NTNU

TMA4280 - project part 2

Håvard Kvamme, Jørgen Vågan, Magnus Aarskaug Rud

April 12, 2015

Abstract

In this project the Kongull cluster have been used to solve the homogenous 2D Poisson problem on the unit square, both Message sending and shared memory solution models have been implemented and compared.

1 Problem Description

In this project the task is to solve the two-dimensional poisson problem as stated below.

$$-\nabla^2 u = f \text{ in } \Omega = (0,1) \times (0,1) \tag{1.1}$$

$$u = 0 \text{ on } \partial\Omega \tag{1.2}$$

The problem is solved using a fast diagonalization method. The derivation of the method can be found in [1] , and the system of equation to be solved can be written as

$$\underline{T}\,\underline{U} + \underline{U}\,\underline{T} = \underline{G} \tag{1.3}$$

where $\underline{T}$ is the discrete second order partial differential operator in one direction, $\underline{G}$ is the discrete loading function multiplied with the steplength squared h^2 and $\underline{U}$ is the discrete solution to the problem. and the method can be summed up into the following three steps:

step 1 - $O(n^2 \log(n))$

Compute $\underline{\tilde{G}} = \underline{Q}^T \underline{G}\,\underline{Q}$ - two matrix-matrix products

step 2 - $O(n^2)$

Compute $\tilde{u}_{i,j} = \frac{\tilde{g}_{i,j}}{\lambda_i + \lambda_j}$ - scalar addition and divition

step 3 - $O(n^2 log(n))$

Compute $\underline{U} = \underline{Q}\underline{\tilde{U}}\,\underline{Q}^T$ - two matrix-matrix products

$\underline{Q}$ and $\underline{\Lambda}$ contains the eigenvectors and the eigenvalues of the discrete second order partial differential operator.

Step 1 and 3 are solved using the infamous fast sine transform.

2 Implementation strategy

The implementation was done in C and can be found in the file `poisson-mpi.c`. The whole algorithm focuses on the tree steps in the previous section. As it should be able to run on supercomputers, it was necessary to use distributed memory model. This was done using MPI. To be able to do **step 1** and **step 3** with the fast sine transform, the external code `fst.f` was used. However, this code demands a problem size (number of rows) $n = 2^k$, where k is some integer. It also needed to work on a whole column at a time. The way **step 1** (**step 3** is solved exactly the same way) is done with the fast sine transform is as follows

$$\underline{\tilde{G}} = \underline{S}^{-1}\left(\left(\underline{S}\,\underline{G}\right)^T\right), \tag{2.1}$$

where S and S^{-1} is the fast- and inverse fast sine transform respectively. So this is done in three steps

$$\underline{G} \leftarrow \underline{S}\,\underline{G}, \tag{2.2}$$

$$\underline{A} \leftarrow \underline{G}^T, \tag{2.3}$$

$$\underline{\tilde{G}} \leftarrow \underline{S}^{-1}\,\underline{A}. \tag{2.4}$$

The second step (2.3) will require sending between processes as long as a column, row, or block layout is used. Therefore it is reasonable to distribute the matrices column-wise among the processes. As n is a power of two, the number of internal nodes, and therefore the size of the matrices, are $m = n - 1$ which is odd. When m is not a multiple of the number of processes p, the extra columns are given to the last process. One could have distributed them more evenly among the processes, but this was tested and did not have a significant impact on the computation time as $n >> p$.

As the fast sine transform need columns, the matrices was allocated column-wise in an attempt to make good use of cache. When transposing the $\underline{G}$, it would better with a row mayor layout, but as there are only two transpose operations and a total to 2 fast sine transforms and 2 inverse fast sine transforms, the column-mayor layout seemed the best choice.

