ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Τεχνικές Βελτιστοποίησης για Παράλληλες Εφαρμογές Μεγάλης Κλίμακας

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νικέλα Α. Παπαδοπούλου

Επιβλέπων: Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2012

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Τεχνικές Βελτιστοποίησης για Παράλληλες Εφαρμογές Μεγάλης Κλίμακας

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νικέλα Α. Παπαδοπούλου

Επιβλέπων: Νεκτάριος Κοζύρης
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Νοεμβρίου 2012.

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

Αθήνα, Νοέμβριος 2012.

.....

Νικέλα Α. Παπαδοπούλου
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Μία από τις μεγαλύτερες προκλήσεις στα σύγχρονα συστήματα παράλληλης επεξεργασίας είναι η ανάπτυξη παράλληλου λογισμικού που κλιμακώνει αποδοτικά. Αρκετές εφαρμογές δεν κλιμακώνουν έπειτα από έναν αριθμό επεξεργαστών, εξαιτίας του αυξημένου κόστους επικοινωνίας, που κυριαρχεί στο συνολικό χρόνο εκτέλεσης. Ο στόχος της παρούσας διπλωματικής εργασίας είναι η μελέτη της επίδοσης της παράλληλης εξίσωσης διάδοσης θερμότητας στον τρισδιάστατο χώρο, ως αντιπροσωπευτικού προβλήματος στην κατηγορία των προβλημάτων επίλυσης μερικών διαφορικών εξισώσεων, σε μια συστοιχία υπολογιστών διασυνδεδεμένη με Gbit Ethernet. Αρχικά, ερευνούμε την επίδοση των απλών παράλληλων υλοποιήσεων του προβλήματος με τη χρήση των OpenMP, MPI και του υβριδικού τους μοντέλου, επισημαίνοντας τους παράγοντες που περιορίζουν την κλιμάκωση. Ακολούθως, υλοποιούμε και ελέγχουμε την επίδοση τριων τεχνικών βελτιστοποιήσης που έχουν προταθεί στη βιβλιογραφία: i) tiling στους βρόχους του υπολογιστικού πυρήνα, ii) συμπίεση των μηνυμάτων του MPI και iii) επικάλυψη των υπολογισμών και της επικοινωνίας, τόσο με τη χρήση ασύγχρονων συναρτήσεων επικοινωνίας όσο και με την ανάθεση της επικοινωνίας και των υπολογισμών σε διαφορετικά νήματα, με τη βοήθεια του OpenMP. Συνοψίζοντας, προτείνουμε τη συνδυαστική χρήση της συμπίεσης μηνυμάτων και της επικάλυψης της επικοινωνίας και των υπολογισμών με υβριδικό μοντέλο MPI/OpenMP και παρατηρούμε βελτίωση της επίδοσης έως και 28%, συγκριτικά με τις απλές παράλληλες υλοποιήσεις.

Λέξεις-Κλειδιά: MPI, OpenMP, υβριδικός προγραμματισμός, τρισδιάστατη εξίσωση διάχυσης θερμότητας, tiling βρόχων, συμπίεση δεδομένων, επικάλυψη επικοινωνίας, συστοιχία SMP

Abstract

One of the most challenging problems in modern parallel processing systems is the development of parallel software that scales efficiently. Several applications do not scale further than a number of processors, due to communication overhead, which dominates the total execution time. The goal of this diploma is to study the performance of the parallel 3D heat equation, which represents the class of PDE solvers, on a cluster with commodity Gbit Ethernet and propose a set of optimization techniques for the reduction of the overall execution time of the application. Initially, we investigate the efficiency of baseline parallel implementations of the problem with OpenMP, MPI and their hybrid model, highlighting performance limiting factors. Subsequently, we implement and test the performance of three optimization techniques proposed in the literature: i) loop tiling of the computational kernel ii) compression of MPI messages and iii) overlapping of computation and communication, both by using non-blocking communication functions and by assigning computation and communication to separate threads, with the aid of OpenMP. To conclude, we propose the combined use of message compression and computation/communication overlapping with a hybrid MPI/OpenMP model, and we notice improvement in performance up to 28%, compared to the baseline parallel implementations.

Keywords: MPI, OpenMP, hybrid programming, 3D heat equation, loop tiling, compression, communication overlapping, SMP cluster

Ευχαριστίες

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

Καταρχάς, θα ήθελα να ευχαριστήσω τον καθηγητή μου κ. Νεκτάριο Κοζύρη, τόσο για την εποπτεία του κατά την εκπόνηση της εργασίας μου, όσο και για τις γνώσεις και την έμπνευση που μου προσέφερε με τη διδασκαλία του, καθ' όλη τη διάρκεια της φοίτησής μου.

Ιδιαίτερα θα ήθελα να ευχαριστήσω το Μεταδιδακτορικό Ερευνητή Γεώργιο Γκούμα για τη συνεχή του καθοδήγηση στη διάρκεια της εκπόνησης της διπλωματικής αυτής εργασίας, καθώς και για τις γνώσεις, την ενθάρρυνση και αισιοδοξία που μου παρείχε.

Επίσης, επιβάλλεται να ευχαριστήσω τους φίλους και συμφοιτητές για την παρουσία τους και την πολύτιμη βοήθειά τους, σε προσωπικό και επιστημονικό επίπεδο. Χωρίς αυτούς, τα φοιτητικά χρόνια που πέρασαν θα ήταν λιγότερο όμορφα και πολύ πιο δύσκολα.

Τέλος, οι θερμότερες ευχαριστίες απευθύνονται στους γονείς μου, για την αγάπη τους, την αμέριστη υποστήριξή τους και την εμπιστοσύνη τους σε κάθε μου επιλογή, από τα πρώτα χρόνια της ζωής μου μέχρι σήμερα, αδιάκοπα και ακούραστα.

Νικέλα Παπαδοπούλου

Contents

Chapter 1

Introduction to parallel processing systems

1.1 Overview

Since the appearance of the first IBM microprocessor chip in 1971, uniprocessor chips have dominated the computing industry for three long decades. This course has recently changed, as the increasing number of transistors per processor chip has set physical limitations to the further increase of the clock frequency, which has constituted the main performance metric of all modern computing systems. High performance microprocessors have been redefined to be multiple cores integrated into the same computing component, namely multicores or many-cores or multicore processors.

The consequences of this new design trend on computer engineering and science are of great importance. If the predictions on the future of multicores [1], which assume the integration of thousands of cores into the same chip, are to be verified, conventional architectures and programming models are to be substituted by parallel architectures and parallel programming models, in order to take full advantage of the available hardware resources and to achieve and maximize performance of applications.

The prevalent class of applications to be benefited from multicores consists of computation-intensive applications. Supercomputers have been employed for years to process computationally demanding scientific applications, such as molecular dynamics, cosmological simulations and mesh computations. Moreover, customer-oriented applications, involving computer graphics, database management and machine learning, do have increasing demands in computational resources, in an effort to manage large datasets and/or reduce response time. However, contemporary sequential algorithms of common applications, even if optimized, are not expected to demonstrate an optimum performance when executed on the novel multiprocessor architectures, without being subjected to non-trivial amendments, in order to effectively distribute their workload to the multiple cores. In other words, sequential algorithms must be redesigned to run

in parallel.

The concept of parallel processing, which has been successfully applied in high performance computing in the past decades, reinforces the prospects of the multicore era. The effectual combination of modern parallel architectures and redesigned parallel algorithms is the core of parallel computers, which can be vaguely classified into clusters, where multiple processors execute the same task and multicore computers, where a single computer constitutes of multiple processing elements.

The emerging challenge is to exploit the processing power provided by the new multicore architectures, by developing parallel programming models, independent of the number of processors , so as to transcend the reliance of software on the clock frequency and establish a productive correlation between application efficiency, performance scaling in accordance to the number of processors and programming effort.

The following sections of this chapter form an outline of parallel processing systems and parallel programming.

1.2 Parallel programming, Amdahl's law and scalability

Although a prevailing view concerning parallel programming is that any speedup via parallelism is considered a success, there is no explicit formula for the parallelization of sequential algorithms. The developer faces the challenge of exploring the potential parallelism of an algorithm, with respect to its semantics, and resolving issues that directly affect the execution time of the parallel program.

Before enumerating the aforementioned issues, we shall introduce some main performance metric of parallel programming. Thus, we define the following [2]:

- $T_p(n)$: the *parallel runtime* of a program of size n on p processors
- $C_p(n) = pT_p(n)$: the *cost* of the parallel program. If $C_p(n) = T^*(n)$, where $T^*(n)$ is the runtime of the fastest sequential program, then the parallel program is called cost-optimal. Using asymptotic execution times, a program is cost-optimal if $\frac{T^*(n)}{C_p(n)} \in \Theta(1)$.
- *Speedup* $S_p(n) = \frac{T^*(n)}{T_p(n)}$: the relative saving of execution time that can be obtained by using a parallel execution on p processors compared to the best sequential algorithm. If the inequality $S_p(n) \leq p$ holds, then the parallel implementation is efficient. If $S_p(n) = p$, the speedup is *linear*.
- *Efficiency* $E_p(n) = \frac{S_p(n)}{p}$: an alternative measure for performance. If $S_p(n) \leq p$, then $E_p(n) \leq 1$

If the cost of the best sequential program is unknown or varies depending on the data set, then speedup is often computed by using a sequential version of the parallel implementation.

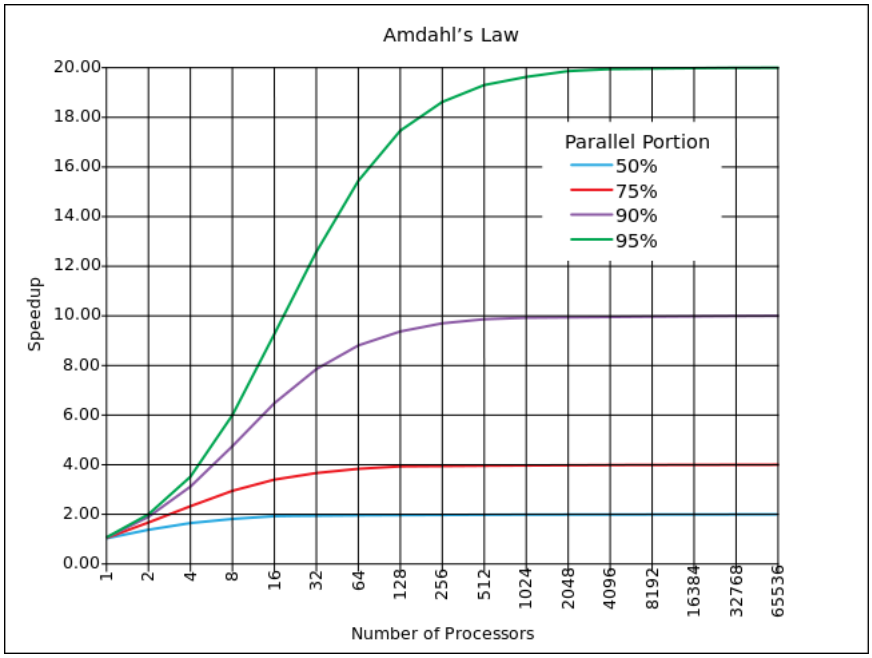


Figure 1.1: Total speedup of a parallel program as parallel fraction and number of processors increase

In the early years of high performance computing, Gene M. Amdahl[3] first denoted some inherent constraints in the process of parallel programming and efficiency attainment. Firstly, there is a fraction of computational load in every application, associated with data management, which cannot be executed in parallel with other computations and acts as a constant overhead to the runtime. Secondly, when the problem's dataset is distributed among processors, irregularity problems may occur, such as inhomogeneous interiors, irregular boundaries, inconsistency issues among variables and asymmetric convergence computations. To model the first restriction and set guidelines for coping with irregularity problems, Amdahl introduced his famous law, which captures their effect on the obtainable speedup. In detail, if $f(0 \leq f \leq 1)$ is a fraction of a problem of size n, which must be executed sequentially, then the total execution time of the problem on p processors is composed of a fraction $fT^*(n)$ of the sequential execution time and the execution time of the fraction, $(1-f)T^*(n)$, parallelized symmetrically on p processors, i.e. $(1-f)T^*(n)/p$. The expression for speedup is:

$$\text{Total Speedup } S_p(n) = \frac{T^*(n)}{fT^*(n) + \frac{1-f}{p}T^*(n)} = \frac{1}{f + \frac{1-f}{p}}$$

Amdahl's law is a useful measure of the best-case execution time for a parallel program. As the number of processors p goes to infinity, the total speedup goes to

$1/f$. If the parallelizable part of a program is relatively small, its speedup would be equally small, reagardless to the number of processing units. For instance, if the sequential part of a program is $f = 90\%$, then its execution time will be at most 10 times faster than the non-parallel program. Figure 1.1 depicts the influence of Amdahl's law in parallel executions of different sequential fractions. In terms of programming demands, Amdahl's law could be interpreted as following: to build an efficient parallel application, a programmer should estimate the prospects of parallelization of each appropriate algorithm and compare their theoretical speedups. Optimizations should be applied on the non-parallelizable fraction of the algorithm and eventually, an overall estimation should reveal whether parallelism offers a solution for an efficient application. Speedup as expressed by Amdahl's law is a measure of weak scaling. In parallel processing, scalability is a qualitative measure describing whether a performance improvement can be reached that is proportional to the number of processors employed. So far, we have assumed parallel programs of constant size. In this case, the execution time is expected to scale up with the number of processors, if not for other performance-restrictive factors, which will be mentioned ahead. A variant of Amdahl's law, proposed by John L. Gustafson [4], deals with the case when the problem size scales with the number of processors and is a measure of strong scaling. If τ_f is the constant execution time of the sequential program part and $\tau_{1-f}(n,p)$ is the execution time of the parallelizable program part for problem size n and p processors, then the scaled speedup of the program is expressed by:

$$S_p(n) = \frac{\tau_f + \tau_{1-f}(n,1)}{\tau_f + \tau_{1-f}(n,p)}$$

If the parallel program is perfectly parallelizable, then:

$$\tau_{1-f}(n,1) = T^*(1) - \tau_f \text{ and } \tau_{1-f}(n,p) = \frac{T^*(1) - \tau_f}{p},$$

so as the problem size n goes to infinity, speedup goes to the number of processors p, if $T^*(n)$ increases monotonically to n.

Scalability analysis is a powerful tool for parallel processing, which can save a lot of programming effort. Amdahl's and Gustafson's laws are appropriate for a vague performance prediction, though more complex methods have been proposed, such as isoefficiency functions [5], which express the required change of the problem size n as a function of the number of processors p.

1.3 Multi-processor architectures

Before expanding on common parallel architectures, we shall introduce Flynn's taxonomy [6] of computer architectures, according to the level of parallelism they employ to process instructions and data streams. The four classes identified are the following:

- SISD: Single instruction, Multiple Data
  A sequential (or uniprocessor) computer. No parallelism employed.

- SIMD: Single Instruction, Multiple Data
  A computer which concurrently processes multiple data streams with a single instruction stream, to perform operations that may be parallelized.

- MISD: Multiple Instruction, Single Data
  Uncommon, non-commercial architecture, used only for scientific purposes, as fault tolerance.

- MIMD: Multiple Instruction, Multiple Data
  Each processor executes its own instruction stream and processes its own data stream. This architecture supports multiple threads (thread-level parallelism). Multicore processors and clusters are examples of MIMD architectures.

Parallel computers are based on MIMD architectures, which can be further classified according to their memory organization, into shared-memory architectures, distributed-memory architectures and hybrid architectures and are profoundly analyzed below.

1.3.1 Shared memory architecture

In a shared-memory architecture, each processor owns a private cache memory hierarchy and all processors share a single physical address space, namely a global memory. A single system bus interconnects all processors, which communicate by sharing variables stored in the global memory. This memory organization, if all memory locations are equidistant to all processors, is also called a symmetric multiprocessor (SMP) and can be viewed in Figure 1.2.

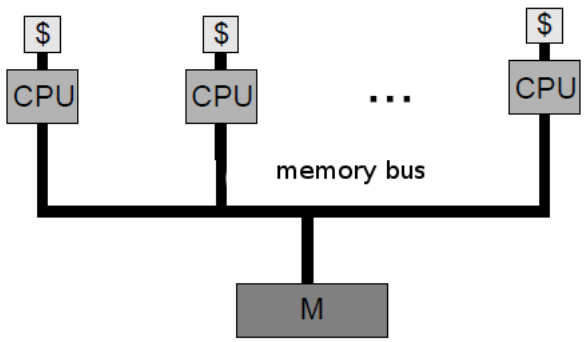


Figure 1.2: Classic Organization of a SMP

If the amount of time taken by any processor to access any global memory address is equal, the architecture organization is called UMA, which stands for Uniform Memory Access. Commercial symmetric multiprocessors have come to use the UMA organization. On the opposite side, the NUMA design, standing for Non Uniform Memory Access, has come as the result of scaling SMP units to create larger multiprocessor systems. In a NUMA design, the memory access

time for each processor depends on the memory location. It can be inferred that NUMA architectures demand special treatment by the operating system, but their asymmetry grants low latency to certain memory accesses and increases memory bandwith.

Processors in shared-memory architectures can still execute distinctive tasks using private virtual memory address space, yet when using shared physical addresses, race conditions are prone to occur. To avoid concurrent accesses to shared addresses and retain memory consistency, we employ synchronization mechanisms, such as locks and atomic operations. Moreover, cache coherent protocols are implemented to impose a universal sequence of accesses to the main memory.

Shared-memory architectures provide convenient mechanisms for parallel programming and execution , as processors exchange data via simple load and store operations on shared variables. However, a prerequisite for the operation of a shared-memory computing system is the use of a system bus and a memory system of a limited bandwith. Adding more processors to the system leads to saturation and thus, no more than 20 or 30 processors can be efficiently connected in a shared-memory model, unless the system bus is substituted by scalable interconnects.

1.3.2 Distributed memory architecture

Distributed memory architectures are a network of separate processing elements, called nodes. Each node owns a processor, a local cache hierarchy and a local main memory. No memory addresses are shared and only the local processor can access the local memory. Message passing, served by the interconnection network, is the only way of communication between the isolated nodes. The organization of a distributed memory architecture is pictured in Figure 1.3.

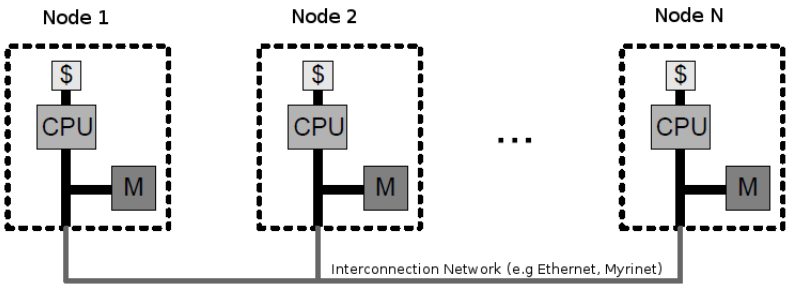


Figure 1.3: Classic Organization of a Distributed Memory Architecture

High performance computing area has been dominated by distributed memory architectures. Clusters are usually built of commodity computers, using the same operating system, physically connected through cables and switches, following some network topology. Software gets involved to manage communication between non-neighboring nodes. To decouple communication operations from

the processor's operations, direct memory access controllers (DMA) and routers are employed, which both enable data transfer directly from the local memory, using auxiliary message buffers.

A major drawback of clusters is their management cost. The cost of managing a cluster of n nodes equals to the cost of managing n computers, whereas the cost of managing a multiprocessor of n cores equals to the cost of managing one single computer. In addition, the use of an interconnection network, compared to a memory bus, adds up latency to the communication process, which increases with the number of nodes.

On the other hand, building up a cluster is a low-cost solution to gain high performance. The absence of shared memory eliminates race conditions, so the distributed memory model is fully scalable and modern clusters, or else supercomputers, may hold thousands of independent nodes, which can be maintained or replaced with no functioning effect on the system. Moreover, though parallel applications require re-engineering to run efficiently on a distributed memory system, convenient programming models which handle message passing between nodes have been developed, allowing programmers to exploit the processing power of the system.

1.3.3 Hybrid architecture

Hybrid memory architectures have come of an effort to combine the benefits of shared-memory and distributed-memory models onto the same computing system. The result can be viewed in Figure 1.4. A hybrid system resembles a distributed memory systems, where a symmetric multiprocessor has taken the place of each single processor node. This design permits parallel processing within each node and scales up in the same way as a distributed memory system.

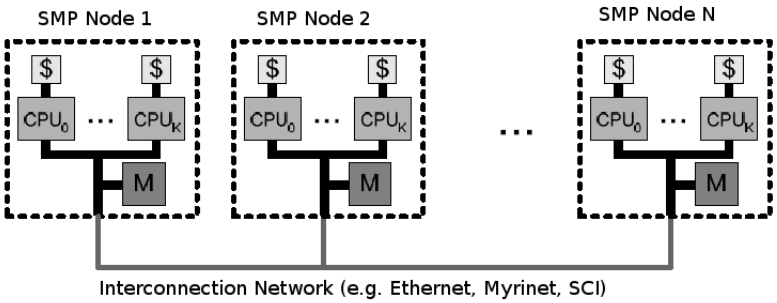


Figure 1.4: Classic Organization of a Hybrid Architecture

1.4 Parallel programming models

Parallel programming models lie between the hardware and the programmer as a user-interface, to facilitate parallel programming on the diverse parallel ar-

chitectures. Thus, respectively to the aforementioned architectures, there exist three parallel programming tendencies: the shared-memory model, the message passing model and a hybrid model. They are implemented as language extensions, runtime libraries of commonly used programming languages or even autonomous execution models.

Parallel programming models serve as the communication abstraction, since they act as a mapping from the generics of a programming language to the system's primitives. This mapping is provided either directly by the hardware, or by the operating system or by user software. In any case, the architecture defines the operations that are feasible and permitted and thus may be supported by the programming model.

As the field of parallel processing matures, the strict assertion of a parallel programming model to an identical hardware design tends to relax. Programming models are evolving towards an organizational structure with baseline communication primitives, practicable on the variant multiprocessor computing systems, yet their classification is still valid, for the evaluation of performance, execution time, memory usage, resource utilization and programming effort.

1.4.1 Shared memory programming model

A shared memory programming model embraces the notion of partitioning a programming task into multiple threads which run in parallel on the cores of a multiprocessor with a shared address space. Communication is launched via load and store operations on the shared address space, on the principle that whenever a processor writes to a shared memory address, all processors accessing the same address will be aware of the change. Synchronization mechanisms, such as barriers and locks, prevent race conditions from affecting the parallel program's correctness.

Shared memory programming models are friendly to programmers, as they facilitate data exchange through a simple annotation of a variable as shared, thus visible and accessible to all processing units. Moreover, such models supply the programmers with several parallel constructs, easily applicable to sequential programs for their parallelization. However, the task of resolving race conditions on a shared dataset may end up to be a perplexing task even for a highly skilled programmer. Managing shared data often leads to subtle and not easily traceable bugs, which in turn demand a thorough and time-consuming process of debugging.

It can be inferred that shared memory programming models are easily and efficiently implemented for shared memory platforms, carrying, though, the disadvantage of their limited scalability. On the other hand, any attempt of implementing such a model on a distributed memory platform requires special, performance degrading software layers and costly hardware support.

A commonly used shared memory programming model is OpenMP, an API that supports parallel programming with C, C++ and Fortran, by a set of compiler directives, library routines and environment variables that affect run-time

behavior. It will be discussed further in following chapters.

1.4.2 Distributed memory programming model

In a message passing programming model, a parallel program is launched as a set of independent processes, where the same instructions may reside on distinct computing nodes or computers. Each process owns its local variables, and sends and receives messages to and from other processes to achieve inter-process communication and data exchange. Message passing is executed by the operating system or by function calls to a library that activates low-level operations. In a naive approach of the message passing model, a send operation involves a local buffer where the message to be sent is stored and a receiving process, whereas its complementary receive operation involves a local buffer where the message to be received will be stored. Modern message passing APIs employ identifiers for processes and tags for messages, which allow the existence of a different message passing model. The message sent by the sending process is copied into an internal system buffer of the runtime system, thus the sending process is able to continue its operations after the copying operation is completed, while the receiving process copies the data from the internal buffer, by decoding the message's tag. This communication protocol is defined as the *"eager"* protocol, while the previously described protocol, where the send operation requires acknowledgement that there exists a matching receive operation, is known as *"rendezvous"*. Message passing models use both protocols to transfer messages efficiently. For instance, a small message may be transferred with a rendezvous protocol, while the eager protocol may be more suitable for a message of large size.

