here is 4, hence each chunk consists of eight 4×4 tiles. As previously mentioned, after three steps all processes run in parallel until step 7. From steps 8 to 10, one more process becomes idle per step and eventually after step 10 all tiles are computed. Pseudocode 4.5 describes this parallel version of the Smith-Waterman algorithm.

Pseudocode 4.5 Simple parallel implementation of Smith-Waterman algorithm

```
for (i=1; i<=m; i+=tile_factor) {

   /* Receive column necessary for computation of the tile */
   if (my_rank!=most_left_process) {
      MPI_Recv(recv_column, sender=left_neighbor);
      unpack(recv_column);
   }

   /* Compute a tile */
   for (ii=i; ii<i+tile_factor; ii++)
      for (j=1; j<=n; j++)
         H[ii][j] = max(0, H[ii-1][j-1] + w[a[ii]][b[j]]),
                           H[ii-1][j] + GAP, H[ii][j-1] + GAP);

   /* Send column to right neighbor */
   if (my_rank!=most_right_process) {
      pack(send_column);
      MPI_Send(send_column, receiver=right_neighbor);
   }
}
```

4.2.4 Overlapping parallel implementation

Now we would like to modify the previous algorithm in order to overlap computation with communication. Therefore, we defer the first computation of each process (except the most left process) for one more step. Instead, these processes receive a second column. This slight adjustment provides this opportunity: While at step t a process is computing a specific tile, simultaneously this process can: (a) receive the column needed for the computation of the next tile at step $t + 1$, and (b) send the column of the tile computed at the previous step $t - 1$. Finally, some actions binary to the initialization operations have to be performed before ending the algorithm. Pseudocode 4.6 describes this parallel version of the Smith-Waterman algorithm that overlaps computation with communication by using separate threads. Similar is the implementation that overlaps computation with communication by utilizing non-blocking communication routines. The differences are that: (a) all *omp* directives are removed, (b) all communication functions are replaced with their corresponding non-blocking version and (c) the *unpack* function inside the main for-loop must be moved after the *compute* function. Of course, appropriate calls of *MPI_Wait* function should be added to ensure that non-blocking communications have been completed.

Pseudocode 4.6 Parallel implementation of Smith-Waterman algorithm that overlaps computation with communication

```
/* first receive */
MPI_Recv(step_1);
unpack(step_1);

/* second receive */
MPI_Recv(step_2);
unpack(step_2);

/* first processing */
Compute(step_1);

# omp parallel
for (step=2 to total_steps-1 ) {

    # omp section {

        /* Receive column for computation of the NEXT tile */
        MPI_Recv(step+1);
        unpack(step+1);

        /* Send column of PREVIOUS tile to right neighbor */
        pack(step-1);
        MPI_Send(step-1);
    }

    # omp section {

        /* Compute tile at CURRENT step */
        Compute(step);
    }

}

/* Send column of semi-final step */
pack(semi-final step);
MPI_Send(semi-final step);

/* Compute the final tile */
Compute(final tile);

/* Send column of final step */
pack(final step);
MPI_Send(final step);
```

4.3 Advection 3D

4.3.1 Algorithm

Advection, in chemistry, engineering and earth sciences, is a transport mechanism of a substance, or a conserved property, by a fluid, due to the fluid's bulk motion in a particular direction. An example of advection is the transport of pollutants or silt in a river. The motion of the water carries these impurities downstream. Another commonly advected property is energy or enthalpy, and here the fluid may be water, air, or any other thermal energy-containing fluid material. Any substance, or conserved property (such as enthalpy) can be advected, in a similar way, in any fluid.

The fluid motion in advection is described mathematically as a vector field, and the material transported is typically described as a scalar concentration of substance, which is contained in the fluid. The *advection equation* [19] is the partial differential equation that governs the motion of a conserved scalar as it is advected by a known velocity field. It is derived using the scalar's conservation law, together with Gauss's theorem, and taking the infinitesimal limit. The advection equation for a scalar u, such as temperature, is expressed mathematically as:

$$\frac{\partial u}{\partial t} + \nabla(u\mathbf{v}) = 0 \tag{4.1}$$

where ∇ is the divergence operator and $\mathbf{v}$ is the velocity vector field. Frequently, it is assumed that the flow is incompressible, that is, the velocity field satisfies $\nabla\mathbf{v} = 0$ (it is said to be solenoidal). If this is so, the above equation reduces to:

$$\frac{\partial u}{\partial t} + \mathbf{v} \cdot \nabla u = 0 \tag{4.2}$$

Furthermore, if we regard speed to be constant, i.e. $\mathbf{v} = (-1, -1, -1)$, then the advection equation becomes:

$$\frac{\partial u}{\partial t} = \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} + \frac{\partial u}{\partial z} \tag{4.3}$$

To solve this equation numerically, we approximate u to a discrete solution defined in a cubic grid. For simplicity, if we use the explicit Euler discretization for the partial derivatives and if we ignore all multiplying factors that come into play, then the numerical equation is:

$$\begin{aligned} u(t,x,y,z) &= u(t-1,x,y,z) + u(t-1,x-1,y,z) + \\ &\quad u(t-1,x,y-1,z) + u(t-1,x,y,z-1) \end{aligned} \tag{4.4}$$

4.3.2 Serial implementation

So far our goal is to compute the values of scalar u after time T in all three dimensions. Conveniently, after calculating scalar u for the $t-th$ moment, one can overwrite this information to the values of u for the $(t-1)th$ moment, since they are not needed any more. Assuming that the limits of our 3D space are: $1 \leq x \leq X$, $1 \leq y \leq Y$ and $1 \leq z \leq Z$, Pseudocode 4.7 describes a serial implementation of the algorithm.

Pseudocode 4.7 Serial implementation of 3D advection algorithm

```
current=1;
previous=0;
for (t = 0; t < T - 1; t++) {
    for (x = 1; x <= X; x++)
        for (y = 1; y <= Y; y++)
            for (z = 1; z <= Z; z++)
                u[current][x][y][z] =
                     u[previous][x][y][z] +
                     u[previous][x-1][y][z] +
                     u[previous][x][y-1][z] +
                     u[previous][x][y][z-1];

    swap(current,previous);
}
```

4.3.3 Baseline parallel implementation

In order to parallelize the serial algorithm we will follow a domain decomposition policy, i.e. each process will be assigned to compute a chunk of the 3D space until time T. Figure 4.3 shows the advection problem in two spatial dimensions plus one time dimension in order to simplify the illustration. This state easily can be generalized to three spatial dimensions plus one time dimension. As it can be concluded from equation 4.4, a process in order to compute its chunk for a moment t needs: (a) elements from the moment $t-1$ that are already possessed by the process, and (b) elements from the moment $t-1$ that do not belong to this process, and specifically these elements are required in order to compute boundary elements of the chunk. It is clear that these dependencies determine which elements have to be sent between processes. Figure 4.4 illustrates the communication pattern for the advection problem in two spatial dimensions plus one time dimension.

The previous description implies fine grain parallelism and this fact brings usually drawbacks, as discussed in section 3.1. Another modification that can be done is to implement the idea of *tiling* to the time dimension in order to implement a coarse-grain parallel program. So every process computes k moments (k here is the tile factor) before communicating with its neighbors. As a result, each process

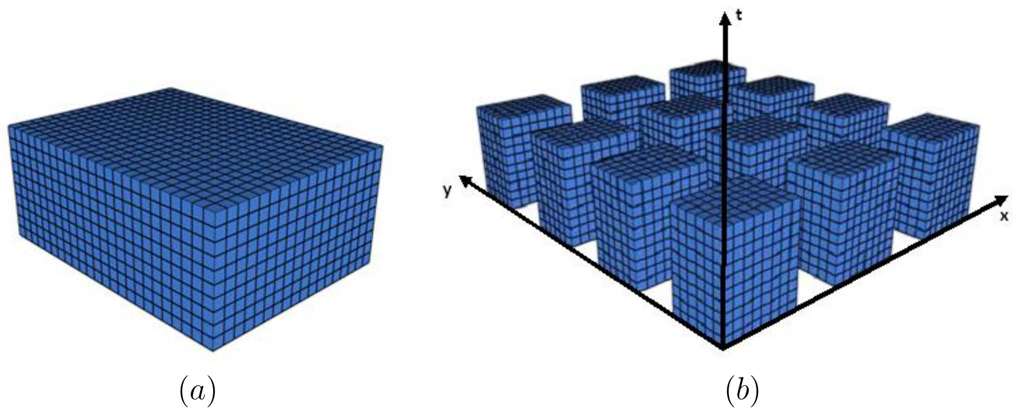


Figure 4.3: (*a*)Advection problem in two spatial dimensions plus a time dimension (*b*)Partitioning of the advection problem to 12 chunks. Each process will be assigned such a chunk.

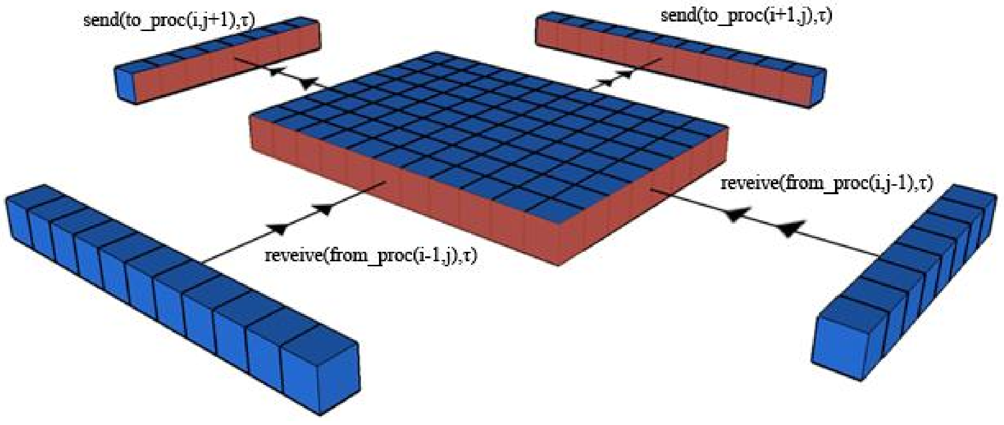


Figure 4.4: Communication pattern for the advection problem in two spatial dimensions plus a time dimension

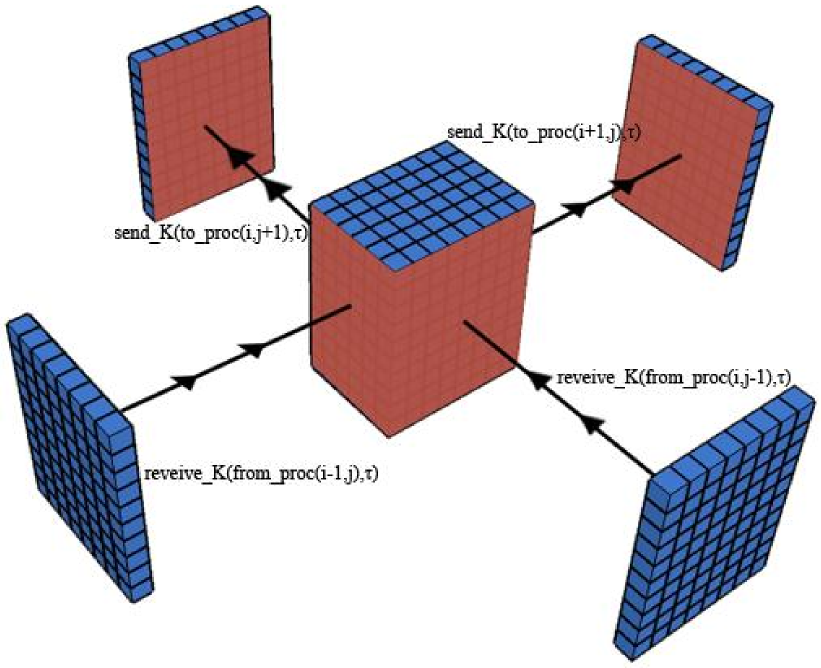


Figure 4.5: Communication pattern for the advection problem in two spatial dimensions plus a time dimension when tiling is applied (tile factor = 10)

sends and receives per iteration more data than it would if just one moment had been computed, but the total amount of data transferred is eventually the same. Figure 4.5 illustrates the modified communication pattern for $k = 10$. The flow of the computation begins from the process that lies on the start of the axis and extends radially along the three axes. Obviously there is an initial delay until each process receives the elements needed to start computations (remind Figure 4.2 where again each process waits for some elements in order to be able to compute). However, this delay is negligible when the chunks are large enough. Pseudocode 4.8 describes this parallel version of the algorithm that uses tiling.

