NTNU – Trondheim
Norwegian University of
Science and Technology

Benchmarking Super Computers

Benchmarks of Clustis3 and Numascale

Elisabeth Solheim

Master of Science in Informatics
Submission date: Januar 2015
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem definition

Benchmark the two supercomputers Clustis3 and Numascale using a 2D heat equation.

Assignment given: 30. January 2013
Advisor: Anne C. Elster, IDI, NTNU

Abstract

In this thesis I will benchmark NTNUs cluster "Clustis 3" and "Numascale", two of IDI NTNUs super computers using the heat equation as a workload. The workload will be changed in size to see how the performance changes. The workload will also be run with different border thicknesses to change how it acts and how that affect the performance on the different computers. Discussion on how to interpret results and optimize node and process layouts can also be found.

Acknowledgment

First I would like to thank my fiancé and family for supporting me and helping me through this master thesis.

Also, many thanks to my supervisor Anne C. Elster for all her help.

Thanks to Malik Khan and Rune Jensen helping me with the super computers. An extra thanks to Malik for commenting on my thesis.

Contents

List of Figures

Chapter 1

Super computers

1.1 Why super computers exist?

Traditionally computation was done serially, one instruction at a time. In many situations this was and still is adequate. In example when doing simple algorithms on small datasets. However, scientific project and businesses often have complex algorithms with large datasets, taking forever to calculate serially on one CPU. Algorithms and data is often dependent on other data, however large parts of the data and algorithms may be independent, not demanding a specific order of execution, making it ideal to be executed at the same time [14].

By making multiple parallel processors cooperate you increase the throughput solving larger problem in a shorter time. Done right, the problem can be solved linearly faster divided on the number of processor, making a near linear speedup (see Chapter 1.3.1).

1.2 Performance limitations for computers

Parallel computing also is one way to solve the performance limitations processors have. For many years the CPU makers made PCs faster by increasing the clock rate, and therefor the number of calculations per second. However, around 2004 Intel hit the "Power Wall" (see Chapter 1.2.1) making it impractical to draw more calculation power out of a single CPU. Intel was forced to start making multi-core CPUs (CPUs with multiple cores).

1.2.1 Pattersons Three Walls

There are limitations in how high performance a computer can have. David Patterson called the limitations for walls[15]. Memory, instruction level parallelism

(ILP) and Power. All connected,so if an engineer optimizes one wall (limitation) he aggravates the other two walls. Together making Patterson's "brick wall": "Power Wall + Memory Wall + ILP Wall = Brick Wall" [7].

The Power Wall

The Power Wall is the limit where the clock rate for a single computer get so high that it get difficult to cool the processor. Either you have to use a material that can withstand higher temperatures or you have to separate the components making the heat. However separating components can make delays by the increased distance between the compontents.

Intel broke the power wall in 2004 with the Teja processor that was supposed to run at 7 Ghz, but never reached that speed, because the microprocessors got to hot and quit working[7]. They then had to change their approach, releasing multi-core processors. The first multi-core processor was a chip with two slower processors connected together.

The Memory Wall

The Memory Wall is the gap between the processor speed and the speed of the memory. That gap grows since the processor speed increases more rapidly than the memory speed. One solution to this problem was the computer cache. The cache temporarily stores copies of data from the most frequently used memory locations. But are of limited use for large data applications since only a part of the dataset can fit in the cache. Having a larger cache will increase both the physical size of the CPU and the power consumption[7].

The ILP wall

ILP (instruction level parallelism) means to run several instructions on different parts of the processor(functional units) at the same time to increase efficiency. Pipelining and multiple issue are the main approaches. In pipelining individual pieces of hardware or functional units processes in sequence. In multiple issue functional units are replicated to execute different instructions at the same time[14].

"ILP Wall means a deeper instruction pipeline really means digging a deeper power hole"[7]

1.3 Performance measurement

1.3.1 Speedup and efficiency

Speedup (S) is defined to be

$$S = \frac{T_{serial}}{T_{parallel}}$$

Where T_{serial} is the time used by the serial version of the program to run and $T_{parallel}$ is the time used by the parallel version on the program to run. The best speedup possible is linear Speedup. Where S = p and p is no of processes. That means that the time used to run the parallel version of the program is the time for the serial divided on the number of processors used.

$$T_{parallel} = \frac{T_{serial}}{p}$$

Efficiency of the parallel program tells how close to linear speedup the program is.

$$E = \frac{S}{p} = \frac{\frac{T_{serial}}{T_{parallel}}}{p} = \frac{T_{serial}}{p \times T_{parallel}}$$

1.3.2 Amdahl's law

Gene Amdahl observed in the 1960s that speedup is limited since not all of the serial program can be parallelized.

If $x\%$ of the serial program can be parallelized then runtime of the parallelized part with p processors will be $\frac{x}{100} \times \frac{T_{serial}}{p}$. The unparallelizable part will take $(1 - \frac{x}{100}) \times T_{serial}$. The speedup S will then be

$$S = \frac{T_{serial}}{\frac{x}{100} \times \frac{T_{serial}}{p} + (1 - \frac{x}{100}) \times T_{serial}} = \frac{1}{\frac{x}{100 \times p} + 1 - \frac{x}{100}}$$

$$\lim_{p \to \infty} \frac{1}{\frac{x}{100 \times p} + 1 - \frac{x}{100}} = \frac{1}{1 - \frac{x}{100}}$$

This means that when the parallelized part is 50 % then the speedup can be no bigger than 2. For x = 75 the max speedup is 4. For x = 90 it is 10 and for x = 95 it is 20.

This means that bigger the percentage that is parallelized, the bigger speedup. When the percent goes towards 100 the curve goes toward infinity as seen in Figure 1.2

This gives that very few problems will experience the larger speedups.

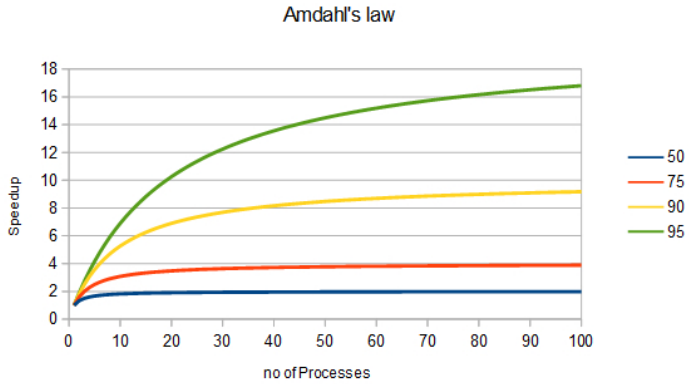

Figure 1.1: The speedup defined by Amdahl's law as the number of processes grow from 1 to 100

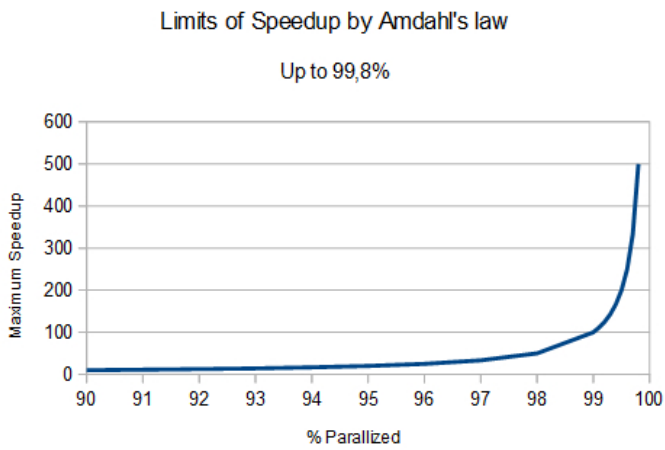

Figure 1.2: The limit of speedup by Amdahl's law from 90 to 99.8% parallelization

1.3.3 Gustafson's law

In 1988 Gustafson came with a reevaluation of Amdahl's law after experiencing several speedups bigger that what the Amdahl's law claimed he would get. [8]

Gustafson thinks that the parallel or vector part scales with the problem size. For example by doubling the number of processors when adding degrees of freedom in a physical simulation. In Amdahl's law the problem is fixed, and the parallel part time is dependent on the number of processors used.

Gustafson's law:
S= Speedup
T_{serial} = Serial time spent on the parallel system
$T_{parallel}$ = Parallel time spent on the parallel system
p = number of processors

$$\begin{aligned} S &= (T_{serial} + T_{parallel} \times p)/(T_{Serial} + T_{parallel}) \\ &= T_{serial} + T_{parallel} \times p \\ &= p + (1-p) \times T_{serial} \end{aligned}$$

This speedup is linear and with different percentages of T_{Serial} the graph look like:

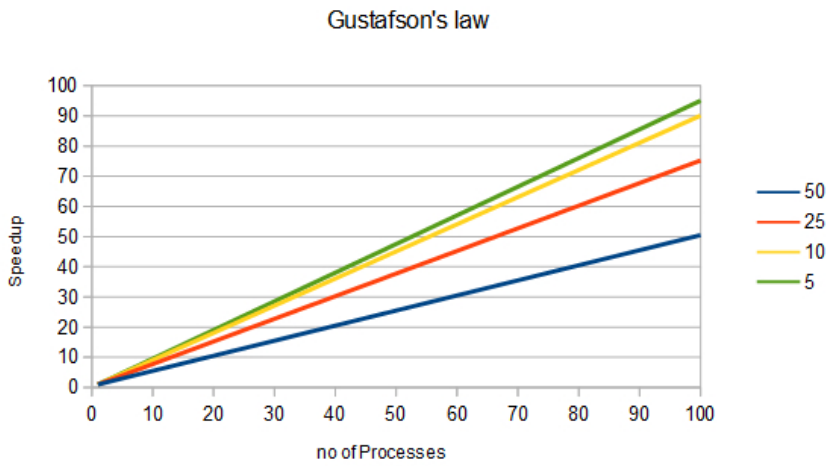

Figure 1.3: The speedup defined by Gustafson's law as the number of processes grow from 1 to 100 for different percentages of T_{Serial}

$$\lim_{p \to \infty} p + (1-p) \times T_{serial} = \infty$$

The Speedup has no limitations by Gustafson's law.

1.4 Flynn's taxonomy

Flynn's taxonomy is a way to classify computer architectures.[14] Flynn divides the computer architectures into 4 classifications.

1.4.1 SISD

SISD, or Single Instruction, Single Data stream, is a sequential computer with no parallelism. The architecture does one single type of instruction on one single data at a time. One example of SISD is add 1 to variable A.

1.4.2 SIMD

SIMD (Single Instruction, Multiple Data stream) is the second architecture with the simplest type of parallelism. With one single type of instruction run on multiple data at a time. One example of SIMD is add 1 to variable A, B and C.

1.4.3 MISD

MISD (Multiple Instruction, Single data) is a less common architecture, applying multiple instructions to one single data at a time. One example of MISD is to add 1 to variable A, Subtract 3 to variable A, Multiply variable A with 4. All at the same time.

1.4.4 MIMD

The last of Flynn's architectures is MIMD (Multiple Instructions, Multiple Data). This is now the most common architecture, making it possible to run multiple instructions on different multiple data. This is the typical multi core super scale system.

There are two types of MIMD systems, shared-memory and distributed-memory systems.

Shared-memory systems

In a shared-memory system each processor can access any memory location through the interconnect.

Distributed-memory systems

In a distributed-memory system each processor has its own private memory. All the processor-memory must be paired through messaging like MPI(see Chapter 1.6).

1.5 Software categorizations

Flynn's taxonomy did not fit perfect with the real world, requiring two new sub-category of MIMD: SPMD and MPMD[14].

1.5.1 SPMD

Single Program, Multiple Data is a software category where a single program runs on multiple data. This single program behaves different on what data it gets. MPI(Chapter 1.6) is typically SPMD, running the same program on different data. An example of an SPMD code is timed_heat found in Appendix C.1.

1.5.2 MPMD

The last software category is Multiple Program, Multiple Data which is multiple programs on multiple data.

MPI has the possibility to run MPMD, but SPMD is the most used way to run MPI[12].

1.6 Message Passing Interface (MPI)

MPI, or Message Passing Interface is a API for message passing on distributed memory systems. [14, p. 83][13]

In this thesis MPI is used to make it possible for different processes to talk between them, making it possible to divide the problem into parallel parts.

MPI is available for multiple programming languages, but here it is used with C. An example of MPI code is shown in Chapter C.1.

A common problem when dividing problems into a grid is when some of the needed information is in the neighbor grid. Luckily this is quite easy in MPI.

MPI uses the SPMD (single program multiple data, Chapter 1.5.1) approach to parallel programming.[14, p. 83]

1.6.1 Send and Receive

All communication between nodes are done through messages. There are a number of different send and receive methods for different purposes. MPI_Send, MPI_Recv, MPI_Sendrecv and MPI_Isend are the ones used by the timed_heat code in Chapter C.1.

1.6.2 Functionality

MPI does not try to solve every parallel programming problem, but rather only the problem of synchronizing data.[13]

1.6.3 Structure

MPI is just a library. However it requires some structure. MPI calls should only be written between the MPI_Init and the MPI_Finalize calls. Also the mpi.h file has to be included. All of the functions, types, macros and constants in MPI start with MPI_ and a capital letter after the underscore, and the whole name in capital letters for the macros and constants. This makes it easier to differentiate between user and MPI stuff.

After the initialization MPI_Comm_size and MPI_Comm_rank are called to get the size and rank for the processes. Size is the total number of processes and rank is the process number that are unique for each process and is a number from 0 to size-1. The rank is the identification of each process. The rank and size is used for messaging purposes.

1.6.4 Master / slave

When dividing data between nodes (processes) in MPI, it is common to use the master / slave strategy, with one master node that controls the data flow. The master divides the original problem into multiple data grids and sends one grid to each slave.

A slave can talk to another slave, but it is that master who receives the end result and stitches it all together.

1.6.5 Common pitfalls

A common pitfall in MPI is that every send and receive must match[14], or the process will hang, waiting for a matching message. Matching means that the parameters like datatype, tag and rank numbers(source and dest) must be identical. The MPI_Send method must also be posted before the MPI_Recv method which can lead to cyclic dependencies. There are several solutions to solve this problem like different sending modes and the MPI_Sendrecv method.

Chapter 2

Benchmarking

Benchmarking is to assess the relative performance of an object. It can be used to tell if a program is effective. Benchmarking has nothing to do with the correctness of a program, only how well it performs.

A example of benchmarking is FLOPs of CPU (floating-point operation per second). It tells how many linear algebra problems a supercomputer can solve in a second.

The difference in architecture and complexity of modern CPUs and compilers makes it hard unpredictable and hard to write useful benchmarks.

Benchmarking efforts 3 inter-related elements: workload, metrics and methodology. Workload is the application or benchmark software for testing the HPC system. Metrics is the basis for comparison that are used as a measure. Methodology is the system of methods that make out the measurement procedure [17].

2.1 Workload Efforts

2.1.1 the High-Performance Linpack Benchmark

The first "LINPACK Benchmark" appeared in the appendix of th LINPACK Users Guide in 1979. It was originally designed for the users of the LINPACK package for estimating execution time. The Workload was a single 100 by 100 system of linear equations of the form

$$Y(I) = Y(I) + T * X(I)$$

This was done on a 23 of the most used computers. They used a 75 by 75 system for the computers that was not big enough to handle a 100 by 100 system and used extrapolation to obtain the results. [10] This Benchmark is called LINPACK 100.

Due to the use of scalable computers with distributed memory in the HPC field they made the Highly-Performance Linpack NxN Benchmark. [4] HPL is the portable implementation of the benchmark. It generate and solves a random dense linear system

$$Ax = b; \qquad A \in R^{n \times n}; \;\; x, b \in R^n$$

HPL then use first LU factorization then backward elimination to solve the system before it checks for correctness of the solution.

The performance given by HPL is not reflect the overall performance of the given system but reflect the performance of solving dense systems of linear equations on the system. [18] Linear equations is a regular problem for hpc.

The Top500 List

The Top500 supercomputers list ranks all commercial accessible supercomputers using HPL as a measure. [18]

The measurements for each of the supercomputers in list are:
Rmax - Maximal LINPACK performance achieved
Rpeak - Theoretical peak performance
Nmax - Problem size for achieving Rmax
N1/2 - Problem size for achieving half of Rmax

The NTNU computer Vilje scored 82th place on this list November 2013.

The Green500 List

The Green500 list ranks the most energy-efficient supercomputers in MegaFLOPS/Watt using TOP500 performance results and wattage use given from the manufacture.

Vilje scored 105th place on this list in November 2013, and 154th place in June 2014 with 738.73 MFLOPS/watt and total power of 537 kW.

2.1.2 GigaTEPS

GigaTEPS was developed as a counterweight to FLOPS.

FLOPS(floating-point operation per second) are raw number-crunching power and tell how many linear algebra problems a supercomputer can solve in a second. GigaTEPS(billions of traversed edges per second) tells how fast the computer can search in large datasets. An Edge is a connection between two data points. An example of two data points is how many that buys book number 1 also buy book number 2.

The hope is that GigaTEPS will spur both researchers and industry toward mastering architectural problems to develop the next generation supercomputers.

Current cluster implementation suffer from high latency data communication witch leads to inefficiency in performance and energy consumption. Scaling graph traversal to multi-node cluster is challenging, w‘seq 1 20‘;hich has led to the creation of alternative metric of supercomputer performance like the Graph500. To achieve better GigaTEPS memory accessibility for CPUs is important since big machines with a high FLOP result gets bad GigaTEPS results. [16].

Graph 500 Benchmark

The Graph 500 benchmark was created by Richard Murphy at the Sandia National Laboratory. The Graph500 uses Breadth-First Search for their Benchmark [16].

The creator of Linpack Jack Dongarra has said that the Graph500 may add to the list of metrics for rating supercomputers but it can not be seen as a definitive number of performance any more than the Linpack is today [1].

HPC Challenge

HPC Challenge is a new benchmark-group that test both computing and widespread memory accessibility. HPC Challenge is under The Defense Advanced Research Projects Agency, the U.S. Department of Energy, and the U.S. National Science

SPEC

SPEC set of computing benchmarks(aimed at better measuring the performance of more everyday components like Web servers) Standard Performance Evaluation Corp. [17]

Green computing

Energy-Aware Big Data Computing is becoming more important [2]. One of the largest problems with computer farms in the heat they produce and have to went away to avoid melting.

Chapter 3

Heat Equation by FTCS

The reason for using the FTCS heat equation in this thesis is that I knew this algorithm well from an exercise in the course "TDT4240 - Paralell computing". Knowing that it fit well to run on a super computer and has its practical uses in the real world.

3.1 Heat equation

The heat equation describes the distribution of heat in a given region over time. It is a partial differential equation that is a equation with one or more partial derivatives of an function u [11, p. 535]. Where u is a function that describes the temperature. The heat equation in two dimensions can be seen as a cut of the three dimensional space.

The two dimensional heat equation is[9]:

$$u_t = c\left(u_{xx} + u_{yy}\right), 0 \leq x, y \leq 1, t \geq 0 \tag{3.1}$$

3.1.1 Boundary conditions

For the heat equation the boundary conditions describe the heat on the edges. There are three types of boundary conditions or so called Boundary Value problems for partial differential equations [11, p. 558]:

Dirichlet Problem u is prescribed on C (boundary) meaning that f(x) is a known function on the boundary.

Neumann Problem $u_n = \partial u/\partial n$ meaning that f'(x) is a known function on the boundary.

Mixed Boundary Value Problem or Robin Problem if u is prescribed on a portion of C and u_n on the rest of C(boundary)

3.1.2 Calculate the constant c

In heat transfer the constant c is $alpha$ that is the thermal diffusivity[5]:

$$\alpha = \frac{k}{\rho c_p} \tag{3.2}$$

where

- α is the thermal diffusivity (the rate at which heat diffuces through a body) measured in $\frac{meter^2}{seconds}$
- k is the thermal conductivity that describes the rate at which heat flows within a body for a given temperature difference measured in $\frac{Watt}{meterKelvin}$
- c_p is the specific heat capasity that is the amount of energy a body stores for each degree increase in temperature measured in $\frac{kJ}{kgKelvin}$
- and ρ is the density tha amount of mass per unit volume measured in $\frac{kg}{meter^3}$

3.2 Numerical solution the heat equation by FTCS

Forward Time and Central Space(FTCS) is a Finite Difference Method for solving partial differential equations numerically. To solve a equation numerically means to approximate it.

$u(t, x, y)$ can be approximated by replacing any derivative by finite differences. Then for any discreet points (t_k, x_i, y_j):

$$u_t \approx \frac{u_{i,j}^{k+1} - u_{i,j}^k}{\Delta t} \tag{3.3}$$

and

$$u_{xx} \approx \frac{u_{i+1,j}^k - 2u_{i,j}^k + u_{i-1,j}^k}{(\Delta s)^2}, u_{yy} \approx \frac{u_{i,j+1}^k - 2u_{i,j}^k + u_{i,j-1}^k}{(\Delta s)^2}, \tag{3.4}$$

Then the heat equation $u_t = c\,(u_{xx} + u_{yy})$ becomes:

$$\frac{u_{i,j}^{k+1} - u_{i,j}^k}{\Delta t} = c\left(\frac{u_{i+1,j}^k - 2u_{i,j}^k + u_{i-1,j}^k}{(\Delta s)^2} + \frac{u_{i,j+1}^k - 2u_{i,j}^k + u_{i,j-1}^k}{(\Delta s)^2}\right) \tag{3.5}$$

witch gives:

$$u_{i,j}^{k+1} = u_{i,j}^k + c \cdot \frac{\Delta t}{(\Delta s)^2}\left(u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k - 4u_{i,j}^k\right) \tag{3.6}$$

Where the spatial mesh points are:
$(x_i, y_j) = (i \cdot \Delta s, j \cdot \Delta s), i, j = 0, 1, 2..., n+1$ where $\Delta s = \frac{1}{n+1}$
Where the temporal mesh points are:
$t_k = k \cdot \Delta t, k = 0, 1, 2, ...,$ for suitably chosen Δt
Where $\Delta t \leq \frac{(\Delta s)^2}{2c}$ for the solution to be stable. [9]

3.3 Implement the numerical solution for a single processor

The serial version is straight forward from Equation 3.6 [9] The serial version is basically to first initialize the u_k+1 and u_k matrixes and then for all i and $j \in [1, n]$ $u\_k + 1 = u\_k + c \times \frac{\Delta t}{(\Delta s)^2} \times (u\_k[i+1, j] + u\_k[i-1, j] + u\_k[i, j+1] + u\_k[i, j-1] - 4 \times u\_k[i, j])$. The pseudocode can be seen in Appendix 1.

$u\_k[i+1, j] + u\_k[i-1, j] + u\_k[i, j+1] + u\_k[i, j-1] - 4 \times u\_k[i, j]$ can be seen as a stencil that is applied along the u_k matrix. The stencil is seen in Figure 3.1.

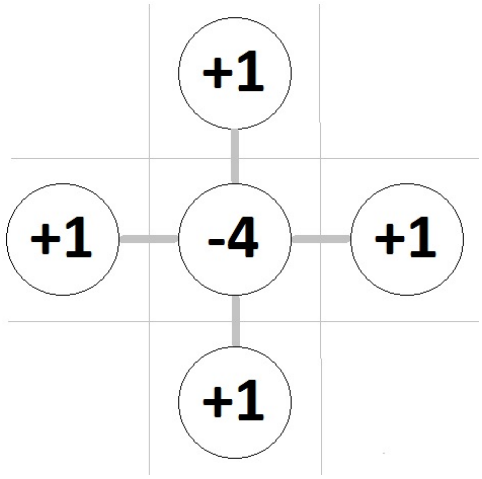

Figure 3.1: The stencil

The border condition must be implemented as well. But since there are three types of border conditions there are different ways the border condition must be implemented.

3.3.1 Dirichlet Problem

This is easy to implement. The matrix is allocated n+2 by n+2 such that it has a border of thickness 1 around itself where the values of the given function is stored.

3.3.2 Neumann Problem

For the Neumann Problem the relation between the values are calculated for each step. For example the code "timed_heat" in Appendix C.1 is an implementation with the Neumann Problem that results in a perfect insulated border the heat is mirrored such that one arm in the opposite side of the stencil in Figure 3.1 is used twice instead of the one on the outside of the matrix.

Chapter 4

Heat Equation solution in parallel

4.1 The timed_heat code

The code found in Appendix C.1 simulates a heat equation solution using the FTCS(Forward-Time Central-Space) method. The simulated system is a piece of copper, tin and aluminum emerged in mercury as shown in Figure 4.1. At the beginning, the copper and tin is at 60 degrees Celsius, the mercury is at 20 degrees Celsius, while the Aluminum is at a 100 degrees Celsius.

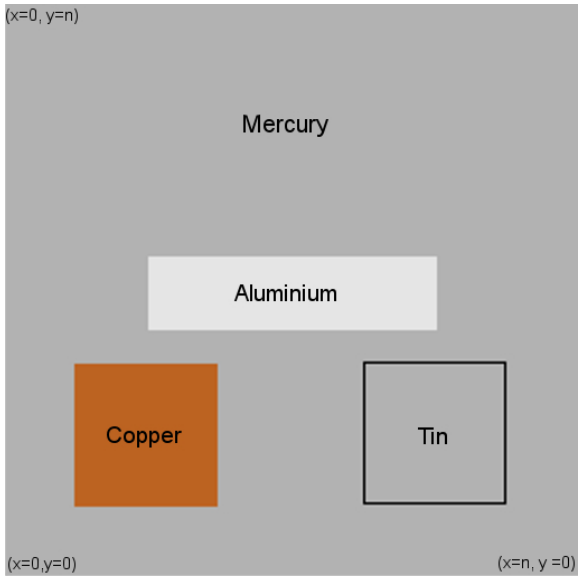

Figure 4.1: The heat system

The system consists of a n times n floats matrix. Each float uses its neighbors to calculate the next step with by applying a stencil. There are 125 000 steps in total.

Snapshots are taken every 160 step. The aluminum is kept at the same temperature at first part of the simulation, and is turned of at step 75000.