In a further classification, message passing models implement communication operations in two manners, synchronous and asynchronous message passing. *Synchronous* message passing refers to the case where both the sending and the receiving process block all their other operations until data exchange is accomplished. The message is immediately stored in the receiving process's local memory and no synchronization mechanism is required, as the collaborating processes share a synchronization point on the completion of communication. Synchronous communication is also defined as *blocking*. In *asynchronous* or *non-blocking* message passing, the message to be delivered is sent by the sending process without waiting for the receiving process to be ready to receive. Both processes may continue with their tasks until lower-level operations deliver the message. A disadvantage of asynchronous communication is that it involves an internal buffer, which, if full, may lead to a deadlock.

Depending on the number of processes due to exchange data on a single communication operation, communication is either *point-to-point* or *collective*. Point-to-point communication takes place when a single process sends data to a single receiving process. Collective communication involves more than two processes, in multiple sending and receiving points. For instance, a process may *broadcast* a set of data to all existing processes or to a subset of processes.

Message passing models have been developed to serve parallel programming on distributed memory computing systems, which have existed for a long time before the appearance of shared memory parallel systems. The need of portable communication facilities for a large set of parallel architectures leaded to the definition of the MPI (Message Passing Interface) standard library in 1992 and many library extensions since then, and has resulted to be the de facto standard for message passing in clusters. A major drawback of message passing models on clusters is that communication efficiency relies on the interconnection network. As the number of nodes in clusters increases, the complexity of the communication subsystem adds a significant overhead to message passing delays, which cannot be modelled with Amdahl's law. MPI is applicable to shared memory architectures as well, though for reasons of performance, the interconnection network is bypassed and message passing is served by shared memory operations.

Programming with MPI is a challenging job. The programmer has to design the parallel program from scratch, to decide about data distribution, message passing patterns and synchronization points and to employ the matching MPI routines. However, although it seems a laborious task, a fine parallel implementation on a message passing platform can be highly efficient and scalable, compared to its shared memory analog.

1.4.3 Hybrid Programming Model

The hybrid programming model is a combination of a shared memory and message passing model. A common hybrid model is the joint use of MPI and OpenMP. This model is implemented on hybrid memory architectures, as described above, where the shared memory model is used to parallelize a program interior a node of a SMP cluster and the message passing model is used for the communication between processes residing on distinct nodes. Hybrid programming implementations will be analyzed thoroughly on following chapters.

Chapter 2

Overview of the problem and the experimental environment

In this chapter, we present our case of study, the problem of solving the heat equation in a discretized three-dimensional space, along with a brief analysis of data dependencies in the resulting serial algorithm. We also provide specifications on our execution environment.

2.1 The heat equation in three dimensions

The heat equation describes the physical phenomenon of heat convection from areas of high temperature to areas of lower temperature within a region over time. If we discretize space and time, we are able to acquire an iterative solver for the equation, based on a stencil computation kernel. Stencil computations are used to solve many scientific computing problems, constituting a case of study and research for the field of parallel programming.

2.1.1 Derivation of the iterative method

If we express the thermal field as $T(x, y, z, t)$ of three spatial variables (x, y, z), within a region Ω of boundary $\vartheta\Omega$, the partial differential heat equation is:

$$\frac{\vartheta T}{\vartheta t} = \sigma(\frac{\vartheta^2}{\vartheta^2 x}T + \frac{\vartheta^2}{\vartheta^2 y}T + \frac{\vartheta^2}{\vartheta^2 z}T) \text{ on } \Omega \text{ and } T(x, y, z) = c \text{ on } \vartheta\Omega \quad (2.1)$$

To discretize this equation [7], we partition the domain in space, using a mesh of $x = 1, ..., X, y = 1, ..., Y, z = 1, ..., Z$ (see Figure 2.1). We then compute the sequence:

$$T^1_{ijk}, T^2_{ijk}, ..., T^{n-1}_{ijk}, T^n_{ijk}, T^{n+1}_{ijk}, ..., T^\infty_{ijk}$$

If we apply Euler-forward discretization on the equation (2.1), taking $\Delta x = \Delta y = \Delta z$. We then have:

$$\frac{T^{n+1}_{ijk} - T^n_{ijk}}{\Delta t} = \frac{\sigma}{\Delta x^3}(T^n_{i-1,j,k} + T^n_{i+1,j,k} + T^n_{i,j-1,k} + T^n_{i,j+1,k} + T^n_{i,j,k-1} + T^n_{i,j,k+1} - 6T^n_{i,j,k})$$

or

$$T_{ijk}^{n+1} = D(T_{i-1,j,k}^{n}+T_{i+1,j,k}^{n}+T_{i,j-1,k}^{n}+T_{i,j+1,k}^{n}+T_{i,j,k-1}^{n}+T_{i,j,k+1}^{n}-6T_{i,j,k}^{n})+T_{i,j,k}^{n}$$

In order to determine the rate of convergence, we write:

$$T_{ijk}^{n} = T_{ijk}^{\infty} + \epsilon_{ijk}^{n} \quad (2.2)$$

where ϵ_{ijk}^{n} is the deviation from the fixed-point, i.e. the steady state. The following equation is also satisfied:

$$\begin{aligned}\tfrac{1}{\Delta x^2}(&T_{i-1,j,k}^{\infty} + T_{i+1,j,k}^{\infty} + T_{i,j-1,k}^{\infty} T_{i+1,j,k}^{\infty} \\ &+ T_{i,j+1,k}^{\infty} + T_{i,j,k-1}^{\infty} + T_{i,j,k+1}^{\infty} - 6T_{ijk}^{\infty}) = 0 \quad (2.3)\end{aligned}$$

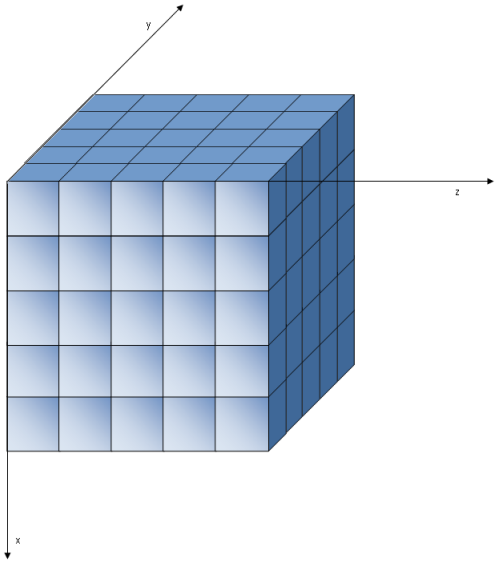


Figure 2.1: Discretization: Applying a 5x5x5 mesh on a cube

By subtracting the two different formulas above we obtain the equation for the convergence error:

$$\begin{aligned}\epsilon_{ijk}^{n+1} = \epsilon_{ijk}^{n} &+ D(\epsilon_{i-1,j,k}^{n} + \epsilon_{i+1,j,k}^{n} + \epsilon_{i,j-1,k}^{n} \\ &+ \epsilon_{i,j+1,k}^{n} + \epsilon_{i,j,k-1}^{n} + \epsilon_{i,j,k+1}^{n} - 6T_{i,j,k}^{n}) \quad (2.4)\end{aligned}$$

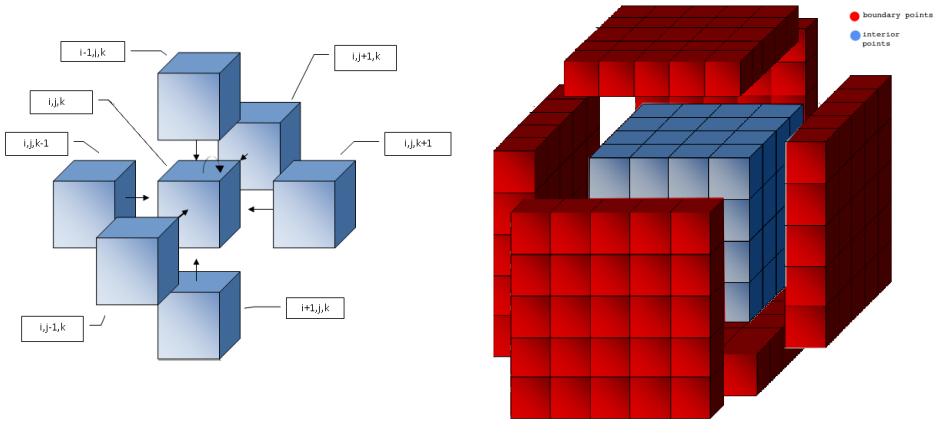
Convergence is obtained if $|\epsilon_{ijk}^{n}| \to 0$. Therefore, we require $D \leq 1/6$. By choosing $D = 1/7$, we obtain the following iterative method for solving equation (2.1):

$$\begin{aligned}T_{ijk}^{n+1} = \tfrac{1}{7}(&T_{i-1,j,k}^{n} + T_{i+1,j,k}^{n} + T_{i,j-1,k}^{n} \\ &+ T_{i,j+1,k}^{n} + T_{i,j,k-1}^{n} + T_{i,j,k+1}^{n} + T_{i,j,k}^{n}) \quad (2.5)\end{aligned}$$

for all the interior points.

2.1.2 Serial algorithm for the heat equation

The equation (2.5) is the kernel of the iterative method for the computation of thermal field's scalar T over time in all dimensions. Having discretized the three-dimensional space, we may represent the scalar T for the t-th moment as a set of data points, stored in a 3D matrix. The update of each datapoint depends on the current position and its neighbors; this constitutes a 7-point stencil (Figure 2.2a). Data dependencies of the $t-th$ moment are limited to the values of the $(t-1)-th$ moment; therefore, it is sufficient to preserve data on two distinct 3D matrices for successive moments in time, namely *current*, where new values are written, and *previous*, from which data is read. These matrices are swapped after each iteration. Initially, all interior,physically cold, points hold zero values and all heated boundary points are initiated with positive double precision values, as shown in Figure 2.2b, which remain constant over the iteration. The convergence criterion of the iterative method is the following:



(a) 7-point stencil: data dependencies

(b) Interior and boundary points on a 5x5x5 mesh

Figure 2.2: Representation of grid points and dependencies

To ensure convergence, equation (2.6) is tested every few time steps, with a value of $\epsilon = 0.001$. Assuming that the limits of the 3D space are: $1 \leq i < X, 1 \leq j < Y, 1 \leq k < Z$ and the limits of time are: $0 \leq t < T$, Pseudocode 2.1 describes a serial implementation of the algorithm.

The algorithm's complexity for a 3D space of n x n x n dimensions is computed as following:

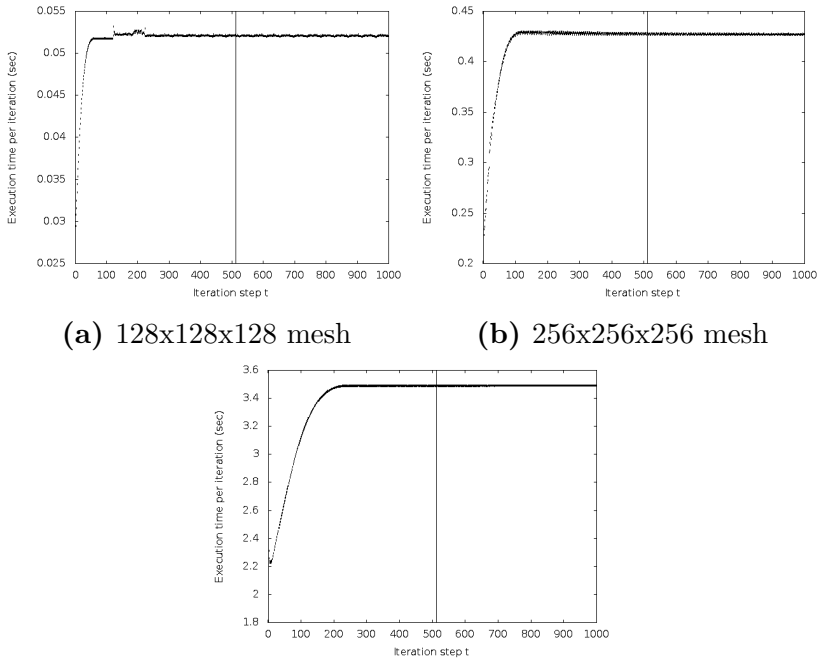
- Complexity of the iterative kernel: $O(Tn^3)$
- Complexity of the convergence criterion (applied $\frac{T}{10}$) times: $O(\frac{T}{10}n^3)$

Pseudocode 2.1: Serial Implementation of 3D Heat Equation Algorithm

```
//Computational Kernel
converged=0;
//Iteration over time
for (t=1;t<T && !converged;t++)
{
  //Computing new values of interior points
  for (i=1;i<X-1;i++)
    for (j=1;j<Y-1;j++)
          for (k=1;k<Z-1;k++)
            //Computation
            current[i][j][k]=(previous[i][j][k]+previous[i-1][j][k]
                              +previous[i+1][j][k]+previous[i][j-1[k]
                              +previous[i][j+1][k]+previous[i][j][k-1]
                              +previous[i][j][k+1])/7;
  swap(current,previous);
  if (t%10==0)
        converged=check_convergence();
}

//Test for Convergence
int check_convergence()
{
  for (i=2;j<X-1;j++)
    for (j=2;j<Y-1;j++)
      for (k=2;k<Y-1;k++)
        if (abs(current[i][j][k]-previous[i][j][k])>=0.001)
          return 0;
  return 1;
}
```

- Total complexity: $O(Tn^3) + O(\frac{T}{10}n^3) = O(Tn^3)$

Our primary experiments indicated that it takes more than 10^7 iterations to reach convergence on a 128x128x128 mesh, depending on the initial boundary values. In an effort to reduce the size of T and total complexity, for reasons of saving time on our experiments, we measured execution time per iteration on 3D grids of various sizes, the results of which are sketched in Figure 2.3, for a window of the first 1000 iterations. These measurements revealed that after non-zero values are spread to the interior points of the grid, execution time per iteration becomes stable, with deviations of microseconds. Henceforward, without loss of generality, for 3D spaces of 128x128x128, 256x256x256 and 512x512x512 points, which are the study cases of our experiments, we set T=512 and $\Delta t = 1$, as the time limit and time step, respectively, for solving heat equation, omitting the test for convergence.



(a) 128x128x128 mesh **(b)** 256x256x256 mesh

(c) 512x512x512 mesh

Figure 2.3: Execution time per iteration for different 3D meshes

2.1.3 Dependence analysis and parallelization prospects

A primitive strategy for seeking out useful parallelism is to look for a *data decomposition* in which parallel tasks perform similar operations on different elements of the data arrays [8]. An appropriate tool to extract parallelism in cases of nested loops, as is the case of our algorithm, is to construct the dependence matrix. The earlier expositions of data dependence and its applications were by

Pseudocode 2.2: Model of n-dimensional perfectly nested loops

```
for j1 ← l1 to u1 do
  for j2 ← l2 to u2 do
    ...
      for jn ← ln to un do
        A[f(j⃗)] = F(A[g1(j⃗)], ..., A[gm(j⃗)])
      end for
    end for
  ...
end for
```

Lamport [9], who developed the concepts of iteration spaces and distance vectors. Later, Wolfe [10] introduced the approach to direction vectors. These concepts have contributed to research on automatic code parallelization, the results of which lie beyond the purposes of our analysis.

An algorithmic model of n-dimensional perfectly nested loops, which best fits our case study, is the following:
The iteration space J^n is rectangular, thus it holds $J^n = \{\overrightarrow{j}(j_1, j_2, ...j_n) \in Z^n \wedge l_i \leq j_i \leq u_i, i = 1, ..., n\}$. We define $\overrightarrow{d_1}, ..., \overrightarrow{d_m}$ as the n-dimensional *dependence distance vectors*, which characterize dependences by the distance between the source $A[f(\overrightarrow{j})]$ and the sinks of dependences $A[g_1(\overrightarrow{j})], ..., A[g_m(\overrightarrow{j})]$ in the iteration space of our loop nest, i.e. $d_1 = f(\overrightarrow{j}) - g_1(\overrightarrow{j}), ..., d_m = f(\overrightarrow{j}) - g_m(\overrightarrow{j})$. The elements of distance vectors are all positive constant integers. *Dependence direction vectors* are defined as n-dimensional vectors such that: $D_k = \begin{cases} \text{"}<\text{"}, & \text{if} \quad d_k > 0 \\ \text{"}=\text{"}, & \text{if} \quad d_k = 0 \\ \text{"}>\text{"} & \text{if} \quad d_k < 0 \end{cases}, k = 1, ..., m.$ Direction vectors are a convenient mechanism to represent the iteration that occurs first, by treating " < " and " > " as arrows. Often, direction vectors have to be used when some dependencies cannot be presented as a finite number of distance vectors. The *dependence matrix*, denoted D, is a nxm matrix containing as columns the dependence vectors of the algorithm and it provides the necessary information for parallelization prospects. In detail, a loop of level i is free of dependencies if:

- the i-th element of all distance vectors is 0 or
- all distance subvectors from 0 to i-1 are lexicographically positive.

For our convenience in the expression of the dependence matrix for the heat equation algorithm, we will recant our suggestion for the use of two three-dimensional matrices. Instead, for the purposes of this section only, we propose the use of a four-dimensional matrix of dimensions T x X x Y x Z, with respect to equation (2.5). Thus, the problem can be expressed in four-dimensional model of perfectly nested loops, as in Pseudocode 2.3.

The equivalent dependence matrix D for this algorithm is the following 4x7

Pseudocode 2.3: The heat equation as a 4D (time x 3D-space)model of perfectly nested loops

```
for t → 0 to T − 1 do
  for i → 1 to X − 2 do
    for j → 1 to Y − 2 do
      for k → 1 to Z − 2 do
        A[t+1][i][j][k]=1/7(A[t][i][j][k]+A[t][i-1][j][k]+A[t][i+1][j][k]
          +A[t][i][j-1][k]+A[t][i][j+1][k]+A[t][i][j][k-1]+A[t][i][j][k
              +1])
      end for
    end for
  end for
end for
```

matrix:

$$D = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{matrix} t \\ i \\ j \\ k \end{matrix}$$

Since there is no row of the matrix of zero elements, we should check for lexicographically positive ordering of subvectors. All subvectors of dimensions 1,2 and 3 are lexicographically positive. Consequently, loops 2, 3 and 4 that iterate in space do not carry any dependencies. The outer loop, i.e. iteration in time is not free of dependencies; therefore, it must be executed sequentially. Another interesting corollary of dependencies in nested loops refers to loop interchange: a loop that carries no dependences cannot carry any dependences that prevent interchange with other loops nested inside it. In our case, loops i,j and k are interchangeable, as long as they remain nested to the outer loop of time.

2.2 Experimental environment

In this section, we present the execution platform of our experiments, with details on the architecture and the interconnection network. Figure 2.4 illustrates the architecture of this platform.

2.2.1 Architecture of the system

Our execution platform is a paradigm of hybrid architectures, i.e. it is a cluster consisting of SMP nodes. Each node has two quad-core processors, namely 8 processors running at 2.00GHz, which share a common level two cache memory in a pairwise way. In total, there are 16 nodes of Intel®Xeon® E5335 processors (Clovertown version) of 4MB L2 cache and 11 nodes of Intel®Xeon® E5405 (Harpertown version) of 6MB L2 cache. Each SMP node also has one memory slot of varying size. Finally, all nodes are connected via an interconnection network to form a cluster; both Gigabit Ethernet and Myrinet communication

networks are available on this cluster.

2.2.2 Interconnection network

The interconnection network significantly affects a cluster's performance, as communication between processors, regardless of processors' frequency, is bounded by network's latency and bandwidth. *Bandwidth* refers to the maximum rate at which the network can propagate information once the message enters the network[11].*Latency* is the time required to send a minimal size message through the network. In the early years of clusters, requirements for communication via a local area network (LAN) were met by Ethernet. Since then, many interconnection networks have been introduced and developed to provide high-speed communication. Nowadays, Gigabit Ethernet and Myrinet are commodity interconnects for clusters.

2.2.2.1 Gigabit Ethernet

Ethernet[12] was introduced in 1973, as an interconnection for the Xerox Altos machines. It was designed as a shared coaxial cable which acted as a broadcast transmission medium for the connected machines, i.e. when a machine transmits a message via Ethernet, the message is broadcast to all hosts of the network; it is the network interface card of each machine that decides whether the message is actually addressed to it and interrupts the processor for reception. Early Ethernet networks used a distributed media access control (MAC) protocol, known as Carrier Sense Multiple Access with Collision Detection (CSMA/CD), which detects ongoing transmissions and postpones any new transmissions until the channel is empty. Modern Ethernet networks are built with switches and full-duplex connections and no longer utilize this protocol, though it is still supported for reasons of compatibility.

Gigabit Ethernet is a modern Ethernet technology of high bandwidth, transmitting frames at a rate of a gigabit per second. Our cluster employs the TCP protocol when Gigabit Ethernet is used. The TCP path supports TCP message communication over Ethernet using the socket interface to the operating system's network protocol stack, thus as described in [13], memory copies are required to transfer data between the kernel and the application, adding overhead to communication operations.

2.2.2.2 Myrinet

Myrinet[14] is a high-speed type of LAN, built by Myricom, as an interconnection to computer clusters. Myrinet physically consists of two fibre optic cables, an upstream and a downstream and machines are connected to it via low-overhead routers and switches; there is no direct connection between two machines on a Myrinet network. The newest Myrinet generation's bandwidth is comparable to the latest Gigabit Ethernet's bandwidth, but is a lightweight protocol, which adds significantly less overhead, compared to Ethernet, therefore

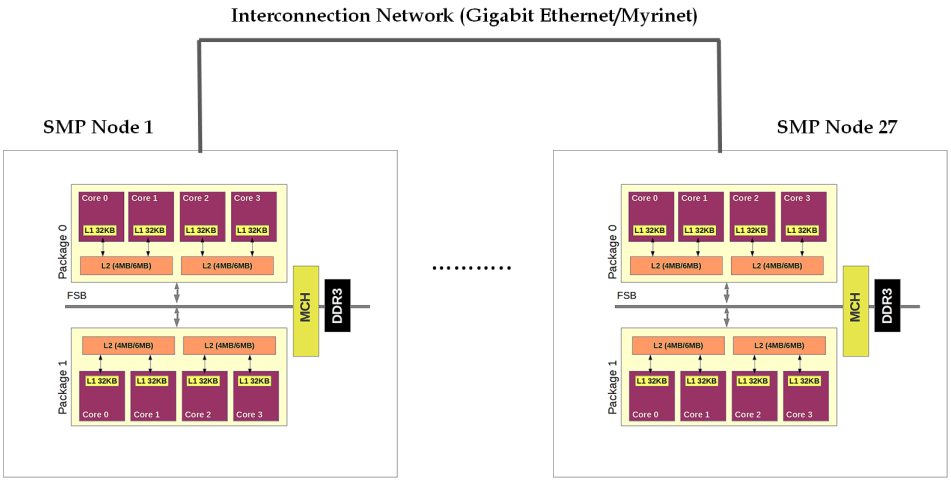


Figure 2.4: An overview of the architecture of the cluster

offering higher throughput and lower latency. Moreover, applications are often aware of Myrinet, thereby bypassing calls to the operating system.

Myrinet's high throughput enables it to operate close to the basic signaling speed of the physical layer. However, Myrinet's appliance and efficacy on supercomputers lies on its low overhead and latency, which limits bottlenecks caused by the latency of message transmissions over the network and affect the performance of parallel programs.

Chapter 3

Baseline parallelization of the heat equation with OpenMP and MPI

In this chapter, we discuss the aforementioned parallel programming models, namely OpenMP and MPI, the process of parallelizing the heat equation solver with both models and their hybrid version and the respective experimental results.

3.1 Parallelization with OpenMP

3.1.1 About OpenMP

OpenMP [15] is an industry standard API for writing multithreaded applications in a shared memory environment, in C/C++ and Fortran, since 1997. It consists of a set of compiler directives, runtime library routines and environment variables and parallelism is explicitly declared by the programmer, who inserts OpenMP directives at key locations in the source code, incrementing the serial program, while the compiler interprets these directives and generates library calls to parallelize code regions.

