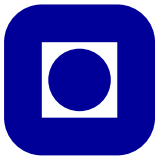
NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK OG ELEKTROTEKNIKK



HOVEDOPPGAVE

Kandidatens navn: Robin Holtet

Fag: Datateknikk

Oppgavens tittel (norsk): Kommunikasjons-reduserende stensil-baserte algoritmer og metoder

Oppgavens tittel (engelsk): Communication-reducing Stencil-based Algorithms and Methods

Oppgavens tekst:

Clusters of PCs have a high processing-power/cost ratio compared to traditional supercomputers. But low-cost clusters also generally have network interconnects with higher latency and lower bandwidth than a supercomputer. This makes the cluster unsuitable for several communication-intensive applications. In some cases it is possible to reduce the amount of communication by doing extra calculation. The optimal amount of extra calculation to do depends on the application and the system.

A benchmark program for finding an optimal amount of extra calculations for a given system will be developed. The program will be tested on several clusters and supercomputers to evaluate the effectiveness of such methods.

Oppgaven gitt:	20. januar 2003
Besvarelsen leveres innen:	16. juni 2003
Besvarelsen levert:	7. juli 2003
Utført ved:	Institutt for datateknikk og informasjonsvitenskap
Veileder:	Anne Cathrine Elster

Trondheim, 7. juli 2003

Anne Cathrine Elster
Faglærer

Abstract

In this work, the potential performance gain from optimizing stencil-based communication for cluster-like environments is studied. Clusters of PCs have a high processing-power/cost ratio compared to traditional supercomputers which are making them increasingly interesting alternatives for HPC applications. However, low-cost clusters also generally have network interconnects with higher latency and lower bandwidth than typical supercomputers. This makes such cluster systems potentially unsuitable for several communication intensive applications.

The specific class of applications considered in this work are domain decomposition problems that exchange border information between each iteration. Examples of such computations arise, for instance, when using finite difference methods to solve partial differential equations (PDEs). These algorithm find approximate solutions by doing iterations that look at local/neighbor data (often called stencils) with a communication step between each iteration that exchange border information. By performing extra calculations, several iteration steps may be done between each communication step, thus saving communication.

The goal of this work is to analyze the tradeoffs between extra calculations and communication for cluster-based systems, and to develop a benchmark program based on the aforementioned stencil-based algorithms. Tests are performed on several different systems, both PC clusters and supercomputers to see how much efficiency can be gained from using this method. We show that cluster systems can benefit greatly from the saved communication. Supercomputers also show higher efficiency, but to a lesser degree. We also consider methods to automatically find the optimal tradeoff.

Acknowledgements

Several people have been of great help to me while working on this project. First I would like to thank my supervisor, Ph.D. Anne C. Elster, for proposing the assignment and giving me lots of help and encouragement. I would also like to thank my boss Jørn Amundsen and my other co-workers at ITEA for help and for being understanding when I at some times was very busy. I will also say thanks to my student colleague Åsmund Østvold for technical help and lots of good ideas. ITEA and NTNU also deserves thanks for letting me use the NOTUR computers and the CLUSTIS cluster for running the benchmark code. Thanks also to the people at SGI for letting me use their Altix 3000 test machine to run the code.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the demand for high-performance computing (HPC) power continues to exceed the capabilities of current single processing and memory units, how to utilize parallel computing efficiently continues to be an important research topic.

Parallel algorithms typically split their compute-intensive tasks into several subtasks to be executed in parallel as either processes or threads each running on a separate processing element/processor.

Computing-intensive can mean several things. Some tasks need so many processor compute cycles that they cannot be finished within a reasonable time on a single processor.

Another limit is memory. Since each processor has a limited number of registers and caches, there is a limit to how much random access memory they each can access efficiently. This is known as the processor-memory bottleneck. Very large memory banks associated with a single processor would be both very expensive and end up with very non-uniform access patterns. Another important reason for going to parallel processing systems, is therefore to have access to the very large amounts of random access memory available for such systems.

A single 64-bit processor can certainly address a lot of virtual memory, but if one cannot run the compute-intensive applications in-core (i.e. fit the problems in the limited random access memory available for single processors), these applications may end up having unbearably long execution times due to the much much slower disk swaps then needed.

In some cases, I/O units can be the main reason to solve a given task in parallel. Fast access to large amounts of data often makes it necessary to use several disks with dedicated processors. Scientific visualization often depends on several graphics pipelines working in parallel, because each pipeline is too slow to get acceptable framerate and resolution.

In the following subchapters, we will briefly introduce the main classes of HPC systems followed by the most common parallel models. A terminology list is provided in section 1.3, whereas section 1.4 outlines the rest of this thesis.

1.1 Types of HPC systems

In the literature, the terms high-performance computers and supercomputers are often used interchangeably. There exists no strict definition of a high performance computer. One could say that currently any computer system delivering over 1 Gigaflop/s; alternatively, any computer that has an order of magnitude, or greater, performance than current top-end workstations, is a high-performance computer or supercomputer. Over time many different technical solution to the problem of making a good high performance computer have emerged. Most of these have been parallel machines with several processors.

Processor speeds have always been limited by current production methods. The space needed between the individual transistors on the semiconductor dice puts an upper limit on the clock frequency. The processor die size and the maximum number of transistors per square cm limit how complex the processor may be. These are all factors who determine the processing power of a processor, and when the limits are reached the natural method of increasing performance is to utilize multiple processors to solve the problem.

Different methods of connecting the processors have been used, and machines have been connected in different topologies like rings, hypercubes, buses and hierarchies.

A basic classification of computer systems was proposed by Flynn [8] in 1972, and this classification, "Flynn's taxonomy" is still used today. The first class is SIMD, Single Instruction Multiple Data. SIMD machines can be utilized most efficiently for data-parallel application, where the same instructions are performed on several elements. MIMD means Multiple Instruction, Multiple Data. These machines can perform different operations on different parts of the data at the same time. All the parallel systems we are interested in, can be classified as MIMD systems.

MIMD machines can be further classified by how tightly they are coupled.

Shared-memory machines: Each processor sees the collected memory of the whole machine as one memory-image. The processors are normaly connected by a shared bus or low-latency interconnects.

Distributed-memory machines: Each processor has its own memory. The processors are normally connected by low-latency interconnects.

Cluster systems: Each processor is a separate unit, and these units are connected by low-latency interconnects or regular Ethernet.

In this paper we will look at the two most different system-classes for parallel MIMD machines, the PC-cluster and the shared-memory supercomputer.

1.1.1 Clusters

The idea behind cluster systems is to use commodity PC hardware to build parallel high-performance machines. Because of mass-production and a market

with hard competition, the components of a PC gives more power at the same price as a traditional supercomputer. Using PC hardware also gives a great choice in which components to use. One problem with clusters is that regular Ethernet has very high latency compared to interconnects used in super-computers. The typical latency of gigabit Ethernet is in the 100-150 μs range, as shown in [13]. Several specialized low-latency network solutions exists with latencies in the 1-4 μs range, as presented in [12] and [14]. However, these are much more expensive than regular Ethernet solutions. Another problem with PCs is that the reliability is lower than that of a supercomputer. Parallel computations can last for days, and when a restart is needed, valuable time is often lost. Therefore, expected uptime is also an important factor when the economics of a cluster system is discussed. Some system vendors like IBM and HP offer specialized cluster-node machines based mostly on commodity hardware, but with added reliability and hot-swapping abilities. These systems are priced between the low-end, "home-made" clusters and the traditional supercomputers.

1.1.2 Supercomputers

Supercomputers are typically machines that are designed for high-performance calculations. They do not need to have multiple CPUs (i.e. be parallel systems), but most, if not all supercomputers today are. The top500 supercomputer list, found at [21] lists the 500 fastest computer systems in the world. The last single-processor systems disappeared from the list in 1998.

The common feature of supercomputer architectures is the use of low-latency (and expensive) interconnections between each processing unit. The supercomputers used in this work are shared-memory machines, where each processor sees the collected memory of the whole machine as one memory image.

1.2 Parallel models

The two main types of parallel algorithms are the data-parallel and the task-parallel algorithms.

Task-parallel algorithms are methods where the parallel tasks are independent so that no communication is needed between the tasks during the computation. Many task-based algorithms are called "trivially parallel" since no sychronization and communication between each task is required, hence making such algorithms fairly easy to implement. Examples of task-parallel applications are rendering of animations, where each processor is assigned separate frames, and running a computation model with different parameters.

Data Parallel algorithms usually have a large dataset, which needs to be divided onto several processors. Then mostly the same operations are done to each part. The intermediate results of one part are often dependent on the

results of some other parts. A delay on one processor, or the network, can thereby delay the whole calculation. Data-parallel algorithms thus generally have a higher demand for communication than task-parallel. The efficiency is often highly dependent on how fast the communication links between the processors are. Efficient use of communication is therefore the greatest challenge when creating an efficient parallel implementation.

The Message Passing Model:

There exists several performance and programming models of parallel computing. These can be classified from the degree of abstraction. A survey of parallel models are found in [23]. In this work we will use the message passing model, on which MPI (the Message Passing Interface) standard [17] is based. In this model, communication primitives such as send, receive, broadcast, barriers, etc are explicit. MPI is hence an interface with a very low abstraction. This gives a good control of what is actually done, and makes it possible to tune the application to maximum performance. The low abstraction also makes the programming more complex than with a more abstract model.

1.3 Important terminology

Speedup Perhaps the most important performance metric in parallel computing is speedup. The speedup shows how much faster the problem is solved when using several processors in parallel compared to using 1 processor. The speedup is given by the formula $S = t(1)/t(n)$, where S is the speedup, t(1) is the execution time of the fastest serial program and t(n) is the execution time of the parallel version. For the sake of simplicity, the parallel version run on one processor is often used instead of creating an optimized serial version.

Linear speedup The speedup is called linear when the problem is solved N times faster with N processors.

Superlinear speedup When the problem is solved more than N times faster with N processors, the parallelization is said to give superlinear speedup. This is only sometimes accomplished with trivially parallel applications, when the division of the problem produces local parts small enough to fit entirely in the cache. Superlinear speedup is mostly a sign that the serial version utilizes the cache badly.

Efficiency Efficiency is the speedup divided by the number of processors. This is the same value as the percentage of linear speedup achieved.

System description Systems are mainly described by processor, disk, memory and network. The number, type and speed of processors is important.

Memory and disk are mainly characterized by capacity, but latency and throughput can also be of importance for some applications. The type, speed and configuration of the interconnect is also important. When talking about clusters, the word node is often used. This usually means one unit consisting of processor, memory and network card. One node can also consist of two or more processors in SMP (Symmetric Multiprocessing) configuration.

Symmetric Multiprocessing (SMP) A configuration of a single computer with multiple processors. Every processor has equivalent, full access to machine resources-memory, peripherals, graphics and other controllers. Additionally, any unshared resources (like L2 cache) have mechanisms to inform all processors of any need to synchronize content.

1.4 Outline

The following chapter will discuss automatic parallel optimizations and related work. Chapter 3 describes how to possibly reduce the communication costs by performing extra calculations, especially on high latency systems, for the problem we chose to consider. A description of these problems – 1-D and 2-D domain decomposition problems that exchange border information between each iteration – in also included.

The implementation of our benchmarking code is discussed in Chapter 4, whereas Chapter 5 presents our results that include benchmarking several HPC systems for various matrix sizes. Conclusions and future work are given in Chapter 6.

The appendixes include usage instructions for the developed benchmark program and code examples from the most important parts.

Chapter 2

Automatic Parallel Optimizations and Related Work

2.1 Automatic Optimization

The project is based on other research in the fields of cluster systems and automatic tuning of algorithms. These are large fields, and are expected to grow as more researchers discover the economic potentials of cluster systems. A large base of applications have been written for supercomputers, and the potential savings from running those on cheaper clusters are large. The need for automatic tuning of the application is also greater on a cluster. There is no standard hardware solution for building a cluster. Several choices exist for mainboard, processor and interconnect solution. This makes most clusters "unique", making it difficult to optimize an application for cluster systems in general. A supercomputer, on the other hand, has a well-defined architecture and supercomputer vendors like SGI and CRAY have their own implementations of MPI, optimized mathematical libraries and optimizing compilers to get maximum performance from the hardware.

A whole new paradigm has emerged called adaptive software or "Automated Empirical Optimization of Software" (EAOS) [25] aimed at making new generations of performance-critical libraries more adaptive to different hardware. Several methods of automatic optimization are actively researched. Some methods aim at automatically generating optimal code, while others use fixed code with parameters who need optimalized values.

2.2 Fixed code

Since our system will have only one parameter to adjust and optimize, we feel that an approach with fixed code is the most appropriate. One system designed to optimize parameters in libraries and programs is Active Harmony, as described

in [24], [10] and [11]. Programs who wish to use this system must export the relevant parameters to a running Active Harmony server. During several runs of the same parts of the program, the server uses timing information to adjust the exported parameters. After some time, an optimal set of parameters is hopefully found.

The ATLAS project (Automatically Tuned Linear Algebra Software) [25], [7] aims to develop a methodology for the automatic generation of highly efficient basic linear algebra routines. The projects looks promising, and the techniques used here should be studied for adptation to stencil-based communication.

2.3 Stencils

Chris Ding and Yun (Helen) He have written a paper on ghost cell expansion and diagonal communication elimination methods in [5]. The paper presents several formulas related to stencil-based communication and mathematical theory of how much can be saved from using the right communication patterns. It also presents timing results from different machines. Ding and He look at both 2- and 3-dimensional matrices. Their 2-dimensional matrix uses so-called 9-point stencils. This means each each node has eight neighbours. Our model will be using the same stencil.

2.4 Performance models

Several parallel models exist to aid the development of parallel algorithms. LogP[4] is a machine model aimed at developing efficient parallel algorithms. The model tries to be general enough to be used on a broad range of problems and hardware. At the same time the model must not be so simplified that the results are not realistic. By finding a good balance between simplicity and detail, the model can be used to describe and analyze problems reliably without beeing too difficult to use. The parameters used by the logP model are based on abstractions of communication bandwidth, computation bandwidth, the communication delay and the efficiency of coupling communication and computation.

The paper "Performance modelling and evaluation of MPI" [1] uses the LogGP model, an extension of the LogP model, as a conceptual framework to model the performance of MPI communications. Simple benchmarks are run on both supercomputers and clusters to verify the performance model. They conclude that the LogGP model can be used with good results by incorporating more details about the platform and protocols used. Although we will focus mainly on empirical studies, we regard crating a mathematical model as a possible future extension to our system.

2.5 Optimization for Clusters

Work is also being done to create a parallel version of LAPACK for clusters [2]. LAPACK is a much-used library for solving linear algebra problems for dense matrices. The result from this work is the Lapack For Clusters (LFC) library. LFC will enable users to use LAPACK as a serial, single-processor interface. By using empirical data from previous runs and combining these with pre-defined rules, a decision is made by the system to solve the problem using one or several processors.

The paper "Efficient resource utilization for parallel I/O in cluster environments" [3] discuss the bottlenecks who occur when several processors use I/O simultaneously. It also looks at what happens when two processors in an SMP machine access the same I/O system at the same time. The paper points at several problems related to I/O in a cluster system being slower than that of a supercomputer.

Chapter 3

Reducing Communication Through Extra Calculations

In this assignment we will be looking at parallel algorithms and the possibility to reduce communication by doing extra computation on each node. The particular algoritm and problem class we will look at is domain decomposition problems that exchange border information between each iteration. One important usage of such methods is in solving partial differential algorithms, PDEs. The solution to the PDE is found by doing several iterations over a large discrete matrix. This matrix is divided onto several processors, who iterate over each local part of the matrix. Because calculations of the values along the border requires the values "outside" the border, i.e. on the neighbor processors, these ghost values are exchanged and stored locally between each iteration. However, by letting the areas given to each processor overlap, several iterations can be done between the exchanges. Each processor then calculates a larger area than what is strictly their "own". This method reduces the number of communication operations by a factor of `n`, where `n` is the width of the extra overlap between the local parts. At the same time the total amount of calculation increases.

The benchmark program we develop will be based on a domain decomposition algorithm where this border width can be varied. At the same time we also want to change various other parameters like matrix size, calculation complexity and communication delay. The code will be developed so that we can run it on a variety of supercomputers and cluster systems, and compare the performance when solving domain decomposition problems.

3.1 Communication Patterns

We have chosen to look at domain decomposition of 2-dimensional matrices. The global matrix can be divided along one or two dimensions, as shown in the Figures 3.1 and 3.2.

We will also look at using two different communication patterns for the two-dimensional division. The first communication pattern uses one step to exchange all border information, both edges and corners. This is shown in Figure 3.3 and Figure 3.5. The second pattern uses two steps to exchange the borders. First, the edges are exchanged. Then the corner values are exchanged. This is shown in Figure 3.4 and Figure 3.6. The performance of these patterns will be compared in the Results chapter.

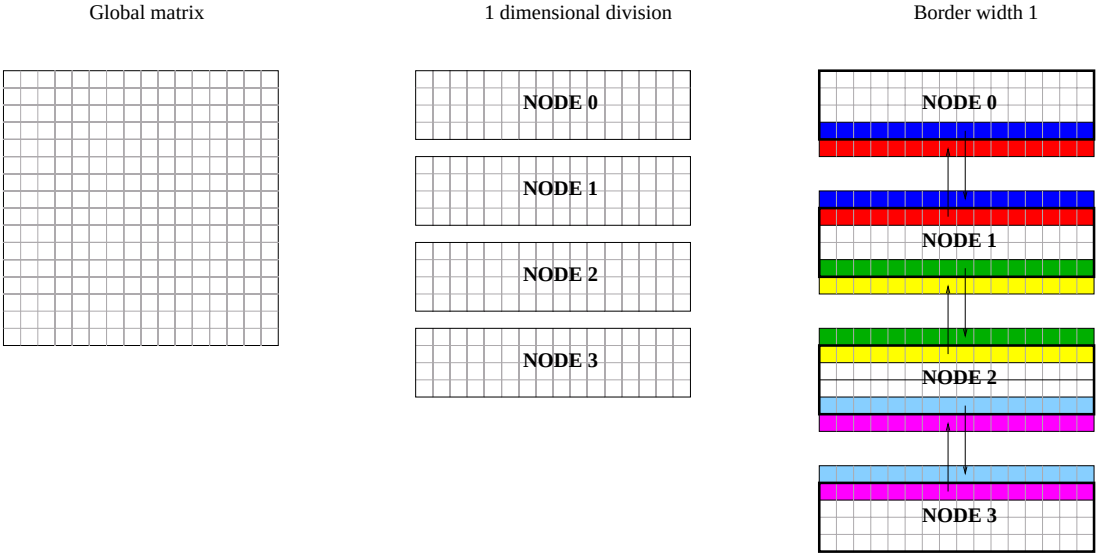


Figure 3.1: 1-dimensional division of matrix

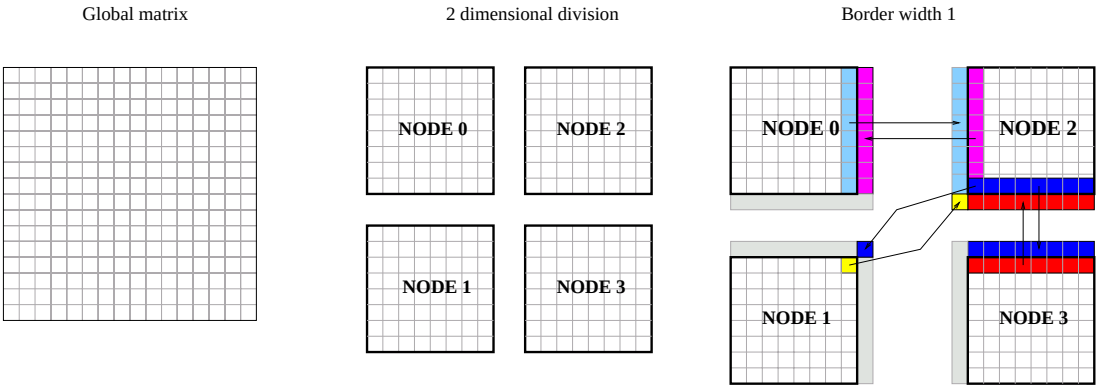


Figure 3.2: 2-dimensional division of matrix

Traditional benchmarks like for example Pallas [18] measure basic system parameters such as processing power, network bandwidth and latency. Other benchmarks like for example Linpack [6] measure the efficiency of a system when running a standard scientific application. By creating a flexible and extensible

