ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Performance prediction models for large-scale parallel applications

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

Ντάγιου Ευριδίκης Βασιλείας

Επιβλέπων: Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής

Αθήνα, Οκτώβριος 2012

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Performance prediction models for large-scale parallel applications

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

Ντάγιου Ευριδίκης - Βασιλείας

Επιβλέπων: Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 11η Οκτωβρίου 2012.

.....	.....	.....
Ν. Κοζύρης	Α. Παγουρτζής	Ν. Παπασπύρου
Αν. Καθηγητής ΕΜΠ	Επ. Καθηγητής ΕΜΠ	Επ. Καθηγητής ΕΜΠ

Αθήνα, Οκτώβριος 2012

.....

Ντάγιου Ευριδίκη Βασιλεία

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Abstract

Performance modeling plays a fundamental role in the design of computer systems. This applies especially to parallel systems where high performance is of key interest. While performance modeling of sequential computer systems already poses a number of important problems, the problem involved with performance modeling of parallel systems is even more fundamental. There exists a wide variety of approaches to the performance modeling of parallel systems, each representing a different trade-off between the accuracy of the analysis and the computational cost involved. In this diploma thesis we propose a model that attempts to predict the resources needed for the maximization of the performance of a parallel application. Similar to some of the existing techniques, the approach is primarily aimed to support the initial phases in the design of parallel systems where the emphasis is on extremely low solution cost, rather than on high accuracy. Our execution platform is a cluster of SMP nodes and there is an available commodity interconnect, Gigabit Ethernet. We evaluate our model in memory bound computational kernel Heat Equation in a 2 dimensional and 3 dimensional space, and receive accurate prediction results not only for the speedup of the applications, but the efficiency as well.

Keywords: parallel computing, performance modelling, performance prediction

Contents

Chapter 1

Introduction

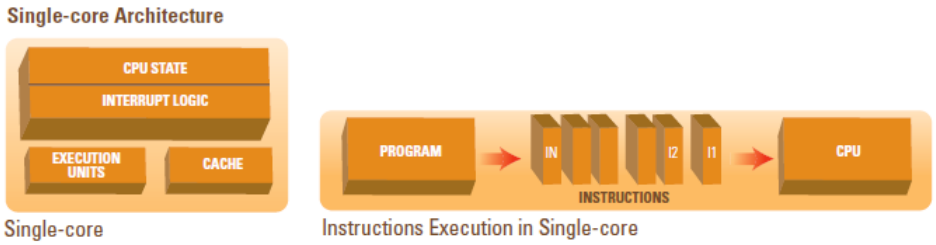
1.1 What is parallel computing?

Parallel Programming [9] is a form of computation in which program instructions are divided among multiple processors (cores, computers) in combination to solve a single problem, thus running a program in less time. Figure 1.1 shows the difference in instruction streams between serial and parallel processing. Designing and developing parallel programs has typically been a very manual process. The programmer is usually responsible for both identifying and actually implementing parallelism.

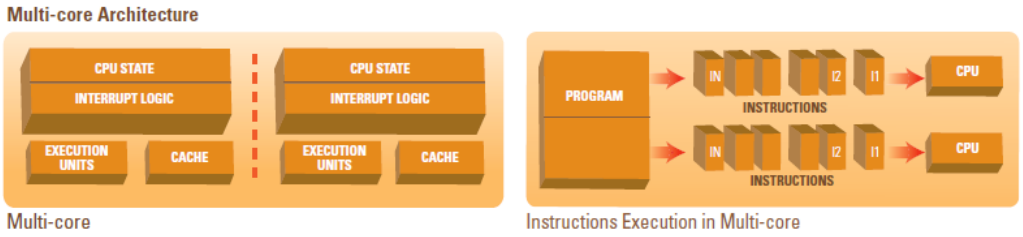
1.2 The Differences Between Serial and Parallel Processing

Computers are inherently serial. Working in parallel makes a lot more sense in many applications, so the parallel computer was invented, a set of serial computers working together. Parallel computers can make some jobs go a lot faster, but not all problems automatically run faster on parallel computers, and each problem must be broken up to run in parallel by a highly trained, and very expensive, parallel programmer.

The Central Processing Unit (CPU) takes instructions from main memory and executes them one at a time. After executing an instruction, the CPU gets the next instruction and continues to execute instructions serially. It can do anything that the programmer can describe in a sequence of instructions that the computer can understand. It is inherently serial.



(a)



(b)

Figure 1.1 (a) Single-core architecture (b) Multi-core architecture

Serial processing is like using a laundromat that has only one washer and dryer. If you have a lot of laundry, it will take a long time. With enough machines you can do dozens of loads in about the same time as one. However, it is not always possible to break problems into parts that can be run simultaneously. If you are summing or searching through a million numbers you can form groups of 1,000 numbers each and process the groups simultaneously on 1,000 processors, finishing 1,000 times faster than on one processor. Other tasks, like dividing two numbers, must be performed sequentially.

The first disadvantage of parallel computing is cost. Good parallel computers used at high performance centers cost millions of dollars. Software and trained programmers for parallel computers are also more expensive. Furthermore, even if a problem can be broken up into parts that can be run at the same time, it can be difficult to coordinate all the parts.

1.3 The need for Parallel Computing

We can start by examining how multicore computers are emerging as a new wave of technology, and how implementation of parallel running applications can better utilize the multicore resources. Multicore processors are formed as a result of combining two or more solo cores into a unit comprising a single integrated circuit, and came into being since improving performance by increasing the clock speed had reached its physical limitations.

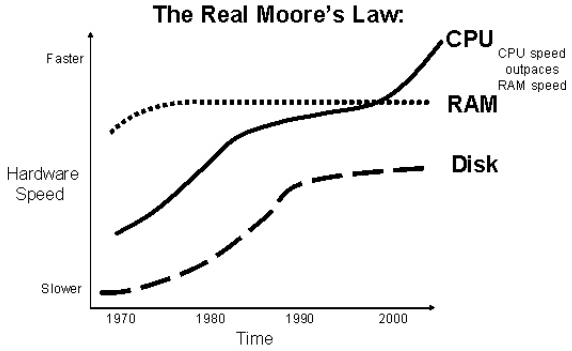


Figure 1.2 Difference between cpu speed and memory speed

The traditional way of programming (sequential programming) will not efficiently take advantage of multicore systems. In order to fully exploit these multicore machines, organizations need to redesign applications so that the processors can treat them as multiple threads of execution. Programmers need to find the optimum spots in their codes to insert the parallel code, divide the work approximately into equal parts that can run simultaneously and associate the precise times for the communication of the threads. A parallel application might also not work efficiently in the traditional Von Neumann model architecture, which implies that architects may have to change their prospect of designing. In other words, the question whether or not it is wise to use parallelism rises.

1.4 Why use parallel computing?

In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel computers can be built from cheap, commodity components. This way, we can save time and money. Many problems are so large and complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory. Such applications require the processing of large amounts of data in sophisticated ways. For example databases, data mining, web search engines, web based business services and numerous engineering applications- from prosthetics to spacecraft - processing millions of operations per second. Both physical and practical reasons pose significant constraints to simply building ever faster serial computers. Issues such as the transmission speeds - the speed of a serial computer is directly dependent upon how fast data can move through hardware and there are absolute limits like the speed of light and the transmission limit of copper wire - economic limitations - it is increasingly expensive to make a single processor faster and using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive - limits to miniaturization e.t.c. make parallel computing a corollary in high-performance computing in general. [12]

Although there are important problems to overcome, it is out of question that there should be made way for the implementation of concurrency.

Chapter 2

Multi-core architectures

2.1 Von Neumann Architecture

In the von Neumann architecture, there exists one shared memory for instructions (program) and data with one data bus and one address bus between processor and memory. Instructions and data have to be fetched in sequential order, limiting the operation bandwidth. Virtually all computers have followed this basic design, differing from earlier computers which were programmed through ”hard wiring”. Computers following this architecture are comprised of four main components:

- Memory
- Control Unit
- Arithmetic Logic Unit
- Input/Output

Random access memory is used to store both program instructions and data. Program instructions are coded data which tell the computer to do something, whereas data is simply information to be used by the program. The control unit fetches instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task. Aritmetic Unit performs basic arithmetic operations and Input/Output is the interface to the human operator.

Well, parallel computers still follow this basic design, just multiplied in units. The basic, fundamental architecture remains the same.

2.2 Flynn’s Classical Taxonomy

There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn’s Taxonomy. Flynn’s taxonomy distinguishes

multi-processor computer architectures according to how they can be classified along the two independent dimensions of Instruction and Data. Each of these dimensions can have only one of two possible states: Single or Multiple.

Single Instruction, Single Data (SISD):

- Single Instruction: Only one instruction stream is being acted on by the CPU during any one clock cycle
- Single Data: Only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and even today, the most common type of computer

Single Instruction, Multiple Data (SIMD):

- Single Instruction: All processing units execute the same instruction at any given clock cycle
- Multiple Data: Each processing unit can operate on a different data element
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.

Multiple Instruction, Single Data (MISD):

- Multiple Instruction: Each processing unit operates on the data independently via separate instruction streams.
- Single Data: A single data stream is fed into multiple processing units.

Multiple Instruction, Multiple Data (MIMD):

- Multiple Instruction: Every processor may be executing a different instruction stream
- Multiple Data: Every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Currently, the most common type of parallel computer - most modern supercomputers fall into this category.
- Most current supercomputers, networked parallel computer clusters and ”grids”, multi-processor SMP computers, multi-core PCs.

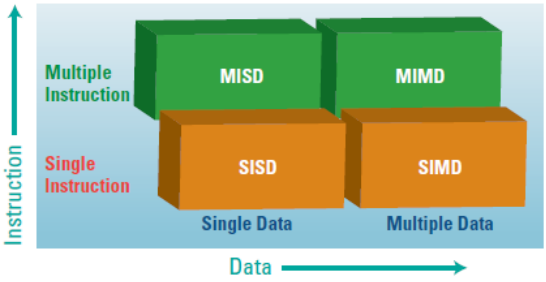


Figure 2.1 Flynn's Classical Taxonomy

2.3 Shared memory

Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space. Multiple processors can operate independently but share the same memory resources. Changes in a memory location effected by one processor are visible to all other processors. Shared memory machines can be divided into two main classes based upon memory access times: UMA and NUMA.

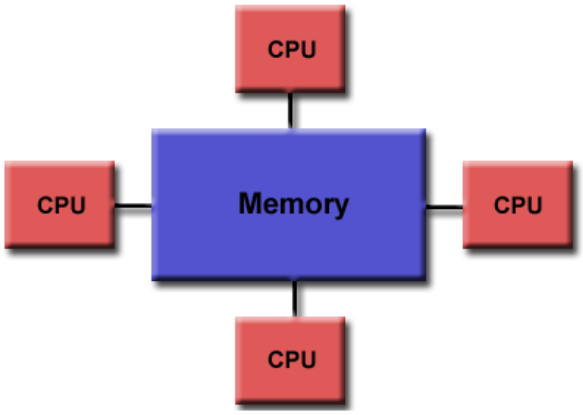


Figure 2.2 Shared memory

Uniform Memory Access (UMA): Most commonly represented today by Symmetric Multiprocessor (SMP) machines, uniform memory access machines have equal access and access times to memory. They are sometimes called Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level. Non-Uniform Memory Access (NUMA) machines are often made by physically linking two or more SMPs. What is important is that not all processors have equal access time to all memories, and memory access across link is slower. If cache coherency is maintained, then may also be called

CC-NUMA - Cache Coherent NUMA.

Advantages and Disadvantages of shared memory model

Despite providing a user-friendly programming perspective to memory and making data sharing between tasks both fast and uniform due to the proximity of memory to CPUs, global address space's primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increase traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management. Moreover, programmer's responsibility for synchronization structures that ensure correct – as the algorithm indicates - access of global memory, adds another level of difficulty in shared memory programming model.

2.4 Distributed memory

Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect interprocessor memory. Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors. Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply here. When a processor needs access to data in another processor, it is up to the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.

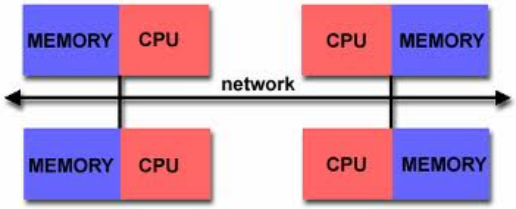


Figure 2.3 Distributed memory

In order for the communication to take place, an interconnection network is needed. [5] The interconnection network is responsible for fast and reliable communication among the processing nodes in any parallel computer. The demands on the network depend on the parallel computer architecture in which the network is used. Only if fast and reliable communication over the network is guaranteed will the parallel system exhibit high performance. Many different interconnection networks for parallel computers have been proposed . Interconnection networks can be categorized according to a number of criteria such as topology,

routing strategy, and switching technique. [6] They vary widely in their cost, their fault tolerance, their simplicity, their amenability to partitioning, and their aggregate bandwidth for local and nonlocal traffic patterns. More about the interconnection network used in our experiments is mentioned in following section.

Advantages and Disadvantages of distributed memory model

Distributed memory model's scaling is very satisfying. Increase in the number of processors results in a proportionate increase in the size of memory. Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherence. On the other hand, the programmer is responsible for many of the details associated with data communication between processors.

2.5 Hybrid Shared - Distributed Memory

The largest and fastest computers in the world today employ both shared and distributed memory architectures. The shared memory component can be a cache coherent SMP machine and/or graphics processing units (GPU). The distributed memory component is the networking of multiple SMP/GPU machines, which know only about their own memory - not the memory on another machine. Therefore, network communications are required to move data from one SMP/GPU to another. Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.

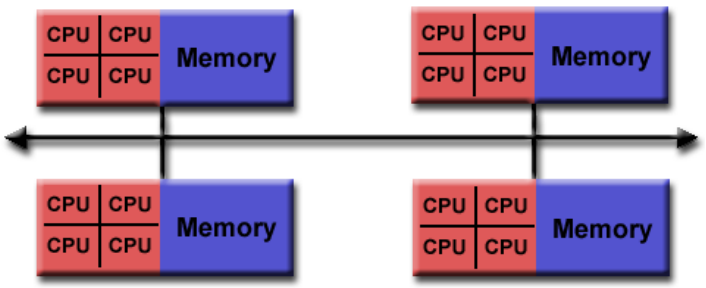


Figure 2.4 Hybrid Shared - Distributed memory

2.6 Programming models in multi-core architectures

A programming model is a bridge between a system developer's natural model of an application and an implementation of that application on available hardware. A programming model must allow the programmer to balance the competing goals of productivity and

implementation efficiency. Implementation efficiency is always an important goal when parallelizing an application, as programs with limited performance needs can always be run sequentially. [11]

Although it might not seem apparent, programming models are not specific to a particular type of machine or memory architecture. In fact, any of these models can theoretically be implemented on any underlying hardware.

2.6.1 Shared Memory Model

In this programming model, tasks share a common address space, which they read and write to asynchronously. Various mechanisms such as locks and semaphores may be used to control access to the shared memory.

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. Program development can often be simplified.

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 conserves memory accesses, cache refreshes and bus traffic that occurs 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. In order to implement shared memory programming, we used the Application Programm Interface OpenMP which will be discussed in following section.

2.6.2 Distributed Memory / Message Passing Model

The sequential paradigm for programming is a familiar one. The programmer has a simplified view of the target machine as a single processor which can access a certain amount of memory, and therefore writes a single program to run on that processor.

The message-passing paradigm is a development of this idea for the purposes of parallel programming. Several instances of the sequential paradigm are considered together. That is, the programmer imagines several processors, with each one encapsulating local data, and writes a program to run on each processor. What seems to be the key is the fact that parallel programming by definition requires cooperation between the processors to solve a task, such as data exchange, synchronisation e.t.c. which requires some means of communication by sending each other messages. Multiple processes can reside on the same physical machine or across an arbitrary number of machines. Thus, the message passing model has no concept of a shared memory space or of processors accessing each other's memory directly, which not only makes it harder to implement but also adds extra overhead in the time needed for the kernel to be executed, which we will call communication time.

A message transfer is when data moves from the main memory of one process to the main memory of another process through the interconnection network used in the platform. There are certain information that need to be provided to the interconnection network to specify the message transfer, such as the name of the processor that is sending the message, as well as the name of the receiving processor. The location of the data on the sending processor, where should the data be left on the receiving processor, the type of the data being sent and how much data is the receiving processor prepared to accept are crucial factors that define communication.