OpenMP employs the *fork/join* model to implement multithreading. In detail, a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed. Each thread has an id attached to it, which can be obtained using a function. The thread id is an integer, and the master thread has an id of 0. After the execution of the parallelized code, the threads join back into the master thread, which continues onward to the end of the program. This model is

illustrated in Figure3.1 By default, each thread executes the parallelized section

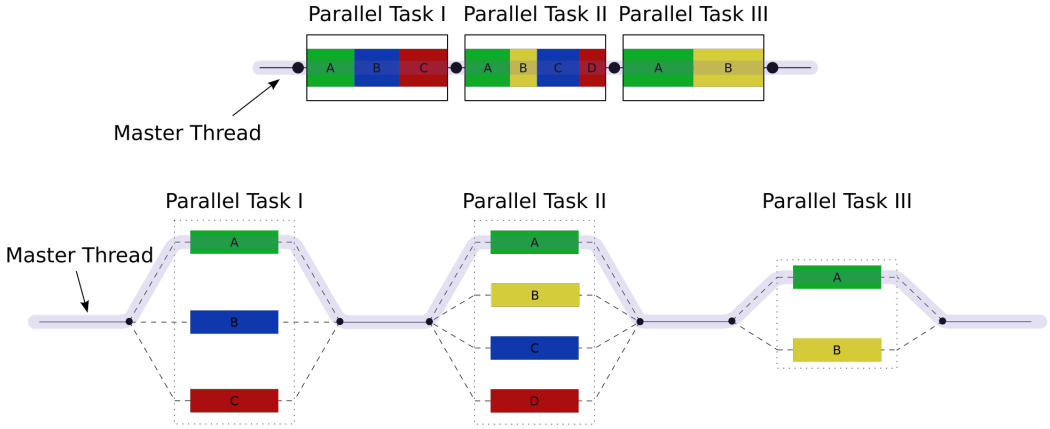


Figure 3.1: An illustration of multithreading where the master thread forks off a number of threads which execute blocks of code in parallel.

of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The number of threads can be assigned by the runtime environment based on environment variables or in code using functions.

3.1.2 Useful OpenMP components

In C/C++, OpenMP directives are specified with the **pragma** preprocessor directive, with the following syntax: **#pragma omp** *directive-name [clause[[,]clause]...] new-line*. The OpenMP program executes sequentially until the *parallel* directive is encountered. This directive defines a parallel region in the form of a structured block and creates a group of threads which execute in parallel the structured block. At the end of the parallel region, threads are synchronized and a flush operation makes the master's thread temporary view of memory consistent with memory. The number of threads may be adjusted dynamically or be defined by the user, either by adding the *num_threads(integer-expression)* clause to the *parallel* directive, or by using the *omp_set_num_threads* function, or by setting the environment variable *OMP_NUM_THREADS*, which specifies the number of threads on runtime. To obtain a thread's id, OpenMP provides the *omp_get_thread_num* function.

Since OpenMP is a shared memory programming model, most variables in OpenMP code are visible to all threads by default. However, sometimes private variables are necessary to avoid race conditions and there is a need to pass values between the sequential part and the parallel region (the code block executed

in parallel), so data environment management is introduced as data sharing attribute clauses by appending them to the OpenMP directive. Our codes deal only with the *shared*, *private* and *firsprivate* clauses, so we will give a bried description here:

- *shared*: the data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the work sharing region are shared except the loop iteration counter
- *private*: the data within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the OpenMP loop constructs are private
- *firstprivate*: like private except initialized to original value

Figure 3.2 depicts the OpenMP data model.

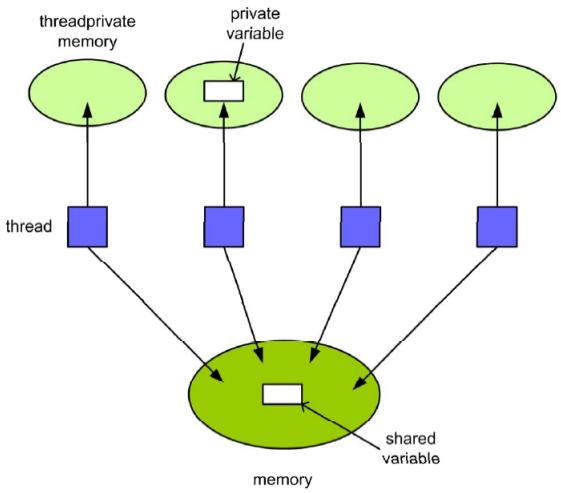


Figure 3.2: OpenMP data model

Various work sharing constructs are offered by OpenMP, in order to specify how to assign independent work to one or all of the threads. The *omp for* construct is employed to split up loop iterations among threads. This construct also allows the user to define the schedule for work-sharing, so as to minimize the wait time of threads at synchronization points, using *static schedule* as the default scheme. Dynamic schedule is characterized by the property that all the threads are allocated iterations before they execute the loop iterations. The iterations are divided among threads equally by default. However, specifying an integer for the parameter chunk will allocate chunk number of contiguous iterations to a particular thread. Moreover, OpenMP supports the *omp single* construct, which declares that a code section, within a parallel region, will be

Pseudocode 3.1: Parallel implementation of the 3D heat equation with OpenMP

```
T=512

#pragma omp parallel num_threads(p) private(t,i,j,k)
{
  for (t=0;t<T;t++)
  {
    #pragma omp for
    for (i=1;i<X-1;i++)
      for (j=1;j<Y-1;j++)
        for (k=1;k<Z-1;k++)
          current[x][y][z]=(previous[x][y][z]+previous[x-1][y][z]
                            +previous[x+1][y][z]+previous[x][y-1[z]
                            +previous[x][y+1][z]+previous[x][y][z-1]
                            +previous[x][y][z+1])/7;
       #pragma omp single
       {
          swap(current,previous);
       }
  }
}
```

executed by a sole thread of the group, implying a synchronization point in the end.

A simple synchronization mechanism provided by OpenMP is the *barrier* directive, which is almost self-explanatory: when a thread of a parallel region reaches the *barrier* directive, it waits until all threads of the group reach this point. Although barriers are implied in the end of parallel regions by default, it is often useful for the programmer to explicitly declare synchronization points within parallel regions.

3.1.3 Parallelization of the 3D heat equation with OpenMP

Parallelization of the 3D heat equation with OpenMP is almost straightforward, on the basis of the dependence analysis in 2.1.3. Since there are no data dependencies between the inner loops of the computational kernel, we apply the *omp parallel for* directive to parallelize the first inner loop which iterates over dimension X in space, using static scheduling. This implementation follows the SPMD (Single Process, Multiple Data) model, as different threads perform the same work on separate chunks of data. Pseudocode 3.1 is the respective implementation for a X x Y x Z mesh.

The number of threads is defined manually by the user. We have chosen to place the *omp parallel* directive outside of the outer loop, so as to minimize the thread management overhead, namely the fork-and-join overhead. For reasons of data consistency, all loop variables have been declared as private, while matrices

current and *previous* are by default shared, to serve communication between different threads.

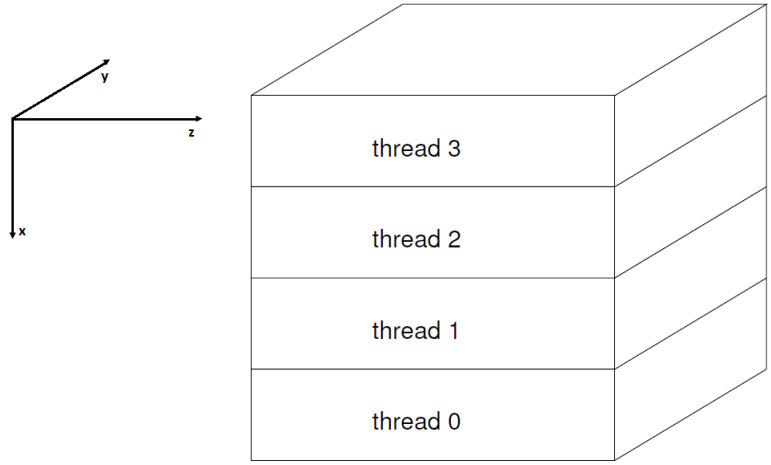


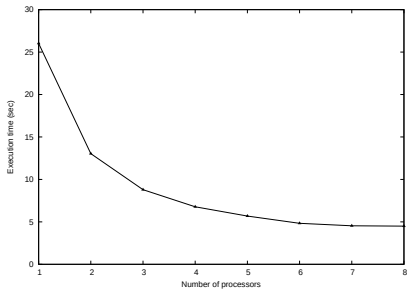
Figure 3.3: Distribution of iterations to OpenMP threads for a parallelization of the outermost loop (x direction) and static schedule. The overall number of iterations (i.e. the number of elements in this direction) is split up into equal chunks for the threads.

To explain our choice of parallelizing the outer loop, we shall introduce the notion of *granularity*. Granularity is a qualitative measure of the ratio of computation to communication. Loop granularity, along with the number of available threads and the number of iterations are the conditions to decide which loop is suitable for parallelization, in case of nested loops with no dependencies [17]. Parallelization of the outer loop is a good choice if the number of iterations of the outer loop is large enough compared to the number of available processors, to reach a good data distribution. Parallelization of an inner loop or all loops will potentially provide smaller pieces of work so they can be distributed evenly between the available threads but this choice suffers from overhead due to work distribution and thread synchronization. In order to achieve large granularity and small parallelization overhead in our implementation, it is preferable to parallelize the outer loop only [18]. The parallelization scheme can be viewed in Figure 3.3.

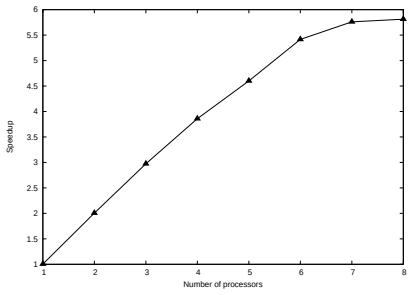
Finally, the *omp single* directive is used to define that a single thread shall undertake the task of swapping the matrices before the next time step, which is necessary for the correctness of the implementation. All threads are synchronized at the end of this code block by the directive's implied barrier, updating their local copies to be consistent with the shared memory's data, before proceeding to the next iteration over time.

3.1.4 Experimental results of the parallel implementation with OpenMP

In this section, we present the experimental results of the execution of our OpenMP implementation for the 3D heat equation on a single clovertown-based node of the cluster, using 1 to 8 cores progressively. We have measured the execution time for three 3D spaces (128x128x128, 256x256x256 and 512x512x512) for 512 iterations over time. Figures 3.4, 3.5 and 3.6 depict the results for the different 3D spaces.

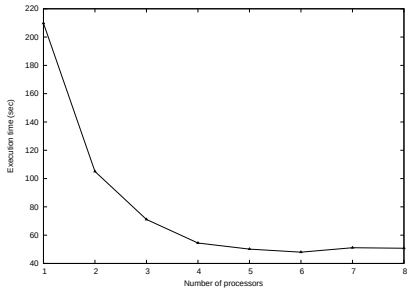


(a) Execution Time

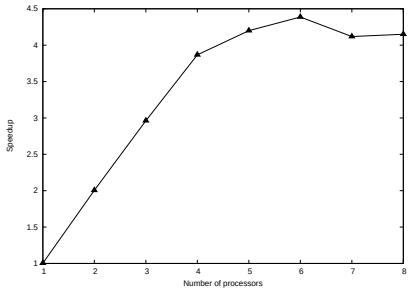


(b) Speedup

Figure 3.4: Results of heat equation with OpenMP for a 3D space 128x128x128



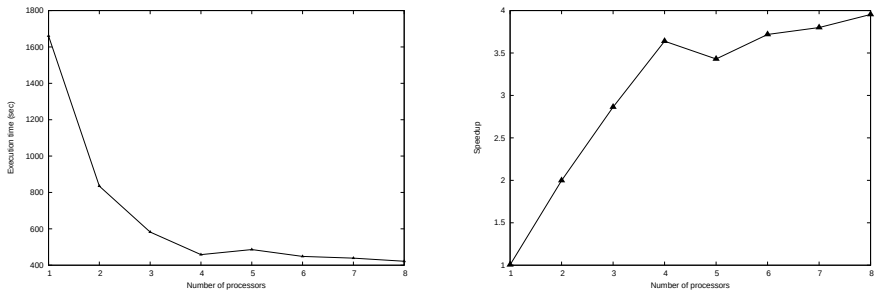
(a) Execution Time



(b) Speedup

Figure 3.5: Results of heat equation with OpenMP for a 3D space 256x256x256

Our first observation is that speedup is linear up to four cores(or else four OpenMP threads), which is an indication that our parallelization approach was right. However, while increasing the number of cores over four, efficiency decreases in a different manner for each of the 3D spaces. For the smallest 3D space of 128x128x128, the implementation scales up to 8 cores, but the speedup



(a) Execution Time **(b)** Speedup

Figure 3.6: Results of heat equation with OpenMP for a 3D space 512x512x512

is worse than linear, while for the 256x256x256 and 512x512x512 spaces, scalability breaks after seven and five cores respectively. This behavior is expected from a *memory-bound* algorithm, as is heat equation. Memory-bound algorithms are algorithms with a high ratio of memory accesses to computations and most stencil computations fall into this category. In our case, where all threads operate on shared data, adding more threads induces more concurrent memory accesses which in turn cause contention on the frontside bus (FSB), through which both packages of the node transfer data from the main memory to their L2 caches. The saturated L2 demands cause the decrease of performance exhibited in our results. The limited working set, namely 32MB, of the 128x128x128 3D space conceals the effect of bandwidth saturation, as there is an equally limited need for memory accesses, but its overall performance is bounded by the same constraints as for larger working sets.

3.2 Parallelization with MPI

3.2.1 About MPI

MPI (Message-Passing Interface) is a message-passing library interface specification. All parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. (Extensions to the "classical" message-passing model are provided in collective operations, remote memory access operations, dynamic process creation, and parallel I/O.) MPI is a specification, not an implementation; there are multiple implementations of MPI. This specification is for a library interface; MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings, which for C, C++, Fortran-77, and Fortran-95, are part of the MPI

standard. [19]

The MPI interface is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to nodes/servers/computer instances) in a language-independent way, with language-specific syntax (bindings), plus a few language-specific features. MPI programs always work with processes, but programmers commonly refer to the processes as processors. Typically, for maximum performance, each CPU (or core in a multi-core machine) will be assigned just a single process. This assignment happens at runtime through the agent that starts the MPI program, normally called mpirun or mpiexec.

MPI library functions include, but are not limited to, point-to-point rendezvous-type send/receive operations, choosing between a Cartesian or graph-like logical process topology, exchanging data between process pairs (send/receive operations), combining partial results of computations (gather and reduce operations), synchronizing nodes (barrier operation) as well as obtaining network-related information such as the number of processes in the computing session, current processor identity that a process is mapped to, neighboring processes accessible in a logical topology, and so on. Point-to-point operations come in synchronous, asynchronous, buffered, and ready forms, to allow both relatively stronger and weaker semantics for the synchronization aspects of a rendezvous-send. Many outstanding operations are possible in asynchronous mode, in most implementations.

3.2.2 Useful MPI components

An MPI program consists of a collection of processes that can exchange messages. Normally, each processor of a parallel system executes one MPI process, and the number of MPI processes started should be adapted to the number of processors that are available. Typically, all MPI processes execute the same program in an SPMD style. All communication operations are executed using a communicator. A communicator represents a communication domain which is essentially a set of processes that exchange messages between each other. The default communicator is the *MPI_COMM_WORLD*, which captures all processes executing a parallel program. Each process of a process group has a unique *rank* within this group which can be used for communication with this process and the process of rank equal to 0 is the *root* process. Although a process is uniquely defined by its group rank, it is often useful to have an alternative representation and access. This is the case if an algorithm performs computations and communication on a two-dimensional or a three-dimensional grid where grid points are assigned to different processes and the processes exchange data with their neighboring processes in each dimension by communication. In such situations, it is useful if the processes can be arranged according to the communication pattern in a grid structure such that they can be addressed via two-dimensional or three-dimensional coordinates. Then each process can easily address its neighboring processes in each dimension. MPI supports such a logical arrangement of pro-

cesses by defining *virtual topologies* for intra-communicators, which can be used for communication within the associated process group. The *MPI_Cart_create* routine creates a virtual Cartesian grid structures of arbitrary dimension. The operations *MPI_Cart_rank* and *MPI_Cart_coords* translate the Cartesian coordinates into its group's rank and vice versa.

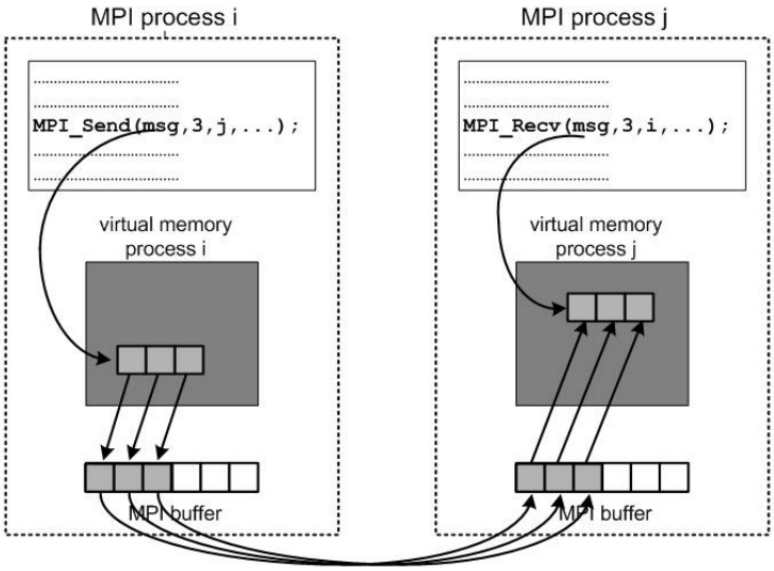


Figure 3.7: The operations of send/receive from a user's level viewpoint

The most basic form of data exchange between processes is provided by point-to-point communication. Two processes participate in this communication operation: A sending process executes a send operation and a receiving process executes a corresponding receive operation. Figure 3.6 represents this scheme from a user level's viewpoint. MPI supports both blocking and non-blocking point-to-point communication operations. *MPI_Send()* and *MPI_Recv()* are MPI's blocking, asynchronous operations for point-to-point communication. This means that an *MPI_Recv()* operation can also be started when the corresponding *MPI_Send()* operation has not yet been started. The process executing the *MPI_Recv()* operation is blocked until the specified receive buffer contains the data elements sent. Similarly, an *MPI_Send()* operation can also be started when the corresponding *MPI_Recv()* operation has not yet been started. The process executing the *MPI_Send()* operation is blocked until the specified send buffer can be reused. For a situation where processes both send and receive data, MPI provides the combined *MPI_Sendrecv()* operation, for which the MPI runtime system guarantees deadlock freedom. However, the use of blocking communication operations can lead to waiting times in which the blocked process does not perform useful work. Often, it is desirable to fill the waiting times with useful operations of the waiting process, e.g., by overlapping communications and computations. This can be achieved by using the non-blocking communication

operations, *MPI_Isend()* and *MPI_Irecv()*, which initiate the sending and receiving of a message respectively and return control to the process. The *MPI_Test()* and *MPI_Wait()* functions can be used to test or wait for the completion of a non-blocking communication operation. The *MPI_Get_count()* function is used to obtain the actual size of a received message.

MPI supports various forms of collective communication. Collective communication operations in MPI are always blocking. *MPI_Bcast()* performs the broadcast operation, where one specific process of a group of processes sends the same data block to all other processes of the group. The *MPI_Reduce* function is used to collect blocks of data from all processes of the group to the root process, combined with the use of a binary operator. Moreover, MPI provides the *MPI_Scatter()* function to scatter blocks of data of equal size from the root process to all processes of the group (there also exists an alternative *MPI_Scatterv()* function for blocks of data of various sizes) and the *MPI_Gather()* (also *MPI_Gatherv()*) to collect blocks of data from all processes to the root process. In addition, MPI provides a synchronization routine, the *MPI_Barrier()*, which acts in a similar way to the *omp barrier*. All processes which belong to a group synchronize when this routine is called.

Finally, MPI provides a set of predefined datatypes, such as *MPI_INT*, *MPI_DOUBLE*, and a set of derived datatypes, which can prove very useful to specify the type of data to be sent, such as structs, vectors and subarrays.

3.2.3 Parallelization of the 3D heat equation with MPI

To parallelize the 3D heat equation with MPI, we adopt a domain decomposition policy. The 3D space is divided into 3D subdomains, which are distributed among processes. Since there exists no shared memory between different processes, data exchange is demanded: the computation of the boundary points of each subdomain on time t lies on the values of their neighbouring points on time t-1, which are stored in the local memory of other processes. Hence, each process needs to communicate with all processes that hold the neighbouring subdomains and receive the necessary data chunks.

The pattern of communication is now straightforward: each process sends its boundary 2D planes to its 6 neighbours and receives 6 2D planes, each from a different neighbour, to accomplish the computation. To clarify this pattern, we employ Figure 3.8, which depicts the communication pattern for a 2D space.

We have chosen to utilize the offered Cartesian virtual topology of MPI, to distribute the 3D subspaces to a virtual 3D grid of processes, which better reflects the logical communication pattern of the processes, compared to linear ranking. We should note that a virtual topology acts as an advisor to the runtime system for the assignment of processes to physical processors in a manner that improves communication performance [20].

To store the data received from neighbouring processes, we have utilized the *halo* or *ghost layer* structure, meaning that we have extended the local matrix

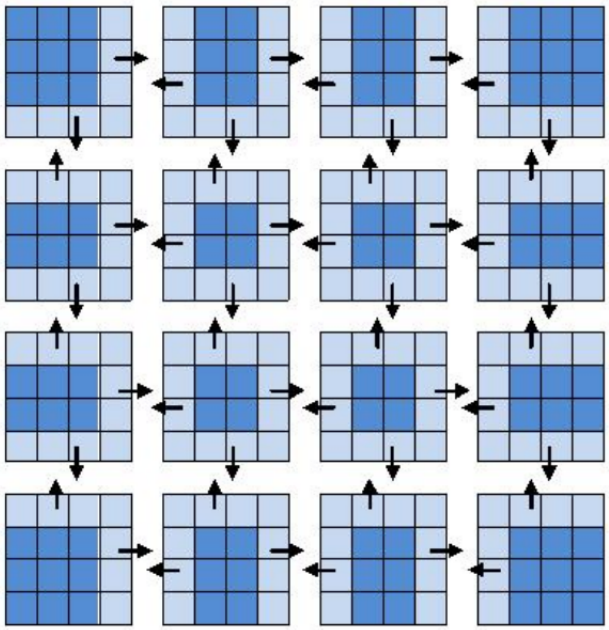


Figure 3.8: Parallel implementation of heat equation in two spatial dimensions. The spatial grid is a 16 x 16 square grid and is partitioned into 16 chunks. Each chunk is assigned to a different process. Pale squares indicate the data that must be exchanged between the processes. The bold arrows illustrate the flow of the exchanged data.

of each process by an outer layer, to host the received data. Halo exchange is depicted in Figure 3.9 for a 2D space.

Pseudocode 3.2: Parallel implementation of the 3D heat equation with MPI

```
T=512
/*Initializations*/
/*Creation of cartesian virtual topology-3D grid of Px x Py x Pz processes
    */
/*Scattering the initial matrix held by the root process to all processes
    */
/*Defining ranks of neighbouring processes*/
/*End of initializations*/
/*Computational kernel-executed by each process*/
for (t=0;t<T;t++)
{
  //Send and receive data
  Pack_data(2D boundaries of myPrevious to buffers)
  Isend(buffers to all neighbouring processes)
  IRecv(buffers from all neighbouring processes)
  Waitall(communications)
  Unpack_data(buffers to halo layers of myPrevious)

  //Computation
  for (i=1;i<=myX;i++)
    for (j=1;j<=myY;j++)
      for (k=1;k<=myZ;k++)
        myCurrent[x][y][z]=(myPrevious[x][y][z]+myPrevious[x-1][y][z]
                            +myPrevious[x+1][y][z]+myPrevious[x][y-1[z
                                ]
                            +myPrevious[x][y+1][z]+myPrevious[x][y][z
                                -1]
                            +myPrevious[x][y][z+1])/7;
  swap(myCurrent,myPrevious);
}
```

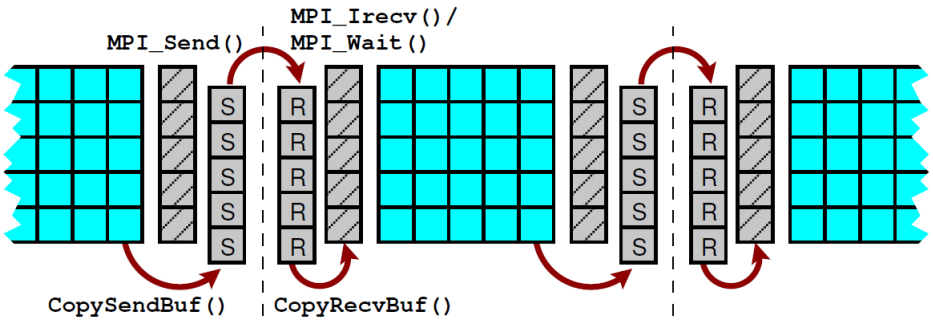
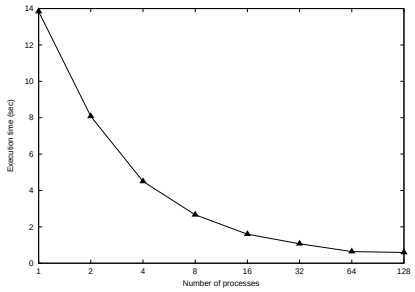


Figure 3.9: Halo communication for the Jacobi solver (illustrated in two dimensions here) along one of the coordinate directions. Hatched cells are ghost layers, and cells labeled "R" ("S") belong to the intermediate receive (send) buffer. The latter is being reused for all other directions. Note that halos are always provided for the grid that gets read (not written) in the upcoming sweep. Fixed boundary cells are omitted for clarity.