Pseudocode 4.8 Parallel implementation of 3D advection algorithm that uses tiling

```
Initialize_3D(tile factor = k);

for (t = 0; t < T; t += k) {

    /* Receive data */
    Receive_from_neighbors(k, (x-1,y,z), (x,y-1,z),(x,y,z-1));
    Unpack_data_received();

    /* Compute data */
    for (tt = 0; tt < k; tt++)
        for (x = 1; x <= my_chunk_X; x++)
            for (y = 1; y <= my_chunk_Y; y++)
                for (z = 1; z <= my_chunk_Z; z++) {
                    u[tt+1][x][y][z] =
                            u[tt][x][y][z] +
                            u[tt][x-1][y][z] +
                            u[tt][x][y-1][z] +
                            u[tt][x][y][z-1];
                }

    /* Copy the data of moment k to moment 0, so that they are
    ready for the next iteration */
    copy_data(u[k][x][y][z] to u[0][x][y][z]);

    /* Send data */
    Pack_data_to_send();
    Send_to_neighbors(k, (x+1,y,z), (x,y+1,z), (x,y,z+1));
}
```

4.3.4 Overlapping parallel implementation

In order to overlap computation with communication we use again the same idea as in subsection 4.2.4 and it is extensively discussed in [22]. So, initially processes receive in advance one more piece of data needed for next computation. After this modification the scheduling becomes: While at time t a process is computing a specific tile, simultaneously this process can: (a) receive the data required for computation at time $t+1$, and (b) send data computed at previous moment $t-1$. Finally, some actions binary to the initialization operations have to be performed before ending the algorithm. This scheduling is shown on Figure 4.6 for an example of two spatial dimensions and a time dimension. Pseudocode 4.9 describes this parallel version of the 3D advection algorithm that overlaps computation with communication by using separate threads. Almost identical is the implementation that overlaps computation with communication by utilizing non-blocking commu-

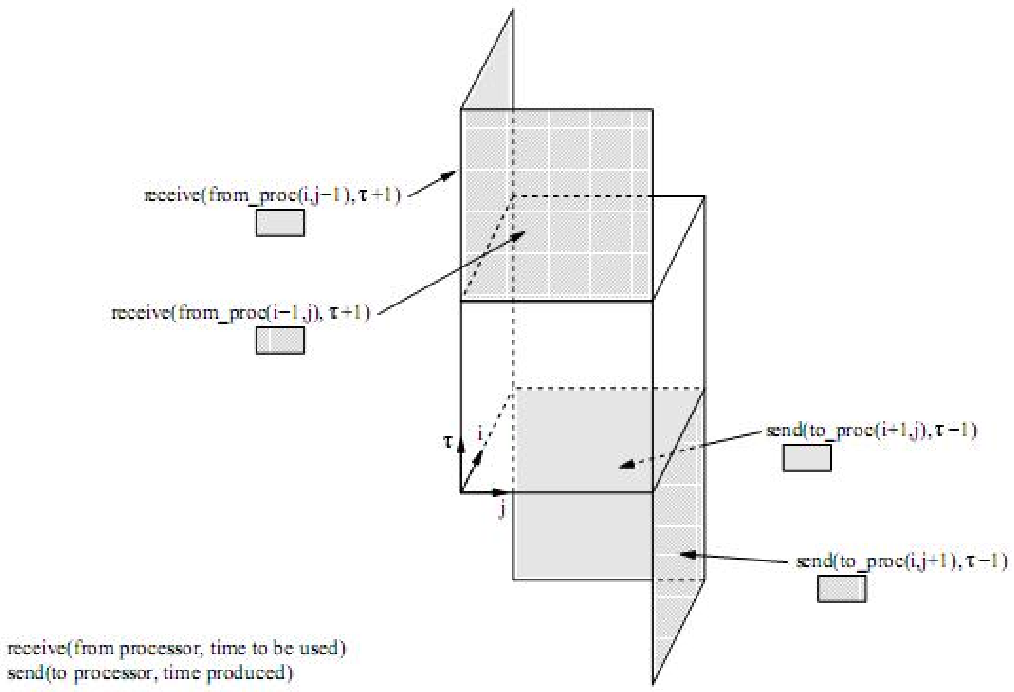

receive(from_proc(i,j−1), τ+1)
receive(from_proc(i−1,j), τ+1)
τ
i
j
send(to_proc(i+1,j), τ−1)
send(to_proc(i,j+1), τ−1)
receive(from processor, time to be used)
send(to processor, time produced)

Figure 4.6: Scheduling for the advection problem in two spatial dimensions plus a time dimension, when computation and communication are overlapped

nication routines. The differences are that: (a) all *omp* directives are removed and (b) all communication functions are replaced with their corresponding non-blocking version. Of course, appropriate calls of *MPI_Wait* function should be added to ensure that non-blocking communications have been completed.

Pseudocode 4.9 Parallel implementation of 3D advection algorithm that overlaps computation with communication

```
Initialize_3D(tile factor = k);

/* First receive */
Receive(time=1);
Unpack_data(time=1);

/* First compute */
Compute_tile(time=1);

/* Second receive */
Receive(time=2);

# omp parallel
for (step = k ; step < T-k; step += k) {

    /* Unpack data received and pack data to be sent */
    # omp single {
        Unpack_data();
        Pack_data();
    }

    # omp barrier

    /* Communication thread */
    # omp section {
        Send(data of time t-1);
        Receive(data of time t+1);
    }

    /* Computation thread */
    # omp section {
        Compute_tile(computation of time t);
    }
}

/* Semifinal send */
Pack_data();
Send();

/* Final compute */
Unpack_data();
Compute_tile(final tile);

/* Final send */
Pack_data();
Send();
```

4.4 Heat equation 3D

4.4.1 Algorithm

The heat equation is an important partial differential equation which describes the distribution of heat (or variation in temperature) in a given region over time. For a function $u(t, x, y, z)$ of three spatial variables (x, y, z) and the time variable t, the heat equation is:

$$\frac{\partial u}{\partial t} = a(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}) \tag{4.5}$$

One way to numerically solve this equation is to approximate all the derivatives by finite differences [20]. We partition the domain in space using a mesh $x = 1, ..., X$, $y = 1, ..., Y$, $z = 1, ..., Z$, and in time using a discretization $t = 0, ..., T$. Using a forward difference at time t, a second-order central difference for the space derivatives and ignoring all multiplying factors, equation 4.5 becomes:

$$\begin{aligned} u(t, x, y, z) \quad = \quad & 1/7 \cdot (u(t-1, x, y, z) + \\ & u(t-1, x-1, y, z) + u(t-1, x+1, y, z) + \\ & u(t-1, x, y-1, z) + u(t-1, x, y+1, z) + \\ & u(t-1, x, y, z-1) + u(t-1, x, y, z+1)) \end{aligned} \tag{4.6}$$

4.4.2 Serial implementation

Having in mind the numerical equation 4.6, we would like to compute the values of scalar u after time T in all three dimensions. Conveniently, after calculating scalar u for the $t-th$ moment, one can overwrite this information to the values of u for the $(t-1)th$ moment, since they are not needed any more. Assuming that the limits of our 3D space are: $1 \leq x \leq X$, $1 \leq y \leq Y$ and $1 \leq z \leq Z$, Pseudocode 4.10 describes a serial implementation of the algorithm.

Pseudocode 4.10 Serial implementation of 3D heat equation

```
current=1;
previous=0;
for (t = 0; t < T - 1; t++) {
    for (x = 1; x <= X; x++)
        for (y = 1; y <= Y; y++)
            for (z = 1; z <= Z; z++)
                u[current][x][y][z] =1/7 * (
                     u[previous][x][y][z] +
                     u[previous][x-1][y][z] +
                     u[previous][x+1][y][z] +
                     u[previous][x][y-1][z] +
                     u[previous][x][y+1][z] +
                     u[previous][x][y][z-1] +
                     u[previous][x][y][z+1] );

    swap(current,previous);
}
```

4.4.3 Baseline parallel implementation

We will adopt a domain decomposition policy to parallelize the serial algorithm, i.e. each process will be assigned to compute a chunk of the 3D space until time T. Figure 4.7 shows the heat equation problem in two spatial dimensions in order to simplify the illustration. This problem easily can be generalized to three spatial dimensions. Equation 4.6 implies that a process in order to compute its portion for a moment t needs: (a) some elements from the moment $t-1$ that are already owned by the process, and (b) some elements from the moment $t-1$ that belong to *all* other neighboring processes, and specifically these elements are compulsory to compute boundaries of the portion. The crucial difference from 3D advection equation is that in this case, there exist dependencies with data of *all* neighboring processes. Figure 4.7 illustrates these dependencies for the heat equation problem in two spatial dimensions.

After the previous explanations, the parallel algorithm is now quite clear: every process sends its boundary elements computed at time $t-1$ to the corresponding neighbors and receives the corresponding boundary elements computed at time $t-1$ from its neighbors. Then, every process continues with the computation of its chunk at time t and the whole procedure is repeated until T iterations are made. Pseudocode 4.11 describes this parallel version of the heat equation.