In general, the sending and receiving processors will cooperate in providing this information. Some of this information provided by the sending processor will be attached to the message as it travels through the system and the interconnection network may make some of this information available to the receiving processor. As well as delivering data, the message passing system has to provide some information about progress of communications. A receiving processor will be unable to use incoming data if it is unaware of its arrival. Similarly a sending processor may wish to find out if its message has been delivered. A message transfer therefore provides synchronisation information in addition to the data in the message. For the time being we are only interested in general concepts rather than the details of particular implementations, which will be furtherly discussed in a following section in which MPI is introduced. [4] There are two types of communication among the processes, which are being discussed. These are point-to-point communication and collective communication.

Point to Point Communication

The simplest form of message passing is a point to point communication which involves a single source and a single destination. A message is sent from the sending processor to a receiving processor. Only these two processors need to know anything about the message. There are several variations on how the sending of a message will take place, concerning the interaction of the message with the program being executed from the processes at this specific time.

The first common distinction is between synchronous and asynchronous communication. Synchronous sends and receives are provided with information about the completion of the message. Asynchronous operations, on the other hand, only know when the message has left.

A useful example would be denoting the main differences between using a fax machine and a mail box in order to send a message to someone. A fax message or registered mail is a synchronous operation. The sender can find out if the message has been delivered. A post card is an asynchronous message. The sender only knows that it has been put into the post-box but has no idea if it ever arrives unless the recipient sends a reply.

The other important distinction is blocking and non-blocking. A blocking send routine will only return after it is safe to modify the application send buffer. This implies a handshaking with the receive process to confirm a safe send. A blocking receive only returns after the data has arrived and is ready for use by the program. In case of non-blocking mode, both send

and receive routines return straight away and allow the execution of the program to continue. Non-blocking operations request the MPI library to perform the operation when it is able, and at some later time the sub-program can test for the completion of the non-blocking operation. It is unsafe to modify the application buffer until the requested non-blocking operation was actually performed by the library. There are ”wait” routines used to do this task. Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

Receiving a message can also be a non-blocking operation. For example turning a fax machine on and leaving it on, so that a message can arrive. You then periodically test it by walking in to the room with the fax to see if a message has arrived.

Collective Communication Operations

In general, all data movement among processes can be accomplished using MPI send and receive routines. More over, a set of standard collective communication routines are defined in MPI. Each collective communication routine has a parameter called a communicator, which identifies the group of participating processes. The collective communication routines allow data movement among all processors or just a specified set of processors. This kind of communication is used extensively in most data-parallel algorithms, and as a result the parallel efficiency of these algorithms depends on efficient implementation of these operations. They are equally applicable to distributed and shared address space architectures.

Chapter 3

Execution environment

In previous sections, 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.

3.1 Architecture of the system

Our system embraces the hybrid parallel architecture. It is a cluster consisting of SMP nodes where each node has two Intel (R) Xeon (R) E5335 processors with four cores each (eight cores in total per node). 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 6 MB. Additionally, each SMP node has one main memory slot, whose size varies from node to node. To form a hybrid system, the 16 available SMP nodes are connected via an interconnection network, Gigabit Ethernet or Myrinet. Our cluster is an heterogeneous system, thus execution time of applications may vary, according to which family of nodes we are measuring it at.

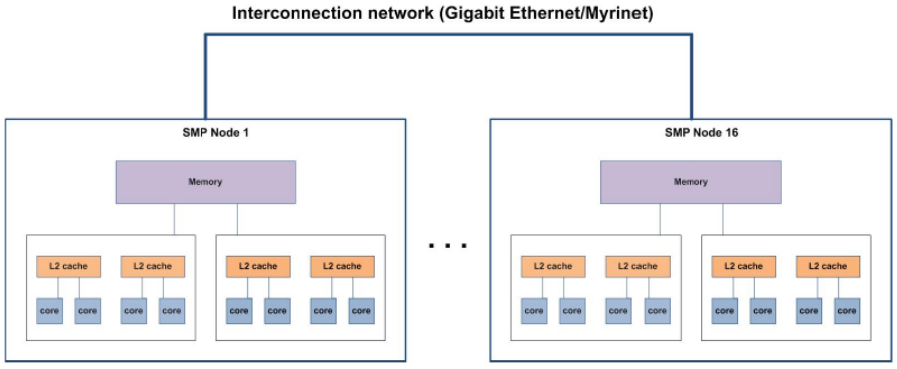


Figure 3.1 Overview or the architecture of the cluster

3.2 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 (overhead). 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. In this thesis, we study the performance of Gigabit Ethernet.

3.3 Gigabit Ethernet

Ethernet [7], 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 [8]. We should have this issue in mind as it will be discussed again in later chapter.

3.4 Implementation of distributed memory model using MPI

The Message Passing Interface Standard (MPI) is a message passing library standard based on the consensus of the MPI Forum, which has over 40 participating organizations, including vendors, researchers, software library developers, and users. The goal of the Message

Passing Interface is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. As such, MPI is the first standardized, vendor independent, message passing library. The advantages of developing message passing software using MPI closely match the design goals of portability, efficiency, and flexibility.

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. 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.

Pseudocode 3.1 MPI code example - Parallel calculation of f(0) + f(1)

```
#include <mpi.h>

int main(int argc,char** argv){
int v0,v1,sum,rank;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

if(rank==1)
        MPI_Send(&f(1),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();

}
```

3.4.1 Open issues in MPI implementation

MPI collective communication functions, such as broadcast and reduce, play a n important role in helping applications achieve good performance. Although a lot of research has been done on collective communication algorithms and some implementations have incorporated optimized algorithms , more work is still needed in some areas. For example, with the advent of multicore chips, MPI applications routinely have multiple processes on a single node connected with multiple processes on other nodes by an interconnection network. Therefore, collective communication algorithms must be designed to effectively use such a hierarchical communication topology. Although research has been done on topology-aware collectives, not all production implementations have incorporated such algorithms yet. Furthermore, optimized algorithms are needed for the entire set of collectives in MPI, not just a select few. The best algorithm for a particular collective communication function often depends on the message size and number of processes. In MPICH2, for example, the MPI collective functions use multiple algorithms, and one of them is selected for a specific message size and number of processes. However, the cutoff points for switching between algorithms are based on measurements performed some time ago on one platform. They may not be right for other platforms. A better approach is needed that determines the right cutoff points for the specific machine being used. Dynamic tuning of algorithms may also be needed.

Chapter 4

Performance modelling

4.1 Differences between serial and parallel performance modelling

The basis for performance tuning on uniprocessors is rooted in the Von Neumann machine model. The Von Neumann architecture has at least two benefits for performance tuning:

- The Von Neumann machine model suggests how to abstract the execution time of an implementation in simple terms, namely,as the number of instructions executed. The simple relationship between instructions executed and elapsed time encourages algorithms to be analyzed mathematically in advance of implementation. This means that mathematically analyzed algorithms will have some relevance to real implementations, since application performance will differ from the results of analysis by only constant factors.
- The Von Neumann machine model guarantees a simple relationship between the execution of program components and overall execution time; the execution time of a program is the sum of the execution times of its parts.

These two characteristics of the uniprocessor environment suggest a straightforward approach to performance tuning on serial machines: programmers start from algorithms which can be analyzed for efficiency, proceed to implementations which are known to be big-O optimal and attempt to reduce the values of the constants by measuring programs in execution, identifying which code segments are most cost effective locations for improvement, e.g. via code profiling. Unfortunately neither of these characteristics holds true in a parallel computing environment. There are two reasons why the serial model of performance efficient programming breaks down when using a parallel machine:

- The addition of multiple processors to a computer allows for an immense range of architectural diversity. Parallel computers are distinguished along many dimensions,

such as whether processors share memory, how processors communicate, how processors synchronize, and how memory is organized.The wide diversity of machine architectures has made it difficult to determine how to abstract parallel machines for analysis. The main analytic platform, the PRAM is too far removed from real machines to provide analyses useful to the implementer. As a result, successful analysis of parallel applications is highly machine specific.

- The simple relationship between the execution time of a program components and the program's overall execution time no longer holds. On a parallel machine, the execution time of an application is no longer the sum of its parts. As a result there is no guarantee that reducing the execution time of a code fragment will affect the running time of the application.

There are a number of implications arising from the breakdown of the uniprocessor model of application development. First of all, parallel programmers cannot in general start from a theoretical analysis of their application, or from an algorithm known to be optimal. As a result, the most common approach to developing a parallel program is to look for opportunities for parallelism in existing code or algorithms. Unfortunately many algorithms and codes allow for parallelism at multiple levels (such as procedure, loop and instruction levels) and at multiple locations within the program. In addition, there can be a wide range of synchronization methods, scheduling strategies, and task implementations appropriate for each source of parallelism. The result is that when comparing the programming process for a parallel machine to that of a serial machine, the guidance available from theory is diminished, yet the number and range of implementation decisions is increased.

A second result of the lack of theoretical basis for parallel program design is that the most common approach to performance tuning is the *measure-modify* paradigm. In the measure - modify paradigm, programmers first implement a parallel program, aiming for correctness, then measure its execution time to decide whether it performs acceptably. If changes are necessary, the program is modified and the process iterates. There are a number of reasons why measure-modify is an inefficient approach to finding the best implementation of a parallel program. First of all, simply measuring an application's performance gives little insight into the reasons behind any performance problems found. Second, even given an understanding of the reasons behind performance problems, it can be difficult to connect problems with the design decisions leading to them. Finally, and most important, after a new implementation is created, the program may exhibit new performance problems, perhaps even worse than the previous version.

Thus, in this diploma thesis, our effort is towards tuning the performance of already implemented algorithms, that is we don't consider two variables (performance modelling and best implementation) in our analysis, rather than we consider the implementation variable as a constant and try to maximize the performance variable in a given architecture system.

4.2 How do we actually measure success?

The task of the parallel software engineer is to design and implement programs that satisfy user requirements for correctness and performance. However, the performance of a parallel program is a complex and versatile issue as mentioned above. After considering costs incurred at different phases of the software life cycle, including design, implementation, execution, and maintenance, we must consider, in addition to the execution time and scalability of the computational kernels, the mechanisms by which data are generated, stored, transmitted over networks, moved to and from disk, and passed between different stages of a computation.

Hence, the goal of the design process is not to optimize a single metric such as speed. Rather, a good design must optimize a problem-specific function of execution time, memory requirements, implementation costs, maintenance costs, hardware requirements, potential for reuse, portability and scalability. Such design optimization involves tradeoffs between simplicity, performance, portability, and other factors.

More specifically, a performance analyst might be involved in any of the following tasks: specifying performance requirements, comparing two or more systems with each other, determining the optimal value of a parameter (system tuning), finding the performance bottleneck, characterising the load on the system, and predicting the performance at future loads. In our case, performance evaluation is concerned with the description, analysis and prediction of the dynamic behaviour of compute and communication systems of a parallel platform. The aim is to understand the behaviour of the system and identify the aspects of the system which are sensitive from a performance point of view.

In the rest of this section we are concerned with the modelling and measurement of just two aspects of algorithm performance: execution time and parallel scalability. We focus on these issues because they are frequently among the most problematic aspects of parallel program design and because they are the most easily formalized in mathematical models. These models can be used to compare the efficiency of different algorithms, to evaluate scalability, and to identify bottlenecks and other inefficiencies. Some of them *before* we invest substantial effort in an implementation, some of them can be used *after* the implementation of a parallel program. Performance models can also be used to guide implementation efforts by showing where optimization is needed.

4.3 The need for performance modelling

Modelling the performance of a parallel program could eventually lead to many conveniences as far as time and money cost are concerned. First of all, utilization of the available resources for a specific application the best way possible, that is, using as less machines as possible but achieving the highest speedup. Afterwards, scheduling of the jobs submitted in order for each one of them to achieve the best possible performance, by using as few hardware resources as possible, the least time. Another example would be designing a model that would predict the performance of an application in all execution environments, even before

designing a specific architecture, or by using each specific machine the least possible time e.g. taking measurements that would outline factors of the machine's architecture that could affect a parallel program's performance.

There are more examples that describe parallel programs' performance prediction need. We discuss the job scheduling problem and try to solve the problem of utilization of the available resources for a specific application the best way possible by presenting our model in Chapter 5.

Parallel systems, such as supercomputers, are valuable resources that are commonly shared by communities of users. Users continually submit jobs to the system, each with unique resource and service-level requirements as well as value to the user and resource owner. The charge of job scheduling is to decide when and how each job should execute in order to maximize the system's aggregate utility to its owners.[19] In other words, job scheduling is a discipline whose purpose is to decide when and where each job should be executed from the perspective of the system.

Scheduling is an inherently reactive discipline, mirroring trends in HPC architectures, parallel programming language models, user demographics, and administrator priorities. No scheduling strategy is optimal for all of today's scenarios, let alone all of tomorrow's.

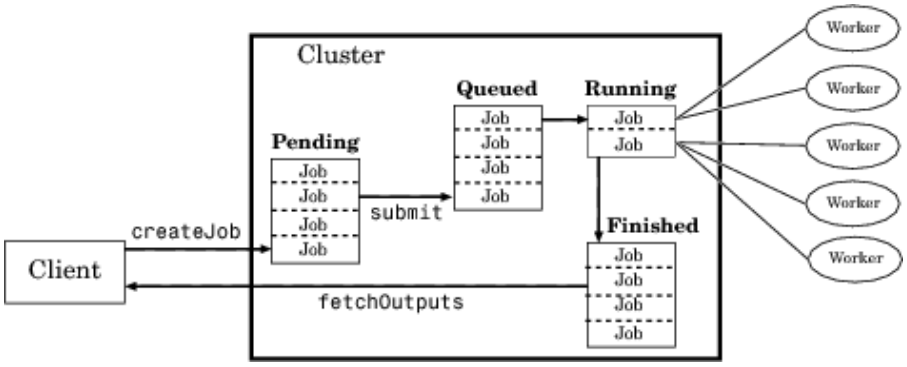


Figure 4.1 Overview of the job scheduling problem

For such a well studied problem, the evaluation of scheduling algorithms has proven surprisingly elusive. The objective of scheduling, and of system administration in general, is to maximize system utility. Unfortunately, utility is not directly observable, but rather is some subjective and context-specific function of many factors. Should one job starve to decrease the running time of five others by 10%? How much more productive will a user be if he knew exactly when his job will run? What if he knew within 20%? These determinations must be informed by context well outside the scope of the scheduler and even then, objectivity is impossible.

We can identify certain qualities as desirable for schedulers, including *performance*, *fairness*, and *predictability*. Because these qualities are largely intangible, many observable metrics intended to mirror them, particularly performance, have emerged and enjoyed wide use.

Predictability is the gap between a job's response or flow time and the user's expectation as created through previous experience. Predictability can indirectly increase productivity by enabling users to anticipate job completion times and plan resource usage accordingly. Some have proposed that predictability, under other realistic assumptions, may be even more central to the user experience than performance.[18]

4.4 When Auto-tuning comes into play

The complexity and diversity of today's parallel architectures overly burdens application programmers in porting and tuning their code. At the very high end, processor utilization is notoriously low, and the high cost of wasting these precious resources motivates application programmers to devote significant time and energy to tuning their codes. This tuning process must be largely repeated to move from one architecture to another, as too often, a code that performs well on one architecture faces bottlenecks on another. As we are entering the era of petascale systems, the challenges facing application programmers in obtaining acceptable performance on their codes will only grow, since there are too many complex architectures with too many possible code transformations to hand optimize every kernel for every architecture. Therefore, we need a general, automated solution.

Auto-tuning provides performance portability across the breadth and evolution of multicore architectures. To assist the application programmer in managing this complexity, much research in the last few decades has been devoted to developing auto-tuning software that employs empirical techniques to evaluate a set of alternative mappings of computation kernels to an architecture and select the mapping that obtains the best performance. Auto-tuning software can be grouped into three categories:

- compiler-based auto-tuners that automatically generate and search a set of alternative implementations of a computation
- application-level auto-tuners that automate empirical search across a set of parameter values proposed by the application programmer
- run-time auto-tuners that automate on-the-fly adaptation of application-level and code-transformation level parameters to react to the changing conditions of the system that executes the application

What is common across all these different categories of auto-tuners is the need to search a range of possible implementations to identify one that performs comparably to the best-performing solution. The resulting search space of alternative implementations can be prohibitively large. Often, an exhaustive search is intractable. Therefore, a key challenge that faces auto-tuners, especially as we expand the scope of their capabilities, involves scalable search among alternative implementations. The trend is to use heuristics to guide the search. The future is to use performance models to guide the search.