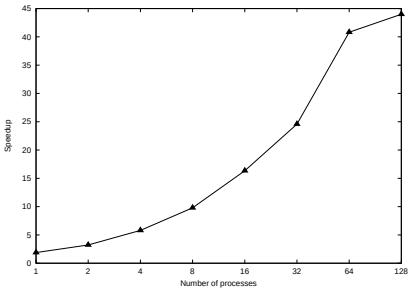
At this point, we should also explain the packing operations. Message passing with MPI requires that messages are stored in contiguous memory addresses, whereas communication in our application involves non-contiguous 2D planes of the 3D space. To perform communication operations, the sending process packs every message into a separate buffer and the corresponding receiving process unpacks the buffer to the appropriate ghost layer. Though MPI provides both packing/unpacking routines and the option to define datatypes, such as subarrays, specialised to serve communication operations, our implementation realizes user-defined packing and unpacking routines, for reasons of maintainability in the optimizations which follow in the next chapters.

3.2.4 Experimental results of the parallel implementation with MPI

In this section, we present the experimental results of the execution of our MPI implementation for the 3D heat equation. We have measured the execution time for three 3D spaces (128x128x128, 256x256x256 and 512x512x512) for 512 iterations over time. Figures 3.10, 3.11 and 3.12 show the results for the different 3D spaces. For each case we use Gigabit Ethernet as interconnection network and we utilize progressively 1, 2, 4, 8, 16, 32, 64 and 128 cores.

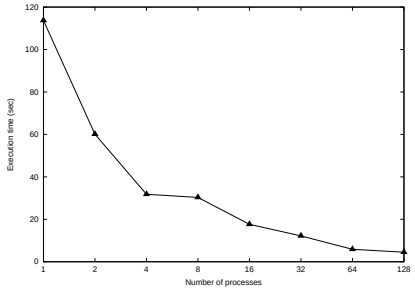


(a) Execution Time

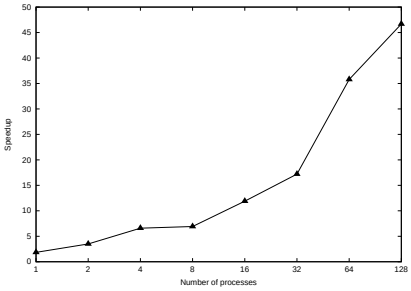


(b) Speedup

Figure 3.10: Results of heat equation with MPI for a 3D space 128x128x128

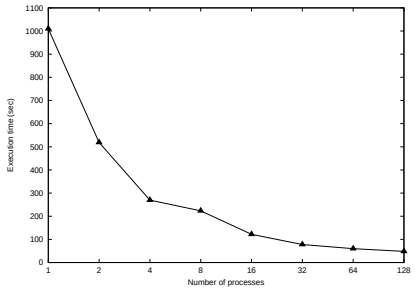


(a) Execution Time

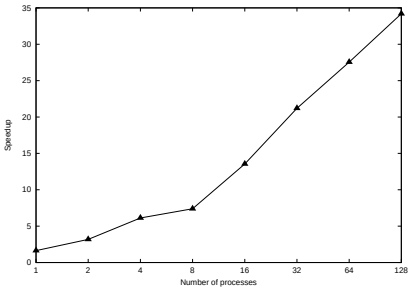


(b) Speedup

Figure 3.11: Results of heat equation with MPI for a 3D space 256x256x256



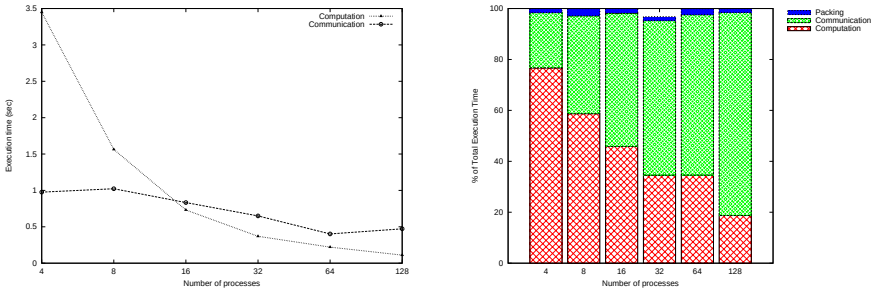
(a) Execution Time



(b) Speedup

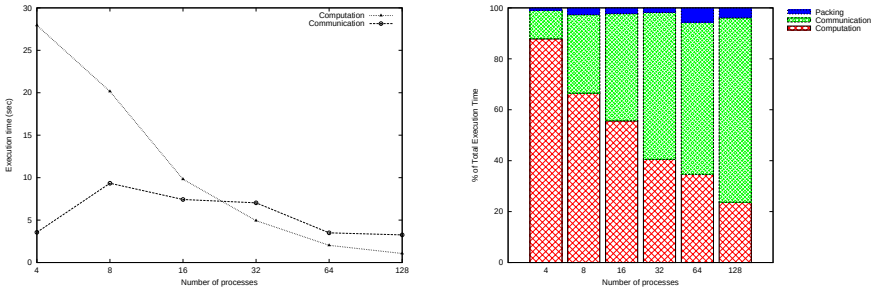
Figure 3.12: Results of heat equation with MPI for a 3D space 512x512x512

For the 3D space of dimensions 128x128x128, our MPI implementation scales linearly up to 16 processes. This observation does not hold for the other two cases, where speedup is linear up to 4 processes and performance decreases after this point. Though the efficiency of the parallel implementation is satisfactory, as it is about 45 times faster for small spaces and 36 times faster for the $512 \times 512 \times 512$ 3D space compared to the serial algorithm, we should focus on interpreting why speedup is not linear for all cases.



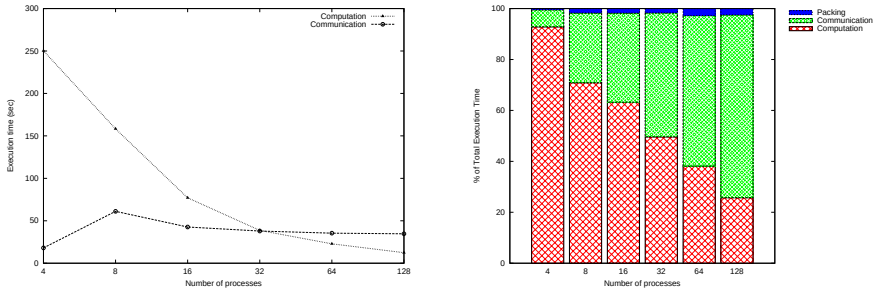
(a) Execution Time **(b)** Percentages of Total Time

Figure 3.13: MPI Communication/Computation breakdown for a 3D space 128x128x128



(a) Execution Time **(b)** Percentages of Total Time

Figure 3.14: MPI Communication/Computation breakdown for a 3D space 256x256x256



(a) Execution Time **(b)** Percentages of Total Time

Figure 3.15: MPI Communication/Computation breakdown for a 3D space 512x512x512

For a better understanding of the parallel program's performance, we have measured and decomposed total execution time to communication and computation time. Figures 3.13, 3.14 and 3.15 illustrate this breakdown. Regardless of the problem's size, computation scales as the number of MPI processes increases. We should remark, though, that, for the smaller problem size, computation time decreases in a faster rate, as the working set fits better to the available cache memory. We should also note that, as we progressively increase the number of processes above 32, each node launches more than two processes. Consequently, memory traffic to the shared L2 cache memory adds a small overhead to computation time.

The case of communication time presents more interesting aspects, concerning its influence to the overall performance. In all three cases of study, communication time does not scale with the number of processes. On the contrary, it appears to be increasing with an indefinable pattern. In the following chapter, we will analyze thoroughly the behavior of communication time and its impact on the performance of MPI programs, but for the purposes of this section, we focus on the point where communication time exceeds computation time, i.e. when we employ 32 MPI processes. Figures 3.13b, 3.14b and 3.15b clearly show that the percentage of communication time dominates the total execution time of the parallel program, and is therefore responsible for the degradation of performance.

3.3 Parallelization with hybrid MPI/OpenMP model

3.3.1 About hybrid MPI/OpenMP

The hybrid MPI/OpenMP model refers to a mixed programming style which employs MPI routines for message passing between distinct nodes and OpenMP directives to orchestrate parallelization of computation on cores lying on the

same node of a cluster. The first attempts to parallelize applications in this style are documented in [21], [22] and [23], with the appearance of clusters with SMP nodes. The aim of this programming style is to optimize performance of parallel implementations by exploiting shared memory within nodes for faster computation and communication operations. Though the case of 3D heat equation does not fall into typical patterns where a hybrid MPI/OpenMP implementation yields a better performance compared to pure MPI, we present its parallelization with hybrid MPI/OpenMP for reasons of completeness.

3.3.2 Parallelization of the 3D heat equation with hybrid MPI/OpenMP

The hybrid version of the 3D heat equation differs from the pure MPI implementation in the part of computations. In the pure MPI version, an MPI process holding a 3D subspace performs computations serially on its 3D matrices. In the hybrid version, computations are parallelized with the aid of OpenMP. This alteration also changes the number of MPI processes created for the parallel implementation: if there are 64 cores available, the standard MPI version would create 64 MPI processes, while the hybrid version may create 32 MPI processes, each one spawning 2 OpenMP threads, or respectively, 16 MPI processes spawning 4 OpenMP threads. The hybrid parallel implementation of 3D heat equation is depicted in Pseudocode 3.3.

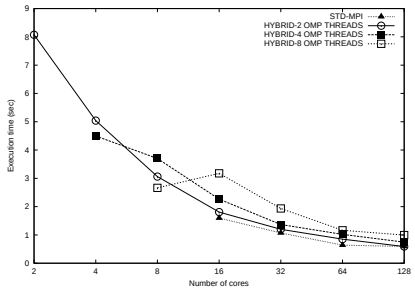
The block of code directing computations has been parallelized with OpenMP in the same way as in the pure OpenMP version, namely by applying the *omp for* directive to parallelize the outer loop of computations. In the following section, we compare the experimental results of the hybrid implementation compared to the pure MPI implementation.

3.3.3 Experimental results of the parallel implementation with hybrid MPI/OpenMP

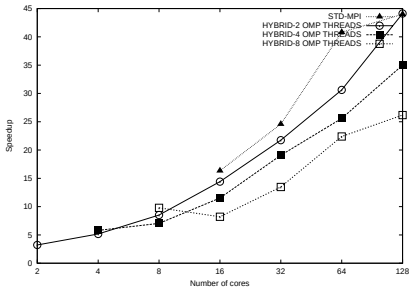
We have experimented with the hybrid parallel implementation of the 3D heat equation on 16 nodes of our cluster, connected with Gigabit Ethernet, for problems sizes of 128^3, 256^3 and 512^3 points of data. We have measured execution time for 2, 4 and 8 OpenMP threads, utilizing up to 128 cores in total. The corresponding results are illustrated in Figures 3.16, 3.17 and 3.18

Pseudocode 3.3: Parallel implementation of the 3D heat equation with hybrid MPI/OpenMP

```
T=512
/*Initializations*/
/*Creation of cartesian virtual topology-3D grid of Px x Py x Pz processes
    */
/*Scattering the initial matrix held by the root process to all processes
    */
/*Defining ranks of neighbouring processes*/
/*End of initializations*/
/*Computational kernel-executed by each process*/
for (t=0;t<T;t++)
{
  //Send and receive data
  Pack_data(2D boundaries of myPrevious to buffers)
  Isend(buffers to all neighbouring processes)
  IRecv(buffers from all neighbouring processes)
  Waitall(communications)
  Unpack_data(buffers to halo layers of myPrevious)

  //Computation
  #pragma omp parallel num_threads(p)
  {
    #pragma omp for private(i,j,k)
    for (i=1;i<=myX;i++)
      for (j=1;j<=myY;j++)
        for (k=1;k<=myZ;k++)
          myCurrent[x][y][z]=(myPrevious[x][y][z]+myPrevious[x-1][y][z]
                              +myPrevious[x+1][y][z]+myPrevious[x][y-1[z
                                  ]
                              +myPrevious[x][y+1][z]+myPrevious[x][y][z
                                  -1]
                              +myPrevious[x][y][z+1])/7;
  }
  swap(myCurrent,myPrevious);
}
```

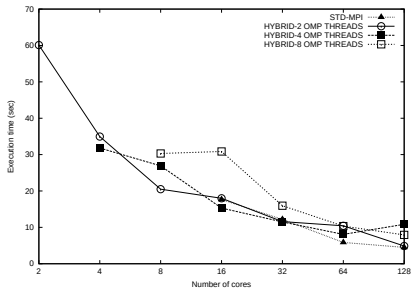


(a) Execution Time

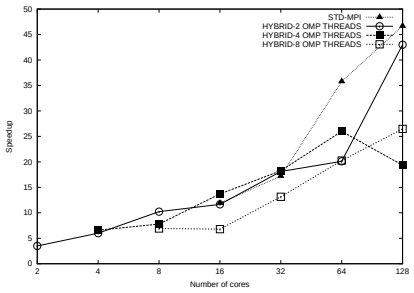


(b) Speedup

Figure 3.16: Results of heat equation with hybrid MPI/OpenMP for a 3D space 128x128x128

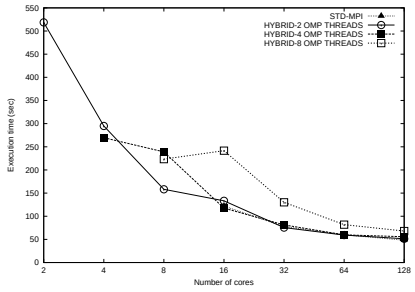


(a) Execution Time

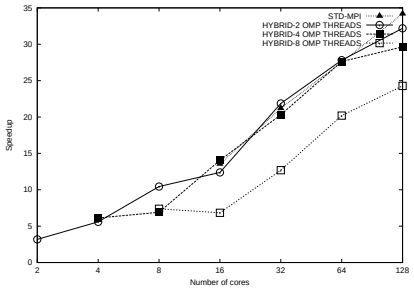


(b) Speedup

Figure 3.17: Results of heat equation with hybrid MPI/OpenMP for a 3D space 256x256x256



(a) Execution Time



(b) Speedup

Figure 3.18: Results of heat equation with hybrid MPI/OpenMP for a 3D space 512x512x512

For all problem sizes, when utilizing over 32 cores, regardless of the number of OpenMP threads used, the pure MPI parallel implementation has a better performance than the hybrid implementation. When utilizing less than 32 cores, though, the hybrid implementation with 2 or 4 OpenMP threads is more efficient that the pure MPI implementation, as 3D spaces of computations are large enough to benefit from OpenMP parallelization. In general terms, memory-bound algorithms are not prone to benefit from hybrid parallelization, as memory traffic degrades the performance instead of increasing it.

Chapter 4

Optimization techniques for the parallel 3D heat equation on clusters of multi-core SMP nodes

In this chapter, we discuss performance limitations of the parallel 3D heat equation when executed on a computer cluster and we introduce our effort to compromise between the overhead of parallelization cost and the gained parallel speedup. We present a set of optimization techniques which aim at reducing the execution time of our parallel implementations and achieving a better speedup, and their evaluation on our computer cluster.

4.1 Understanding performance limitations

In the previous chapter, we have studied the parallelization of the 3D heat equation with OpenMP and MPI and the corresponding experimental results. This analysis revealed several factors that limit the performance of our parallel programs. In this section, we enumerate these factors, their origin and their impact on our parallel programs.

4.1.1 A common performance model

The most common cost model used in algorithm design for large-scale multiprocessors specifies execution time T as a function of problem size N, number of processors p and other algorithm and hardware characteristics:

$$T = f(N, p, ...)$$

The execution time of a processor can be decomposed into computation time (T_{comp}), communication time (T_{comm}) and idle time (T_{idle}). Thus for processor j it holds:

$$T^j = T^j_{comp} + T^j_{comm} + T^j_{idle} \quad (4.1)$$

In our experiments, we have defined the execution time of the parallel programs as the time that elapses from when the first processor starts executing on the problem to when the last processor completes execution. Thus, the parallel execution time can be modelled as the average of the execution times of all processors:

$$T = \tfrac{1}{p}\sum_{j=1}^{p} T^j = \frac{1}{p}\sum_{j=1}^{p} T^j_{comp} + T^j_{comm} + T^j_{idle} \quad (4.2)$$

A simple and straightforward way of modelling computation time is to extract the number of operations (ops) required by an algorithm and multiply it by the CPU speed (in sec/op). The resulting estimate on a single processor is:

$$T^j_{comp} = ops\ of\ Algorithm \times CPU\ speed \quad (4.3)$$

The above model assumes that the CPU can be fed with data by the memory subsystem at a rate that can always cover the CPU's needs. However, this is not the case in modern systems containing multiple cores, where memory bandwidth contention affects the time of computation. We use the term *operational intensity* (OI) (in operations/byte) to mean floating point operations per byte of DRAM 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 memory rather than between the processor and the caches. Thus, operational intensity predicts the DRAM bandwidth needed by a kernel on a particular computer. In this case, an upper bound of the computational time is provided by the *Roofline model* [24] as:

$$T^j_{comp} = ops\ of\ Algorithm\ \times max(CPU speed, \tfrac{1}{Memory\ Bandwidth \times OI}) \quad (4.4)$$

The communication delays arise from various sources and they can be categorized in three classes:

- Message size dependent delay: This delay is a consequence of copying the message into buffers and due to finite capacity of the interconnection network. We capture these delays in a constant t_w, henceforth referred to as the per-word transfer time. In fact, if the bandwidth of a link B_w GB/s and the size of a word is k bytes, t_w is equalt to $\frac{k}{B_w}$
- Path length dependent delay: This delay is a consequence of the overhead at intermediate network interfaces and the signal propagation time on each link. This delay is presented by the constant t_h, also referred to as the per-hop time.
- Startup latency: This delay is independent of message length and number of hops. It is incurred due to handshaking between processors, buffer management and routing and is represented by t_s

The communication time model [25] for a message of size m traversing d hops of a cut-through routed network is:

$$T_{comm}^{per\ message} = t_s + t_h d + t_w m \quad (4.5)$$

On shared memory address systems, $t_h d$ is constant for all memory modules and can be subsumed into t_s. In distributed memory systems, it usually holds that $t_s >> t_h$ or $mt_w >> t_h$ and since d can be relatively small, the effect of the per-hop time can be ignored. In general terms, if an application sends $M = g(n, p)$, where n is the problem size and p is the number of processors and each message is of size $m = h(n, p)$, the model for communication time is:

$$T_{comm} = M \times (t_s + t_w \times h(n, p)) \quad (4.6)$$

Idle time can be difficult to determine, since it often depends on the order in which operations are executed. A processor may be idle due to lack of computation or lack of data. Lack of computation occurs when load is not balanced among processors executing a parallel program. Lack of data occurs when the computation and communication required to generate remote data are performed.

In parallel programming models of shared memory, as OpenMP, which perform communication through direct memory accesses, we are not able to measure communication time per se, but we assume that equation (4.4) finely models the time spent in the computation component and the time spent in memory bandwidth contention, or else the portion of time stalled in shared resources, which is the effective communication time of processors lying on the same node. Thereafter, a simplified performance model for OpenMP parallel programs is:

$$T_{omp} = T_{comp} + T_{idle} \quad (4.7)$$

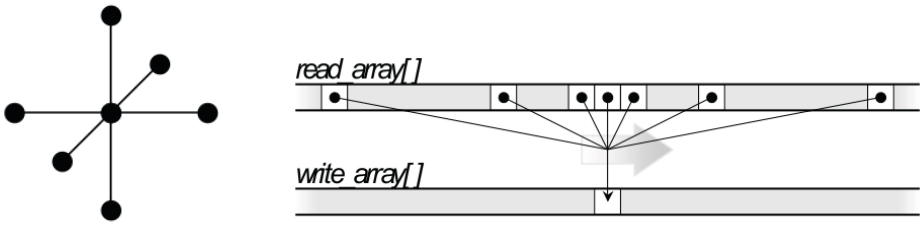
where T_{idle} is the aggregated processors' idle time when synchronization is required and load is imbalanced or when a block of code within a parallel region is executed by a single thread.

4.1.2 Performance limitations in computation time

While studying the execution times of our parallel implementation with OpenMP, we have mentioned that iterative stencil computations are memory-bound algorithms. Our stencil calculations for the heat equation perform global sweeps through data structures that are typically much larger than the capacity of the available data caches. Although the amount of data reuse is 4 out of 7 points of the stencil for a full iteration over space, the memory access pattern (Figure 4.1) along with the cache capacity restrictions limit potentiality of exploiting temporal or spatial locality. Another drawback of the heat equation is *read* and *write* operations are separated, therefore storage requirements are increased. As a result, our scheme of computation achieves a low fraction of theoretical peak performance, since data from main memory cannot be transferred

fast enough to avoid stalling the computational units on modern microprocessors [26].

The drawbacks of our scheme of computations have a major impact on our parallel implementation with OpenMP, especially in cases of large working sets, as is the 3D space of 512x512x512 points, namely a working set of 2GB. As we increase the number of threads, more than one threads are launched on the same package, sharing the L2 cache memory and conflict or capacity misses become inevitable. Even when threads are launched on different packages of the same node, the execution ends up in an excessive cache-to-cache data transfer between the two packages, which creates hotspots in the memory bus and degrade performance. Moreover, the limited memory bus bandwidth is quickly saturated, as more threads are introduced, under the pressure of serving multiple requests. In terms of equation (4.4), for each thread we add, the number of operations decreases, as the working set is distributed to threads, but memory traffic induces a decrease to OI, so computation time for each processor is $T^j_{comp} > \frac{T_{comp}\ of\ serial\ algorithm}{number\ of\ processors}$. In an overall, the speedup gained from the parallelization of the 3D heat equation on any shared memory model is always undermined by the overhead of fetching data from main memory.



(a) Conceptualization of stencil in 3D space **(b)** Mapping of stencil from 3D space onto linearly allocated array space

Figure 4.1: Stencil visualization

Parallelization of the heat equation with MPI does not suffer from the aforementioned factors to the same extent, as the total working set is split to the local memory of each MPI process. We should denote that in our experiments, MPI processes are mapped to processors in a different manner from OpenMP threads. MPI launches each process on a different node in a round-robin fashion. To activate all 8 cores on a node, the total number of MPI processes will be $number\ of\ nodes\ employed \times 8$, but the local working set will be $\frac{total\ working\ set}{number\ of\ nodes\ employed \times 8}$, while for our OpenMP parallel implementation, when all 8 cores on a node are activated, each thread operates on working set of size $total\ working\ set/8$, which is notably larger. Still, limitations hold when the total working set is very large, the local working set exceeds L2 cache capacity and data needs to be fetched from main memory. Moreover, when distinct MPI processes are launched on processors of the same node or package, they utilize a shared L2 cache and the memory bus. In conclusion, to improve performance

of the parallel versions of the 3D heat equation, it is necessary that we take into consideration the effect of the pattern of memory accesses and exploit locality and take advantage of memory hierarchies.

4.1.3 Performance limitations in communication time

As the performance model described in equation (4.7) denotes, OpenMP communication time is integrated into computation time and our analysis does not benefit from examining them separately. On the other hand, communication time is a large portion of the total execution time of our MPI parallel implementation and execution results of the parallel 3D heat equation with MPI have confirmed that communication time affects scaling and overall performance of the program. In the previous chapter, we have referred to the scaling pattern of communication time as indefinable. We will now show that this pattern is determined by the effect the pattern of halo exchanges and the equivalent message size.