The n by n floats are split into smaller areas that are computed by one process or rank. The local area has a border surrounding it that are used for storing data sent from the neighbor. The neighbors send the borders to each other within the border exchange function.

4.1.1 Global variables

- **SIZE** is number of floats. The system is nxn floats big where n is SIZE.
- **WRITETOFILE** is if the program should write to file or not
- **NSTEPS** is number of steps. I have used 125000 for the benchmarking
- **CUTOFF** is when to cut of the heat. Meaning when to stop setting the aluminum to 100 degrees. I have used 75000 for the benchmarking
- **BORDER** is the thickness of the border
- **temperature** is the matrix that rank 0 uses to store all temperature data in before writing to file
- **local_material** is the local matrix for storing the material constants
- **local_temp** is the local matrices for storing temperatures. Swaps between two for odd and even numbered steps.

4.2 Methods

4.2.1 Main method

The main method is the core of the program. All other methods are called from the main method. Here the MPI is initialized and finalized, time measures is taken, the heat system is set up and the steps are executed.

Initialization

As explained in Chapter 1.6.3 every MPI needs to have a MPI_Init and MPI_Finalize call. Initialization in timed_heat[C.1] also includes finding rank and process count.

Splitting the workload

Splitting the workload is done by creating a layout of processes by creating dimensions out of the number of processes that are used. Different layouts can be seen in Chapter 4.4.

The dimensions are then set up using MPI_Dims_create before cart communication is set up. The dimensions are then used to calculate the local dimensions(local_dims) by dividing SIZE on the dimensions. If the SIZE is not equally divided on the dimension not all of the numbers will be calculated or sent to rank 0 for writing to file. This error is shown in Chapter 4.3.2.

Solving the non-evenly dividable size problem

One solution to the problem of size being non-evenly divided is to pad the local_dims in the direction that the problem occur such that all the local areas are the same size. This makes it possible for all the processes to use the same datatype for sending their areas to rank 0 for collection data. This also makes the load balance as equal as possible for small grid x and y values but do not fully utilize all the threads for larger dimensions like 31 times 1. The result is sent to rank 0 that only write the first SIZE numbers in each row and the first SIZE rows to file.

Allocate space

When the space needed for local matrices and the temperature matrix are found the space for them are allocated. The local matrices are initialized right before the vector types for border exchange is created and committed.

Execute the steps

The steps are iterated NSTEPS time and consist of:

- If the step is smaller than the CUTOFF constant the external_heat method are called resulting that the area of the aluminum are set to 100 °Celsius
- The border_exchange method(Chapter 4.2.2) is called on every nth step, where n is BORDER.
- On the remaining steps the update_border method(Chapter 4.2.4) is called.
- The ftcs_solver method are then called(see Chapter 4.2.3).
- The boundaries method are called(see Chapter 4.2.5)
- The Filename is created and the collect_area method is called for every nth step, where n is SNAPSHOT.

The time measure are then ended before the space matrices are freed up, the MPI_Finalize method are called and the result are printed to screen or logged.

4.2.2 Border exchange

In the border_exchange method the content in the local data of BORDER size of one rank is sent to another ranks border so that rank can use the border to compute their local areas.

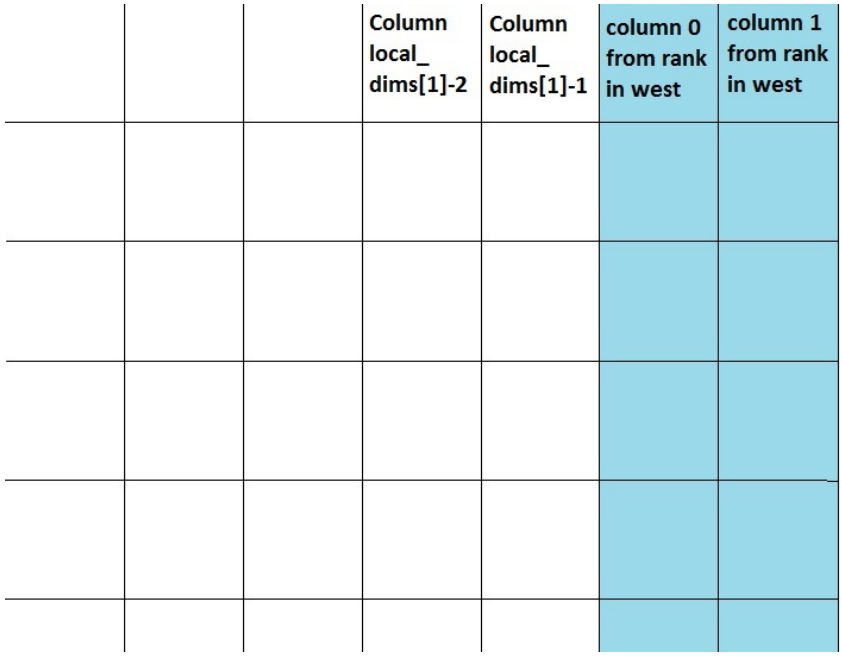

Figure 4.2: East border with border thickness 2 after calling border_exchange.

4.2.3 FTCS solver

The FTCS solver method takes the step as an argument and calculates the local temperature matrix for the next step. This is the same equation as Equation 3.6 in Chapter 3.2 where $local\_temp[step+1][y][x] = u_{x,y}^{k+1}$.

$$\begin{aligned} local\_temp[step+1][y][x] = local\_temp[step][y][x] + local\_material[y][x] \times \\ (local\_temp[step][y-1][x] + local\_temp[step][y+1][x] + \\ local\_temp[step][y][x-1] + local\_temp[step][y][x+1] \\ -4 * local\_temp[step][y][x]) \end{aligned} \tag{4.1}$$

4.2.4 Border update

The border update is a method for applying the FTCS stencil in the borders instead of using the border update.

The border updated is called on steps that has a rest-value of 1 to BORDER-1 when divided on BORDER. For step with rest-value of 1 the BORDER-1 rows or columns closest to the local area is calculated. For step with rest-value 2 the BORDER-2 rows or columns closest the local area are calculated... For step with rest-value BORDER-1 the row or column closest to the local area are calculated. This is an optimization since it is the column or row closest to the local area that are needed for the FTCS solver but that takes its values from the column that are one column or row further from the local area.

4.2.5 Boundaries

The boundaries method implements the Neumann boundary condition.

4.3 Visual results

Visual result are very useful for error-checking. When you know how it should look like it is easy to see if something is not correctly computed.

4.3.1 Correct output for size 256

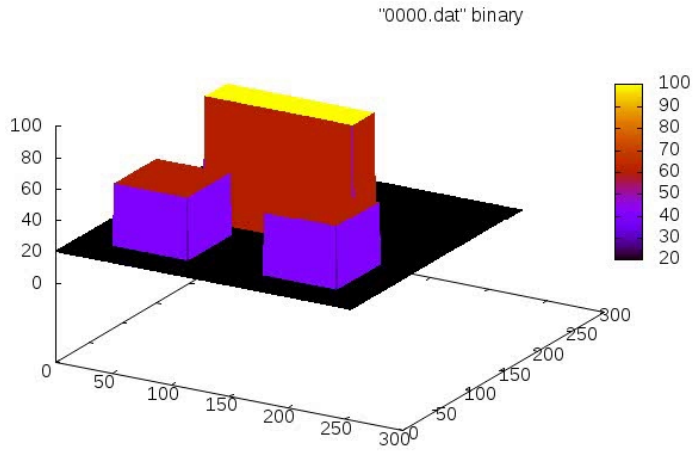

Figure 4.3: The heat system at step 0: Mercury is the black field. The copper is at the left and the tin is at the right. The Aluminum is at the back.

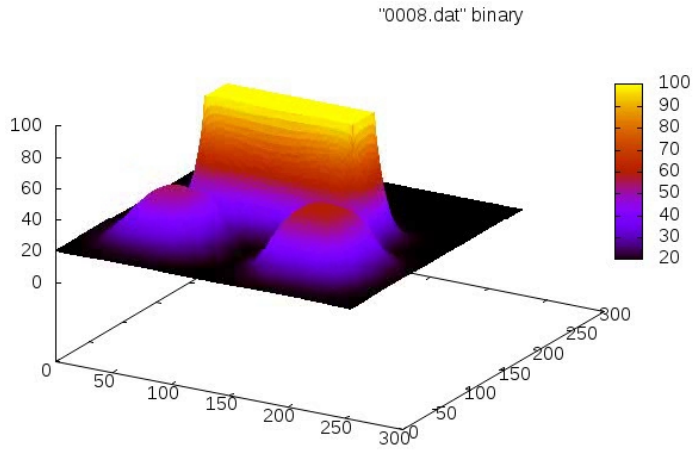

Figure 4.4: The heat system at step 1280: Here you can see that the copper is a better heat conductor than tin. The Aluminum is kept at 100 degrees Celsius. The mercury is receiving heat from the other metals.

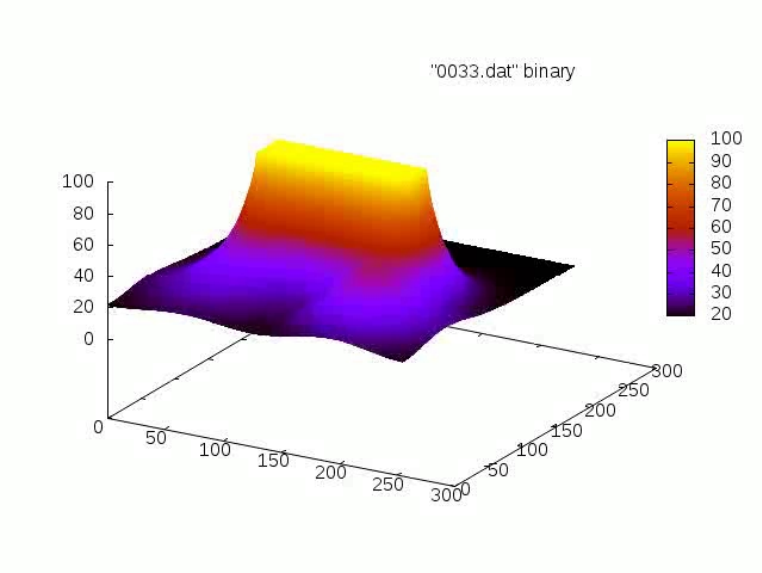

Figure 4.5: The heat system at step 5280: Here is the temperature for the copper and tin almost the same as for the mercury.

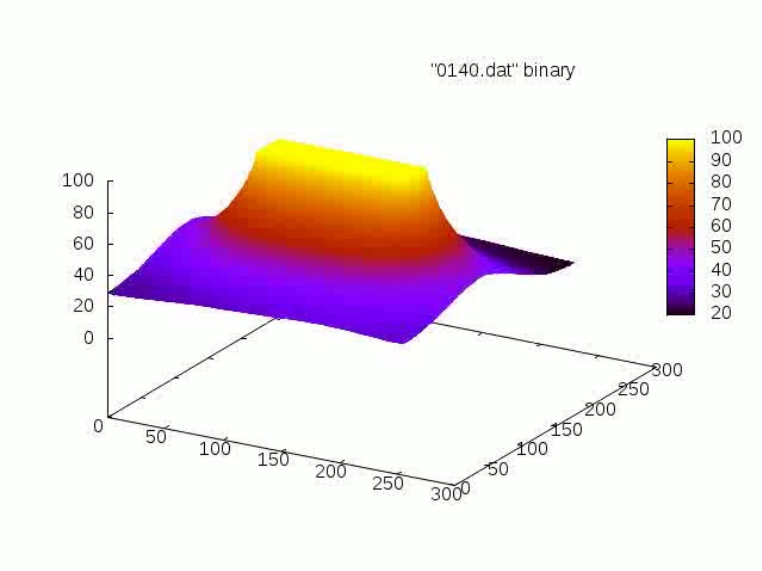

Figure 4.6: The heat system at step 22400: The aluminum is still kept at 100 degrees Celsius.

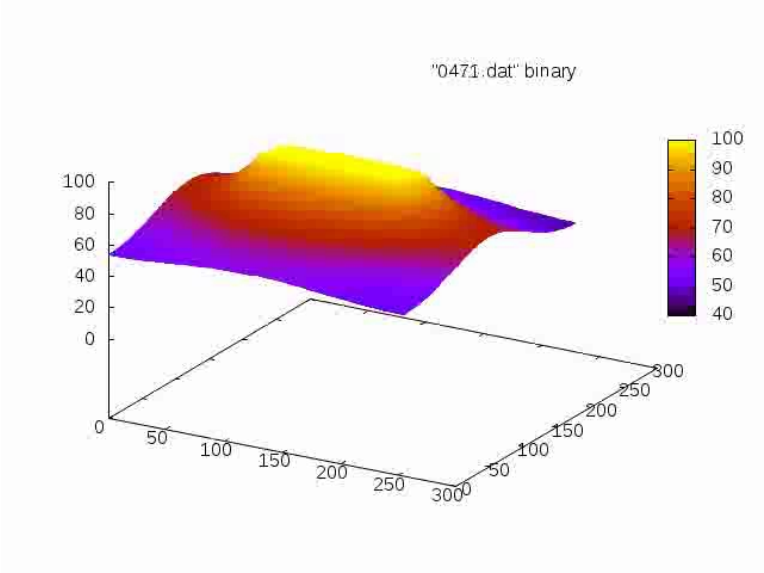

Figure 4.7: The heat system at step 75360: Here the aluminum is now longer kept at 100 degrees celsius

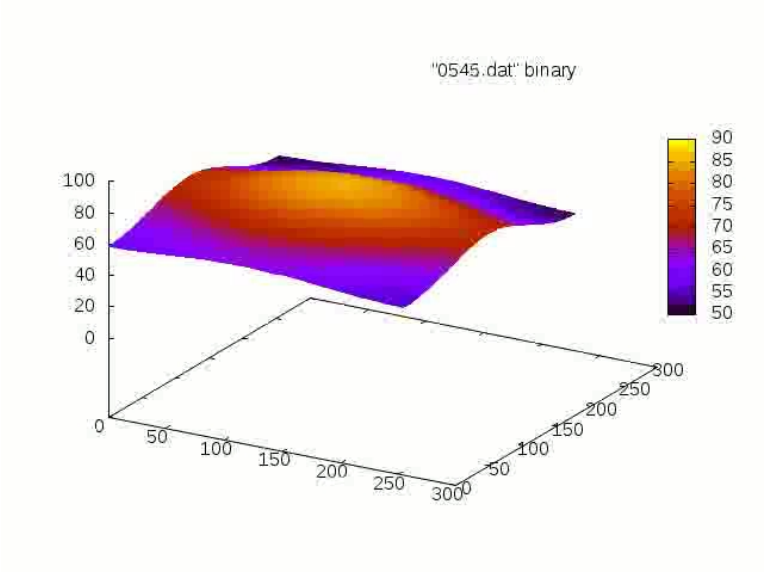

Figure 4.8: The heat system at step 87200: The system has almost found equilibrium.

4.3.2 Error examples

Here are some errors, and how the visualization gives clue of what is wrong. The sizes 256, 512 and 1024 are not evenly divided into the grid sizes(see Chapter 4.4 Figure 4.12). This can be seen in Figure 4.9, where not all the rows or columns are computed and sent to rank. Here you can see how the problem looks for 3 processes when only the y directions is affected. In figure 4.10 you see both y and x direction is affected. The local area is shifted one left for each line and there is zero-values in the end of where the areas should go. Changing the number of processor used will often change how data is distributed, giving some headache if you haven't though of all the corner cases.

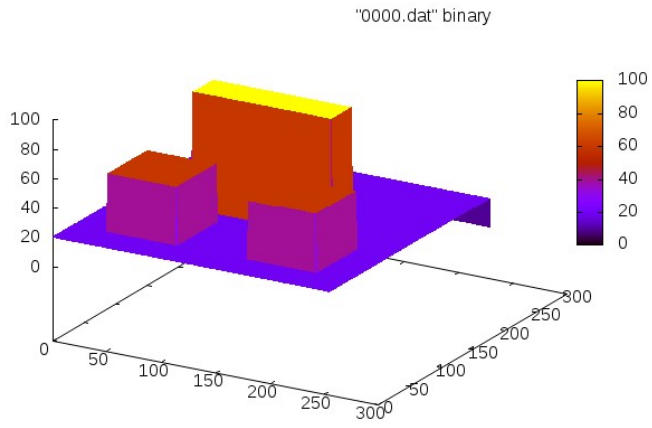

Figure 4.9: Row 256 not computed or sent to rank 0 for writing to file for 3 processes. See the sudden drop in the back

In Figure 4.11 there is a problem with update_border method resulting in valleys in the heat-distribution, where the borders are. This error was a result of using the wrong local_temp buffers. The stencil was applied for local_temp[step+1] using local_temp[step] when the correct buffers are to apply the stencil to local_temp[step] using the data in local_temp[step-1]. Since the border thickness was 2, the border_update method overwrote the results in the next step, decreasing the shared border temperatures rapidly.

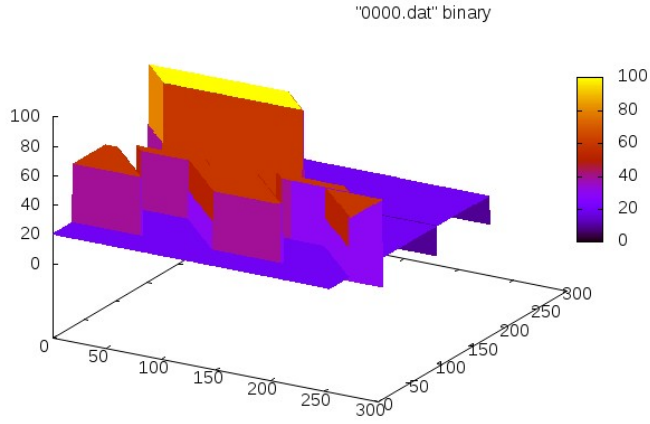

Figure 4.10: Row 256 and column 256 not computed or sent to rank 0 for writing to file for 9 processes

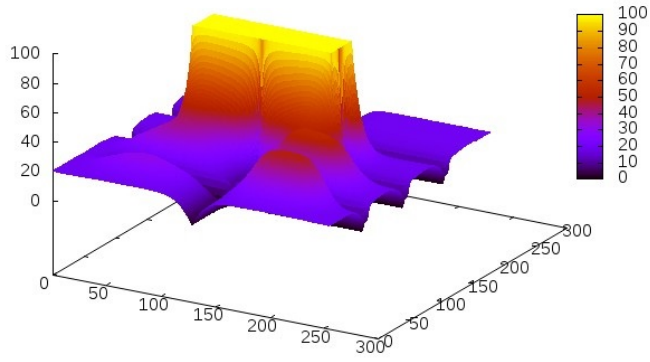

Figure 4.11: Error in the update_border method for step no. 1280, size 256 and border thickness 2.

4.4 Process layout

MPI are used to set up the dimensions using MPI_Dims_create as explained in Chapter A.3.2. The Cartesian grid in the code is two-dimensional. The grids for no of processes from 1 to 50 can be seen in Figure 4.12. Here you can see that the processes are grouped as closely together as possible and still make an rectangle. Therefore are grids with no processes that are prime numbers only 1 process wide.

no. of Processes	y	x	no. of processes	y	x
1	1	1	26	13	2
2	2	1	27	9	3
3	3	1	28	7	4
4	2	2	29	29	1
5	5	1	30	6	5
6	3	2	31	31	1
7	7	1	32	8	4
8	4	2	33	11	3
9	3	3	34	17	2
10	5	2	35	7	5
11	11	1	36	6	6
12	4	3	37	37	1
13	17	1	38	19	2
14	7	2	39	13	3
15	5	3	40	8	5
16	4	4	41	41	1
17	17	1	42	7	6
18	6	3	43	43	1
19	19	1	44	11	4
20	5	4	45	9	5
21	7	3	46	23	2
22	11	1	47	47	1
23	23	1	48	8	6
24	6	4	49	7	7
25	5	5	50	10	5

Figure 4.12: Process layout for 1 to 50 processes with MPI where y is height and x is width

In Figure 4.13 you can see a layout for 3 processes. The grid chosen for the 3 processes is a 3 by 1 grid. For 3 processes all the processes have boundaries in the west and east with coords[1] = dims[1]-1 = 1-1 = 0.
Rank 0 have a north border with coords[0] = 0 and rank 2 have a south border with coords[1] = dims[0] - 1 = 3-1 = 2 This means that borders are sent over north and south borders. For a 256 times 256 system with border thickness 1 each local_temp is 258 x 87 (SIZE + 2*BORDER)x(SIZE/3 + 1 +2*BORDER).

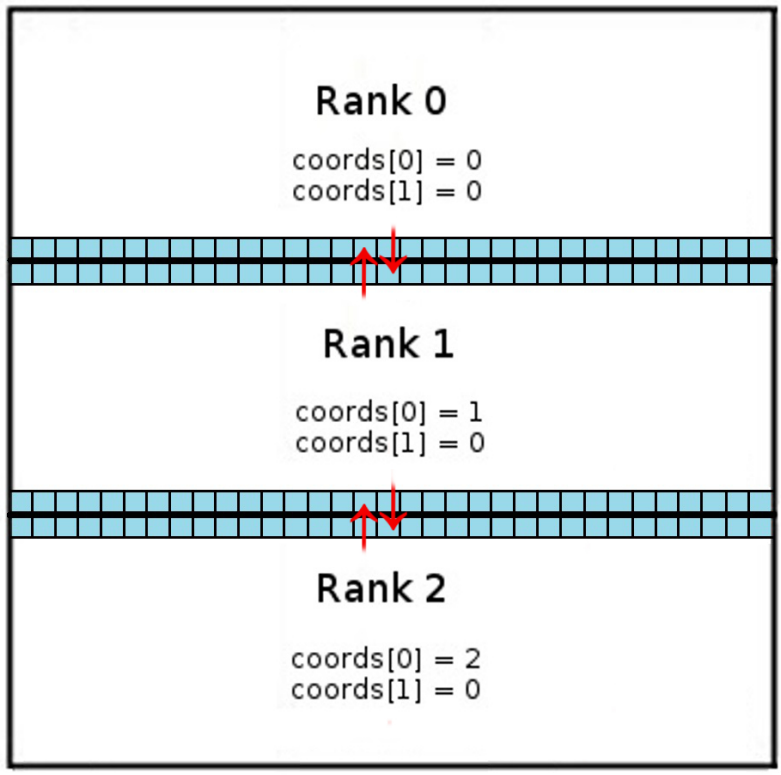

Figure 4.13: Layout with 3 processes.

In Figure 4.14 you can see a layout for 8 processes. The grid chosen for 8 processes is a 4 by 2 grid. Here borders are sent bought west/east and north/south. Here rank 0, rank 2, rank 4 and rank 6 has coords[1] = 0 and therefore has a boundary in the west. Rank 1, 3, 5 and 7 has coords[1] = dims[1]-1 = 2-1 = 1 and a boundary in the east. Here the local area is 32 by 128 floats large with local_temp 34 by 130 floats large for BORDER 1 and SIZE 256.

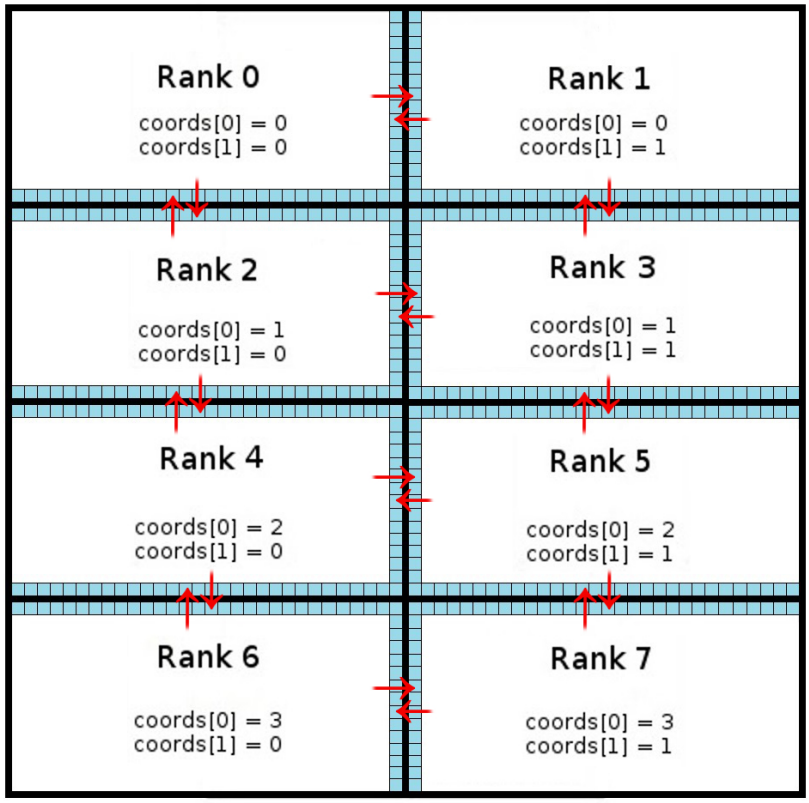

Figure 4.14: Layout with 8 processes.

The grid for 16 processes is 4 by 4 as seen in Figure 4.15. Here we have 4 processes that have 4 borders(rank 5,6,9 and 10) they need to send receive for each border exchange.