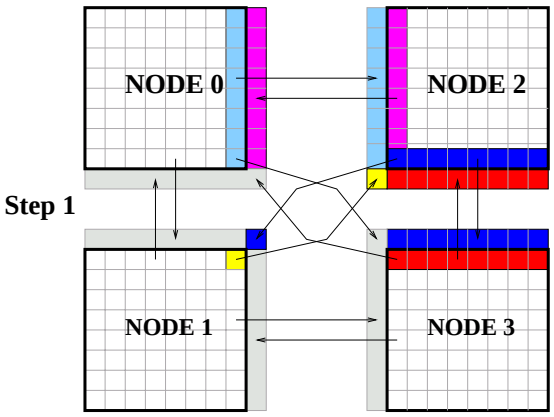


Figure 3.3: Communication pattern 1

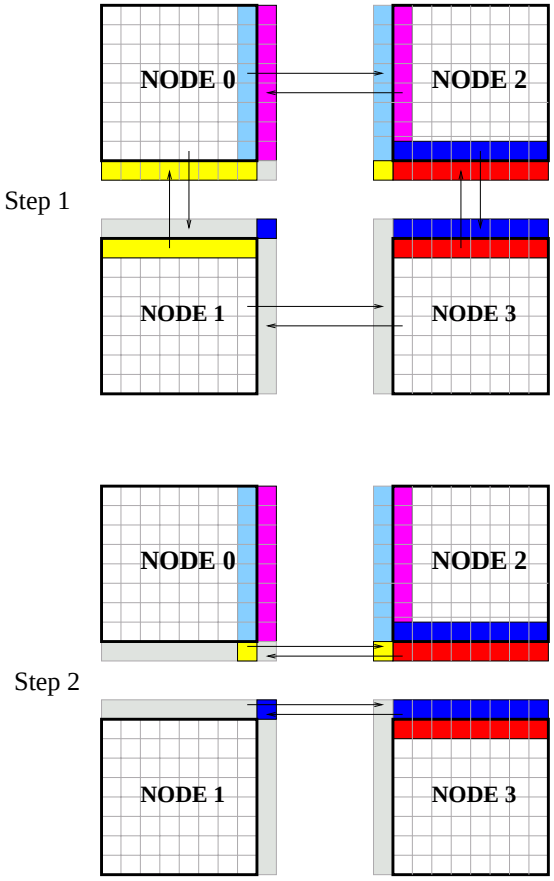


Figure 3.4: Communication pattern 2

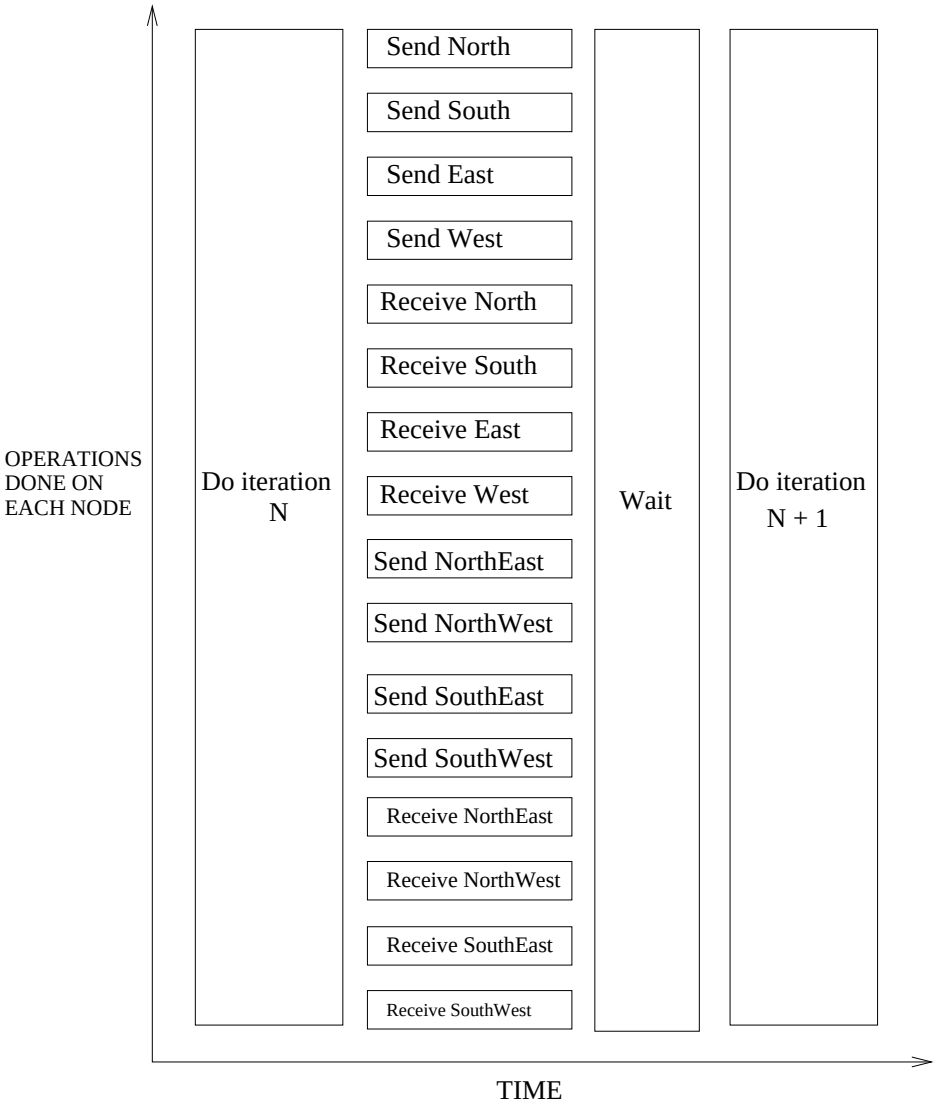


Figure 3.5: Timeline for communication pattern 1

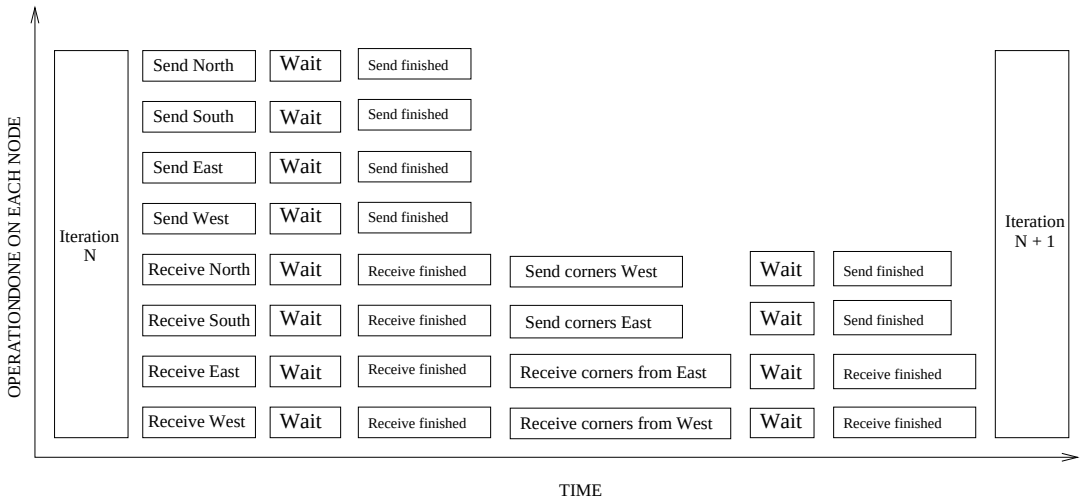


Figure 3.6: Timeline for communication pattern 2

system, we will make a benchmark system based on the principles of doing extra calculations to save communication in all kinds of domain decomposition systems. This benchmark can then be used to show what the potential gain is when switching from a traditional approach to the communication-saving approach of doing domain decomposition. The benchmark will give a maximum possible efficiency when using communication-reducing methods. This efficiency should be equal or better than the efficiency gained using traditional stencil-based communication without any communication reduction.

Having a benchmark like this can hopefully make it easier to evaluate and performance-tune a system. For example, if a system is considered mainly for solving PDEs, it should be easier to run the benchmark than to install and configure a complete solver system. This could get a quick hint as to how suitable the system is.

3.2 Reduce communication

Because there is a large difference in communication latency between a supercomputer and a low-cost cluster system, it is not given that this method of optimization works for both types of systems. A cluster has about the same computational power as a supercomputer (within a factor n), but a much higher latency in the communication link (order of magnitude). Because of this, the chance of getting a high efficiency without this optimization should be higher on the supercomputer. On the cluster, on the other hand, the communication will delay the whole calculation, leding to lower efficiency. Because of this, we suspect that the cluster has more potential to be tuned to better performance. This will be investigated as the code is run on several clusters and supercomputers.

3.3 Optimize for SMP Machines

Another potential for optimization is in clusters based on dual-cpu machines. In such clusters, the communication between the two processors on the same motherboard will have a much lower latency than between processors of different nodes. We suspect that the optimal amount of extra work to do is not the same for these two classes of communication links. By measuring how long time each send/receive uses, it should be possible to have different overlap width on every edge, and thus optimize further.

3.4 Benchmarking

The code finds the optimal amount of extra calculations to do.This value can be used as a performance metric to describe the system, as it shows how well-suited the system is for doing these kinds of calculations. If optimal performance is found with a high degree of overlapping calculation, the system cannot reach as good efficiency as a system where little overlap is optimal. This can show that the network is too slow or not properly configured to optimally solve these kinds of problems. The benchmark also shows how much is lost due to slow network, and can be used when evaluating different systems. As low-latency hardware solutions are more expensive than regular networking, this code can hopefully help in choosing the right type of network for applications based on domain decomposition.

Chapter 4

The benchmarking code

4.1 Introduction

The code is written in C++, using MPI 1 [17], [22] for communication. This chapter describes the code structure and clarifies the most important design decisions.

4.2 Design

4.2.1 Structure

The program is divided into classes. Subclasses are named after a simple numbering scheme. The class hierarchy is shown in figure 4.1. When adding new functionality and methods to the program, new leaf classes can easily be added. For example, to add a new communication pattern for 1-dimensional division, a new `DataTypes1Dn` class must be created and assigned a number in the `Values` class. Matrix1D must then be updated to recognize the new `DataType`. The code uses polymorphism to minimize the number of affected classes when adding new subclasses.

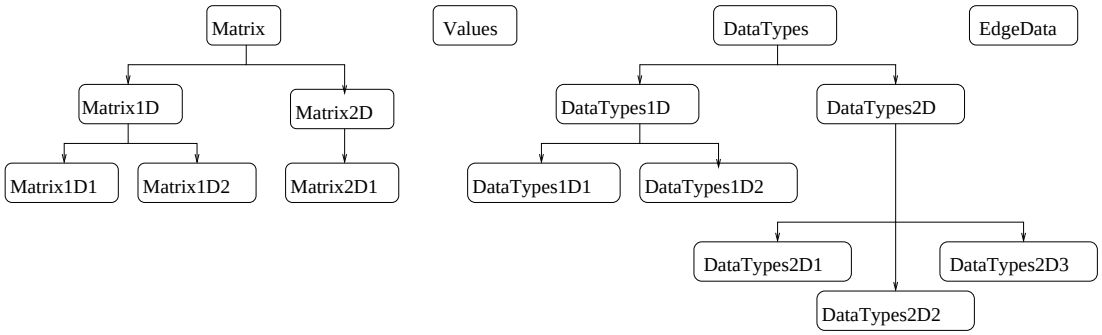


Figure 4.1: The classes used

4.2.2 Classes

The responsibility of the different classes are as follows:

Matrix Contains the local matrix on each node. Handles initialization of the matrix and the mask array, which decides for each point if it is to be updated. The Matrix class also performs calculations and delay loops, and can thus be considered as the main class of the benchmark system. Currently, Matrix subclasses for 1- and 2-dimensional division of the matrix are implemented. Note that both these matrices are two-dimensional, it is just the division which is varied like shown in the figures 3.1 and 3.2. When the program is run, one Matrix1D or Matrix2D instance is created. This instance is one of the subclasses of MatrixND. This is automatically chosed based on the given DataType parameter. The Matrix instance then creates the correct DataTypes instance. Matrix1D uses the DataTypes1D1, DataTypes1D2, etc. classes. Similar for Matrix2D.

DataTypes The `DataTypes` class handles the MPI datatypes used for exchanging borders. The class also contains the code for doing the exchange. For each of the Matrix subclasses, different Datatype classes can be used. The datatype classes implement different communication patterns and timing. When experimenting with different methods of doing automatic tuning, new `DataTypes` subclasses can be created, using the existing 1- or 2-dimensional `Matrix` classes.

EdgeData Contain information about one edge. This includes rank of neighbour, edge overlap width and timing data.

Values Normally, only one instance of Values exists for each run. This class contains all the parameters to the program. The class also deals with the parsing of the command line.

FileIO This class contains the function to write the result matrix to an `pgm` image file. This is mainly to "see" that the code has calculated something, and easily verify that the communication works correctly. The image file can be viewed with the free `xv` image viewer.

The `Matrix::diffuse()` function does a "smoothing" of the matrix. The matrix is initialized like in figure 4.2 , and the resulting matrix will look similar to figure 4.3. This function is a great help when developing new communication patterns, as most mistakes should result in a different-looking picture.

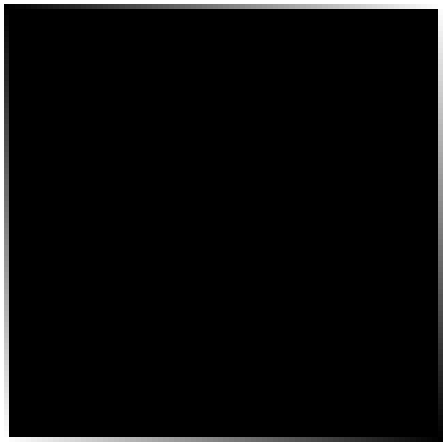


Figure 4.2: Initialized matrix

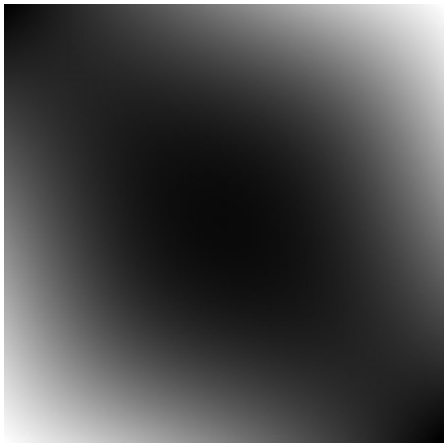


Figure 4.3: Result of calculations, 6000 iterations

Listing 4.1: Border exchange

```
MPI_Request req[16];
MPI_Status  stat[16];
int sendCount = 0;
exchangeData( 0, -1, sendN, recvN, &req[0], &req[1] );
exchangeData( 0, 1, sendS, recvS, &req[2], &req[3] );
exchangeData( -1, 0, sendW, recvW, &req[4], &req[5] );
exchangeData( 1, 0, sendE, recvE, &req[6], &req[7] );

exchangeData( -1, -1, sendNW, recvNW, &req[8], &req[9] );
exchangeData( 1, -1, sendNE, recvNE, &req[10], &req[11] );
exchangeData( -1, 1, sendSW, recvSW, &req[12], &req[13] );
exchangeData( 1, 1, sendSE, recvSE, &req[14], &req[15] );
MPI_Waitall( 16, req, stat );
```

4.3 Dividing the matrix

When the dataset is divided in one dimension, the ghost points exchanged are two continious arrays of length `GlobalSize`. The points are sent and received as `MPI_FLOAT`. The one-dimensional division is shown in figure 3.1.

When the matrix is divided in two dimentions, MPI datatypes for the different edges and corners are created. The division is done as shown in figure 3.2.

Two different communication patterns are used. The first pattern is as shown in figure 3.5. The corresponding code is listed in Listing 4.3.

The second pattern is as shown in figure 3.6. The relevant code is listed in Listing 4.3.

4.4 MPI Datatypes

When the dataset is divided in two dimensions, the ghost points are defined as MPI subarray datatypes. When the border is decided once for each run, the locations of the ghost points are static. When trying to optimize the border using timing values while running, the datatypes should be defined once at the beginning, and used with an offset. The redefifinition of datatypes will probably create too much overhead.

4.5 Benchmarking

Because our code is only supposed to be used for benchmarking, it does not do any real calculations, except to see whether the communication is correct. The code accesses every cell in the local part of the grid. This is to ensure that

Listing 4.2: Border exchange

```
MPI_Request req[8];
MPI_Status  stat[8];
int sendCount = 0;
exchangeData( N.rank, -1, N.send, N.recv, &req[0], &req[1] );
exchangeData( S.rank, 1, S.send, S.recv, &req[2], &req[3] );
exchangeData( W.rank, 0, W.send, W.recv, &req[4], &req[5] );
exchangeData( E.rank, 0, E.send, E.recv, &req[6], &req[7] );
MPI_Waitall( 8, req, stat );

exchangeData( N.rank, -1, sendNW, recvSE, &req[0], &req[1] );
exchangeData( N.rank, -1, sendNE, recvSW, &req[2], &req[3] );
exchangeData( S.rank, 1, sendSW, recvNE, &req[4], &req[5] );
exchangeData( S.rank, 1, sendSE, recvNW, &req[6], &req[7] );
MPI_Waitall( 8, req, stat );
```

the compiler does not optimize away any important parts. It also makes the test more realistic with respect to cache-usage. When the data size grows, after a certain point the whole local part cannot fit in cache, and calculation slows down. This effect gives an extra performance gain when adding more processors to large problems. Unfortunately, adding processors also increase the total amount of communication operations.

Since most of what is done between calculations is memory access, the application becomes more memory-bound than CPU bound. To add flexibility, an option is added to skip a part of these calculations and calculate only every n'th matrix value. The program parameter for calculation delay can then be used to add a CPU-bound active-wait delay.

4.6 C++ comments

It is often argued whether to use C or C++ when making time-critical applications. The first versions of the program were written in C, but to make the structure more clear, and to make it easier to add more types of communication stencils and algorithms, we divided the code into classes. We have found no noticeable performance penalty from moving to C++. This is shown in figure 4.4

According to what we believe, the execution times should start rather high, find a lower minimum value and rise again as the border width increased. Looking at the graph shows that this is not the case. Instead, the execution times have several peaks. This is something we will be looking more closely at in the Results chapter.

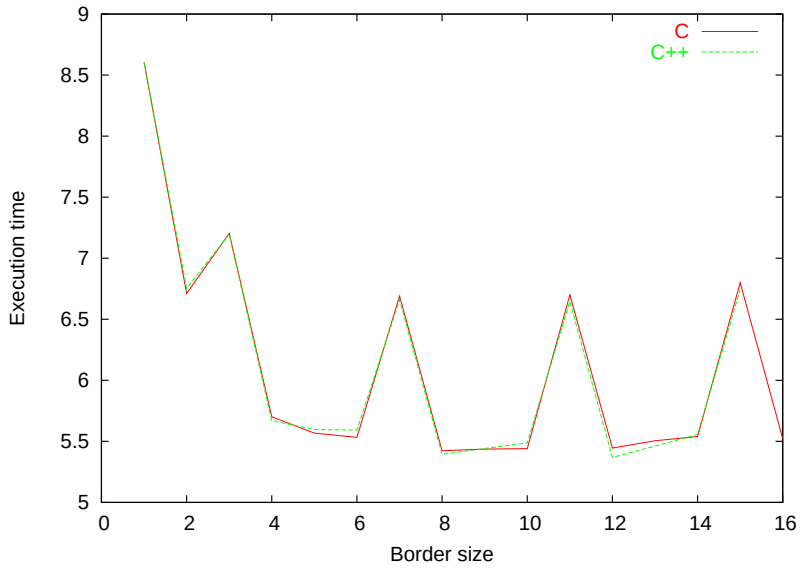


Figure 4.4: Runtimes for C and C++ versions

4.7 MPI C++ bindings

The first versions of the code used the C++ bindings for MPI, because this seemed reasonable given that the rest of the code was written in C++. But as access was gained to the different systems used in the test, we discovered that the bindings were not supported on all of the machines. Also, on the Clustis machine 5.2.1, the performance was approximately 30% lower than when using the C bindings. On the Altix machine 5.2.3, the C++ bindings refused to work with the newest GCC 3.X compiler versions[15], and the Intel compiler[16]. Restricting the program to only use an older (< 3.0) version of GCC was an undesireable option, as this version has no optimization for the Itanium processor. Most new cluster systems will likely have the 3.X version of GCC installed as the default compiler. In the end, the choice was made to revert to using only C bindings for MPI on all machines tested.

It should be noted that it is not concluded that the MPIch C++ bindings are incompatible with GCC 3.X or the Intel compiler, it would probably work with correct configuration. But in the quest of making the program portable and able to utilize the maximum performance of each system, we found that mixing C and C++ in this way is acceptable.

4.8 Timing

When timing our application, two different methods are used. They produce comparable results, but with different precision. The reason different methods

are used, is that the most precise timing is not portable.

Time Stamp Counter The first-choice of timing method is to use the INTEL x86 instruction RDTSC (Read Time Stamp Counter). The time stamp counter is a 64-bit value which is the number of clock-cycles since the processor was started. This gives the highes possible accuracy, but is not supported on all platforms used. Because of this, it is only used when timing the individual send operations. Reading the time stamp counter is done with assembly statements, and looks like: (C macro)

```
 #define RDTSC(ll) asm
volatile (".byte 0xf; .byte 0x31" : "=A" (ll))
```

This function is only supported on INTEL Pentium 1 systems and above.

UNIX timing For system with non-INTEL processors, the C function `clock_gettime` is used. This function gives microsecond or millisecond precision, dependent on the operating system and architecture, and should be supported on most UNIX systems.

This method of timing gives the actual time consumption of the application. On a multithreaded system, other tasks could be interleaved between the various part of our system. This would give wrong, and probably very random results. But since the nodes we have used when testing are used exclusively by the benchmark program, the only other threads are parts of the operating system or system environment. This "noise" will always be present, and it is correct to include these in the timing, as this benchmark tries to measure real-use performance, not some theoretical maximum performance. Other methods of timing could be to query the OS for total cpu usage by the program. This would probably give higher speedup-results, but make it harder to measure the time used by communication. It would also hide the time spent waiting for communication to finish.

4.9 Extended timing

We wanted to study how much time the communication across each edge used. For this version with extended timing, the communication had to be done less optimally than the basic version, where timing was only done for each complete communication step.

Normally, 4 edges are exchanged using immediate sends and receives. When all receives are finished, the corner values are exchanged in the same way. When corner exchanges are finished, the next iteration is started. Obviously, this communication pattern makes it hard to see how much time each exchange takes. Therefore, when doing the extended timing, another pattern must be used. First,

the time is recorded. Then four immediate sends are done. Then a loop probes for received messages. When a message is received, the sender ID is checked, the time of receive stored and then the message is stored in the correct place. The code is shown in Listing 6.1.

For comparison there is also a serial version of the code, runing on only one processor with the communication operations totally removed. (i.e. not the parallel version run on one node, as this would give a slightly wrong value for speedup).