Message passing for each MPI process is defined by the data distribution strategy that we follow. By distributing the 3D space of calculation to a 3D grid of processes, as defined in section 3.2.3, each process is expected to send six 2D planes to each neighbouring processes and receive an equal amount of data from all neighbouring processes on every time step. This approach is *coarse-grained*, as the ratio of computation to communication is relatively low. For a 3D space of dimensions $X \times Y \times Z$, $X = Y = Z = n$ and a 3D grid of $p_x \times p_y \times p_z$ processes and the demand to solve the heat equation on T time steps, each process performs computations of total complexity $O(T\frac{n^3}{p_x p_y p_z})$ and send and receive operations of overall complexity $O(T(2\frac{n}{p_x}\frac{n}{p_y} + 2\frac{n}{p_x}\frac{n}{p_z} + 2\frac{n}{p_y}\frac{n}{p_z})) = O(T\frac{n^2}{p_x p_y p_z})$. The ratio of computation to communication is:

$$\frac{Computation}{Communication} = \frac{O(T\frac{n^3}{p_x p_y p_z})}{O(T\frac{n^2}{p_x p_y p_z})} = O(n)$$

As noted in [27], we should expect a speedup equal to the ratio of computation to communication, namely a linear speedup, a prediction which is not in accordance with our experimental results. Equation (4.6) suggests that communication time to send a single message is the sum of a startup time plus the time required to transmit the message over the interconnection network. As the program nature is such that each message needs to be transmitted separately to a different process and we cannot accumulate distinct messages or employ a broadcast communication scheme, the parallel program is obliged to pay the"fee" of startup time for every message. Startup time is both software and hardware dependent. Although from the user's point of view data is transferred from user memory on one process to user memory on another process, the MPI library or the kernel inflicts the creation of a temporary copy in system memory at the sending node, to maintain the semantics of non-blocking send operation or to free the processor from having to wait on a slow network. Also, latency of the interconnection

network is added up on startup time. By increasing the number of MPI processes, the number of messages requiring to be transmitted concurrently over the interconnection network increases drastically and bandwidth contention occurs, affecting the per-word transfer time t_w and increasing the total communication time per message.

The per-word transfer time is also intensive to the size of the message. The total network bandwidth cannot be exploited as efficiently when large messages are transferred. Moreover, communication time per message is proportional to the size of the message, as indicated by the factor $t_w m$ of equation (4.5). Finally, packing and buffering messages to implement send/receive operations add up extra overhead to the startup time for messages of big size.

Experimental results of section 3.2.4 depict that scalability of communication time for the $128 \times 128 \times 128$ 3D space, which is a relatively small working set and is not drastically affected by the message size, breaks at 32 processors, not randomly. In accordance to our options of scheduling MPI processes, when utilizing up to 16 processors, each node launches at most a single process, while utilizing 32 processors implies that 2 processes execute on each node. Intra-node traffic is higher, causing stalls to communication operations; also, inter-node communication comes to play. Modern MPI implementations are aware of their processes' location on the cluster and bypass the interconnection network for inter-node communication, using shared memory features instead, but startup time for communication is affected anyhow. Yet we observe that communication time reduces again on 128 processors, where data distribution is coarse enough to eliminate the effect of message size. For the two larger working sets we measured, communication time rises and falls as we increase the number of processors, as a ping pong effect of the gradual decrease of message size and the network bandwidth contention. In conclusion, we are now able to define the parallel MPI implementation of 3D heat equation as a bandwidth-bound implementation.

4.1.3.1 Existence of idle time

We refer the reader back to section 3.1.3, where the baseline parallel implementation with OpenMP is presented and commented. For reasons of correctness, the task of swapping the arrays of computation is assigned to a single thread. While this task is performed, all threads but one remain idle, waiting for its execution to complete. This case is the aforementioned case of load imbalance. Unfortunately, there is no way to distribute computational load otherwise, so as to eliminate idle time in this particular approach of parallelization, without violating the order of execution. Thankfully, though, the overhead added by idle time to the total execution time is insignificant for the performance of the parallel program.

Idle time of the MPI parallel implementation is the result of the dependence of computational operations on the completion of communication operations. On every time step, data values of the previous time step need to be received by a process before current data values are computed. In effect, the interval

from the moment a process calls the MPI's non-blocking routines to send and receive the required messages to the moment when remote communications are completed and the call of the *MPI_Waitall()* function returns, notifying the process on the completion, is actually the idle time of each time step, as the processor is not blocked, but remains idle due to lack of data. Considering the stalls in communication, as described in the previous section, idle time turns to a significant portion of the total execution time as the bulk of messages grows. Henceforth, we will treat idle time as a portion of communication time, as we cannot measure separately the portion of communication time spent on chip and the portion spent remotely on the components of the interconnection network.

4.2 Minimizing computation time: Tiling

In this section, we introduce an optimization technique for the improvement of the performance of computation time. We exploit a proposed loop transformation technique that focuses on adapting the computational kernel to the underlying cache hierarchy, targeting to improve locality of references and reduce accesses to main memory, thereby eliminating memory bandwidth contention. We implement the technique on the serial computational kernel for a primary result analysis and present the equivalent experimental results.

4.2.1 Overview of tiling

Tiling is a well-known transformation which improves locality by moving reuses to the same data closer in time. It has been successfully applied to algorithms which involve stencil computations in two dimensions, where $O(n^3)$ computations are performed over $O(n^2)$ data and tiling can exploit $O(n)$ temporal reuse to greatly reduce cache misses. Rivera and Cheng [28] suggested the appropriate loop transformation to exploit temporal reuse and improve performance of a 7-point stencil computation, as is our case study, the 3D heat equation, is a 2D tiling transformation.

4.2.2 Implementing 2D Tiling

The 3D heat equation computational kernel, as introduced in chapter 2, accesses 7 rows of the 3D matrix *previous* in three adjacent 2D planes at the same time, on every time step. For a n^3 grid in space, the leading *previous*[i-1][j][k] and the trailing *previous*[i+1][j][k] array references have a distance of $2n^2$ elements. To support all group reuse between references on *previous*, the L1 cache needs to be able to hold two entire $n \times n$ planes, so only 3D arrays of size $M \times 32 \times 64$ can fully exploit reuse for our cluster's 32K L1 caches. Even for a larger L2 cache, group reuse is lost for large arrays.

The goal of the proposed 2D tiling method is to preserve group reuse across the outermost loop on the X dimension (referred to as i loop, in accordance with Figure 4.2) by effectively reducing the size of the plane during the execution of

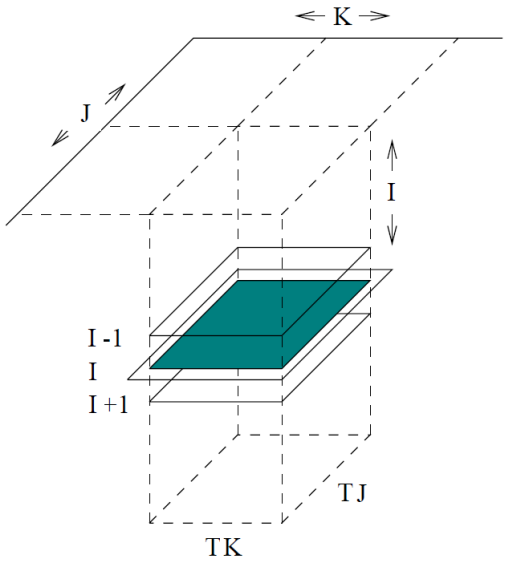


Figure 4.2: Access pattern of tiled 3D heat equation

the i loop. This is accomplished by *tiling* the inner two (j and k) loops. First, j and k are strip-mined to form tile-controlling loops jj and kk. Next, jj and kk are permuted to the outermost level. A tiled version of the 3D heat equation of Pseudocode 2.1 appears in Pseudocode 4.1.

Achieving reuse after applying this basic tiling transformation depends on the choice of tile dimensions TJ and TK. Since we tile only the j and k loops, the i loop iterates across all array planes but executed only iteration points inside a (n-2)*TJ*TK block. The shaded region represents the points in *iteration* space accessed on a single i loop iteration while the surrounding three unshaded regions represent the *array* points of *previous* accessed on a single i loop iteration. Because of the stencil pattern, the array tile consists of two $TJ \times TK$ regions

Pseudocode 4.1: Tiled 3D heat equation

```
//Iteration over time
for (t=0;t<T;t++)
{
  //Computing new values of interior points
  for (jj=1;jj<Y-1;jj+=TJ)
    for (kk=1;kk<Z-1;kk+=TK)
      for (i=1;i<X-1;i++)
        for (j=jj;j<min(jj+TJ,Y-1);j++)
          for (k=kk;k<min(kk+TK,Z-1);k++)
            current[i][j][k]=(previous[i][j][k]+previous[i-1][j][k]
                              +previous[i+1][j][k]+previous[i][j-1][k]
                              +previous[i][j+1][k]+previous[i][j][k-1]
                              +previous[i][j][k+1])/7;
    swap(current,previous);
}
```

located in array planes I-1, I+1 as well as a third $(TJ + 2) \times (TK + 2)$ region located in array plane I. In order to preserve all reuse within the $(n-2) \times TJ \times TK$ block, it is sufficient that the cache holds a $3 \times (TJ + 2) \times (TK + 2)$ subarray. Since the iteration space is partitioned into approximately $n^2/(TJ \times TK)$ blocks of size $(n - 2) \times TJ \times TK$, we multiply to obtain the total number of elements brought into the cache across the whole loop: $n^3(TJ + 2)(TK + 2)/(TJ \times TK)$. We can then divide by the cache line size L to estimate the number of cache lines fetched. However, since n^3/L is invariant under different tile sizes, we divide out this constant, resulting in the cost function: $Cost(TJ, TK) = (TJ + 2)(TK + 2)/(TJ \times TK)$, which is minimal when TJ and TK have the smallest possible difference. In other terms, square tiles are favored.

4.2.3 Experimental results for 2D tiling

We have measured the execution time of the serial computational kernel transformed with 2D tiling, as proposed in Pseudocodes 4.1 and 4.2, for three grids of $128 \times 128 \times 128$, $256 \times 256 \times 256$ and $512 \times 512 \times 512$ double precision points and for 512 iterations over time, testing a range of blocking factors. The results are organized in Table 4.1.

Grid Size	Tile Size	Size of tile with minimal execution time	Execution Time (sec)	Execution Time of tiled version / Execution Time of standard version
128^3	TJ [4...128] TK [4...128]	TJ 64 TK 128	26.07539	0.999
256^3	TJ [4...256] TK [4...256]	TJ 128 TK 256	209.62799	1.0005
512^3	TJ [4...512] TK [4...512]	TJ 64 TK 512	1637.85518	0.99

Table 4.1: Experimental results of the serial 3D heat equation with 2D tiling

The experimental results reveal that tiling does not work as an optimization to the 3D heat equation algorithm. Even when picking the size of tiles offering the minimal execution time, there is no effectiveness of the tiled version compared to the standard version. In fact, tiles sizes that provide the minimal execution time do not fit into our L1 cache. Still, this behaviour is not unexpected. Modern compilers are designed to perform internal loop transformations, to optimize execution and take full advantage of the cache hierarchy. The standard version is executed sequentially on a single core with an L2 cache of 6MB, thus large enough to store at least 3 $Y \times Z$ planes of the given 3D grids, hence neighbouring points of

the stencil fit in cache. Moreover, the standard serial version has been compiled using the -O3 flag of *gcc* and, since variation of execution times is limited to some milliseconds, we can assume that tiling optimizations may have been applied by the compiler already, making the comparison between the standard and the tiled version "unfair". Also, as pointed in [29], tiling could be more effective for larger grid sizes. Finally, the discussed tiling technique only affects temporal locality and not the degree of data reuse over time. Consequently, the slight variation of the memory access pattern offered by tiling is not adequate to improve the execution time of the computational kernel.

Experiments with tiling on our parallel implementation with MPI have been proven equally ineffective and are not worth to be demonstrated. The proposed 2D tiling has failed to optimize computation time, but in general terms, tiling is a valid scheme for optimizations on stencil computations and more complex implementations of tiling, which better apply to modern microprocessors' architectures, appear in bibliography and should be taken into consideration for future work.

4.3 Minimizing communication time: Compression

In this section, we introduce an optimization technique for the reduction of communication time of our MPI parallel implementation. We examine various compression algorithms, focusing on applying them onto the messages exchanged between MPI processes to reduce their size, aiming to reduce the factor $t_w m$ of communication time and improve the overall performance. We evaluate the compression ratio and time of each algorithm, apply the most promising to our MPI parallel implementation and present the respective experimental results.

4.3.1 Overview of compression

Formerly, we have documented the bandwidth-bound nature of the MPI parallel 3D heat equation. In order to reduce the impact of communications, we target to compress the exchanged messages in a transparent way, and increase the throughput of data exchanges, which is an important factor for the performance of parallel programs on clusters with communication networks of high latency and low bandwidth. During the past few years, many researchers in the field of scientific parallel computing have focused on compression techniques, either towards the development of efficient compression algorithms for double-precision floating point data [30], which are the dominant data types of scientific computations, or towards integrating compression of data on MPI protocol. CoMPI, an extended MPI run-time, developed by a group of researchers from University Carlos III in Madrid [31] has been the motivation for our work with compression on MPI messages.

Compression of MPI messages poses important requirements. The process of

compression must be lossless, so as to preserve information. Also, the compressor must produce the smallest overhead possible on execution time and memory requirements. In order to achieve benefit from compression, the additional computation time of the compression algorithm has to be lower than the time saved during the communication. On these terms, it seems more important to seek for fast compression rather than a high compression ratio. As for memory requirements, we favour algorithms that perform compression with the least possible requirements for extra buffering space, so as not to affect computation time on the attempt of minimizing communication time.

4.3.2 Compression algorithms

With respect to the requirements for fast and lossless compression, we have selected a group of compressors and we present them in this section.

4.3.2.1 RLE

Run Length Encoding [32], [33] is a simple method for lossless compression. It simply replaces repeated bytes with a short description of which byte to repeat, and how many times to repeat it. Though simple and obviously very inefficient fore general purpose compression, it can be very useful for compressing binary data (it is used in JPEG compression, for instance).

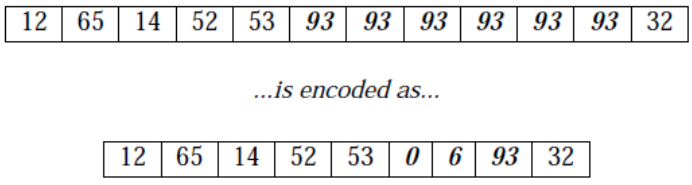


Figure 4.3: The principle of run length encoding

An example of how a run length encoding algorithm can encode a data stream is shown in Figure 4.3, where six occurrences of the symbol '93' have been replaced with three bytes: a marker byte ('0' in this case), the repeat count ('6'), and the symbol itself ('93'). When the RLE decoder encounters the symbol '0', which is used as the marker byte, it will use the following two bytes to determine which symbol to output and how many times to repeat the symbol.

The RLE implementation we use for our optimizations on MPI is developed by Markus Geelnard as a component of a compression library in C [34]. The particular method is an efficient one. Instead of coding runs for both repeating and non-repeating sections, a special marker byte is used to indicate the start of a repeating section. Non-repeating sections can thus have any length without being interrupted by control bytes, except for the rare case when the special marker byte appears in the non-repeating section (which is coded with at most

two bytes). For optimal efficiency, the marker byte is chosen as the least frequent (perhaps even non-existent) symbol in the input stream. The worst case compression result is: $compressed\ size\ (bytes) = \frac{257}{256} \times original\ size + 1\ (bytes)$.

4.3.2.2 Shannon-Fano

Shannon-Fano coding was invented by Claude Shannon [35] and Robert Fano [36] in 1949. The principle of Shannon-Fano coding is to replace each symbol with an alternate binary representation, whose length is determined by the probability or frequency of the particular symbol. Common symbols are represented with a few bits, while uncommon symbols are represented by many bits. The algorithm produces a very compact representation of each symbol, but it does not deal with the ordering or repetition of symbols or sequences of symbols.

The key to Shannon-Fano coding is to find new binary representations for each symbol. The solution is to make a histogram of the uncompressed data stream in order to compute the frequency of each symbol. A binary tree is then created by recursively splitting the histogram in halves, where each half in each recursion should weigh as much as the other half. The weight is $\sum_{k=1}^{N} symbolcount_k$, where N is the number of symbols in the branch and $symbolcount_k$ is the number of occurrences of the symbol k. The coder uses the tree to find the optimal representations for each symbol.The decoder uses the tree to uniquely identify the start and stop of each code in the compressed data stream: by traversing the tree from top to bottom while reading the compressed data bits, selecting branches based on each individual bit in the data stream, the decoder knows that a complete code has been read once a leaf node is reached.

To comprehend the algorithm's operations, we will use an example. Figure 4.4 shows the process of Shannon-Fano encoding step by step. For a stream of uncompressed data of 10 bytes (Figure 4.4a), the Shannon-Fano tree is built (Figure 4.4b) and the frequencies and encodings for each byte are computed (Figure 4.4c), leading to a compressed data stream of 24 bits or 3 bytes (Figure 4.4d). The drawback of the algorithm is the need for storage of the Shannon-Fano tree to perform decoding, which is though negligible for large input data streams.

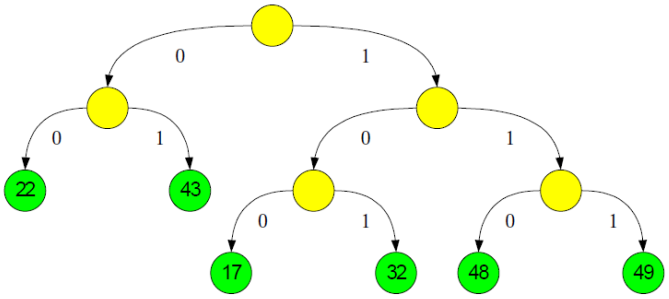
The Shannon-Fano implementation we use is also a component of Geelnard's compression library and has been implemented on the principles we described. The worst case compression result is: $compressed\ size\ (bytes) = \frac{101}{100} \times original\ size + 384\ (bytes)$.

4.3.2.3 Huffman

Huffman compression is an optimized version of the Shannon-Fano algorithm, developed by David Huffman, student of Shannon, in 1951 [37]. Huffman managed to avoid the major flaw of the suboptimal Shannon-Fano coding by building the tree from bottom-up instead of from the top down. In detail, instead of recursively splitting the histogram into equally weighing halves, the tree is built

32	22	22	43	49	22	22	17	48	43

(a) Uncompressed data stream, used as input



(b) Shannon-Fano tree for the data stream

Symbol	Frequency	Code
22	4	00
43	2	01
17	1	100
32	1	101
48	1	110
49	1	111

(c) Symbol frequencies and encoding for the uncompressed data stream

101	00	00	01	111	00	00	100	110	01

(d) Compressed data

Figure 4.4: Shannon-Fano compression: the encoding process for a 10-byte data stream

by successively joining the two least weighing nodes until there is only a single node left - the root of the tree. Huffman algorithm is optimal in the sense that any change in the binary encoding will result in a less compact representation. Because of Huffman's optimality, it is almost always preferred to Shannon-Fano and it is used widely on multimedia codecs.

We will use the Huffman implementation of Geelnard's compression library, as it offers a very compact form for the Huffman tree, requiring only 10 bits symbol on average. The worst case compression result is: $compressed\,size\,(bytes) = \frac{101}{100} \times original\ size + 320\,(bytes)$.

4.3.2.4 FPC

FPC compression has been recently developed by Martin Burtscher and Paruj Ratanaworabhan [38], in an attempt to build a lossless, single-pass, linear-time compression algorithm for double-precision floating point data that maximizes throughput.To achieve this, the algorithm does not handle the sign, exponent and mantissa of the IEEE-754 standard separately, but interprets all doubles as 64-bit integers, processing them with integer arithmetic to achieve fast compression. The key notion of the implementation is data prediction.

FPC compresses linear sequences of IEEE 754 double-precision floating-point values by sequentially predicting each value, xoring the true value with the predicted value, and leading-zero compressing the result.it uses variants of an *fcm* [39] and a *dfcm* [40] value predictor to predict the doubles. Both predictors are effectively hash tables. The more accurate of the two predictions, i.e., the one that shares more common most significant bits with the true value, is xored with the true value. The xor operation turns identical bits into zeros. Hence, if the predicted and the true value are close, the xor result has many leading zeros. FPC then counts the number of leading zero bytes, encodes the count in a three-bit value, and concatenates it with a single bit that specifies which of the two predictions was used. The resulting four-bit code and the nonzero residual bytes are written to the output. The latter are emitted verbatim without any encoding. FPC outputs the compressed data in blocks. Each block starts with a header that specifies how many doubles the block encodes and how long it is (in bytes). The header is followed by the stream of four-bit codes, which in turn is followed by the stream of residual bytes. To maintain byte granularity, which is more efficient than bit granularity, a pair of doubles is always processed together and the corresponding two four-bit codes are packed into a byte. In case an odd number of doubles needs to be compressed, a spurious double is encoded at the end. This spurious value is later eliminated using the count information from the header. Figure 4.5 depicts the operations of the compression process.

Decompression works as follows. It starts by reading the current four-bit code, decoding the three-bit field, reading the specified number of residual bytes, and zero-extending them to a full 64-bit number. Based on the one-bit field, this number is xored with either the 64-bit fcm or dfcm prediction to recreate the original double. This lossless reconstruction is possible because xor is reversible.

For the purposes of our experiments, we use Burtcher's and Ratanaworabhan's most recent implementation of FPC, found in [41]. The worst case compression result is: $compressed\ size\ (bytes) = \frac{17}{16} \times original\ size + 8\ (bytes)$.

4.3.2.5 LZO

Lempel-Ziv-Oberhumer (LZO) compressor [42] is a modern implementation of the Lempel-Ziv compression algorithm developed in 1977 [43]. This compressor follows a dictionary compression scheme; it does not use a predictive statistical model, as Huffman does, but it stores strings of previously input symbols in a dictionary. LZO is an optimized version, having a linear time complexity of

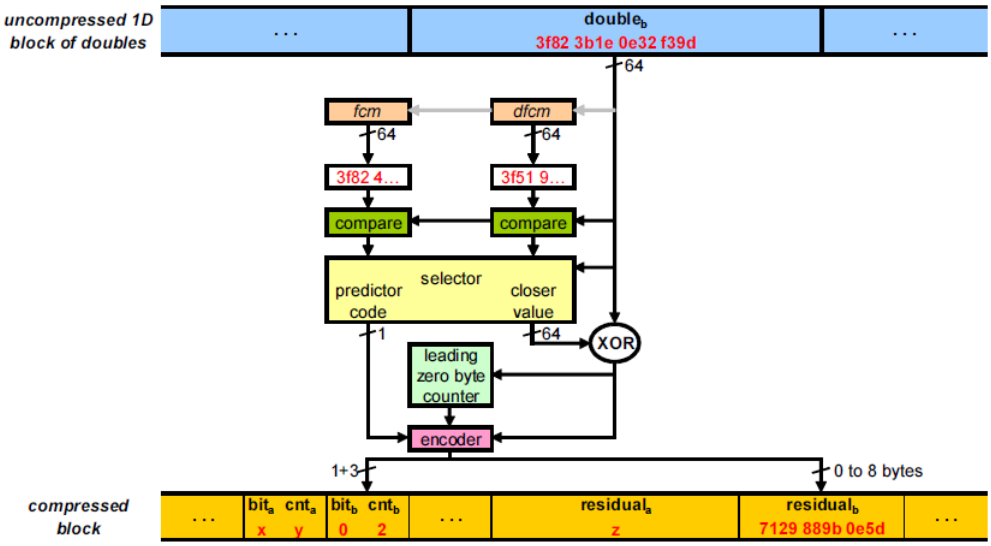


Figure 4.5: FPC compression algorithm overview

$N(d)$, where N is the total number of the input symbols and d is the size of the dictionary.

The idea behind the LZO compression algorithm is to take the RLE algorithm a few steps further by replacing sequences of bytes with references to previous occurrences of the same sequences. For simplicity, the algorithm can be thought of in terms of string matching. In written text, certain strings tend to occur quite often, and can be represented by pointers to earlier occurrences of the string in the text. The idea is, of course, that pointers or references to strings are shorter than the strings themselves. A string reference is typically represented by a unique marker, an offset count and the a string length. Depending on the coding scheme a reference can either have a fixed length or a variable length. The latter is often preferred since that allows the coder to trade reference size for string size (i.e. it may be worth the increased size in the reference representation if the string is long enough).

For our experiments, we use the *miniLZO* implementation developed by Oberhumer, which is a lightweight subset of LZO. It supports overlapping compression and in-place decompression of data blocks, whose size must be the same for compression and decompression. *miniLZO* compresses a block of data into *matches* (a sliding dictionary) and runs of non-matching literals to produce good results on highly redundant data and deals acceptably with non-compressible data. The worst case compression result is: $\textit{compressed size}\ (\textit{bytes}) = \frac{17}{16} \times \textit{original size} + 64\ (\textit{bytes})$.

4.3.3 Evaluation of compression algorithms

When data compression is used in transmission, the goal is to increase the transmission speed, which depends on the message size, the time required to generate the coded message and the time required to decode the message and recover the original ensemble. Therefore, it is necessary to evaluate each compression algorithm we have presented, with different sizes of messages and datasets of varying contents, including redundant datasets (*redundancy* of a compression buffer is defined as the percentage of entries with zero values). For this, we have created synthetic datasets of four types:

- Redundant: A buffer filled with zeros (100% redundancy level)
- Semi-Redundant: A buffer half filled with zeros in random places and half filled with random values (50% redundancy level)
- Random: A buffer filled with random values
- Stencil: The solution of the heat equation on a single dimension, i.e. a buffer of doubles within a limited range. We should note that, as the size of the buffer grows, data of the stencil testcase demonstrate a pattern.