The main differences between the compilers and the auto-tuners, could be summarized in these facts:

- Auto-tuners are dataset aware, where compilers are oblivious.
- Auto-tuners are motif-oriented, not code-oriented.
- Auto-tuners can change the data structures, loop structures or even the algorithm at runtime to achieve better performance

4.5 Modelling execution time

Before we examine performance models, we should gain a better understanding of what is meant by some concepts used in parallel computing.

Speedup: The speed of a program is the time it takes the program to excecute. Speedup is defined as the time it takes a program to execute in serial (with one processor) divided by the time it takes to execute in parallel (with many processors). The formula for speedup is:

$$S = \frac{T(1)}{T(j)}$$

Where T(j) is the time it takes to execute the program when using j processors.
When computing speedup, the best serial algorithm and fastest serial code must be compared. Frequently, a less than optimal serial algorithm will be easier to parallelize. Even in such a case, it is unlikely that anyone would use serial code when a faster serial version exists. Thus, even though the underlying algorithms are different, the best serial run time from the fastest serial code must be used to compute the speedup for a comparable parallel application.[15]

Efficiency E=S/p, where S is the speedup observed when using p processors. Execution time is not always the most convenient metric by which to evaluate parallel algorithm performance. As execution time tends to vary with problem size, execution times must be normalized when comparing algorithm performance at different problem sizes. Efficiency is a related metric that can sometimes provide a more convenient measure of parallel algorithm quality. It characterizes the effectiveness with which an algorithm uses the computational resources of a parallel computer in a way that is independent of problem size, or, in other words, it represents the fraction of time that processors spend doing useful work. Typically $E \leq 1$, othewise we have superlinear speedup which should be closely remarked.

Scalability refers to a parallel system's ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. There are two common notions of scalability. The first is strong scaling, which is defined as how the solution time varies with the number of processors for a fixed total problem size. The second is weak scaling, which is defined as how the solution time varies with the number of processors for a fixed problem

size per processor.

Latency is a measure of time delay experienced in a system, the precise definition of which depends on the system and the time being measured.

Bandwidth is a term used to refer to the amount of data that can be transmitted in a fixed amount of time, representing the available or consumed data communication resources expressed in bits/second or multiples of it (kilobits/s, megabits/s etc.). For digital devices, the bandwidth is usually expressed in bits per second (bps) or bytes per second. For analog devices, the bandwidth is expressed in cycles per second, or Hertz (Hz). The bandwidth is particularly important for I/O devices. For example, a fast disk drive can be hampered by a bus with a low bandwidth.

Parallel overhead is the amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as task start-up time, synchronizations, data communications, software overhead imposed by parallel compilers, libraries, tools, operating system, etc, task termination time.

We define the execution time of a parallel program as the time that elapses from when the first processor starts executing on the problem to when the last processor completes execution. During execution, processor i is either computing ($T_{i,comp}$), communicating ($T_{i,comm}$), or idling ($T_{i,idle}$). Hence, total execution time T can be defined in two ways: as the sum of computation, communication, and idle times on an arbitrary processor j,

$$T = T_{j,comp} + T_{j,comm} + T_{j,idle}$$

or as the sum of these times over all processors divided by the number of processors P,

$$T = \frac{1}{P}(T_{j,comp} + T_{j,comm} + T_{j,idle})$$

in order to find the mean value of total execution time.

4.5.1 Computation time

The computation time of an algorithm is the time spent performing computation rather than communicating or idling. If we have a sequential program that performs the same computation as the parallel algorithm, we can determine by timing that program. Otherwise, we may have to implement key kernels. Computation time will normally depend on some measure of problem size, whether that size is represented by a single parameter N or by a set of parameters N1 , N2... If the parallel algorithm replicates computation, then computation time will also depend on the number of tasks or processors. In a heterogeneous parallel computer (like the cluster we are running our programs to), computation time can vary according to the processor on which computation is performed.

Computation time will also depend on characteristics of processors and their memory systems. For example, scaling problem size or number of processors can change cache performance or the effectiveness of processor pipelining. As a consequence, one cannot automatically assume that total computation time will stay constant as the number of processors changes.

4.5.2 Communication Time

The communication time of an algorithm (T_{comm}) is the time that its tasks spend sending and receiving messages. Two distinct types of communication can be distinguished: interprocessor communication and intraprocessor communication. In interprocessor communication, two communicating tasks are located on different processors. This will always be the case if an algorithm creates one task per processor. In intraprocessor communication, two communicating tasks are located on the same processor.

4.5.3 Idle Time

Both computation and communication times are specified explicitly in a parallel algorithm; hence, it is generally straightforward to determine their contribution to execution time. Idle time (T_{idle}) can be more difficult to determine, however, since it often depends on the order in which operations are performed.

A processor may be idle due to lack of computation or lack of data. In the first case, idle time may be avoided by using load-balancing techniques. In the second case, the processor is idle while the computation and communication required to generate remote data are performed. This idle time can sometimes be avoided by structuring a program so that processors perform other computation or communication while waiting for remote data. This technique is referred to as overlapping computation and communication, since local computation is performed concurrently with remote communication and computation. Such overlapping can be achieved in two ways. A simple approach is to create multiple tasks on each processor. When one task blocks waiting for remote data, execution may be able to switch to another task for which data are already available. This approach has the advantage of simplicity but is efficient only if the cost of scheduling a new task is less than the idle time cost that is avoided. Alternatively, a single task can be structured so that requests for remote data are interleaved explicitly with other computation.

4.6 Performance Models

Before starting a parallelization project, developers may wish to estimate the amount of performance increase (speedup) that they can realize. As mentioned earlier, that is not the case when it comes to parallel programs, meaning that modelling the performance *before* the implementation is a very difficult and often not accurate project. It is easily understood that the need for a simple and accurate form that models the performance of an application in a

multicore system, such as a parallel platform, will be especially needed. “The model need not be perfect, just insightful”. [1]

As far as parallel platforms are concerned, a satisfying model should take into account some parameters of the platform, as well as some parameters of the application and predict what the performance will be. The use of too many details would be impractical and the model would appear to be too complicated and circumvent the first of our requirements, simplicity. On the other hand, had the information included been too poor, the result would be misleading, hence not helpful.

The two traditional performance evaluation techniques are:

- Analytic modelling
- Simulation
- Measurement

We mention briefly the basic chacteristics of each technique.

The *analytic* technique is based on theoretical operation count of the algorithm and it’s implementation. It can be done without the computer system in question, but the accuracy in many cases is low. The main advantage of analytical methods is low-cost. However, constructing analytical models of parallel applications requires a thorough understanding of the algorithms and their implementations. Most of such models are constructed manually by domain experts, which limits their accessibility to normal users. Moreover, a model built for an application cannot be applied to another one. The *queueing theory* predicts the total time the jobs remain in a system. It views the system as a collection of resources for which jobs wait in queues to get access to.

Simulation techniques can capture detailed performance behavior at all levels, and can be used automatically to model a given program. However, an accurate system simulator is extremely expensive, not only in terms of simulation time but especially in their memory requirements. Existing simulators are still inadequate to simulate the very large problems that are of interest to high-end users.

The *measurement based analytic modelling* uses an algebraic timing formula based on theoretical time dependencies. There exist free parameters in the model, which need measurements to be calibrated. It is an often used performance evaluation technique.[17]

4.6.1 Amdahl′s Law

If the percentage of serial code execution that could be executed in parallel is known (or estimated), one can use Amdahls law to compute an upper bound on the speedup of an application without actually writing any concurrent code. Amdahl′s law is a law governing the speedup of using parallel processors on a problem, versus using only one serial processor. It is used to find the maximum expected improvement to an overall system when only part of the system is improved. This is due to the fact that the speedup of a program using multiple

processors in parallel computing is limited by the time needed for the sequential fraction of the program, which means that when parallelizing a kernel, a fraction of it will not be able to be parallelized, and will constitute the bottleneck for the performance.

More specifically, Amdahl′s law states that if P is the proportion of a program that can benefit from parallelization, and (1 − P) is the proportion that cannot be parallelized (remains serial), then the maximum speedup that can be achieved by using N processors is:

$$S(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

As N gets bigger by adding more processors, the maximum speedup tends to 1 / (1 − P). As an example, if P is 90%, then (1-P) is 10%, and the problem can be sped up by a maximum of a factor of 10, no matter how large the value of N used. For this reason, parallel computing is only useful for either small numbers of processors, or problems with very high values of P, the so-called embarassingly parallel problems. Hence, an important result of Amdahl's law is that we should only parallelize sections of the code that occupy the largest percentage of execution time.

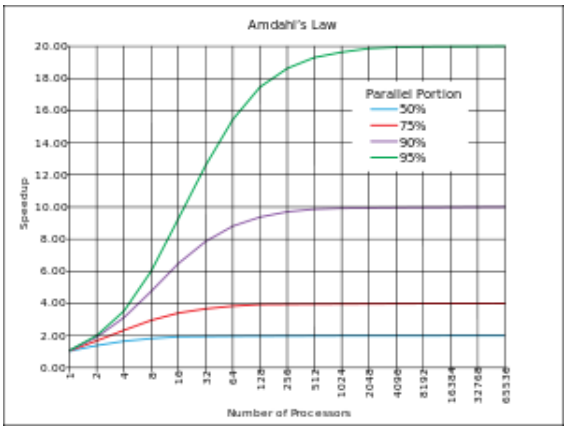


Figure 4.2 Amdahl′s law

P can be estimated by using the measured speedup (SU) on a specific number of processors (NP) using

$$Pestimated = \frac{\frac{1}{SU}-1}{\frac{1}{NP}-1}$$

P estimated in this way can then be used in Amdahl′s law to predict speedup for a different number of processors. [14] Another way of estimating, maybe even more accurately, the percentage of the serial code is profiling.

Objections to Amdahl′s Law

In the early days of parallel computing, it was widely believed that the effect of a sequential component eventually limiting the speedup, would limit the utility of parallel computing to a small number of specialized applications – in which parallelising actually made sense e.g. the sequential component would not be dominant. However, practical experience shows that this inherently sequential way of thinking is of little relevance to real problems. To understand why, let us consider a noncomputing problem.

Assume that 999 of 1000 workers on an expressway construction project are idle while a single worker completes a "sequential component" of the project. We would not view this as an inherent attribute of the problem to be solved, but as a failure in management. For example, if the time required for a truck to pour concrete at a single point is a bottleneck, we could argue that the road should be under construction at several points simultaneously. Doing this would undoubtedly introduce some inefficiency - for example, some trucks would have to travel further to get to their point of work - but would allow the entire task to be finished more quickly. Similarly, it appears that *almost all computational problems admit parallel solutions.* The scalability of some solutions may be limited, but this is due to communication costs, idle time, or replicated computation rather than the existence of *sequential components.*

Amdahl′s law has been criticized for ignoring real-world overheads such as communication, synchronization, and other thread management, as well as the assumption of infinite-core processors. In addition to not taking into account the overheads inherent in concurrent algorithms, one of the strongest criticisms of Amdahl′s law is that as the number of cores increases, the amount of data handled is likely to increase as well. Amdahl′s Law assumes a fixed data set size for whatever number of cores is used and that the percentage of overall serial execution time will remain the same.

Amdahl′s law only tells us how much speedup can be achieved - it does not clarify anything about which factors really play an important role in that process. Therefore, it is only of modest practical use. This is also due to the fact that not only the processor's throughput is an important factor in computing - the memory is, too.

4.6.2 Gustafson's Law

In counterpoint to Amdahl's Law is Gustafson's Law.[18] Gustafson observed that rather than trying to reduce the time needed to solve a fixed size problem, parallel processing is generally used to increase the size of the problem that can be solved in a fixed time.

According to Gustafson's law the execution time of the program on a parallel computer is decomposed into:

$$(a + b)$$

where a is the sequential time and b is the parallel time, on any of the P processors. (Overhead is ignored.)

The key assumption of Gustafson is that the total amount of work to be done in parallel varies linearly with the number of processors. Then practical implication is of the single processor being more capable than the single processing assignment to be executed in parallel with (typically similar) other assignments. This implies that b, the per-process parallel time, should be held fixed as P is varied. The corresponding time for sequential processing is $a+P\cdot b$. Speedup is accordingly: $\frac{(a+P\cdot b)}{(a+b)}$

Defining $\alpha = \frac{a}{(a+b)}$ to be the sequential fraction of the parallel execution time, we have

$$S(P) = \alpha + P \cdot (1 - \alpha) = P - \alpha \cdot (P - 1)$$

Thus, if α is small, the speedup is approximately P, as desired. It may even be the case that α diminishes as P (together with the problem size) increases; if that holds true, then S approaches P monotoneous with the growth of P.

Thus Gustafson's law seems to rescue parallel processing from Amdahl's law. It is based on the idea that if the problem size is allowed to grow monotonically with P, then the sequential fraction of the workload would not ultimately come to dominate. This is enabled by means of having most of the assignments individually containable within a single processor's scope of processing; thus a single processor may provide for multiple assignments, while a single assignment shouldn't span more than a single processor. This is also the rule for relating projects with work-sites, having multiple projects per site, but only one site per project.

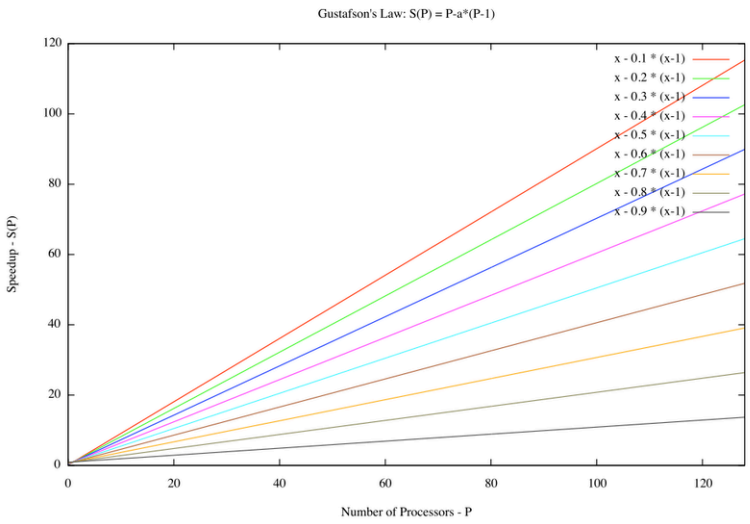


Figure 4.3 Gustafson′s law. Even though parallelism can't reduce the execution time of fixed size problems to an arbitrarily small time, it clearly does permit the solution of larger problems in a fixed time.

Amdahl's law vs Gustafson's law: A Driving Metaphor

Amdahl's Law approximately suggests:

"Suppose a car is traveling between two cities 60 miles apart, and has already spent one hour traveling half the distance at 30 mph. No matter how fast you drive the last half, it is impossible to achieve 90 mph *average* before reaching the second city. Since it has already taken you 1 hour and you only have a distance of 60 miles total; going infinitely fast you would only achieve 60 mph total average speed."

Gustafson's Law approximately states:

"Suppose a car has already been traveling for some time at less than 90mph. Given enough time and distance to travel, the car's average speed can always eventually reach 90mph, no matter how long or how slowly it has already traveled. For example, if the car spent one hour at 30 mph, it could achieve this by driving at 120 mph for two additional hours, or at 150 mph for an hour, and so on."

4.6.3 Roofline model

The roofline model was introduced as an easy to understand performance model, capable of identifying performance bottlenecks. This model gives a rough *computation time* estimate based on the assumption that performance is limited by either peak memory bandwidth or by peak ALU thoughput, unlike Amdahl's law that states that performance is limited by the sequential component of the algorithm. The roofline model is processor specific: it correlates the application with the architecture of the system [3].

This can be explained by the fact that off-chip memory bandwidth is often the constraining resource in system performance. Hence, we want a model that relates processor performance to off-chip memory traffic. Toward this goal, we use the term *operational intensity* which refers to operations per byte of main memory traffic, defining total bytes accessed, as those bytes that go to the main memory after they have been filtered by the cache hierarchy. That is, we measure traffic between the caches and main memory rather than between the processor and the caches. Thus, operational intensity predicts the DRAM bandwidth needed by a kernel on a particular computer, as it is a typical feature of any kernel.