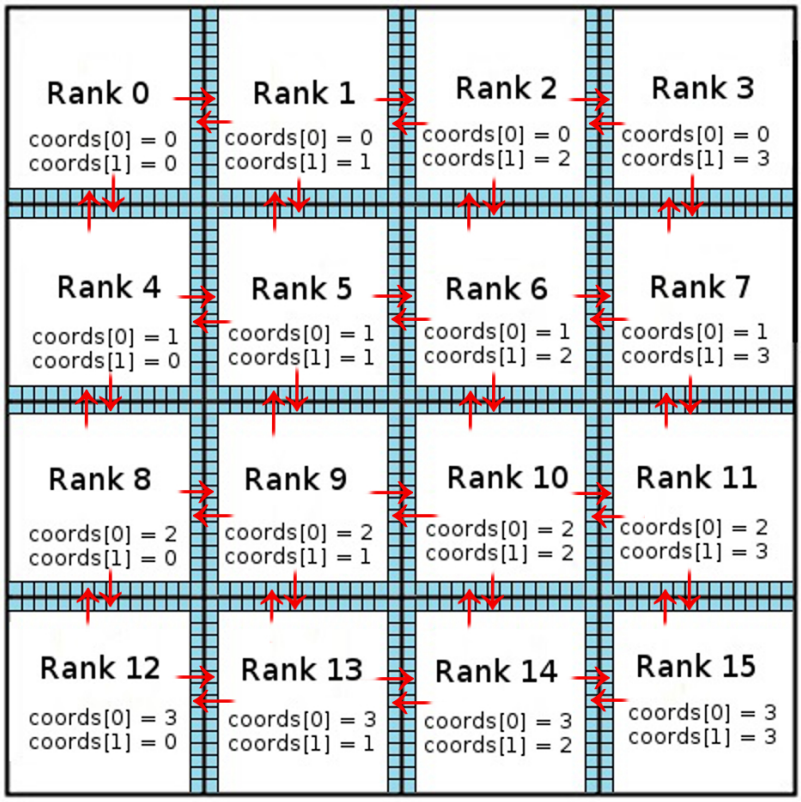

Figure 4.15: Layout with 16 processes.

4.5 Changing global variables

4.5.1 Changing size of the system

Changing the SIZE variable gives a bigger problem to solve since there are more numbers to compute for each step. For example by changing the size from 256 to 512 are there four times more numbers to compute for each step. In a serial version this would mean that the program would take four times more time to solve the program with size 256 than with 512. The results for the parallel version can be seen in Chapter 6.

4.5.2 Changing border thickness

Changing the border thickness means that the borders exchange method are called fewer times but more data are sent each times. The values in the border are calculated on the steps where the borders exchange method are not called. This means that the same amount of data are sent, but more numbers are calculated. The results can be seen in 6.3.

4.5.3 Writing to file

Changing the WRITETOFILE constant makes the program create a file with the given filename and write the content of the temperature matrix to that file.

4.5.4 NSTEPS and CUTOFF

Changing NSTEPS changes the number of steps the program executes. The more steps are executed, the more time passes.

CUTOFF is the when to cut of the heat to the aluminum. Bigger size of CUTOFF leads to that the external heat method being called more times.

4.6 Other examples of implementations on multiple processors

Parallelizing the serial version can be done by splitting the area into a grid or strips. Here two examples of parallelization of heat solution with stripe-partition is discussed.

4.6.1 HEAT2D Example - Parallelized C Version

One example is using a master / slave strategy to split the work up among the slave processes [3]. This example is for a Dirichlet Problem where the edges are kept at value 0.

The main matrix is split into strips of size n/slave-count by the master process where slave-count is the number of slaves that is the number of processes minus one (p-1). It the size is not evenly dividable the first n mod (p-1) get one extra row. Then the neighbors (left and right) are sent to the slaves by the master together with start and end point to compute, numbers of rows to compute, where those rows start (offset) and initial values for u. The master process then waits until the slaves are finished with the step and gets the result back by using standard blocking MPI_Recv. When the result is received at the end and written to disk.

After that the slaves receive the data sent to them set up the start and end row to compute out from the offset before it starts the steps. Each step starts with exchanging boundaries before updating the local u matrix. When the steps are done the result are returned to the master.

4.6.2 Horak and Gruber - Parallel Numerical Solution of 2D Heat Equation

The paper by Horak and Gruber [9] splits the matrix into strips that are (n+2)/(number of processes) high and n+2 wide, where n is the size of the matrix with border-thickness 1. The borders are used for storing the border condition in, adding an extra border where the row from the neighbor border is stored, so the stencil can use the values locally, decreasing the need for message passing.

4.7 Parallelized Versions Compared

The timed_heat version uses the MPI_Dims_create method discussed in Appendix A.3.2 to create a grid that is as dense as possible. This makes it possible to test with both gridded and striped partitions since dims can be preset before calling the MPI_Dims_create method. For striped partitions only it is probably better to do it like in the examples in Chapter 4.6 since the smaller code is faster to run.

The timed also uses a different solution to the system size not being evenly dividable by padding the local areas. This was done to ensure that the Data-type was equal for all the processes but it is probably not necessary for them to be equal. It seemed like a good idea at the time when the code was written. This does probably not affect the runtime since the fastest process must wait on the slowest anyway. But this does affect the use of memory so this a optimization problem that should be fixed in the future.

Like the Horak and Gruber version[9] the timed_heat uses a border. But the border in timed_heat can be changed. This is discusses in Chapter 4.5.2. The timed_heat also uses a offset as in the form of the world coordinates local_origin and rows as local area variable in the form of local_dims.

Like the "HEAT2D Example" the timed_heat as a sort of master slave strategy where rank 0 collect the results and write them to file but rank 0 also does as much work as the others.

Chapter 5

The Clustis3 and Numascale

The computers used in this thesis is named Clustis3 and Numascale and belongs to NTNU IDI (Department of Computer Information Science).

5.1 Clustis3

Clustis3 is a cluster. Clusters are supercomputers that are a system of computers connected together by a network. [14, p. 35]

Clusters have distributed memory so all communication must be done by messaging. MPI (Message Passing Interface see Chapter 1.6) is made for distributed memory systems like clusters.

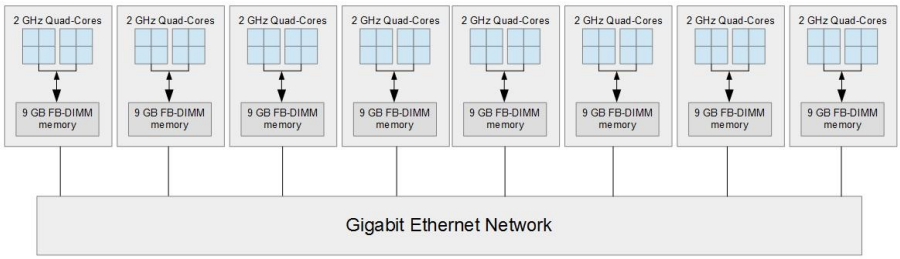

Figure 5.1: Clustis3 architecture the computing part

Clustis3 was installed in 2009 so it is old in computer years and consist of 9 nodes where 8 of them are used for computing and the last one is used for logging into the computer. Each of the nodes is a ProLiant DL160GS Server with two E5405 2.0 GHz Quad-Core , 9GB FB-DIMM memory, 160GB SATA harddrive and two GbE

network cards. [6] This gives that Clustis has 8 cores times 8 nodes equals to 64 cores altogether, but only 5 of the nodes worked at the time of my testing which resulted in only 40 cores being available. The architecture of the computing part of Clustis3 can be seen in Figure 5.1.

In Chapter 6.1 you can see the run times at Clustis3.

5.2 Numascale

The Numascale is a mainframe computer meaning that the hardware is created specifically for to be used for this. It is a shared memory system, but it also can be used with MPI.

The Numascale has 5 nodes connected together with a hypertransport network. Each node has two sockets with a AMD processor connected to it. Each processor has 8 cores. Each node has 128GB of memory and a NVIDIA GTX980 Graphics card. Each of these has 4GB of GDDR5 and uses PCI-E 3.0.

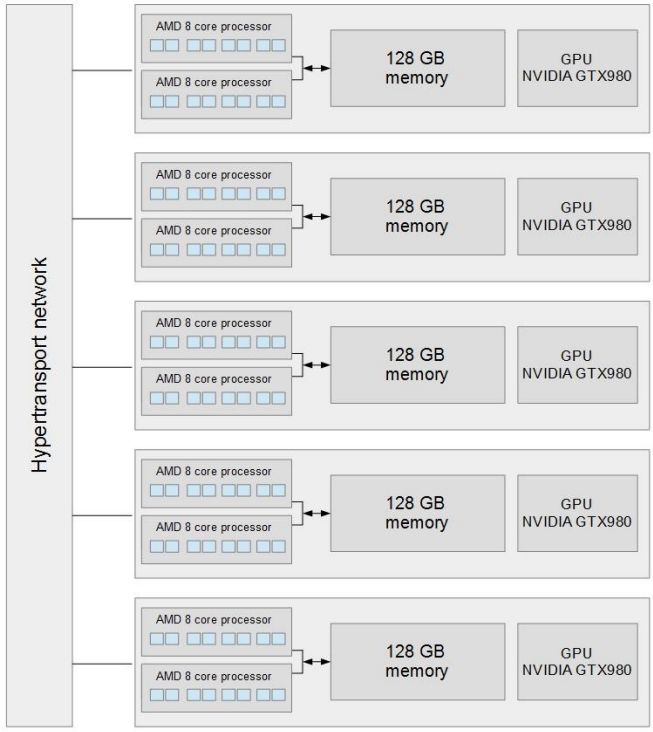

Figure 5.2: Numascale architecture

Numascale has two virtual CPUs per core by using SMT(simultaneous multi-threading). With SMT threads switches after each instruction to make use of the multiple functional units of the processor [14, p. 29]. This can give better or worse runtimes depending on the program.

Chapter 6

Running Heat Equation on Clustis3 and Numascale

The code used in these benchmark runnings on the two super computers Clustis3 and Numascale is discussed in this chapter. The code ran on multiple processors can be found in Appendix C.1. For one process the heat.c code found in Appendix C.2 is used.

6.1 Running heat equation on Clustis3 with different Sizes

On Clustis3 the code is run best with one process per core. This because of the process context switch who is too slow, only making processes wait on each other. This is common for older systems that doesn't support rapid threadswitching[14, p. 29]. To guarantee one processor per rank, I used one node for 1 to 8 processes, two nodes for 9 to 16 processes, three for 17 to 24, four for 25 to 32 and five nodes for 33 to 40 processes.

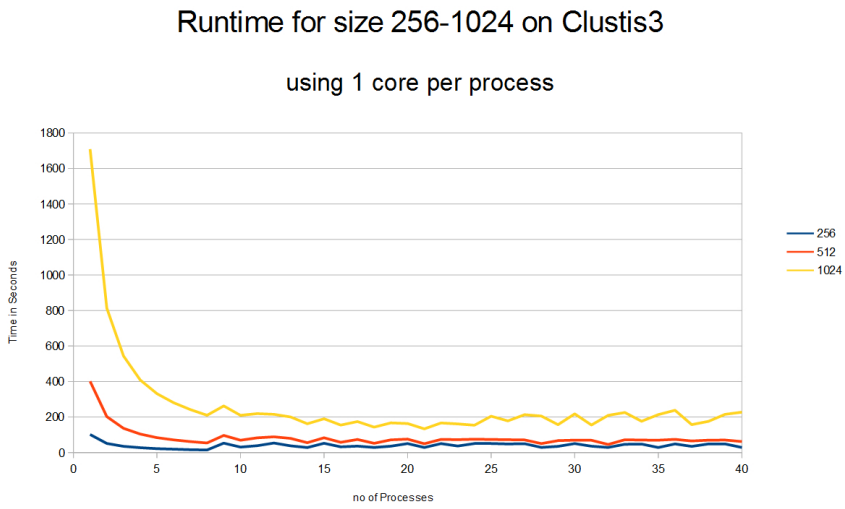

Figure 6.1: The runtime of different sizes on Clustis3

In Figure 6.1 the runtimes for one to forty processes for every sizes are compared. You can see the runtime slowly decreases from one to eight processes and then increases for higher number of processes with a lot of peaks and valleys.

When the size is doubled the amount of numbers to compute are quadrupled since the system is of size n by n. However, even if the data set is quadrupled, the time is only doubled between 256 and 512, and some more than doubled between 512 and 1024. This may be a sign that the processors aren't using 100% of each CPU.

Node layouts with width of 2 (10, 14 and 18 processes) are typically faster. This is explained in section 6.1.1.

One reason for 256 having higher runtimes for more than one node, can be that the size is too small to benefit from using more processes on several nodes. The communication costs are probably larger than the computational.

See figure 6.2 and 6.3 for more details.

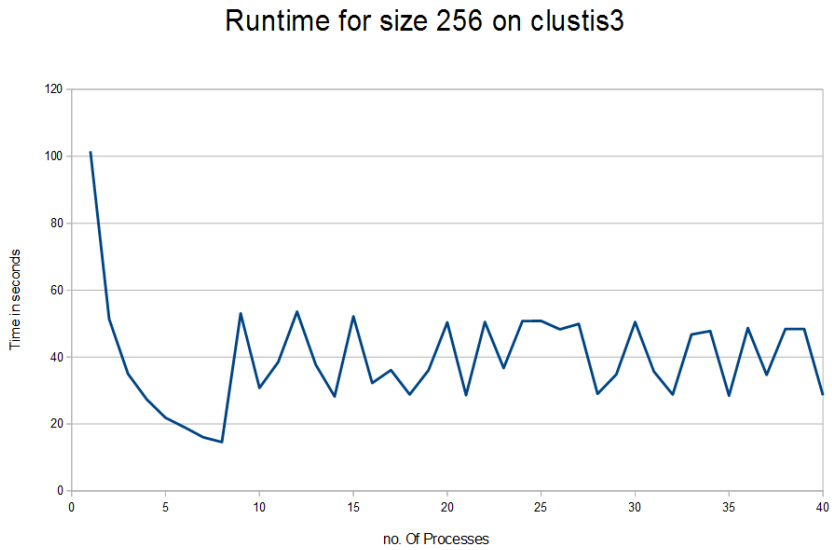

Figure 6.2: The runtime with size 256 on Clustis3 using 1 to 5 nodes

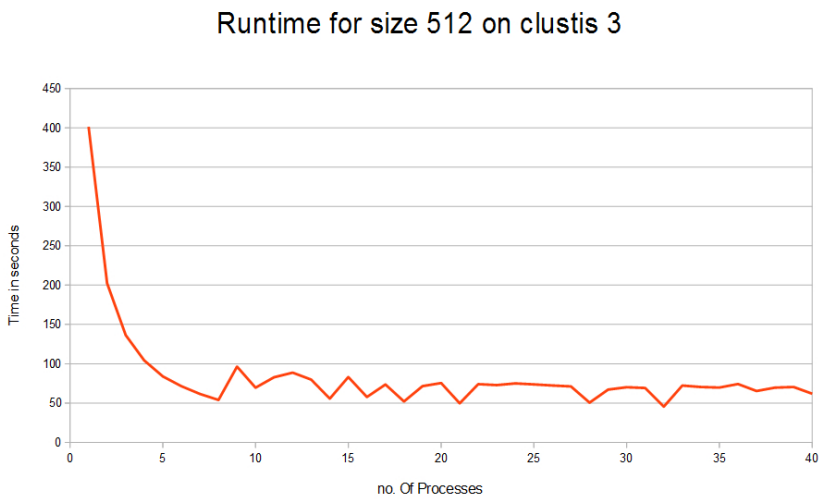

Figure 6.3: The runtime with size 512 on Clustis3

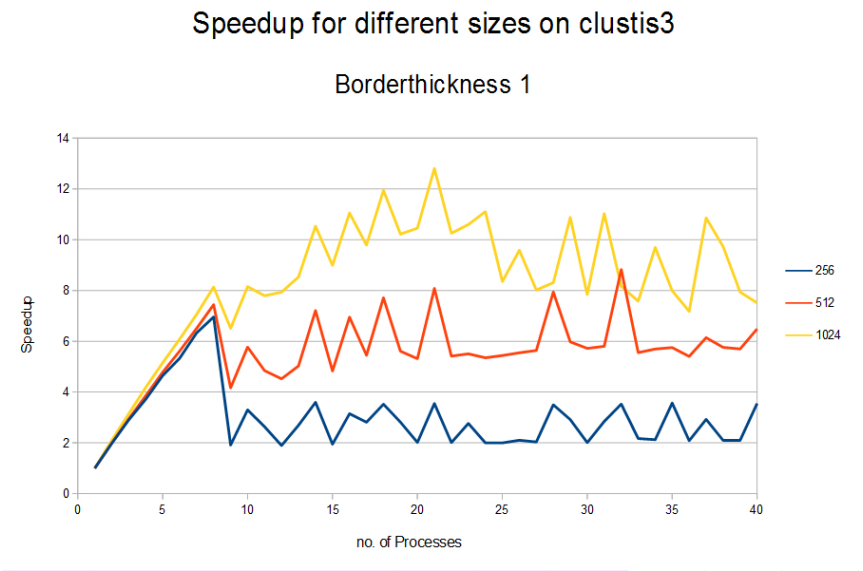

Figure 6.4: The speedup of different sizes on Clustis3

In Figure 6.4 the speedup($\frac{T_{Serial}}{T_{Parallel}}$[1]) for different sizes are posted. Here you can see that size 1024 benefits more than one node.

In Figure 6.5 the efficiency ($\frac{T_{Serial}}{T_{Parallel} \times p}$[1]) of different sizes is posted. An efficiency of 100% would be linear speedup. The speedup is almost linear for 1 node, but when communication comes into the picture, the speedup staggers. It is clear that the processors aren't fully utilized for the smaller sizes. On the larger sizes the CPU is utilized better, making the communication cost a smaller part of the runtime, as seen as speedup gaps between the sizes.

[1] where T_{Serial} is runtime for 1 processor and $T_{Parallel}$ is runtime for p processors

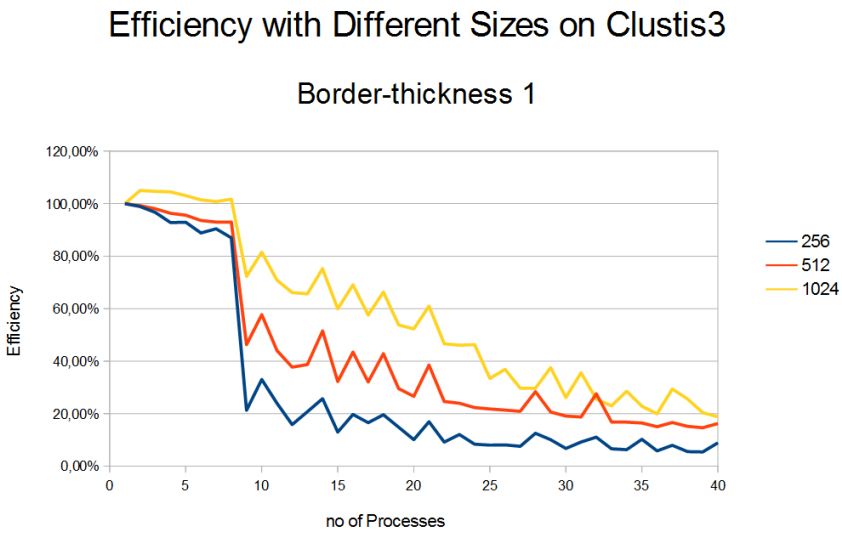

Figure 6.5: The efficiency of Different Sizes on Clustis3

6.1.1 The Node Layout

At first the results when using more than one node were slow. I decided to check which node each of the processes ran on. This can be seen in Appendix D.1: "Node Layout First Run".

I found out that for most layouts the processors all had neighbors on another node. This was the worst possible layout, since all the traffic went from one node to another, no internal node communication. Only the layouts with a width same as the number of nodes, had neighbors on the same node. This explains why 10, 14, 16, 18, 21, 28, 32 35 and 40 processes had faster runtimes.

6.1.2 Using rankfiles

To decide the process layout, which node each process is located, one may use a rankfile. There also exists execution flags for mpirun, but none of those gave the desired effect for Clustis3.

The rankfile maps each processes to a node, a slot and a core that they run on. The new layout is listed in Appendix D.2 "Node Layout Using Rankfiles". In the rankfiles the processes is distributed in a way, such that the first 8 processes runs on the first node, the next 8 on the second and so on. For number of processes that are 3 wide like 9, 12 and 18 the first 6 goes on the first node and the next 6 on the seconds. This way you minimize the messaging between the node.

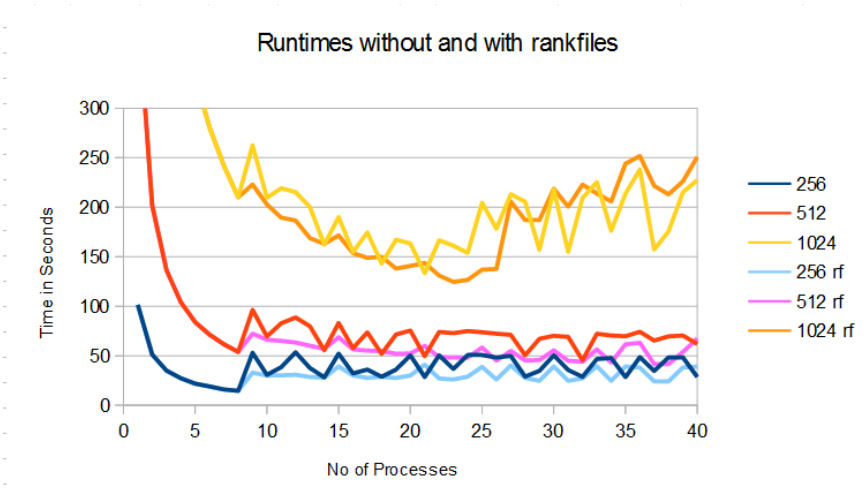

Figure 6.6: The runtime without and with using rankfile on Clustis3

The runtime is seen in Figure 6.6. Here you see that the curve is smoother for the runs marked "256 rf", "512 rf" and "1024 rf" (where rf is rankfile).

8 processes still got the fastest runtime for size 256. For size 512 the fastest runtime is for 37 processes and for size 1024 there is a decrease for up to 28 processes with 23 as the fastest. The speedup in Figure 6.7 and the efficiency in Figure 6.8 give a better picture of this.

15, 21, 25, 27, 30, 33, 35, 36, 39 and 40 processes has a bad layout with the rankfiles because of corners that make the process send two borders between different nodes. This is easy to see as valleys for size 256 and 512 in the speedup graph in Figure 6.7. Size 1024 is different. There is a large speedup for up to 24 processes. Above 24 processors the speedup drops suddenly. This is most likely because the cost of communication is higher than the benefit of more computational nodes.

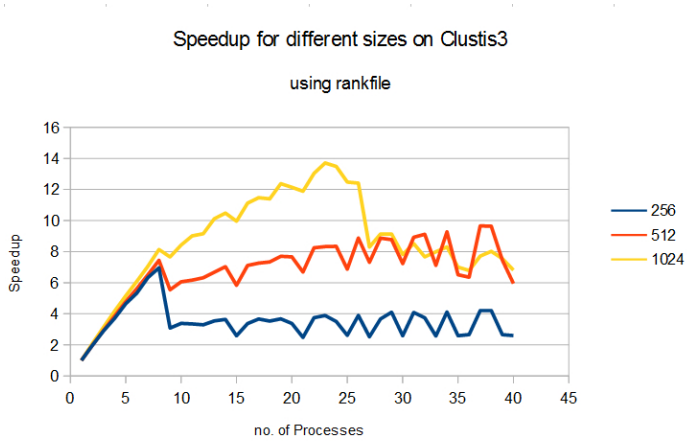

Figure 6.7: The runtime without and with using rankfile on Clustis3

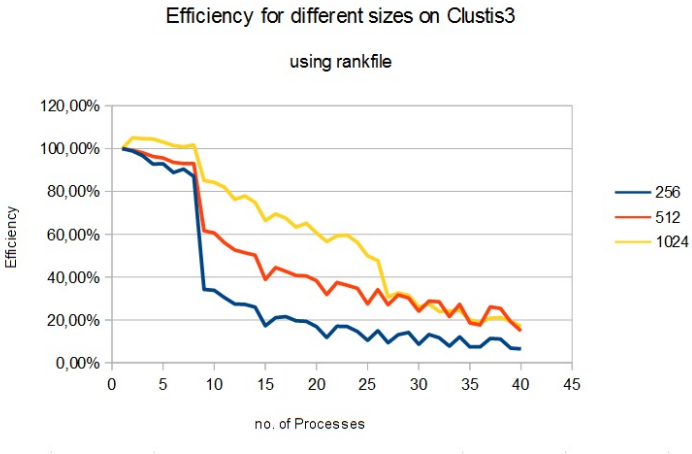

Figure 6.8: The runtime without and with using rankfile on Clustis3

6.2 Running heat on Numascale with different sizes

On Numascale the code is run with the "bind-to core"-flag for mpirun. The "bind-to core"-flag binds each process to a core. I.e. rank 0 runs on CPU 0, rank 1 runs on CPU 1, ..., rank n-1 runs on CPU n-1 and rank n runs on CPU n. Each CPU is a virtual CPU, where each physical CPU has 2 virtual CPUs using SMT (simultaneous multithreading, see Chapter 5.2).

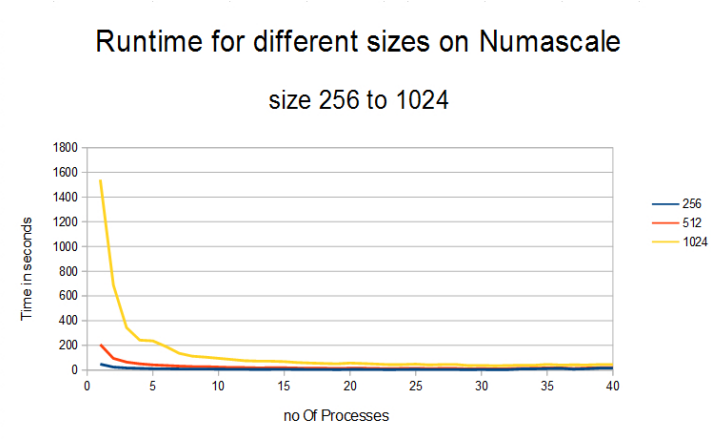

Figure 6.9: The runtime of different sizes on Numascale

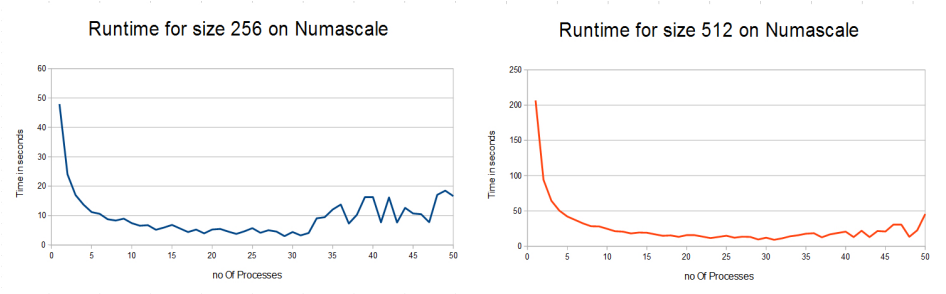

Figure 6.10: The runtime of different sizes on Numascale

The Figure 6.9 and 6.10 shows the results of running the heat system with size 256 to 1024 on Numascale. Here there is a decrease in time from 1 to 32 processes with higher runtimes for processes that have a dense layout like 25 and a low for the striped ones with odd-numbered processor count. The lowest runtime is for 29 processes. For more than 32 processes there is are higher runtimes because more than 1 node is used. The difference is largest for size 256 since the cost of communicating over more than 1 node is much larger than the benefit of getting more

computer power. The largest speedup can be seen around 32 processes Figure 6.13.