All buffers contain double-precision floating point data. We have measured buffers of size 5,10,20,50,100,200,500,1000,1500 and 2000 kB, which is more or less the size of the messages sent and received in our MPI implementation of the parallel 3D heat equation. The metrics to evaluate each algorithm are the *compression ratio*, i.e. the ratio of the size of compressed data to the original size, and the aggregation of compression and decompression time. An important note on the utilization of compression algorithms is that, as all compression algorithms, apart from the FPC algorithm, are designed for compression of byte streams and not doubles, we had to typecast each 64-bit double to a stream of 8 bytes. This typecast had to be done for the FPC algorithm as well, because by design the FPC algorithm treats doubles as integers. Table 4.2 illustrates the results of this evaluation.

Shannon-Fano and Huffman algorithms have a fairly good compression ratio, but their performance in terms of compression/decompression time is poor compared to the other algorithms, so we have excluded them from our further study, as fast compression is our primary concern. RLE and LZO achieve a great compression ratio for data with some level of redundancy and data coming from a stencil computation, but result to a negative compression for the *Random* testcase. The best performing algorithm for random data is the FPC algorithm, also having a decent performance for the rest of the testcases. In an overall, compression algorithms demonstrate better compression ratios when datasets follow some pattern compared to random data. We should conduct further experiments to decide whether it is worth to apply compression techniques on sets of random numbers, for the purposes of reducing MPI communication time. We will proceed with presenting the integration of compression to the MPI parallel 3D heat equation.

Size (kB)	Pattern	Ratio					Time (msec)				
		RLE	SF	Huffman	LZO	FPC	RLE	SF	Huffman	LZO	FPC
5	Redun	0.1	12.5	12.5	0.9	6.5	0.03	0.11	0.09	0.04	0.02
5	Semi_R	49.0	57.2	55.8	50.1	67.0	0.04	0.87	0.78	0.04	0.03
5	Random	100.1	101.2	99.6	100.5	93.6	0.03	1.18	1.05	0.01	0.03
5	Stencil	100.0	104.3	102.5	100.5	89.0	0.03	1.22	1.08	0.01	0.03
10	Redun	0.0	12.5	12.5	0.7	6.4	0.07	0.20	0.18	0.05	0.05
10	Semi_R	47.9	52.8	52.0	48.2	66.5	0.08	1.24	1.10	0.08	0.05
10	Random	100.1	98.7	96.6	100.4	93.5	0.06	1.85	1.66	0.02	0.06
10	Stencil	100.1	102.0	100.9	100.4	88.6	0.06	1.89	1.71	0.02	0.05
20	Redun	0.0	12.5	12.5	0.5	6.3	0.14	0.39	0.35	0.07	0.09
20	Semi_R	47.9	51.2	50.8	46.8	67.0	0.16	1.95	1.77	0.15	0.10
20	Random	100.2	95.9	95.3	100.4	93.4	0.12	3.12	2.92	0.03	0.11
20	Stencil	100.1	101.1	100.3	100.3	88.4	0.11	3.20	3.00	0.01	0.10
50	Redun	0.0	12.5	12.5	0.5	6.3	0.35	0.96	0.85	0.13	0.24
50	Semi_R	48.0	50.3	50.0	45.3	66.5	0.39	4.06	3.80	0.40	0.26
50	Random	100.2	94.9	94.5	100.4	93.4	0.30	6.96	6.69	0.04	0.27
50	Stencil	100.2	100.8	100.1	100.4	88.2	0.28	7.02	6.76	0.07	0.25
100	Redun	0.0	12.5	12.5	0.5	6.3	0.70	1.92	1.70	0.28	0.45
100	Semi_R	48.7	50.3	50.1	45.7	67.8	0.78	7.65	7.26	0.80	0.53
100	Random	100.2	94.5	94.2	100.4	93.3	0.60	13.41	13.06	0.11	0.55
100	Stencil	78.1	88.9	86.8	79.1	69.3	0.60	11.86	11.91	0.14	0.47
200	Redun	0.0	12.5	12.5	0.5	6.3	1.40	3.88	3.45	0.63	0.80
200	Semi_R	48.6	50.0	49.9	45.7	67.2	1.71	14.82	14.30	1.63	1.10
200	Random	100.2	94.4	94.1	100.4	93.4	1.26	26.38	25.81	0.36	1.09
200	Stencil	55.3	70.5	66.9	55.7	50.1	1.28	18.06	18.88	0.34	0.88
500	Redun	0.0	12.5	12.5	0.4	6.3	3.50	9.86	8.71	1.71	2.17
500	Semi_R	48.3	49.8	49.7	45.3	67.0	4.27	36.14	35.19	4.04	2.70
500	Random	100.3	94.3	94.1	100.4	93.3	3.28	65.19	64.23	0.75	2.74
500	Stencil	35.0	47.1	46.9	35.7	33.2	3.30	31.72	31.29	1.17	2.24
1000	Redun	0.0	12.5	12.5	0.4	6.3	7.04	19.68	17.51	3.53	4.86
1000	Semi_R	48.5	49.9	49.8	45.5	67.2	8.59	71.92	70.22	8.12	5.45
1000	Random	100.3	94.3	94.1	100.4	93.3	6.70	130.06	127.80	1.34	5.46
1000	Stencil	24.8	36.9	36.8	25.3	25.0	6.70	50.53	50.05	2.55	4.75
1500	Redun	0.0	12.5	12.5	0.4	6.3	14.061	39.576	35.278	7.307	9.632
1500	Semi_R	48.2	49.7	49.6	45.3	67.0	17.315	143.387	139.985	16.334	11.03
1500	Random	100.3	94.2	94.0	100.4	93.3	13.75	259.969	257.421	3.004	10.913
1500	Stencil	20.2	32.5	32.4	20.6	21.4	13.622	82.978	82.739	5.555	8.319
2000	Redun	0.0	12.5	12.5	0.4	6.3	14.06	39.58	35.28	7.31	9.63
2000	Semi_R	48.4	49.8	49.6	45.3	67.1	17.32	143.39	139.99	16.33	11.03
2000	Random	100.3	94.2	94.1	100.4	93.3	13.75	259.97	257.42	3.00	10.91
2000	Stencil	17.5	29.8	29.7	18.0	19.2	13.62	82.98	82.74	5.56	8.32

Table 4.2: Compression Ratio and Compression/Decompression Time for double-precision data

4.3.4 Integration of compression into the MPI implementation

In the light of the results depicted in Table 4.2, we have selected the FPC compression algorithm to compress MPI messages. To justify this choice, we should sketch the features of our dataset and messages. As we have already already mentioned, at the entry to the iterative kernel the 3D space is initialized with constant non-zero values at its boundaries. During the first iterations of the computational kernel, most MPI messages exchanged are 100% redundant,

Pseudocode 4.2: Parallel Implementation of 3D heat equation with MPI and FPC compression

```
T=512
/*Initializations*/
/*Creation of cartesian virtual topology-3D grid of Px x Py x Pz processes
    */
/*Scattering the initial matrix held by the root process to all processes
    */
/*Defining ranks of neighbouring processes*/
/*End of initializations*/
/*Computational kernel-executed by each process*/
for (t=0;t<T;t++)
{
  //Send and receive data
  Pack_data(2D boundaries of myPrevious to buffers of doubles)
  FPC_Compress(buffers to compressed buffers)
  Isend(compressed buffers to all neighbouring processes)
  IRecv(compressed buffers from all neighbouring processes)
  Waitall(communications)
  FPC_Decompress(compressed buffers to buffers of doubles)
  Unpack_data(buffers of doubles to halo layers of myPrevious)

  //Computation
  for (i=1;i<=myX;i++)
    for (j=1;j<=myY;j++)
      for (k=1;k<=myZ;k++)
        myCurrent[x][y][z]=(myPrevious[x][y][z]+myPrevious[x-1][y][z]
                            +myPrevious[x+1][y][z]+myPrevious[x][y-1][
                                z]
                            +myPrevious[x][y+1][z]+myPrevious[x][y][z
                                -1]
                            +myPrevious[x][y][z+1])/7;
  swap(myCurrent,myPrevious);
}
```

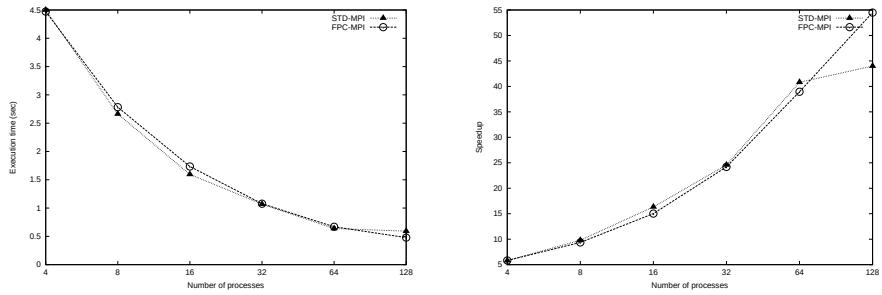
containing only zero values. While non-zero values propagate from the boundaries to the interior of the 3D space, redundancy level falls, until all values are non-zero. The image of each submatrix after $n/2$ iterations, for an initial grid of size n^3, is much like a set of random values, and such are MPI messages. It would take a large number of iterations to acquire a pattern similar to that of our *stencil* test in section 4.3.3.1, where neighbouring points have values differing in a few decimal points, but we have limited our times steps to 512 in all our experiments. Under these conditions, the most suitable compression algorithm is FPC algorithm, which has proven the most efficient when coming to random datasets. The parallel implementation of the 3D heat equation with MPI, where all messages are compressed with the FPC algorithm is depicted in Pseudocode 4.2

The implementation is straightforward: a "layer" of compression is added between packing and sending operations and a "layer" of decompression is added between receiving and unpacking operations. Each process, as a sender, packs the non-contiguous double-precision data to be sent to buffers, as in the standard MPI parallel implementation. These buffers are then compressed with the FPC algorithms and the new compressed messages are sent. After communication

operations are completed, received messages are decompressed into their original form and unpacked to the halo layers. The only thing left is to determine whether the overall performance of this version outcomes the performance of the standard MPI version, by examining the experimental results of the following section.

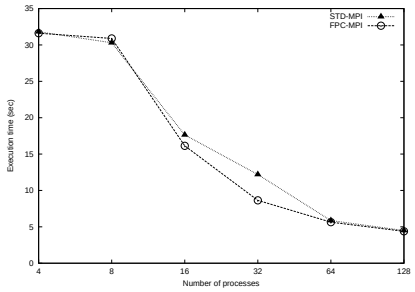
4.3.5 Experimental Results for MPI with compressed messages on the 3D heat equation

In this section, we present the experimental results of the execution of our MPI implementation for the 3D heat equation, with the integration of message compression/decompression with the FPC compression algorithm. We have measured the execution time for three 3D spaces (128x128x128, 256x256x256 and 512x512x512) for 512 iterations over time. Figures 4.6, 4.7 and 4.8 show the results for the different 3D spaces. For each case we use Gigabit Ethernet as interconnection network and we utilize progressively 1, 2, 4, 8, 16, 32, 64 and 128 cores.

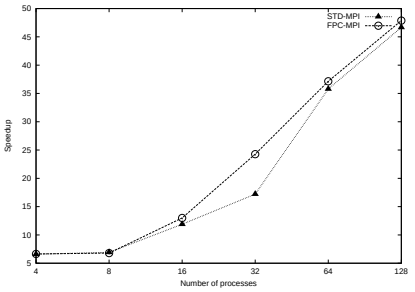


(a) Execution Time **(b)** Speedup

Figure 4.6: Comparative results of heat equation with MPI and FPC compression for a 3D space 128x128x128

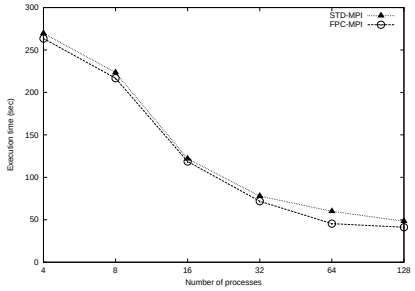


(a) Execution Time

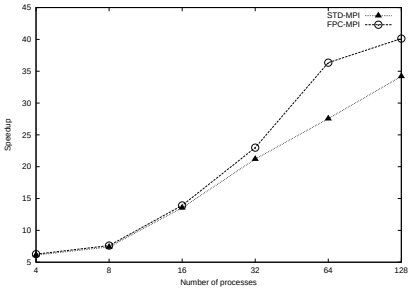


(b) Speedup

Figure 4.7: Comparative results of heat equation with MPI and FPC compression for a 3D space 256x256x256



(a) Execution Time



(b) Speedup

Figure 4.8: Comparative results of heat equation with MPI and FPC compression for a 3D space 512x512x512

The experimental results show that compressing MPI messages with FPC in the case of the 3D heat equation is a successful optimization technique. A better speedup is observed compared to the standard MPI implementation, whose execution time is greater in all cases utilizing more than 8 cores. If we define *performance improvement* as:

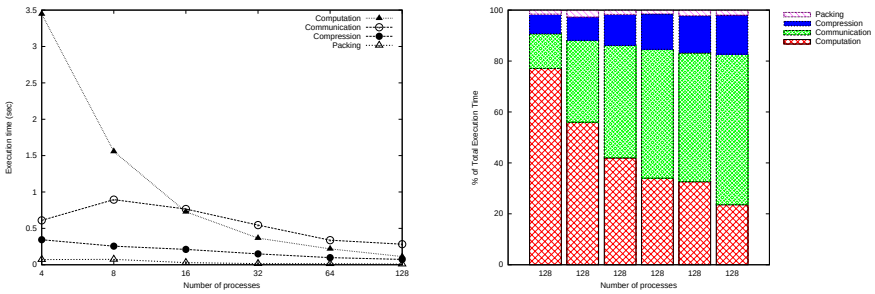
$$Improvement = \frac{exec\_time_{MPI\_STD} - exec\_time_{MPI\_FPC}}{exec\_time_{MPI\_STD}} \times 100\%,$$

Table 4.3 aggregates the results in speedup and improvement for the MPI_FPC version for 128 MPI processes.

Grid Size	Speedup		Improvement (%)
	MPI-STD	MPI-FPC	
128^3	44	54.5	19.27
256^3	46.7	47.9	2.5
512^3	34.2	40.1	14.79

Table 4.3: Speedup and Improvement of the MPI implementation of 3D heat equation with compression on 128 cores

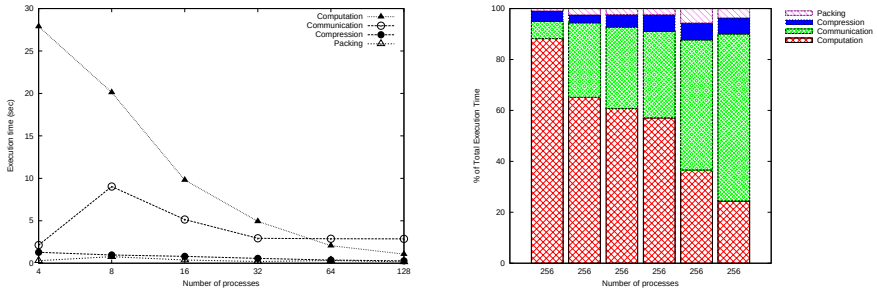
Figures 4.9a, 4.10a and 4.11a are more illustrative on the reduction of communication overhead. The difference between communication time and computation time is decreased, in relation to the standard MPI version, leading to a more efficient implementation. However, as mentioned before, we have to examine the aggregated communication and compression time versus communication time of the baseline version for the interpretation of results. Figures 4.9b, 4.10b and 4.11b indicate that the percentage of communication and computation time over total execution time when employing FPC compression is smaller than the percentage of communication time over total execution time in the baseline MPI implementation, which may be interpreted into a smaller communication overhead, less dominant on the performance of the parallel program.



(a) Execution Time **(b)** Percentages of Total Time

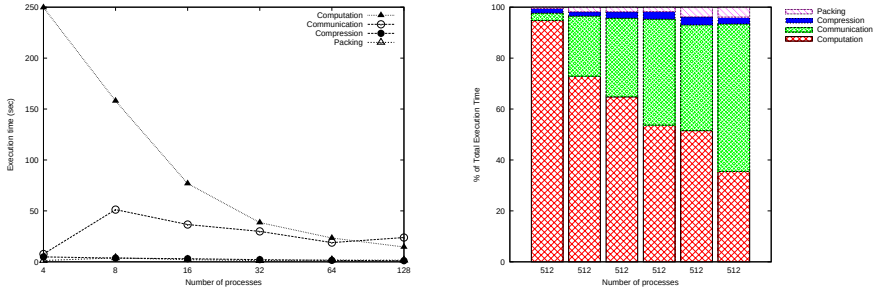
Figure 4.9: MPI Communication/Computation/Compression breakdown for a 3D space 128x128x128

For a fair evaluation of the use of compression on our MPI implementation, we have conducted additional experiments of the MPI-FPC version, where our initial 3D space is initialized with random values on both boundaries and interior. The respective experimental results are depicted in Figure 4.12a, in comparison to our MPI-STD version, also initialized with random values, for a 512^3 grid. Performance improvement is not as significant as in our previous case study, of zero-value interior points, though we should focus on the fact that there is no deterioration on execution time with the integration of compression. Figure 4.12b, which illustrates compression ratio for the different initializations



(a) Execution Time **(b)** Percentages of Total Time

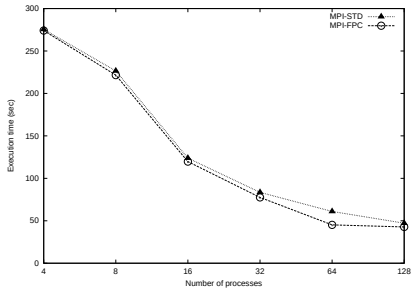
Figure 4.10: MPI Communication/Computation/Compression breakdown for a 3D space 256x256x256

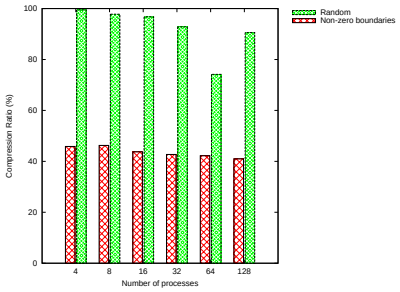


(a) Execution Time **(b)** Percentages of Total Time

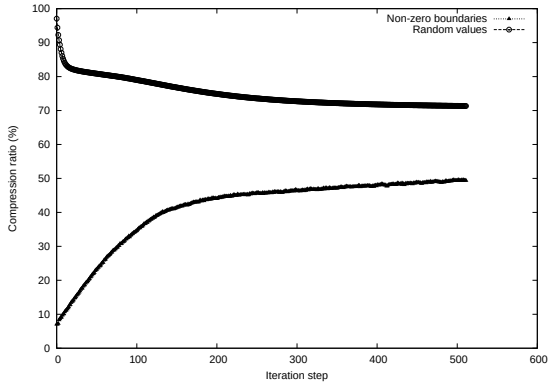
Figure 4.11: MPI Communication/Computation/Compression breakdown for a 3D space 512x512x512

we tested, shows how FPC algorithm achieves poor compression ratio for random data, thus not decreasing MPI messages enough to diminish communication overhead. Figure 4.12c justifies our assumptions that compression with FPC is appropriate for the dataset of the parallel 3D heat equation. For initialization with random data, compression ratio is high during the first steps of iteration, where data of the buffers to be compressed and sent exhibit no pattern, and falls as the effect of the stencil computation data smooths the difference between data points. On the contrary, when initializing only the boundaries of the 3D space on which we solve heat equation, compression is very efficient during the first iteration steps, where buffers of messages are highly redundant and tends to be less efficient as non-zero values spread to the interior. For a higher number of iterations, i.e. high enough to reach convergence, where a pattern appears, we should see a reduced compression ratio. In conclusion, compressing MPI messages with FPC optimizes the overall performance of the parallel 3D heat equation.





(a) Execution Time for a 3D space initialized with random data values **(b)** Compression Ratio for different initialization



(c) Compression Ratio per time step for different initializations

Figure 4.12: Comparative results of MPI with FPC compression for a 3D space 512x512x512 of different initialization

4.4 Concealing communication overhead: Overlapping Communication/ Computation

In this section, we propose an optimization technique for executing communication and computational operations concurrently, to eliminate the effect communication overhead appearing in our parallel implementation of the 3D heat equation with MPI due to network bandwidth contention. We discuss the potential of the technique, the ways to implement it and the application and experimental results on heat equation.

4.4.1 Overview of overlapping communication with computation

The reader should be familiar by now to the phenomenon of limited efficiency of the MPI parallel 3D heat equation due to communication pitfalls. In this study, we have sketched communication overhead over a limited number of processors. On modern clusters, though, where thousands of processors coexist and computation scales up with the number of cores, the increasing communication time, which exceeds 95% of total execution time, determines total execution time. Even in a favourable scenario, where communication time remains constant as the number of utilized processors increase - for instance, in case we compress MPI messages as described in the previous section - performance is still dominated by communication overhead. The idea of overlapping computation and communication [44], [45] aims at concealing communication overhead by exploiting the period of time while communication operations are carried out and perform computational operations in parallel.

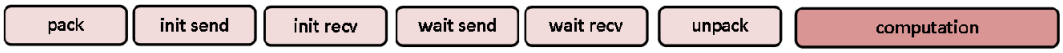


Figure 4.13: Non-overlapping implementation of heat equation: execution pattern for a single thread for each time step

In our MPI implementation of 3D heat equation, each process receives messages from all neighbouring processes before proceeding with computation of new values of the 3D subspace assigned to it, as in Figure 4.13. However, information received from neighbouring processes is only necessary for computing new values of points resting on the boundary of the subspace. Thus, if we regard computation of interior points and computation of boundary points as two different tasks, the former may be executed concurrently with communication. Then, execution time would obey a model similar to the following:

$$T = max(T_{comm}, T_{comp\_interior}) + T_{comp\_boundaries}$$

which, at first sight, seems minor compared to the non-overlapping scheme, where $T = T_{comm} + T_{comp}$. However, to achieve parallel execution of computation and communication, re-engineering of our standard MPI implementation of 3D heat equation is demanded and new performance-bounding issues occur and require careful treatment to prevent them from degrading parallel performance.

4.4.2 Implementation Techniques for the overlapping of communication and computation

4.4.2.1 Overlapping with non-blocking communication functions

Figure 4.14 illustrates how total tranfer time of a message for a point-to-point communication of MPI depends on local operations and remote operations.

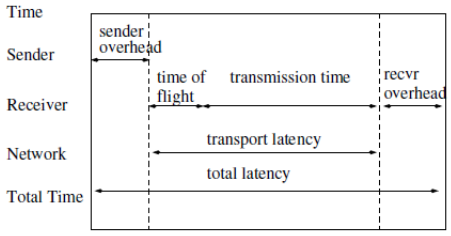


Figure 4.14: Message transfer time for a point-to-point MPI communication operation

Pseudocode 4.3: Overlapping computation and communication by using non-blocking functions

```
initiate_non_blocking_communication()
do
  compute()
while (communication ≠ done)
```

Sender's and receiver's overhead is time spent on local operations as buffering and unbuffering a message to/from internal buffers and generating acknowledgement for the correctness of the operation. Time of flight is what we earlier defined as *latency* for modern interconnection networks and corresponds to the amount of time it takes for the first bit of the message to reach the receiver's side. The actual transmission time is the factor $t_w m$ of equation (4.6), i.e. the size of message in words multiplied by the per-word transfer time, which depends on the bandwidth of the link. If we sum up sender's overhead, receiver's overhead and time of flight, we acquire the startup time t_s of equation (4.6).

Transmission time through the interconnect turns to be the largest portion of communication time while the number of processors increases. If blocking communication functions are employed, all operations of the sending/receiving processor are blocked until the message is transferred. If non-blocking communication functions are employed, processors are free to perform any computational operations during transmission time. The latter is the key to achieve an overlapping scheme for computation and communication. Pseudocode 4.3 is a naive implementation of this scheme. We refer the reader to [45], where this method is analyzed extensively.

The presented scheme for overlapping has been very popular during '90s and '00s, when clusters were composed of uniprocessors connected via high performance interconnection networks. Although it seems appealing and effective and relatively simple to implement, it has some well concealed drawbacks. Firstly, only a certain portion of communication time is actually overlapped, namely transmission time through the interconnection network, as processors remain busy during local operations. Modern protocols, as Ethernet, demand data copies between the application and the kernel, which is taken up by the core launching an MPI process and may prove time-consuming. Consequently, the

Pseudocode 4.4: Overlapping computation and communication with helper threading

```
#pragma omp parallel num_threads(2)
{
  if (my_thread_ID=0)
    communicate()
  else
    compute()

  #pragma omp barrier
  /*other operations*/
}
```

target of overlapping total communication time with computation time is affected by the protocols of the interconnection network and only high performance interconnects may accomplish it. Also, operations as packing and unpacking data into or from buffers before and after communication operations are de facto serialized. For these and more reasons, in modern clusters, built from multicores, the use of this method has subsided and been substituted by the method we will describe in the following subsection.