The proposed Roofline model ties together CPU peak performance, operational intensity, and memory bandwidth in a 2D graph.

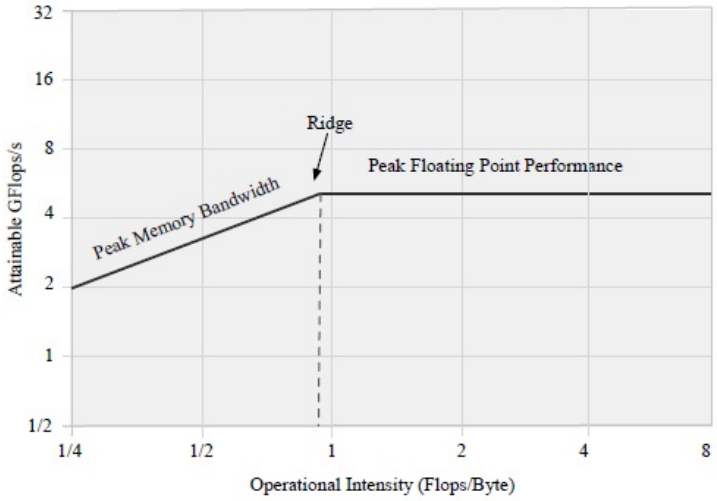


Figure 4.4 Roofline model

The Roofline Model can be explained best by providing an example that can be seen in the Figure above. The x-axis refers to the operational intensity defined above, the y-axis shows the peak floating point performance. The horizontal line to the right is the maximum floating point performance for a given system. Obviously, the overall maximum performance can never be increased above that line – for any operational intensity – since this line indicates the hardware limit. This is the Roofline. The slanted line to the left is the maximum memory performance the system can deliver under a given operational intensity. It shows that a system can be slowed down by the memory architecture. This conforms to the principle of memory hierarchy. If all computation were to take place in higher level caches, processing would be faster, but again, access to memory would take longer.

Essentially, the Roofline model can be condensed to the formula:

$$AttainableGFlops/s = min(\frac{PeakFloatingPointPerformance}{PeakMemoryBandwidth \cdot OperationalIntensity})$$

The ridge point marks an important factor in determining the potential of a system and offers insight into the computer's overall performance. Its position on the x-axis shows how operationally intensive a problem has to be to achieve the maximum performance. In other words, the x-coordinate of the point is the minimum operational intensity required to achieve maximum performance. If the ridge point is far to the right, then only kernels with very high operational intensity can achieve the maximum performance of that computer. If it is far to the left, then almost any kernel can potentially hit maximum performance. [1] In essence, the ridge point is a hint for programmers that shows how difficult it is to optimize the system.

As a concrete example, we could imagine that a kernel A has an operational intensity of 0.8 Flops/Byte, whereas another kernel B could use 8 Flops/Byte. The dashed line marks the operational intensity of this kernel A. As it "hits the roof" exactly at the ridge point, the maximum possible speed for this operation is bound by the peak floating point performance of the CPU, and not the memory. If the kernel had a lower operational intensity, the memory would impose limits on the maximum speed and vice-versa.

Peak CPU performance and memory bandwidth is collected via micro benchmarks. Operational intensity is derived from manual computation (pencil and paper) and depends on whether the working set we consider fits fully in on-chip caches, or not.

The roofline model can undergo many improvements, adding more ceilings that lead to more accurate prediction of the performance of the kernel on a specific multi-core system.

To sum up, roofline sets an upper bound on performance of a kernel depending on the kernel's operational intensity. If the operational intensity is relatively big, the column that represents it hits the flat part of the roof, and performance is compute-bound, otherwise it hits the slanted part of the roof, which means performance is ultimately memory- bound. What is important is that Roofline model will be unique to each architecture and so will the coordinates of a kernel.

The computational performance of an algorithm can either be *compute-bound*, where there are a large number of computations per data element, or *memory-bound*, where there are many data elements per computation. Memory bound, more specifically, refers to a situation in which the time to complete a given computational problem is decided primarily by the amount of available memory to hold data. In other words, the limiting factor of solving a given problem is the memory access speed. For both approaches, determining the maximum allowable amount of resources provided is key for optimization.

4.6.4 Hockney model

The Hockney's model is one of the most meaningful model to describe point-to-point communication on parallel machines with distributed memory. The communication time between two nodes in a distributed-memory platform can be decomposed into the sum of the time to prepare a message for transmission and the time taken by the message to traverse the network to its destination. The principal parameters that determine the communication latency are as follows:

- Startup time (t_s): The startup time is the time required to handle a message at the sending and receiving nodes. This includes the time to copy the data from user space to the communication engine, prepare the message (header, trailer, error correction, etc.), execute the routing algorithm and establish the interface between the sender and re-

ceiver nodes. In other words, startup time is the time needed to send a 0-byte message. This delay is incurred once per message.

- Per-hop time (t_h): After a message leaves a node, it takes a finite amount of time to reach the next node in its path within the communication network. The time taken by the header of a message to travel between two directly-connected nodes in the network is called the per-hop time. The per-hop time is related to the latency incurred by the interconnect's hardware.
- Per-word transfer time (t_w): This time is related to the channel bandwidth of the interconnection network (b). Each word takes $t_w = \frac{1}{b}$ to traverse the link.

The communication cost for a message of size m (bytes) transferred between two nodes that are l hops away is:

$$T_{comm} = t_s + lt_h + mt_w$$

mt_w or $\frac{m}{b}$ represents the transmission delay in passing an m-byte message through a network with an asymptotic bandwidth of b Mbytes/s.

In general, it is very difficult for the programmer to consider the effect of the per-hop time. Many message passing libraries like MPI offer little control to the programmer on the mapping of processes to physical processors. Even if this control was granted to the programmer, it would be quite cumbersome to include such parameters in the design and implementation of the algorithm. On the other hand, several network architectures rely on routing mechanisms that include a constant number of steps. This means that the effect of the per-hop time can be included in t_s. Finally, in typical cases it holds $t_s \gg t_h$ or m $t_w \gg t_h$ and since l can be relatively small, the effect of the per-hop time can be ignored, leading to this simplified model for the communication cost of a single message:

$$T_{comm} = t_s + mt_w$$

Many communication scenarios in real-life applications involve collective communication including multiple nodes. Broadcast (one-to-all) and reduction (all-to-one) operations are based on point-to-point communication, but are implemented organizing the participating nodes in a tree or other topologies and utilizing concurrent point-to-point communication between the processes. Thus, in this case the communication cost is modelled as:

$$T_{comm} = (t_s + mt_w) \log p$$

The question is, how are we able to determine the startup time and the bandwidth of the interconnection network? There are numerous benchmarks that are introduced in order to approach these quantities.

Modern high-end machines feature multiple processor packages, each of which contains multiple independent cores and integrated memory controllers connected directly to dedicated

physical RAM. These packages are connected via a shared bus, creating a system with a heterogeneous memory hierarchy. Since this shared bus has less bandwidth than the sum of the links to memory, aggregate memory bandwidth is higher when parallel threads all access memory local to their processor package than when they access memory attached to a remote package. [13]

Running the STREAM benchmark in our cluster, gave us the result that the bandwidth of our platform is approximately 5.8 GB/s. Thus, communication will be estimated by using

$$Tcomm = t_s + m * (1/5.8 * 10^6),$$

where m is the message size in bytes.

Drawbacks of the Hockney model

Though the Hockney model can be very accurate, there are certain parameters which are not included in it.

- The time required to send a message from one processor to another is independent of both processor location and the number of other processors that may be communicating at the same time.
- While accurate for many algorithms and on many architectures, this model can break down if a computer's interconnection network has properties different from the ideal, particularly if an application generates many messages.

Other models are also used in modelling the communication among processes, like LogP, PlogP e.t.c. which can be more accurate than Hockney model - depending on the nature of the problem - but are not mentioned in this diploma thesis.

Chapter 5

A model based on experimentation

This model better fits the characteristics of a *measurement based analytic* model. It considers the total of the processors in our cluster as a search space, and by taking certain measurements that outline not only the architecture of the system, but the main characteristics of the algorithm as well, attempts to predict the region in the search space where the speedup is maximized. The effort towards predicting separately computation and communication time of a kernel also leads to conclusions about how these two main factors of the performance of a parallel program can be optimized and thus lead to its auto-tuning.

The element that was observed while experimenting on modelling the performance of Heat2D application, was that the computation time had a common behaviour among iterations with close working set sizes. Thus, when predicting the computation time of an application, that is memory bound like the Heat equation computational kernel, there is no need to have information about a specific working set, as long as we have sufficient information about the group it belongs to. The information we will be needing is discussed later in this chapter. This method lead us to the assertion that what is important in predicting computation time, is the working set size to cache size ratio. 'The bigger it gets, the lower the speedup is', could become a rule.

In order to approach communication time, we use a kernel specific benchmark. That is, we try to model the main communication pattern of each kernel by using certain simpler MPI functions, like *blocking sends* and *blocking receives* or the *unblocking* ones, and then calculate how many of these operations occur in the kernel. What is important to keep in mind is whether communication occurs between processes in the same node or not, as well as what percentage of the messages are sent in parallel.

5.1 Computation Time

Our main effort was towards understanding the way each one of the aforementioned groups scales when we either add mpi processes when running in one clone, or increase the number of clones used (with one mpi process in each clone) in order to gain a clearer insight

on the effect of the interconnection network. In order to understand the way we will predict the performance, we present in a graph the ensemble of the preceding combinations.

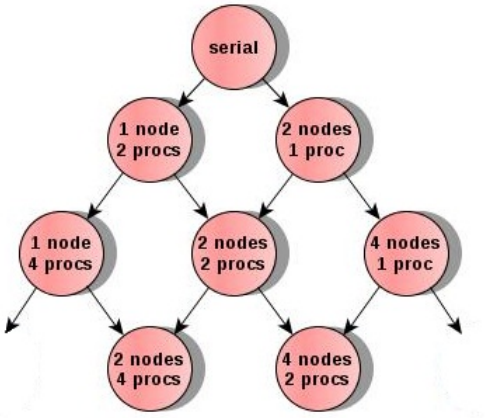


Figure 4.4 Overview of the combinations

When adding a second mpi process in a clone e.g. when moving from node 2 nodes / 1 MPI process to node 2 nodes / 2 MPI processes, the second package of cores is used, thus the L2 cache size is doubled, though the workspace size used in each clone remains the same. Instinctively, we are lead to believe that the performance's behaviour will change.

Experiments showed that seperating working sets' measurements into four groups, would result in having the supervision of computation time for every input size. Thus, we measured the speedup of computation time when processes are placed in one clone, for working set sizes that are smaller, equal, larger than cache size, as well as working sets that are very large to fit to the cache. The expected results are shown in Figure 4.5. It is important to mention again that these are the expected results for a memory bound application, thus similar results for other kinds of applications would lead to a different supervision of the problem.

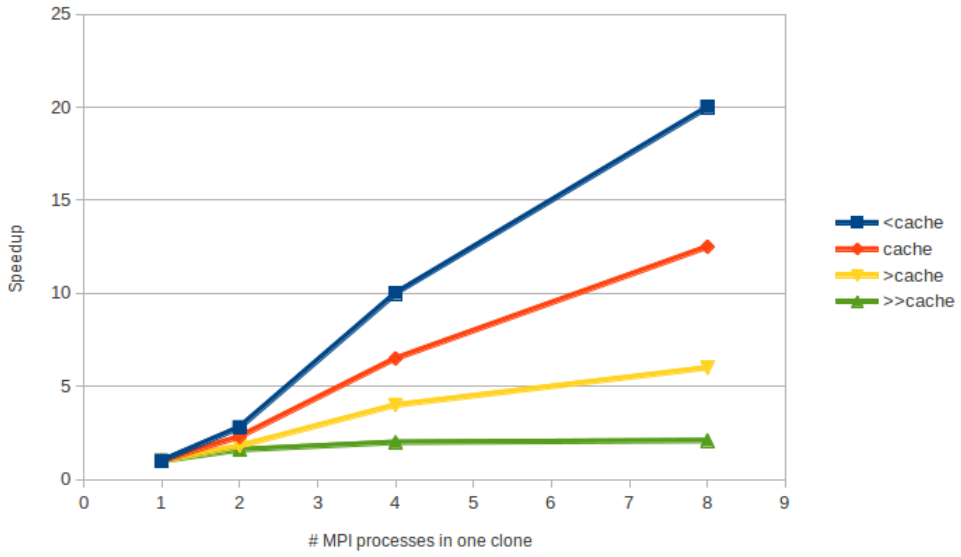


Figure 4.5 Expected results, for memory bound kernels, when measuring in one clone for working set sizes that have a certain relationship to the cache size

It is notable how speedup is expected to raise faster as working set size becomes smaller, or in other words, as the working set to cache size ratio is reduced.

First group consists of working sets that are smaller than cache size, which means the above mentioned ratio is below 1. The next one, groups working sets whose size is almost equal to the cache's size. Third group is about working sets that are very large to fit to the cache. Experiments showed that the ratio in this group is estimated between 1.3 and 2.6, which means that the last group gathers working sets with a ratio of 2.6 or more.

In order to initialise the graph and begin calculating the speedup of *inside* nodes, we also need measurements in which we geminate the number of clones and keep one process in each clone. The results are not similar to the above ones, since interconnection network's effect comes in, and also there is no expected form of them, apart from the fact that we expect the speedup to raise as we add nodes.

As mentioned earlier, the architecture of the system allows us to have up to 8 mpi processes in the same clone, and the number of clones available is 16. Hence, our measurements will be a total of 28 for computation time alone.

We claim that these measurements alone, can lead us to a safe conclusion on which combinations of number of clones and mpi processes would result in the best performance of the

application. It is of absolute importance that we take into consideration similar results concerning communication time, which is discussed afterwards.

Our main assertion is that in each of the nodes we can assume the new speedup with the use of the speedup that the neighborhood clones have as well as our 'leading' measurements. After experimenting on how to combine our data, in order to approach as much as possible the real speedup of a node, we reached the conclusion that a node should only use it's right father's value of speedup. Keeping the same working set size in each clone and geminating the number of mpi processes allows us to predict more accurately the new speedup due to the fact that we already have this type of measurements as input, whereas the implied rule that would have us consider that doubling the number of cores with the same number of mpi processes in each one - which means using the left father's speedup to predict the child's speedup - would lead to a multiplied by a factor of 2 speedup, did not seem to apply in this situation.

The abovementioned rule could be summarized in

$$S_{2n} = S(r) * \frac{f(2n)}{f(n)}$$

where S(r) stands for the right father's speedup, and f(x) is the function whose result when called, is the value of speedup when there are x mpi processes in a clone, given the fact that we have already determined the group in which the node belongs. The algorithm we used to predict computation time is presented in Pseudocode 4.1.

Pseudocode 4.1 Algorithm used for computation time prediction in each node

```
initialization_(); // we initialize the outer nodes
                   //of the graph with our measurements

for(every inside node) {
        ratio = ws (MB) / cache size;
                   //determining the group each node belongs to
}

for (every inside node){

        // compute transition from a node with
        // n processes to a node with 2n processes
        // for every node's group

        transition = measured(2n) / measured(n);

        speedup = right_fathers_speedup * transition ;
}
```

5.2 Communication Time

In comparison to the aforementioned models, we try to develop a more detailed model of communication performance that takes into account the impact of *competition for bandwidth* on communication costs. Two processors may need to send data over the same wire at the same time. Typically, only one message can be transmitted simultaneously, so the other message will be delayed. However, for many practical purposes it suffices to think of the two processors as sharing the available bandwidth.

What was observed is that when processes are placed in the same node, the traffic that occurs causes a greater time delay, than when processes are placed in separate nodes, due to differences between the interconnection network and bus bandwidth. As the number of processes increases, these differences become more obvious. In Figure 4.7 we present this specific fact. We also present the way processes are placed among the nodes in Figure 4.8, using graph nodes as MPI processes.