29 is a prime number. There is a small decrease for prime-numbered processes that have a striped partition (y by 1 grid). There is also lower runtimes for prime-numbered processes over 32. This is probably because some most send data to 2 to 4 other processes for each step while for the odd-numbered there is 1 to 2 processes with a larger amount of data instead. The processes that are communicated with are also closer for striped than dense because of the distribution of processes to CPUs.

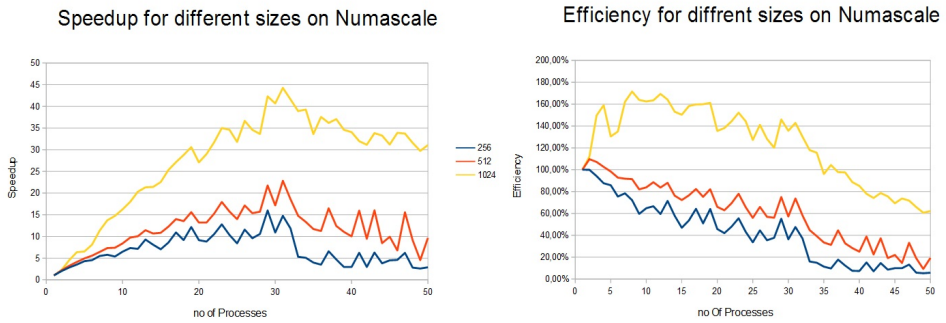

Figure 6.11: The speedup and efficiency of running different sizes on Numascale

The efficiency in Figure 6.11 also show that size 1024 is the most efficient with better than linear speedup (efficiency is over 100% compared to the serial version). Size 512 also have near 100% efficiency with 1 node. This probably comes from that the serial times for size 512 and 1024 being a bit big since size 512 is 4.3 times larger than size 256 and size 1024 is 7.5 times larger when both should be 4. But general it is still that the bigger the size is the bigger the efficiency is.

6.2.1 Compared to Clustis3

The Numascale is newer and has faster hardware, making it the fastest of the two computers. Clustis3 uses about twice the time on the serial version than Numascale. The fastest time for each size is also 5 times slower with size 256 and 512. For size 1024 it is about 4 times slower.

Both the Clustis3 runs and Numascale runs with different sizes shows that running on 1 node gives the best time if the size is not big as size 1024 on Clustis3 is. In that way Numascale is better since it has 16 cores on each node while Clustis3 has 8.

The curve for Numascale is smooth like for the rankfiles on Clustis3(see Chapter 6.1.2) since the distribution of processes among cores is more favorable. Numas-

cale however favors striped partitions more than Clustis3 does.

It is visible that the network between the node is faster since there is a smaller drop in the efficiency graph in Figure 6.11 after 32 than for 8 with Clustis3 in Figure 6.5. For the smallest size 256 at Clustis3 there is a 60% drop after 8 processes, but for Numascale there is only a 20% drop after 32 processes.

6.2.2 Splitting the workload into striped partitions on Numascale

When testing partition types, it looked like Numascale favored the striped partitions best with different sizes. I also tested striped partitions with different sizes, using number of processes that create a evenly dividable local area. The evenly dividable numbers are 1, 2, 4, 8, 16, 32, 64 and 128.

The layout is changed for each runs with setting dims[0] to 1 or setting dims[1] to 1. For 256 1*p, 512 1*p and 1024 1*p (vertical striped partitions) the layout is with dims[0] = 1 meaning that the grid is 1 process high and p processes wide, where p is no of processes. For 256 p*1, 512 p*1 and 1024 p*1 (horizontal striped partitions) the layout is with dims[1] = 1 meaning that the grid is 1 process wide and p processes high.

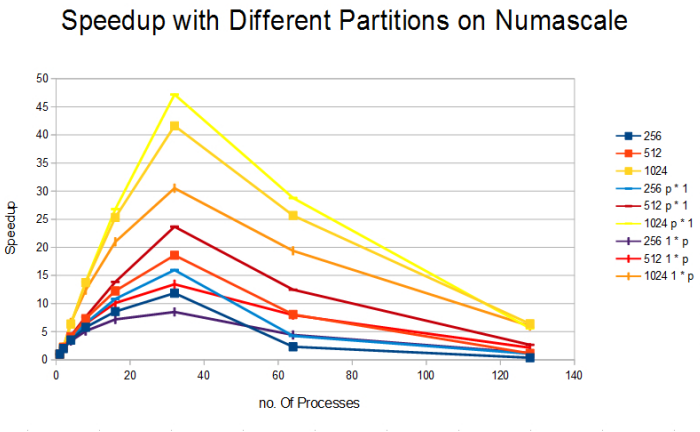

Figure 6.12: The speedup of running different sizes with horizontal and vertical striped partitions on Numascale

In Figure 6.12 the speedup for a run with number of processes that create a evenly dividable local areas are posted. The speedup shows that 32 processors that only use 1 node gives the best speedup. Using 64 processes is really bad, even the largest size. To get a better speedup, some of the tricks discussed in future work (see Chapter 7.2) has to be used.

The horizontal striped has the best speedup overall for size 512 and 1024. For size

256 the horizontal striped grids have the best for up to 32. The horizontal stripes are the easiest to send computing wise, that is probably why it got the best results. This is because the messages is of a simultaneous part of the local memory area. 32 processors for size 512 and 1024 got better time than 29 processors in the earlier runs.

The vertical has the worst speedup for all sizes, except for 128 processes where it is the same as for horizontal striped. The vertical stripes are the most complex to send computing wise since the message consists of floats that are strided in the local memory area.

The dense grid has a average speedup for all sizes, except for size 256 at 64 and 128 processes. The dense grid has both vertical and horizontal messages, explaining the average speedup.

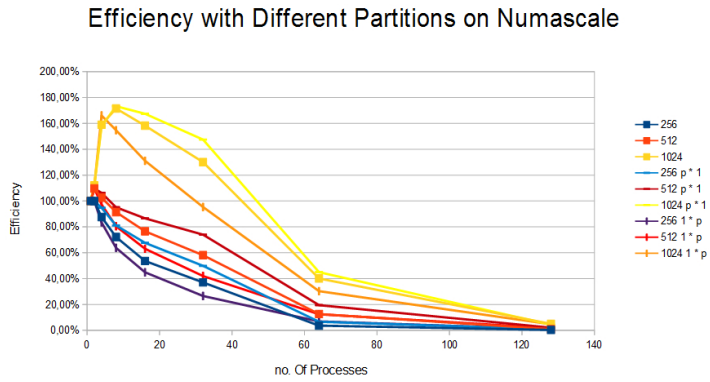

Figure 6.13: The efficiency of running different sizes with horizontal and vertical striped partitions on Numascale

The efficiency in Figure 6.13 show what the largest sizes are most efficient with the horizontal striped partition towards 32 processes, when the efficiency drops.

6.3 Changing size of Border

6.3.1 Without using rankfiles on Clustis3

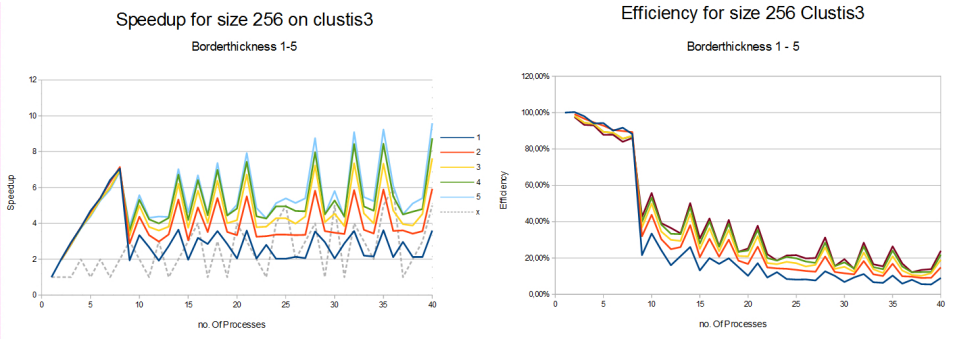

Figure 6.14: The speedup and efficiency with different border thicknesses and size 256 on Clustis3 without using rankfiles

In Figure 6.14 you see the speedup($\frac{T\_Serial}{T\_Parallel}$) and efficiency($\frac{Speedup}{p}$) for size 256 on Clustis3. For up to 8 threads there is almost none difference in using borders or not. For processes using more than one node there is a increase in speedup and efficiency. Those of the processes that have neighbor processes on the same node has a increase in speedup (that is for 10, 14, 16, 18, 21, 28, 32, 35 and 40). The bigger the border, the bester the speedup is.

Border thickness 2 saves in average 10,35 seconds while 3 saves 3.45, 4 saves 1.7 and border thickness 5 saves 1.01 seconds in average for process 9 to 40 between itself and the border thickness that is one narrower. This shows that border thickness 2 is the most effective for this setup.

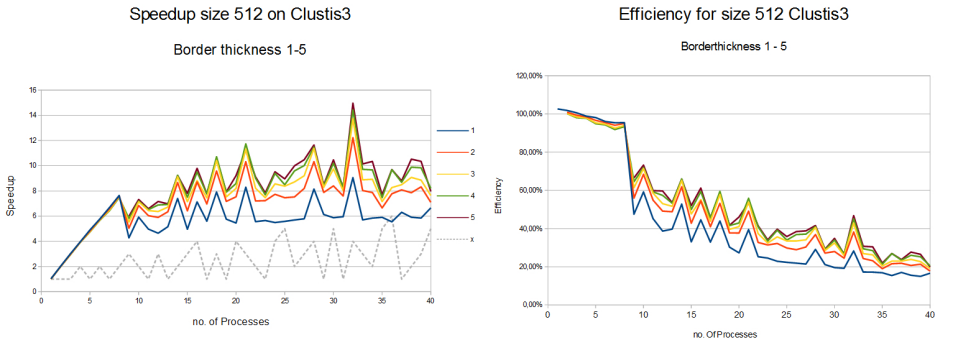

Figure 6.15: The speedup with different border thicknesses and size 512 on Clustis3

Most of result for size 256 can also be seen for size 512 in Figure 6.14. 10, 14, 16, 18 and 32 has a large increase in speedup but 38 and 40 doesn't have this large increase the larger the borders are.

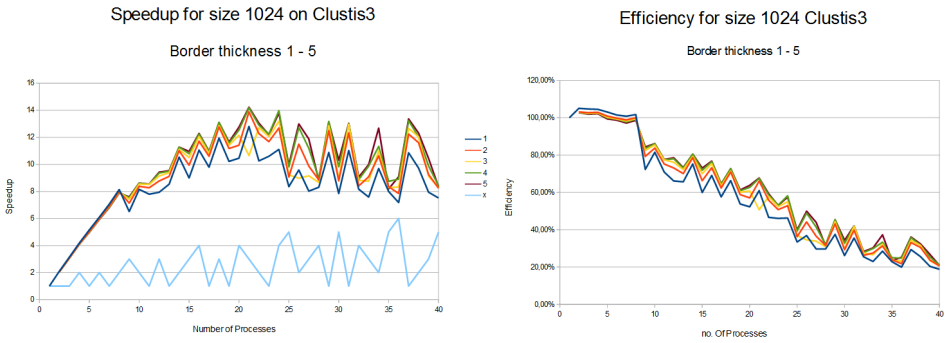

Figure 6.16: The speedup with different border thicknesses and size 1024 on Clustis3

In Chapter 6.1 we saw that the graph for size 1024 was different. 21 has still the highest speedup except from with border thickness .

6.3.2 With Rankfile on Clustis3

Figure 6.17 shows the border use gives a higher speedup with the rankfiles as well. For size 256 and border thickness 4 and 5 a higher speedup than for 8 is achieved. It is smoother with thicker borders. It is also lesser speedup increase to achieve the thicker the borders get. By example is 4 and 5 very close but 1 and 2 is far apart.

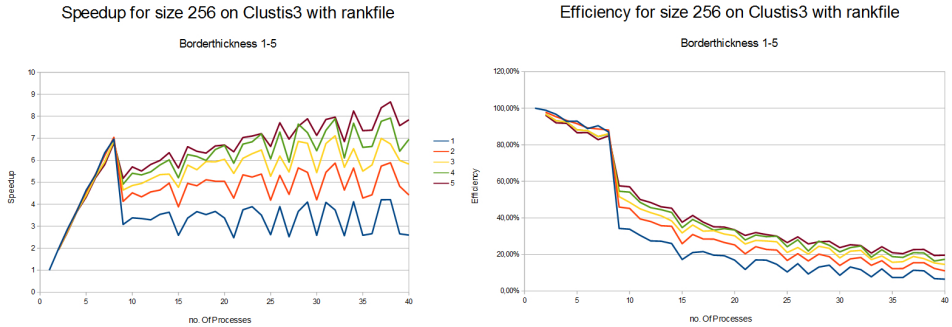

Figure 6.17: The speedup with different border thicknesses and size 256 on Clustis3

For size 512 seen in Figure 6.17 there is the same results as for size 256. The curve is even more smooth since layout becomes less important for bigger border thicknesses.

For most of the results that good results in 6.3.1 there is a valley because of the rankfile creating worse layouts. For example 35 processors has a big valley since it has a corner where it has to send to another node two times.

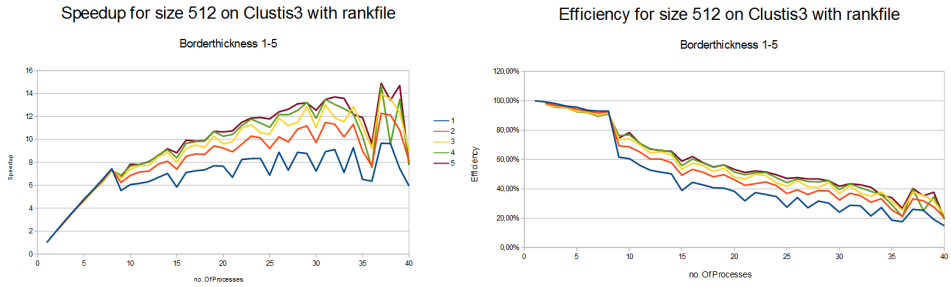

Figure 6.18: The speedup with different border thicknesses and size 512 on Clustis3

Size 1024 in Figure 6.19 has a visible increase in speedup with border thickness 2 but for each increase in border thickness the increase is reduces. The graphs are

similar as for the border thickness 1 but gets a bit smoother as for size 256 and 512. There is one exception with border thickness 3 on 21 processors that may be a disturbance since the rest of the border thicknesses has an improvement for 21 processes.

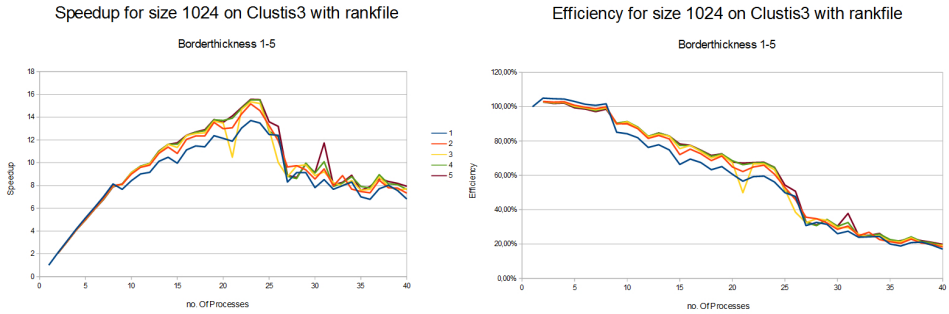

Figure 6.19: The speedup with different border thicknesses and size 1024 on Clustis3

Size	Average 1 to 2	Average 2 to 3	Average 3 to 4	Average 4 to 5
256	10.35 sec	3.45 sec	1.70 sec	1.01 sec
512	10.60 sec	4.18 sec	0.57 sec	1.61 sec
1024	11.20 sec	0.17 sec	4.12 sec	1.76 sec

Figure 6.20: Average difference between border thicknesses for 9 to 40 processes

Figure 6.20 shows that border thickness 2 is the most effective and that the time saved decreases for each increase in border thickness. The timed saved is about the same for all the sizes witch gives that the time saved is not directly related to size.

6.3.3 Numascale

For size 256 seen in Figure 6.21 the border thickness 2 and 3 is an improvement for all number of processes. With border thickness 4 there is an improvement for dense layouts and decline in speedup for the ones that are one wide witch create a more smooth graph since the layout gets less important. Border thickness 5 is even slower for up to 32 processes. For over 33 processes the border thicknesses 4 and 5 is slower for the layouts that are 1 wide but gives a higher speedup elsewhere. 5 has the highest speedup most of the time.

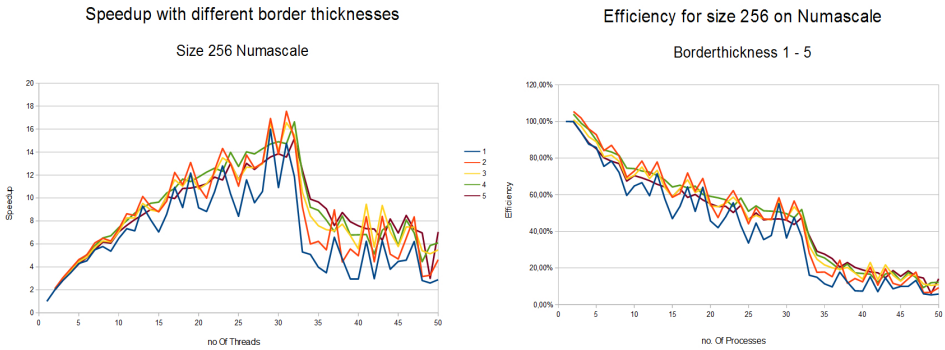

Figure 6.21: The speedup and efficiency with different border thicknesses and size 256 on Numascale

For size 512 the border with thickness 2 gives increase in speedup for most number of processes. Higher border thickness gives decreasing or small speedups.

Number of Processes that are prime-numbered and that are 23 or higher has a decrease or very small increase in speedup with border thickness over 1.

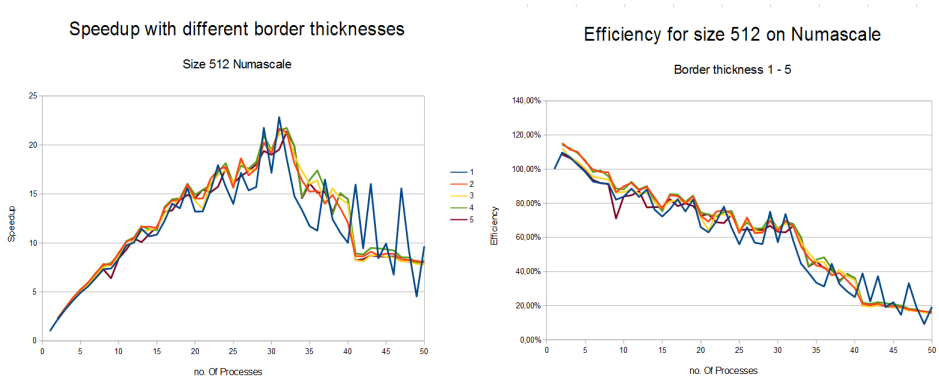

Figure 6.22: The speedup and efficiency with different border thicknesses and size 512 on Numascale

For size 1024 gives border thickness 2 an increase in the speedup while 3 gives a worsening. Border thickness 4 gives the best speedup for some of the number of processes while border thickness 5 is worse than 4 for all number of processes.

Using thicker borders have a better effect for over 32 processes on Numascale with size 1024.

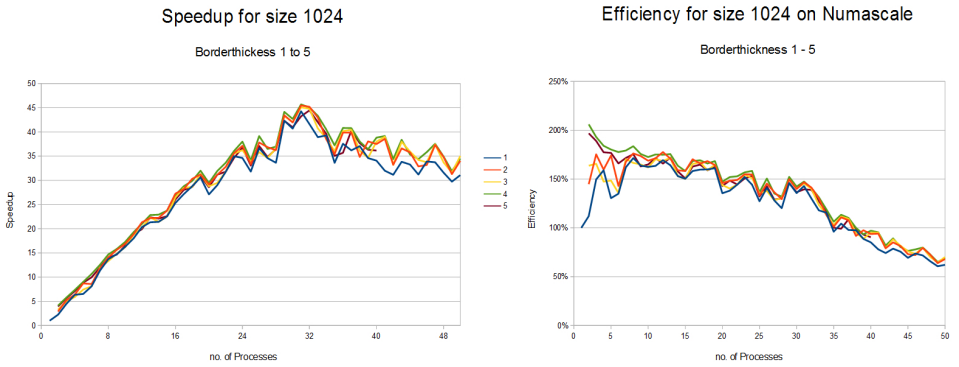

Figure 6.23: The speedup and efficiency with different border thicknesses and size 1024 on Numascale

Size	Average 1 to 2	Average 2 to 3	Average 3 to 4	Average 4 to 5
256	3.35 sec	1.88 sec	1.70 sec	1.01 sec
512	0.38 sec	-0.09 sec	0.81 sec	-0.87 sec
1024	2.92 sec	0.17 sec	0.96 sec	-1.74 sec

Figure 6.24: Average difference between border thicknesses for 33 to 50 processes

Figure 6.24 shows that the time saved on using border with different sizes. Size 256 saved the most time in using borders on more than 32 processes.

Compared to Clustis3

Borders on Numascale doesn't make such a big difference as on Clustis3. That is probably because the cores on Numascale is already well utilized with the multi-threading(see Chapter 5.2).

6.4 Writing to file

The number of processes run should not impact writing to file times since writing to file is done by rank 0. All the is sent to rank 0 for all the runs so the extra time needed is actually the time rank 0 uses to write to file since the rest of the processes must wait on rank 0 to finish the task.

6.4.1 Numascale

For Numascale there is used 29 threads.

Size	Time 29 processes with write to file	Time 29 processes	Difference
256	3.88 sec	3.00 sec	0.88 sec
512	11.60 sec	9.50 sec	2.10 sec
1024	44.96 sec	36.42 sec	8.55 sec

The difference in time between size 256 and 512 is 2,4 times slower and between 512 and 1024 it is 4 times slower. The 4x between 512 and 1024 indicates that this can be linear but more data is needed for determining that.

6.4.2 Clustis3

For Clustis3 there is used 8 processes.

Size	Time 8 processes with write to file	Time 8 processes	Difference
256	27.57 sec	14.81 sec	12.76 sec
512	90.86 sec	54.85 sec	36.02 sec
1024	256.03 sec	213.94 sec	42.09 sec

The difference in time between size 256 is 2.82 times bigger than for 512 and the time between size 512 and 1024 is 1.17 times bigger. This indicates that the differences in time is not linear.

6.4.3 Write time compared

The differences in time between Numascale and Clustis3 are large. The Clustis3 uses 5 to 17 more time to store than Numascale. This is because of the storing device for Clustis3 is farther away since Clustis3 is a cluster and Numascale is a mainframe.

6.5 Max min and runtimestabilty

All results are a mean of at least 20 runs. This amount is kept as low as possible to have time to run the code with as many different changes as possible and high enough to avoid that single runs make a big impact on the results. The parts with the large differences gives runtimes that are the least trustworthy timings.

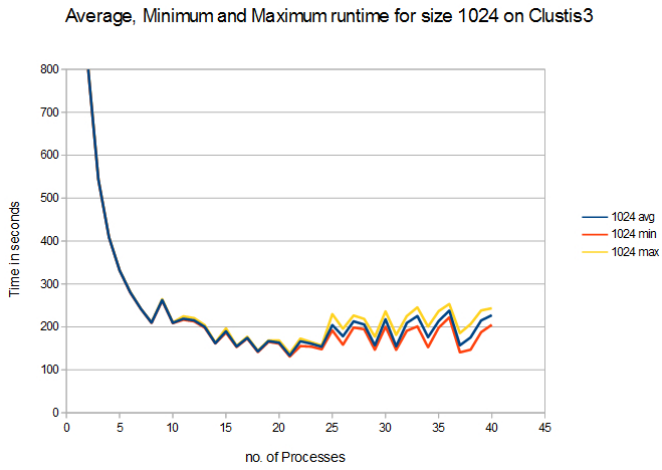

Figure 6.25: Min, max and average runtimes for size 1024 on Clustis3

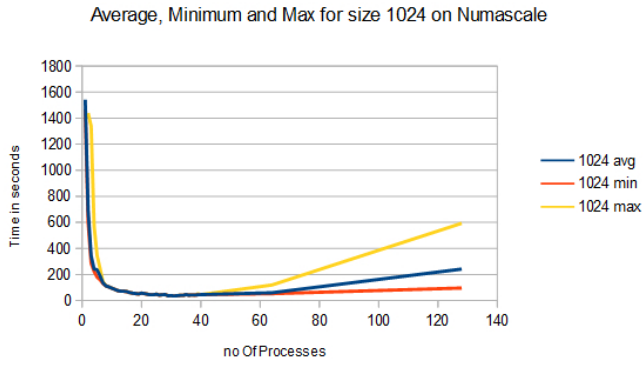

Figure 6.26: Min, max and average runtimes for size 1024 on Numascale

Size 1024 had the biggest time differences for both the computers with the larger number of processes seen in Figure 6.25 for Clustis3 and Figure 6.26 for Numascale. For Numascale is there is also some runs with the smaller number of processes that created a small dent in the average graph.