Listing 4.3: Extended timing

```
\label{sec:extended-timing}
sendCount = 0;
if( N.rank != -1 )
  MPI_Isend( m->mat, 1, N.send, N.rank, 0,
  m->comm, &req[sendCount++] );
if( S.rank != -1 )
  MPI_Isend( m->mat, 1, S.send, S.rank, 0,
  m->comm, &req[sendCount++] );
if( E.rank != -1 )
  MPI_Isend( m->mat, 1, E.send, E.rank, 0,
  m->comm, &req[sendCount++] );
if( W.rank != -1 )
  MPI_Isend( m->mat, 1, W.send, W.rank, 0,
  m->comm, &req[sendCount++] );

RDTSC(timeTmp);

if( N.rank != -1 )
  timeCommNStart = timeTmp;
if( S.rank != -1 )
  timeCommSStart = timeTmp;
if( E.rank != -1 )
  timeCommEStart = timeTmp;
if( W.rank != -1 )
  timeCommWStart = timeTmp;

for( int i = 0; i < numNeighbours; i++ ) {
  MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, m->comm, &status );

  if( status.MPI_SOURCE == N.rank ) {
    MPI_Recv( m->mat, 1, N.recv, N.rank, 0, m->comm, &stat[0] );
    RDTSC(timeCommNEnd);
  }
  else if( status.MPI_SOURCE == S.rank ) {
    MPI_Recv( m->mat, 1, S.recv, S.rank, 0, m->comm, &stat[1] );
    RDTSC(timeCommSEnd);
  }

  else if( status.MPI_SOURCE == E.rank ) {
    MPI_Recv( m->mat, 1, E.recv, E.rank, 0, m->comm, &stat[2] );
    RDTSC(timeCommEEnd);
  }

  else if( status.MPI_SOURCE == W.rank ) {
    MPI_Recv( m->mat, 1, W.recv, W.rank, 0, m->comm, &stat[3] );
    RDTSC(timeCommWEnd);
  }
```

Chapter 5

Results

5.1 Introduction

The most important test is how much performance can be gained by varying the amount of extra calculation. This is tested with both 1- and 2-dimensional division of the matrix and with varying matrix sizes.

We also wanted to see how much, if any, overhead was added to the code by introducing the extra timing for each edge. In addition, the different communication patterns used are compared.

The number of iterations are varied according to the machine performances, to make sure the code runs long enough to get reliable timing results. The tests were done with the following matrix sizes:

- 192×192
- 512×512
- 1024×1024

On all machines, tests with 4 nodes were run. On some machines, tests with 16 and 64 nodes were also run.

For all the matrix sizes, several runs were made, each time testing every border width between 1 and 20. For each machine tested, the most relevant and/or interesting results are shown.

5.2 The machines used

Several machines were used to test the benchmark code. On each machine we have chosen the compilers based on what we regarded as natural. For the cheap clusters, GCC is a natural choice. For a processor like the Itanium, having a very complex instruction set [9], we assume that Intel's own compiler [16] is the best suited. For test runs on SGI machines, the SGI MIPSPro compiler is used.

5.2.1 Clustis

Clustis is a cluster of PCs. It consists of 32 nodes, interconnected by a switched 100Mb Ethernet network. Each node has a 1.46Ghz AMD Athlon processor and 2Gb ram. Results are based on code compiled with the GCC compiler, version 3.2.2. The optimization settings used for Clustis is `-O2`

5.2.2 Gridur

Gridur is a SGI Origin 3800 supercomputer [19] with 512 processors. As this is a production-machine, we could use a maximum of 256 processors for a few test. Results are based on code compiled with SGI MIPSPro C++ compiler, version 7.4. The optimization settings used for Gridur is `-Ofast=ip35`

5.2.3 Altix

Altix is a SGI Itanium-2 supercluster [20]. The machine is based on the Intel Itanium-2 processor, and is configured as a shared-memory machine. Results are based on code compiled with the Intel C++ compiler version 7.1. The optimization settings used for Altix is `-ftz -O2`

5.2.4 Nanna

Nanna is a minimal cluster system which consists of two dual Pentium 3 500MHz machines, connected by a 100Mbit Ethernet network. The reason to include this cluster in addition to Clustis, is that it is an SMP (Symmetric Multiprocessing) machine. We wanted to get detailed timing results from SMP machines. It is also the only machine over which we have complete control. This guarantees that the whole system only has one user when the tests are run. It also gives the possibility to have complete control of the configuration. The system runs Redhat Linux 8.0 and mpich version 1.2.5. The results are based on code compiled with the GCC C++ compiler, version 3.2.2. Th optimization settings used for Nanna is `-O2`

5.3 Comparing communication patterns

First we wanted to compare the communication patterns mentioned in section 4.3, using 1- and 2-dimensional division. The tests were run on one cluster, Clustis, and the supercomputer Gridur.

5.3.1 Clustis

The results from Clustis are shown in Figure 5.1 and 5.2. These figures show that 1-dimensional division gives the best results when using 4 nodes. On the

cluster, there is no great difference between the various communication patterns when using 4 nodes, but the one-step pattern for 2-dimensional division shows bad performance when using 16 nodes.

5.3.2 Gridur

The results from Gridur are shown in Figure 5.3 and Figure 5.4. On the supercomputer, the 1-dimensional decomposition is far quicker than the 2-dimensional. This is probably because the 2-dimensional decomposition needs the data to be copied at least once, while the 1-dimensional datatypes are designed to send and receive the data directly.

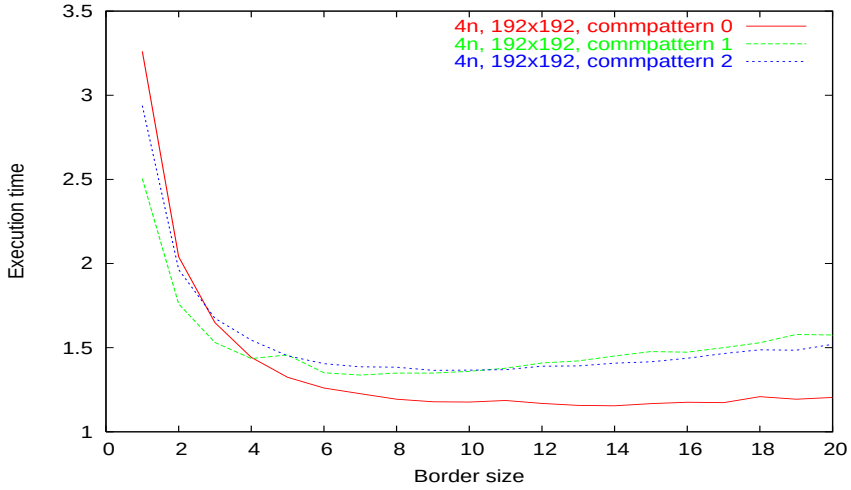
5.4 Peaks in result

The timing data shows that some border widths produce unnormally high runtimes. This is clearly something which needs closer attention. Inspection of the source code reveals nothing wrong. Multiple runs show the same result, so this is clearly not a case of sporadic traffic on the network or other load on the nodes. Also, when running the test through the batch system, the nodes assigned are exclusive. This mean that other users could only interfere the test-runs by overloading the network switch. This again, could make an impact on the total runtime, but not on the calculation part. To check whether the cache has an impact on calculation performance, tests are also done with several different matrix sizes. Using small matrices increases the possibility that the complete matrix is stored in cache, which should be positive for the performance. How clearly the peak effect shows depends on the matrix size as shown in figure 5.5. This figure also shows that it is the calculation, and not the communication part which produces the effect.

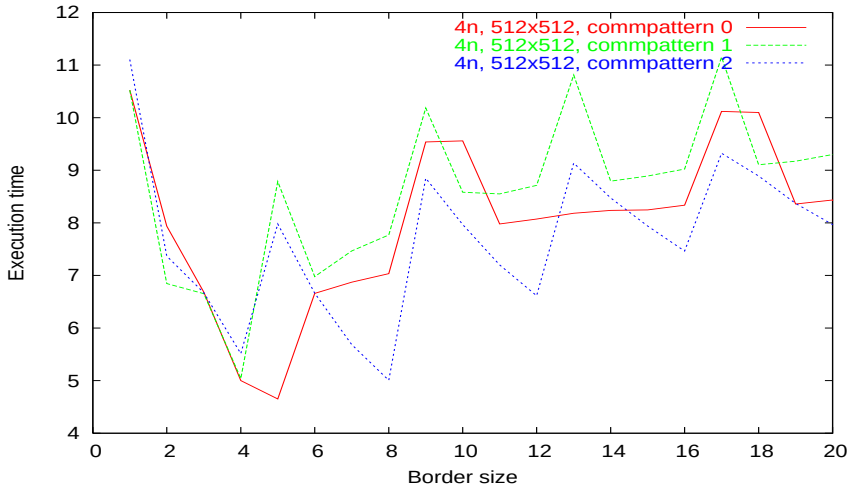
Our theory is that the cache can have something to do with the peaks. The use of cache could somehow be dependent on how many times the matrix data is accessed between each run. The communication uses MPI datatypes, and there is a possibility that the MPI code copies the data to another buffer before it is sent. This might lead to parts of the matrix being thrown out of the cache, which again leads to cache misses in the next calculation iteration.

5.4.1 Optimization

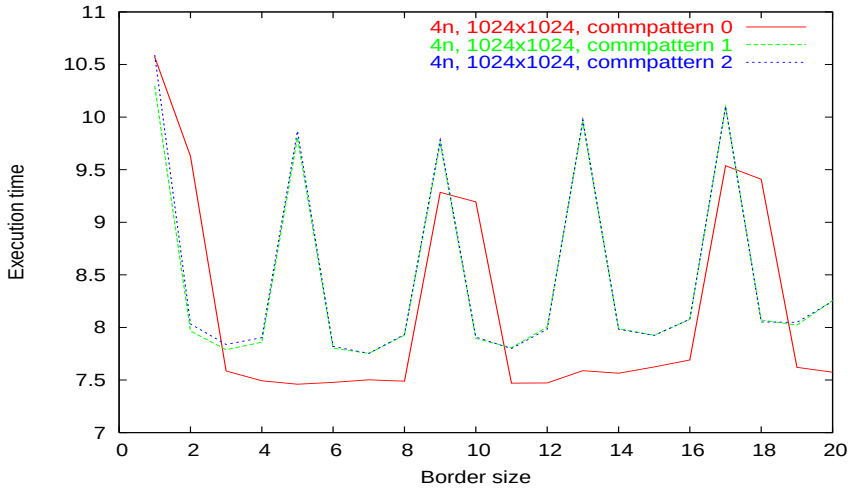
We also suspect that compiler optimizations can add to this effect, so this is also tested. The whole program is re-compiled with all optimizations turned off. The results in figure 5.6 show runtimes on Clustis for the un-optimized version. This figure looks more like it was expected to do. With this version, the performance gain from changing the edge width is very small, because the communication uses



(a) 192×192 matrix, 5000 iterations

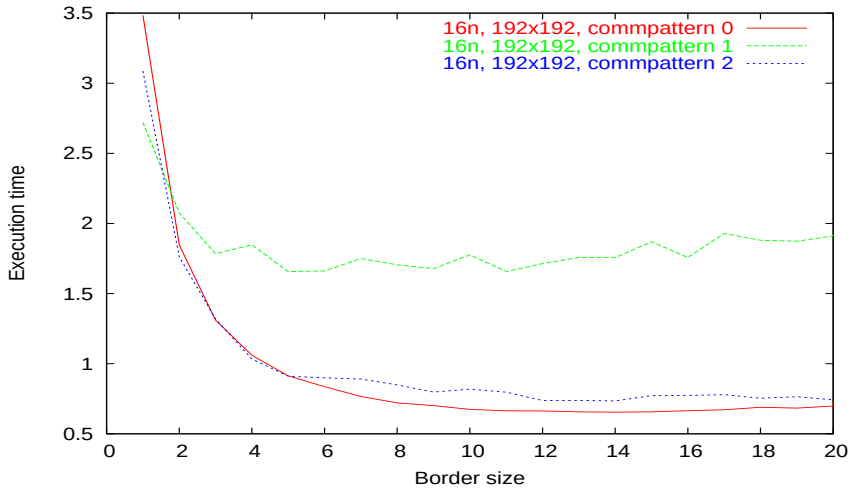


(b) 512×512 matrix, 2000 iterations



(c) 1024×1024 matrix, 1000 iterations

Figure 5.1: Comparing communication patterns for 4 nodes on Clustis



(a) 192×192 matrix, 5000 iterations

Figure 5.2: Comparing communication patterns for 16 nodes on Clustis

only using a small amount of the total time. This is because the calculation code is un-optimized. To get more realistic results without optimizing, the option to calculate every n'th pixel is used. The same test, calculating only every 5th pixel, gives the results in figure 5.7. This figure shows as expected the same trend as the former test.

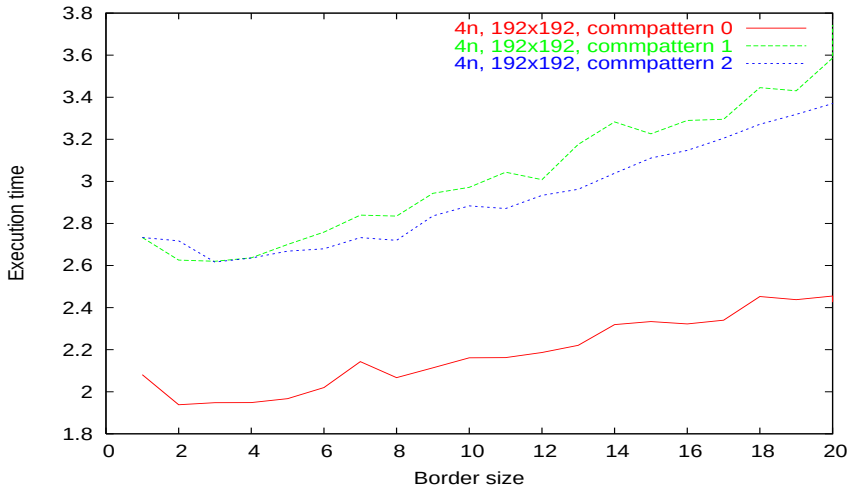
To fully understand this effect, more study is needed, maybe even closer examination of the optimized and unoptimized assembly code generated by the compiler. We feel that this is beyond the scope of this assignment. We also feel quite sure that the effect has mainly with compiler optimization to do.

5.5 Results from the serial version

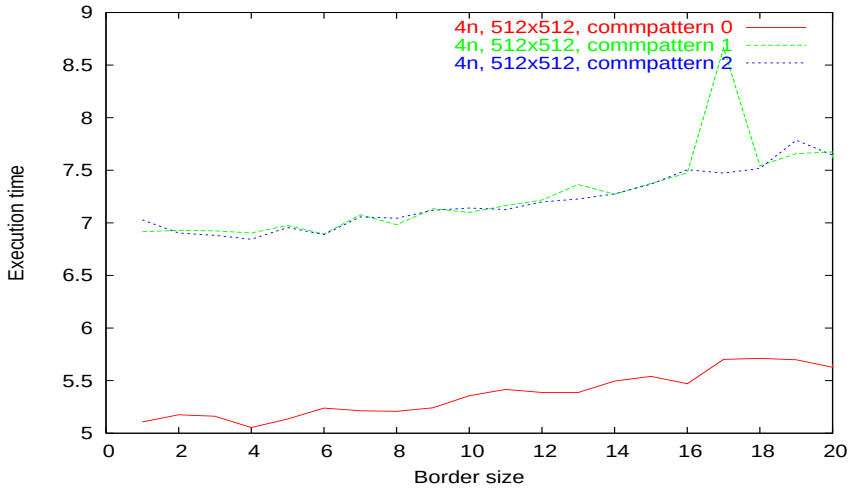
Table 5.1 shows the runtimes of the serial version on the different machines, with different data sizes. These values are used as the basis when finding the efficiency of the parallel versions.

	192×192, 5000 iter	512×512, 2000 iter	1024×1024, 1000 iter
Altix	5.09s	17.02s	36.16s
Gridur	7.09s	18.55s	50.3s
Clustis	4.93s	20.23s	43.35s
Nanna	8.75s	38.49s	82.28s

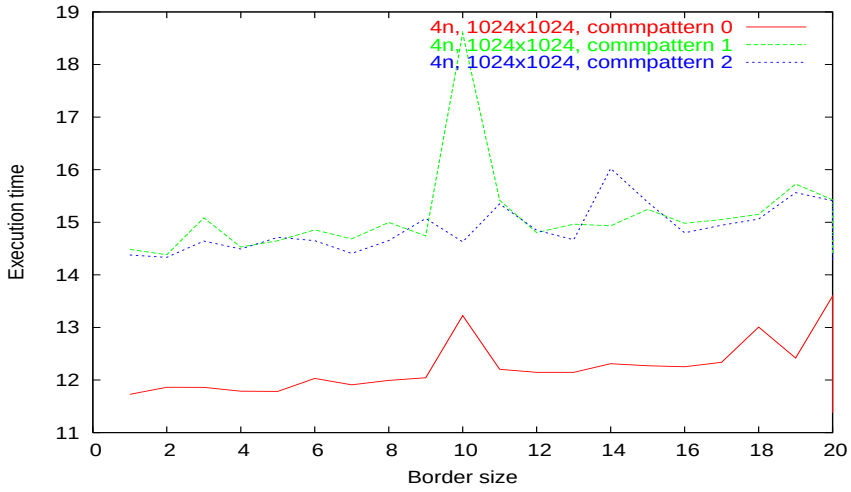
Table 5.1: Runtimes of serial version



(a) 192×192 matrix, 5000 iterations

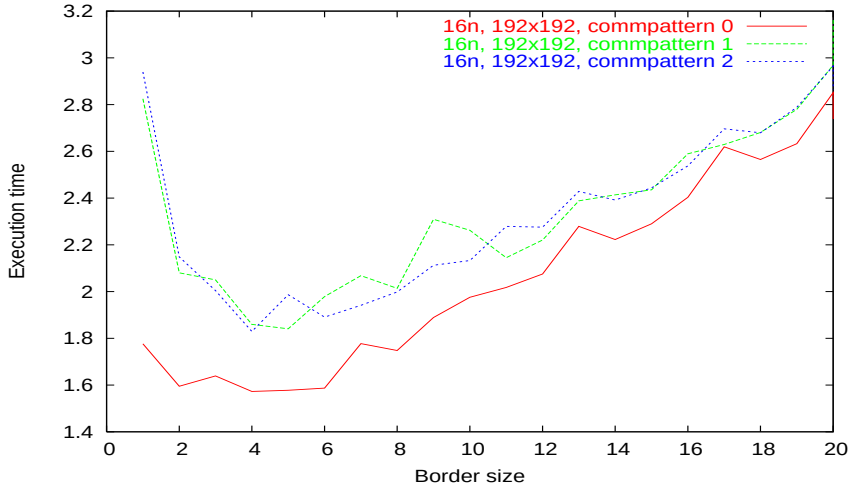


(b) 512×512 matrix, 2000 iterations

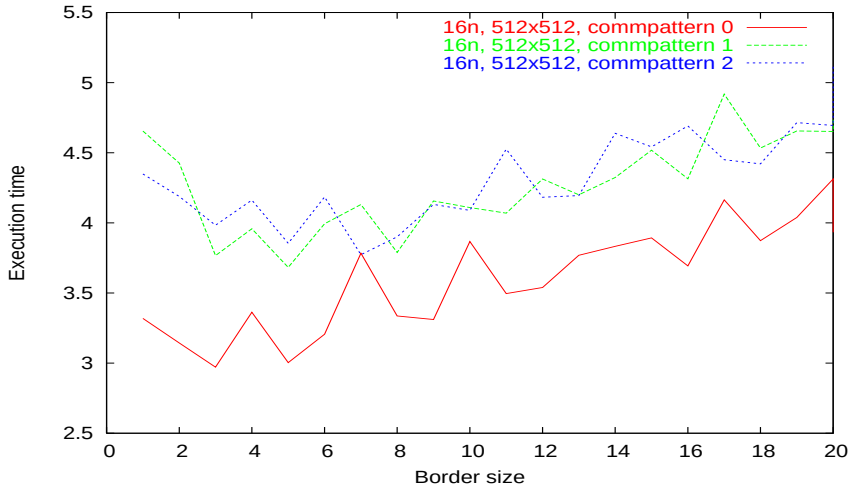


(c) 1024×1024 matrix, 1000 iterations

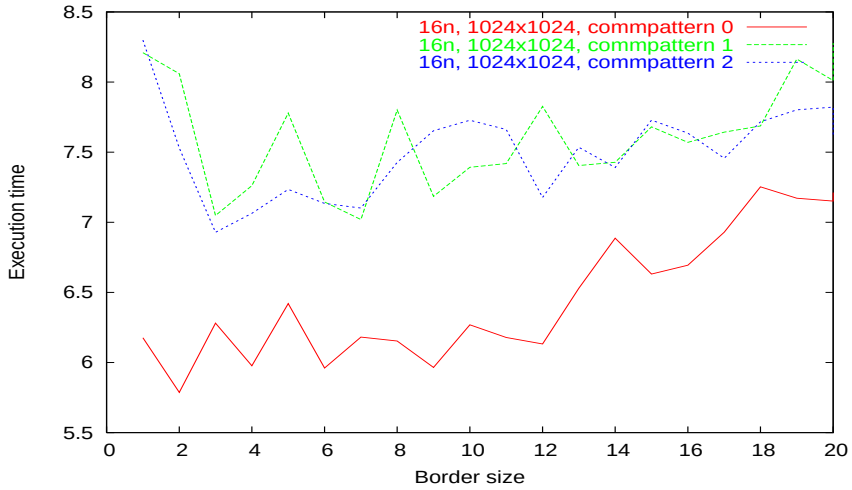
Figure 5.3: Comparing communication patterns for 4 nodes on Gridur