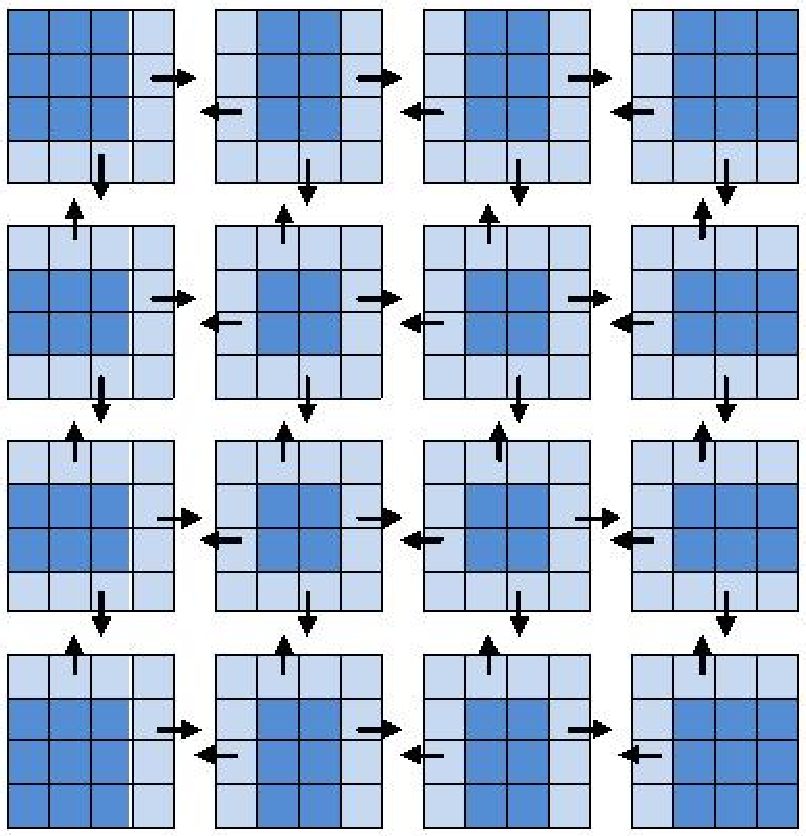


Figure 4.7: Parallel implementation of heat equation in two spatial dimensions. The spatial grid is a 16×16 square grid and is partitioned into 16 chunks. Each chunk is assigned to a different process. Pale squares indicate the data that must be exchanged between the processes. The bold arrows illustrate the flow of the exchanged data.

Pseudocode 4.11 Parallel implementation of 3D heat equation

```
current=1;
previous=0;
for (t = 0; t < T - 1; t++) {

    /* Send and receive data */
    Pack_data(boundaries of t-1);
    ISend(boundaries of t-1 to neighbors);
    IRecv(boundaries of t-1 from neighbors);
    Wait_communication();
    Unpack_data(boundaries of t-1);

    /* Computation of time t */
    for (x = 1; x <= my_X; x++)
        for (y = 1; y <= my_Y; y++)
            for (z = 1; z <= my_Z; z++)
                u[current][x][y][z] =1/7 * (
                    u[previous][x][y][z] +
                    u[previous][x-1][y][z] +
                    u[previous][x+1][y][z] +
                    u[previous][x][y-1][z] +
                    u[previous][x][y+1][z] +
                    u[previous][x][y][z-1] +
                    u[previous][x][y][z+1] );

    swap(current,previous);
}
```

4.4.4 Overlapping parallel implementation

Since data dependencies in heat equation are quite complex and involve every neighbor of each process, a non-trivial modification must be done aiming to overlap computation with communication. Let us remind Figure 4.7 to explain this adjustment with an example. In this scheduling that allows overlapping, a process computes the *bold* elements of its chunk for the moment t and *in parallel* receives and sends the respective boundaries computed at time $t-1$. This parallelism is valid because the computation of *bold* elements does not demand data from neighboring processes. Finally, the computation and the communication thread are synchronized and the process is able to compute its boundaries for moment t, since now it owns all needed elements from neighbors. Obviously this last part of computation is not overlapped at all. The overlapping scheme is analogous for the heat equation at three spatial dimensions. Pseudocode 4.12 describes the parallel version of the 3D heat equation that overlaps computation with communication by using separate threads. Similar is the implementation that overlaps computation with communication by utilizing non-blocking communication routines. The differences are that: (a) all *omp* directives are removed and (b) the *Wait_communication()* and *Unpack_data()* functions inside the main for-loop must be moved *before* the *Compute_boundaries_of_chunk()* function.

Pseudocode 4.12 Parallel implementation of 3D heat equation that overlaps computation with communication

```
current=1;
previous=0;
# omp  parallel
for (t = 0; t < T - 1; t++) {

   # omp section {
      /* Send and receive data */
      Pack_data(boundaries of t-1);
      ISend(boundaries of t-1 to neighbors);
      IRecv(boundaries of t-1 from neighbors);
      Wait_communication();
      Unpack_data(boundaries of t-1);
   }

   # omp section {
      /* Computation of "central elements" for time t */
      for (x = 2; x <= my_X-1; x++)
         for (y = 2; y <= my_Y-1; y++)
            for (z = 2; z <= my_Z-1; z++)
               u[current][x][y][z] =1/7 * (
                   u[previous][x][y][z] +
                   u[previous][x-1][y][z] +
                   u[previous][x+1][y][z] +
                   u[previous][x][y-1][z] +
                   u[previous][x][y+1][z] +
                   u[previous][x][y][z-1] +
                   u[previous][x][y][z+1] );
   }

   # omp single {
      /* Computation of boundaries for time t */
      Compute_boundaries_of_chunk();
   }

   swap(current,previous);
}
```

4.5 Implementation issues

In these section we discuss several implementation issues that affect the performance of the parallel applications.

4.5.1 Assigning work to specific threads

In subsection 2.3.1 we mentioned that an alternative for work sharing at OpenMP is to assign separate code segments to different threads according to their thread-id in an SPMD (Single Process, Multiple Data) style. This option has been followed in order to define the code for the computation and the communication thread, despite the fact that all pseudocodes presented in this chapter imply the usage of the *sections* construct. By this way a thread is enforced to execute either the communication or the computation code for *all* iterations of an execution. If we used the *sections* construct, thread 0 could, e.g., execute the communication section for an iteration and at a next iteration it could run the computation section. This interchange of code sections among threads must be avoided because it restricts re-utilization of level one cache (keep in mind that each core has its own level one data and instruction cache). Initial experiments showed this performance behavior. Moreover, now we *must* put explicit *omp barriers* after the *if* statement that differentiates communication and computation code, so that thread synchronization is accomplished (a *sections* construct imposes thread synchronization).

4.5.2 Binding threads to cores

The ability of the scheduler of an operating system to bind a process (or thread) to a specific core is called CPU affinity. In the Linux operating system, *sched_affinity* is a system call that gives to programmers the aforementioned opportunity. This system call has as arguments the id of a process and a affinity mask and determines on which cores this process can run according to the given affinity mask.

Although it is not clear with a first sight, we desire the communication and the computation thread to run on cores sharing the same level two cache in order to access shared data with trivial latency. Granted that communication thread except communication executes packing and unpacking of data as well, it accesses data used by the computation thread. Thus, at the preliminaries of each program, we use *sched_affinity* system call to ensure that computation and communication threads of a specific MPI process run on cores sharing the *same* level two cache.

4.5.3 Minimizing OpenMP overhead

Creating and reclaiming threads come along with additional OpenMP overhead. First of all, it is pivotal for our application to create these threads just once and reuse them during the iterations of an execution, therefore we should move the *omp parallel* directive before the main *for* loop of the program. Another important issue that demands thorough consideration is to declare the most suitable type for a variable used in *parallel* construct. Obviously some variables *must* be private to each thread in order to have valid execution of the program while other variables *must* be shared between threads. Finally, there are variables whose type is indifferent from a semantics view. However, as mentioned in subsection 2.3.1, a thread has its local copy for a private variable, so multiple accesses to this variable could have better performance than multiple reads if the variable was shared. Hence, the type of these variables is meticulously chosen at every application.

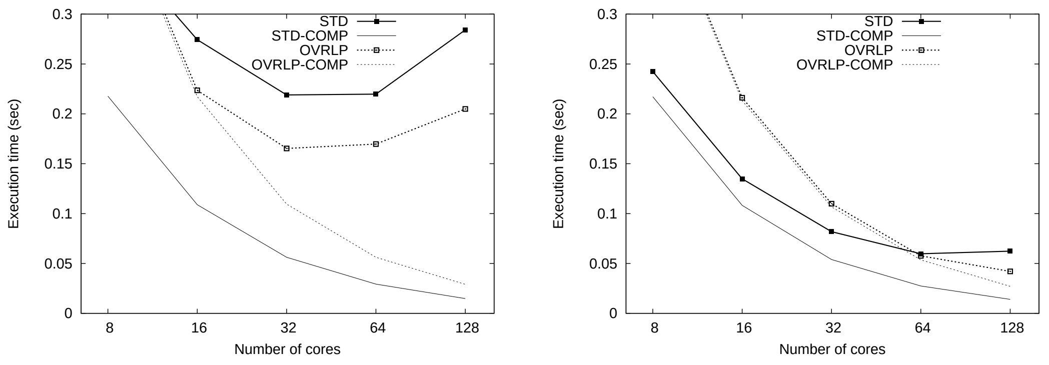
Chapter 5

Experiments

5.1 Floyd-Warshall

In this section we present the experimental results of Floyd-Warshall implementations and we will try to interpret the behavior of the application. In all Figures that will appear, compact lines refer to the simple parallel implementation while dashed lines correspond to the implementation that overlaps computation with communication. Moreover, the lines having “squares” at distinct points, represent the total execution time of the application at that points and the lines that do *not* have “squares” stand for the computation time of the applications. The vertical distance of compact lines at a point is the communication time of the simple parallel application as equation 3.1 indicates. If we recall equation 3.2, we conclude that such a fact is not true for the dashed lines. Let n the number of nodes of the graph. We will study three cases depending on n ($n = 1024$, $n = 2048$ and $n = 4096$) and for each case we use as interconnection network first Gigabit Ethernet and then Myrinet. For each of the previous cases we used 8, 16, 32, 64 and 128 cores. Here we must emphasize that in the case of the parallel programs with overlapping feature, half cores execute computations and half communications. For instance, when we exam the points of the dashed curves that correspond to 128 cores, we should have in mind that 64 cores are dedicated to computations and the rest 64 cores to communication.

In Figure 5.1 we can see the case where $n = 1024$. First of all, we conclude that the scalability is poor when Ethernet is used as interconnection network. Specifically, the best performance for the simple implementation is appeared when 32 cores are used. When the number of cores increases to 64 and 128, despite the fact that computation time continues to halve as expected, communication time enhances dramatically and subsequently total execution time is even larger. The implementation that overlaps communication with computation displays its best performance when 32 cores are used (16 for computations and 16 for communication) and it is 25% lower than the best time of the simple im-



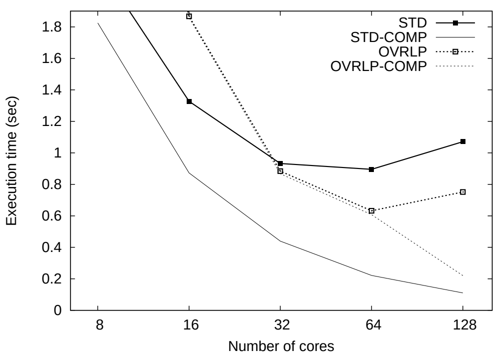
$(a) Ethernet$ $(b) Myrinet$

Figure 5.1: Floyd-Warshall results for a graph with 1024 nodes

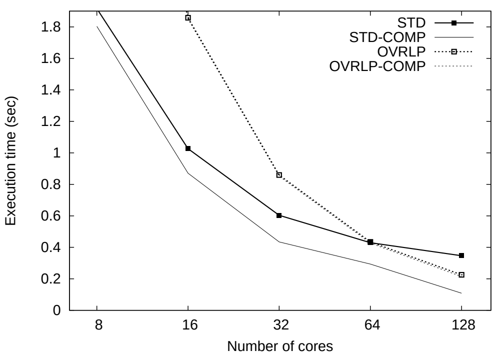
plementation. This is achieved because the total time is now "approximately" $max(computation\ time, communication\ time)$, where the latter times refer to the simple implementation and the case of 16 cores. We mentioned the word "approximately" because now the communication time could be slightly larger due to various implementation overheads. The case of the overlapping version does not scale further than 32 processors, because the communication time becomes so significant that even when computation and communication are overlapped, the maximum of the corresponding times remains huge.