4.4.2.2 Overlapping with helper threading

The technique outlined in this subsection involves the use of a hybrid MPI/OpenMP model for the overlapping of computation and communication. As noted in [46], such models are natural for modern clusters, made up of shared memory nodes, where MPI can be used for inter-node communication and shared memory can be exploited by OpenMP for intra-node communication. The motivation of applying this particular model derives from the imbalance in performance of computation and communication as we increase the number of threads: instead of assigning both computation and communication on the same execution thread, we divide threads into computation and communication threads. Shared memory constructs allow each pair of computation and communication threads to run in parallel on the same node of an SMP cluster in an asymmetric fashion. Thereby, communication and computation are overlapped and, at the same time, a better balance is achieved, as we exclude some threads from computational operations, which scale up anyway, and devote them on the non-scaling communication operations.

The model is implemented as following: each MPI process spawns two OpenMP threads, one to undertake all operations relative to communication and one to undertake computations. Pseudocode 4.4 is an overview of this general model.

There are two highlights concerning the model: a parallel region of two OpenMP threads is initiated before sharing communication and computation tasks among threads, and a barrier is added to correctly orchestrate the execution of the threads.

The proposed scheme has certain advantages over the aforementioned over-

lapping scheme with non-blocking communication functions. Firstly, packing and unpacking operations and any other operation relevant to communication is assigned to the communication thread and executes in parallel with computation, instead of being serialized. Moreover, local communication operations are usually executed by one of the cores of the node, therefore total communication time is overlapped with computation, enabling the model to execute efficiently in commodity interconnects. On the other hand, this hybrid MPI/OpenMP model needs extra consideration on the mapping topology of MPI processes and OpenMP threads, such that intra-node memory traffic and OpenMP startup overhead remain low. Another consideration is the fact that when utilizing p processors, we actually overlap computation and communication for $p/2$ processors of a pure-MPI version. However, since communication time is dominating overall execution time when employing a large number of processors, we expect an improved overall performance with this scheme. Finally, synchronization points are necessary in this model to preserve the semantics of the program.

4.4.3 Overlapping implementations of 3D heat equation

4.4.3.1 Overlapping with non-blocking communication functions

Figure 4.15: Overlapping implementation of heat equation with non-blocking communication: execution pattern for a single thread for each time step

Overlapping with non-blocking communication functions is pretty straightforward. We transform our standard MPI implementation by dividing computations to separate blocks of code: computations on the interior points are executed in parallel with communication operations and computations on the boundary points are executed on completion of halo exchanges. All threads of execution are symmetric and their execution pattern, displayed in Figure 4.15, is only slightly altered, compared to the standard MPI version and has minimal requirements in programming effort. Pseudocode 4.5 implements this version.

4.4.3.2 Overlapping with helper threading

Implementing overlapping of computation and communication with the technique of helper threading on the 3D heat equation requires a careful approach and extensive re-engineering, as a hybrid MPI/OpenMP model is employed. Figure 4.16 visualizes the execution pattern for a communication and a computation thread, which are asymmetric in this case. While the computation thread executes computations in the interior of the 3D subspace, the communication thread

Pseudocode 4.5: Overlapping implementation of the 3D heat equation with MPI and non-blocking communication functions

```
T=512
/*Initializations*/
/*Creation of cartesian virtual topology-3D grid of Px x Py x Pz processes
    */
/*Scattering the initial matrix held by the root process to all processes
    */
/*Defining ranks of neighbouring processes*/
/*End of initializations*/
/*Computational kernel-executed by each process*/
for (t=0;t<T;t++)
{
  //Send and receive data
  Pack_data(2D boundaries of myPrevious to buffers)
  Isend(buffers to all neighbouring processes)
  IRecv(buffers from all neighbouring processes)

  //Computation of interior points
  for (i=2;i<=myX-1;i++)
    for (j=2;j<=myY-1;j++)
      for (k=2;k<=myZ-1;k++)
        myCurrent[x][y][z]=(myPrevious[x][y][z]+myPrevious[x-1][y][z]
                            +myPrevious[x+1][y][z]+myPrevious[x][y-1][
                                z]
                            +myPrevious[x][y+1][z]+myPrevious[x][y][z
                                -1]
                            +myPrevious[x][y][z+1])/7;
  Waitall(communications)
  Unpack_data(buffers to halo layers of myPrevious)
  //Computation of boundaries
  compute_boundaries_of_subspace()
  swap(myCurrent,myPrevious);
}
```

performs packing,send/receive operations and unpacking. After their tasks are completed, threads meet at a barrier, before one of them begins computation of data points at the boundaries. Pseudocode 4.6 sketches the described implementation, with certain differentiations which are discussed below.

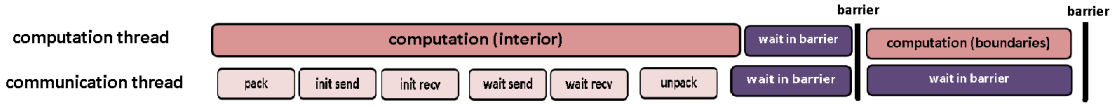


Figure 4.16: Overlapping implementation of heat equation with helper threading: execution pattern for a pair of threads for each time step

An OpenMP parallel region of two threads is created once, embracing all iteration loops, instead of creating a parallel region within the outer loop, to avoid a multiple overhead of thread creation. It is thereby necessary to declare loop variables as private and it has proven more efficient to declare the 3D matrices involved in computations as *firstprivate*, namely private but initiated with their values before entering the parallel region, instead of keeping them shared among threads. We have assigned computation of boundaries to the communication thread, to exploit possible data reuse on a cache level among computation of boundaries and the precedent unpacking operation. Since communication thread does not enter the block of code where computation of boundaries is defined, unless all communication operations have finished, there is no need for synchronization of threads at this point. Before proceeding with swapping the 3D matrices, though, a barrier is explicitly declared to synchronize threads and flush memory.

Code segments are assigned to threads according to their thread-id to achieve an SPMD (Single Process, Multiple Data) model of work sharing. Though OpenMP offers constructs for the parallel execution of code segments, as the *omp section* construct, it is more efficient to preserve the mapping of each code segment to a certain thread to exploit L1 instruction and data cache locality, i.e. computation is always executed by the thread of id 0 and communication by the thread of id 1.

Another important consideration has been the topology mapping of MPI processes and OpenMP threads to processors of each node of our cluster, also known as processor affinity. Given the architecture of each node, the most efficient pattern is to launch at most 2 MPI processes on each 4-core package, so that spawned OpenMP threads run on cores of the same package and share the L2 cache hierarchy, avoiding latency on accessing shared data. For the implementation of this mapping, we have employed a system call of the Linux operating system, used by its scheduler to bind processes onto specific cores, the *sched_affinity* system call, which given the id of a thread and an affinity mask, determines on which core the process can run, according to the affinity mask.

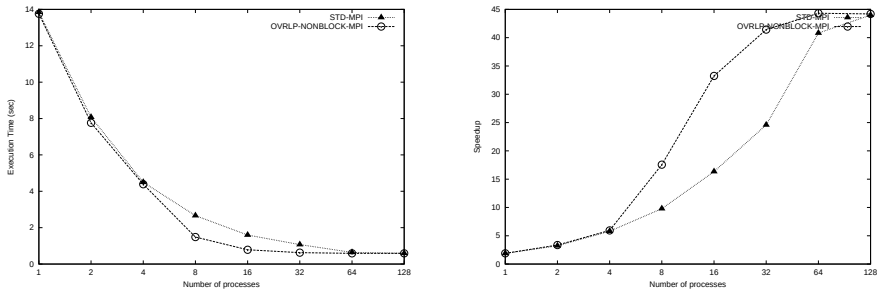
Pseudocode 4.6: Overlapping implementation of the 3D heat equation with hybrid MPI/OpenMP

```
T=512
/*Initializations*/
/*Set affinity on processors*/
/*Creation of cartesian virtual topology-3D grid of Px x Py x Pz processes
    */
/*Scattering the initial matrix held by the root process to all processes
    */
/*Defining ranks of neighbouring processes*/
/*End of initializations*/
/*Computational kernel-executed by each process*/
#pragma omp parallel private(t,i,j,k) firstprivate(myCurrent,myPrevious)
    num_threads(2)
{
  for (t=0;t<T;t++)
  {

    if (my_thread_ID==0) /*Computation thread*/
    {
      //Computation of interior points
      for (i=2;i<=myX-1;i++)
        for (j=2;j<=myY-1;j++)
          for (k=2;k<=myZ-1;k++)
            myCurrent[x][y][z]=(myPrevious[x][y][z]+myPrevious[x-1][y][z]
                                +myPrevious[x+1][y][z]+myPrevious[x][y
                                    -1][z]
                                +myPrevious[x][y+1][z]+myPrevious[x][y][
                                    z-1]
                                +myPrevious[x][y][z+1])/7;
    }
    else if (my_thread_ID==1) /*Communication thread*/
    {
      Pack_data(2D boundaries of myPrevious to buffers)
      //Send and receive data
      Isend(buffers to all neighbouring processes)
      IRecv(buffers from all neighbouring processes)
      Waitall(communications)
      Unpack_data(buffers to halo layers of myPrevious)
    }

    if (my_thread_ID==1)
    {
      //Computation of boundaries
      compute_boundaries_of_subspace()
    }

    #pragma omp barrier

    swap(myCurrent,myPrevious);
  }
}
```

4.4.4 Experimental results for communication/computation overlapping

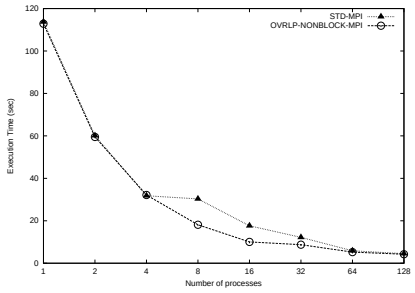
In this section, we present the experimental results from the execution of our implementations of overlapping for the 3D heat equation. We have measured the execution time for three 3D spaces (128x128x128, 256x256x256 and 512x512x512) and for 512 iterations over time. Gigabit Ethernet is used as interconnection network and we utilize progressively 1, 2, 4, 8, 16, 32, 64 and 128 cores (2 up to 128 cores for the helper threading version).

Figures 4.17, 4.18 and 4.19 exhibit the experimental results for our implementation where communication and computation are overlapped with the use of non-blocking communication functions, compared to the results of the standard MPI parallel version of 3D heat equation. In general, this version suffers from the same problems as the pure MPI version, but achieves a better speedup, especially for large 3D spaces, confirming the theoretical model which assumes performance improvement. If we were able to measure the implementation utilizing a high performance interconnection network, e.g. Myrinet, instead of Gigabit Ethernet, we should see a much more notable increase in speedup.

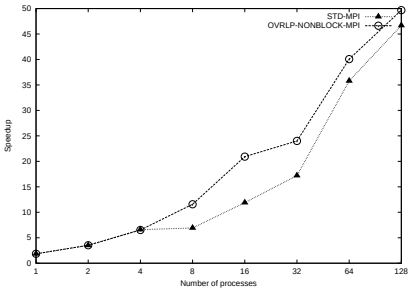


(a) Execution Time **(b)** Speedup

Figure 4.17: Comparative results of heat equation for overlapping computation/communication with non-blocking communication functions for a 3D space 128x128x128

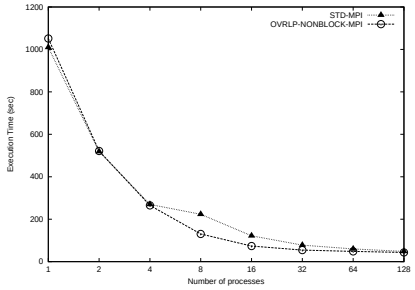


(a) Execution Time

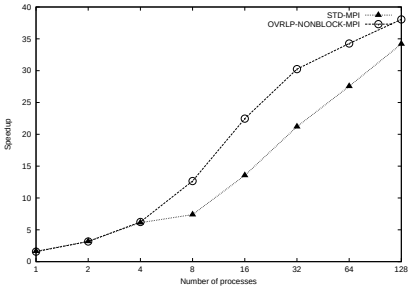


(b) Speedup

Figure 4.18: Comparative results of heat equation for overlapping computation/communication with non-blocking communication functions for a 3D space 256x256x256



(a) Execution Time

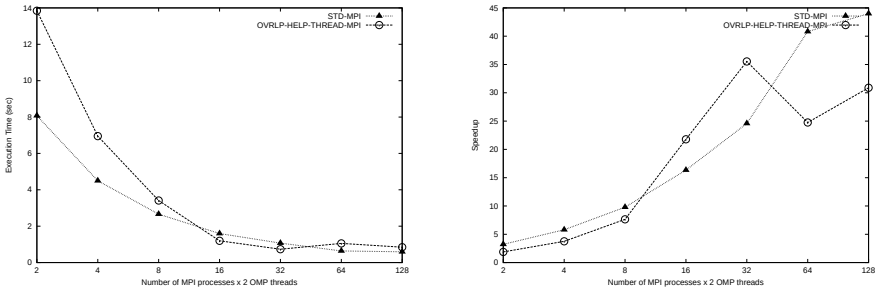


(b) Speedup

Figure 4.19: Comparative results of heat equation for overlapping computation/communication with helper threading for a 3D space 512x512x512

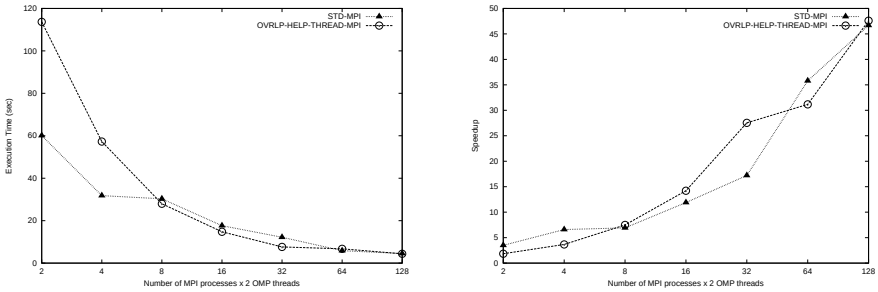
Figures 4.20, 4.21 and 4.22 illustrate the experimental results for our implementation of overlapping with hybrid MPI/OpenMP, or else helper threading. We observe that for the smaller grids of 128^3 and 256^3 data points, the technique does not induce any remarkable improvement on performance and the pure MPI implementation scales up better in most cases. We should note here that we do not expect the helper threading implementation to perform better than the pure MPI implementation for few cores, where computation time is still large and communication time is not yet an overhead. Improvement is expected mainly in those combinations of space size and number of utilized cores where the MPI version's communication and computation are almost equal; referring to the experimental results of section 3.2.4 , this proportion occurs on 16 or 32 cores. Indeed, measurements on the helper threading version verify each superiority compared to the standard MPI version for 16 and 32 cores. Performance measurements on the $512 \times 512 \times 512$ space are encouraging for the success of

overlapping with helper threading as an optimization. A total speedup of 39.9 is observed on 128 cores, corresponding to an improvement of about 20% against the pure MPI version. Consequently, helper threading can be considered as a promising optimizing technique for parallel applications with extensive communication operations on modern clusters.



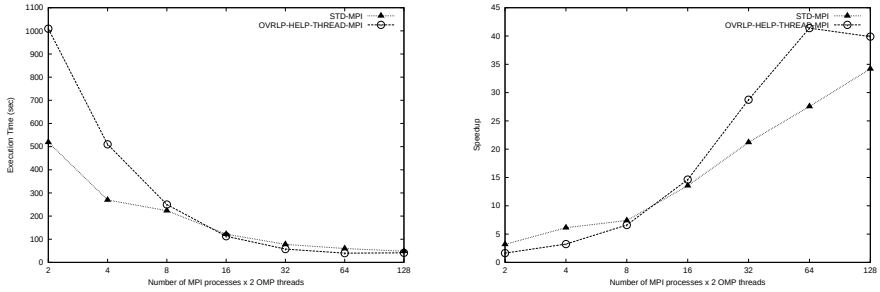
(a) Execution Time **(b)** Speedup

Figure 4.20: Comparative results of heat equation for overlapping computation/communication with helper threading for a 3D space 128x128x128



(a) Execution Time **(b)** Speedup

Figure 4.21: Comparative results of heat equation for overlapping computation/communication with helper threading for a 3D space 256x256x256



(a) Execution Time **(b)** Speedup

Figure 4.22: Comparative results of heat equation for overlapping computation/communication with helper threading for a 3D space 512x512x512

Chapter 5

Towards an efficient parallel implementation of 3D heat equation on modern clusters: blending optimization techniques

Up to this point, we have performed an extensive analysis on the parallelization of 3D heat equation, factors limiting parallel performance and proposed optimization techniques. In this chapter, we discuss the potential of combining the most successful optimization techniques, with respect to the characteristics of modern clusters, aiming to overcome performance limitations by exploiting capabilities of modern clusters to the highest possible degree. As a conclusion, we present the experimental results of our attempt, along with an overall evaluation of all parallel versions of 3D heat equation developed for the present work.

5.1 Motivation

In the year 2012, supercomputers appearing on top of the Top500 [47] list are build up of hundreds of thousands of processors with some Petaflop/s performance. Scientific parallel applications, intended for running on such powerful systems, should be designed and engineered to scale on these machines. Motivated by this demand, we examine the potential of engineering a parallel version of the 3D heat equation, which is a paradigm of stencil computations, capable of scaling on a supercomputer of petaflop computational power.

Hitherto, the obstacle for achieving performance of the parallel 3D heat equation has been communication overhead, caused by the communication pattern of the application and the limited bandwidth of the interconnection network. We have presented MPI message compression and computation/communication

overlapping, as performance optimizations, in an effort to overcome communication overhead. Experiments have proven the efficiency of these techniques on the parallel implementation. Looking further ahead, we are summoned to pick the optimization techniques that would best perform on a supercomputer, towards implementing an efficient parallel program.

Compression of MPI messages is certainly a successful technique, beating the problem of communication overhead at its root: by reducing message size, contention of network bandwidth is prevented and communication time decreases, resulting to an improved speedup of the parallel application. There is a better potential of applying compression to MPI messages on modern clusters, as their computational power would reduce compression time, which adds up to total execution time, thus producing an effective parallel application.

Overlapping of computation and communication does not minimize communication time per se, but successfully mitigates its effect on total execution time and speedup. Both presented schemes for overlapping have demonstrated satisfactory results, though the efficiency of each scheme depends on conditions: the overlapping scheme utilizing non-blocking communication functions is expected to execute more efficiently on clusters with a high performance interconnection network, while the hybrid MPI/OpenMP overlapping scheme is expected to execute more efficiently when more cores are available to undertake execution of computation and communication individually. Under these constraints and with respect to the architecture of modern clusters, we opt for the hybrid MPI/OpenMP scheme, as number of cores is over-sufficient for the implementation and no restrictions are posed on the performance of the interconnection network.

The idea we present in this chapter is the implementation of a hybrid MPI/OpenMP version of the 3D heat equation, which incorporates overlapping and compression of messages with the FPC compression algorithm. Apart from studying each technique distinctively and ascertaining their optimizing character, we presume that compressing messages benefits the efficiency of overlapping, since the proportion of communication time to computation time falls with compression, enabling a more efficient coverage through overlapping. Details on the implementation of the combinatorial optimizing scheme follow in the next section.

5.2 Implementing parallel 3D heat equation with message compression and computation/communication overlapping with helper threading

To implement parallel 3D heat equation with the suggested blending of optimization techniques, we have integrated message compression with FPC into the parallel implementation which overlaps computation and communication with helper threading, with minor changes. Compression and decompression of mes-

Pseudocode 5.1: Overlapping implementation of the 3D heat equation with hybrid MPI/OpenMP and FPC comrpession

```
T=512
/*Initializations*/
/*Set affinity on processors*/
/*Creation of cartesian virtual topology-3D grid of Px x Py x Pz processes*/
/*Scattering the initial matrix held by the root process to all processes */
/*Defining ranks of neighbouring processes*/
/*End of initializations*/
/*Computational kernel-executed by each process*/
#pragma omp parallel private(t,i,j,k) firstprivate(myCurrent,myPrevious) num_threads(2)
{
  for (t=0;t<T;t++)
  {

    if (my_thread_ID==0) /*Computation thread*/
    {
      //Computation of interior points
      for (i=2;i<=myX-1;i++)
        for (j=2;j<=myY-1;j++)
          for (k=2;k<=myZ-1;k++)
            myCurrent[x][y][z]=(myPrevious[x][y][z]+myPrevious[x-1][y][z]
                                +myPrevious[x+1][y][z]+myPrevious[x][y-1][z]
                                +myPrevious[x][y+1][z]+myPrevious[x][y][z-1]
                                +myPrevious[x][y][z+1])/7;
    }
    else if (my_thread_ID==1) /*Communication thread*/
    {
      Pack_data(2D boundaries of myPrevious to buffers of doubles)
      FPC_Compress(buffers to compressed buffers)
      Isend(compressed buffers to all neighbouring processes)
      IRecv(compressed buffers from all neighbouring processes)
      Waitall(communications)
      FPC_Decompress(compressed buffers to buffers of doubles)
      Unpack_data(buffers of doubles to halo layers of myPrevious)
    }

    if (my_thread_ID==1)
    {
      //Computation of boundaries
      compute_boundaries_of_subspace()
    }

    #pragma omp barrier

    swap(myCurrent,myPrevious);
  }
}
```

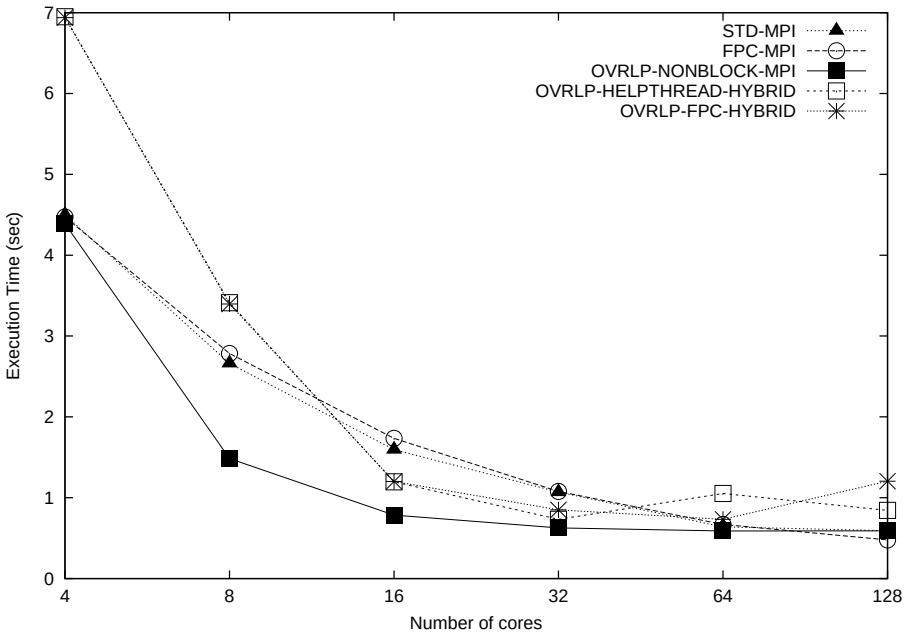
sages is assigned to the communication thread and compressed messages are sent and received instead of the original packed buffers. The implementation is described in Pseudocode 5.1.

The system call *sched_affinity* is employed in this implementation as well, to assure pinning of MPI processes and OpenMP threads to the desired processors.