(a) 192×192 matrix, 5000 iterations

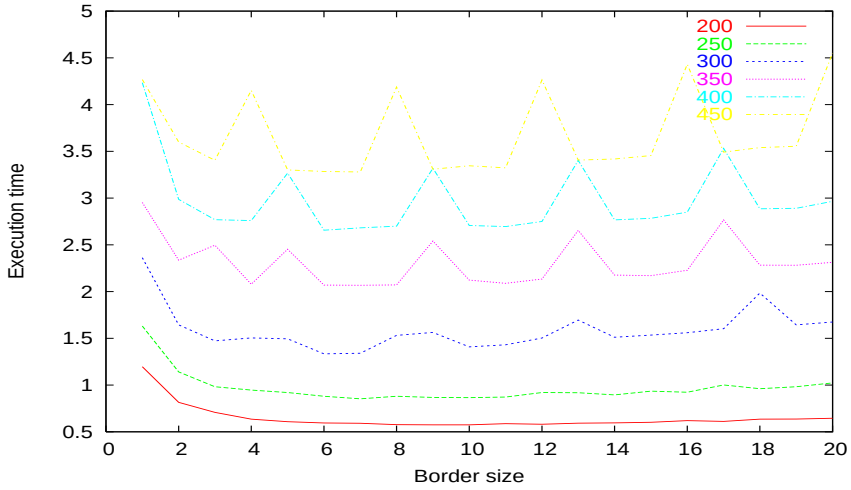


(b) 512×512 matrix, 2000 iterations

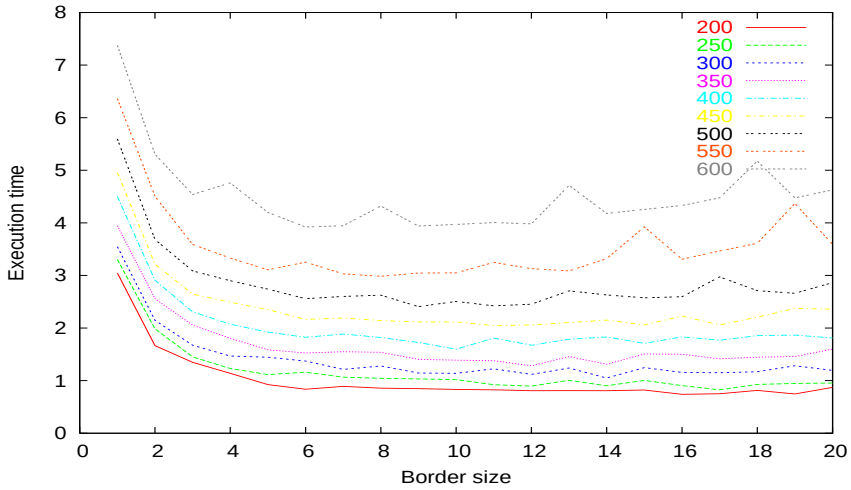


(c) 1024×1024 matrix, 1000 iterations

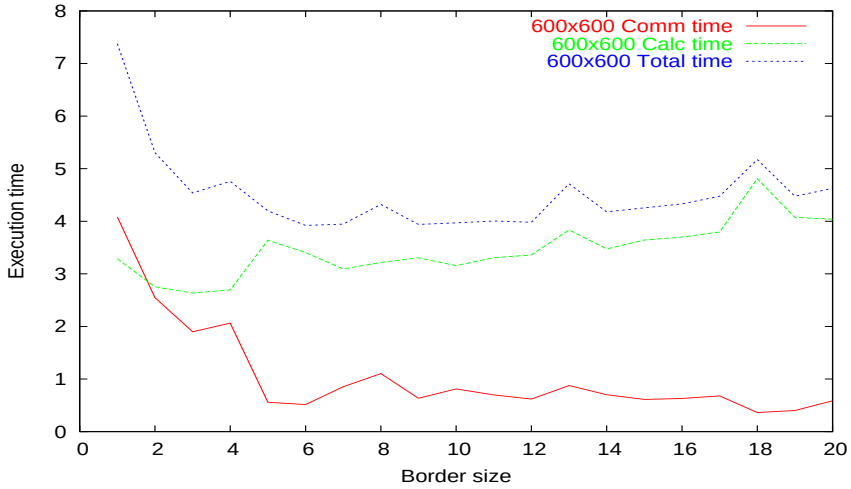
Figure 5.4: Comparing communication patterns for 16 nodes on Gridur



(a) 4 nodes, Clustis



(b) 16 nodes, Clustis



(c) 16 nodes, Clustis, Comm and Calc time

Figure 5.5: Peaks depend on matrix size

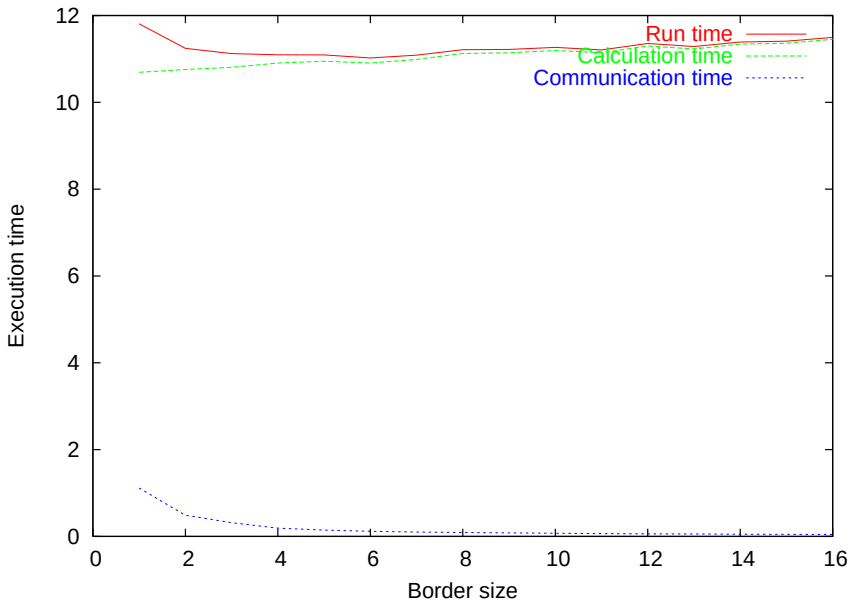


Figure 5.6: Runtimes for unoptimized version

5.6 1-dimensional division of the matrix

The first test is done with a 1-dimensional division of the matrix. The border width was varied between 1 and 20. When several runs are done, some results show great variance. To get a more reliable result, several runs are done, and the average is calculated.

5.6.1 Results from Clustis

Using 16 processors and 1-dimensional division of the matrix gives the results in Figure 5.8. The best times are found when doing between 6 and 12 iterations per exchange. The greatest efficiency increase is found with the small matrix, where the saved communication makes the solving more than 4 times faster than the basic parallel version.

5.6.2 Results from Gridur

Using 4 processors on Gridur gives the results shown in Figure 5.9, while using 16 processors gives the results in Figure 5.10. The performance increase is, as expected, rather marginal compared to the cluster. The runtimes quickly start to rise because of the extra calculations.

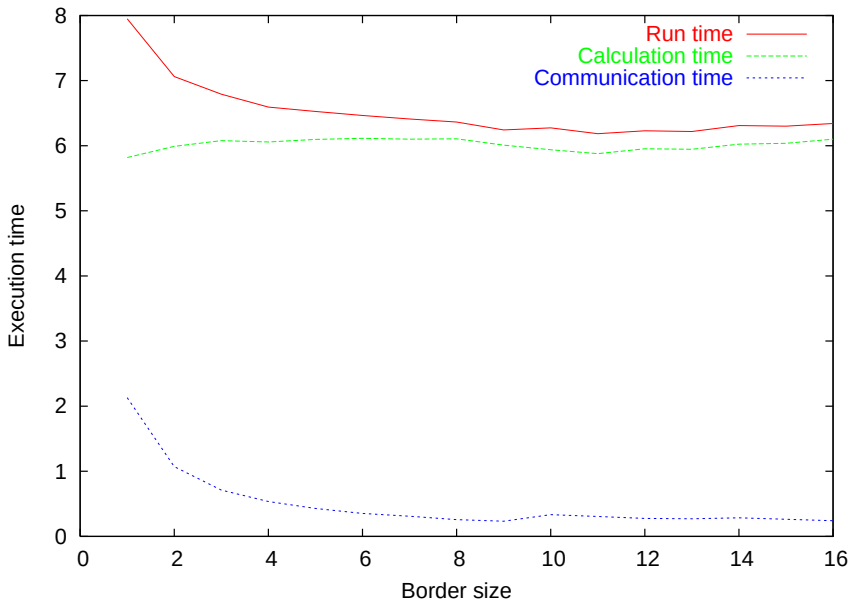


Figure 5.7: Unoptimized version, calculating every 5th point

5.6.3 Results from Nanna

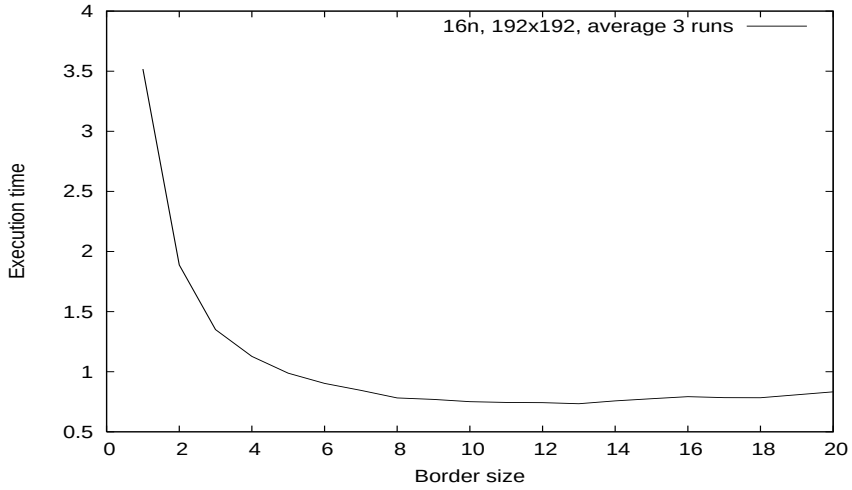
When using 1-dimensional division of the matrix, the results in Figure 5.11 are found. The performance increase with the small matrix is noticeable, but nothing like the Clustis results. This corresponds nicely with the fact that Nanna has approximately one third the processor speed of Clustis, while at the same time having fast SMP connections between two pairs of processors. This means only half of the messages must travel through the network.

5.7 2-dimensional division of the matrix

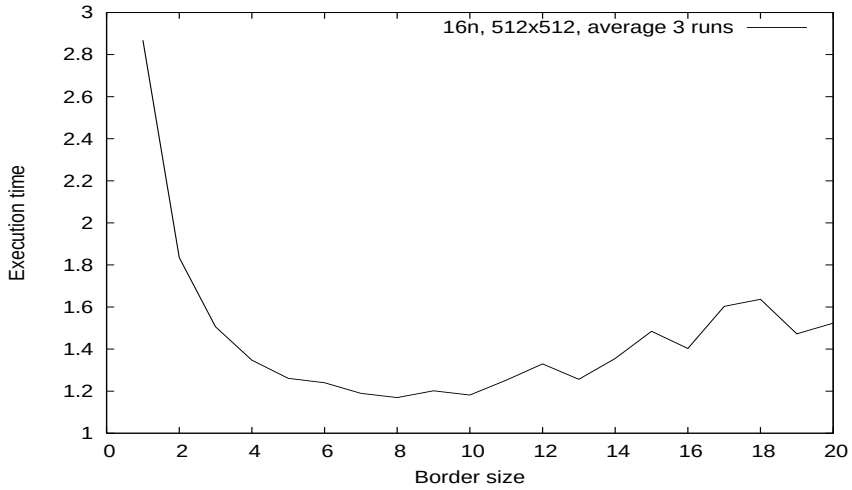
The second part of the test is done with a 2-dimensional division of the matrix. With 2-dimensional division of the matrix, the timing was also extended to measure the fraction time spent exchanging each edge. Communication pattern 2, as described in Section 3.1, is used throughout. These tests show that when using dual-cpu nodes, the communication time is unequally distributed among the edges of each processor.

5.7.1 Results from Clustis

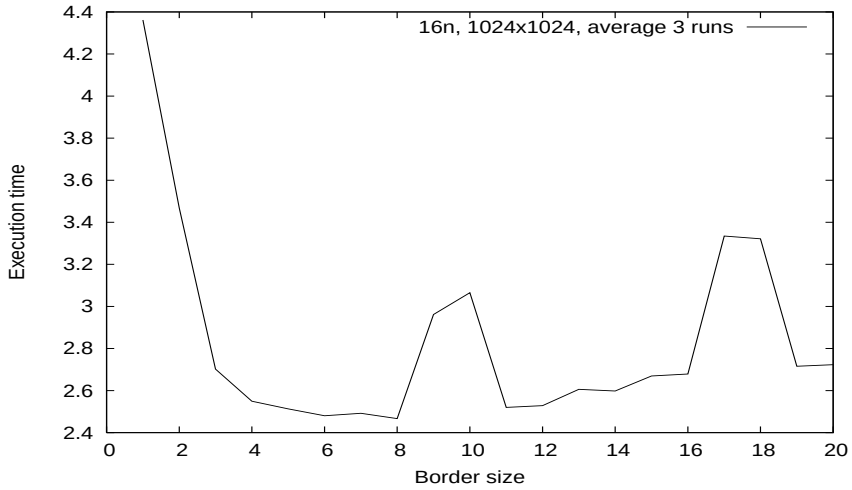
Using 4 processors, the results in Figure 5.12 are obtained. Again, the method proves very effective. The effect with several peaks, as mentioned earlier, is very clear on the largest matrix. The speedup when compared to the serial version



(a) 192×192 matrix, 5000 iterations

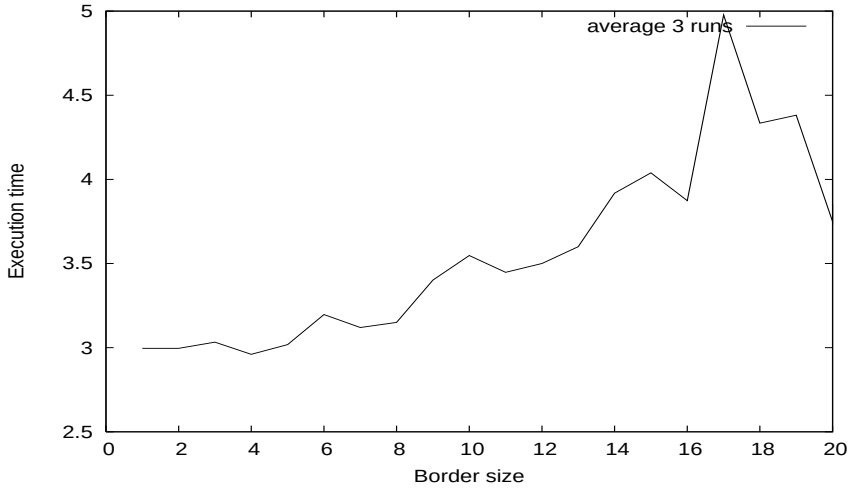


(b) 512×512 matrix, 1000 iterations

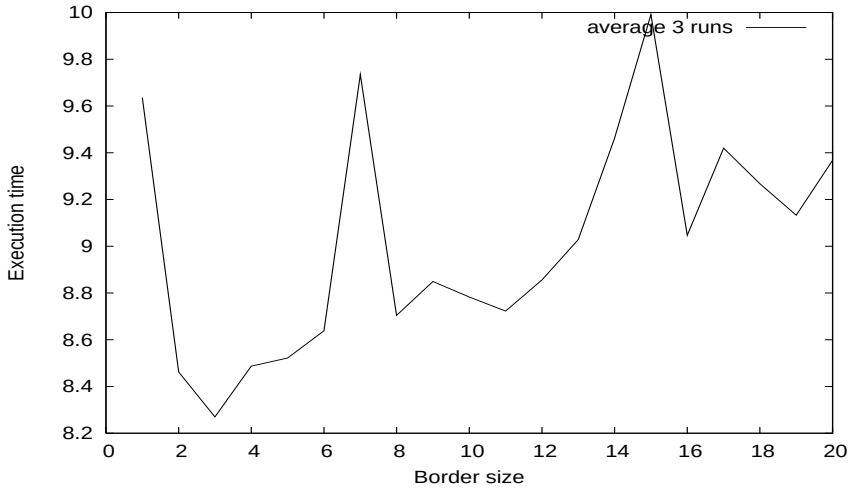


(c) 1024×1024 matrix, 1000 iterations

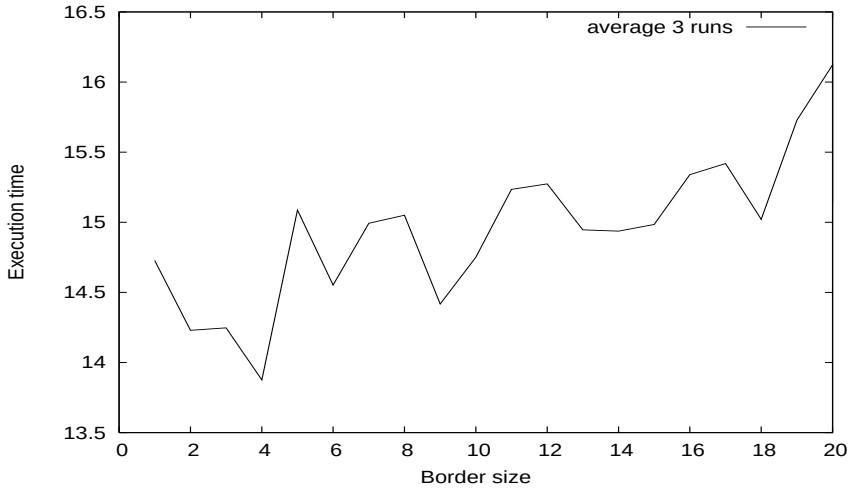
Figure 5.8: Runtimes for 16 nodes on Clustis. 1-dimensional division



(a) 192×192 matrix, 5000 iterations

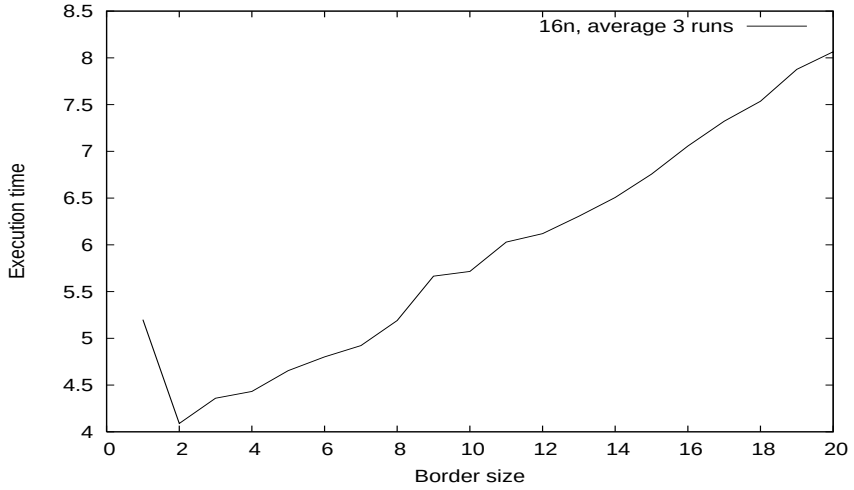


(b) 512×512 matrix, 2000 iterations

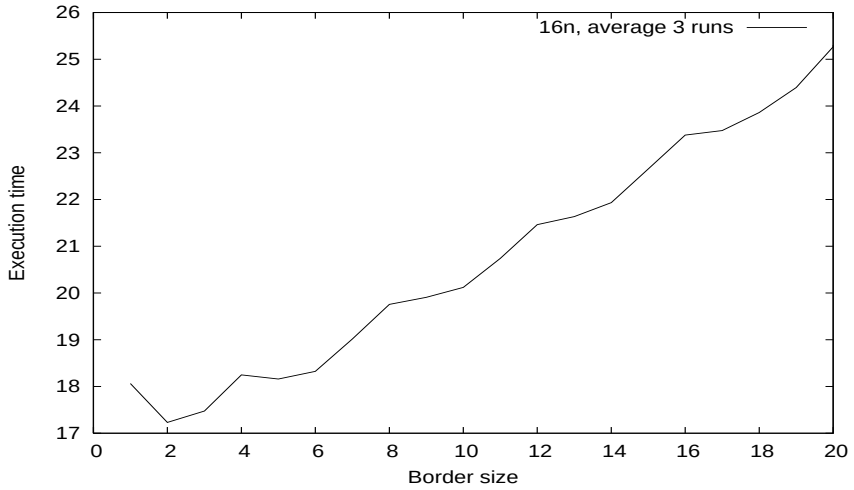


(c) 1024×1024 matrix, 1000 iterations

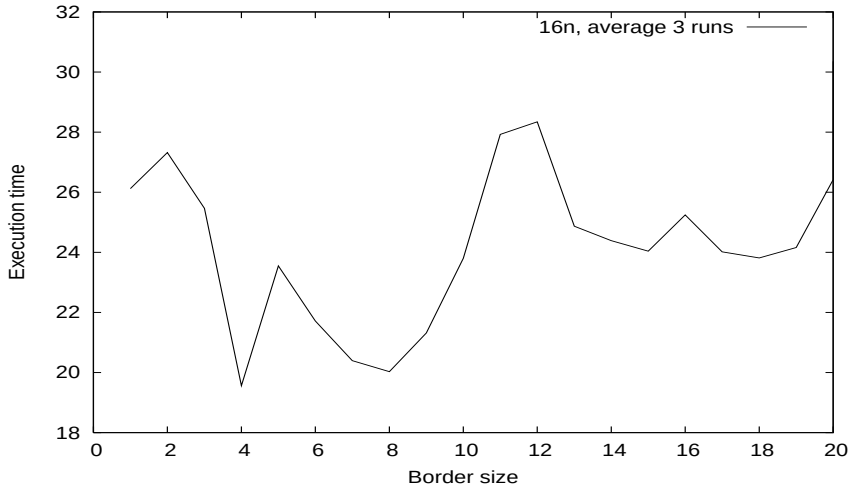
Figure 5.9: Runtimes for 4 nodes on Gridur