The behaviors of both implementations change when Myrinet comes into play. As described in subsection 2.2.2, Myrinet shows up much better performance than Ethernet and this has impact to the communication time of the application. So, the simple implementation scales up to 64 cores, where it exhibits the best performance. When the number of cores doubles, the total time is approximately constant since the communication time increases and is almost 3,5 times larger than computation time. This behavior can be explained because in this state (128 cores) all the sources of the cluster are utilized, i.e. many cores try to use the interconnection network and even the communication among cores in a specific SMP node is problematic. On the other hand, the implementation that overlaps computation with communication continues to scale up to 128 cores. The overall state here reminds the ideal behavior one would expect from overlapping and essentially it improves the performance of the simple implementation by 30%.

Figure 5.2 illustrates the case where $n = 2048$. Here the situation is similar to the previous one. First we analyze the situation where Ethernet is used. The simple implementation scales up to 32 cores and as we can see, at 64 cores the total execution time is almost constant and then (128 cores) it increases. On the other hand, the implementation that uses overlapping scales up to 64 cores and improves the performance of the simple program by 29%. When the number of

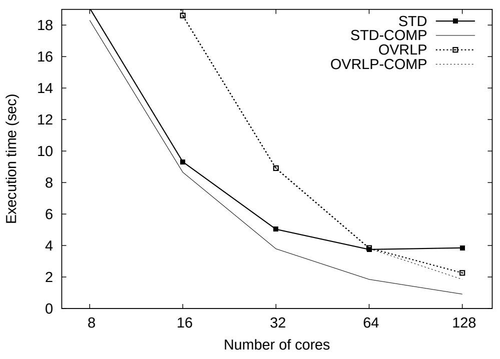


$(a) Ethernet$

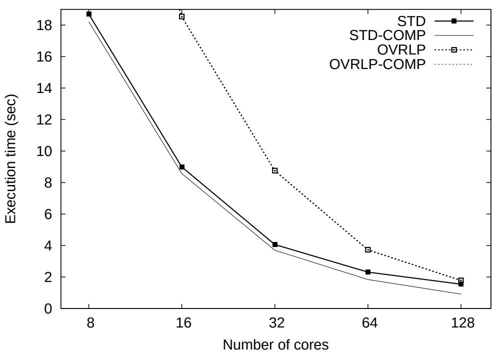


$(b) Myrinet$

Figure 5.2: Floyd-Warshall results for a graph with 2048 nodes



$(a) Ethernet$



$(b) Myrinet$

Figure 5.3: Floyd-Warshall results for a graph with 4096 nodes

cores goes to 128, communication time again becomes so large that overlapping is meaningless.

However, the scene changes when Myrinet is the interconnection network of the cluster. We observe that both implementations scale up to 128 cores, however the simple program shows very poor scalability from 64 to 128 and this image resembles to Figure 3.2, where we relied to explain the motivation for overlapping computation with communication. Indeed, the overlapping implementation continues to scale adequately, and offers an improvement by 35% (in comparison with the best performance of the simple implementation).

Lastly, Figure 5.3 presents the rest experimental results where $n = 4096$. When Ethernet is used, the naive implementation has moderate scalability, this is to say that from 64 to 128 cores the total time does not reduce because communication

Number of nodes	Percentage performance improvement	
	Ethernet	Myrinet
1024	25%	30%
2048	29%	35%
4096	40%	-16%

Table 5.1: Percentage performance improvement of Floyd-Warshall parallel implementation when overlapping is applied. A negative amount indicates how much "slower" is the overlapping version.

becomes the dominating factor. When we bring into play the overlapping scheme, the application scales even further and eventually improves the performance by 40%.

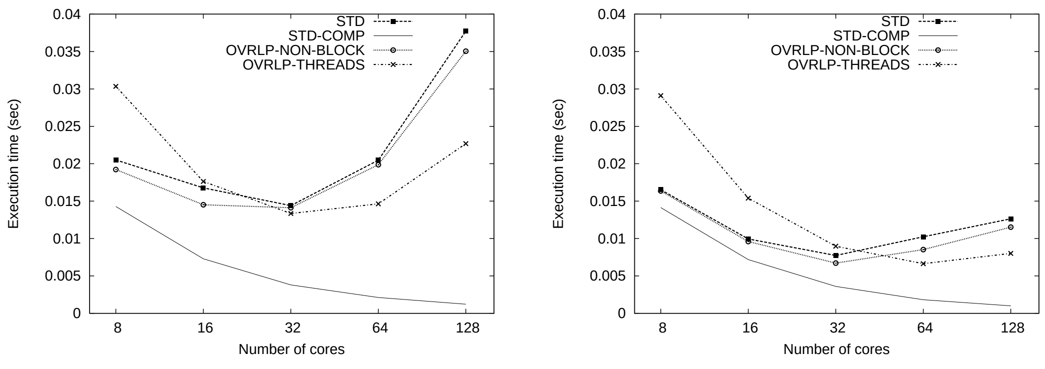
On the other hand, when Myrinet is reclaimed, the simple parallel application scales up to 128 cores. However, the most effective situation (which is the one where 128 cores participate in execution) seems to be very critical as communication time is 41% of total time. This means that provided we used 256 cores, if communication time remained the same and if the computation time decreased by 50%, then (at this hypothetical state) communication time would be 58% of total time.

The overlapping version however is a promising one as it scales ideally up to 128 cores. Although this implementation does not improve the performance of the naive version, it is worth saying that its total time is approximately the computation time of the naive version that uses the *same* number of computational cores. This fact was expected since at this input size ($n = 4096$), computation is the dominating factor. Having this in mind, we conclude that if we had totally 256 cores available, then the total time of the overlapping version would be the computation time of the naive version when 128 cores are utilized. Combining this thought with the conclusion of the previous paragraph, we can assume that at 256 cores the overlapping version would be remarkably faster than the simple parallel version.

In table 5.1 we summarize the impact of overlapping computation with communication at Floyd-Warshall parallel implementation.

5.2 Smith-Waterman

In this section we exhibit the experimental results of Smith-Waterman implementations and we will try to figure out their behavior. Let n the length of both sequences to be aligned. We will study four cases depending on n ($n = 4K$, $n = 8K$, $n = 16K$ and $n = 32K$) and for each case we use as interconnection network first Gigabit Ethernet and then Myrinet. For each of the previous cases we utilize consecutively 8, 16, 32, 64 and 128 cores. Here we should mention that this application has another parameter that affects performance, and specifically this



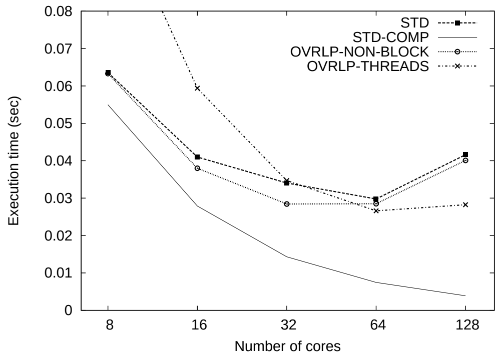
$(a) Ethernet$ $(b) Myrinet$

Figure 5.4: Smith-Waterman results for sequences with 4K length

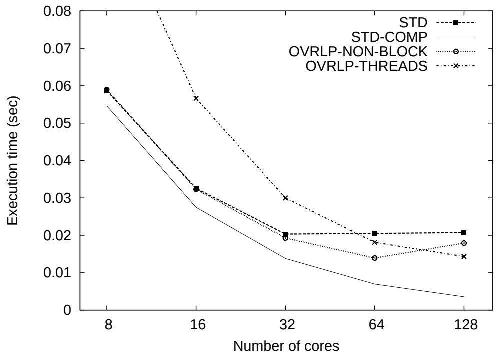
argument is the *tile factor*. As explained in subsection 4.2.3, tile factor determines the number of tiles that each process has to compute and essentially it regulates the *granularity* of the application. Moreover this factor also specifies how many tiles are computed completely in parallel. For example, when 128 cores are dedicated to the simple parallel program, all cores compute tiles fully parallel after 127 “steps”. So if the tile factor is large enough and the tiles that are available to each process are less that 127, then there is not any moment where all cores operate in parallel. Therefore we execute the experiments for various values of tile factor and the best results at each case are displayed. We refer the reader to [21], where finding the most appropriate tile size is discussed in detail. For each experiment we display in a figure the performance of the simple parallel program and the behavior of two alternative programs that overlap computation with communication.

In Figure 5.4 we can see the case where $n = 4K$. Let us concentrate first on the case of Ethernet. We can see that the naive parallel implementation scales poorly up to 32 processors and thereafter the total execution time increases abruptly. The programs that implement overlapping behave similarly referring to the scalability but they have improved performance as it can be shown in Figure 5.4(a). Analogous is the scalability of the applications when we use Myrinet as interconnection network. This poor performance can be justified if we have a look at the computation time of the standard program. We conclude that when more than 32 cores are used, the communication time (vertical distance of the two lines that are related with the standard program) is extremely augmented, hence the total time is too large and also we would not expect to gain much from overlapping as communication time is itself so significant.