Chapter 7

Conclusions and future work

7.1 Conclusions

Comparing Numascale and Clustis3, you can see that larger and newer computers are faster, however the design of the parallel program is also important to achieve effective and low runtimes. An example can be seen in Chapter 6.1, where the the distribution of processes among cores was important for the best runtime for Clustis3. Numascale gets the best results with horizontal striped partitions using one node, as discussed in Chapter 6.2.2. Increasing the border thickness on Clustis3 gives faster runs, but does not work well for Numascale, probably because of simultaneous multithreading (SMT), see Chapter 6.3.

When calculating small samples (heat equation sizes) of data, both Clustis3 and Numascale does a lot better only using one node with maximum CPUs on that node. This is because of the communication cost has a higher time penalty than doing the calculations locally. Keeping communication low between nodes is crucial, optimizing and minimizing the exchanged data. On larger data sets the distribution costs grows short compared to the calculations. Distribute the data in such a way that every node and core has an equal amount of work, is another way to decreasing communication.

Benchmarking is hard. A lot of variables makes it difficult to find what is really interesting. The difference computer architectures makes it hard to compare different computers. However there are many good benchmarking tools, but all require optimization to get the best results.

7.2 Future work

7.2.1 Testing without SMT

Turning SMT (simultaneous multithreading) off on Numascale might give better runtimes since SMT depends on the problem [14, p. 29]. Increasing border-thickness probably gets better results on Numascale with the SMT off.

7.2.2 Different benchmarking

Using different benchmarking techniques and comparing the results between the to computers. Other benchmarking could be FLOPS (Floating points per second), Linpack and many more. It is important to remember that different super computers have different architecture, making a direct number compare between two computers wrong.

7.2.3 Measuring communication

Optimize memory and communication usage. By focusing on the memory usage and how much time is spent communicating, you could find how effective the programs are in those terms, and most likely get more power out of the computers.

7.2.4 Benchmark power usage

Green computing is on the rise and energy usage is now more critical than ever. The effectiveness of a super computer per Watt is increasingly important to help with the worlds pollution problems. It can also be an economical initiative to keep the power consumption lower.

7.2.5 Comparing MPI to P-threads or OpenMP on Numascale

Since Numascale has a shared memory architecture, it would be interesting to compare the MPI version with a P-thread version. Most likely the P-thread and OpenMP versions would be faster, since they are made for shared memory.

7.2.6 Optimizations

Here are some suggested optimizations for the code.

7.2.7 Striped partitions

The code tested uses different dimension for splitting up the workload. A program that only splits the area in horizontal stripes is much easier to compute since it is easier to write and does less in the set up phase. It would probably therefore

take less time to compute. This was also the split up that gave the best result for Numascale, see Chapter 6.2.2.

Non-even dividable areas solution

The size of the system is only evenly dividable on a few numbers (1, 2, 4, 8, 16, 32, 64, ...). Therefore there is some rows or columns that are leftover. This can be solved by adding one more row or column to the local area. To get equal size for the local areas the local area can be padded.

A solution that may be better, and is easy to implement with the striped partitions, is to start with the first processor and give one more row until there are no more leftovers left so the processors have different local areas. This would be nice to compare with the padded solution.

7.2.8 Optimizing for L cache

Optimizing for the different L1-L3 caches may increase the performance of the program and give you a better speedup.

References

[1] M. Anderson, "Better Benchmarking for Supercomputers," *IEEE Spectrum*, vol. 48, no. 1, pp. 12 – 14, 2011. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5676366

In this article Mark Anderson arguments why gigateps also should be a benchmark.

[cited at p. 13]

[2] S. Balaji and W. Feng, "Understanding Power Measurement Implications in the Green500 List," 2010, pp. 245 – 251.

Article about measuring power(FLOPS/watt) for supercomputers

[cited at p. 13]

[3] B. Barney and G. L. Gusciora, "Heat2d example - parallelized c version," 2013. [Online]. Available: https://computing.llnl.gov/tutorials/mpi/samples/C/mpi_heat2D.c

Heat parallized 2D example

[cited at p. 35]

[4] J. J. Dongarra, P. Luszczek, and A. Petitet. (2001) The LINPACK Benchmark: Past Present and Future. [Online]. Available: http://www.netlib.org/utk/people/JackDongarra/PAPERS/hpl.pdf

Paper about LINPACK and the different variations of the LINPACK benchmark.

[cited at p. 12]

[5] eFunda, "Heat equation (temperature determination)," 2015. [Online]. Available: http://www.efunda.com/formulae/heat_transfer/conduction/overview_cond.cfm

This webpage tells how to find the thermal diffusivity

[cited at p. 16]

[6] A. C. Elster. (2011). [Online]. Available: http://www.idi.ntnu.no/~elster/tdt4200/f2011/tdt4200-f2011-lec01-02.pdf/

Lecture about the computers at IDI NTNU

[cited at p. 38]

[7] R. Fish, "The future of computers - Part 1: Multicore and the Memory Wall," 2011. [Online]. Available: http://www.edn.com/design/systems-design/4368705/The-future-of-computers--Part-1-Multicore-and-the-Memory-Wall

Article about the history of processors which explain the limits in different ways to solve memory wall.

[cited at p. 2]

[8] J. L. Gustafson. (1988) Reevaluating Amdahl's law. [Online]. Available: http://mprc.pku.edu.cn/courses/architecture/autumn2005/reevaluating-Amdahls-law.pdf

In this paper Gustafson suggest a new law for speedup instead of Amdahls law.

[cited at p. 5]

[9] V. Horak and P. Gruber, "Parallel Numerical Solution of 2-D Heat Equation," in *Parallel Numberics '05*, https://www.cosy.sbg.ac.at/events/parnum05/book/horak1.pdf, 2005, pp. 47 – 56.

This paper suggest a numerical solution of the two dimensional heat equation for both single processor and parallel processors.

[cited at p. 15, 17, 35, 36]

[10] J. J. D. jr, J. R. Bunch, and G. W. Stewart, *LINPACK Users' Guide*. SIAM, 1979.

User Guide for LINPACK. Tells the history about the LINPACK benchmark

[cited at p. 11]

[11] E. Kreyszig, *Advanced Engineering Mathematics 9th*. Wiley, 2006.

Math book

[cited at p. 15]

[12] OpenMPI. (2015). [Online]. Available: http://www.open-mpi.org/doc/

Documentation for Open MPI. Open MPI version 1.3 and 1.8 is the MPI software installed in Clustis3 and Numascale

[cited at p. 7, 71, 72, 73, 74, 75, 76, 77]

[13] P. Pacheco, *Parallel programming with MPI*. Morgan Kaufmann, 1997.

Book about the fundamentals in parallel programming with MPI examples

[cited at p. 8]

[14] ——, *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011, iSBN 978-0-12-374260-5.

Book about the fundamentals in parallel programming.

[cited at p. 1, 2, 6, 7, 8, 9, 37, 39, 41, 62, 71, 72, 73, 74, 75, 76, 77]

[15] D. A. Patterson. (2006) Future of Computer Architecture. [Online]. Available: http://www.slidefinder.net/f/future_computer_architecture_david_patterson/patterson/6912680

Slides from a talk David A. Patterson held in February 2006 talking about the change in Computer Architecture over time and new limitation (or so called walls) in architecture

[cited at p. 1]

[16] N. Satish, C. Kim, J. Chhugani, and P. Dubey, "Large-Scale Energy-Efficient Graph Traversal: A Path to Efficient Data-Intensive Supercomputing," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, November 2012, pp. 1 – 11. [Online]. Available: http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6468457

This paper is about gigaTEPS and the Graph500 benchmark. It talkes about why the gigaTEPS exist and how to implement it

[cited at p. 13]

[17] T. R. Scogland, C. P. Steffen, T. Wilde, P. Florent, S. Coghlan, N. Bates, W.-c. Feng, and E. Strohmaier, "A Power-Measurement Methodology for Large-Scale, High-Performance Computing," 2014, pp. 149 – 159.

This paper talks about how to measure power and benchmarking in general.

[cited at p. 11, 13]

[18] Top500. (2014) Linpack Benchmark. [Online]. Available: http://www.top500.org/project/linpack/

About the Linpack Benchmark and why it was choosen for the Top500 list

[cited at p. 12]

Appendices

Appendix A

MPI functions

A.1 Structure functions

A.1.1 MPI_Init

MPI_Init takes in pointers to argc and argv and initializes MPI.[12] No MPI routines should be called before MPI_Init. (tells the MPI system to do all of the necessary setup[14, p. 86] like define MPI_COMM_WORLD)

After the initialization MPI_Comm_size and MPI_Comm_rank are called to get the size and rank for the processes. Size is the total number of processes and rank is the process number that are unique for each process and is a number from 0 to size-1. The rank is the identification of each process. The rank and size is used for messaging purposes.

A.1.2 MPI_Finalize

MPI_Finalize cleans up all the MPI states and must be called by all processes before exit.[12] No MPI routines should be called after MPI_Finalize.

A.2 Send and receive

All communication between nodes are done through messages. There are a number of different send and receive methods for different purposes. MPI_Send, MPI_Recv, MPI_Sendrecv and MPI_Isend are the ones used by the timed_heat code in Chapter C.1.

A.2.1 Modes

There are four modes for MPI sends. Standard, synchronous, ready and buffered. [14, p. 323]

Standard

In standard mode the MPI implementation chooses between blocking or copy the message to its own storage[14, p. 323]. MPI_Send is a standard send and is blocking in open-mpi[12].

Synchronous

Blocks until a matching receive is posted[14, p. 323]. MPI_Ssend is a synchronous send.

Ready

Recieve must be posted before the send[14, p. 323]. MPI_Rsend is a ready send.

Buffered

A copy of the message is buffered if a matching receive hasn't been posted[14, p. 323]. The buffer is provided by the user not the MPI implementation. MPI_Bsend is a blocking send.

A.2.2 MPI_Send

MPI_Send is a blocking send. [12]. With blocking means that the program will hang until receive is posted. This method has six parameters: buf, count, datatype, dest, tag and comm.

- buf is the address to the buffer you are sending from
- count is the number of elements you are sending
- datatype is the MPI data-type of the elements you are sending
- dest is process you are sending to
- tag is an integer message tag
- comm is the communicator (see chap A.2.5)

A.2.3 MPI_Recv

MPI_Recv is a standard mode blocking receive[12]. With blocking means that the program will hang until the receive has a matching send(see 1.6.5). This method has seven parameters: count, datatype, source, tag, comm, buf and status

- count is the maximum number of element to receive
- datatype is the data-type of the elements you are receiving
- source is the process you are receiving from
- tag is a integer message tag
- comm is the communicator (see chap A.2.5)
- buf is the buffer to save the elements in
- status is a structure explained in chap A.2.3.

MPI_Status

The MPI Status is a structure with at least three variables: MPI_SOURCE, MPI_TAG and MPI_ERROR. MPI_SOURCE is the rank that sent the message and MPI_TAG is the tag sent. MPI_ERROR is an code for identifying errors. MPI_Get_count method can be used to get the count of elements received.[14, p. 92-93] With the MPI_STATUS_IGNORE argument this part of functionality of the MPI_Recv method is ignored.

A.2.4 MPI_Sendrecv

MPI_Sendrecv is method that is a combination of MPI_Send and MPI_Recv and prevents deadlocks when dealing with cyclic dependencies since the subsystem deals with instead of the user [12]. MPI_Sendrecv has ten parameters: sendbuf, sendcount, sendtype, dest, sendtag, recvcount, recvtype, source, recvtag and comm.

- sendbuf is the buffer to send from
- sendcount is number of elements to send
- sendtype is the send MPI datatype
- dest is the destination rank
- sendtag is the integer message tag for sending
- recvcount is the max number of elements to receive

- recvtype is the datatype of the elements to be received
- source is rank that you are receiving from
- recvtag is the integer receive message tag
- comm is the communicator (see chap A.2.5)

A.2.5 Communicator

MPI_Comm_size and MPI_Comm_rank uses a communicator as an argument. A communicator is a a collection of processes that can send messages to each other[14, p. 87]. MPI_COMM_WORLD is the communicator used for all the process and is set up by MPI_Init.

MPI_Cart_create

Set up a new communicator for Cartesian topology with information for the topology attached.[12] Takes in six parameters: old_comm, ndims, dims, periods, reorder and comm_cart:

- old_comm is the old communicator
- ndims is the number of dimensions in the Cartesian grid
- dims is the size of the dimensions in the grid
- periods is a array with booleans of ndims size to specify if the grid is periodic or not
- reorder is a boolean to tell if the ranks can be reordered or false if the new group is identical to the old one.
- comm_cart is the new communicator

The MPI_Cart_create method is used in timed_heat to create a communicator called cart that is used in the MPI_Sendrecv methods in border_exchange and to get information about the topology through using the MPI_Cart_coords method that is used to store the data in the right place in the collect_area method.

MPI_Cart_coords

MPI_Cart_coords returns the coordinates for a process in a Cartesian topology communicator. [12] Has four parameters: comm, rank, maxdims and coords

- comm is the communicator for the Cartesian topology

- rank is the process number
- maxdims is the length of vector coordinate in the calling program
- coords is the out-parameter and is a integer array of size ndims(as created with MPI_Cart_create)

MPI_Cart_shift

MPI_Cart_shift returns the shifted source and destination ranks and is often used together with MPI_Sendrecv in a Cartesian process topology. [12] Has five parameters: comm, direction, disp, rank_source and rank_dest

- comm is a communicator with an Cartesian structure
- direction is the direction the shift is performed and is a coordinate dimension
- disp is the displacement. Negative numbers gives upward shift and positive downward shift.
- rank_source is the rank of the source process
- rank_dest is the rank of the destination process

MPI_Cart_shift is used in timed_heat to get the north, south, west and east rank by shifting downward in both directions.

A.2.6 Collective Communication

Collective communication is when more than two processes are involved in the same communication function.

MPI_Reduce

Perform a global reduce operation on all members on a communicator and stores the result on the root specific rank[12]. Exists also a MPI_Allreduce that stores the result on all the processes[14, p. 106].

MPI_Bcast

Broadcast data from one process to all the other processes in the communicator [14, p. 106].

MPI_Scatter

Split a vector into pieces and scatter them among the processes starting with process 0[14, p. 111].

MPI_Gather

Gather pieces of vector on one process[14, p. 112]. Opposite of MPI_Scatter. Exist also a Allgather method that gathers from all processes and distributes it to all processes[12].

MPI_Alltoall

All processors send same type and amount to each other[12].

A.3 Other functionality used

A.3.1 Data-types

Datatypes is used to make it easier to send areas of data between processes.

MPI_Type_vector

MPI_Type_vector takes in five parameters: count, blocklength, stride, oldtype and create a vector data-type. A vector data-type means a data-type that are blocks of equal data-type in strides. [12]

- count is number blocks in the new datatype
- blocklength is number of elements in each block
- stride is the number of elements between start of each block
- oldtype is the datatype of the elements
- newtype is the handle to the datatype

MPI_Type_commit

MPI_Type_commit takes in a data-type as a parameter and is called so that data-type can be used to communicate the the content of the matrices in timed_heat(Chapter C.1) with different addresses.[12]

A.3.2 Setting up dimensions

MPI_Dims_create are used to set up the dimensions in the code.

MPI_Dims_create

MPI_Dims_create takes in three parameters: nnodes, ndims and dims.

- nnodes is an integer and is the number of nodes in a grid.
- ndimes is an integer and is the number of Cartesian dimensions.
- dims is an integer array of size ndimes that specifies the number of nodes in each dimension

MPI_Dims_create helps to select a balanced distribution of processes in Cartesian grid there the dimensions are set to be as close to each other as possible. If a number in ndimes are set to be a positive integer before calling MPI_Dims_create that number will not be changed. Negative numbers will cause an error. [12]

A.3.3 Time measurement

Time measurement is taken using MPI_Wtime. MPI_Wtime is called right after initiation of MPI and right before finalization of MPI to create a time measure that include as much of the program as possible. MPI_Barrier(MPI_COMM_WORLD) is called before each MPI_Wtime call to ensure that all the ranks are at place in the program. Time from rank 0 is used in measurements since rank 0 is used as the master(see Chapter 1.6.4)

MPI_Wtime

Has no parameters and return the time since an arbitrary time in seconds as a floating-point number. Times returned are local to the different nodes that called them[12].

MPI_Barrier

MPI_Barrier takes in a communicator as a parameter and blocks until all processes in the communicator has called it[14, p. 122].

Appendix B

Pseudo Code

Algorithm 1 Serial version of the numerical solution

```
double[n+2][n+2] u_k1, u_k;
double c, delta_t, delta_s;
Initalize u_k1, u_k with initial values
for all steps do
    for i=0;i<n;i++ do
        for j=0;j<n;j++ do
            u_k1 = u_k + c × (Δt/(Δs)²) × (u_k[i+1, j] + u_k[i−1, j]+
                   u_k[i, j+1] + u_k[i, j−1] − 4 × u_k[i, j])
        end for
    end for
    Update boundary conditions
    Swap u_k and u_k1
end for
```

Algorithm 2 main method

```
1: function MAIN(argc  argv)
2:     Initialize MPI
3:     Start Timing
4:     Set up dimension
5:     Set up cart communication
6:     Find local_dims
7:     if SIZE is not evenly divided into dims[0] then
8:         Increment local_dims[0] for all ranks
9:         Find new SIZE for height
10:        Find computing area for ranks at the bottom for all the padding
11:    end if
12:    if SIZE is not evenly divided into dims[1] then
13:        Increment local_dims[1] for all ranks
14:        Find new SIZE for width
15:        Find computing area for ranks at the rightmost side for all the padding
16:    end if
17:    Set up and Initialize matrices
18:    Initialize values for local matrices
19:    Commit Vector types for border exchange
20:    for all steps do
21:        if step < CutOff then
22:            Set heated area to 100 degrees Celsius on local matrix
23:        end if
24:        if step%BORDER == 0 then
25:            BORDER_EXCHANGE(step)
26:        else
27:            BORDER_UPDATE(step)
28:        end if
29:        FTCS_SOLVER(step)
30:        BOUNDARIES(step)
31:        if step%SHAPSHOT == 0 then
32:            Create Filename for file to be printed
33:            COLLECT_AREA(step, filename)
34:        end if
35:    end for
36:    End Timing
37:    Free up memory
38:    Finalize MPI
39:    print out the result of the timing to screen
40: end function
```

Algorithm 3 Border exchange

```
1: function BORDER_EXCHANGE(step)
2:     Send my content of border size to the west and receive from the east
3:     Send my content of border size to the east and receive from the west
4:     Send my content of border size to the north and receive from the south
5:     Send my content of border size to the south and receive from the north
6: end function
```

Algorithm 4 FTCS_solver

```
1: function FTCS_SOLVER(step)
2:     for y = 0 → largest local y do
3:         for x = 0 → largest local x do
4:             Apply stencil for local_temp[step+1][y][x]
5:         end for
6:     end for
7: end function
```

Algorithm 5 Border update

```
1:  function BORDER_UPDATE(step)
2:      my = largest local y and mx = largest local x
3:      if has neighbor in the west then
4:          for x = step%BORDER − BORDER → x = −1 do
5:              for all y do
6:                  Apply stencil for local_temp[step][y][x]
7:              end for
8:          end for
9:      end if
10:     if has neighbor in the west then
11:         for x = mx + BORDER − step%BORDER → mx + 1 do
12:             for all y do
13:                 Apply stencil for local_temp[step][y][x]
14:             end for
15:         end for
16:     end if
17:     if has neighbor in the north then
18:         for y = step%BORDER − BORDER → −1 do
19:             for all x do
20:                 Apply stencil for local_temp[step][y][x]
21:             end for
22:         end for
23:     end if
24:     if has neighbor in the north then
25:         for y = my + BORDER − step%BORDER → my + 1 do
26:             for all x do
27:                 Apply stencil for local_temp[step][y][x]
28:             end for
29:         end for
30:     end if
31: end function
```

Algorithm 6 Collect Area

```
1:  function COLLECT_AREA(step, filename)
2:      Send local_temp data to rank 0
3:      if rank == 0 then
4:          receive and store data in temperature matrix
5:          if ( thenWRITETOFILE)
6:              open new out file with name filename
7:              write content of temperature matrix to out file
8:              close file
9:          end if
10:     end if
11:     Wait for rank 0 to finish
12: end function
```

Algorithm 7 Boundaries

```
1: function BOUNDARIES(step)
2:     if Rank has a west boundary then
3:         Apply stencil for local_temp[step+1][y][0] using local_temp[step][y][1]
4:         twice instead of local_temp[step][y][-1]
5:     end if
6:     if Rank has a east boundary then
7:         Apply stencil for local_temp[step+1][y][largest local x] using
8:         local_temp[step][y][largest local x - 1] twice instead of
9:         local_temp[step][y][largest local x + 1]
10:    end if
11:    if Rank has a north boundary then
12:        Apply stencil for local_temp[step+1][0][x] using local_temp[step][1][x]
13:        twice instead of local_temp[step][-1][x]
14:    end if
15:    if Rank has a south boundary then
16:        Apply stencil for local_temp[step+1][largest local y][x] using
17:        local_temp[step][largest local y - 1][x] twice instead of
18:        local_temp[step][largest local y + 1][x]
19:    end if
20:
21:    if my local_temp contains the 0, 0 corner of the whole system then
22:        Apply stencil for that corner using local_temp[step][1][0] and
23:        local_temp[step][0][1] twice
24:    end if
25:    if my local_temp contains the 0, SIZE-1 corner of the whole system then
26:        Apply stencil for that corner using local_temp[step][1][largest local x]
27:        and local_temp[step][0][largest local x - 1] twice
28:    end if
29:    if my local_temp contains the SIZE-1, 0 corner of the whole system then
30:        Apply stencil for that corner using local_temp[step][largest local y][1]
31:        and local_temp[step][largest local y - 1][0] twice
32:    end if
33:    if my local_temp contains the SIZE-1, SIZE-1 corner of the whole system
   then
34:        Apply stencil for that corner using
35:        local_temp[step][largest local y - 1][largest local x] and
36:        local_temp[step][largest local y][largest x - 1] twice
37:    end if
38: end function
```

Appendix C

Source Code

The following chapter includes the most important source code files.

C.1 Benchmarkingexample: Heat equation solved by FTCS

The Code is from TDT4200 Parallel Programming course fall 2010 and is written by Jan Christian Meyer with exception of methods that I have added that are update_border and logTime plus line 437 to 458 for reading arguments for the program and starting timing, 468 to 515 for padding the local areas, 561 to 563 for end time measuring and 570 to 574 for calling logTime method.