(a) 192×192 matrix, 5000 iterations

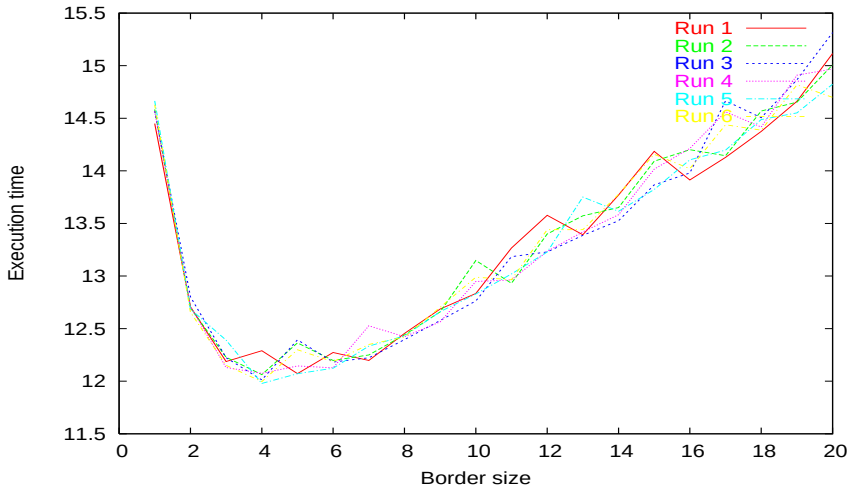


(b) 512×512 matrix, 2000 iterations

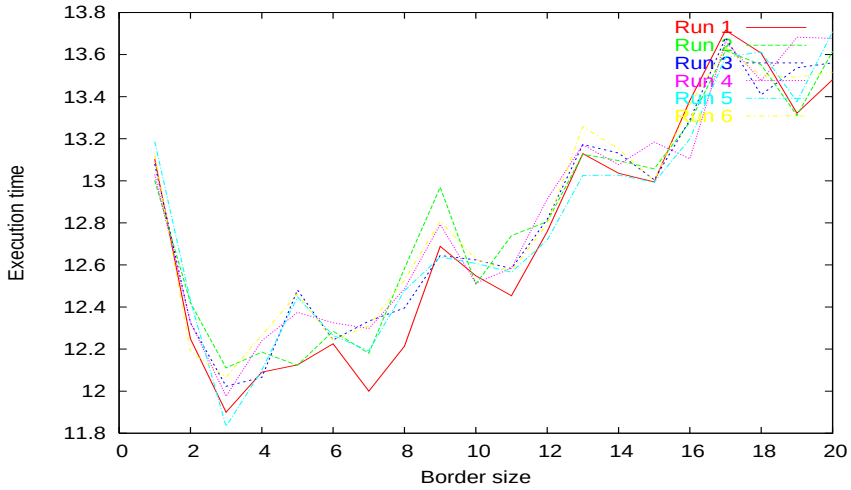


(c) 1024×1024 matrix, 1000 iterations

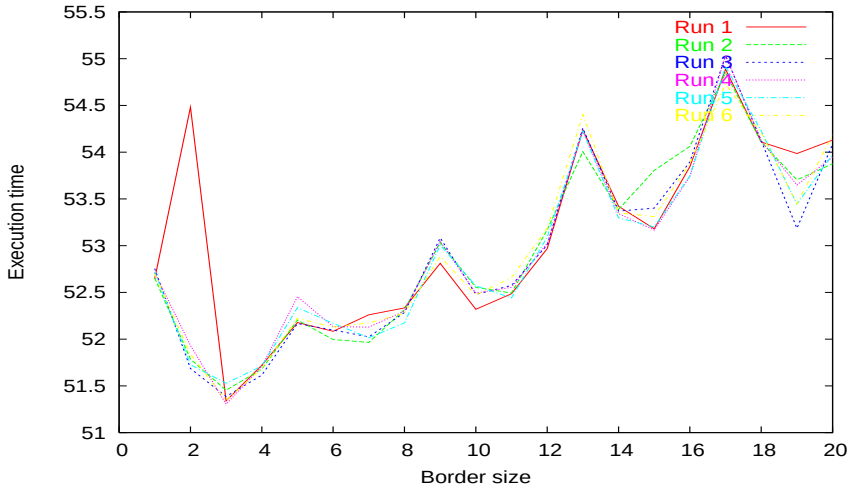
Figure 5.10: Runtimes for 16 nodes on Gridur



(a) 192×192 matrix, 5000 iterations



(b) 512×512 matrix, 2000 iterations



(c) 1024×1024 matrix, 1000 iterations

Figure 5.11: Runtimes for 4 nodes on Nanna

is shown in Table 5.2. Several matrix sizes show superlinear speedup. This is probably because the whole large matrix cannot fit in cache. When the serial version was made, no method was used to make sure the cache was utilized efficiently. In fact, going through the large, contineous memory area, guarantees that most values of the matrix must be reloaded from RAM every iteration. Because of this, we regard the performance difference between border width 1 and the optimal border as more interesting than the total speedup from the linear version.

	192×192, 5000 iter	512×512, 2000 iter	1024×1024, 1000 iter
Border 1	0.55	2.73	4.74
Optimal Border	2.66	4.74	5.74

Table 5.2: Speedup on 4 nodes, Clustis

5.7.2 Results from Gridur

The results using 4 nodes are shown in Figure 5.13. Several runs are graphed together to show how variying the results are. The method has very little effect on the two largest matrices. The average values are shown in Figure 5.14.

The tests using 16 processors show that the method is more effective than when using only 4 processors. The average results for the different matrix sizes are shown in Figure 5.15.

Using 64 processors gives the results shown in figure 5.16. Also here, the method proves effective.

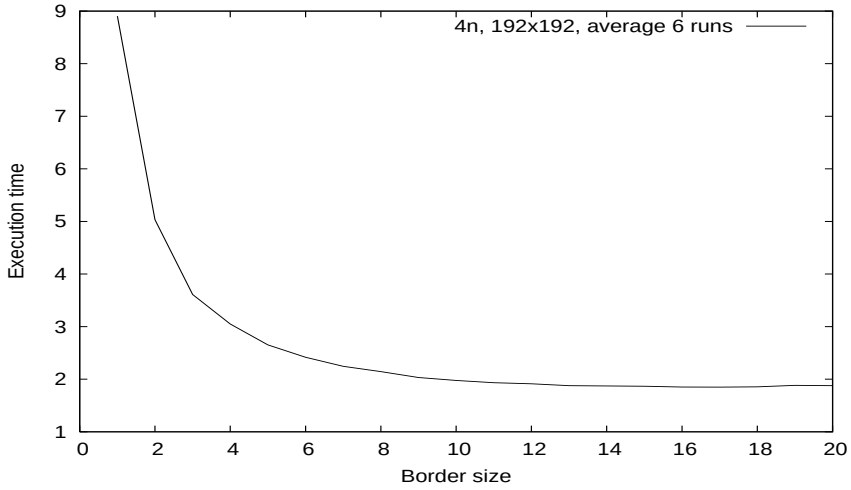
5.7.3 Results from Altix

Using 4 processors, the results in figure5.17 are obtained. Calculating the average of the runs gives the graphs in Figure 5.18. The curves look more like those of the Gridur supercomputer than those of the clusters. This is not surprising, as Altix has the same type of interconnect as Gridur.

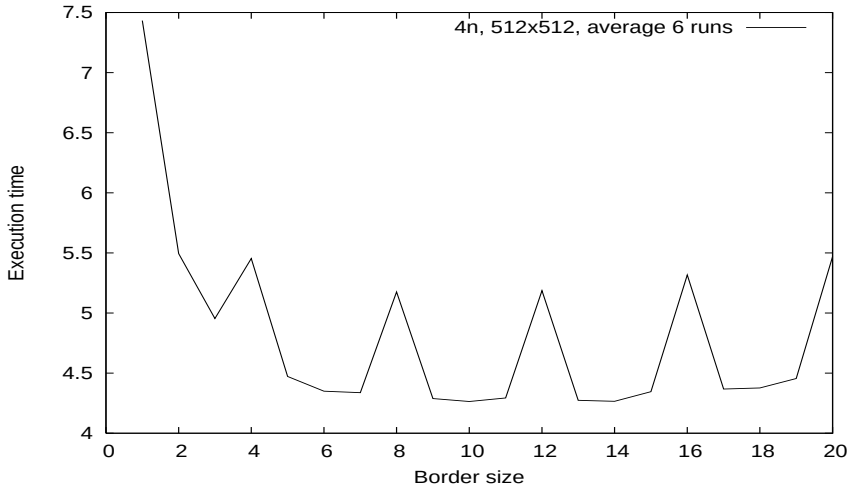
5.7.4 Results from Nanna

The results from using 4 processors are shown in Figure 5.19. Again, the results are quite similar when using 1- and 2-dimensional division. The speedup when compared to the serial version is shown in Table 5.3. Again, a superlinear speedup is found.

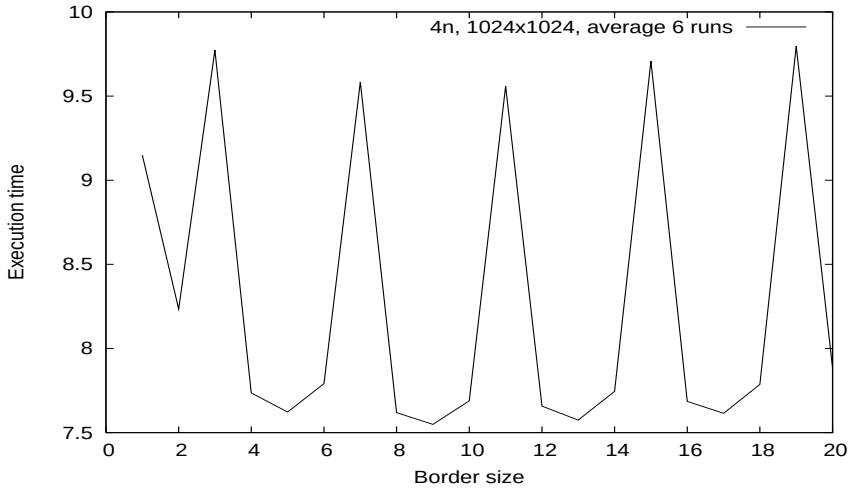
Extended timing is especially interesting for Nanna, as the system consists of two dual-cpu PCs. We assume that communication along the two edges within the SMP machine will be faster than the communication between the two machines.



(a) 192×192 matrix, 5000 iterations

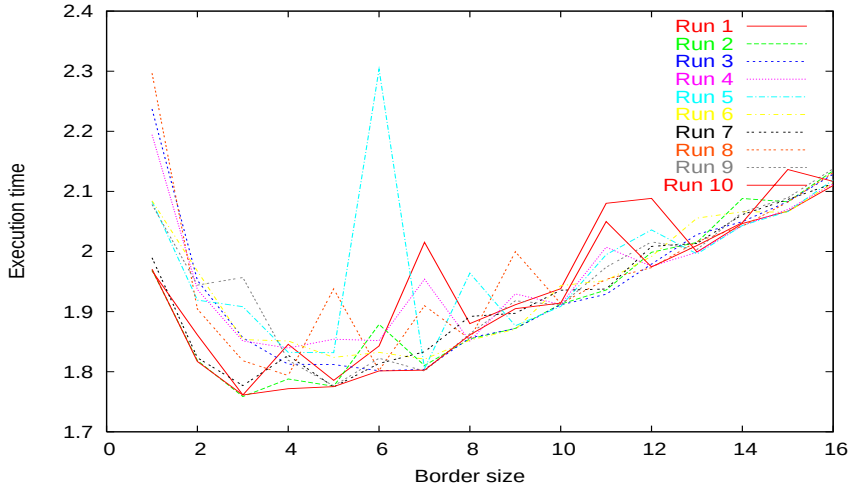


(b) 512×512 matrix, 2000 iterations

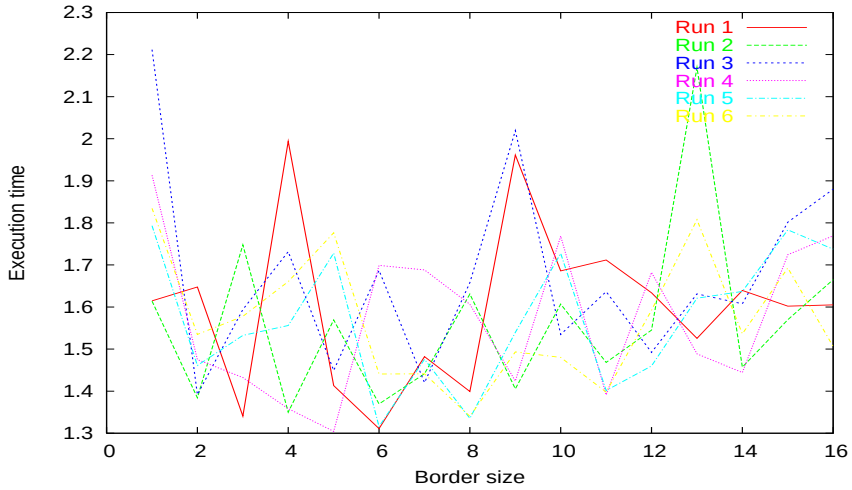


(c) 1024×1024 matrix, 1000 iterations

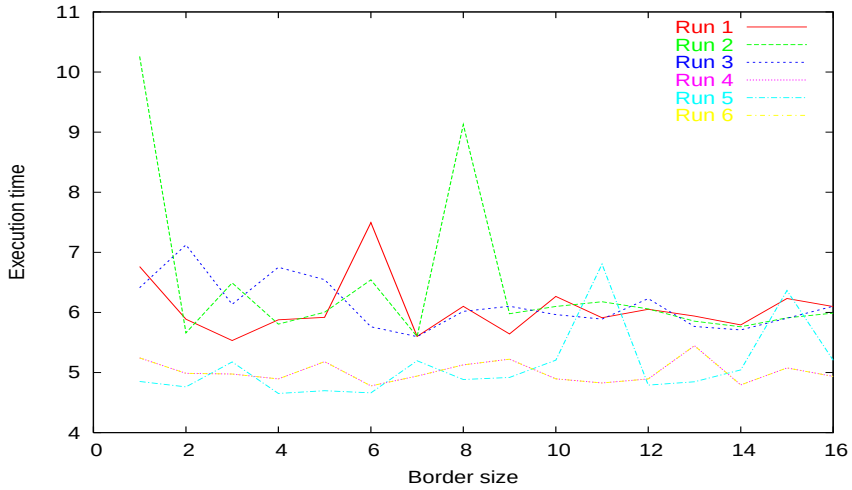
Figure 5.12: Runtimes for 4 nodes on Clustis. 2-dimensional division



(a) 192×192 matrix, 5000 iterations

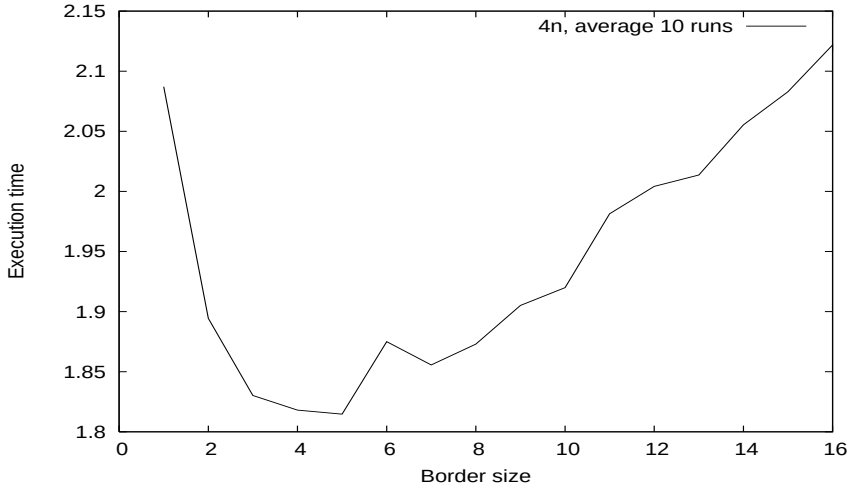


(b) 512×512 matrix, 1000 iterations

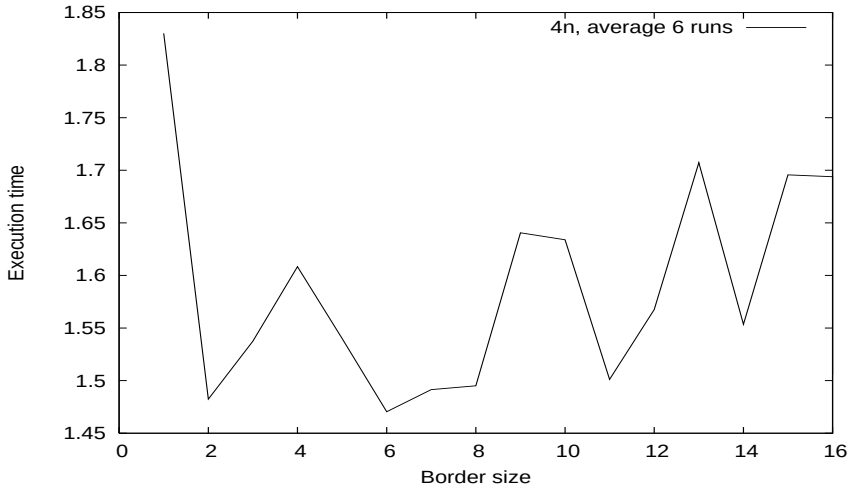


(c) 1024×1024 matrix, 1000 iterations

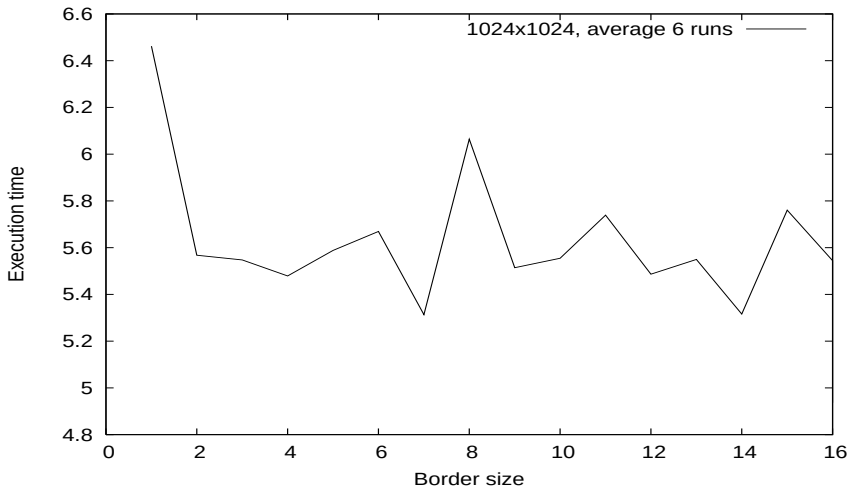
Figure 5.13: Runtimes for 4 nodes on Gridur



(a) 192×192 matrix, 5000 iterations

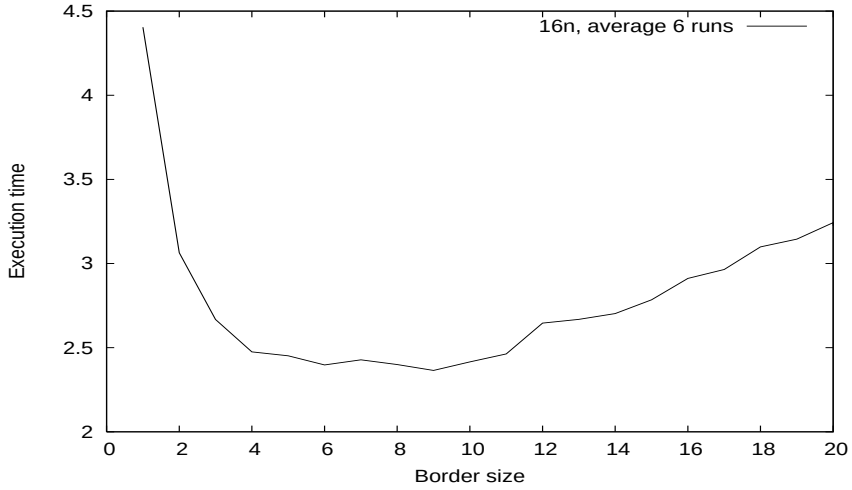


(b) 512×512 matrix, 2000 iterations

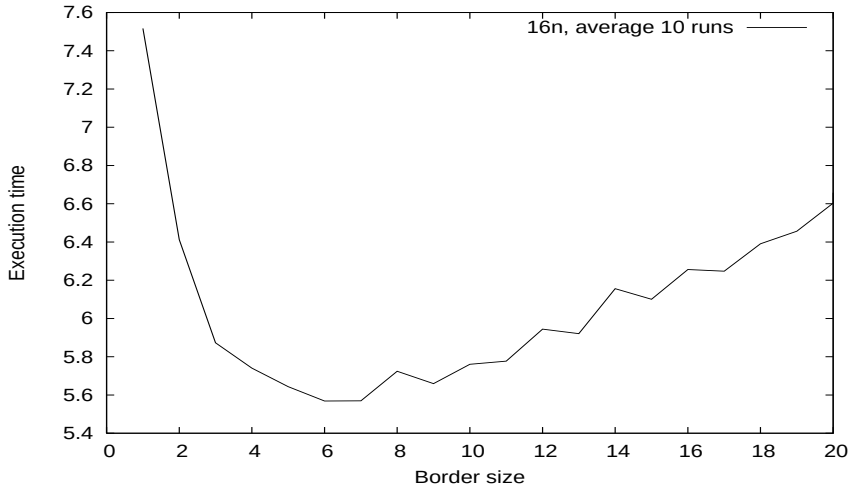


(c) 1024×1024 matrix, 1000 iterations

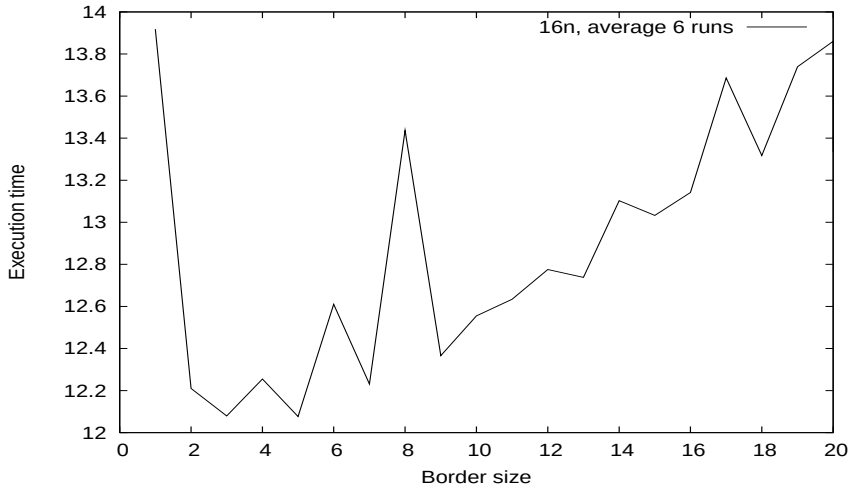
Figure 5.14: Runtimes for 4 nodes on Gridur, average