Next we examine the case where each sequence has length 8K (Figure 5.5). When Ethernet is utilized, we observe that the pure parallel implementation has satisfactory scalability up to 64 cores, then communication factor becomes heavy



$(a)Ethernet$

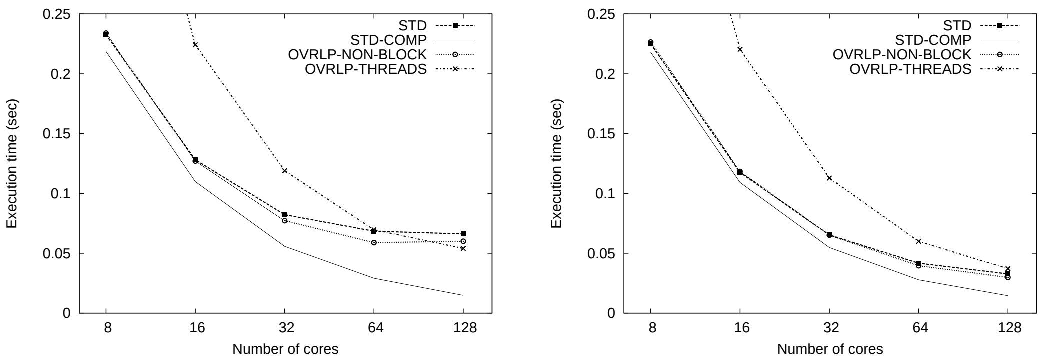


$(b)Myrinet$

Figure 5.5: Smith-Waterman results for sequences with 8K length

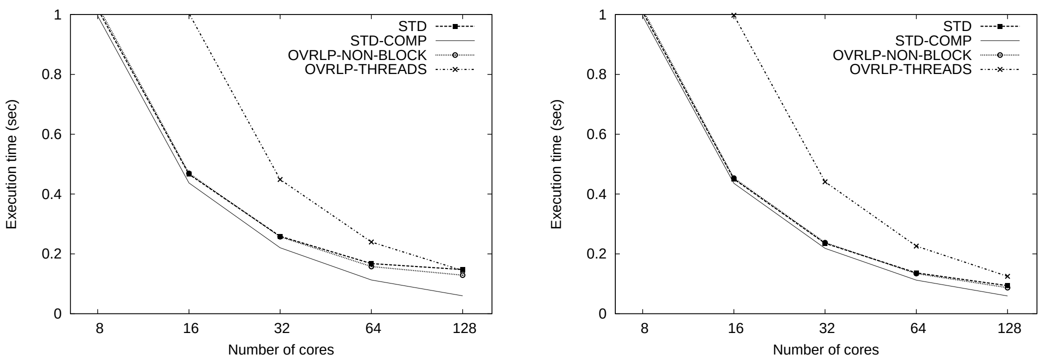
and the total time increases. On the other hand, the applications that overlap communication with computation improve the overall performance. To be more precise, the overlapping version that is based on non-blocking communication achieves its best execution time when uses 32 threads. Thereafter, overlapping with non-blocking communication becomes inefficient especially due to the interconnection network, if we recall the comments of subsection 3.2.3. The alternative overlapping implementation which handles separate threads scales even further (64 cores) and improves the performance. This is easily explained if we do not forget the fact that half cores execute computation and half communicate, which means that communication and computation behavior should be identical to the naive parallel program using 32 cores. So, the overlapping version does not suffer from exceeding communication as it is happening when 64 cores execute communication. It is worth saying however, that the first overlapping implementation accomplishes the same level of improvement with the alternative version by consuming half resources. When Myrinet is used, all applications exhibit better communication performance, and this gives eventually the opportunity for acceptable scalability. Although the simple parallel application does not scale further than 32 cores, there is such a ratio between computation time and communication time that overlapping manages to improve remarkably the overall performance and the implementations that use overlapping continue to scale (the first implementation up to 64 cores and the latter up to 128 cores). Again we emphasize that the first implementation with overlapping scheduling optimizes the simple parallel program by using half resources in comparison with the alternative implementation.

In the third experiment (Figure 5.6), again we can mark insufficient scalability of the simple parallel application. The programs that exploit overlapping scheduling optimize the total execution time and particularly the version that handles helper threading displays the best performance when Ethernet is used and all 128



$(a) Ethernet$ $(b) Myrinet$

Figure 5.6: Smith-Waterman results for sequences with 16K length



$(a) Ethernet$ $(b) Myrinet$

Figure 5.7: Smith-Waterman results for sequences with 32K length

cores participate in execution. Though, in the case of Myrinet, interesting conclusions can be drawn. Up to the state where 64 cores are used, computation is the dominating factor and in the next state (128 cores) the ratio of computation time to communication time is 1:1. So, the implementation with the non-blocking message passing improves slightly the ultimate performance of the pure parallel program while the other implementation does not. This is expected, because in the latter program only 64 threads execute computations, therefore computation time itself is comparable with the total execution time of the naive program at the final state (128 cores).

Finally, in Figure 5.7 we investigate the last case where $n = 32K$. As the length of sequences is large enough and computation is the major portion of total execution time, the experimental results are akin to the previous case. Here,

Length of sequences	Percentage performance improvement			
	Ethernet		Myrinet	
	non-block. commun.	helper threads	non-block. commun.	helper threads
4 K	2%	7%	13%	14%
8 K	5%	11%	32%	30%
16 K	11%	18%	9%	-13%
32 K	13%	2%	7%	-33%

Table 5.2: Percentage performance improvement of Smith-Waterman parallel implementation when overlapping is applied. A negative amount indicates how much "slower" is the overlapping version.

the most effective program is the parallel version that overlaps computation with communication by using non-blocking communication.

In table 5.2 we summarize the results of each experiment.

5.3 Advection 3D

Here we exhibit the experimental results of the 3D advection equation and we will try to interpret their behavior. Our study case consists of three 3D spaces ($64\times64\times64$, $128\times128\times128$ and $256\times256\times256$) and we desire to solve the equation until discrete time $T = 500$. For each case we use as interconnection network first Gigabit Ethernet, then Myrinet and we utilize consecutively 8, 16, 32, 64 and 128 cores. Similarly to the previous application, this one has also a parameter that affects performance, the *tile factor*. Additionally, as explained in subsection 4.3.3, the 3D space is divided into chunks and assigned to processes that are placed in a 3D grid. Thus, for a granted number of processes there are many ways to organize them in 3D grid and this organization affects substantially communication and computation time. So, it is preferred for a process to exchange data with processes that run on physically adjacent cores, and moreover, the topology of the 3D process-grid determines the morphology of the chunks. The latter fact, not so obviously, affects the computation time of a chunk because cache issues come into play. Therefore we execute the experiments for various values of tile factor, for all possible 3D grids of processes and the best results at each case are displayed. For each experiment we display in a figure the performance of the simple parallel program and the behavior of two alternative programs that overlap computation with communication.

In Figure 5.8 we see the experimental results for a 3D space $64\times64\times64$. When we select Ethernet to be the interconnection network among the SMP nodes, the scalability is insufficient for all three applications. This is expected if we have a look at the computation time of the simple parallel program. The communication

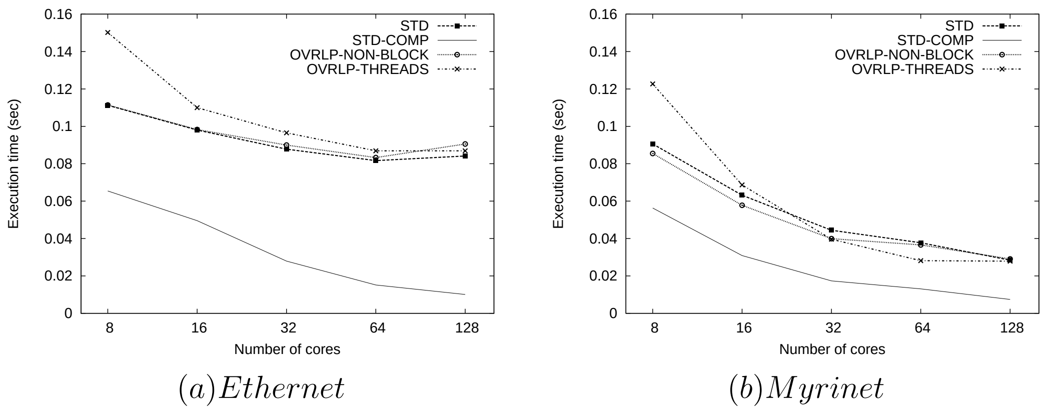


Figure 5.8: Results of advection equation for a 3D space $64 \times 64 \times 64$

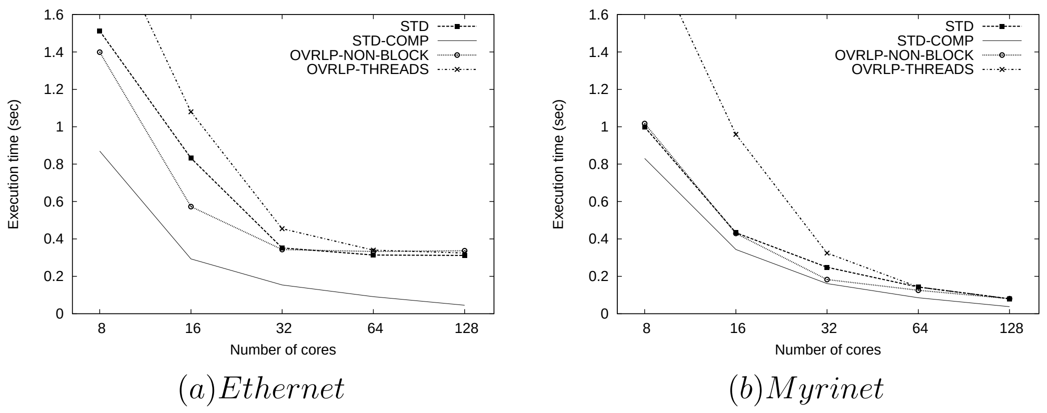
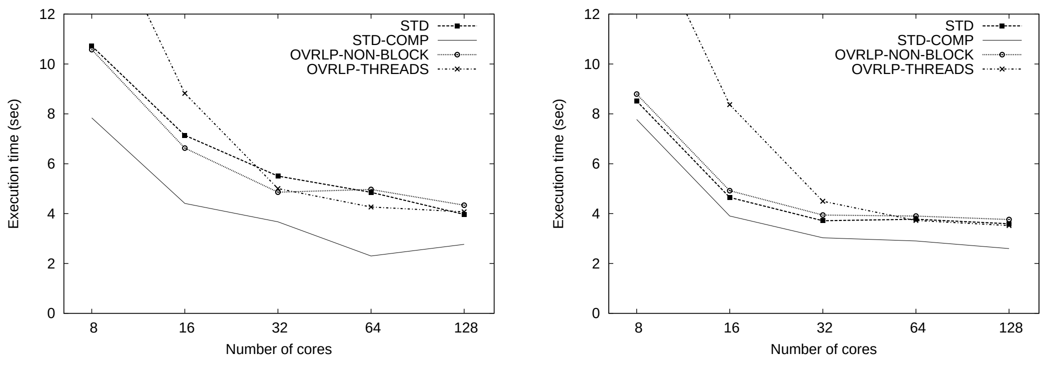


Figure 5.9: Results of advection equation for a 3D space $128 \times 128 \times 128$

time of the naive program is huge in comparison with the computation time even when just few cores participate in execution. So, the maximum of these times, which is the communication component, is approximately the total execution time in the applications that overlap computation with communication and there are not favorable conditions for improvement. In the case of Myrinet, scalability is slightly better, however the communication remains the dominating factor at an excessive rate. Hence, the overall performance is not improved when overlapping is applied. However, we should note that the overlapping version with helper threading exhibits satisfactory scalability up to 64 cores, while the naive implementation has the same performance when 128 cores are used, and consequently double resources are used.