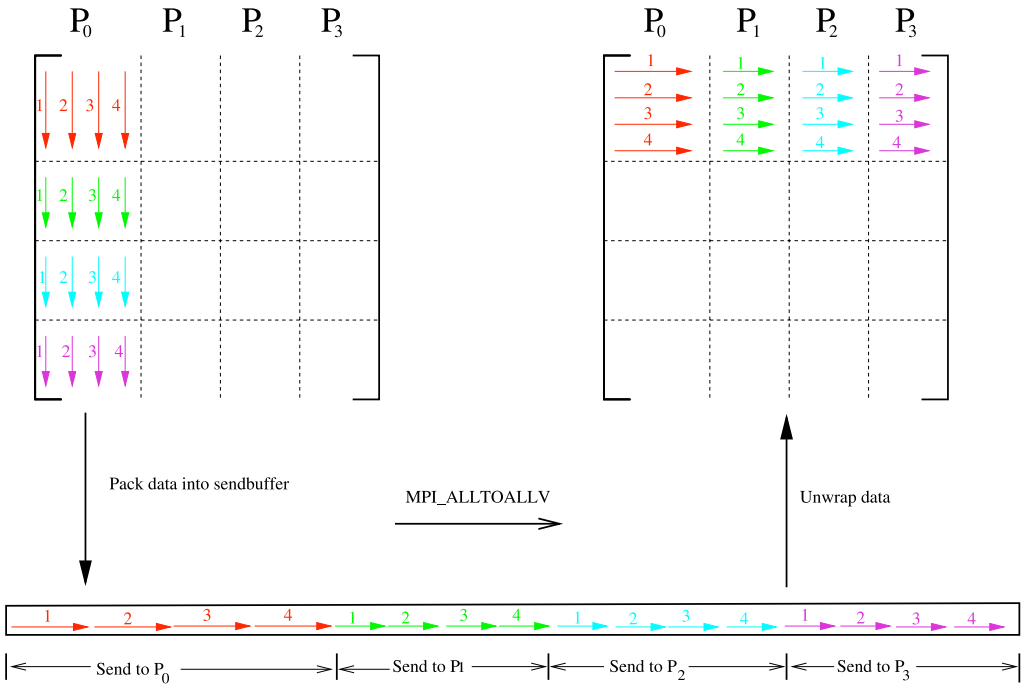

Figure 2.1: The transpose operation using message passing: the packing and unpacking of data. The figure is due to Bjarte Hægland.

When transposing the matrix, all the processes need to send data to each other. This is in principle done as in Figure 2.1. However, as the matrix of each process is structured column-wise, the local matrix ($m \times r_i$ for process i) is first transposed locally (to $r_i \times m$). Then the data is sent to the different processes with `MPI_Alltoallv`, and the data is restructured to create the transpose.

The implementation is done so that only two matrices are needed (in addition to some vectors of length n). Thus 2.3 might be a bit misleading when it comes to memory usage. With only two matrices each process use approximately $2n^2/p$ doubles of memory, and each node use around a total of $2n^2/nodes$. For the larges computations done in this project, $n = 2^{14} = 16384$, each node use around $4/nodes$ GB of memory.

2.1 OpenMP

A distributed memory model is needed to run on multiple nodes. However, it might be faster to use threads and shared memory on each machine, than dividing into MPI processes. Therefore, openMP was used to achieve this. All calls to the fast sine transform operates at one column at a time, therefore this was easily parallelized. A static scheduler was used, under the assumption that the fast sine transform takes a fixed amount of time for a give problem size. OpenMP was also used to create the $\underline{G}$ matrix and doing **step 2** as they are also embarrassingly parallel. The only difficult problem to parallelize was the loops used in the transposing. However, this was achieved by using openMP for each MPI process. For many MPI processes (eg. many nodes), this will cause a lot of opening and closing of threads, but as this problem is only run on a maximum of 3 nodes, it is not considered a problem.

As all the parallelization is pretty straight forward and the computation time of the fast sine transform only varies with n, the program should be well load balanced. As mentioned earlier one process gets up to $p-1$ more columns than the others. However, as $n >> p$ the program should still be well load balanced.

3 Description of kongull cluster

The Kongull cluster is a CentOS 5.3 linux cluster. It has 113 physical nodes, where 1 is used for login and 4 is used for I/O. The remaining 108 nodes are used for computing. The nodes are equipped with 2× 6-core 2.4 GHz processors, with 6× 512KiB (1 kibibyte = 1024 byte) L1-cache and a common 6 MiB (1 mebibyte = 1024^2 bytes) L3-cache. The compute nodes are spread out on 4 racks, (rack 0: 40 nodes, rack 2: 40 nodes, rack 1: 16 nodes; rack 3: 12 nodes) where the first 3 hold the HP AMP Opteron based nodes (2.4 GHz core speed) , while the last rack (rack 3) holds the new Dell intel based nodes (2× 2.60 GHz 8-core processors). In total 1344 compute nodes, with a "Fat Tree" network layout.

4 Results and Analysis

The results in this section was obtained by running the program `poisson-mpi.c` on the Kongull cluster. To reproduce the results the environment should be set by sourcing `environment.sh` and compiled using `compileKongull.sh`. Each job was submitted individually to a batch queue, and all jobs can be run with `results.sh`. All the plots can then be generated by running `plotting.R`.