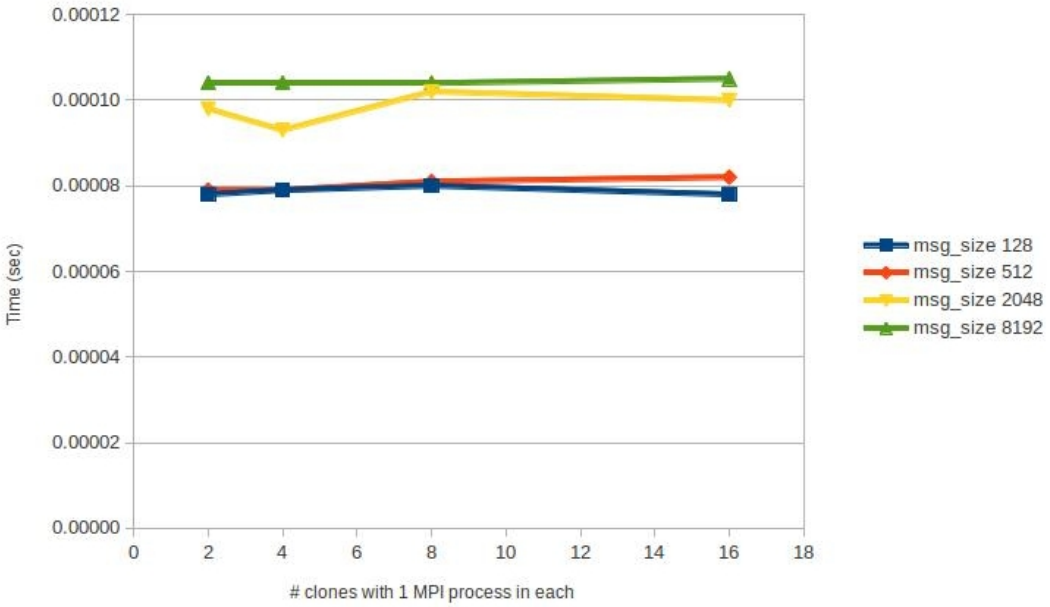


Figure 4.7 Processors sharing the available interconnection network bandwidth

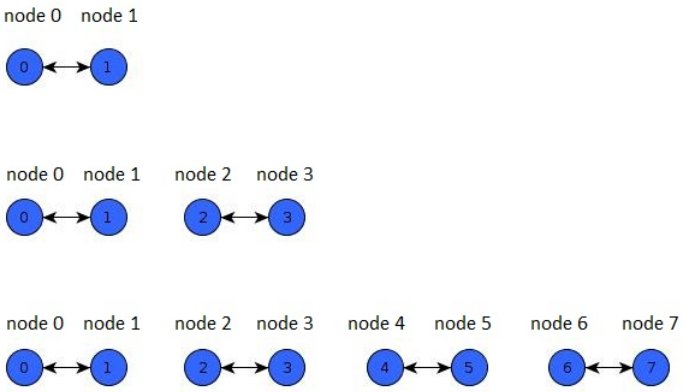


Figure 4.8 Placement of the processes among the available nodes

As the number of processes and clones will increase the interconnection network will probably not continue having this behaviour, and the time needed will increase as well, though with a small rate. Thus, in order to keep the simplicity this model assumes that the network's behaviour will be the same as we add processes and clones.

Figure 4.7 changes when processes are placed in the same node, as shown in Figure 4.9 (a), where all the processes were placed as in Figure 4.9 (b).

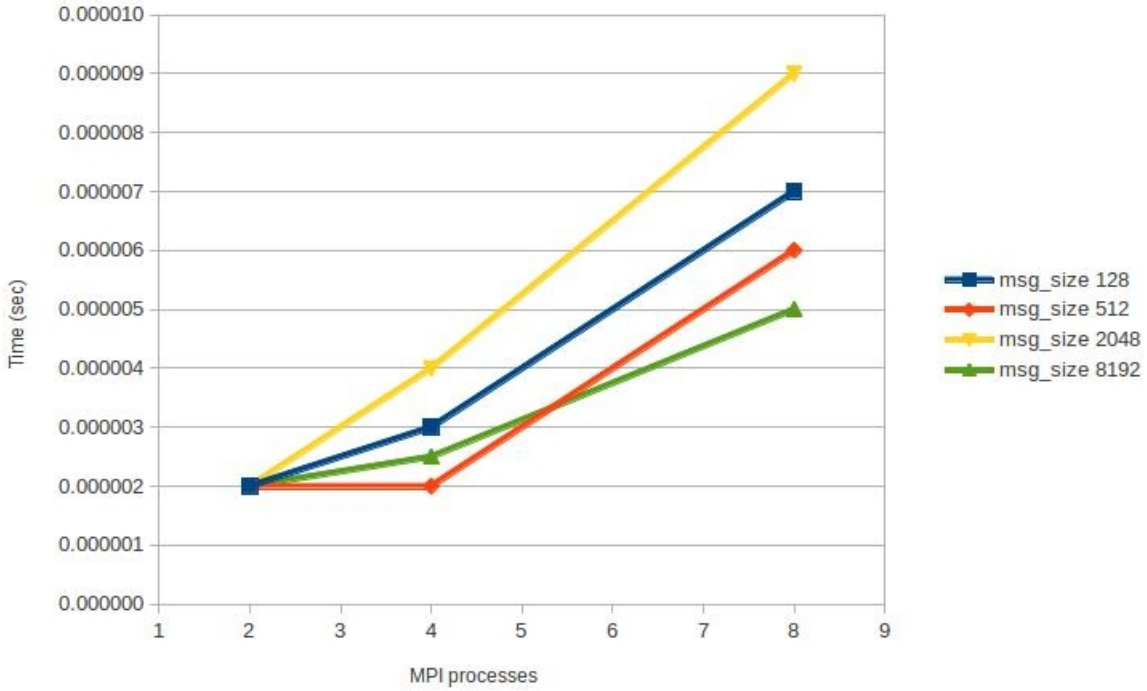


Figure 4.9 (a) Processors sharing the available bus bandwidth (b) Processes communicating inside one node

As far as calculating communication time in each application, what should be noted is the fact that we measure total execution time of the kernel and substract the total computation time. This fact leads us to consider as communication time, the communication along with the idle time of the execution.

Since the benchmark we use is kernel specific, communication model is further explained in each application.

Chapter 6

Applications

6.1 Heat equation 2D

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) of two spatial variables (x, y) and the time variable t, the heat equation is:

$$\frac{\theta u}{\theta t} = \alpha(\frac{\theta^2 u}{\theta x^2} + \frac{\theta^2 u}{\theta y^2}) \qquad (1)$$

One way to numerically solve this equation is to approximate all the derivatives by finite differences [16]. We partition the domain in space using a mesh x = 1, ..., X, y = 1, ..., Y 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, the previous equation becomes:

$$\begin{aligned} u(t,x,y,z) = \alpha(&u(t-1,x,y) + u(t-1,x-1,y) + \\ &u(t-1,x+1,y) + u(t-1,x,y-1) + \\ &u(t-1,x,y+1)) \end{aligned} \qquad (2)$$

Having in mind the latter numerical equation, we would like to compute the values of scalar u after time T in both 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 2D space are: $1 \leq$ x $\leq$ X, $1 \leq$ y $\leq$ Y, Pseudocode 5.1 describes a serial implementation of the algorithm.

Pseudocode 5.1 for the serial implementation of the Heat 2D equation

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

        swap (current, previous) ;
}
```

6.1.1 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 2D space until time T. This problem easily can be generalized to three spatial dimensions. Equation (2) 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 *neighbouring processes* (stencil computation)

and specifically these elements are compulsory to compute boundaries of the portion.

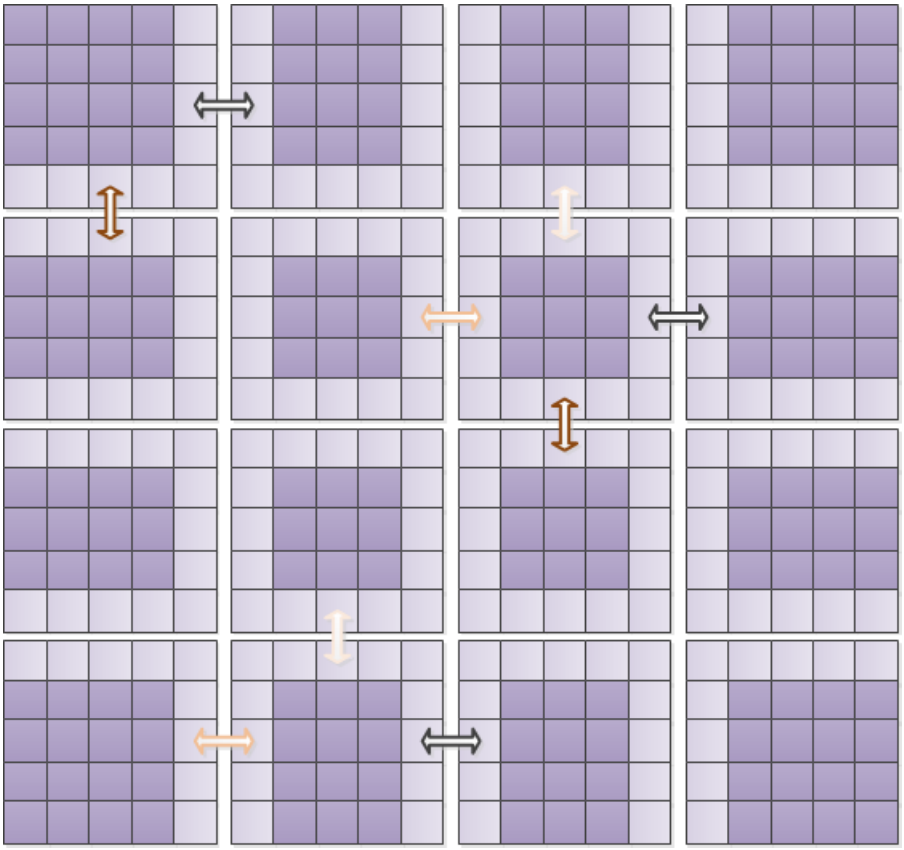


Figure 5.1 Overview of the communication pattern of Heat 2D application

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, as shown in Figure 5.1. Then, every process continues with the computation of its chunk at time t and the whole procedure is repeated until T iterations are made. The parallelization of the kernel is also presented in Pseudocode 5.1.

Pseudocode 5.1 Parallel version of the Heat 2D equation

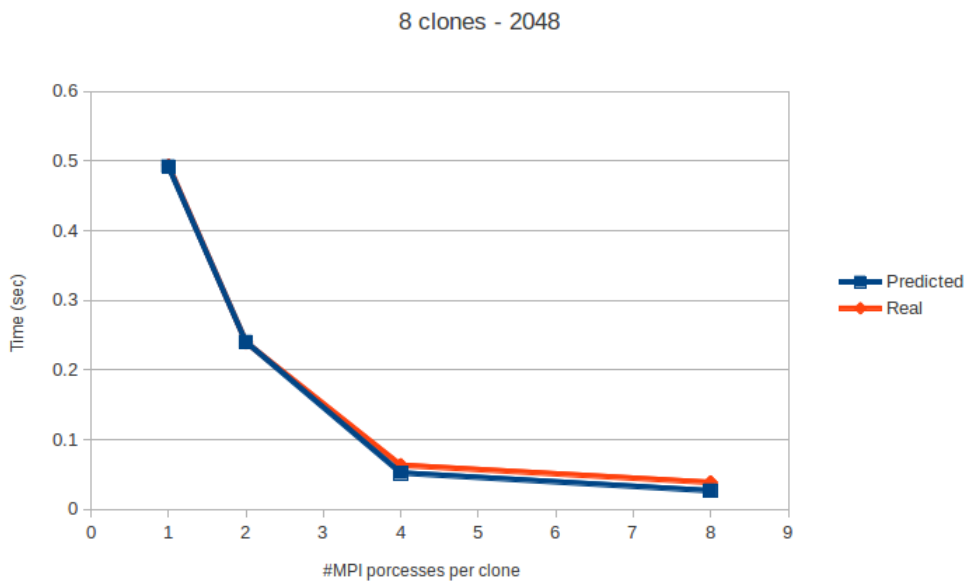
```
if (rank == 0) {
            allocate_full_matrix(A);
            initialize(A);
}

allocate_local_matrix(lA);
```

```
distribute_matrix(A,lA);

left =... //find out ids of neighbors
right =...
up = ...
down = ...

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

// Send and receive data

if (left) {
        Pack data (boundaries of t-1);

        MPI_SendRecv()
        (boundaries of t-1 to and from neighbours);

        Unpack data (boundaries of t-1);
}

if (right) ...

// Computation of time t

for (x = 1;x <= my_X;x++)
        for (y = 1;y <= my_Y;y++)

        u[current][x][y] = a *
                        (u[previous][x][y] +
                        u[previous][x-1][y] +
                        u[previous][x+1][y] +
                        u[previous][x][y-1] +
                        u[previous][x][y+1]);