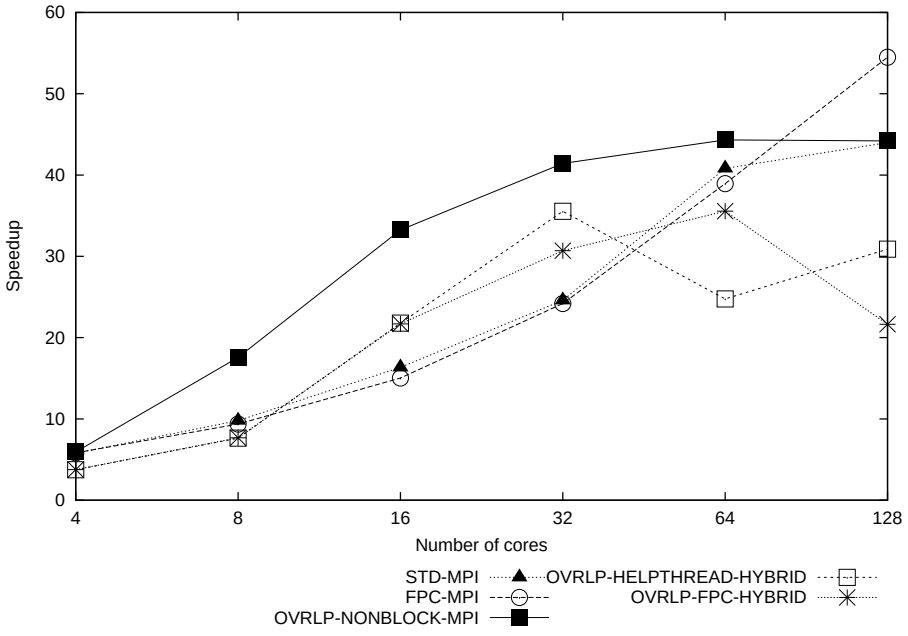
5.3 Overall experimental results

In this section, we present the experimental results of our proposed implementation which combines overlapping and compression, in comparison to the standard MPI version, the MPI version with FPC compression and the two parallel implementations with overlapping. As usual, we have experiment on three different grids of sizes $128 \times 128 \times 128$, $256 \times 256 \times 256$ and $512 \times 512 \times 512$, using Gigabit Ethernet and progressively increasing the numbers of cores from 2 up to 128.

Figures 5.1, 5.2 and 5.3 depict the overall experimental results for the differ-

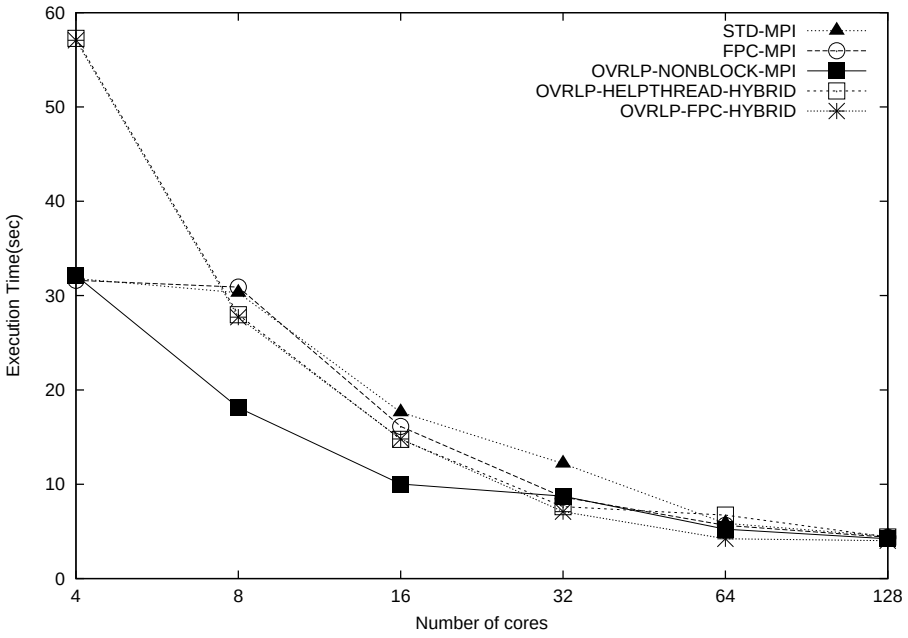


(a) Execution Time

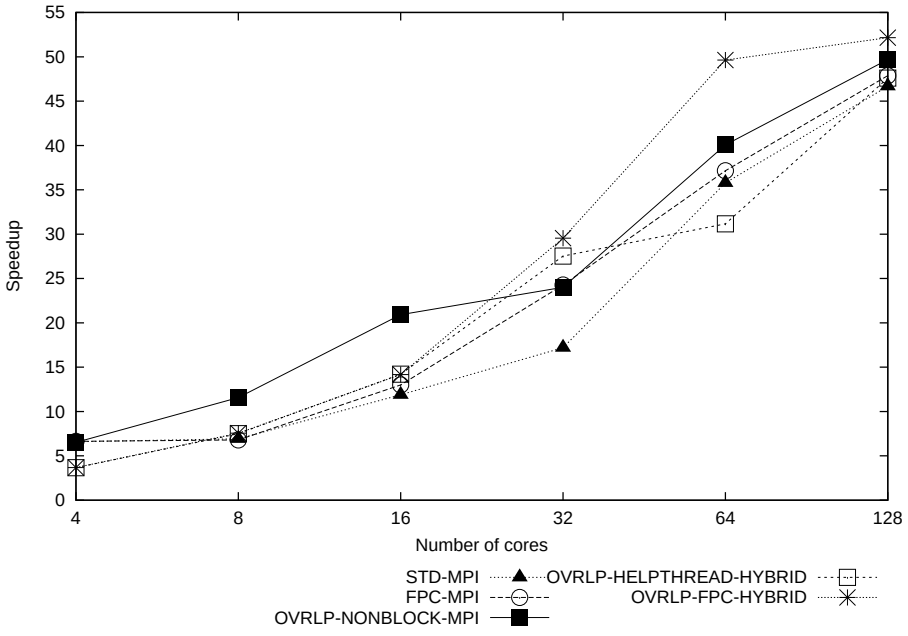


(b) Speedup

Figure 5.1: Overall results of the parallel versions of the 3D heat equation for a 3D space 128x128x128

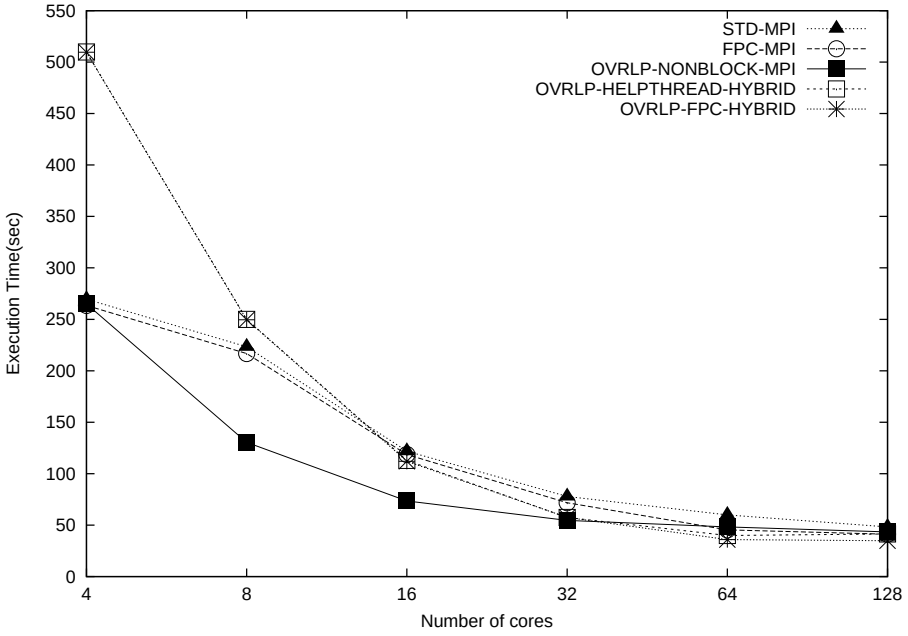


(a) Execution Time

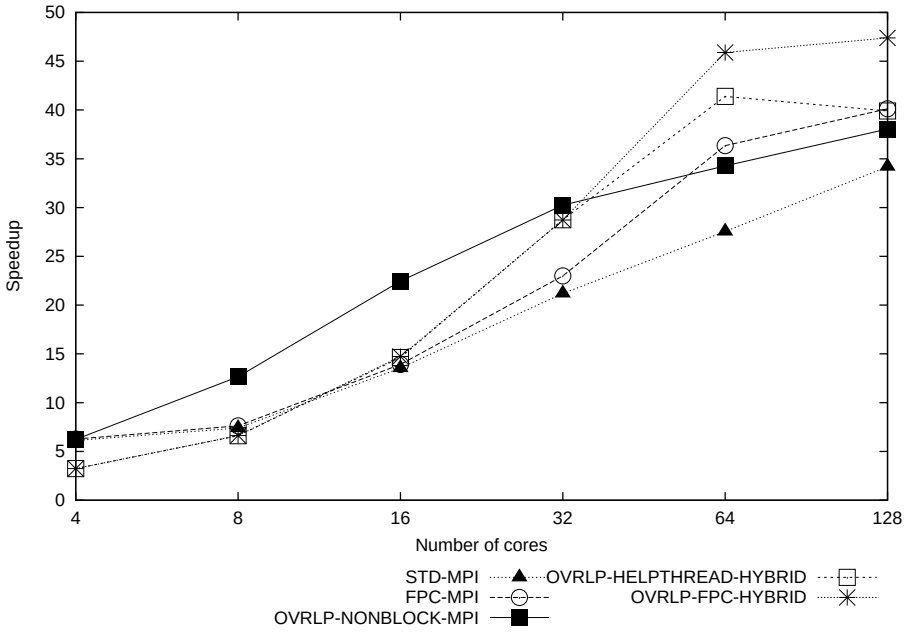


(b) Speedup

Figure 5.2: Overall results of the parallel versions of the 3D heat equation for a 3D space 256x256x256



(a) Execution Time



(b) Speedup

Figure 5.3: Overall results of the parallel versions of the 3D heat equation for a 3D space 512x512x512

ent 3D spaces. For the smaller 3D space of 128^3 data points, the best performing version is FPC-MPI version, namely the pure MPI version with compression. This result is expected for such a small 3D space, where computation time is very small compared to communication time and overlapping schemes fail to efficiently conceal a portion of communication time, while compression succeeds to reduce communication time and induce a better speedup compared to the pure MPI version. For the 256^3 3D space, the best speedup is achieved from our parallel version combining compression and overlapping, verifying the predictions on the success of the method. Overlapping version with non-blocking communication functions and helper threading follow in the ranking of best performing optimization schemes. The largest grid of 512^3 also verifies the superiority of the technique of combining compression and overlapping, compared to the standard MPI version or the implementation of single optimizing techniques. Following up, the version of MPI with FPC compression and the version of hybrid MPI/OpenMP with overlapping demonstrate an improvement in execution time and speedup against the standard MPI implementation. Since the 512^3 grid is a representative size problem for scientific applications, we rely on the respective experimental results to rank the performance of the various versions developed. We are now in the position to recognize that both message compression and overlapping communication and computation with helper threading deal efficiently with the problem of communication overhead appearing in the standard MPI implementation of the 3D heat equation and their combination paves way to parallel implementations that scale up as the number of utilized cores increases. Figure 5.4 and Table 5.1 summarize speedup for the various versions on 128 cores for all problem sizes. Improvement for the 512^3 problem size on 128 cores reaches 28% over the standard MPI implementation.

Grid Size	Speedup					Best performing version
	MPI-STD	MPI-FPC	OVRLP-NONBLOCK-MPI	OVRLP-HELPTHREAD-HYBRID	OVRLP-FPC-HYBRID	
128^3	44	54.5	44.2	30.9	21.6	MPI-FPC
256^3	46.7	47.9	49.7	47.6	52.2	OVRLP-FPC-HYBRID
512^3	34.2	40.1	38	39.9	47.4	OVRLP-FPC-HYBRID

Table 5.1: Speedup results for all parallel version of 3D heat equation on 128 cores

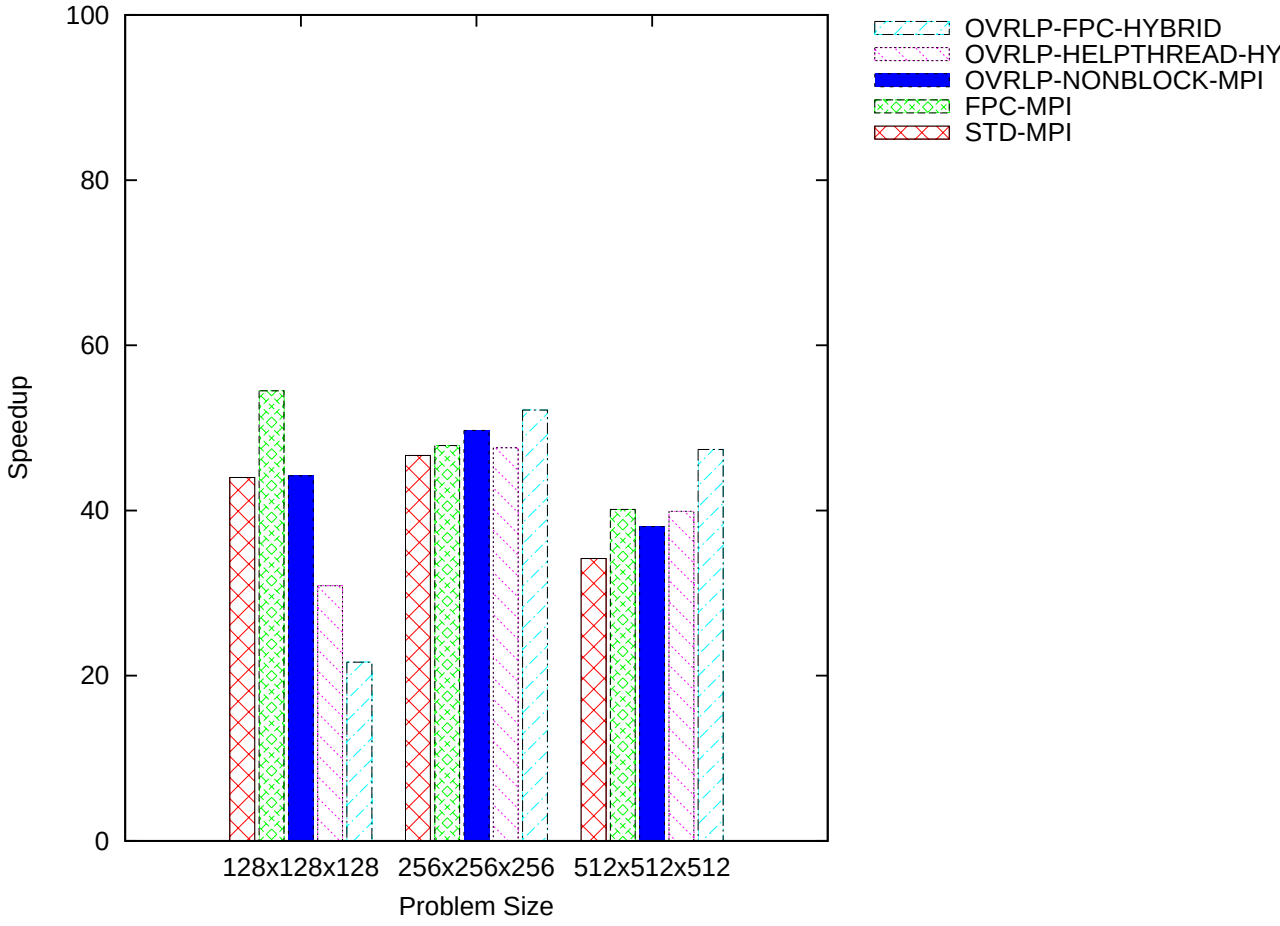


Figure 5.4: Speedup results for all parallel versions of 3D heat equation on 128 cores

Chapter 6

Conclusions and Future Work

In this diploma thesis, we have examined the behaviour of a set of optimizations techniques on the parallel implementation of the 3D heat equation. The 3D heat equation, as a member of the family of PDE solvers with stencil computations, has a "difficult" memory access pattern, which degrades parallel performance on shared memory models, and its parallel implementation with MPI has a "halo" communication pattern: each process exchanges messages with all its neighbours. Under these conditions, it was necessary to explore optimization techniques that fit the computation and communication pattern of the problem and provide efficient parallel performance.

Firstly, we evaluated the performance of the baseline OpenMP, MPI and hybrid MPI/OpenMP parallel implementation of the problem. Shared memory parallelization revealed that the application suffers from multiple memory accesses, which saturate the bandwidth of the memory bus. The same scheme holds for the traditional hybrid MPI/OpenMP model. Message passing parallelization, on the other hand, suffers from communication overhead, as the halo communication pattern invokes bulk message transferring via the interconnection network, which is easily contented with the increase of utilized processors. Consequently, as the number of utilized processors scales up, communication time is the largest portion of the total execution time.

To overcome the latency arising from multiple memory accesses, we tested the potential of applying 2D loop tiling on our OpenMP and hybrid MPI/OpenMP parallel implementations. However, the proposed scheme did not demonstrate any positive results on its application to the serial algorithm, thus we dropped it.

To minimize communication time of the MPI application, our primal target was to minimize the message size by means of compression. We experimented with several compression algorithms, in order to estimate their performance on double precision data, in terms of compression and decompression time and compression ratio. An algorithm oriented in compressing double precision data, the FPC algorithm, was chosen and integrated into our MPI parallel implementation, achieving a great improvement in communication time. Total execution time was also reduced, but compression and decompression time add up some

extra overhead to the application, restraining the overall improvement. Nevertheless, speedup increased in all cases we tested.

Overlapping of computation and communication is a well-known optimizing scheme which aims at exploiting the idle time of processors occurring due to lack of computation data, while communication operations take place. The MPI parallel 3D heat equation requires data exchanges for the computation of the boundaries on each 3D subdomain held by a process. Computation of the interior points is free to execute in parallel with communication operations, if resources are available. The first overlapping scheme that we implemented employs non-blocking MPI communication functions to allows computation of interior points to execute while the processor waits for communication operations to complete. This implementation exhibited some encouraging results, though it is expected to perform even better on a high performance interconnection network, as Myrinet. The second overlapping scheme recruits two OpenMP threads per MPI process to assign them the tasks of computation and communication, capable of executing in parallel. A considerable improvement has been observed in the execution results of the latter technique, although it is more resource consuming compared to the previously discussed implementations and the high communication to computation time ratio impedes a fine overlapping.

Considering the features of modern clusters, built up by thousands of cores of high computational power, we estimated that, regarding all optimization techniques tested, compression of messages and overlapping with helper threading are the most promising and viable for the improvement of efficiency of parallel programs. Since we managed to reduce the ratio of communication time to computation time with the utilization of compression of MPI messages, we presumed that, if we apply the overlapping scheme alongside with compression, we shall see a highly efficient parallel implementation of the 3D heat equation, able to scale up to hundreds of cores, overcoming communication overhead. We implemented the proposed scheme and the experimental results verified our assumption and demonstrated a total speedup of 47.4 on a 512^3 grid and an improvement of 28% compared to the baseline MPI version.

As a continuation of the present work, we believe that experiments on larger parallel systems should be conducted to verify and qualify the success of the combination of compression of MPI messages and overlapping with helper threading as an optimization technique. Moreover, the technique should be evaluated on a suite of variant scientific parallel applications, in order to establish our thesis that it acts beneficially on communication overhead. Finally, it is necessary to experiment with compression algorithms not evaluated in the present work, in the context of mining a compression algorithm of low execution time and good compression ratio, expedient for integration to MPI applications.

Bibliography

[1] 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, *The Landscape of Parallel Computing Research: A View from Berkeley*. Technical Report No. UCB/EECS-2006-183, 2006

[2] Thomas Rauber, Gudula Rünger, *Parallel Programming For Multicore and Cluster Systems*. Springer-Verlag Berlin Heidelberg, 2010 pp.162-164

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

[4] J.L. Gustafson, *Reevaluating Amdahl's Law*. Comm. ACM, Vol.31, No.5, 1988, pp. 532-533

[5] Grama, A., Gupta, A., Kumar, V., *Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures*, IEEE Parallel and Distributed Technology, 1993

[6] Michael J. Flynn, *Some Computer Organizations and Their Effectiveness*, IEEE Transactions On Computers, Vol. C-21, No.9, 1972

[7] George Em Karniadakis, Robert M. Kirby II, *Parallel Scientific Computing in C++ and MPI*, Cambridge University Press, 2003

[8] Randy Allen, Ken Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*, Morgan Kaufmann Publishers, 2002

[9] L. Lamport, *The parallel execution of DO loops*, Communications of the ACM, 17(2):83-94, 1974

[10] M.J.Wolfe, *Optimizing Supercompilers for Supercomputers*, Ph.D. Thesis, 1982

[11] John L. Hennessy, David A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd edition, Morgan Kaufmann, 2002, pp. 574

[12] Mark Norris, *Gigabit Ethernet technology and applications*, Artech House, 2003

[13] Supratik Majumder, Scott 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

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

[15] *OpenMP*, The OpenMP ARB, http://www.openmp.org

[16] http://en.wikipedia.org/wiki/OpenMP

[17] Alejandro Duran, Raúl Silvera, Julita Corbalán, Jesús Labarta, *Runtime Adjustment of Parallel Nested Loops*, In Proc. of the 19th ACM International Conference on Supercomputing , 2005

[18] H. Jin, M.Frumkin, J. Yan, *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance*, NAS Technical Report NAS-99-011, 1999

[19] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard-Version 2.2*, 2009

[20] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra, *MPI: The Complete Reference*, The MIT Press, 1996

[21] S.W. Bova, C. Breshears, C. Cuicchi, Z. Demirbilek, H.A. Gabb, *Dual-level Parallel Analysis of Harbor Wave Response Using MPI and OpenMP*, The International Journal of High Performance Computing Applications, 2000

[22] L.A. Smith, P. Kent, *Development and Performance of a Mixed OpenMP/MPI Quantum Monte Carlo Code*, Proceedings of the First European-Workshop on OpenMP, Lund, Sweden, 1999, pp. 6-9.

[23] Hybrid MPI/OpenMP programming for the SDSC teraflop system, Online Volume 14(3), 1999

[24] S. W. Williams, A. Waterman, and D. A. Patterson, emphRoofline: An insightful visual performance model for floating-point programs and multicore architectures, Technical Report UCB/EECS-2008-134, EECS Department, University of California, Berkeley, 2008

[25] Ananth Grama, Vipin Kumar, Sanjay Ranka, Vineet Singh, *On Architecture Independent Design and Analysis of Parallel Programs*, ICCS '01 Proceedings of the International Conference on Computational Science-Part II, 2001

[26] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick, *Stencil Computation Optimization and Auto-tuning on State-of-the-Art*

Multicore Architectures, SC '08 Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 2008

[27] Alaa Ismail El-Nashar, *To parallelize or not to parallelize, speed up issue*, International Journal of Distributed and Parallel Systems (IJDPS) Vol.2, No.2, 2011

[28] Gabriel Rivera, Chau-Wen Tseng, *Tiling Optimizations for 3D Scientific Computations*, Proceedings of the 2000 ACM/IEEE conference on Supercomputing, 2000

[29] Haohuan Fu, Robert G. Clapp, Olav Lindtjorn, Tengpeng Wei, Guangwen Yang, *Revisiting finite difference and spectral migration methods on diverse parallel architectures*, Computers & Geosciences, Volume 43, June 2012, pp. 187-196

[30] M. Burtscher, P. Ratanaworabhan, *High Throughput Compression of Double-Precision Floating-Point Data*, Data Compression Conference, 2007, pp. 293-302

[31] Rosa Filgueira, David E. Singh, Alejandro Calderón and Jesús Carretero, *CoMPI: Enhancing MPI Based Applications Performance and Scalability Using Run-Time Compression*, Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer Berlin / Heidelberg, 2009

[32] R. Zigon, *Run length encoding*, Dr. Jobb's Journal 14(2), 1989

[33] P. Daley, *Run length encoding revisited*, Dr. Jobb's Journal, 14(5), 1989

[34] Markus Geelnard, *Basic Compression Library, API Version 1.2*, http://bcl.comli.eu , 2006

[35] C.E. Shannon, *A Mathematical Theory of Communication*, Bell System Technical Journal 27:279-423, 1948

[36] R.M. Fano, *The Transmission of Information*, Technical Report No. 65, Research Laboratory of Electronics at MIT, 1949

[37] D.A. Huffman, *A Method for the Construction of Minimum-Redundancy Codes*, Proceedings of the I.R.E.,1952, pp 1098–1102

[38] M. Burtscher, P. Ratanaworabhan, *FPC: A High-Speed Compressor for Double-Precision Floating-Point Data*, IEEE Transactions on Computers, Vol. 58, No. 1, 2009, pp. 18-31

[39] Y. Sazeides and J. E. Smith, *The Predictability of Data Values*, 30th International Symposium on Microarchitecture, 1997, pp.248-258

[40] B. Goeman, H. Vandierendonck and K. Bosschere, *Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency*, Seventh International Symposium on High Performance Computer Architecture, 2001, pp. 207-216. January 2001

[41] M. Burtscher, P. Ratanaworabhan, *The FPC Double-Precision Floating-Point Compression Algorithm and its Implementation*, http://www.csl.cornell.edu/ burtscher/research/FPC, 2009

[42] Oberhumer, M.F.X.J., *Lzo real time data compression library*, http://www.oberhumer.com/opensource/lzo, 2005

[43] Jacob Ziv, Abraham Lempel, *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory, 23(3), 1977, pp. 337-343

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

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

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

[47] http://top500.org