The results are similar for the 3D space $128 \times 128 \times 128$ when Ethernet is used



(a)*Ethernet* (b)*Myrinet*

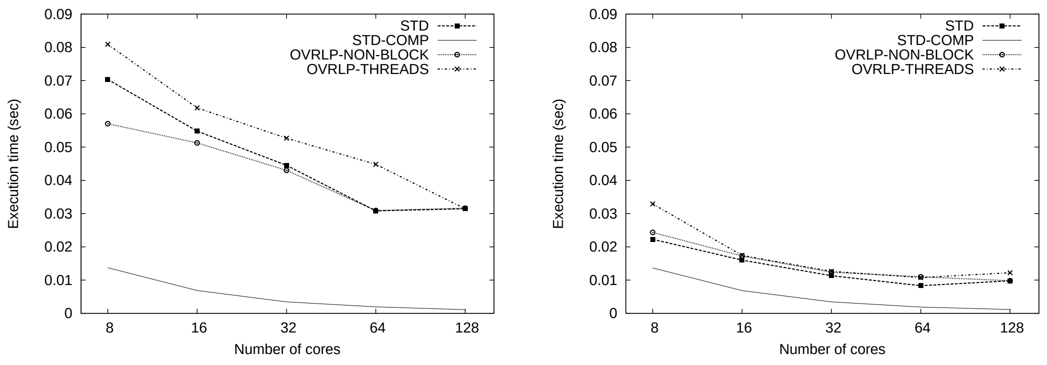
Figure 5.10: Results of advection equation for a 3D space $256 \times 256 \times 256$

(Figure 5.9(a)). The communication factor is severely larger than computation time and overlapping computation with communication is meaningless. On the other hand, as Myrinet supports communication in a much more efficient way, the dominating factor now is the computation time. In the last state (128 cores), the ratio of computation time to communication time is 1:1 and thus we can assume that in the next stage overlapping would be beneficial for the overall performance. This assumption is also strengthened by the experimental result of Figure 5.9(b), where we see that the naive parallel implementation and the applications that overlap computation with communication have approximately the same performance at the final stage (128 cores).

In Figure 5.10 we observe the final experiments for a 3D space $256 \times 256 \times 256$. Again, the applications scale poor (up to 32 cores) in both Ethernet and Myrinet. The explanation now is that granted the 3D space is large enough, every process has to compute a bulky chunk which, in turn, makes the computation component heavy. In addition to this, such a chunk now does not fit in the cache of a core and we do not expect to benefit much from cache reutilization. After all, we deduce that when the 3D space is large enough, the computation time is the major portion of total execution time and there are not favorable preconditions to apply overlapping.

5.4 Heat equation 3D

In this section we present the experimental results of the 3D heat equation. Our study case consists of three 3D spaces ($64 \times 64 \times 64$, $128 \times 128 \times 128$ and $256 \times 256 \times 256$) and we want to solve the equation until discrete time $T = 100$. For each case we use as interconnection network first Gigabit Ethernet, then Myrinet and we utilize progressively 8, 16, 32, 64 and 128 cores. Similarly with the



$(a) Ethernet$ $(b) Myrinet$

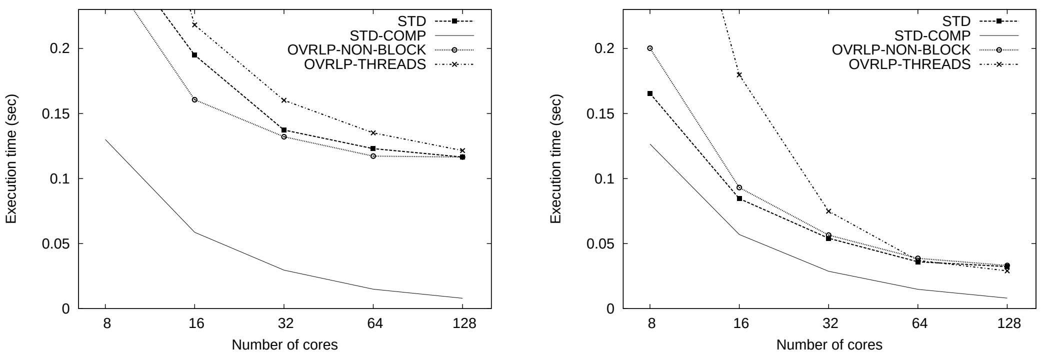
Figure 5.11: Results of heat equation for a 3D space $64 \times 64 \times 64$

advection equation, again the 3D space is divided into chunks and then assigned to processes that are placed in a 3D grid. There are many ways to organize a given number of processes in a 3D grid and this organization affects eventually communication and computation time. Therefore we execute the experiments for all possible 3D grids of processes and the best results at each case are illustrated.

In Figure 5.11 we can see the results for a 3D space $64 \times 64 \times 64$. Either we run the applications over Ethernet or over Myrinet, their scalability is inadequate. If we notice the ratio of computation time to communication time, we draw the conclusion that communication is such greater than computation that overlapping would not be profitable for the application. This extravagant communication is justified if we recall the communication pattern analyzed in subsection 4.4.3 and realize the considerable amount of data which has to be transferred between the processes. Additionally, the 3D space is relatively small, thus computations can be completed quickly and the parallelism of this application is fine-grained.

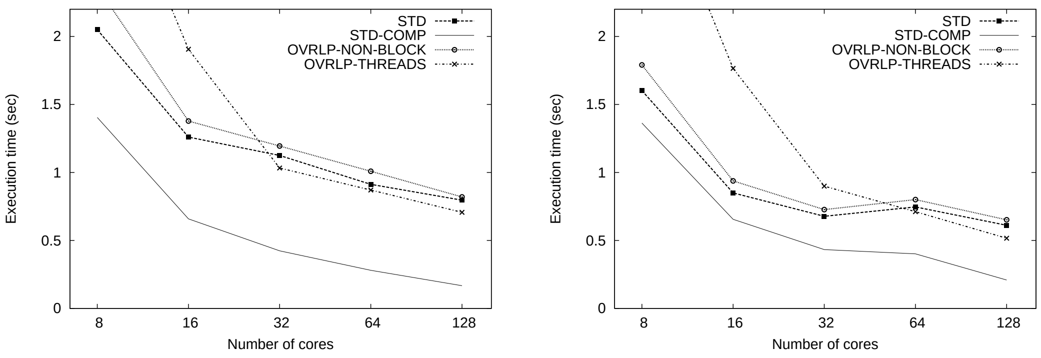
The next experimental result (3D space $128 \times 128 \times 128$) is exhibited in Figure 5.12. So, when the programs are executed over Ethernet, they do not scale further that 32 cores due to enormous communication overhead. On the other hand, when we exploit the Myrinet infrastructure, the analogy between computation time and communication time is propitious for applying the optimization of overlapping. Indeed, in Figure 5.12(b) we see that the overlapping version with helper threading scales up to 128 cores and improves the performance of the simple parallel program by 10%. The alternative implementation that takes advantage of the non-blocking communication functions has finally the same behavior with the naive parallel implementation.

The final experimental result which involves heat equation in a 3D space $256 \times 256 \times 256$ is displayed in Figure 5.13. In the first case (Ethernet as interconnection network), the naive parallel application scales up to 128 cores, however



$(a) Ethernet$ $(b) Myrinet$

Figure 5.12: Results of heat equation for a 3D space $128 \times 128 \times 128$



$(a) Ethernet$ $(b) Myrinet$

Figure 5.13: Results of heat equation for a 3D space $256 \times 256 \times 256$

Size of 3D space	Percentage performance improvement			
	Ethernet		Myrinet	
	non-block. commun.	helper threads	non-block. commun.	helper threads
64^3	0%	-2%	-18%	-29%
128^3	0%	-4%	-3%	10%
256^3	-3%	11%	-7%	16%

Table 5.3: Percentage performance improvement of 3D heat equation parallel implementation when overlapping is applied. A negative amount indicates how much "slower" is the overlapping version.

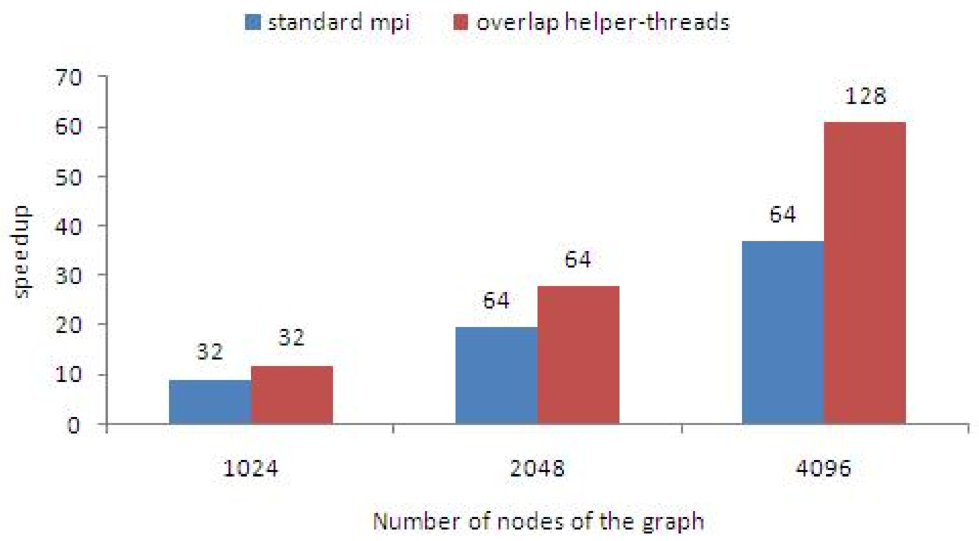
the implementation that uses helper threading for communication and overlaps computation with computation improves the best performance by 11%. The alternative implementation of the overlapping scheduling does not indicate better performance, instead it actually behaves like the naive parallel program. If we exploit Myrinet as interconnection network, the standard parallel program scales only up to 32 cores and so does the implementation that uses non-blocking functions to overlap computation with communication. However, the alternative implementation of overlapping with helper threading continues to improve its performance up to 128 cores and finally its best performance is 16% greater than the best performance exhibited by the standard parallel implementation. The ratio of computation to communication is almost unit in the state where 64 cores execute operations, so one would expect improvement by overlapping computation with communication and that is the case.

In table 5.3 we summarize the results recorded during these last experiments.

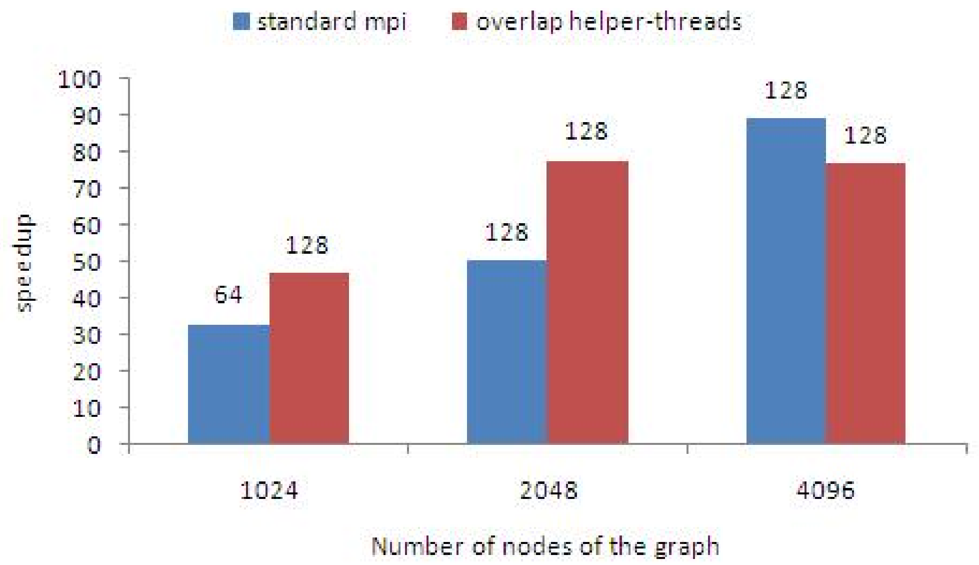
5.5 Overall experimental results

Until now we have evaluated the experimental results by examining the execution time. In this section we investigate the results from the scope of the speedup the parallel implementations have over the serial programs. For each one of the aforementioned applications we present two figures, one regarding Ethernet and one for Myrinet. The best speedup of the parallel implementations is given in the form of bars and above each bar we indicate the number of cores utilized to achieve this result.