(a) 192×192 matrix, 5000 iterations

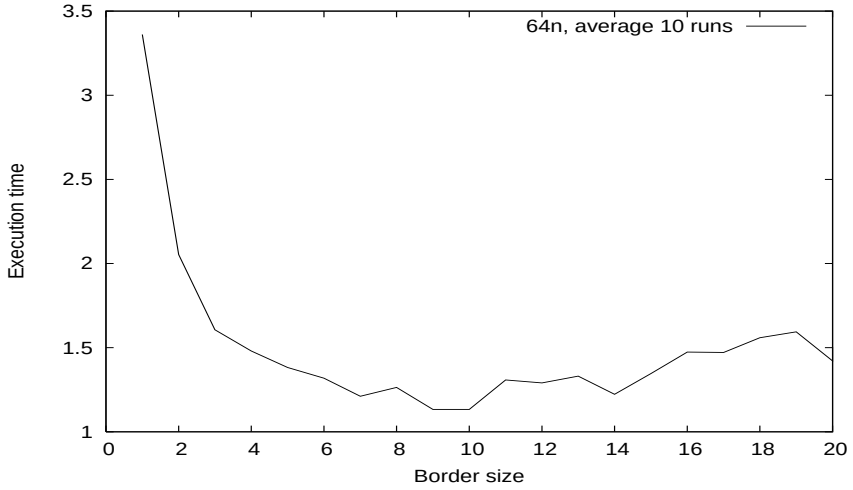


(b) 512×512 matrix, 10000 iterations

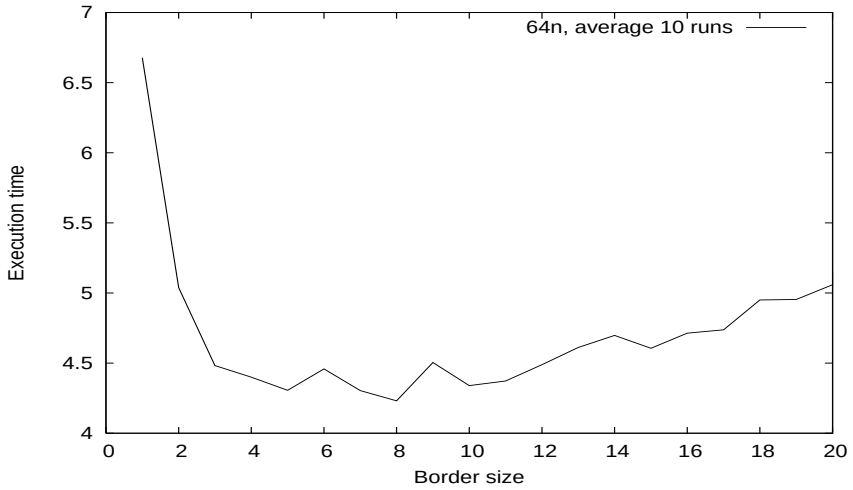


(c) 1024×1024 matrix, 5000 iterations

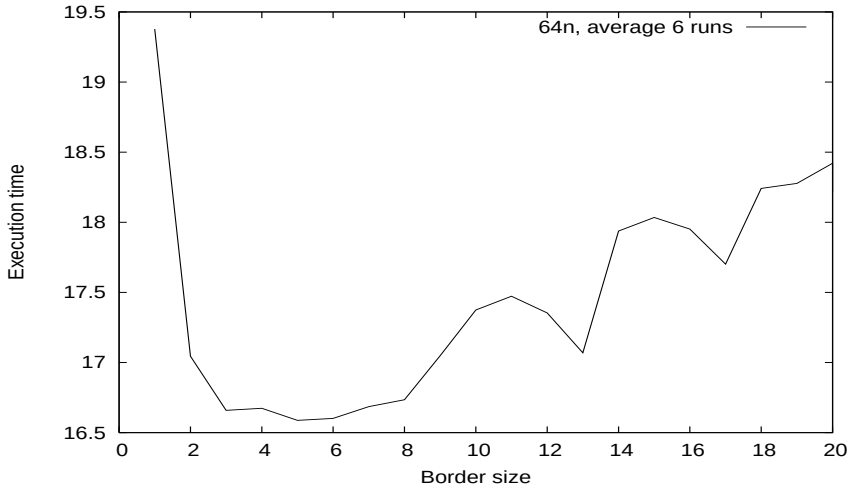
Figure 5.15: Runtimes for 16 nodes on Gridur, average



(a) 240x240 matrix, 15000 iterations

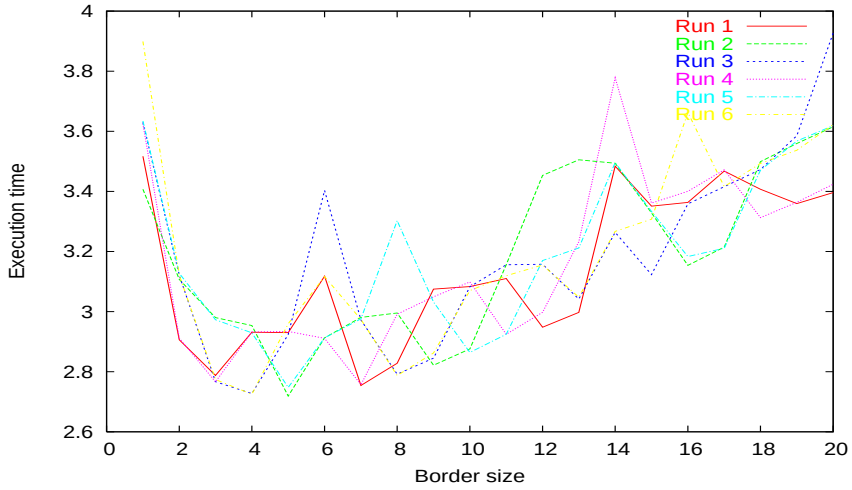


(b) 800x800 matrix, 10000 iterations

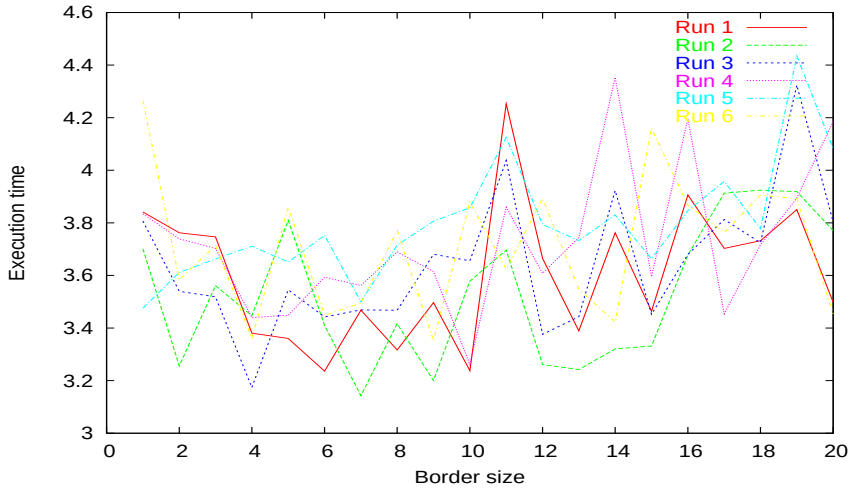


(c) 1600x1600 matrix, 10000 iterations

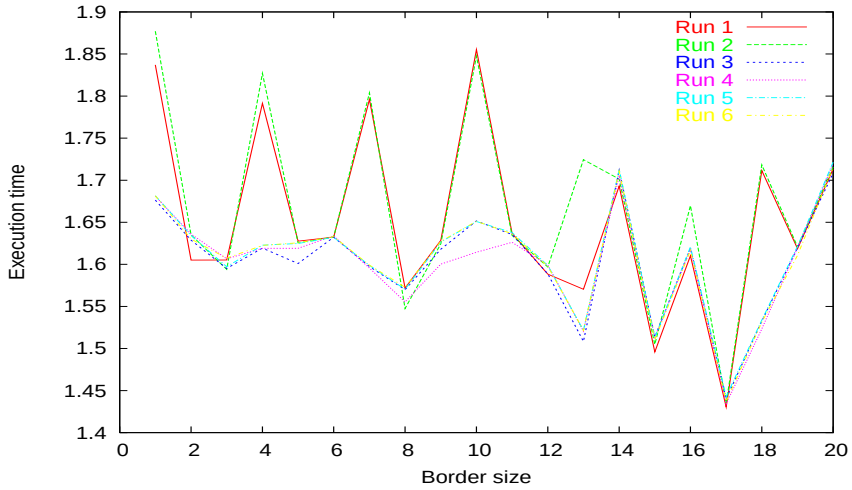
Figure 5.16: Runtimes for 64 nodes on Gridur, average



(a) 192×192 matrix, 5000 iterations

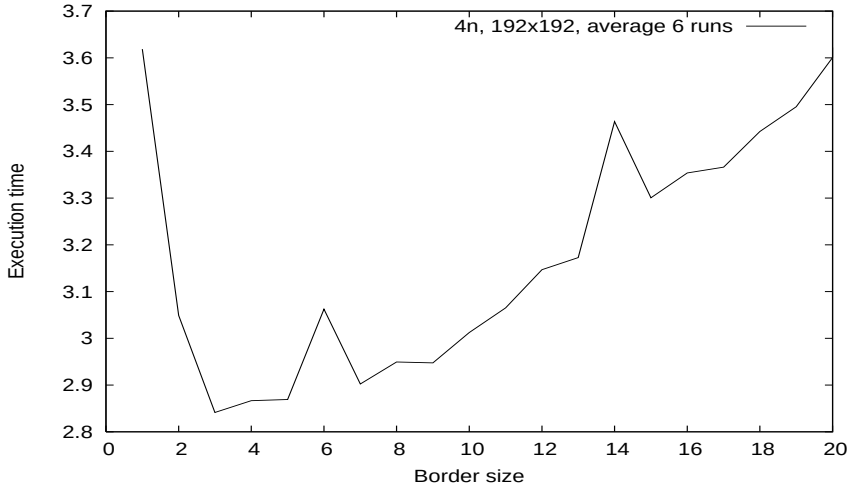


(b) 512×512 matrix, 1000 iterations

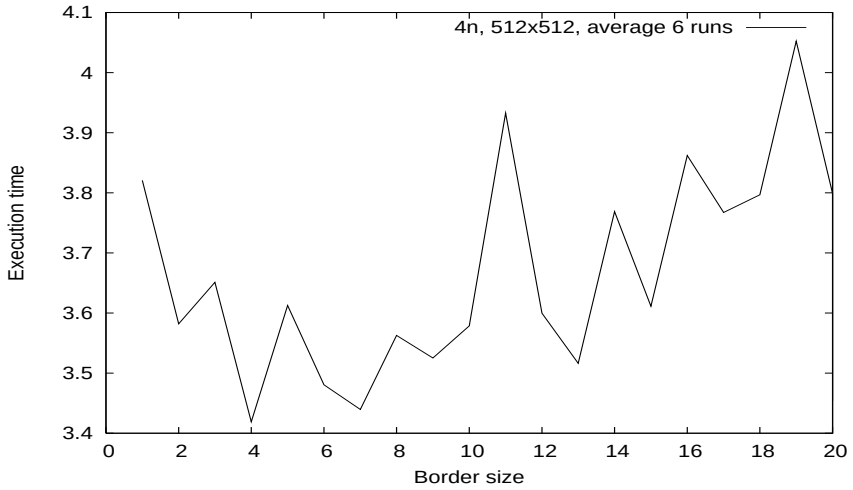


(c) 1024×1024 matrix, 1000 iterations

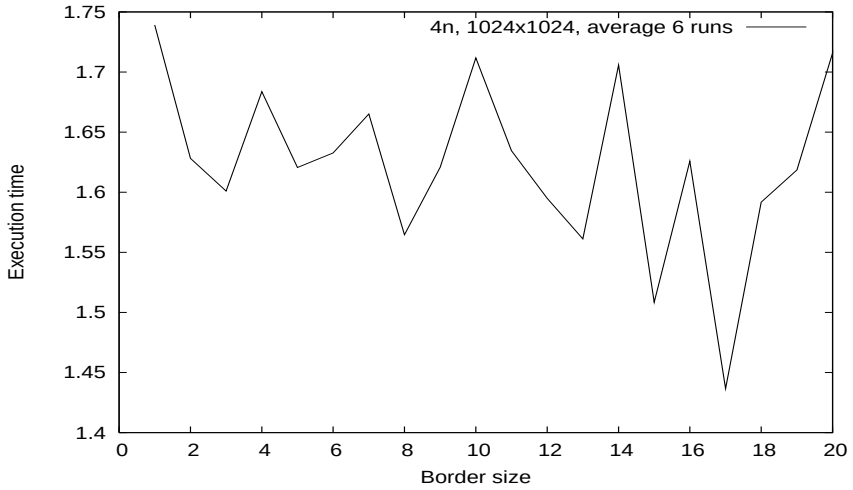
Figure 5.17: Runtimes for 4 nodes on Altix



(a) 192×192 matrix, 5000 iterations



(b) 512×512 matrix, 1000 iterations



(c) 1024×1024 matrix, 1000 iterations

Figure 5.18: Runtimes for 4 nodes on Altix, average

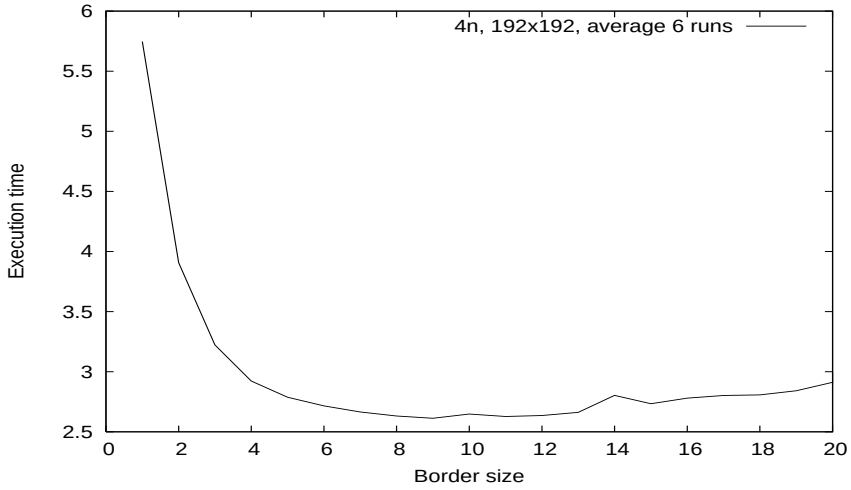
	192×192, 5000 iter	512×512, 2000 iter	1024×1024, 1000 iter
Border 1	1.52	3.59	4.17
Optimal Border	3.35	4.70	4.60

Table 5.3: Speedup on 4 nodes, Nanna

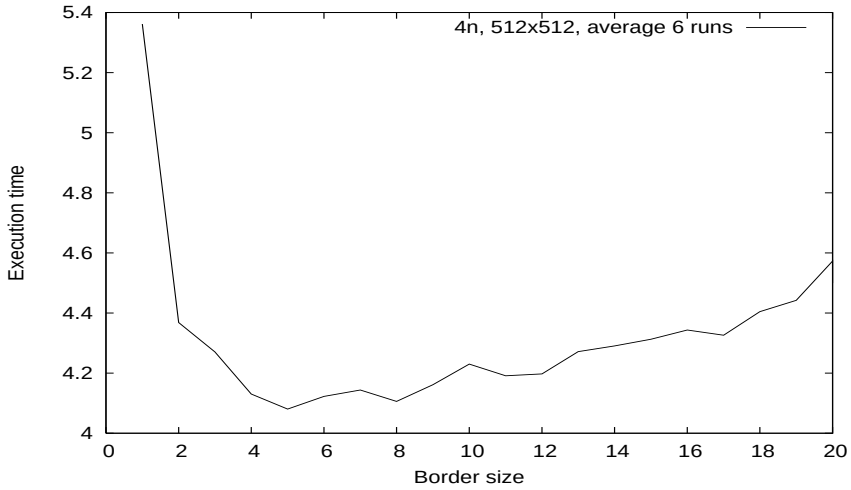
When doing extended timing, the results are as shown in Figure 5.20. The graphs are based on the average values of 6 independent runs. The communication times of each edge differ most on the largest matrix size. The method of doing extra calculation has shown to be most effective for small matrix sizes. By using the potential efficiency increase of giving each edge an optimal border, it could perhaps be possible to develop a method which works good for both small and large matrices.

5.8 Active Harmony

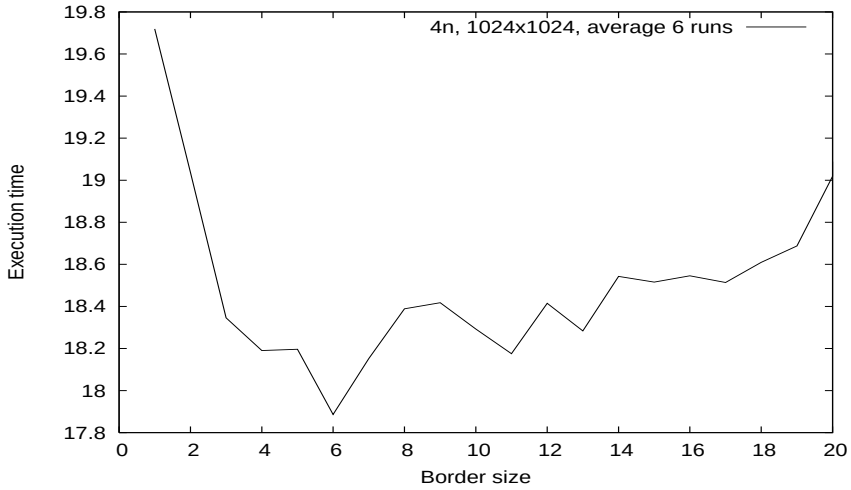
We planned to test the Active Harmony system mentioned in [24], as we felt it could have great potential for creating an auto-tunable version of our application. But too much incompatibility with the systems at hand made this impossible. Active Harmony depends on a library, "hrtime", which provides high resolution timing of process run times. This library again depends on a Linux kernel patch, which only exists for the 2.4.10 version of the kernel. The only system which we had complete access to, Nanna 5.2.4, unfortunately had a hard disk controller which refused to work with this kernel version. After we felt enough time had been used trying to get all the pieces working together, a choice was made to focus on other parts of the project instead. Because of the modular design of our program, it should be fairly easy to add a datatype utilizing Active Harmony or similar systems at a later point.