swap (current, previous);
}
```

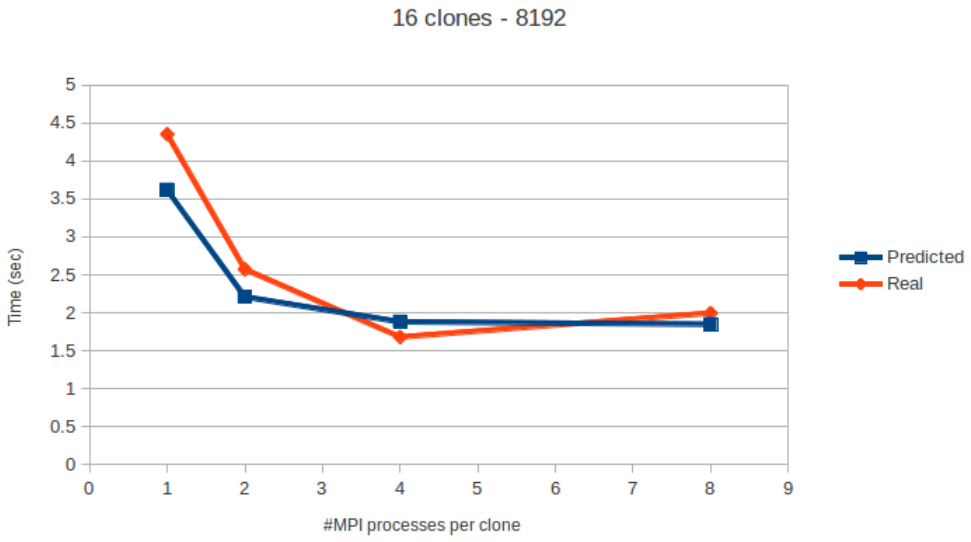
6.1.2 Modelling computation time

We used our main computation model in order to predict the scalability of the computation part of the kernel. The workspaces in which we experimented included multiples of the size of the cache.

We can see in the figures above, that computation time is closely predicted, which was expected, since there is no reason for computation time not to scale normally when processes or nodes are added. Nevertheless, there are some notable deviations in some figures, which can be caused by the differences among the processors of our cluster. Since our model refers to homogeneous systems, it was inevitable for some deviations to occur. We present some examples which represent the best and the worst prediction curves, and belong to different working set sizes.



(a)



(b)

Figure 5.2 (a) Predicted and real computation time for working space 2048 x 2048 when 8 clones are used with 1 - 8 processes in each (b) Predicted and real computation time for working space 8192 x 8192 when 16 clones are used with 1 - 8 processes in each.

6.1.3 Modelling communication time

In order to model communication time, we used a simple *ping - pong* benchmark. It consists of a *sendreceive* operation between 2, 4, 8 or 16 processes in two clones. The processes were placed in a pairwise way so that each pair of processes that performs a sendreceive operation uses the interconnection network e.g. each one of the two processes is placed in a different clone,as shown in figure 5.3 below.

The results of this benchmark were quite expected. Firstly the time cost is logical to grow when the message size augments, and secondly when adding more processes in a clone the cost of communication rises due to traffic problems. Figure 5.4 outlines these results.

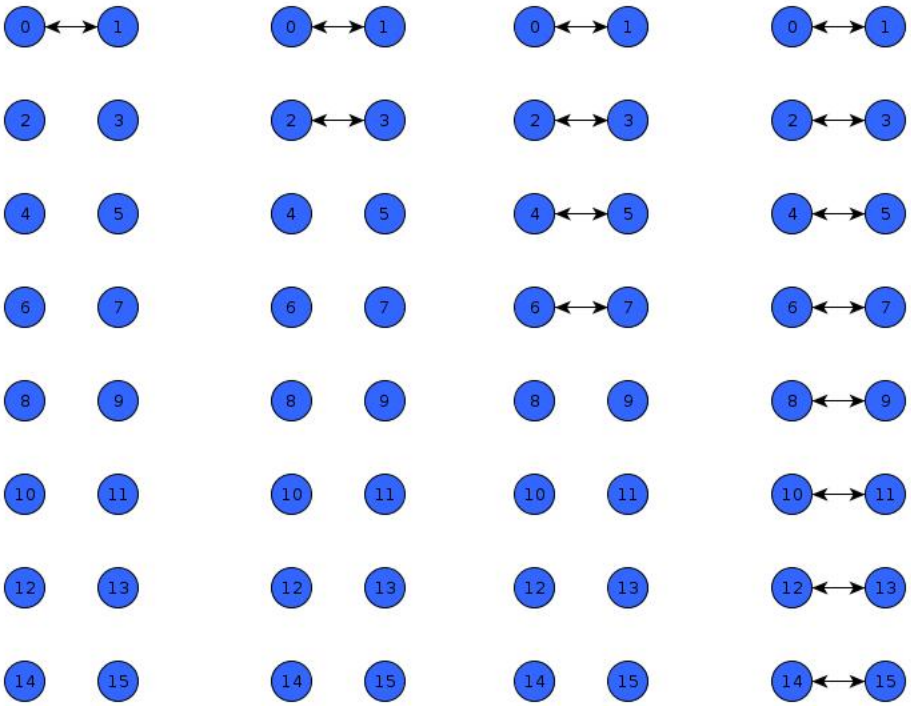


Figure 5.3 Placement of MPI processes in the ping - pong benchmark.

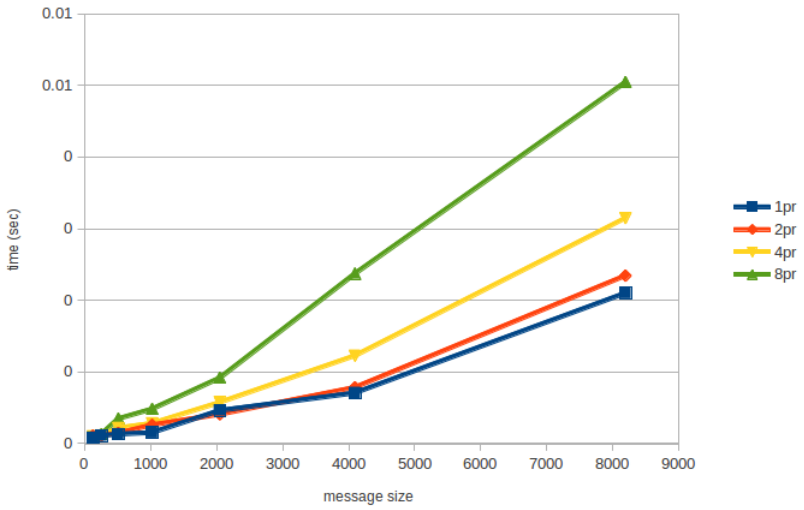


Figure 5.4 Time cost for fixed message sizes and numbers of processes used in the benchmark.

Implementing that benchmark, can lead us to prediction of the communication in every node of our graph. That is because we assumed that certain operations take place at the same

time, as mentioned in Chapter 4 and shown in Figure 4.8. Another example of the communication operations that happen in parallel is shown in Figure 5.5.

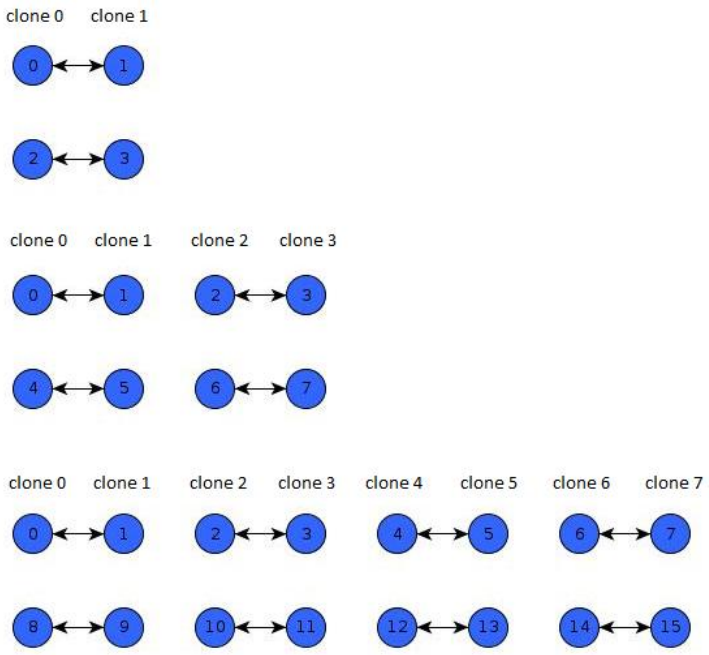
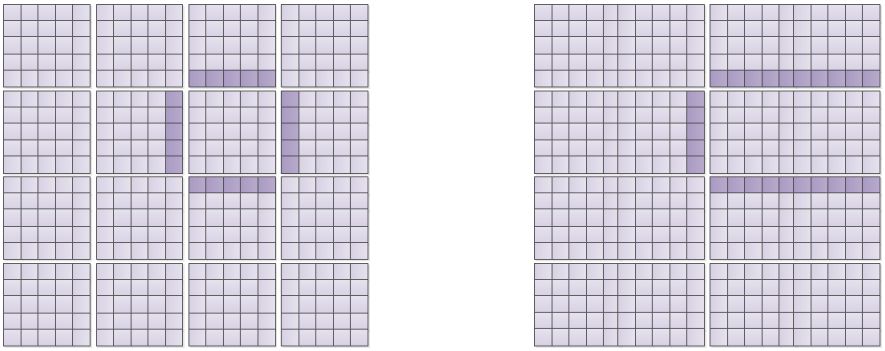


Figure 5.5 Another example of the communication operations that occur in parallel.

Hence, experiments' results denoted that the communication between processes 0 and 1, as well as processes 2 and 3 happens concurrently, and that is also the case for the rest of the combinations presented. Thus, what defines communication time is the number of sendreceives that cannot be parallelized.

What communication pattern of Heat 2D kernel indicates is that each process has to complete the operations with each of the neighbours serially, thus in a quadratic topology of processors, 4 such operations of a specific message size will take place serially. If the topology is not quadratic, the size of the messages will vary according to which neighbour the process is sending a message to. In the following figure, we present the messages that occur serially in each case.



(a) **(b)**

Figure 5.6 Message size when processor topology is (a) quadratic (b) rectangular

While experimenting on approaching the communication time of each combination by calculating how many sendreceives operations and of what message size occured, as well as what percentage of them would take place in parallel, it was clear that the minimum communication time was the one that was approachable. As mentioned before, we incorporate idle time in communication time measurements. In Heat2D kernel, each of the processes, completes the sendreceive operation and carries on to the computation part, for T times. The total time of execution - which we are measuring - does not vary among the processes, due to the barriers used right before we measure. The computation time though, presents some variations due to lack of homogenity in the processors of our cluster. In the next figure we show more precisely what part of the execution time we measure as communication time.

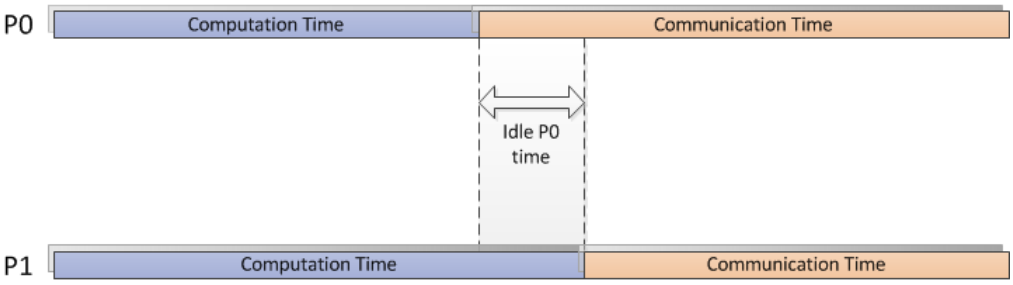
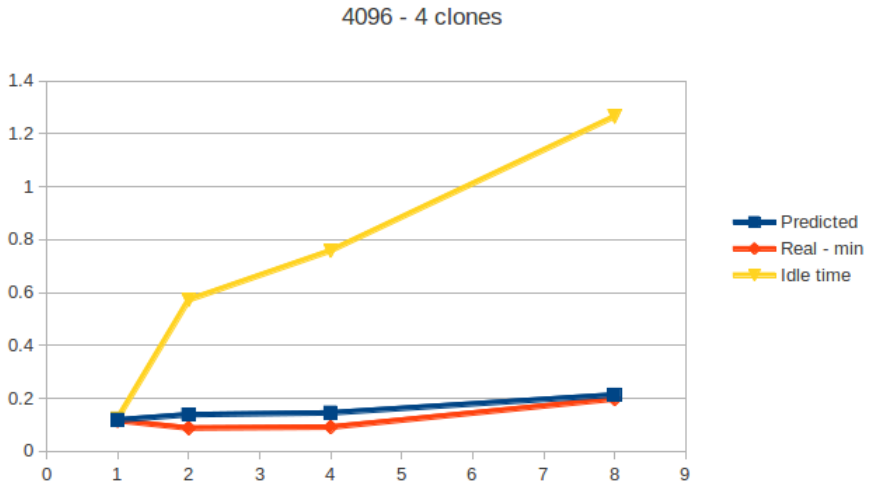


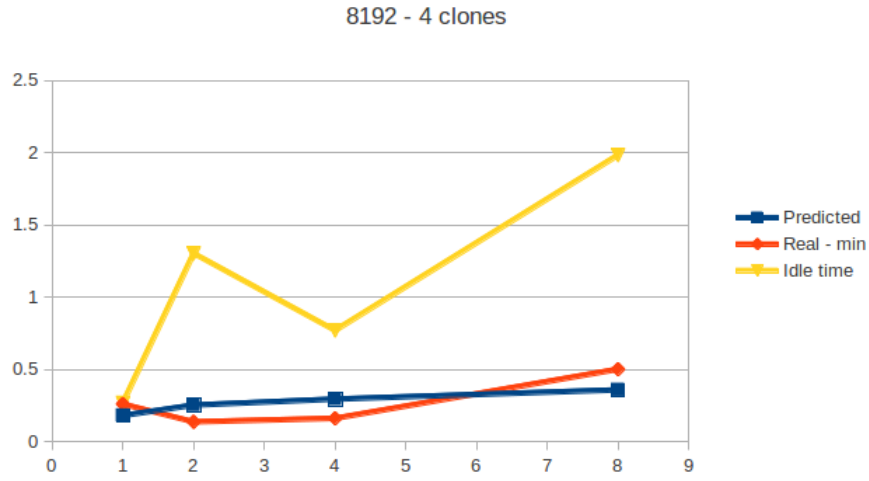
Figure 5.6

Thus, with the use of the communication specific benchmark we can predict the minimum communication time. This fact was made clear, when we measured the cost of the exact communication pattern of the kernel separately, and confirmed that it is equal to the time we predicted.

In the following figures we show the predicted, along with the minimum communication time and the total communication time which incorporates the idle time, for certain input sizes and numbers of processors.



(a)



(b)

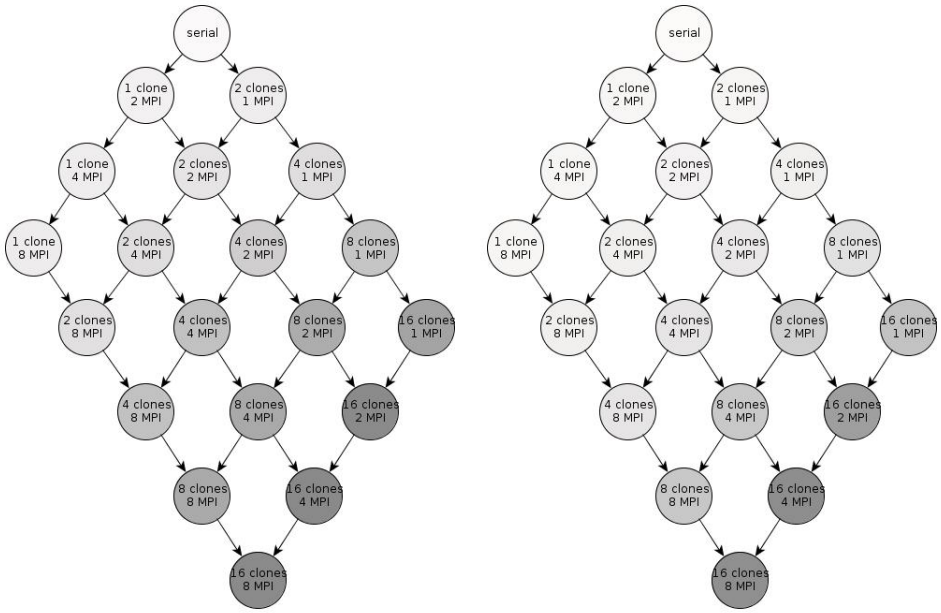
Figure 5.7 Examples of communication time prediction

6.1.4 Presenting total speedup graphs

This model's goal is to predict which nodes of the graph representing our cluster, lead to the highest speedup. Therefore, we add the predicted computation and communication time and produce the total speedup graphs in which every node is coloured in the gray scale, according to its speedup' s declination from the highest one. Comparison with the real speedup graphs will produce the result and determine whether or not the approach of the model would be advisable.

What needs to be mentioned is that in order to produce the graphs we used a different scale, according to the maximum speedup that was either predicted or measured. Thus, when two nodes in different graphs are sharing the same colour, it does not necessarily mean that their speedups were equal, unless the max predicted and max measured speedup were the same.

We present the speedup graphs for three working set sizes. A very large to fit to the cache working set, a quite large for the cache, and the last one which has the size of the cache. Their speedup graphs are shown in Figure 5.8.



(a) (b)

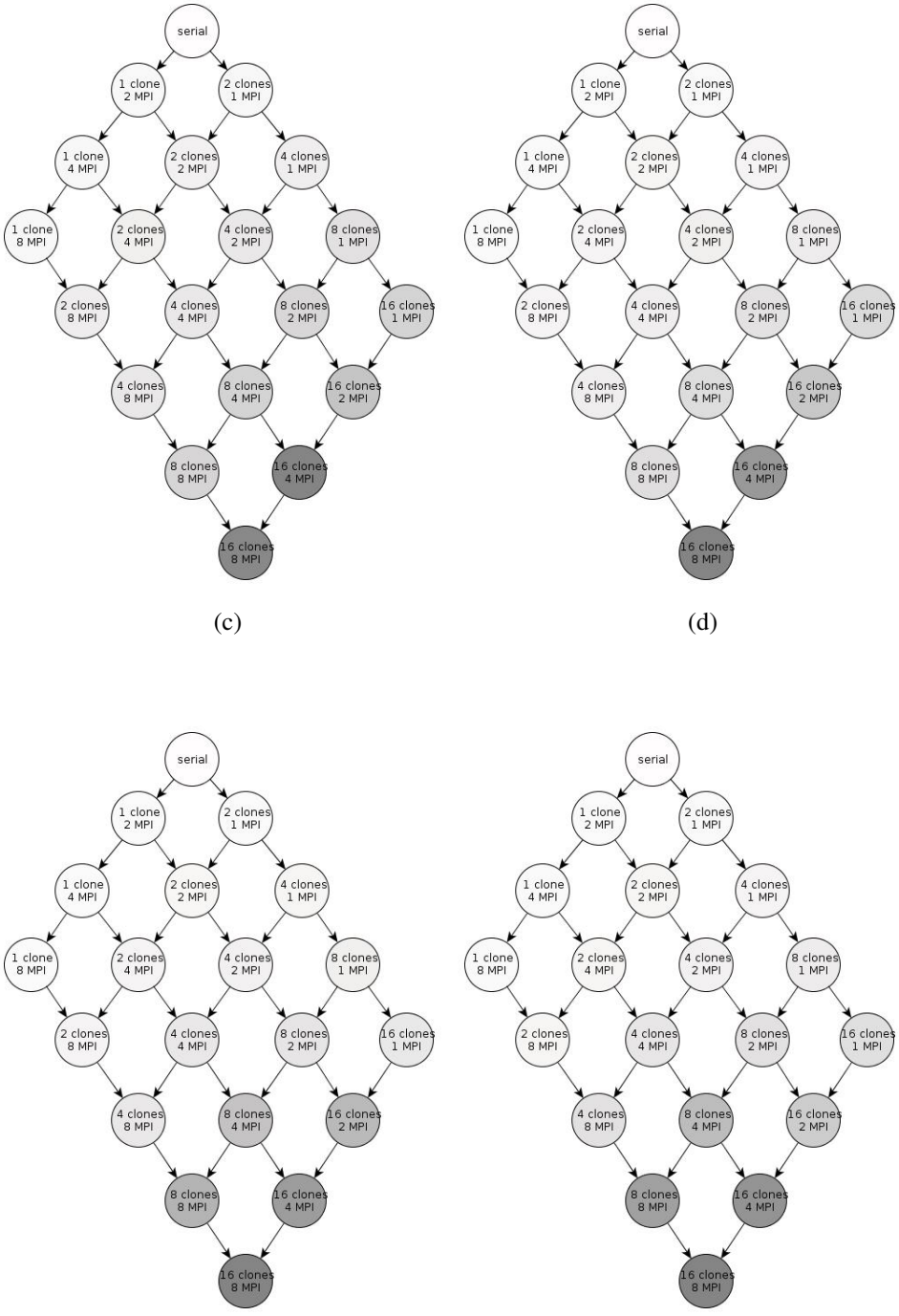
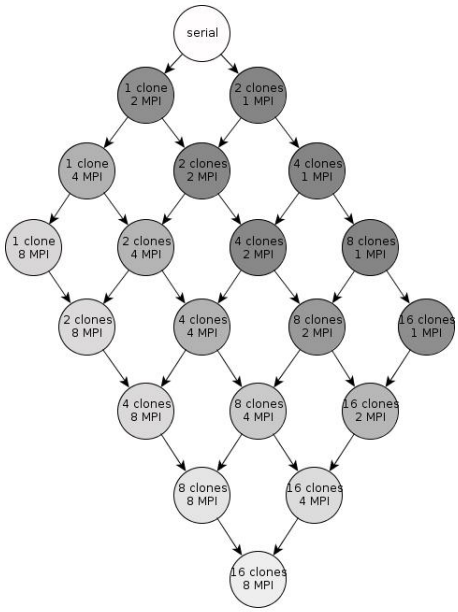


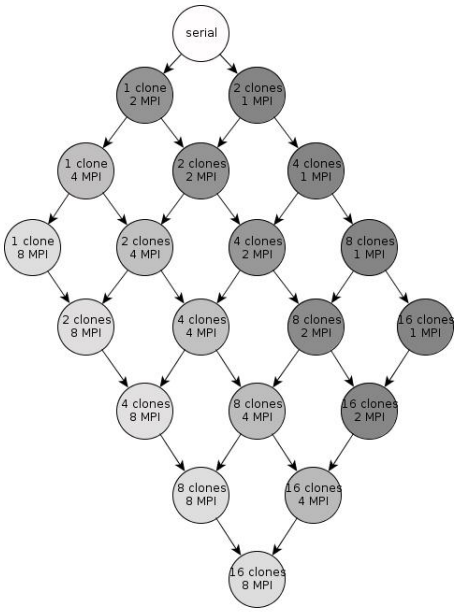
Figure 5.8 Working set size 8192 x 8192 (a)Graph with actual speedup (b)Graph with predicted speedup, Working set size 4096 x 4096 (c)Graph with actual speedup (d)Graph with predicted speedup, Working set size 2048 x 2048 (e)Graph with actual speedup (f)Graph with predicted speedup.

6.1.5 Presenting total efficiency graphs

Since speedup is not always the most convenient metric by which we evaluate parallel algorithm performance, we also produce efficiency graphs, in order to predict the effectiveness with which an algorithm uses the computational resources of a parallel computer, which can sometimes be more important. For the efficiency graphs, the scale we used was maximized at 1, since $E \leq 1$.



(a)



(b)

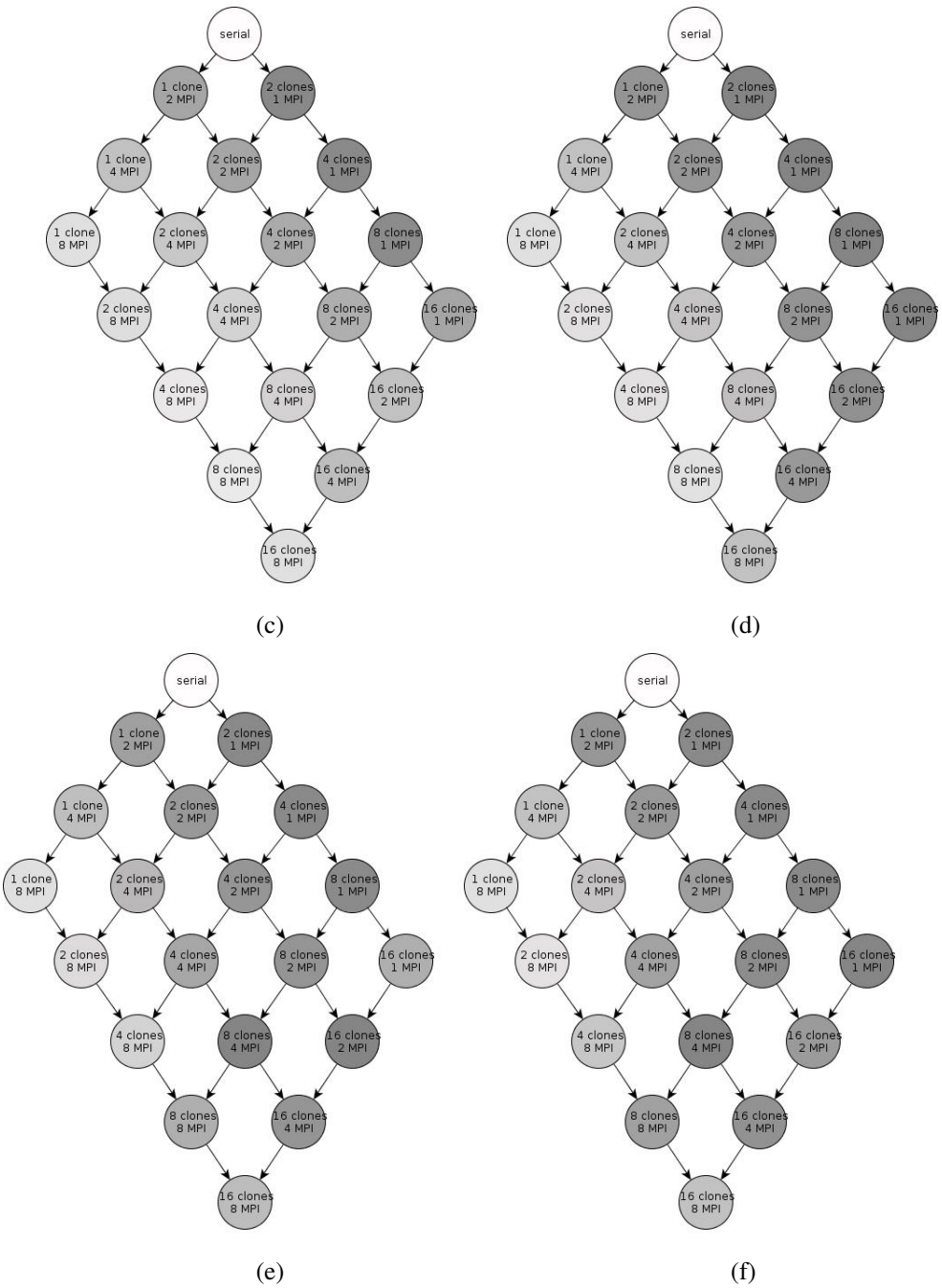


Figure 5.9 Working set size 8192 x 8192 (a)Graph with actual efficiency (b)Graph with predicted efficiency, Working set size 4096 x 4096 (c)Graph with actual efficiency (d)Graph with predicted efficiency, Working set size 2048 x 2048 (e)Graph with actual efficiency (f)Graph with predicted efficiency.

6.1.6 Conclusions

Total speedup and efficiency graphs are a simple and easy way to present our prediction, as well as check its deviation from the real speedup graph that is produced with the results of our measurements.

One of the important conclusions that need to be remarked was that the increase of the number of processors lead to the continuous increase of the speedup. In a larger scale cluster we would expect that the speedup would stop raising or even begin to fade as we add processing threads, since communication between processes would become the dominant factor. That is not the case in our cluster, since computation remains the main part of the total execution time even when the processing threads are continuously doubled. Thus, we notice that, in general, the more processes we add the better speedup we receive.

Viewing the speedup graph as a tree, we notice that in each height the leaves have an equal total number of processing threads, though placed differently among the clones. According to the produced graph we notice that in every working set size it is wise, from a performance point of view, not to allow the processes to share the same clone. That is, checking the leaves from left to right, speedup will increase. For example, assuming we want to use a total of 8 MPI processes. Placing each one of them into different nodes results in higher speedup than using the combination 2 clones with 4 mpi processes per clone.

As far as the efficiency graphs are concerned, another information that is derived would be that along with MPI processes that compute Heat 2D kernel, another application could run in the rest processors of the clones, as long as that application is not memory bound like Heat 2D. If so, the speedup would decrease dramatically. But if a compute bound computational kernel was executed, then the speedup would remain in the same levels and exploitation of the existing hardware would be maximized. The *scheduling* problem has yet many issues to overcome though, as mentioned previously in chapter 4.

We also present a table with the accuracy of the model predicting the right node of the graph for the applilcation to be executed at, as far as speedup is concerned.

Table 6.1: Prediction and measurement results for highest speedup node

Input size	Predicted Node	Real Node	Speedup Difference
2048 x 2048	16 cl - 8 procs/clone	16 cl - 8 procs/clone	0%
4096 x 4096	16 cl - 8 procs/clone	16 cl - 8 procs/clone	0%
8192 x 8192	16 cl - 4 procs/clone	16 cl - 4 procs/clone	0%

6.2 Heat equation 3D

The heat equation is studied, though when applied in a 3D space this 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:

$$\frac{\theta u}{\theta t} = \alpha(\frac{\theta^2 u}{\theta x^2} + \frac{\theta^2 u}{\theta y^2} + \frac{\theta^2 u}{\theta z^2}) \qquad (1)$$

As previous, using a forward difference at time t, a second-order central difference for the space derivatives and ignoring all multiplying factors, the previous equation becomes:

$$\begin{aligned} u(t,x,y,z) = \alpha(&u(t-1,x,y,z) + u(t-1,x-1,y,z) + \\ &u(t-1,x+1,y,z) + u(t-1,x,y-1,z) + \\ &u(t-1,x,y+1,z) + u(t-1,x,y,z-1) + \\ &u(t-1,x,y,z+1)) \end{aligned} \qquad (2)$$

The serial implementation of Heat3D equation, similarly to Pseudocode 5.1 of the serial implementation of Heat 2D equation, is presented:

Pseudocode 5.3 Serial implementation of Heat 3D 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 <= my_Z;z++)

                u[current][x][y][z] = a *
                        (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) ;
}
```