In Figure 5.14, we illustrate the speedup of the Floyd-Warshall parallel implementations. In the case of Ethernet, we observe that the speed up of the baseline parallel implementation is poor and it increases as the size of graph becomes larger. This is expected mainly due to two reasons. Basically, when the size of the graph is larger, then the computational load of every process is greater as well and the parallelism tends to become coarse-grain. Moreover, as the graph becomes large



$(a) Ethernet$



$(b) Myrinet$

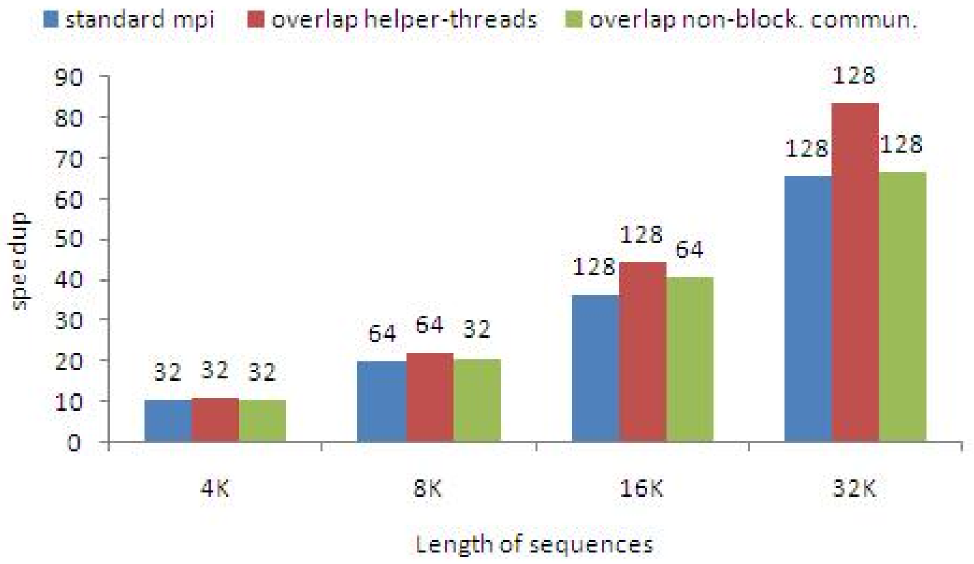
Figure 5.14: Speedup of Floyd-Warshall parallel implementations

enough it does not fit to the cache of a core and thus the serial program suffers from cache misses, while the aggregate cache size in the parallel implementations is sufficient (e.g. when 64 cores participate in execution the available "aggregate" cache is 64 times greater). The baseline parallel program scales no more that 64 cores and the overlapping program demonstrates notable improvement. We emphasize the case of the graph with 4096 nodes where the overlapping version achieves a speedup 67% greater than the naive implementation, however double resources should be dedicated. In the case of Myrinet, again we observe that as the size of the graph increases the speedup is even more satisfactory. In contrast to the previous case, the high performance interconnection provides the opportunity for better scalability and hence higher speedup. Again, in the first two case studies the overlapping version improves drastically the best performance. In the last experiment, the overlapping implementation has not adequate number of cores for the computation needed (just half cores execute computations) and naturally it does not overcome the performance of the baseline parallel implementation.

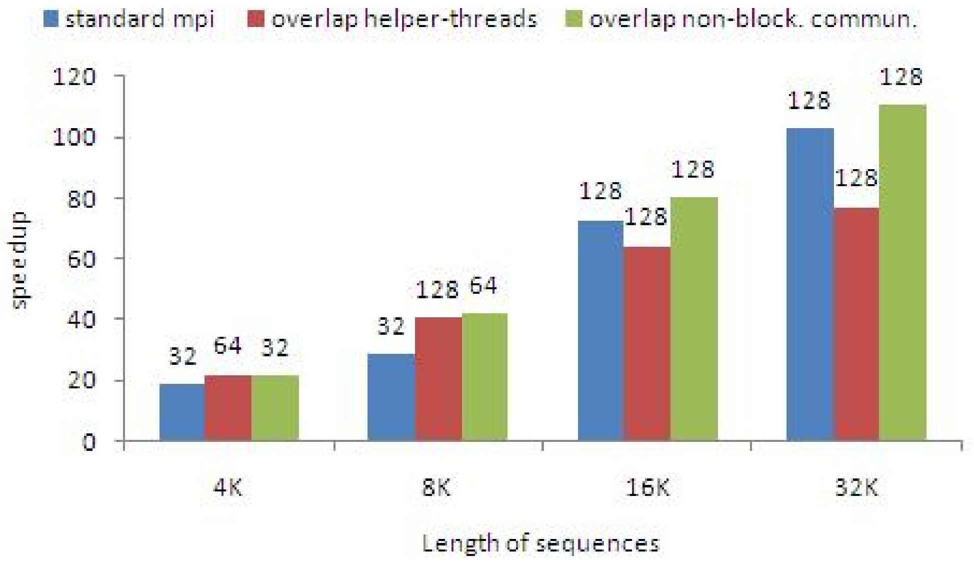
In Figure 5.15 we illustrate the speedup accomplished by the parallel implementations for the Smith-Waterman algorithm. Similarly to the previous application, longer inputs provide the opportunity for better scalability and acceptable speedup even when Ethernet is used as interconnection network. Also, in the case of commodity interconnect (Ethernet), the overlapping version with helper threading improves considerably the speedup of the naive parallel program, but the alternative version with non-blocking functions is not suitable as the network infrastructure limits the overlapping of computation with communication. The landscape changes when Myrinet is utilized since the overlapping version with non-blocking functions scales better than the rest parallel implementations. Particularly for long sequences-inputs, the speedup approaches the ideal value. Furthermore, we note that the sequences of 8K length are of great practical importance as this is a common length of proteins and in this case, overlapping implementations accelerate the execution significantly.

In Figure 5.16 we show the experimental results regarding the 3D advection equation. First of all, we figure out that in the case of Ethernet the speedup is extremely low and the interconnection network does not support successfully the intensive communication of the application. Therefore, all parallel applications do not scale up efficiently. When Myrinet is utilized, the applications show up acceptable speedup only for the second case study (3D space $128 \times 128 \times 128$) and this is explained thoroughly in previous section. In this application and for these inputs there are not favorable preconditions to apply overlapping and it is verified by the experimental results.

Finally, in Figure 5.17 we see the results regarding the parallel implementations of the 3D heat equation. The reader should keep in mind that this application has inherently complex communication pattern and the parallelism is inevitably fine-grained. As a result, the speedup is not high either we utilize Ethernet or Myrinet as interconnection network. However, the overlapping implementation with helper