(a) 192×192 matrix, 5000 iterations

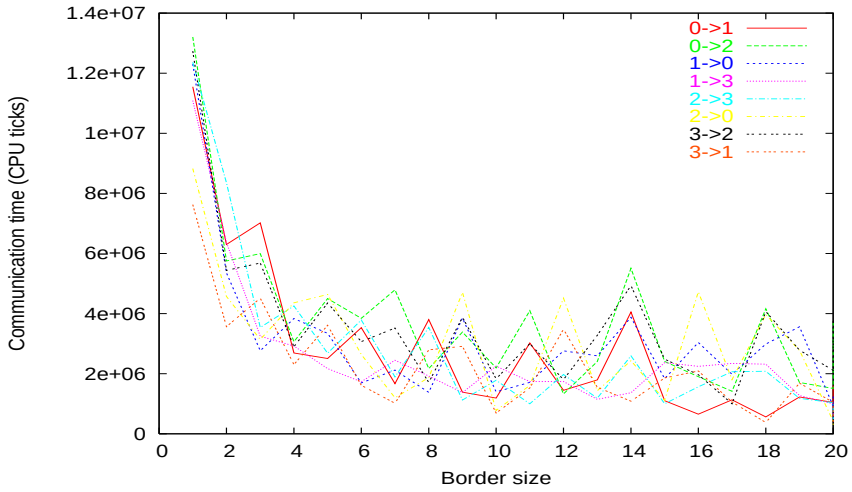


(b) 512×512 matrix, 1000 iterations

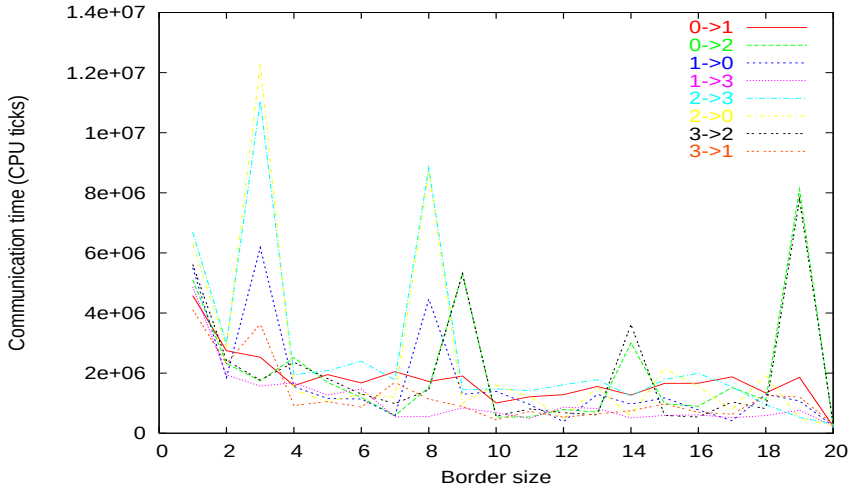


(c) 1024×1024 matrix, 1000 iterations

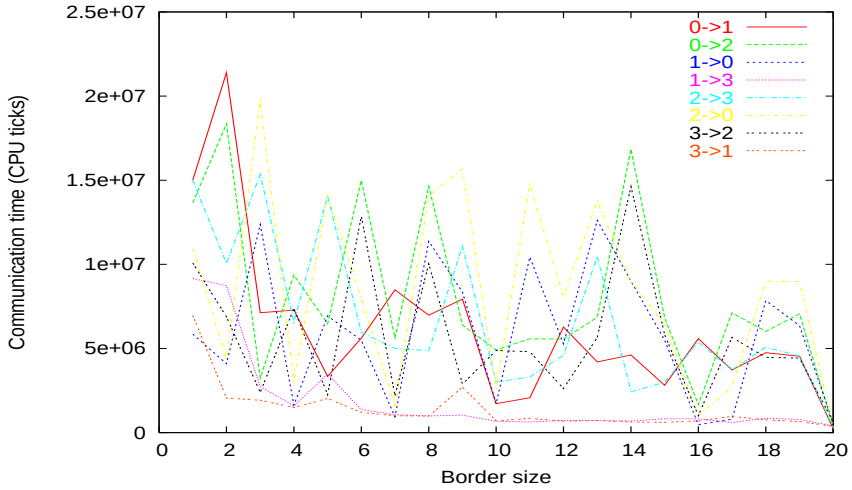
Figure 5.19: Runtimes for 4 nodes on Nanna, average



(a) 192×192 matrix, 5000 iterations



(b) 512×512 matrix, 2000 iterations



(c) 1024×1024 matrix, 1000 iterations

Figure 5.20: Communication times for each edge on Nanna

Chapter 6

Conclusions and Future Work

In this work we investigated the benefits of doing extra calculations to save communication costs for 1-D and 2-D stencil-based algorithms with domain decompostions.

It is clear from our results that the proposed method gives good efficiency benefits for this class of problems. How much performance gain one achieves in general will depend, as we expected, on the size of the problem and complexity of the calculations.

The code developed as part of this work proved to be a good benchmarking program for HPC systems. Running this benchmark on different types of machines gave results as expected: The cluster systems benefited greatly from the reduced communication, whereas the supercomputer system more quickly started to suffer from the added calculations. This is expected since the supercomputer system we tested on (a current top 500 SGI system) has a much faster interconnection network than the cluster systems we tested, who had 100Megabit Ethernet interconnects typical for such systems.

The optimal border widths for clusters were generally larger than those for supercomputers. Also, for the largest problem sizes on the supercomputer, the method shows no particular performance gain. Variable runtimes caused by system "noise" such as other running processes showed to have more impact on performance than the saved communication achieved through added computations.

Whether the grid is divided along one or two dimensions also proved to be relevant regarding the performance. Division along one dimension gives the best results when using 4 processors. When using 16 processors, 2-dimensional division is a better choice. This was valid for both the cluster systems and the supercomputers tested.

For some systems there is not necessarily just one optimal amount of overlap for a given problem size, since the execution time went up and down as the overlap was increased. We assume this to be an effect of compiler optimization combined with changing cache usage patterns when the border width changes. It

also illustrates how hard it is to manually optimize code using these techniques.

6.1 Extended Timing

The method of combining exchanges with timing shows great potential for SMP systems. Taking the time of the send and receive operations for each edge every n-th iteration adds just a tiny amount of time to the total runtime. At the same time, these values show the large difference in latency between an SMP processor pair and two processors connected by Ethernet. Timing each edge separately shows that the difference in time used is increasing when the message size increases. Using this timing information to adjust each edge separately should give more efficiency increase for systems with large matrices. We therefore propose this method be used in automatically optimizing algorithms.

6.2 Future work

This work can be extended in many directions. The following subsections list some of the most obvious ones that we hope to look into in the near future.

6.2.1 Automatic Tuning

Using the extended timing information to vary the amount of extra calculation individually for each edge seems like a feasible method to increase efficiency. Good heuristics should be found to stabilize the border widths when optimal values are found. At the same time, a method which is able to adapt to changing conditions is often even better. Our method may be shown to be particularly useful on systems where the processes are automatically migrated by the operating system.

6.2.2 Existing Libraries

Future work should include looking at the possibilities of enhancing existing libraries and applications to use this method of optimization. As mentioned earlier this is an active field of research. Clusters as an alternative to supercomputers is also a relatively new idea, and there exists a large amount of applications who require a supercomputer to run with good efficiency. Looking more closely at existing applications would make it possible to extract performance-data from realistic use in addition to the benchmark results. A study should be made of key applications to measure the applicability and efficiency of tuning them for cluster-like environments. The already mentioned API “Active Harmony” should also be tested on stencil-based communication.

6.2.3 Mathematical Model

It could be possible to create a mathematical model for finding the optimal parameters, given system timing data. The main problem is probably to determine if there are too many factors to consider.

6.2.4 Other Algorithms

It would be interesting to look at other classes of algorithms used in parallel applications, to explore and exploit trading extra calculation for communication.

6.2.5 Usability

The program is supposed to be used as a benchmark for stencil based communication. As it is now, it is fully usable, but a list of possible extensions could include:

- 3D. The program should be extended from using only 2 dimensional matrices to also use 3 dimensional data.
- Command-line check. The program currently does not check for mis-typed optional command line parameters. The program will only give warnings when mandatory parameters are missing.
- Scripts. The package consists of several scripts for doing test runs and creating graphs. This could be controlled by a master script, submitting jobs and creating graphs of the results.
- Standard values. It would be interesting to have a "standard test" with predefined parameter values. This would make it easier to compare results from different systems and facilitate the creation of a results database. To find good values, more research should probably be done, testing the code on several computers and comparing these with real-life applications.

Bibliography

[1] Khalid Al-Tawil and Csaba Andreas Moritz. Performance modeling and evaluation of MPI. *Journal of Parallel and Distributed Computing 61*, pages 202–223, 2001.

[2] Zizhong Chen, Jack Dongarra, Piotr Luszczek, and Kenneth Roche. Self adapting software for numerical linear algebra and LAPACK for clusters. Technical report, University of Tennessee, 2003.

[3] Yong Eun Cho. *Efficient resource utilization for parallel I/O in cluster environments*. PhD thesis, University of Illinois at Urbana-Champaign (UIUC), 1999.

[4] David Culler, Richard Karl, and David Patterson. Logp: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[5] Chris Ding and Yun (Helen) He. Effective methods in reducing communication overhead iu solving pde problems on distributed-memory computer architectures. Technical report, Lawrence Berkeley National Laboratory, http://www.nersc.gov/research/SCG/acpi/pubs/ghc2002.ppt, 2002.

[6] Jack Dongarra. Performance of various computers using standard linear equations software. Technical report, University of Tennessee, Knoxville TN, 37996, url:http://www.netlib.org/benchmark/performance.ps, 1985.

[7] Jack Dongarra and Victor Eijkhout. Self-adapting numerical software for next generation applications. Technical report, University of Tennessee, 2002.

[8] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computing C-21, No. 9, Sep 1972, pp 948-960*, C-21(9):948–960, Sep 1972.

[9] Richard Gerber. *The Software Optimization Cookbook. High-Performance Architectures Recipes for the Intel Architecture*. Intel Press, 2002.

[10] Jeffrey K. Hollingsworth and Peter J. Keleher. Prediction and adaptation in Active Harmony. Technical report, University of Maryland, 1998.

[11] Jeffrey K. Hollingsworth, Peter J. Keleher, and Dejan Perkovic. Exposing application alternatives. Technical report, Department of Computer Science, University of Maryland, 1999.

[12] Scampi benchmark results, url:http://ilacs.com/index.php?loc=8.

[13] Gigabit ethernet to the desktop, url:http://www.accs.com/p_and_p/gigabit/results_lmbench2. 2003.

[14] Dolphin interconnect solutions inc - benchmarks, url:http://www.dolphinics.com/products/benchmarks.html.

[15] Gnu software webpage, url:http://www.gnu.org.

[16] Intel webpage, url:http://www.intel.com.

[17] MPI - the message passing interface standard, url:http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[18] Pallas GmbH, url:http://www.pallas.de.

[19] Sgi origin 3000 supercomputer, url:http://www.sgi.com/origin/3000/overview.html.

[20] Sgi altix 3000 supercluster, url:http://www.sgi.com/servers/altix/.

[21] Top500 supercomputer sites, url:http://www.top500.org/.

[22] Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc, 1997.

[23] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–161, 1998.

[24] Christian Tapus, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: Towards automatic performance tuning. Technical report, Department of Computer Science, University of Maryland, 2002.

[25] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing 27*, 2001.

Appendix A

Compilation

The application has a makefile which can be used when compiling and testing the code. The makefile is tested with GNU make, and features separate settings for the systems used in the tests. The hostname is used to determine the target system, which is obviously not very portable. For future extensions, some time could be spent investigating GNU automake or similar sytems to automate the build process. When compiling on other systems than the ones defined in the Makefile, default settings are used. It is advised to look at the different options used for different systems if the compilation fails. The parts of the Makefile who need to be changed are:

```
HOST = $(shell hostname)
CLUSTIS = ClustIS
GRIDUR = gridur

ifeq ($(HOST), $(CLUSTIS))
MPI_CC = mpicc
MPI_C++ = mpiCC
CFLAGS = -Wall -static -O2
MPI_C++LINKER = $(MPI_C++)
endif

ifeq ($(HOST), $(NANNA))
MPI_CC      = /usr/share/mpich-1.2.5/bin/mpicc
MPI_C++      = /usr/share/mpich-1.2.5/bin/mpiCC
CFLAGS     = -Wall -O2
MPI_C++LINKER = $(MPI_C++)
endif

ifeq ($(HOST), $(ALTIX))
MPI_CC = ecp -ftz -D_NO_ASM
```

```
MPI_C++ = ecp -ftz -D_NO_ASM
MPI_C++LINKER = $(MPI_C++) -lmpi
endif
```

The code were compiled with optimizations turned on, as this is a natural thing to do with high performance code. During development we noticed that some of the versions gave erroneous results on some platforms with maximum optimization. These errors were found by looking at the image of the matrix calculated by the code. We did not try any further to find the cause of these errors. As long as the code worked with a lower optimization setting we assumed the errors to be with the compiler.

In addition to the compilation, the timing had to be done differently on different platforms. Only some Intel platforms support the high-precision RDTSC function, which reads the clock cycle counter. On other systems, the clock_gettime function can be used, which gives microsecond resolution. The last option is to use the time function. This function gives millisecond precision.

Appendix B

Usage

This chapter describes how to run the test code and the various options.

B.1 Batch

The test code uses both the processor and network to a high degree. As with every benchmark it is important to give the program exclusive right to the nodes it uses. This is normally done through a batch system, and example scripts are part of the program package. If possible, one should assure that other nodes are not limiting the available network bandwidth. This can reduce sytem performance when using only some of the nodes in a system with common bus or ring network.

B.2 Command line

The following command line switches control how the program is run.

-dim The dimension of the global matrix, in pixels. Width and height are equal. The size of the matrix must also be divisible by the number of processors in each direction. If not, the remainder part of the matrix will not be calculated, and the timing results will not be correct.

-iter The number of iterations to perform. This number should be set to a high enough value to get accurate results.

-b The width of the extra border around the local area. This is the value controlling how much extra calculation to do. The total number of communication steps are given by `ITER B`. The border must at least be 1 for the algorithm to work. The maximum border width is not given, but it obviously makes little sense to use a border so large that the local part of the matrix approaches the global matrix size.

-npx and -npy Number of processors in X and Y direction. Controls how the global grid is divided. When npx is 1, the division is the same as when using the 1D matrix. `npx * npy` must equal the number of processors. Also, when using the 1D matrix and datatypes, `npx` must be 1, and `npy` must be equal to the processor count.

-commP Controls the communication pattern used. At the time, the following patterns are implemented, as listed in the file `Values.h`:

- **0: DT1D** 1 dimensional division of the global matrix. MPI Datatypes are not used, to optimize for 1 dimensional division (no intermediate saves necessary).
- **1: DT2D_V1** 2 dimensional division of the global matrix, following the communication pattern given in figure ..
- **2: DT2D_V2** 2 dimensional division of the global matrix, following the communication pattern given in figure ..
- **3: DT2D_V3** 2 dimensional division of the global matrix, following the communication pattern given in figure .., and also trying to automatically optimize the widths of the borders using extended timing data.

-profRate Controls how often to do extended timing. This extended timing is not as optimal, as described in section .., and should not be done very often. A number between ... and ... gives good results with the automatic optimizing. (not finished yet...)

-calcRate Controls how point in the matrix to calculate. A value of 1 means every point in the matrix is calculated. A value `n > 1` means every n'th point is calculated. This is useful when simulating large sparce matrices, or in conjunction with a calculation delay to simulate a more CPU bound application. Default value is 1.

-plot A value of 0 gives output more suited for humans to read. A value of 1 produces output suited for graphing with gnuplot or similar programs. Suitable gnuplot scripts are part of the program package.

-doRealCalc A value of 1 makes the code do real calculations. This can be used to make sure all memory is accessed. This is useful when one wants to watch the effects of cache. A value of 0 skips calculations. The delayCalc parameter can then be used to make sure each iteration uses the same amount of time.

-calcDelay The delay, in millisecond, to wait before starting communication after each series of iterations. (one series is `borderWidth` iterations). Default value is 0.

-commDelay The delay, in millisecond, to wait after each communication step. This means each communication step completes after a minimum of `commDelay` microseconds. Default value is 0.

-writeResult If > 0, write result matrix to output file "out.pgm".

Appendix C

Helper Programs

To make the benchmark program more usable, both during development and testing, and for future users, a number of extra utilities and scripts were written. These are described in the following sections.

C.1 Job scripts

Several job scripts are provided to help running the benchmark code times with different parameters. These job scripts can be used as a quick-start and should probably be modified to suit the batch system on the machine tested. The supplied job scripts are made to be run with the PBS and LSF scheduling systems. Separate scripts are made for running on 4, 8, 16, 64 and 256 nodes. This is done because the mentioned batch systems accept only one specification of required number of nodes per job script. When mixing 4- and 8-node jobs in one script, for example, the batch system will wait until 8 nodes are available, and then exclusively reserve these 8 even when the 4 node jobs are run. To run the tests, it is therefore better to have one master script to submit several jobs.

As an example, the PBS script which runs all border widths between 1 and 20, 3 times, will look like:

```
#!/bin/sh

#PBS -N Matrix4n1
#PBS -j eo
#PBS -l nodes=4
#PBS -q que

cd diplom/gridCpp

for i in 1 2 3
do
```

```
for j in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
do
comd="mpirun -np 4 -machinefile $PBS_NODEFILE ./main -dim 512 \
-b $j -iter 2000 -npx 2 -npy 2 -commP 3 -profRate 50 -plot 1"
$comd >> 4n2_run$i.dat
done
done
```

The output files are named after the following schema: `<numberofnodes>n<size-number>_run<runnumber>.dat`. The size-number is a predefined number. In this paper we defined 1=192x192 matrix, 2=512x512 matrix etc. Separate scripts are created for each matrix size.

C.2 Gnuplot scripts

Gnuplot is a free program, and it is used to create graphs from the execution time results. Gnuplot can be used either by writing the commands directly, or by first creating a script describing what to plot. A number of these gnuplot scripts were written while making the graphs for this paper. These scripts are included as part of the package to shorten the time from installation to seeing the first results.

As an example, the script for graphing all 6 runs of a 4-node test looks like:

```
set term postscript eps color "Helvetica" 18
set output '4n1.eps'
set xlabel "Border size"
set ylabel "Execution time"
set data style lines
plot "4n1_run1.dat" using 1:3 title "Run 1", \
"4n1_run2.dat" using 1:3 title "Run 2", \
"4n1_run3.dat" using 1:3 title "Run 3", \
"4n1_run4.dat" using 1:3 title "Run 4", \
"4n1_run5.dat" using 1:3 title "Run 5", \
"4n1_run6.dat" using 1:3 title "Run 6"
```

C.3 Average

The average program creates a file suitable for plotting with gnuplot from output files from several runs. The supplied job scripts does several runs with the same parameters. Average takes a number of result files, and writes a new file with the average values.

`Usage: average <in-file 1> <in-file 2> ... <in-file n> <out-file>`

C.4 Median

The median program creates a file suitable for plotting with gnuplot from output files from several runs. Median takes a number of result files, and writes a new file with the median values.

```
Usage: median <in-file 1> <in-file 2> ... <in-file n> <out-file>
```

C.5 Sort

The sort program is used when doing extended timing. Then, all the output is put into one file. To plot the extended timing data for each node, "sort" creates a result file for each node and puts the relevant data in each file.

```
Usage: sort <in-file> <number-of-nodes>
```

Appendix D

Code Examples

The bodies of the most relevant classes are included in the following sections.

D.1 Variables

The most important variables used are:

globalSize X and Y size of global matrix.

localSizeX X size of local matrix part.

localSizeY Y size of local matrix part.

border Size of overlapping area. Equals how many operations to do between each exchange of border values.

profileRate How often to do extended timing of communication.

commPattern Choose 1- or 2-dimensional division and communication pattern. The valid values are defined in the `Values.h` file.

calcRate Determines how often to calculate a point in the matrix. 1 means calculate every point, n means calculate every n'th point. This is useful when simulating large sparce matrices, or in conjunction with a calculation delay to simulate a more CPU bound application. Default value is 1.

calcDelay To adjust the time used by calculation, a delay can be added to each iteration step. This value is given in microseconds. Default value is 0.

commDelay The microsecond value of commDelay is the delay added to each communication step. This means, each communication step completes after a minimum of `commDelay` microseconds. Default value is 0.

These variables are controlled by the command-line parameters described in Section B.2

D.2 Main

```
#include "main.h"

int main(int argc, char** argv) {

  MPI_Init( &argc, &argv);
  values = new Values( argc, argv );
  if( values->errStatus() ) {
    programExit( values->errStatus() );
  }
  values->printParams();

  /* Initialize matrix. Which datatype to use depends on
     chosen communication pattern. */
  switch( values->commPattern ) {
  case DataTypes::DT1D_V1:
    values->print( "# Using 1D Matrix Ver1" );
    matrix = new Matrix1D1( values );
    break;
  case DataTypes::DT1D_V2:
    values->print( "# Using 1D Matrix Ver2" );
    matrix = new Matrix1D2( values );
    break;
  case DataTypes::DT2D_V1:
  case DataTypes::DT2D_V2:
  case DataTypes::DT2D_V3:
  case DataTypes::DT2D_V4:
    values->print( "# Using 2D Matrix Ver1" );
    matrix = new Matrix2D1( values );
    break;
  default:
    values->print( "# Unknown datatype\n" );
    programExit( values->WRONG_DATATYPE );
  }

  /* Do calculations */
  matrix->diffuse();

  /* Print time usage */
  matrix->printTotalTimeUsage();
  matrix->printTimeUsage();

  if( values->writeResult ) {
    /* Gather result on node 0 */
    matrix->collectGrid();

    /* Write to image file */
    matrix->writePGMFile();
  }

  /* Clean up and exit */
  programExit( 0 );
}


int programExit( int errorCode ) {
  if(values->rank == 0 && errorCode != 0 ) {
    std::cout << "Error " << errorCode << std::endl;
  }
  MPI_Finalize();
  delete matrix;
  delete values;
  exit( errorCode );
}
```