6.2.1 Parallel implementation

The domain decomposition policy that is adopted is the same as the one used to parallelize Heat 2D equation. Each process will be assigned to compute a chunk of the 3D space until time T. What is different this time has to do with the implementation of the communication between processes. Whereas in the implementation of 2D space *blocking* functions were used (MPI_Sendrecv()), when parallelizing the 3D Heat equation, we used *nonblocking* functions (MPI_Isend(), MPI_Irecv()), as shown in Pseudocode 5.4. Differences in the performance are discussed later, though comparisons between the two implementations can only be in regard to the accuracy of the prediction. The parallelization of the computational kernel in 3D space is introduced in Pseudocode 5.4.

Pseudocode 5.4 Parallel version of the Heat 3D equation

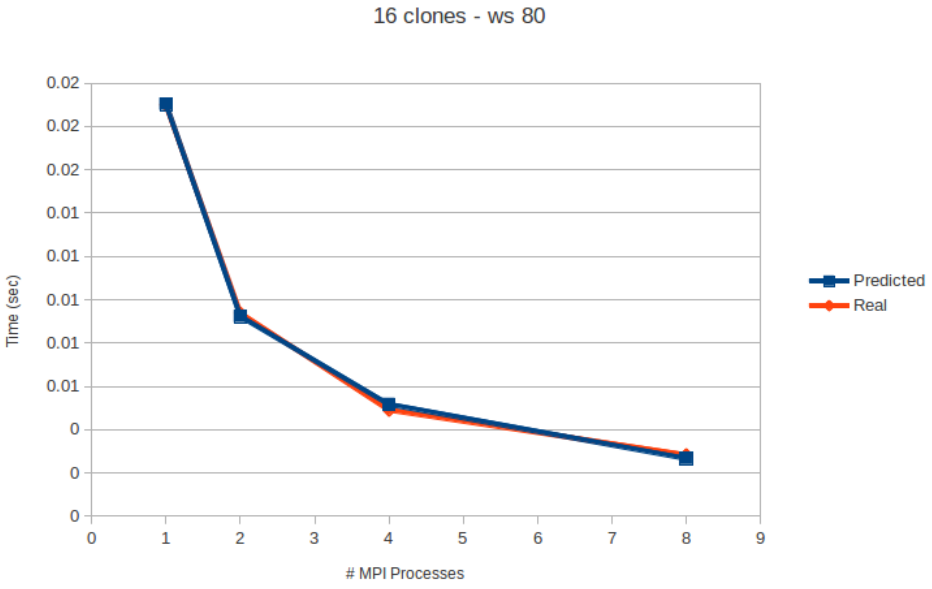
```
if (rank == 0) {
             allocate_full_matrix(A);
             initialize(A);
}

allocate_local_matrix(lA);
distribute_matrix(A,lA);

left =... //find out ids of neighbors
right =...
up = ...
down = ...
front = ...
back = ...


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

// Send and receive data

if (left) {
        Pack data (boundaries of t-1);

        MPI_Isend()
```

```
        MPI_Irecv()
        (boundaries of t-1 to and from neighbours);

        Unpack data (boundaries of t-1);
}

if (right) ...


// 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] = a *
                        (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);
}
```

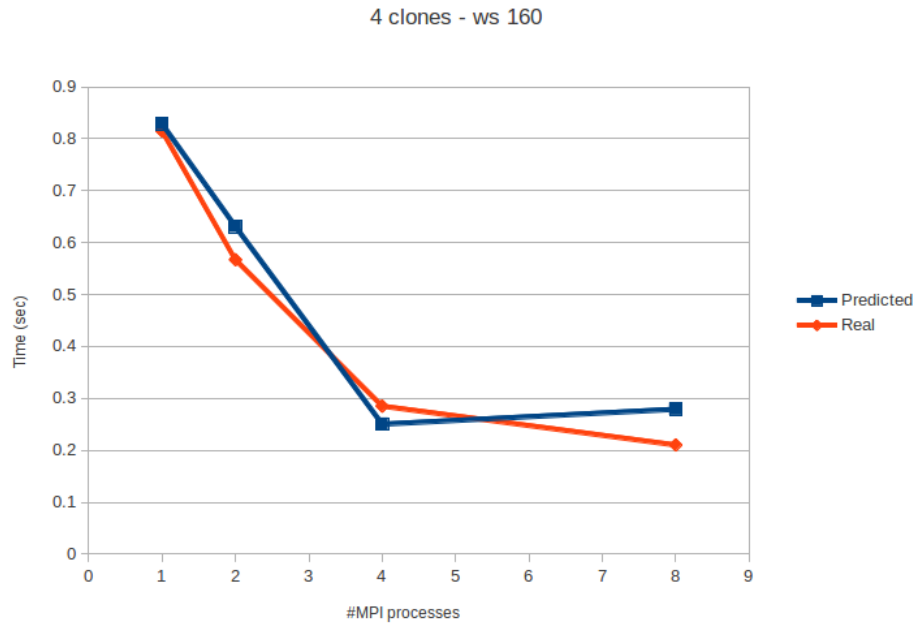
6.2.2 Modelling computation time

We used our main computation model in order to predict the scalability of the computation part of the kernel. The workspaces in which we experimented included multiples of the size of the cache.