Listing C.1: Heat

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include <tgmath.h>
5  #include <mpi.h>
6
7  /*
8   * Physical quantities:
9   * k                    : thermal conductivity      [Watt / (meter Kelvin)]
10  * rho                  : density                   [kg / meter^3]
11  * cp                   : specific heat capacity    [kJ / (kg Kelvin)]
12  * rho * cp             : volumetric heat capacity  [Joule / (meter^3 Kelvin)]
13  * alpha = k / (rho*cp) : thermal diffusivity       [meter^2 / second]
14  *
15  * Mercury:
16  * cp = 0.140, rho = 13506, k = 8.69
17  * alpha = 8.69 / (0.140*13506) =~ 0.0619
18  *
19  * Copper:
20  * cp = 0.385, rho = 8960, k = 401
21  * alpha = 401.0 / (0.385 * 8960) =~ 0.12 [0.1162453618]
22  *
23  * Tin:
24  * cp = 0.227, k = 67, rho = 7300
25  * alpha = 67.0 / (0.227 * 7300) =~ 0.040
26  *
27  * Aluminium:
```

```
28   * cp = 0.897, rho = 2700, k = 237
29   * alpha = 237 / (0.897 * 2700) =~ 0.098 [0.097857054]
30   */
31  #define MERCURY 0.0619
32  #define COPPER 0.116
33  #define TIN 0.040
34  #define ALUMINIUM 0.098
35
36
37  /* Size of the computational grid - 256x256 square */
38  #define SIZE 256
39
40  /* Write to File 1=true and 0=false */
41  #define WRITETOFILE 0
42
43  /* Parameters of the simulation: how many steps, and when to cut off the heat */
44  #define NSTEPS 125000
45  #define CUTOFF 75000
46
47  /* How often to dump state to file (steps).
48   * 16 is realtime at 25fps, this is in 10x time
49   */
50  #define SNAPSHOT 160
51
52  /* Indexing macros for the global view on rank 0 */
53  #define TEMP(i,j) temperature[(i)*widthSIZE+(j)]
54
55  /* Test condition to see if a global coordinate is in my local area
56   * BOX is within main area
57   * BORBOX is whole local area
58   */
59  #define BOX(y,x) (                                          \
60        (y)>=local_origin[0] &&                               \
61        (y)<local_origin[0]+local_dims[0] &&                  \
62        (x)>=local_origin[1] &&                               \
63        (x)<local_origin[1]+local_dims[1]                     \
64  )
65  #define BORBOX(y,x)(                                                                \
66            (y)>=(local_origin[0]-border ) &&                       \
67        (y)<(local_origin[0]+local_dims[0]+border) &&             \
68        (x)>=(local_origin[1]- border ) &&                         \
69        (x)<(local_origin[1]+local_dims[1]+border )                                 \
70  )
71
72  /* Local material constant (LMAT) and temperature (LTEMP) indexing macros */
73  #define LMAT(i,j)    local_material[                          \
74           ((i)+border )*(local_dims[1]+2*border)+(j)+border     \
75  ]
76
77  #define LTEMP(s,i,j)     local_temp[((s)%2)][                 \
78       ((i)+border)*(local_dims[1]+2*border)+(j)+border          \
79  ]
80
81  /* Arrays for the simulation data */
82  float
83      *temperature,        // Temperature field (in global domain on rank 0)
84      *local_material,     // Local part of the material constants
85      *local_temp[2];      // Local part of the temperature (2 buffers)
86
87  /* Variables for time measurement */
88  double start, end;
89
90  /* Discretization: 5cm square cells, 2.5ms time intervals */
91  const float
92      h  = 5e-2, //was 5e-2
93      dt = 2.5e-3;
94
95  /* Local state */
96  int
97      size, rank,                     // World size, my rank
98      dims[2],                        // Size of the cartesian
```

```
99      periods[2] = { false, false },  // Periodicity of the cartesian
100     coords[2],                      // My coordinates in the cartesian
101     north, south, east, west,       // Neighbors in the cartesian
102     local_dims[2],                  // Size of local subdomain
103     local_origin[2],                // World coordinates of (0,0) local
104     local_realdims[2],                          // Computing area
105     padding[2] = {0,0},                                 // Size of padding added
106     border = 1,
107     systemSIZE = SIZE,
108     widthSIZE = SIZE,
109     heightSIZE = SIZE;
110
111 MPI_Comm cart;
112 MPI_Datatype
113     global_area, local_area,        // Vectors for collecting subdomains
114     border_row, border_col;         // Vectors for border exchange
115
116 void logTime( void );
117 void ftcs_solver ( int step );
118 void update_border( int step );
119 void boundaries ( int step );
120 void border_exchange ( int step );
121 void commit_vector_types ( void );
122 void external_heat ( int step );
123 void configure_geometry ( void );
124 void collect_area ( int step, char *filename );
125 void write_matrix ( FILE *out, float *data );
126
127
128 void
129 ftcs_solver ( int step )
130 {
131     /* The FTCS solution */
132     for ( int y=0; y<local_realdims[0]; y++ )
133         for ( int x=0; x<local_realdims[1]; x++ )
134             LTEMP(step+1,y,x) = LTEMP(step,y,x) + LMAT(y,x) * (
135                 (LTEMP(step,y-1,x) + LTEMP(step,y+1,x) +
136                  LTEMP(step,y,x-1) + LTEMP(step,y,x+1)) - 4.0*LTEMP(step,y,x)
137             );
138 }
139
140
141 void
142 boundaries ( int step )
143 {
144     /* The Neumann boundary condition */
145
146     /* my and mx are the largest y and x numbers in the main area of the local_temp
147        and local_material matrices */
148     int my = local_realdims[0]-1, mx = local_realdims[1]-1;
149
150     // I have a west boundary
151     if ( coords[1] == 0 ){
152         //Apply the stencil for the west column
153         for ( int i=0; i<local_realdims[0]; i++ )
154             LTEMP(step+1,i,0) = LTEMP(step,i,0) + LMAT(i,0) * (
155                 (2*LTEMP(step,i,1) + LTEMP(step,i-1,0) + LTEMP(step,i+1,0))
156                 - 4.0*LTEMP(step,i,0)
157             );
158     }
159
160     // I have an east boundary
161     if ( coords[1] == dims[1]-1 ){
162         //Apply the stencil for the east column
163         for ( int i=0; i<local_realdims[0]; i++ )
164             LTEMP(step+1,i,mx) = LTEMP(step,i,mx) + LMAT(i,mx) * (
165                 (2*LTEMP(step,i,mx-1) + LTEMP(step,i-1,mx) + LTEMP(step,i+1,mx))
166                 - 4.0*LTEMP(step,i,mx)
167             );
168     }
169
```

```
170     // I have a north boundary
171     if ( coords[0] == 0 ){
172         //Apply the stencil for the northern row
173         for ( int i=0; i<local_realdims[1]; i++ )
174             LTEMP(step+1,0,i) = LTEMP(step,0,i) + LMAT(0,i) * (
175                 2*LTEMP(step,1,i) + LTEMP(step,0,i-1) + LTEMP(step,0,i+1)
176                 - 4.0 * LTEMP(step,0,i)
177             );
178     }
179
180     // I have a south boundary
181     if ( coords[0] == dims[0]-1 ){
182         //Apply the stencil for the southern row
183         for ( int i=0; i<local_realdims[1]; i++ )
184             LTEMP(step+1,my,i) = LTEMP(step,my,i) + LMAT(my,i) * (
185                 2*LTEMP(step,my-1,i) + LTEMP(step,my,i-1) + LTEMP(step,my,i+1)
186                 - 4.0 * LTEMP(step,my,i)
187             );
188     }
189
190     //Apply the stencil for the corners
191     if ( BOX(0,0) )
192         LTEMP(step+1,0,0) = LTEMP(step,0,0) + LMAT(0,0) * (
193             2*LTEMP(step,1,0) + 2*LTEMP(step,0,1) - 4.0 * LTEMP(step,0,0)
194         );
195     if ( BOX(0,systemSIZE-1) )
196         LTEMP(step+1,0,mx) = LTEMP(step,0,mx) + LMAT(0,mx) * (
197             2*LTEMP(step,1,mx) + 2*LTEMP(step,0,mx-1) - 4.0 * LTEMP(step,0,mx)
198         );
199     if ( BOX(systemSIZE-1,0) )
200         LTEMP(step+1,my,0) = LTEMP(step,my,0) + LMAT(my,0) * (
201             2*LTEMP(step,my,1) + 2*LTEMP(step,my-1,0) - 4.0 * LTEMP(step,my,0)
202         );
203     if ( BOX(systemSIZE-1,systemSIZE-1) )
204         LTEMP(step+1,my,mx) = LTEMP(step,my,mx) + LMAT(my,mx) * (
205             2*LTEMP(step,my-1,mx)+2*LTEMP(step,my,mx-1) - 4.0*LTEMP(step,my,mx)
206         );
207 }
208
209 void
210 update_border ( int step ){
211         /* my and mx are the largest y and x numbers in the main area of
212     the local_temp and local_material matrices */
213     int my = local_realdims[0]-1, mx = local_realdims[1]-1;
214     int startPos = (step%border )-border ;
215         //Update west border if there is a neighbor in west
216         if(coords[1] != 0){
217                 //apply the stencil for the numbers in the border this must be done for
218         // the one columns closest to main area
219                 for(int x=startPos;x<0;x++){
220                         if(coords[0] == 0){
221                 //can not use north border for y=0
222                 LTEMP(step,0,x) = LTEMP(step-1,0,x) + LMAT(0,x) * (
223                         (2.0*LTEMP(step-1,1,x) + LTEMP(step-1,0,x-1) +
224                         LTEMP(step-1,0,x+1)) - 4.0*LTEMP(step-1,0,x)
225                 );
226             }else{
227                 LTEMP(step,0,x) = LTEMP(step,0,x) = LTEMP(step-1,0,x) + LMAT(0,x) * (
228                         (LTEMP(step-1,-1,x) + LTEMP(step-1,1,x) +
229                         LTEMP(step-1,0,x-1) + LTEMP(step-1,0,x+1)) - 4.0*LTEMP(step-1,0,x)
230                 );
231             }
232             if(coords[0] == (dims[0]-1)){
233                 //can not use south border for y=my since my+1 is wrong
234                 LTEMP(step, my,x) = LTEMP(step-1,my,x) + LMAT(my,x) * (
235                         (2.0*LTEMP(step-1,my-1,x) +
236                         LTEMP(step-1,my,x-1) + LTEMP(step-1,my,x+1)) - 4.0*LTEMP(step-1,my,x)
237                 );
238             }else{
239                 LTEMP(step,my,x) = LTEMP(step-1,my,x) + LMAT(my,x) * (
240                         (LTEMP(step-1,my-1,x) + LTEMP(step-1,my+1,x) +
```

```
241                         LTEMP(step-1,my,x-1) + LTEMP(step-1,my,x+1)) - 4.0*LTEMP(step-1,my,x)
242                 );
243             }
244                         for ( int y=1; y<my; y++ ){
245                 LTEMP(step,y,x) = LTEMP(step-1,y,x) + LMAT(y,x) * (
246                         (LTEMP(step-1,y-1,x) + LTEMP(step-1,y+1,x) +
247                         LTEMP(step-1,y,x-1) + LTEMP(step-1,y,x+1)) - 4.0*LTEMP(step-1,y,x)
248                 );
249             }
250                 }
251
252         }
253
254
255         //Update eastborder if there is a neighboor in east
256         if(coords[1] != (dims[1]-1)){
257                 for(int x=(mx-startPos); x>mx; x--){
258                         if(coords[0] == 0){
259                 //can not use north border for y=0
260                 LTEMP(step,0,x) = LTEMP(step-1,0,x) + LMAT(0,x) * (
261                         (2.0*LTEMP(step-1,1,x) + LTEMP(step-1,0,x-1) +
262                         LTEMP(step-1,0,x+1)) - 4.0*LTEMP(step-1,0,x)
263                 );
264             }else{
265                 LTEMP(step,0,x) = LTEMP(step,0,x) = LTEMP(step-1,0,x) + LMAT(0,x) * (
266                         (LTEMP(step-1,-1,x) + LTEMP(step-1,1,x) +
267                         LTEMP(step-1,0,x-1) + LTEMP(step-1,0,x+1)) - 4.0*LTEMP(step-1,0,x)
268                 );
269             }
270             if(coords[0] == (dims[0]-1)){
271                 //can not use south border for y=my since my+1 is wrong
272                 LTEMP(step, my,x) = LTEMP(step-1,my,x) + LMAT(my,x) * (
273                         (2.0*LTEMP(step-1,my-1,x) +
274                         LTEMP(step-1,my,x-1) + LTEMP(step-1,my,x+1)) - 4.0*LTEMP(step-1,my,x)
275                 );
276             }else{
277                 LTEMP(step,my,x) = LTEMP(step-1,my,x) + LMAT(my,x) * (
278                         (LTEMP(step-1,my-1,x) + LTEMP(step-1,my+1,x) +
279                         LTEMP(step-1,my,x-1) + LTEMP(step-1,my,x+1)) - 4.0*LTEMP(step-1,my,x)
280                 );
281             }
282                         for ( int y=1; y<my; y++ )
283                                 LTEMP(step,y,x) = LTEMP(step-1,y,x) + LMAT(y,x) * (
284                         (LTEMP(step-1,y-1,x) + LTEMP(step-1,y+1,x) +
285                         LTEMP(step-1,y,x-1) + LTEMP(step-1,y,x+1)) - 4.0*LTEMP(step-1,y,x)
286                 );
287                 }
288
289         }
290
291
292         //Update northborder if there is a neighboor in north
293         if(coords[0] != 0){
294                 for(int y=startPos;y<0;y++){
295                         if(coords[1] == 0){
296                                 //can not use westborder for x=0
297                                 LTEMP(step,y,0) = LTEMP(step-1,y,0) + LMAT(y,0) * (
298                         (LTEMP(step-1,y-1,0) + LTEMP(step-1,y+1,0) +
299                         2.0*LTEMP(step-1,y,1)) - 4.0*LTEMP(step-1,y,0)
300                 );
301                         }else{
302                                 LTEMP(step,y,0) = LTEMP(step-1,y,0) + LMAT(y,0) * (
303                         (LTEMP(step-1,y-1,0) + LTEMP(step-1,y+1,0) +
304                         LTEMP(step-1,y,-1) + LTEMP(step-1,y,1)) - 4.0*LTEMP(step-1,y,0)
305                 );
306                         }
307                         if(coords[1] == (dims[1]-1)){
308                                 //can not use eastborder for x=mx
309                                 LTEMP(step,y,mx) = LTEMP(step-1,y,mx) + LMAT(y,mx) * (
310                         (LTEMP(step-1,y-1,mx) + LTEMP(step-1,y+1,mx) +
311                         2.0*LTEMP(step-1,y,mx-1)) - 4.0*LTEMP(step-1,y,mx)
```

```
312                     );
313                             }else{
314                                     LTEMP(step,y,mx) = LTEMP(step-1,y,mx) + LMAT(y,mx) * (
315                             (LTEMP(step-1,y-1,mx) + LTEMP(step-1,y+1,mx) +
316                             LTEMP(step-1,y,mx-1) + LTEMP(step-1,y,mx+1)) - 4.0*LTEMP(step-1,y,mx)
317                     );
318                             }
319                             for(int x=1;x<mx;x++){
320                                     LTEMP(step,y,x) = LTEMP(step-1,y,x) + LMAT(y,x) * (
321                             (LTEMP(step-1,y-1,x) + LTEMP(step-1,y+1,x) +
322                             LTEMP(step-1,y,x-1) + LTEMP(step-1,y,x+1)) - 4.0*LTEMP(step-1,y,x)
323                     );
324                             }
325                     }
326             }
327
328
329             //Update southborder if there is a neighboor in south
330             if(coords[0] != (dims[0]-1)){
331                     for(int y=(my-startPos);y>my; y--){
332                             if(coords[1] == 0){
333                                     //can not use westborder for x=0
334                                     LTEMP(step,y,0) = LTEMP(step-1,y,0) + LMAT(y,0) * (
335                             (LTEMP(step-1,y-1,0) + LTEMP(step-1,y+1,0) +
336                             2.0*LTEMP(step-1,y,1)) - 4.0*LTEMP(step-1,y,0)
337                     );
338                             }else{
339                                     LTEMP(step,y,0) = LTEMP(step-1,y,0) + LMAT(y,0) * (
340                             (LTEMP(step-1,y-1,0) + LTEMP(step-1,y+1,0) +
341                             LTEMP(step-1,y,-1) + LTEMP(step-1,y,1)) - 4.0*LTEMP(step-1,y,0)
342                     );
343                             }
344                             if(coords[1] == (dims[1]-1)){
345                                     //can not use eastborder for x=mx
346                                     LTEMP(step,y,mx) = LTEMP(step-1,y,mx) + LMAT(y,mx) * (
347                             (LTEMP(step-1,y-1,mx) + LTEMP(step-1,y+1,mx) +
348                             2.0*LTEMP(step-1,y,mx-1)) - 4.0*LTEMP(step-1,y,mx)
349                     );
350                             }else{
351                                     LTEMP(step,y,mx) = LTEMP(step-1,y,mx) + LMAT(y,mx) * (
352                             (LTEMP(step-1,y-1,mx) + LTEMP(step-1,y+1,mx) +
353                             LTEMP(step-1,y,mx-1) + LTEMP(step-1,y,mx+1)) - 4.0*LTEMP(step-1,y,mx)
354                     );
355                             }
356                             for(int x=1;x<mx;x++){
357                                     LTEMP(step,y,x) = LTEMP(step-1,y,x) + LMAT(y,x) * (
358                             (LTEMP(step-1,y-1,x) + LTEMP(step-1,y+1,x) +
359                             LTEMP(step-1,y,x-1) + LTEMP(step-1,y,x+1)) - 4.0*LTEMP(step-1,y,x)
360                     );
361                             }
362
363                     }
364             }
365     }
366
367     void
368     commit_vector_types ( void )
369     {
370         MPI_Type_vector ( heightSIZE/dims[0], local_dims[1], dims[1]*local_dims[1],
371             MPI_FLOAT, &global_area
372         );
373         MPI_Type_vector ( heightSIZE/dims[0], local_dims[1], local_dims[1]+2*border ,
374             MPI_FLOAT, &local_area
375         );
376         MPI_Type_commit ( &local_area );
377         MPI_Type_commit ( &global_area );
378
379         /* Commit the types for the border exchange */
380         MPI_Type_vector ( border , local_dims[1]+2*border, local_dims[1]+2*border ,
381             MPI_FLOAT, &border_row
382         );
```

```
383     MPI_Type_vector ( local_dims[0], border , local_dims[1]+2*border ,
384         MPI_FLOAT, &border_col
385     );
386     MPI_Type_commit ( &border_row );
387     MPI_Type_commit ( &border_col );
388 }
389
390
391 void
392 border_exchange ( int step )
393 {
394     /* east -> me -> west */
395     MPI_Sendrecv (
396         &LTEMP(step,0,0), 1, border_col, west, 0,
397         &LTEMP(step,0,local_dims[1]), 1, border_col, east, 0,
398         cart, MPI_STATUS_IGNORE
399     );
400     /* west -> me -> east */
401     MPI_Sendrecv (
402         &LTEMP(step,0,local_dims[1]-border ), 1, border_col, east, 0,
403         &LTEMP(step,0,-border ), 1, border_col, west, 0,
404         cart, MPI_STATUS_IGNORE
405     );
406     /* south -> me -> north */
407     MPI_Sendrecv (
408         &LTEMP(step,0,-border ), 1, border_row, north, 0,
409         &LTEMP(step,local_dims[0],-border ), 1, border_row, south, 0,
410         cart, MPI_STATUS_IGNORE
411     );
412     /* north -> me -> south */
413     MPI_Sendrecv (
414         &LTEMP(step,local_dims[0]-border ,-border ), 1, border_row, south, 0,
415         &LTEMP(step,-border,-border), 1, border_row, north, 0,
416         cart, MPI_STATUS_IGNORE
417     );
418 }
419
420 void logTime( void){
421         char* filename= "log.txt";
422         FILE *out = fopen ( filename, "a" );
423         if (out == NULL)
424         {
425             printf("Error opening file!\n");
426             exit(1);
427         }
428         fprintf ( out, "%f;%d;%d;%d;%d;%d;%d;%d;%d;%d;\n", end-start, systemSIZE,
429                   border, NSTEPS, CUTOFF, size, dims[0], dims[1], WRITETOFILE);
430         fclose ( out );
431
432 }
433
434 int
435 main ( int argc, char **argv )
436 {
437     MPI_Init ( &argc, &argv );
438     MPI_Comm_size ( MPI_COMM_WORLD, &size );
439     MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
440
441     if(argc > 1){
442         if((atoi(argv[1])%256)==0){
443                 systemSIZE = atoi(argv[1]);
444                 widthSIZE = systemSIZE;
445                 heightSIZE = systemSIZE;
446         }else{
447                 if(rank == 0)
448                         printf("Size error %s is not diviable by 256\n", argv[1]);
449         }
450         if(atoi(argv[2])<=10){
451                         border = atoi(argv[2]);
452         }else{
453                 if(rank == 0)
```

```
454                         printf("%s is too large as a border\n", argv[2]);
455         }
456
457     }
458     /* Start timing */
459     MPI_Barrier(MPI_COMM_WORLD);
460     start = MPI_Wtime();
461
462
463     MPI_Dims_create ( size, 2, dims );
464     MPI_Cart_create ( MPI_COMM_WORLD, 2, dims, periods, 0, &cart );
465     MPI_Cart_coords ( cart, rank, 2, coords );
466     MPI_Cart_shift ( cart, 0, 1, &north, &south );
467     MPI_Cart_shift ( cart, 1, 1, &west, &east );
468
469     local_dims[0] = local_realdims[0] = systemSIZE / dims[0];
470     local_dims[1] = local_realdims[1] = systemSIZE / dims[1];
471
472     //Pad the matrixes so they are diviable by the dimensions
473         if((local_dims[0]*dims[0])!= systemSIZE){
474         //padding to make it bigger
475         local_realdims[0] += 1;
476                 local_dims[0] += 1;
477
478         heightSIZE = local_dims[0]*dims[0];
479
480         padding[0] = heightSIZE - systemSIZE;
481
482         if(coords[0] == (dims[0]-1)){
483                 //threads at the bottom should not compute the padding
484                 local_realdims[0] -= padding[0];
485                 if(padding[0]> local_dims[0]){
486                         local_realdims[0] = 0;
487                 }
488         }
489         if(padding[0]> local_dims[0]){
490                 //fix padding bigger than local_dims[0] problem
491                 if((padding[0]-(local_dims[0]*(dims[0]-coords[0]))) > 0){
492                         local_realdims[0] -= padding[0]-(local_dims[0]*(dims[0]-coords[0]));
493                         if(local_realdims[0]<0)
494                                 local_realdims[0] = 0;
495                 }
496         }
497     }
498     if((local_dims[1]*dims[1])!= systemSIZE){
499         //padding to make it bigger
500         local_realdims[1] += 1;
501                 local_dims[1] += 1;
502
503         widthSIZE = local_dims[1]*dims[1];
504
505         padding[1] = widthSIZE - systemSIZE;
506
507         if(coords[1] == (dims[1]-1)){
508                 //rightmost threads should not compute the padding
509                 local_realdims[1] -= padding[1];
510         }
511         if(padding[1]> local_dims[1]){
512                 //fix padding bigger than local_dims[0] problem
513                 if((padding[1]-(local_dims[1]*(dims[1]-coords[1]))) > 0){
514                         local_realdims[1] -= padding[1]-(local_dims[1]*(dims[1]-coords[1]));
515                         if(local_realdims[1]<0)
516                                 local_realdims[1] = 0;
517                 }
518         }
519     }
520
521     local_origin[0] = coords[0]*local_dims[0];
522     local_origin[1] = coords[1]*local_dims[1];
523
524     size_t lsize_full = (local_dims[0]+2*border)*(local_dims[1]+2*border);
```

```
525     local_material = malloc ( lsize_full * sizeof(float) );
526     local_temp[0] = malloc ( lsize_full * sizeof(float) );
527     local_temp[1] = malloc ( lsize_full * sizeof(float) );
528
529     if ( rank == 0 ){
530         temperature = calloc(widthSIZE*heightSIZE, sizeof(float));
531     }
532
533     commit_vector_types (); //Commit Vector types for borderexchange
534     configure_geometry(); //Set up the LMAT and LTEMP
535
536     /* Main integration loop: NSTEPS iterations, impose external heat
537      * until CUTOFF iterations have passed
538      */
539      MPI_Barrier(MPI_COMM_WORLD);
540     /* Imposed temperature from outside */
541     for ( int step=0; step<NSTEPS; step++ )
542     {
543         if ( step < CUTOFF )
544             external_heat ( step );
545
546         if((step%border)==0){
547                 //Exchange the border every borderth step
548                 border_exchange ( step );
549         }else{
550                 //Need to be updated so the FTCS solver can use the row or column closest to itself
551                 update_border( step );
552         }
553         ftcs_solver ( step );
554         boundaries ( step );
555
556         if((step % SNAPSHOT) == 0 )
557         {
558             char filename[15];
559             sprintf ( filename, "data/%.4d.dat", step/SNAPSHOT );
560             collect_area ( step, filename );
561         }
562
563     }
564
565     /* End timing */
566     MPI_Barrier(MPI_COMM_WORLD);
567     end = MPI_Wtime();
568
569     if ( rank == 0 )
570         free (temperature);
571     free (local_material), free (local_temp[0]), free (local_temp[1]);
572     MPI_Finalize();
573
574     /* Print out timing */
575     if (rank == 0) {
576         printf("RUNTIME %4.6f SIZE %d NSTEPS %d CUTOFF%d THREADS %d in y %d x %d\n",
577                 end-start, systemSIZE, NSTEPS, CUTOFF, size, dims[0], dims[1]);
578             logTime();
579     }
580
581     exit ( EXIT_SUCCESS );
582 }
583
584
585 void
586 external_heat ( int step )
587 {
588     /* Imposed temperature from outside */
589     for ( int y=(systemSIZE/2)-(systemSIZE/16); y<=(systemSIZE/2)+(systemSIZE/16); y++ )
590         for ( int x=(systemSIZE/4); x<=(3*systemSIZE/4); x++ )
591         {
592             if ( BORBOX(y,x) )
593                 LTEMP ( step, y-local_origin[0], x-local_origin[1] ) = 100.0;
594         }
595 }
```

```
596
597
598 void
599 configure_geometry ( void )
600 {
601     /* Initialization: fill the pool with mercury */
602     for ( int y=-border; y<(local_dims[0]+border); y++ )
603     {
604         for ( int x=-border; x<(local_dims[1]+border); x++ )
605         {
606             LMAT(y,x) =  MERCURY * (dt/(h*h));
607             LTEMP(1,y,x) = LTEMP(0,y,x) = 20.0;
608         }
609     }
610
611     /* Set up the two blocks of copper and tin */
612     for ( int y=(systemSIZE/8); y<(3*systemSIZE/8); y++ )
613         for ( int x=(systemSIZE/8); x<(systemSIZE/2)-(systemSIZE/8); x++ )
614         {
615             if ( BORBOX(y,x) )
616             {
617                 LMAT(y-local_origin[0],x-local_origin[1]) =
618                     COPPER * (dt/(h*h));
619                 LTEMP(0,y-local_origin[0],x-local_origin[1]) = 60.0;
620             }
621             if ( BORBOX(y,systemSIZE-x) )
622             {
623                 LMAT(y-local_origin[0],(systemSIZE-x)-local_origin[1]) =
624                     TIN * (dt/(h*h));
625                 LTEMP(0,y-local_origin[0],(systemSIZE-x)-local_origin[1]) = 60.0;
626             }
627
628         }
629     /* Set up the heating element in the middle */
630     for ( int y=(systemSIZE/2)-(systemSIZE/16); y<=(systemSIZE/2)+(systemSIZE/16); y++ )
631         for ( int x=(systemSIZE/4); x<=(3*systemSIZE/4); x++ )
632         {
633             if ( BORBOX(y,x) )
634                 LMAT(y-local_origin[0],x-local_origin[1]) =
635                     ALUMINIUM * (dt/(h*h));
636         }
637 }
638
639
640 void
641 collect_area ( int step, char *filename )
642 {
643     MPI_Request req;
644     MPI_Isend ( &LTEMP((step%SNAPSHOT),0,0), 1, local_area, 0, 0, cart, &req );
645     if ( rank == 0 )
646     {
647         int co[2];
648         for ( int r=0; r<size; r++ )
649         {
650             MPI_Cart_coords ( cart, r, 2, co );
651             MPI_Recv (
652                 &TEMP( co[0]*local_dims[0], co[1]*local_dims[1] ),
653                     1, global_area, r, 0, cart, MPI_STATUS_IGNORE
654             );
655         }
656         if(WRITETOFILE){
657                 FILE *out = fopen ( filename, "w" );
658                 write_matrix ( out, temperature );
659                 fclose ( out );
660                 printf ( "Snapshot at step %d\n", step );
661         }
662     }
663     MPI_Wait ( &req, MPI_STATUS_IGNORE );
664 }
665
666
```