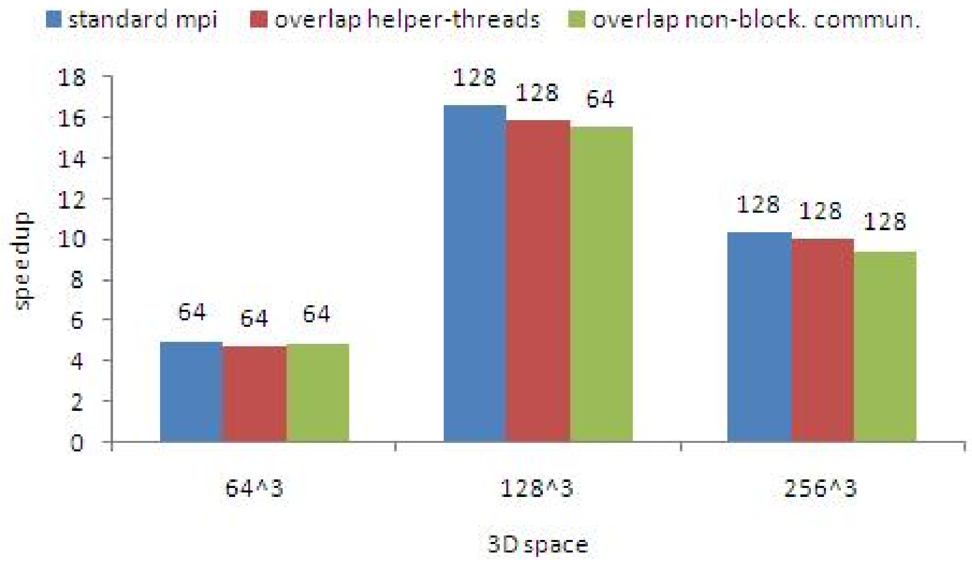


$(a) Ethernet$

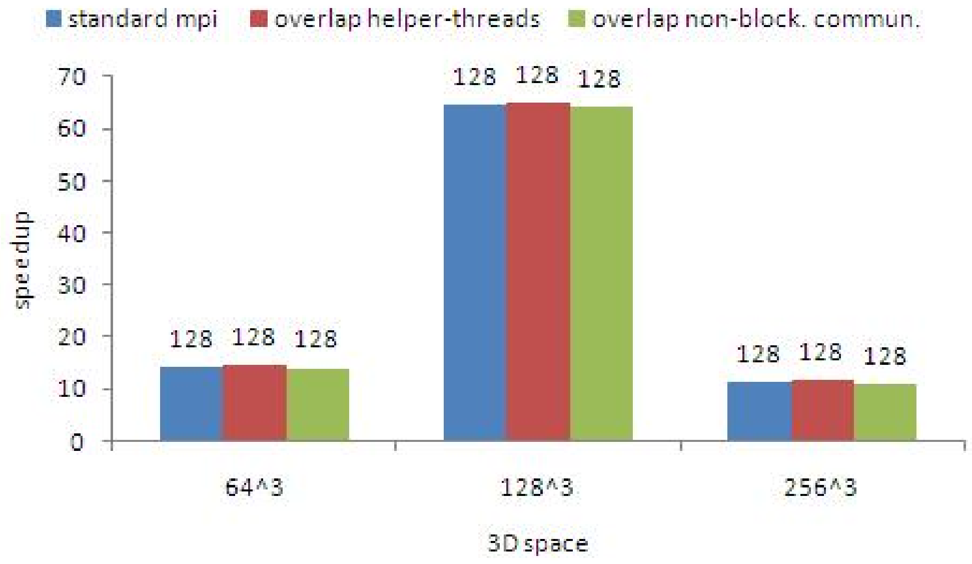


$(b) Myrinet$

Figure 5.15: Speedup of Smith-Waterman parallel implementations

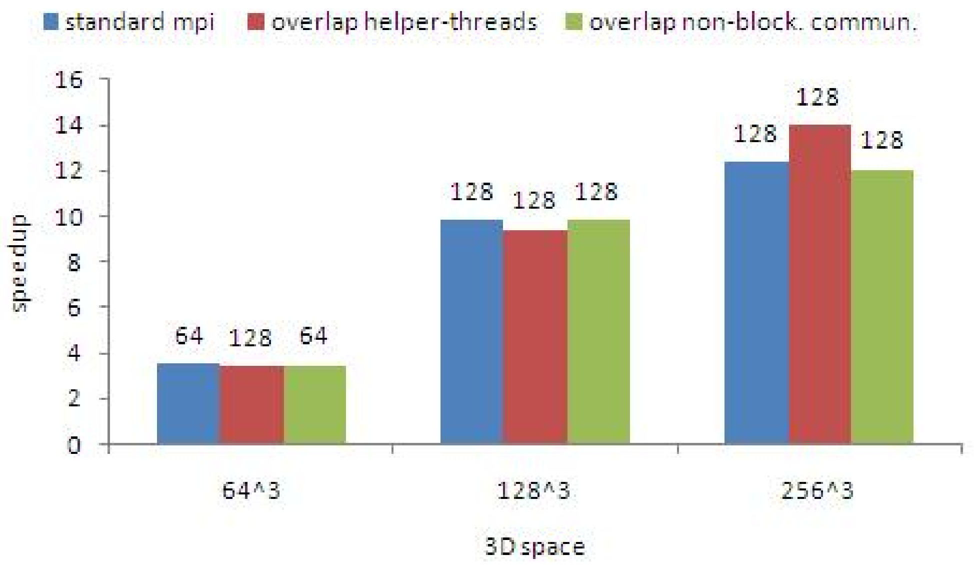


(a)*Ethernet*

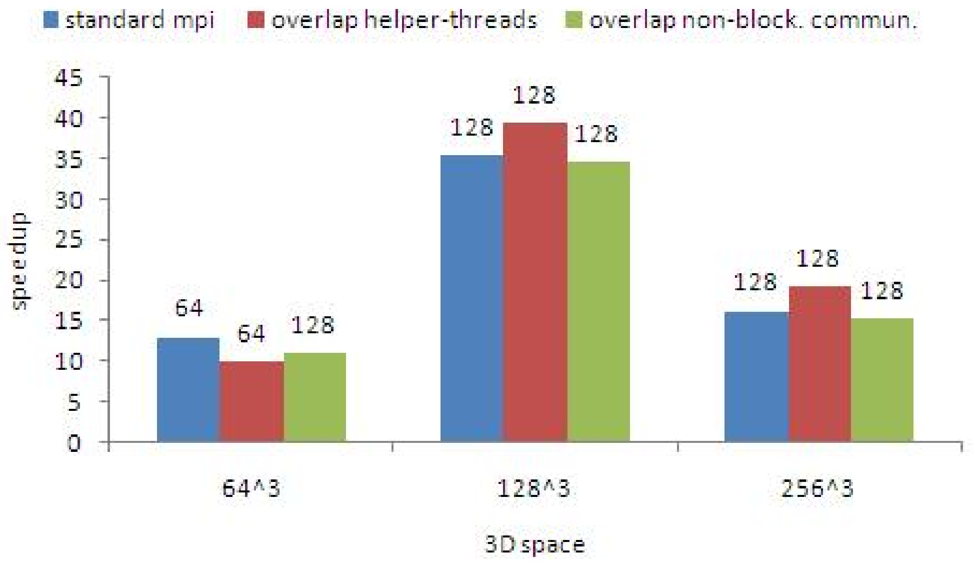


(b)*Myrinet*

Figure 5.16: Speedup of 3D advection equation parallel implementations



(a)*Ethernet*



(b)*Myrinet*

Figure 5.17: Speedup of 3D heat equation parallel implementations

threads boosts up the performance of the baseline program in several cases using the same resources.

Chapter 6

Conclusions

In this diploma thesis we examined the behavior of four parallel applications that represent corresponding families of programs: Floyd-Warshall uses collective communication, Smith-Waterman is a typical algorithm of dynamic programming, 3D advection equation requires the communication between processes only towards a specific direction and the 3D heat equation has a “halo” communication pattern, i.e. every process exchanges data with *all* of its neighbors.

These applications have problematic scalability due to excessive communication overhead either we utilize commodity interconnects (Ethernet) or a high performance interconnection network (Myrinet). Therefore we modified these parallel applications in order to overlap computation with communication and optimize the overall performance. We implemented this optimization technique in two different ways. Initially we used non-blocking communication, which is a method suitable for clusters popular in previous decades since each node had a uniprocessor. The alternative implementation utilizes separate threads for the communication and computation tasks and this technique is supported by a hybrid programming model which in turn is popular for the modern clusters consisting of SMP nodes.

The overall goal is to investigate the viability of this optimizing technique not only for clusters with high performance interconnects, but also for clusters with commodity networks as they are the majority in high performance computing. Generally, the experimental results have shown that the performance of the naive parallel applications is notably improved when overlapping is applied. However, which one of the overlapping implementations is more suitable for an application depends on its attributes and essentially on the interconnection network of the cluster.

In clusters with commodity interconnects, helper threading seems to be much more advantageous than the overlapping implementation with non-blocking functions. The latter functions do not behave “ideally” on networks like Ethernet, i.e. they waste computational resources and actually the overlapping is limited. On the other hand, a high performance interconnect, e.g. Myrinet is propitious for the latter overlapping implementation and these programs exhibit the best speedup.

It is worth saying that helper threading demands in general more computational resources because half of them are dedicated to computations and the rest to communications. However, this method shows up better scalability because the contention for network resources is reduced (half of the available cores execute communication operations and require network resources).

Despite the fact that the algorithmic changes needed to overlap computation and communication are common for the two aforementioned techniques, implementation issues come into play and affect the programmability of the method with helper threads. While the "traditional" method with non-blocking functions is quite straightforward and does not demand special care from the programmer's side, helper threading needs thorough implementation in order to accomplish high level performance. Primarily, the developer has to determine which parameters of the hybrid program will bring the optimal result, e.g. which data should be shared among threads, which place is the most suitable for synchronization among threads etc.

Since overlapping looks to be profitable in several cases, a tool that automatically applies this optimization in parallel applications would be very useful to developers. The preconditions that are favorable for applying this method are known, i.e. the computation/communication ratio should be approximately unit, and also, in some cases the algorithmic modifications can be conducted mechanically in order to allow overlapping. So a performance model of the naive parallel application should be sufficient for a tool to detect when it is meaningful to optimize the application with this technique. The development of such a tool would be an interesting future work.

Finally, as the clusters with SMP nodes become even more widespread and the hybrid programming blooms, we suggest as a future work the conversion of our overlapping implementations so that they utilize variable number of threads for the communication and the computation part. Then our implementations would be a special case, where each part is assigned to a separate thread, and furthermore this mixed model provides the opportunity for various parallelization schemes according to the available execution platform.

Bibliography

[1] Amdahl, G., *The validity of the single processor approach to achieving large-scale computing capabilities.* In Proceedings of AFIPS Spring Joint Computer Conference, Atlantic City, N.J., AFIPS Press, April 1967

[2] Flynn, M., *Some Computer Organizations and Their Effectiveness.* IEEE Transactions on Computers, 1972

[3] Norris, Mark, *Gigabit Ethernet Technology and Applications.* Artech House, 2002

[4] S. Majumder and S. Rixner, *Comparing Ethernet and Myrinet for MPI Communication.* In Proceedings of 7th Workshop on languages, compilers, and run-time support for scalable systems, Houston Texas, 2004

[5] Nanette J. Boden , Danny Cohen , Robert E. Felderman , Alan E. Kulawik , Charles L. Seitz , Jakov N. Seizovic , Wen-king Su, *Myrinet: A Gigabit-per-Second Local Area Network.* IEEE Micro, 1995

[6] B. Chapman, G. Jost, R. van der Pas, D.J. Kuck (foreword), *Using OpenMP: Portable Shared Memory Parallel Programming.* The MIT Press, 2007

[7] Pacheco, Peter S., *Parallel Programming with MPI.* Morgan Kaufmann, 1997

[8] Tsanakas P., Koziris N., Papakonstantinou G, *Chain Grouping: A Method for Partitioning Loops onto Mesh-Connected Processor Arrays.* IEEE Transactions on Parallel and Distributed Systems, 2004

[9] Drossitis I., Goumas G., Koziris N., Papakonstantinou G., Tsanakas P, *Evaluation of Loop Grouping Methods based on Orthogonal Projection Spaces.* International Conference on Parallel Processing, Toronto, Canada, 2000

[10] Abhinav Bhatele, Pritish Jetley, Hormozd Gahvari, Lukasz Wesolowski, William D. Gropp and Laxmikant V. Kale, *Architectural constraints to attain 1 Exaflop/s on three scientific application classes.* Accepted for the IEEE International Parallel and Distributed Processing Symposium (IPDPS'2011), Anchorage, USA, 2011

[11] T.V. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser, *Active Messages: A Mechanism for Integrated Communication and Computation.* In Proceedings of 25 Years ISCA: Retrospectives and Reprints, 1998

[12] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, *Optimizing bandwidth limited problems using one-sided communication and overlap.* In The 20th International Parallel and Distributed Processing Symposium (IPDPS), 2006

[13] G. Goumas, A.Sotiropoulos and N. Koziris, *Minimizing Completion Time for Loop Tiling with Computation and Communication Overlapping.* Proceedings of the 2001 International Parallel and Distributed Processing Symposium, IEEE Press, San Francisco, California, 2001

[14] Rolf Rabenseifner, Georg Hager, and Gabriele Jost, *Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes.* In Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP '09), 2009

[15] Floyd, Robert W., *Algorithm 97: Shortest Path.* Communications of the ACM, 1962

[16] Warshall, Stephen, *A theorem on Boolean matrices.* Journal of the ACM, 1962

[17] Smith F. Temple, Waterman S. Michael, *Identification of Common Molecular Subsequences.* Journal of Molecular Biology, 1981

[18] J. Ramanujam, P. Sadayappan, *Tiling Multidimensional Iteration Spaces for Multicomputers.* Journal of Parallel and Distributed Computing, 1992

[19] R. Courant, K. Friedrichs, and H. Lewy, *On the Partial Difference Equations of Mathematical Physics.* IBM Journal of Research and Development, 1967

[20] K.W. Morton, D.F. Mayers, *Numerical Solution of Partial Differential Equations: An Introduction.* Cambridge University Press, Cambridge, England, 1994

[21] Goumas G., Drosinos N., Koziris N., *Communication-aware Supernode Shape.* IEEE Transactions on Parallel and Distributed Systems, 2008

[22] Goumas G., Anastopoulos N., Ioannou N., Koziris N., *Overlapping Computation and Communication in SMT Clusters with Commodity Interconnects.* International Conference on Cluster Computing, 2009