As far as computation time is concerned, our results were similar to the ones we produced in Heat 2D equation, that is the deviations from the real computation time were quite small, as shown in Figure 5.10.



(a)

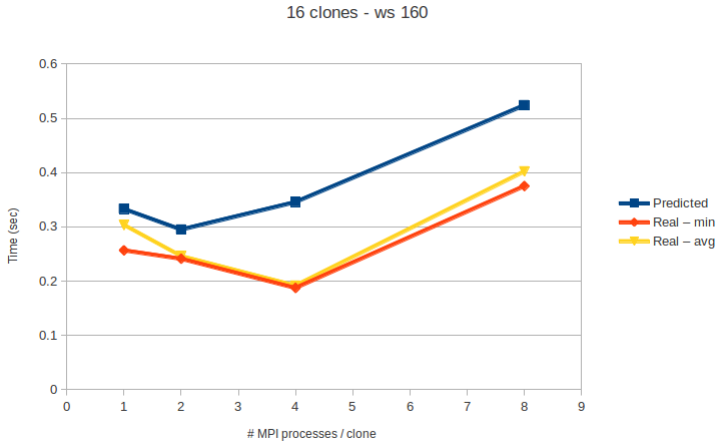


(b)

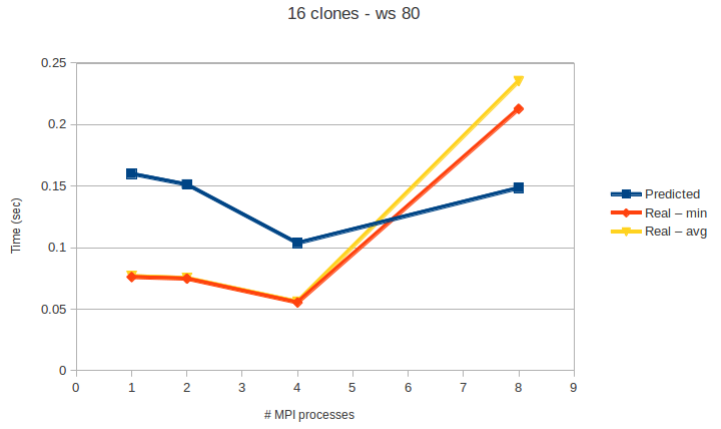
Figure 5.10 (a) Predicted and real computation time for working space 80 x 80 x 80 when 16 clones are used with 1 - 8 processes in each (b) Predicted and real computation time for working space 160 x 160 x 160 when 4 clones are used with 1 - 8 processes in each.

6.2.3 Modelling communication time

In order to model communication time, we used the same benchmark as in Heat 2D equation, with messages sizes that fit the ones that are sent between processes in the 3D space, as well as non - blocking MPI functions. Different implementation of communication though, lead to different results. Non - blocking functions were used in this application, though this fact might not be the cause of different results. Before we proceed to further comments, let us see certain results of communication prediction.



(a)



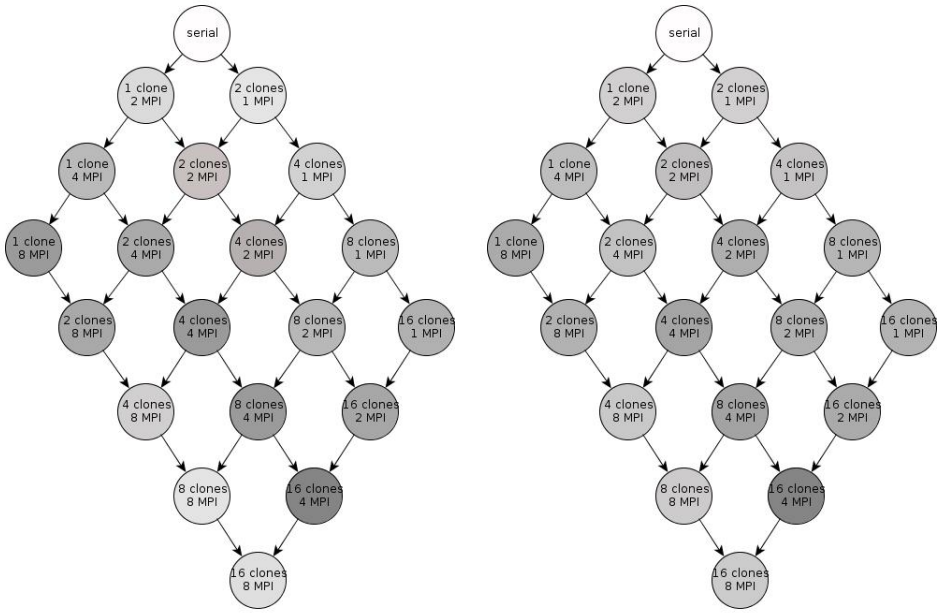
(b)

Figure 5.11 Examples of communication time prediction (a) Working space 160 x 160 x 160 when 16 clones are used with 1 - 8 processes in each (b) Working space 80 x 80 x 80 when 16 clones are used with 1 - 8 processes in each.

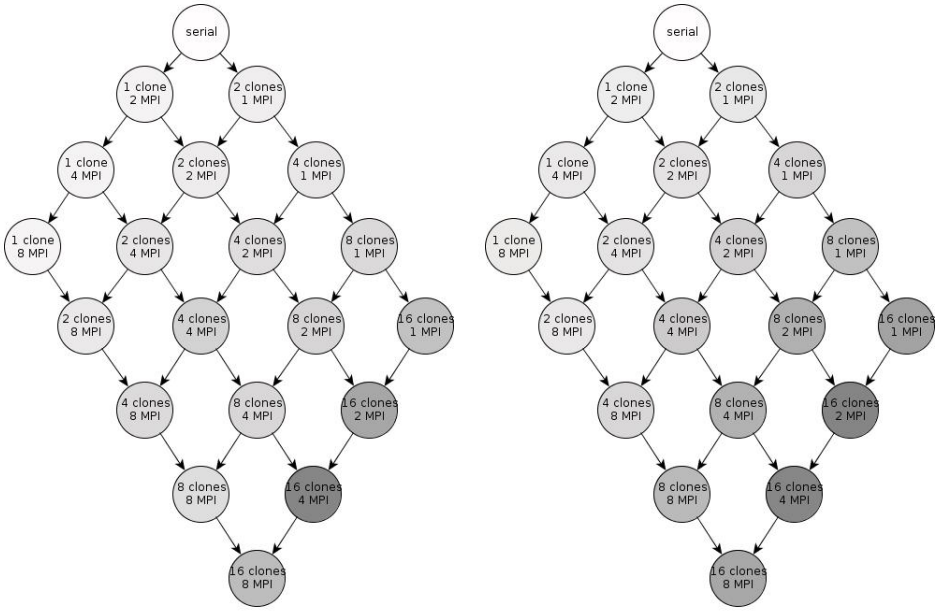
Noticing these prediction examples, there are two main facts to state. Firstly, the prediction of communication time is not as close as in 2D space of the same application - but generally overestimated - though the shape of the curves is very similar. Nevertheless, real average communication time is equal to real minimum communication time. The first thing that would appear to be the reason for these differences between the two applications is the change of communication's implementation functions. A simple experiment that we performed, showed us that this is not the case. Implementing Heat 2D application with non - blocking functions, and taking the same measurements, showed that what seems to be the main cause for this difference is the computational load that costs a lot less compared to communication time, and thus, the idle time of P0 in Figure 5.6 appears to be really small, since the heterogeneity of the cluster's processors does not affect the computation part a lot.

6.2.4 Presenting total speedup graphs

The overestimation of communication time, changes the predicted speedup, and thus our prediction may not be considered ultimately correct. Nevertheless, since our main goal is to find the area in our search space that produces the best speedup, overestimation might not alter these results, and thus our goal will probably be accomplished. In fact, presentation of the speedup graphs confirms that this is the case.



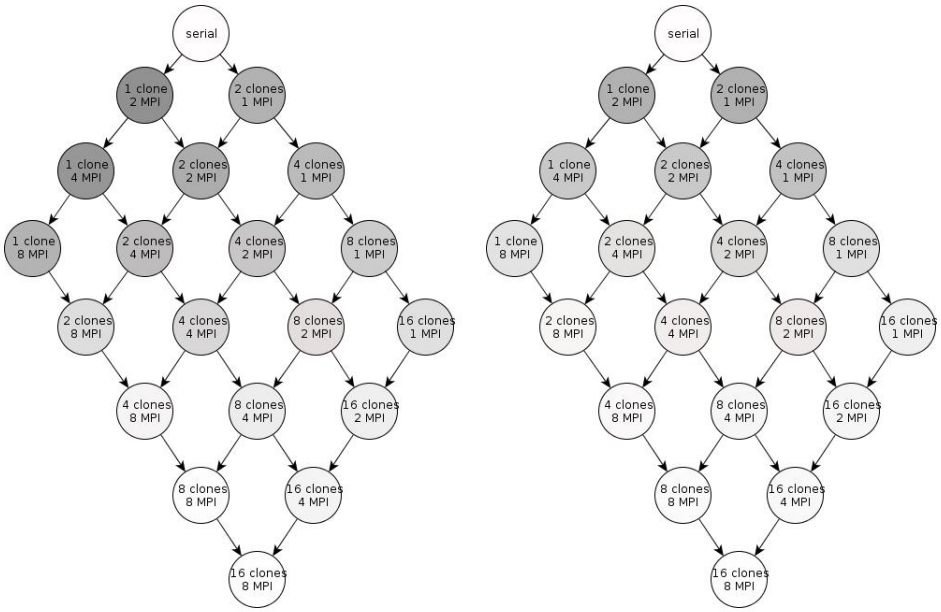
(a) (b)



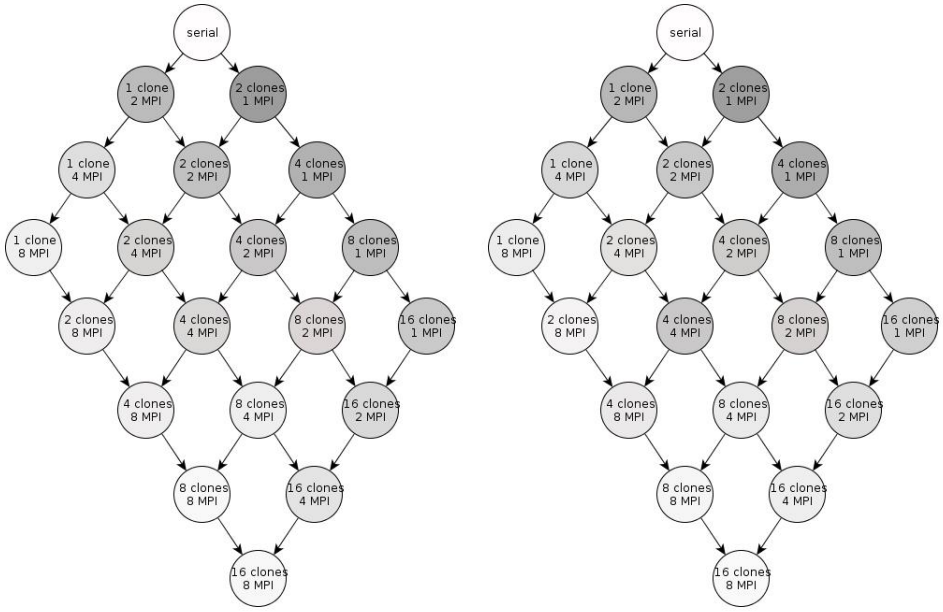
(c) (d)

Figure 5.12 Working set size 80 x 80 x 80 (a)Graph with actual speedup (b)Graph with predicted speedup, Working set size 160 x 160 x 160 (c)Graph with actual speedup (d)Graph with predicted speedup.

6.2.5 Presenting total efficiency graphs



(a) (b)



(c) (d)

Figure 5.13 Working set size 80 x 80 x 80 (a)Graph with actual efficiency (b)Graph with predicted efficiency, Working set size 160 x 160 x 160 (c)Graph with actual efficiency (d)Graph with predicted efficiency.

6.2.6 Conclusions

In this application, we noticed that the conclusions as far as the speedup graphs are concerned are being altered compared to the ones in the formerly studied application. More specifically, in Heat2D application it was clear that when using more clones, with each process being the only one in each clone, the speedup would be maximized. This is not the case when it comes to Heat3D application, since the grayest nodes of the speedup graphs are not met in the rightest nodes in every case e.g. for input size 80 x 80 x 80.

A different picture is observed when it comes to efficiency graphs as well, especially in the above mentioned input size, since the most efficiency-grant nodes are the ones were a small number of processes in total is used.

Overestimation of communication time is a fact that cannot be ignored. Nevertheless, the calculation of communication time in the experiment based model is an inherent overestimation, since we assume that processors' communication occurs between processes that are placed in different clones.

In the table below, we calculate the cost of our misprediction of the highest speedup node.

In other words, we compare the speedup that we would achieve if we executed the application using the predicted resources to the best one that was achieved.

Table 6.2: Prediction and measurement results for highest speedup node

Input size	Predicted Node	Real Node	Speedup Difference
80 x 80 x 80	16 cl - 4 procs/clone	16 cl - 4 procs/clone	0%
160 x 160 x 160	16 cl - 2 procs/clone	16 cl - 4 procs/clone	28%

Chapter 7

Future Work

For designers of large-scale parallel computers, it has been long desired to predict performance of parallel applications on various design alternatives at the design phase.

During this diploma thesis we studied the performance modelling of parallel applications. Attempts towards bridging the complex and analytic performance models that already exist with a simpler and experiment-driven model were made. Also, using an experiment based model to frame the job scheduling problem in a cluster, we reached to certain conclusions. Although the level of accuracy we reached could be considered as satisfiable, there are certain aspects of the performance of a parallel program that were not analysed. Thus, more experiments concerning different sizes of input, as well as using a different execution environment should be continued. Also, different communication patterns will be studied in detail, in order for the generality of the model to increase.

Bibliography

[1] Roofline: An Insightful Visual Performance Model for Multicore Architectures . Samuel Williams, Andrew Waterman, and David Patterson

[2] Parallel Processing Systems - 2011 Lecture Notes, Computing Systems Laboratory NTUA

[3] The boat hull model: adapting the roofline model to enable performance prediction for parallel computing

[4] Writing Message-Passing Parallel Programs with MPI - Course Notes , Neil MacDonald, Elspeth Minty, Tim Harding, Simon Brown - Edinburgh Parallel Computing Centre , The University of Edinburgh

[5] Interconnection networks for parallel computers - Michael Jurczyk, University of Missouri–Columbia , Howard Jay Siegel , Purdue University , Craig Stunkel , IBM T. J. Watson Research Center

[6] Inside Parallel Computers: Trends in Interconnection Networks , Howard Jay Siegel, Purdue University , Craig B. Stunkel, IBM T3. Watson Research Center

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

[8] 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

[9] “The Problems with Threads”, Edward A. Lee, Professor, Chair of EE, Associate Chair of EECS, EECS Department, University of California at Berkeley.

[10] The Landscape of Parallel Computing Research: A View from Berkeley , Krste Asanovic , Ras Bodik , Bryan Christopher Catanzaro , Joseph James Gebis , Parry Husbands , Kurt Keutzer , David A. Patterson , William Lester Plishker , John Shalf , Samuel Webb Williams , Katherine A. Yelick . Electrical Engineering and Computer Sciences - University of California at Berkeley.

[11] Measuring NUMA effects with the STREAM benchmark - Lars Bergstrom University of Chicago, Chicago IL 60637, USA

[12] Uses and abuses of Amdahl's law - S. Krishnaprasad, Journal of Computing Sciences in Colleges, December 2001.

[13] Michael J. Quinn. Parallel Programming in C with MPI and OpenMP. McGraw-Hill, 2004.

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

[15] Performance modelling of parallel applications - Erich Strohmaier, UT, 1999.

[16] J. Gustafson, "Reevaluating Amdahl's Law", Communications of the ACM , Volume 31, Number 5, May 1988.

[17] Job Scheduling on Parallel Systems, Jonathan Weinberg, University of California, San Diego 2006.

[18] Classifying scheduling policies with respect to higher moments of conditional response time, A. Wierman and M. Harchol-Balter. Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, New York, USA, 2005.