```
667 void
668 write_matrix ( FILE *out, float *data )
669 {
670     float size = (float) systemSIZE;
671     fwrite ( &size, sizeof(float), 1, out );
672     for ( float x=0; x<systemSIZE; x+=1.0 )
673         fwrite ( &x, sizeof(float), 1, out );
674     for ( int y=0; y<systemSIZE; y++ )
675     {
676         float len = (float) y;
677         fwrite ( &len, sizeof(float), 1, out );
678         fwrite ( &data[y*widthSIZE], sizeof(float), systemSIZE, out );
679     }
680 }
```

C.2 Heat equation solved by FTCS serial version

This code is a simplification of timed_heat (Appendix C.1) this is run on the systems to get the timing with 1 process.

Listing C.2: Serial Heat

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include <tgmath.h>
5  #include <mpi.h>
6
7  /*
8   * Physical quantities:
9   * k                    : thermal conductivity      [Watt / (meter Kelvin)]
10  * rho                  : density                   [kg / meter^3]
11  * cp                   : specific heat capacity    [k] / (kg Kelvin)]
12  * rho * cp             : volumetric heat capacity  [Joule / (meter^3 Kelvin)]
13  * alpha = k / (rho*cp) : thermal diffusivity       [meter^2 / second]
14  *
15  * Mercury:
16  * cp = 0.140, rho = 13506, k = 8.69
17  * alpha = 8.69 / (0.140*13506) =~ 0.0619
18  *
19  * Copper:
20  * cp = 0.385, rho = 8960, k = 401
21  * alpha = 401.0 / (0.385 * 8960) =~ 0.12 [0.1162453618]
22  *
23  * Tin:
24  * cp = 0.227, k = 67, rho = 7300
25  * alpha = 67.0 / (0.227 * 7300) =~ 0.040
26  *
27  * Aluminium:
28  * cp = 0.897, rho = 2700, k = 237
29  * alpha = 237 / (0.897 * 2700) =~ 0.098 [0.097857054]
30  */
31 #define MERCURY 0.0619
32 #define COPPER 0.116
33 #define TIN 0.040
34 #define ALUMINIUM 0.098
35
36
37 /* Size of the computational grid - 256x256 square */
38 #define SIZE 256 //was 256
39
40 /* Write to File 1=true and 0=false */
41 #define WRITETOFILE 0
42
43 /* Parameters of the simulation: how many steps, and when to cut off the heat */
44 #define NSTEPS 125000 //was 125000
```

```
45  #define CUTOFF 75000 //was 75000
46
47  /* How often to dump state to file (steps).
48   * 16 is realtime at 25fps, this is in 10x time
49   */
50  #define SNAPSHOT 160 //was 160
51
52  /* Local material constant (LMAT) and temperature (LTEMP) indexing macros */
53  #define MAT(i,j)    material[                                   \
54          ((i)*systemSIZE)+(j)          \
55  ]
56
57  #define TEMP(s,i,j)     temperature[((s)%2)][((i)*systemSIZE)+(j)]
58
59  /* Arrays for the simulation data */
60  float
61      *temperature[2],        // Temperature field (in global domain on rank 0)
62      *material;              // Local part of the material constants
63
64  /* Variables for time measurement */
65  double start, end;
66
67  int systemSIZE = SIZE;
68
69  /* Discretization: 5cm square cells, 2.5ms time intervals */
70  const float
71      h  = 5e-2,
72      dt = 2.5e-3;
73
74  void logTime(void);
75  void ftcs_solver ( int step );
76  void external_heat ( int step );
77  void configure_geometry ( void );
78  void collect_area ( int step, char *filename );
79  void write_matrix ( FILE *out, float *data );
80
81
82  void
83  ftcs_solver ( int step )
84  {
85      /* my and mx are the largest y and x numbers in the main area of the local_temp and local_material matrices */
86      int my = systemSIZE-1, mx = systemSIZE-1;
87
88      /* The FTCS solution */
89      for ( int y=1; y<my; y++ ){
90          for ( int x=1; x<mx; x++ ){
91              TEMP(step+1,y,x) = TEMP(step,y,x) + MAT(y,x) * (
92                  (TEMP(step,y-1,x) + TEMP(step,y+1,x) +
93                   TEMP(step,y,x-1) + TEMP(step,y,x+1)) - 4.0*TEMP(step,y,x)
94              );
95          }
96      }
97      for ( int i=1; i<my; i++ )
98          TEMP(step+1,i,0) = TEMP(step,i,0) + MAT(i,0) * (
99              (2*TEMP(step,i,1) + TEMP(step,i-1,0) + TEMP(step,i+1,0))
100             - 4.0*TEMP(step,i,0)
101         );
102     for ( int i=1; i<my; i++ )
103             TEMP(step+1,i,mx) = TEMP(step,i,mx) + MAT(i,mx) * (
104                 (2*TEMP(step,i,mx-1) + TEMP(step,i-1,mx) + TEMP(step,i+1,mx))
105                 - 4.0*TEMP(step,i,mx)
106             );
107     for ( int i=1; i<mx; i++ )
108         TEMP(step+1,0,i) = TEMP(step,0,i) + MAT(0,i) * (
109             2*TEMP(step,1,i) + TEMP(step,0,i-1) + TEMP(step,0,i+1)
110             - 4.0 * TEMP(step,0,i)
111         );
112     for ( int i=1; i<mx; i++ )
113         TEMP(step+1,my,i) = TEMP(step,my,i) + MAT(my,i) * (
114             2*TEMP(step,my-1,i) + TEMP(step,my,i-1) + TEMP(step,my,i+1)
115             - 4.0 * TEMP(step,my,i)
```

```
116             );
117
118         TEMP(step+1,0,0) = TEMP(step,0,0) + MAT(0,0) * (
119             2*TEMP(step,1,0) + 2*TEMP(step,0,1) - 4.0 * TEMP(step,0,0)
120         );
121         TEMP(step+1,0,mx) = TEMP(step,0,mx) + MAT(0,mx) * (
122                 2*TEMP(step,1,mx) + 2*TEMP(step,0,mx-1) - 4.0 * TEMP(step,0,mx)
123         );
124         TEMP(step+1,my,0) = TEMP(step,my,0) + MAT(my,0) * (
125             2*TEMP(step,my,1) + 2*TEMP(step,my-1,0) - 4.0 * TEMP(step,my,0)
126         );
127         TEMP(step+1,my,mx) = TEMP(step,my,mx) + MAT(my,mx) * (
128             2*TEMP(step,my-1,mx)+2*TEMP(step,my,mx-1) - 4.0*TEMP(step,my,mx)
129         );
130     }
131
132     void logTime( void){
133             char* filename= "log.txt";
134             FILE *out = fopen ( filename, "a" );
135             if (out == NULL)
136             {
137                 printf("Error opening file!\n");
138                 exit(1);
139             }
140             fprintf ( out, "Runtime = %f systemSIZE %d NSTEPS %d CUTOFF%d THREADS %d in " +
141                       "y=%d, x=%d WriteTofile(%d)\n", end-start, systemSIZE, NSTEPS,
142                       CUTOFF, 1, 1, 1, WRITETOFILE);
143             fclose ( out );
144
145     }
146
147     int
148     main ( int argc, char **argv )
149     {
150         MPI_Init ( &argc, &argv );
151         if(argc > 1){
152             int argv1 = atoi(argv[1]);
153             if((argv1%256) == 0){
154                 systemSIZE = argv1;
155             }
156         }
157
158         start = MPI_Wtime();
159         material = malloc ( systemSIZE * systemSIZE * sizeof(float) );
160         temperature[0] = malloc(systemSIZE*systemSIZE * sizeof(float));
161         temperature[1] = malloc(systemSIZE*systemSIZE * sizeof(float));
162
163         configure_geometry(); //Set up the LMAT and LTEMP
164
165         /* Main integration loop: NSTEPS iterations, impose external heat
166          * until CUTOFF iterations have passed
167          */
168         /* Imposed temperature from outside */
169         for ( int step=0; step<NSTEPS; step++ )
170         {
171             if ( step < CUTOFF )
172                 external_heat ( step );
173             ftcs_solver ( step );
174             if((step % SNAPSHOT) == 0 )
175             {
176                 char filename[15];
177                 sprintf ( filename, "data/%.4d.dat", step/SNAPSHOT );
178                 collect_area ( step, filename );
179             }
180         }
181
182         /* End timing */
183         end = MPI_Wtime();
184
185         free (temperature[0]);
186         free (temperature[1]);
```

```
187     free (material);
188
189     printf("RUNTIME␣%4.6f␣systemSIZE␣%d␣NSTEPS␣%d␣CUTOFF%d␣THREADS␣%d␣in␣y␣%d␣x␣%d\n",
190             end-start, systemSIZE, NSTEPS, CUTOFF, 1, 1, 1);
191     //logTime();
192     MPI_Finalize();
193     exit ( EXIT_SUCCESS );
194 }
195
196
197 void
198 external_heat ( int step )
199 {
200     /* Imposed temperature from outside */
201     for ( int y=(systemSIZE/2)-(systemSIZE/16); y<=(systemSIZE/2)+(systemSIZE/16); y++ )
202         for ( int x=(systemSIZE/4); x<=(3*systemSIZE/4); x++ )
203         {
204             TEMP ( step, y, x) = 100.0;
205         }
206 }
207
208
209 void
210 configure_geometry ( void )
211 {
212     /* Initialization: fill the pool with mercury */
213     for ( int y=0; y<systemSIZE; y++ )
214     {
215         for ( int x=0; x<systemSIZE; x++ )
216         {
217             MAT(y,x) =  MERCURY * (dt/(h*h));
218             TEMP(1,y,x) = TEMP(0,y,x) = 20.0;
219         }
220     }
221
222     /* Set up the two blocks of copper and tin */
223     for ( int y=(systemSIZE/8); y<(3*systemSIZE/8); y++ )
224         for ( int x=(systemSIZE/8); x<(systemSIZE/2)-(systemSIZE/8); x++ )
225         {
226             MAT(y,x) = COPPER * (dt/(h*h));
227             TEMP(0,y,x) = 60.0;
228
229             MAT(y,(systemSIZE-x)) = TIN * (dt/(h*h));
230             TEMP(0,y,(systemSIZE-x)) = 60.0;
231         }
232     /* Set up the heating element in the middle */
233     for ( int y=(systemSIZE/2)-(systemSIZE/16); y<=(systemSIZE/2)+(systemSIZE/16); y++ )
234         for ( int x=(systemSIZE/4); x<=(3*systemSIZE/4); x++ )
235         {
236                 MAT(y,x) = ALUMINIUM * (dt/(h*h));
237         }
238 }
239
240
241 void
242 collect_area ( int step, char *filename )
243 {
244     if((WRITETOFILE == 1)){
245         //FILE *out = fopen ( filename, "w" );
246         FILE *out = fopen ( "testwrite.dat", "w" );
247         write_matrix ( out, temperature[((step)%2)] );
248         fclose ( out );
249         printf ( "Snapshot␣at␣step␣%d\n", step );
250     }
251 }
252
253
254 void
255 write_matrix ( FILE *out, float *data )
256 {
257     float size = (float) systemSIZE;
```

```
258         fwrite ( &size, sizeof(float), 1, out );
259         for ( float x=0; x<systemSIZE; x+=1.0 )
260             fwrite ( &x, sizeof(float), 1, out );
261         for ( int y=0; y<systemSIZE; y++ )
262         {
263             float len = (float) y;
264             fwrite ( &len, sizeof(float), 1, out );
265             fwrite ( &data[y*systemSIZE], sizeof(float), systemSIZE, out );
266         }
267     }
```

Appendix D

Node Layouts for Clustis3

In this chapter which node each of the processors used when running on Clustis3 are listed. The ranks are distributed so that the first is rank 0, the next is rank 1 like:

rank 0	rank 1	rank 2	...	rank x-1
rank x	rank x+1	...	...	rank 2x-1
...				...
rank (y-1)x	rank (y-1)x + 1	...	rank n-2	rank n-1

where n is the number of processes, y is the dimension height and x is the dimension width so that n = x $\times$ y.

D.1 Node Layout First Run

D.1.1 9 Processes

9 processes has a layout that are 3 processes high and 3 processes wide.

6	7	6
7	6	7
6	7	6

D.1.2 10 Processes

10 processes has a layout that are 5 processes high and 2 processes wide.

6	7
6	7
6	7
6	7
6	7

D.1.3 11 Processes

11 processes has a layout that are 11 processes high and 1 processes wide.

6	7	6	7	6	7	6	7	6	7	6

D.1.4 12 Processes

12 processes has a layout that are 4 processes high and 3 processes wide.

6	7	6
7	6	7
6	7	6
7	6	7

D.1.5 13 Processes

13 processes has a layout that are 13 processes high and 1 processes wide.

6	7	6	7	6	7	6	7	6	7	6	7	6

D.1.6 14 Processes

14 processes has a layout that are 7 processes high and 2 processes wide.

6	7
6	7
6	7
6	7
6	7
6	7
6	7

D.1.7 15 Processes

15 processes has a layout that are 5 processes high and 3 processes wide.

6	7	6
7	6	7
6	7	6
7	6	7
6	7	6

D.1.8 16 Processes

16 processes has a layout that are 4 processes high and 4 processes wide.

6	7	6	7
6	7	6	7
6	7	6	7
6	7	6	7

D.1.9 17 Processes

17 processes has a layout that are 17 processes high and 1 processes wide.

0	4	5	0	4	5	0	4	5	0	4	5	0	4	5	0	4

D.1.10 18 Processes

18 processes has a layout that are 6 processes high and 3 processes wide.

0	4	5
0	4	5
0	4	5
0	4	5
0	4	5
0	4	5

D.1.11 19 Processes

19 processes has a layout that are 19 processes high and 1 processes wide.

0	4	5	0	4	5	0	4	5	0	4	5	0	4	5	0	4	5	0

D.1.12 20 Processes

20 processes has a layout that are 5 processes high and 4 processes wide.

0	4	5	0
4	5	0	4
5	0	4	5
0	4	5	0
4	5	0	4

D.1.13 21 Processes

21 processes has a layout that are 7 processes high and 3 processes wide.

0	4	5
0	4	5
0	4	5
0	4	5
0	4	5
0	4	5
0	4	5

D.1.14 22 Processes

22 processes has a layout that are 11 processes high and 2 processes wide.

0	4
5	0
4	5
0	4
5	0
4	5
0	4
5	0
4	5
0	4
5	0

D.1.15 23 Processes

23 processes has a layout that are 23 processes high and 1 processes wide.

0	4	5	0	4	5	0	4	5	0	4	5	0	4	5	0	4	5	0	4	5	0	4

D.1.16 24 Processes

24 processes has a layout that are 6 processes high and 4 processes wide.

0	4	5	0
4	5	0	4
5	0	4	5
0	4	5	0
4	5	0	4
5	0	4	5

D.1.17 25 Processes

25 processes has a layout that are 5 processes high and 5 processes wide.

4	5	6	7	4
5	6	7	4	5
6	7	4	5	6
7	4	5	6	7
4	5	6	7	4

D.1.18 26 Processes

26 processes has a layout that are 13 processes high and 2 processes wide.

4	5
6	7
4	5
6	7
4	5
6	7
4	5
6	7
4	5
6	7
4	5
6	7
4	5

D.1.19 27 Processes

27 processes has a layout that are 9 processes high and 3 processes wide.

4	5	6
7	4	5
6	7	4
5	6	7
4	5	6
7	4	5
6	7	4
5	6	7
4	5	6

D.1.20 28 Processes

28 processes has a layout that are 7 processes high and 4 processes wide.

4	5	6	7
4	5	6	7
4	5	6	7
4	5	6	7
4	5	6	7
4	5	6	7
4	5	6	7

D.1.21 29 Processes

29 processes has a layout that are 29 processes high and 1 processes wide.

4	5	6	7	4	5	6	7	4	5	6	7	4	5	6	7	4	5	6	7	...
4	5	6	7	4	5	6	7	4												

D.1.22 30 Processes

30 processes has a layout that are 6 processes high and 5 processes wide.

4	5	6	7	4
5	6	7	4	5
6	7	4	5	6
7	4	5	6	7
4	5	6	7	4
5	6	7	4	5

D.1.23 31 Processes

31 processes has a layout that are 31 processes high and 1 processes wide.

4	5	6	7	4	5	6	7	4	5	6	7	4	5	6	7	4	5	6	7	...
4	5	6	7	4	5	6	7	4	5	6										

D.1.24 32 Processes

32 processes has a layout that are 8 processes high and 4 processes wide.

4	5	6	7
4	5	6	7
4	5	6	7
4	5	6	7
4	5	6	7
4	5	6	7
4	5	6	7
4	5	6	7

D.1.25 33 Processes

33 processes has a layout that are 11 processes high and 3 processes wide.

0	4	5
6	7	0
4	5	6
7	0	4
5	6	7
0	4	5
6	7	0
4	5	6
7	0	4
5	6	7
0	4	5

D.1.26 34 Processes

34 processes has a layout that are 17 processes high and 2 processes wide.

0	4
5	6
7	0
4	5
6	7
0	4
5	6
7	0
4	5
6	7
0	4
5	6
7	0
4	5
6	7
0	4
5	6

D.1.27 35 Processes

35 processes has a layout that are 7 processes high and 5 processes wide.

0	4	5	6	7
0	4	5	6	7
0	4	5	6	7
0	4	5	6	7
0	4	5	6	7
0	4	5	6	7
0	4	5	6	7

D.1.28 36 Processes

36 processes has a layout that are 6 processes high and 6 processes wide.

0	4	5	6	7	0
4	5	6	7	0	4
5	6	7	0	4	5
6	7	0	4	5	6
7	0	4	5	6	7
0	4	5	6	7	0

D.1.29 37 Processes

37 processes has a layout that are 37 processes high and 1 processes wide.

0	4	5	6	7	0	4	5	6	7	0	4	5	6	7	0	4	5	6	7	...
0	4	5	6	7	0	4	5	6	7	0	4	5	6	7	0	4				

D.1.30 38 Processes

38 processes has a layout that are 19 processes high and 2 processes wide.

0	4
5	6
7	0
4	5
6	7
0	4
5	6
7	0
4	5
6	7
0	4
5	6
7	0
4	5
6	7
0	4
5	6
7	0
4	5

D.1.31 39 Processes

39 processes has a layout that are 13 processes high and 3 processes wide.

0	4	5
6	7	0
4	5	6
7	0	4
5	6	7
0	4	5
6	7	0
4	5	6
7	0	4
5	6	7
0	4	5
6	7	0
4	5	6

D.1.32 40 Processes

40 processes has a layout that are 8 processes high and 5 processes wide.

0	4	5	6	7
0	4	5	6	7
0	4	5	6	7
0	4	5	6	7
0	4	5	6	7
0	4	5	6	7
0	4	5	6	7
0	4	5	6	7

D.2 Node Layout Using Rankfiles

D.2.1 9 Processes

9 processes has a layout that are 3 processes high and 3 processes wide.

6	6	6
6	6	6
7	7	7

D.2.2 10 Processes

10 processes has a layout that are 5 processes high and 2 processes wide.

6	6
6	6
6	6
6	6
7	7

D.2.3 11 Processes

11 processes has a layout that are 11 processes high and 1 processes wide.

6	6	6	6	6	6	6	6	7	7	7

D.2.4 12 Processes

12 processes has a layout that are 4 processes high and 3 processes wide.

6	6	6
6	6	6
7	7	7
7	7	7

D.2.5 13 Processes

13 processes has a layout that are 13 processes high and 1 processes wide.

6	6	6	6	6	6	6	6	7	7	7	7	7

D.2.6 14 Processes

14 processes has a layout that are 7 processes high and 2 processes wide.

6	6
6	6
6	6
6	6
7	7
7	7
7	7

D.2.7 15 Processes

15 processes has a layout that are 5 processes high and 3 processes wide.

6	6	6
6	6	6
6	6	7
7	7	7
7	7	7

D.2.8 16 Processes

16 processes has a layout that are 4 processes high and 4 processes wide.

6	6	6	6
6	6	6	6
7	7	7	7
7	7	7	7

D.2.9 17 Processes

17 processes has a layout that are 17 processes high and 1 processes wide.

0	0	0	0	0	0	0	0	4	4	4	4	4	4	4	4	5

D.2.10 18 Processes

18 processes has a layout that are 6 processes high and 3 processes wide.

0	0	0
0	0	0
4	4	4
4	4	4
5	5	5
5	5	5

D.2.11 19 Processes

19 processes has a layout that are 19 processes high and 1 processes wide.

0	0	0	0	0	0	0	0	4	4	4	4	4	4	4	4	5	5	5

D.2.12 20 Processes

20 processes has a layout that are 5 processes high and 4 processes wide.

0	0	0	0
0	0	0	0
4	4	4	4
4	4	4	4
5	5	5	5

D.2.13 21 Processes

21 processes has a layout that are 7 processes high and 3 processes wide.

0	0	0
0	0	0
0	0	4
4	4	4
4	4	4
4	5	5
5	5	5

D.2.14 22 Processes

22 processes has a layout that are 11 processes high and 2 processes wide.

0	0
0	0
0	0
0	0
4	4
4	4
4	4
4	4
5	5
5	5
5	5

D.2.15 23 Processes

23 processes has a layout that are 23 processes high and 1 processes wide.

0	0	0	0	0	0	0	0	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5

D.2.16 24 Processes

24 processes has a layout that are 6 processes high and 4 processes wide.

0	0	0	0
0	0	0	0
4	4	4	4
4	4	4	4
5	5	5	5
5	5	5	5

D.2.17 25 Processes

25 processes has a layout that are 5 processes high and 5 processes wide.

4	4	4	4	4
4	4	4	5	5
5	5	5	5	5
5	6	6	6	6
6	6	6	6	7

D.2.18 26 Processes

26 processes has a layout that are 13 processes high and 2 processes wide.

4	4
4	4
4	4
4	4
5	5
5	5
5	5
5	5
6	6
6	6
6	6
6	6
7	7

D.2.19 27 Processes

27 processes has a layout that are 9 processes high and 3 processes wide.

4	4	4
4	4	4
4	4	5
5	5	5
5	5	5
5	6	6
6	6	6
6	6	6
7	7	7

D.2.20 28 Processes

28 processes has a layout that are 7 processes high and 4 processes wide.

4	4	4	4
4	4	4	4
5	5	5	5
5	5	5	5
6	6	6	6
6	6	6	6
7	7	7	7

D.2.21 29 Processes

29 processes has a layout that are 29 processes high and 1 processes wide.

4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	6	6	6	6	...
6	6	6	6	7	7	7	7	7												

D.2.22 30 Processes

30 processes has a layout that are 6 processes high and 5 processes wide.

4	4	4	4	4
4	4	4	5	5
5	5	5	5	5
5	6	6	6	6
6	6	6	6	7
7	7	7	7	7

D.2.23 31 Processes

31 processes has a layout that are 31 processes high and 1 processes wide.

4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	6	6	6	6	...
6	6	6	6	7	7	7	7	7	7	7									

D.2.24 32 Processes

32 processes has a layout that are 8 processes high and 4 processes wide.

4	4	4	4
4	4	4	4
5	5	5	5
5	5	5	5
6	6	6	6
6	6	6	6
7	7	7	7
7	7	7	7

D.2.25 33 Processes

33 processes has a layout that are 11 processes high and 3 processes wide.

0	0	0
0	0	0
0	0	4
4	4	4
4	4	4
4	5	5
5	5	5
5	5	5
6	6	6
6	6	6
6	6	7

D.2.26 34 Processes

34 processes has a layout that are 17 processes high and 2 processes wide.

0	0
0	0
0	0
0	0
4	4
4	4
4	4
4	4
5	5
5	5
5	5
5	5
6	6
6	6
6	6
6	6
7	7

D.2.27 35 Processes

35 processes has a layout that are 7 processes high and 5 processes wide.

0	0	0	0	0
0	0	0	4	4
4	4	4	4	4
4	5	5	5	5
5	5	5	5	6
6	6	6	6	6
6	6	7	7	7

D.2.28 36 Processes

36 processes has a layout that are 6 processes high and 6 processes wide.

0	0	0	0	0	0
0	0	4	4	4	4
4	4	4	4	5	5
5	5	5	5	5	5
6	6	6	6	6	6
6	6	7	7	7	7

D.2.29 37 Processes

37 processes has a layout that are 37 processes high and 1 processes wide.

0	0	0	0	0	0	0	0	4	4	4	4	4	4	4	4	5	5	5	5	...
5	5	5	5	6	6	6	6	6	6	6	6	7	7	7	7	7				

D.2.30 38 Processes

38 processes has a layout that are 19 processes high and 2 processes wide.

0	0
0	0
0	0
0	0
4	4
4	4
4	4
4	4
5	5
5	5
5	5
5	5
6	6
6	6
6	6
6	6
7	7
7	7
7	7

D.2.31 39 Processes

39 processes has a layout that are 13 processes high and 3 processes wide.

0	0	0
0	0	0
0	0	4
4	4	4
4	4	4
4	5	5
5	5	5
5	5	5
6	6	6
6	6	6
6	6	7
7	7	7
7	7	7

D.2.32 40 Processes

40 processes has a layout that are 8 processes high and 5 processes wide.