D.3 DataTypes2D3

```
#include <time.h>
#include "DataTypes2D3.h"
#include "Values.h"
#include "Matrix2D.h"

DataTypes2D3::DataTypes2D3( Values *v, Matrix2D *m ) :
  DataTypes2D( v, m ) {
  int otherCoords[2];
  numNeighbours = 0;
  /* All borders are initially equal */
  N.border = S.border = E.border = W.border = v->border;
  maxBorder = 20;
```

```
  minBorder = 1;
  setupDataTypes();

  otherCoords[0] = v->rankX - 1;
  otherCoords[1] = v->rankY;
  if( checkCoords( otherCoords ) ) {
    MPI_Cart_rank( m->comm, otherCoords, &W.rank );
    numNeighbours++;
  }
  else {
    W.rank = MPI_PROC_NULL;
  }

  otherCoords[0] = v->rankX + 1;
  otherCoords[1] = v->rankY;
  if( checkCoords( otherCoords ) ) {
    MPI_Cart_rank( m->comm, otherCoords, &E.rank );
    numNeighbours++;
  }
  else {
    E.rank = MPI_PROC_NULL;
  }

  otherCoords[0] = v->rankX;
  otherCoords[1] = v->rankY - 1;
  if( checkCoords( otherCoords ) ) {
    MPI_Cart_rank( m->comm, otherCoords, &N.rank );
    numNeighbours++;
  }
  else {
    N.rank = MPI_PROC_NULL;
  }

  otherCoords[0] = v->rankX;
  otherCoords[1] = v->rankY + 1;
  if( checkCoords( otherCoords ) ) {
    MPI_Cart_rank( m->comm, otherCoords, &S.rank );
    numNeighbours++;
  }
  else {
    S.rank = MPI_PROC_NULL;
  }
}


DataTypes2D3::~DataTypes2D3() {
}


/*
  Check if the coordinates in coords belong to a valid process in the
  process grid. Return true (1) if valid, false (0) if not.
*/
int
DataTypes2D3::checkCoords( int *coords ) {
  if( ( coords[0] < 0 ) || ( coords[0] >= v->sizeX ) ||
      ( coords[1] < 0 ) || ( coords[1] >= v->sizeY ) )
    return 0;
  return 1;
}


void
DataTypes2D3::exchangeData( int rank, int dRankY,
                            MPI_Datatype sendDataType,
                            MPI_Datatype recvDataType,
                            MPI_Request *sendReq,
                            MPI_Request *recvReq ) {
  MPI_Isend( m->mat, 1, sendDataType, rank,
             XCH_NORMAL, m->comm, sendReq );
  MPI_Irecv( m->mat, 1, recvDataType, rank,
             XCH_NORMAL, m->comm, recvReq );
}


/*
  Exchange border values with neighbours.
*/
void
DataTypes2D3::exchangeBorder() {
  MPI_Request req[16];
  MPI_Status  stat[16];

  exchangeData( N.rank, -1, N.send, N.recv, &req[0], &req[1] );
  exchangeData( S.rank, 1, S.send, S.recv, &req[2], &req[3] );
  exchangeData( W.rank, 0, W.send, W.recv, &req[4], &req[5] );
  exchangeData( E.rank, 0, E.send, E.recv, &req[6], &req[7] );
  MPI_Waitall( 8, req, stat );
```

```
  exchangeData( N.rank, -1, sendNW, recvSE, &req[0], &req[1] );
  exchangeData( N.rank, -1, sendNE, recvSW, &req[2], &req[3] );
  exchangeData( S.rank, 1, sendSW, recvNE, &req[4], &req[5] );
  exchangeData( S.rank, 1, sendSE, recvNW, &req[6], &req[7] );
  MPI_Waitall( 8, req, stat );
}


/*
  Exchange border values with neighbours.
  Take times for each border.
*/
void
DataTypes2D3::exchangeBorderTimed() {

  MPI_Request req[16];
  MPI_Status  stat[16];
  MPI_Status status;
  int sendCount;
  sendCount = 0;
  if( N.rank != -1 ) {
    RDTSC( N.timeCommStart);
    MPI_Isend( m->mat, 1, N.send, N.rank,
               XCH_TIMED, m->comm, &req[sendCount++] );
  }
  if( S.rank != -1 ) {
    RDTSC( S.timeCommStart);
    MPI_Isend( m->mat, 1, S.send, S.rank,
               XCH_TIMED, m->comm, &req[sendCount++] );
  }
  if( E.rank != -1 ) {
    RDTSC( E.timeCommStart);
    MPI_Isend( m->mat, 1, E.send, E.rank,
               XCH_TIMED, m->comm, &req[sendCount++] );
  }
  if( W.rank != -1 ) {
    RDTSC( W.timeCommStart);
    MPI_Isend( m->mat, 1, W.send, W.rank,
               XCH_TIMED, m->comm, &req[sendCount++] );
  }

  for( int i = 0; i < numNeighbours; i++ ) {
    MPI_Probe( MPI_ANY_SOURCE,  XCH_TIMED, m->comm, &status );
    if( status.MPI_SOURCE == N.rank ) {
      MPI_Recv( m->mat, 1, N.recv, N.rank,
                XCH_TIMED, m->comm, &stat[0] );
      RDTSC(N.timeCommEnd);
      }
    else if( status.MPI_SOURCE == S.rank ) {
      MPI_Recv( m->mat, 1, S.recv, S.rank,
                XCH_TIMED, m->comm, &stat[1] );
      RDTSC(S.timeCommEnd);
    }

    else if( status.MPI_SOURCE == E.rank ) {
      MPI_Recv( m->mat, 1, E.recv, E.rank,
                XCH_TIMED, m->comm, &stat[2] );
      RDTSC(E.timeCommEnd);
    }

    else if( status.MPI_SOURCE == W.rank ) {
      MPI_Recv( m->mat, 1, W.recv, W.rank,
                XCH_TIMED, m->comm, &stat[3] );
      RDTSC(W.timeCommEnd);
    }
    else {
      std::cout << "Other source " << status.MPI_SOURCE << "\n";
    }
  }

  timeComm = 0;
  if( N.rank != -1 ) {
    N.timeCommEnd -= N.timeCommStart;
    N.timeComm += N.timeCommEnd;
    timeComm += N.timeComm;
  }
  if( S.rank != -1 ) {
    S.timeCommEnd -= S.timeCommStart;
    S.timeComm += S.timeCommEnd;
    timeComm += S.timeComm;
  }
  if( E.rank != -1 ) {
    E.timeCommEnd -= E.timeCommStart;
    E.timeComm += E.timeCommEnd;
    timeComm += E.timeComm;
  }
  if( W.rank != -1 ) {
```

```
    W.timeCommEnd -= W.timeCommStart;
    W.timeComm += W.timeCommEnd;
    timeComm += W.timeComm;
  }
}


void
DataTypes2D3::printTimeUsage() {
  if( !( v->rank > 0 && v->onlyRootPrint > 0) ) {
    if( v->plot > 0 ) {
      if( v->border == 1 && v->rank == 0 ) {
        /* Assume first run in test script... */
        printf( "# Matrix 2D Datatypes V3. Size:%d Iter:%d Npx:%d Npy:%d ProfRate:%d\n",
                v->globalMatSize, v->iterations,
                v->sizeX, v->sizeY, v->profileRate );
        printf( "# border  rank  total_time  comm_time  calc_time  commN  commS  commE  commS\n" );
      }
      printf( "%d \t %d \t %f \t %f \t %f \t %6lld %6lld %6lld %6lld\n",
              v->border, v->rank, m->getTotalTimeUsage(),
              m->timeComm, m->timeCalc, N.timeComm, S.timeComm, E.timeComm, W.timeComm );
    }
    else {
      timeComm /= 100;
      printf( "Rank %d total comm:%6lld ", v->rank, timeComm );
      if( N.timeComm != 0 ) printf( "CommN:%9llu ", N.timeComm );
      if( timeComm != 0 ) printf( "(%2lld%%) ", N.timeComm/timeComm );
      if( S.timeComm != 0 ) printf( "CommS:%9llu ", S.timeComm );
      if( timeComm != 0 ) printf( "(%2lld%%) ", S.timeComm/timeComm );
      if( E.timeComm != 0 ) printf( "CommE:%9llu ", E.timeComm );
      if( timeComm != 0 ) printf( "(%2lld%%) ", E.timeComm/timeComm );
      if( W.timeComm != 0 ) printf( "CommW:%9llu ", W.timeComm );
      if( timeComm != 0 ) printf( "(%2lld%%) ", W.timeComm/timeComm );
      printf("\n");
    }
  }
}


/*
  Setup datatypes for alternative communication pattern
*/
void DataTypes2D3::setupDataTypes() {
  int border;
  int sizes[] =
    {
      0,0
    };
  int subsizes[] =
    {
      0,0
    };
  int starts[] =
    {
      0,0
    };

  /* Size of local grid, including border values */
  sizes[0] = m->matSizeX;
  sizes[1] = m->matSizeY;
  blockSizeX = m->blockSizeX;
  blockSizeY = m->blockSizeY;
  border = v->border;

  /* Corner is 1 point */
  subsizes[0] = 1;
  subsizes[1] = 1;

  /* Upper left send and receive corners */
  starts[0] = blockSizeX+2*border-1;
  starts[1] = 2*border-1;
  MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                            MPI_ORDER_FORTRAN, MPI_DOUBLE, &sendNW );
  starts[0] = 0;
  starts[1] = 0;
  MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                            MPI_ORDER_FORTRAN, MPI_DOUBLE, &recvNW );

  /* Upper right send and receive corners */
  starts[0] = 0;
  starts[1] = 2*border-1;
  MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                            MPI_ORDER_FORTRAN, MPI_DOUBLE, &sendNE );
  starts[0] = blockSizeX+2*border-1;
  starts[1] = 0;
  MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                            MPI_ORDER_FORTRAN, MPI_DOUBLE, &recvNE );
```

```
/* lower left send and receive corners */
starts[0] = blockSizeX+2*border-1;
starts[1] = blockSizeY;
MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                          MPI_ORDER_FORTRAN, MPI_DOUBLE, &sendSW );
starts[0] = 0;
starts[1] = blockSizeY+border;
MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                          MPI_ORDER_FORTRAN, MPI_DOUBLE, &recvSW );

/* lower right send and receive corners */
starts[0] = 0;
starts[1] = blockSizeY;
MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                          MPI_ORDER_FORTRAN, MPI_DOUBLE, &sendSE );
starts[0] = blockSizeX+border;
starts[1] = blockSizeY+border;
MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                          MPI_ORDER_FORTRAN, MPI_DOUBLE, &recvSE );

/* Horizontal line */
subsizes[0] = blockSizeX;
subsizes[1] = 1;

/* Start of upper horizontal line in grid */
starts[0] = border;
starts[1] = border;
MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                          MPI_ORDER_FORTRAN, MPI_DOUBLE, &N.send );

/* Start of upper horizontal neighbour region */
starts[0] = border;
starts[1] = 0;
MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                          MPI_ORDER_FORTRAN, MPI_DOUBLE, &N.recv );

/* Start of lower horizontal line in grid */
starts[0] = border;
starts[1] = blockSizeY;
MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                          MPI_ORDER_FORTRAN, MPI_DOUBLE, &S.send );

/* Start of lower horizontal neighbour region */
starts[0] = border;
starts[1] = blockSizeY+border;
MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                          MPI_ORDER_FORTRAN, MPI_DOUBLE, &S.recv );

/* Vertical line */
subsizes[0] = 1;
subsizes[1] = blockSizeY;

/* Start of left vertical line in grid */
starts[0] = border;
starts[1] = border;
MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                          MPI_ORDER_FORTRAN, MPI_DOUBLE, &W.send );

/* Start of left vertical neighbour region */
starts[0] = 0;
starts[1] = border;
MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                          MPI_ORDER_FORTRAN, MPI_DOUBLE, &W.recv );

/* Start of right vertical line in grid */
starts[0] = blockSizeX;
starts[1] = border;
MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                          MPI_ORDER_FORTRAN, MPI_DOUBLE, &E.send );

/* Start of right vertical neighbour region */
starts[0] = blockSizeX+border;
starts[1] = border;
MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                          MPI_ORDER_FORTRAN, MPI_DOUBLE, &E.recv );

/* Total size, including borders */
sizes[0] = m->matSizeX;
sizes[1] = m->matSizeY;

/* Size of local matrix, excluding borders */
subsizes[0] = blockSizeX;
subsizes[1] = blockSizeY;
starts[0] = border;
starts[1] = border;
/* Datatype used to send one part of the total matrix when
   calculations are finished*/
MPI_Type_create_subarray( 2, sizes, subsizes, starts,
```

```
                                MPI_ORDER_FORTRAN, MPI_DOUBLE, &sendMat );

  sizes[0] = v->globalMatSize;
  sizes[1] = v->globalMatSize;
  starts[0] = 0;
  starts[1] = 0;
  /* Datatype used to receive total, finished matrix on node 0 */
  MPI_Type_create_subarray( 2, sizes, subsizes, starts,
                                MPI_ORDER_FORTRAN, MPI_DOUBLE, &recvMat );

  /* Commit all created types */
  MPI_Type_commit( &sendNW );
  MPI_Type_commit( &recvNW );
  MPI_Type_commit( &sendNE );
  MPI_Type_commit( &recvNE );
  MPI_Type_commit( &sendSW );
  MPI_Type_commit( &recvSW );
  MPI_Type_commit( &sendSE );
  MPI_Type_commit( &recvSE );
  MPI_Type_commit( &N.send );
  MPI_Type_commit( &N.recv );
  MPI_Type_commit( &S.send );
  MPI_Type_commit( &S.recv );
  MPI_Type_commit( &E.send );
  MPI_Type_commit( &E.recv );
  MPI_Type_commit( &W.send );
  MPI_Type_commit( &W.recv );
  MPI_Type_commit( &sendMat );
  MPI_Type_commit( &recvMat );
}
```

D.4 Matrix2D1

```
#include "Matrix2D1.h"

#include "Values.h"
#include "DataTypes2D.h"
#include "DataTypes2D2.h"
#include "DataTypes2D3.h"
#include "DataTypes2D4.h"

Matrix2D1::Matrix2D1( Values *values ) : Matrix2D( values ) {
  mat = new double[ matSizeX * matSizeY ];
  memset( mat, 0, sizeof(double) * matSizeX * matSizeY );
  mask = new int[ matSizeX * matSizeY ];
  memset( mask, 0, sizeof(int) * matSizeX * matSizeY );
  if( v->rank == 0 ) {
    fullMat = new double [ globalMatSize * globalMatSize ];
  }
  initValues();
}


Matrix2D1::~Matrix2D1() {
  delete[] mat;
  delete[] mask;
  delete[] fullMat;
  delete dt;
}


void
Matrix2D1::diffusionStep( int x0, int y0, int x1, int y1, int count ) {
  int x;
  int y;
  int offset, offsetY;
  int border = v->border;
  int localCount = 0;
  for ( y = y0; y < y1; y++ ) {
    offsetY = matSizeX * ( y + border ) + border;
    for ( x = x0; x < x1; x++ ) {
      if( ( localCount++ ) == v->calcRate ) { //only do every calcRate'th calculation
        localCount = 0;

        /* Determine if the point is to be calculated. On each row,
           every 2nd point is calculated each time, as described in
           the documentation.
        */
        if ( ( ( x + y ) % 2 ) != ( ( count ) % 2 ) && ( getMask( x, y ) != 1 ) ) {
          offset = offsetY + x;
          mat[ offset ] = 0.25 * ( mat[ offset - matSizeX ] + mat[ offset + matSizeX ] +
                                   mat[ offset - 1 ] + mat[ offset + 1 ]);
```

```
        }
      }
    }
  }
}


/*
  Do the calculation on the local part of the matrix.
*/
void
Matrix2D1::diffuse() {
    int i, step;
    int profRuns = 0;
    int dummy = 0;
    unsigned int profileCount = 0;
    int iterations = v->iterations;
    int localIterations = v->localIterations;
    double timeCalcStart, timeCalcEnd;
    double timeCommStart, timeCommEnd;
    timeComm = timeCalc = 0;

    /* Find starttime */
    time_s = MPI_Wtime();

    for (i = 0; i < iterations / localIterations; i++) {

      /* Exchange border values with neighbours */
      timeCommStart = MPI_Wtime();
      if( v->profileRate != 0 && profileCount % v->profileRate == 0 ) {
        //        dt->exchangeBorderTimed();
        profRuns++;
      }
      else {
        dt->exchangeBorder();
      }
      profileCount++;
      timeCommEnd = timeCalcStart = MPI_Wtime();

      delayCalc();

      /* Do localIterations iteration between each exchange of values */
      for( step = localIterations-1; step >= 0; step-- ) {
        diffusionStep( -step, -step, blockSizeX+step, blockSizeY+step, dummy );
        dummy++;
      }

      timeCalcEnd = MPI_Wtime();
      timeCommEnd -= timeCommStart;
      timeCalcEnd -= timeCalcStart;
      timeComm += timeCommEnd;
      timeCalc += timeCalcEnd;
    }

    /* Do some more iterations if iterations is not divisible by localIterations */
    /*
    for( i = iterations % localIterations; i > 0; i--) {

      dt->exchangeBorder();
      diffusionStep( 0, 0, blockSizeX, blockSizeY, dummy );
      dummy++;
    }
    */
  /* Determine endtime */
  time_e = MPI_Wtime();
  time_e -= time_s;

  /* Find average time used */
  MPI_Reduce( &time_e, &time_s, 1, MPI_DOUBLE, MPI_SUM, 0, comm );
}


/* Print total time usage */
void
Matrix2D1::printTotalTimeUsage() {
  if( v->rank == 0 ) {
    if( v->plot == 0 ) {
      std::cout << "Total time used: " << getTotalTimeUsage() << std::endl;
    }
  }
}


/* Print detailed time usage */
void
Matrix2D1::printTimeUsage() {
  if( v->rank == 0 && v->plot == 0 ) {
    printf( "Rank %d Comm time: %f(%d) Calc time: %f(%d)\n",
```

```
            v->rank, timeComm, (int)(100*timeComm/getTotalTimeUsage()),
            timeCalc, (int)(100*timeCalc/getTotalTimeUsage()) );
  }
  dt->printTimeUsage();
}


double
Matrix2D1::getTotalTimeUsage() {
  return time_s / v->numNodes;
}

/*
  Initialize values on the boundary of the total system. See
  documentation for more details on this. Also set mask to avoid these
  values from being changed during calculations.
*/
void
Matrix2D1::initValues() {
  int i;

#ifdef SET_VALUE
  int j;
  printf("InitValues %d x %d\n", blockSizeX, blockSizeY );
  for( i = 0; i < blockSizeY; i++ ) {
    for( j = 0; j < blockSizeX; j++ ) {
      setLocal( j, i, (float)( rank * 32 + 64 ) );
    }
  }
#endif
  for( i = 0; i < blockSizeY; i++ ) {
    if( v->rankX == 0 ) {
      setMask( 0, i );
      setLocal( 0, i, (float)( i+ v->rankY * blockSizeY ) /
                ( v->globalMatSize - 1 ) );
    }
    if( v->rankX == v->sizeX-1 ) {
      setMask( blockSizeX - 1, i );
      setLocal( blockSizeX - 1, i, (float)1-(float)( i + v->rankY * blockSizeY )
                / ( v->globalMatSize - 1 ) );
    }
  }

  for( i = 0; i < blockSizeX; i++ ) {
    if( v->rankY == 0 ) {
      setMask( i, 0 );
      setLocal( i, 0, (float)( i + v->rankX * blockSizeX ) /
                ( v->globalMatSize - 1 ) );
    }
    if( v->rankY == v->sizeY-1 ) {
      setMask( i, blockSizeY - 1 );
      setLocal( i,  blockSizeY - 1, (float)1-(float)( i + v->rankX * blockSizeX ) /
                ( v->globalMatSize - 1 ) );
    }
  }
}

inline void
Matrix2D1::setLocal(int x, int y, double value) {
  mat[ matSizeX * ( y + v->border ) + x + v->border ] = value;
}

inline double
Matrix2D1::getLocal(int x, int y) {
  return mat[ matSizeX * ( y + v->border ) + x + v->border ];
}

inline void
Matrix2D1::setMask(int x, int y ) {
  mask[ matSizeX * ( y + v->border ) + x + v->border ] = 1;
}

inline int
Matrix2D1::getMask( int x, int y ) {
  return mask[ matSizeX * ( y + v->border ) + x + v->border ];
}


/*
   Gather the parts of the matrix on
   process 0.
*/
void
Matrix2D1::collectGrid() {
  int i;
  int recvOfs;
  int dims[2];
  MPI_Request request;
```

```
  MPI_Status status, status2;
  /* All processes send their part */
  MPI_Isend( mat, 1, dt->sendMat, 0, 2, comm, &request );

  /* Receive messages on node 0 */
  if( v->rank == 0 ) {
    for( i = 0; i < v->numNodes; i++ ) {

      /* Determine source of message */
      MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status );

      /* Use source rank to find position in 2D system */
      MPI_Cart_coords ( comm, status.MPI_SOURCE, 2, dims );

      /* Calculate where to put received part in total matrix */
      recvOfs = dims[0] * blockSizeX + dims[1] * v->globalMatSize * blockSizeY;

      /* Receive data */
      MPI_Recv( &fullMat[recvOfs], 1, dt->recvMat, status.MPI_SOURCE, 2, comm, &status2 );
    }
  }
}
```

D.5 Values

```
#ifndef VALUES_H
#define VALUES_H 1

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

/*
Contains program parameters and other common data.
*/
class Values {
 public:

  /* How many milliseconds to delay after each calculation step */
  unsigned int delayCalc;

  /* How many milliseconds to delay after each calculation step */
  unsigned int delayComm;

  /* If > 0, only root prints results to stdout */
  int onlyRootPrint;

  /* If > 0, result matrix is written to image file "out.pgm" */
  int writeResult;

  /* If percentage above average comm. time is higher than this value,
     try other border value */
  int adjustmentThreshold;

  /* Automatically adjust border around this value */
  int targetBorder;

  int doRealCalc;

  /* If > 0, output is printed in a format recognized by the plot
     scripts used for creating graphs of the result. If 0, output is
     printed in human-readable format */
  int plot;

  /* Number of processors in x direction */
  int sizeX;

  /* Number of processors in y direction */
  int sizeY;

  /* Size of grid in each direction */
  int globalMatSize;

  /* Number of iterations */
  int iterations;

  /* Number of local iterations between data exchange */
  int localIterations;

  /* Size of border */
```

```
  int border;

  /* X position of process in carthesian system*/
  int rankX;

  /* Y position of process in carthesian system*/
  int rankY;

  /* Whether parseCommandLine failed */
  int fail;

  /* Number of nodes used */
  int numNodes;

  /* Rank of process */
  int rank;

  /* Communication pattern chosen */
  int commPattern;

  /* How often to do profile sends */
  int profileRate;

  /* How often to calculate (every [calcRate]'th pixel) */
  int calcRate;

  /* Error messages used when exiting after an error */
  enum errormsgs { OK=0, WRONG_DATATYPE=1, NPX1DERROR=2 };

  Values( int argc, char **argv );
  int errStatus();
  int parseCommandLine( int argc, char **argv );
  int GetIntArg( int *argc, char **argv, char *switchName, int *val );
  int GetDoubleArg( int *argc, char **argv, char *switchName, double *val );
  int GetStringArg( int *argc, char **argv, char *switchName, char **val );
  int IsArgPresent( int *argc, char **argv, char *switchName );
  void print( char *str );
  void printParams();
};

#endif
```