0	0	0	0	0
0	0	0	4	4
4	4	4	4	4
4	5	5	5	5
5	5	5	5	6
6	6	6	6	6
6	6	7	7	7
7	7	7	7	7

Appendix E

Runtime results in seconds

E.1 Runtime for size 256 on Clustis3 with border-thickness 1-5

256	1	2	3	4	5
1	101.53				
2	51.33	51.9	52.44	52.43	52.76
3	35.01	35.5	36.33	36.32	36.79
4	27.36	27.2	27.45	27.45	27.68
5	21.85	22.19	23.04	23.03	23.45
6	19.05	18.97	19.29	19.3	19.53
7	16.04	16.36	17.15	17.17	17.52
8	14.59	14.41	14.75	14.78	14.96
9	53.03	35.79	31.08	28.74	26.85
10	30.78	23.53	20.62	19.37	18.47
11	38.53	30.81	27.08	24.37	23.88
12	53.57	34.52	28.59	25.75	23.49
13	37.69	30.45	26.96	23.92	23.61
14	28.23	19.31	16.59	15.32	14.65
15	52.16	33.62	27.43	24.57	22.34
16	32.25	21.05	17.66	16.05	15.41
17	36.11	29.26	25.52	23.1	22.81
18	28.82	19.01	16.14	14.74	13.97
19	36.06	29.23	25.67	23.18	23.15
20	50.37	30.8	24.64	21.32	20.41
21	28.61	18.69	15.3	13.84	12.99
22	50.49	31.62	27.18	23.42	21.18
23	36.72	31.32	26.96	24.1	24.07

256	1	2	3	4	5
24	50.76	30.51	23.99	20.8	20.08
25	50.79	30.51	24.02	20.79	19.08
26	48.29	30.77	25.82	21.93	20.03
27	49.91	30.62	23.48	22.02	19.07
28	29.01	17.67	14.23	12.92	11.76
29	34.81	28.84	25.32	22.88	22.89
30	50.48	29.58	22.65	19.49	17.72
31	35.69	30.19	26.76	23.46	23.69
32	28.8	17.57	13.99	12.21	11.32
33	46.75	28.27	22.59	20.82	18.82
34	47.79	29.97	25.73	21.87	19.64
35	28.46	17.48	14.04	12.19	11.14
36	48.7	28.74	21.81	18.67	16.86
37	34.68	28.53	26.04	22.89	22.84
38	48.4	30.04	26.53	22.11	20.14
39	48.39	28.88	22.72	21.46	19.03
40	28.59	17.35	13.5	11.77	10.73

E.2 Runtime for size 512 on Clustis3 with border-thickness 1-5

512	1	2	3	4	5
1	401.31				
2	202.19	204	205.4	205.54	
3	136.43	138.19	139.47	140.05	
4	104.13	104.47	105.13	105.29	
5	83.95	85.22	86.38	86.8	
6	71.46	71.86	72.48	72.78	
7	61.66	62.53	63.55	64.06	
8	53.93	54.15	54.73	55.12	
9	96.33	81.76	74.55	70.91	68.78
10	69.56	60.2	57.67	57.24	56.24
11	82.85	68.22	63.95	63.21	62.53
12	88.73	69.83	64.7	59.81	57.5
13	79.81	64.89	61.3	59.47	59.08
14	55.67	47.53	45.12	44.57	44.62
15	83.07	64.16	57.64	54.93	52.79
16	57.75	47.26	46.66	43.36	42.05
17	73.62	59.16	55.28	53.06	53.05

512	1	2	3	4	5
18	52.04	42.97	39.51	38.45	39.02
19	71.55	57.45	54.72	52.16	51.96
20	75.51	54.67	50.19	47.94	44.69
21	49.66	39.91	36.38	35.1	36.64
22	74.08	57.18	50.12	45.7	45.17
23	72.89	56.97	55.19	53.57	52.49
24	75.01	53.26	48.11	43.85	43.25
25	73.79	55.21	49.15	48.5	45.98
26	72.34	54.74	47.26	42.96	41.22
27	71.17	50.24	44.74	41.13	39.31
28	50.54	39.88	36.2	35.85	35.36
29	67.16	52.28	49.79	48.92	48.56
30	70.15	49.04	42.24	40.4	39.35
31	69.17	54.22	51.41	49.71	49.97
32	45.48	33.68	29.99	28.54	27.52
33	72.26	51.3	46.37	42.4	40.61
34	70.47	52.31	46.15	42.59	39.81
35	69.76	61.95	57.56	55.17	53.53
36	74.25	52.97	49.82	42.52	42.47
37	65.33	50.94	48.49	47.42	46.8
38	69.67	52.41	45.4	41.65	39.14
39	70.44	49.48	46.51	41.87	39.8
40	61.88	58.06	53.97	50.22	51.76

E.3 Runtime for size 512 on Clustis3 with border-thickness 1-5

1024	1	2	3	4	5
1	1708.83				
2	813.52	828.59	829.28	830.13	831.42
3	544.08	554.53	556.24	557.24	558.99
4	408.86	415.08	416.5	416.65	418.2
5	331.61	338.78	340.75	341.93	343.96
6	280.73	285.45	286.7	287.13	288.65
7	242.15	246.98	248.31	249.87	251.28
8	210.02	213.57	214.9	215.28	216.68
9	262.55	239.56	230.11	227.2	225.14
10	209.65	204.01	200.11	198.58	198.32
11	219.24	206.66	200.7	200.59	200.13

1024	1	2	3	4	5
12	215.37	194.83	186.99	182.56	181.32
13	200.25	187.63	182.09	180.04	179.29
14	162.24	155.21	153.87	151.6	151.59
15	190.09	171.98	162.72	158.48	155.98
16	154.64	145.89	141.71	139.62	139.05
17	174.59	161.29	158.15	156.22	155.77
18	143.13	133.87	132.79	130.47	130.52
19	167.22	152.98	149.8	147.72	146.71
20	163.48	149.74	140.61	136.34	134
21	133.49	123.13	160.57	120.52	120.16
22	166.65	139.25	134.53	132.54	131.01
23	161.29	146.36	141.95	139.71	140.22
24	153.91	134.71	129.58	122.34	123.87
25	204.61	188.45	185.51	174.19	171.28
26	178.3	148.7	190.43	134.32	131.57
27	213.15	173.14	186.72	152.75	143.89
28	205.69	193.14	197.49	190.39	194.23
29	157.09	136.82	132.89	129.75	130.06
30	217.93	194.75	191.37	174.05	165.85
31	155.03	138.59	131.81	131.51	131.11
32	209.31	203.5	193.91	192.56	188.77
33	225.48	188.14	195.27	173.16	171.15
34	176.25	160.79	155.46	150.82	134.82
35	213.81	204.65	206.46	195.39	208.71
36	238.11	217.86	205.09	191.84	188.18
37	157.36	139.57	134.93	129.16	127.93
38	175.69	147.36	142.07	142.04	138.79
39	215.25	184.72	187.34	172.91	163.66
40	227.42	207.53	206.63	203.03	204.92

E.4 Runtime for size 256 on Clustis3 with border-thickness 1-5 using rankfile

256	1	2	3	4	5
1	101.53				
2	51.33	51.9	52.44	52.43	52.76
3	35.01	35.5	36.33	36.32	36.79
4	27.36	27.2	27.45	27.45	27.68
5	21.85	22.19	23.04	23.03	23.45

256	1	2	3	4	5
6	19.05	18.97	19.29	19.3	19.53
7	16.04	16.36	17.15	17.17	17.52
8	14.59	14.41	14.75	14.78	14.96
9	32.95	24.57	21.85	20.67	19.61
10	30.02	22.46	20.93	18.77	17.81
11	30.31	23.41	20.53	19.04	18.43
12	30.83	22.25	19.72	18.55	17.47
13	28.66	21.86	18.99	17.55	16.95
14	27.91	20.47	18.9	16.87	16
15	39.24	26.15	21.29	19.52	17.99
16	30.11	20.51	17.56	16.22	15.34
17	27.7	20.97	18.23	16.44	15.84
18	28.76	19.85	17.09	16.94	16.05
19	27.63	20.12	17.13	15.65	15.27
20	30.08	20.12	16.79	15.21	15.17
21	40.97	23.73	18.8	17.32	15.9
22	27.08	19.02	16.67	15.07	14.43
23	26.09	19.38	16.12	14.84	14.3
24	28.95	18.89	15.7	14.1	14.08
25	38.84	24.27	19.25	16.75	15.32
26	26.07	19.1	16.39	13.94	13.17
27	40.26	22.83	18.57	17.18	14.59
28	27.65	17.96	14.81	13.28	13.46
29	24.75	18.63	14.99	13.99	12.87
30	39.25	24.16	18.67	15.79	14.24
31	24.83	18.61	15.01	13.79	12.93
32	27.14	17.29	14.27	12.87	12.75
33	39.5	21.85	17.87	16.63	14.82
34	24.67	17.98	15.55	13.21	12.32
35	39.2	23.71	18.44	15.41	13.82
36	38.13	22.92	17.54	15.32	13.78
37	24.14	17.7	14.52	13.06	12.1
38	24.13	17.24	15.06	12.82	11.73
39	38.23	21.08	16.94	15.84	13.4
40	39.07	22.97	17.43	14.6	12.94

E.5 Runtime for size 512 on Clustis3 with border-thickness 1-5 using rankfile

512	1	2	3	4	5
1	401.31				
2	202.19	204	205.4	205.54	
3	136.43	138.19	139.47	140.05	
4	104.13	104.47	105.13	105.29	
5	83.95	85.22	86.38	86.8	
6	71.46	71.86	72.48	72.78	
7	61.66	62.53	63.55	64.06	
8	53.93	54.15	54.73	55.12	
9	72.38	64.3	60.64	58.44	59.74
10	66.22	58.5	54.19	52.27	51.22
11	65.09	56.06	52.06	51.3	51.42
12	63.49	55.46	51.94	50.02	49.73
13	60.07	51.2	47.25	46.48	46.83
14	57.07	49.56	45.43	44.04	43.66
15	68.76	54.25	50.14	47.91	45.46
16	56.43	47.09	43.59	41.62	40.43
17	55.27	45.96	42.23	40.95	40.74
18	54.69	46.2	43.05	40.78	40.61
19	52.1	42.54	38.98	37.55	37.48
20	52.35	43.4	41.78	39.05	37.69
21	59.98	45	40.93	38.48	37.35
22	48.66	41.9	36.55	35.82	34.92
23	48.21	39.01	35.54	34.1	33.86
24	48.1	39.59	37.86	35.11	33.7
25	58.39	43.61	38.41	36.31	34.07
26	45.22	39.25	33.86	32.99	32.36
27	54.86	41.07	35.82	33.02	31.79
28	45.25	36.86	34.94	32.06	30.64
29	45.75	35.9	31.25	30.34	30.44
30	55.5	41.3	36.36	33.89	32.01
31	44.96	34.97	30.79	29.77	29.76
32	44.02	35.46	33.78	30.75	29.29
33	56.4	39.27	34.8	31.67	29.57
34	43.26	35.47	31.21	32.78	32.94
35	61.59	44.68	36.14	39.09	33.71
36	63.15	52.17	43.69	53.25	41.71

512	1	2	3	4	5
37	41.56	32.7	28.75	27.3	26.96
38	41.63	33.12	29.67	41.83	29.91
39	53.83	37.18	32.52	29.73	27.3
40	67.46	49.43	44.52	51.63	51.63

E.6 Runtime for size 1024 on Clustis3 with border-thickness 1-5 using rankfile

1024	1	2	3	4	5
1	1708.83				
2	813.52	828.59	829.28	830.13	831.42
3	544.08	554.53	556.24	557.24	558.99
4	408.86	415.08	416.5	416.65	418.2
5	331.61	338.78	340.75	341.93	343.96
6	280.73	285.45	286.7	287.13	288.65
7	242.15	246.98	248.31	249.87	251.28
8	210.02	213.57	214.9	215.28	216.68
9	223.01	210.76	211.68	209.97	211.05
10	202.82	189.8	187.26	186.9	187.79
11	189.52	178.09	177	176.17	176.34
12	186.67	174.61	172.49	171.97	172.22
13	168.82	157.86	156.49	155.28	155.09
14	163.04	150.22	147.22	147.13	147.18
15	171.6	158.02	150.7	146.95	145.47
16	153.65	141.84	137.92	138.22	137.7
17	148.93	138.42	136.02	134.8	134.54
18	149.94	138.34	135.33	133.41	132.49
19	138.09	126.18	125.17	124.13	123.92
20	140.78	131.58	126.72	124.61	126.13
21	143.73	130.77	163.02	122.88	121.1
22	131.04	119.67	116.36	115.66	115.08
23	124.67	112.67	111.38	110.16	109.78
24	126.72	117.22	112.44	110.13	109.96
25	136.95	128.84	132.07	133.78	125.72
26	137.68	143.26	170.61	139.1	129.52
27	205.88	177.51	194.88	193.95	192.28
28	187.05	175.43	175.86	198.61	196.32
29	187.13	182.21	174.16	171.47	173.1
30	218.76	199.42	192.64	187.86	187.18

1024	1	2	3	4	5
31	200.61	180.75	185.72	169.34	145.65
32	222.89	214.67	207.3	215.83	212.36
33	213.9	192.62	214.11	207.97	206.39
34	205.83	222.52	202.44	194.7	192
35	244.07	228.81	225.49	215.93	225.7
36	251.83	232.29	223.07	219.26	215.96
37	221.56	202.02	195.46	190.85	198.42
38	213.06	219.2	215.61	209.82	204.67
39	226.2	219.88	225.06	211.57	208.79
40	250.87	233.33	221.65	223.22	215.4

E.7 Runtime size 256 on Numascale with border-thickness 1-5

256	1	2	3	4	5
1	47.94	25.23	27.56	25.11	
2	23.99	22.73	23.75	23.04	24.03
3	16.97	15.68	16.39	16.12	16.98
4	13.68	12.49	13.07	12.53	13.57
5	11.17	10.33	10.79	10.68	11.27
6	10.59	9.50	9.89	9.46	9.97
7	8.73	7.87	8.39	8.22	8.75
8	8.31	7.38	7.66	7.36	7.79
9	8.94	7.66	7.76	7.15	7.90
10	7.40	6.57	6.78	6.46	6.81
11	6.54	5.56	5.80	5.97	6.30
12	6.72	5.66	5.80	5.52	5.90
13	5.16	4.73	5.03	5.23	5.61
14	5.92	5.18	5.35	5.02	5.33
15	6.81	5.45	5.43	4.98	5.43
16	5.59	4.93	4.72	4.60	4.74
17	4.39	3.92	4.14	4.43	4.82
18	5.23	4.30	4.35	4.12	4.43
19	3.94	3.66	3.97	4.20	4.41
20	5.24	4.36	4.45	4.05	4.35
21	5.43	4.80	4.27	3.91	4.27
22	4.55	3.86	3.94	3.81	4.06
23	3.75	3.35	3.55	3.89	4.15
24	4.63	3.69	3.68	3.43	3.68

256	1	2	3	4	5
25	5.70	4.35	4.14	3.76	4.14
26	4.14	3.49	3.78	3.42	3.68
27	5.00	3.80	3.80	3.47	3.84
28	4.54	3.67	3.70	3.36	3.67
29	3.00	2.83	2.90	3.26	3.53
30	4.39	3.48	3.40	3.22	3.46
31	3.25	2.73	2.89	3.26	3.54
32	4.05	3.17	3.10	2.88	3.16
33	9.05	5.19	4.57	3.96	3.86
34	9.43	8.00	5.69	5.22	4.84
35	12.07	7.70	6.32	5.38	4.97
36	13.75	8.74	6.62	5.93	5.30
37	7.28	5.35	6.71	6.78	6.31
38	10.23	10.82	6.21	5.69	5.49
39	16.25	8.62	7.13	7.08	6.04
40	16.28	9.64	8.61	7.06	6.33
41	7.69	5.73	5.07	7.03	6.54
42	16.15	10.79	8.37	9.35	6.57
43	7.63	5.71	5.13	6.67	7.59
44	12.63	9.39	6.79	6.17	5.86
45	10.71	10.23	8.31	8.20	6.90
46	10.45	7.38	6.41	5.93	5.65
47	7.72	5.74	6.43	6.89	6.60
48	17.04	15.18	8.87	10.77	6.89
49	18.47	14.65	9.32	8.18	16.00
50	16.62	10.36	8.75	7.87	6.81

E.8 Runtime size 512 on Numascale with border-thickness 1-5

512	1	2	3	4	5
1	206.74				
2	94.26	89.69	92.24	90.43	95.27
3	64.33	61.81	64.41	61.40	64.69
4	50.38	46.93	49.56	47.24	49.90
5	42.05	39.60	41.22	39.30	41.75
6	37.14	34.56	36.06	35.05	36.78
7	32.17	30.04	31.15	29.77	32.17
8	28.27	26.34	27.53	26.97	28.33
9	27.98	26.56	26.64	25.92	32.32
10	24.65	22.99	23.87	23.46	24.60

512	1	2	3	4	5
11	21.23	20.45	21.36	20.30	22.20
12	20.58	19.69	20.10	19.56	19.79
13	18.06	17.73	18.51	17.65	20.50
14	19.36	17.74	18.28	18.16	18.99
15	19.07	17.94	17.91	18.30	17.75
16	16.88	15.26	15.92	15.14	15.69
17	14.77	14.44	14.91	14.29	15.52
18	15.27	14.35	14.23	14.21	14.38
19	13.25	12.95	13.32	12.88	13.88
20	15.66	14.22	14.45	13.86	14.19
21	15.65	14.22	15.36	13.41	13.38
22	13.55	12.52	12.84	13.01	13.64
23	11.52	11.83	12.24	12.02	13.14
24	13.12	11.64	11.84	11.40	11.76
25	14.78	13.23	12.85	12.84	12.85
26	12.05	11.09	11.29	11.57	12.30
27	13.46	12.24	11.90	11.76	11.92
28	13.16	11.76	11.78	11.31	11.48
29	9.50	10.22	10.17	9.88	10.67
30	12.05	10.81	10.67	10.61	10.88
31	9.05	9.54	9.80	9.63	10.59
32	11.12	9.71	9.72	9.51	9.68
33	14.00	11.42	10.93	10.42	10.40
34	15.49	12.64	11.94	14.10	14.19
35	17.66	13.57	12.90	12.58	12.86
36	18.37	13.50	12.63	11.88	13.67
37	12.54	14.76	14.76	13.33	13.59
38	16.66	13.88	13.28	16.02	15.73
39	18.70	15.25	14.21	13.69	13.77
40	20.63	17.12	14.79	14.31	14.27
41	12.96	23.89	25.05	23.18	25.10
42	21.89	23.95	25.47	23.41	24.81
43	12.90	22.71	23.72	21.79	23.71
44	24.50	23.62	23.52	21.90	23.94
45	20.83	23.21	24.10	22.11	24.11
46	30.56	23.29	24.26	22.38	23.99
47	13.28	24.69	25.46	24.18	25.39
48	22.58	25.05	25.53	24.28	25.50
49	45.59	25.28	26.31	25.88	25.79
50	21.41	25.87	26.49	25.32	25.60

E.9 Runtime size 1024 on Numascale with border-thickness 1-5

1024	1	2	3	4	5
1	1541.08				
2	687.89	532.51	470.75	373.92	391.52
3	343.33	292.98	310.01	265.98	271.75
4	242.37	241.23	261.36	209.29	217.13
5	236.34	177.07	207.58	170.98	174.35
6	190.30	179.73	189.33	144.60	154.73
7	135.86	131.31	129.20	122.84	128.26
8	112.32	109.29	115.66	104.91	109.79
9	104.53	98.63	103.81	97.46	105.08
10	94.87	91.31	95.18	89.30	93.43
11	85.70	81.79	82.26	79.98	81.69
12	75.84	72.29	76.22	73.42	77.44
13	72.23	69.39	69.01	67.52	68.90
14	71.93	69.24	71.92	67.26	69.74
15	68.36	64.93	68.42	64.92	68.19
16	60.87	56.56	58.24	57.19	59.17
17	56.77	54.66	54.19	53.55	55.01
18	53.55	50.85	53.84	51.60	53.84
19	50.34	49.36	49.24	48.17	49.86
20	56.88	52.85	53.56	52.30	53.92
21	53.14	49.48	52.31	48.26	49.43
22	48.56	47.02	48.56	45.84	48.50
23	44.03	43.27	43.39	42.72	44.50
24	44.51	41.50	42.52	40.55	41.93
25	48.45	46.27	45.53	45.18	46.64
26	42.03	40.76	43.04	39.36	41.55
27	44.55	41.81	44.49	42.23	44.27
28	45.81	42.53	42.15	41.70	42.26
29	36.42	35.48	35.58	34.90	36.42
30	37.88	36.73	36.66	36.11	37.63
31	34.81	33.88	34.18	33.72	35.65
32	37.03	34.10	34.41	34.20	34.66
33	39.59	35.86	37.83	35.55	36.67
34	39.23	40.01	39.81	38.03	38.91
35	45.84	43.53	42.78	41.40	43.96
36	41.03	38.67	38.36	37.75	43.19

1024	1	2	3	4	5
37	42.58	38.64	38.12	37.79	38.26
38	41.57	44.22	42.67	40.54	40.90
39	44.55	40.51	44.46	42.30	42.43
40	45.24	41.09	40.56	39.69	42.62
41	48.19	39.94	39.51	39.36	
42	49.47	46.40	46.42	44.82	
43	45.55	42.11	40.58	40.15	
44	46.32	43.07	42.86	43.60	
45	49.40	46.87	45.04	44.81	
46	45.46	46.45	45.67	43.00	
47	45.69	41.28	41.17	41.03	
48	48.77	44.24	45.71	44.11	
49	51.83	49.27	48.79	48.31	
50	49.58	45.27	43.98	44.70	

E.10 Runtime for dense layout on Numascale

Dense	256	512	1024		
1	47.94	206.74	1541.08		
2	23.99	94.26	687.89		
4	13.68	50.38	242.37		
8	8.31	28.27	112.32		
16	5.59	16.88	60.87		
32	4.05	11.12	37.03		
64	20.72	25.65	60.05		
128	133.18	183.08	241.18		

E.11 Runtime for horzontal striped layout on Numascale

Striped horizontal p 1	256	512	1024		
1	47.94	206.74	1541.08		
2	23.99	94.26	687.89		
4	12.65	48.60	244.04		
8	7.39	27.16	111.16		
16	4.43	14.93	57.52		
32	3.00	8.73	32.66		
64	11.29	16.58	53.54		
128	44.89	77.87	275.12		

E.12 Runtime for vertical striped layouts on Numscale

Striped vertical 1 p	256	512	1024
1	47.94	206.74	1541.08
2	23.99	94.26	687.89
4	14.37	53.08	231.92
8	9.40	32.12	124.50
16	6.68	20.49	73.48
32	5.64	15.39	50.50
64	10.86	25.99	79.52
128	39.82	96.56	256.87

E.13 Write to file runtime Numascale

size	29 processes write to file	29 not write to file	difference
256	3.881135	2.99786	0.883275
512	11.601198	9.504451	2.096747
1024	44.960638	36.4152	8.545438

E.14 Write to file runtime Clustis3

size	8 processes write to file	8 not write to file	difference
256	27.569904	14.80545	12.764454
512	90.864921	54.846053	36.018868
1024	256.028561	213.941491	42.08707

E.15 Avg, min and max runtime for size 1024 on Clustis3

1024	1024 avg	1024 min	1024 max
1	2422.378979	2421.768586	2422.9345
2	813.519605	812.694452	814.842656
3	544.077871	543.766894	544.681657
4	408.858205	408.523244	409.819141
5	331.613308	331.37566	332.007108
6	280.727989	279.973628	281.767408
7	242.145639	241.919111	242.293015
8	210.024925	209.443913	210.861328
9	262.553683	261.311	264.734173
10	209.648175	208.968582	210.717832
11	219.242048	216.852869	224.759808
12	215.374302	213.079243	220.723745
13	200.253474	198.265569	204.140066
14	162.243683	161.579098	163.663311
15	190.088177	186.713366	196.903126
16	154.643041	153.280083	156.004919
17	174.586739	173.215009	177.279032
18	143.128051	141.615983	144.844975
19	167.221949	165.317552	168.907659
20	163.477	161.21619	168.419689
21	133.486495	131.051969	138.845156
22	166.652463	155.582566	172.933967
23	161.285173	154.28603	164.836688
24	153.910379	148.081592	156.921367
25	204.608912	192.239084	230.120314
26	178.299863	158.574112	195.769827
27	213.145683	198.751118	226.698795
28	205.693586	194.731913	218.791786
29	157.093172	146.914952	177.303474
30	217.930828	200.647688	236.336393
31	155.025522	146.670998	181.533802
32	209.3131	191.120429	225.460724
33	225.48484	201.267986	245.646857
34	176.249794	152.694141	200.727538
35	213.80691	198.262178	236.912395
36	238.107908	222.344429	253.487872

1024	1024 avg	1024 min	1024
37	157.361371	140.935038	185.702443
38	175.691343	146.972283	206.643026
39	215.2497	187.98071	238.478398
40	227.415573	205.080118	243.61974

E.16 Avg, min and max runtime for size 1024 on Numascale

1024	1024 avg	1024 min	1024 max
1	1541.078063	1464.526123	1684.711531
2	687.885181	598.883633	1438.87211
3	343.330976	286.939379	1339.612725
4	242.367617	224.493612	575.330802
5	236.336981	180.109697	346.236028
6	190.297526	159.378185	247.624921
7	135.864616	130.275795	143.953006
8	112.317986	111.561871	114.3686
9	104.532985	104.329174	105.041151
10	94.87049	93.731297	95.333164
11	85.704242	85.527797	85.884439
12	75.839616	75.682307	75.97157
13	72.228493	72.038277	72.544747
14	71.934975	70.959715	72.389219
15	68.35869	68.104127	68.670578
16	60.867865	60.720927	61.166346
17	56.767441	56.69845	56.881022
18	53.54689	53.30568	53.761994
19	50.339546	50.23431	50.462416
20	56.879542	56.790041	56.971877
21	53.13514	52.921525	53.270293
22	48.564998	47.93172	49.12428
23	44.033014	43.961474	44.219269
24	44.5108	44.474359	44.558255
25	48.447748	48.349805	48.61348
26	42.03183	41.236813	42.390491
27	44.553432	44.454208	44.623591
28	45.811985	45.734629	45.935369
29	36.4152	36.367895	36.507541
30	37.880413	37.786346	37.941231

1024	1024 avg	1024 min	1024 max
31	34.80898	34.75953	34.876979
32	37.02857	36.922462	37.165665
33	39.594942	39.299406	39.885415
34	39.229388	39.119241	39.361703
35	45.843063	45.099635	46.320982
36	41.026257	40.881199	41.142444
37	42.580639	42.395435	42.913026
38	41.573295	41.40543	42.059404
39	44.549339	44.033859	51.667259
40	45.250978	45.114912	45.486224
64	60.04845	52.397297	118.948531
128	241.179272	95.921863	592.762949