4.1 Convergence analysis

First it is important to verify that the code works correctly. This was done by comparing the results to an analytical solution and plotting the error for different problem sizes n. The loading function chosen for this task was

$$f = 5\pi^2 \sin(\pi x)\sin(2\pi y), \tag{4.1}$$

with the solution

$$u = \sin(\pi x)\sin(2\pi y). \tag{4.2}$$

The max norm was used to measure the error. The calculations were run with two threads on 1, 3 and 6 MPI processes, and the results can be found in Figure 4.1. This plot is a log-log plot, with a reference line of slope -2. As the points follow the line, this means the method has quadratic convergence. As the numerical scheme is a

second order method, the figure confirms that our implementation is correct. For the rest of this section all tests are done with the f in Eq. (4.1), unless specified otherwise.

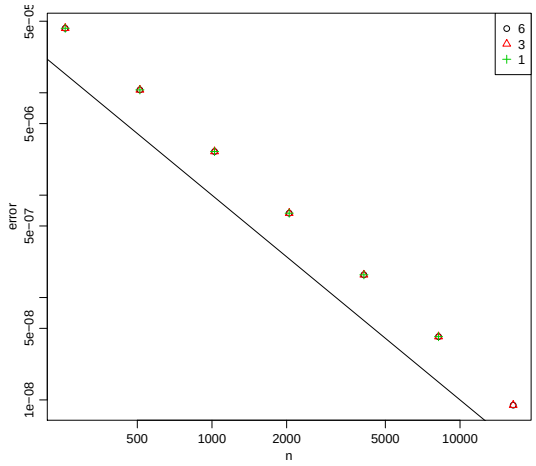

Figure 4.1: Loglog plot of the error as function of n. A reference line with slope -2 is drawn. The problem is run on two threads with the number of MPI processes specified in the legend.

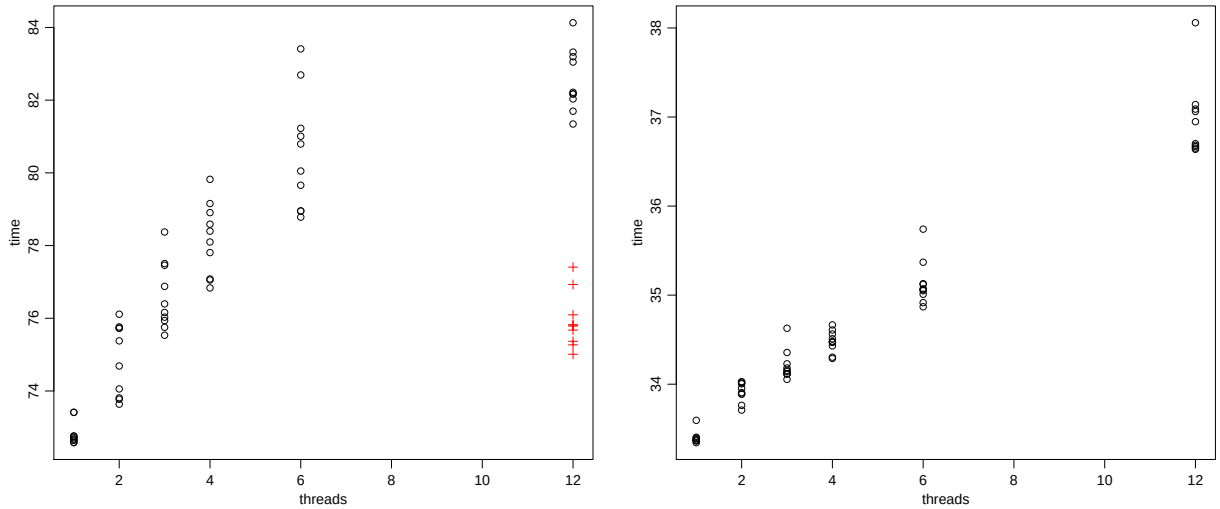

Figure 4.2: Times (in seconds) for running MPI processes vs threads with $n = 2^{14}$. The total number of processors used are 12 per node, so the number of MPI processes per node is $12/threads$. The left figure is run on one node, while the right is run on three nodes. The red crosses are run without any MPI sending at all.

4.2 Shared or distributed memory models

In order to show the difference between shared and distributed memory parallelization the problem was runned on a single node where all 12 processors were utilized. The point was to investigate the performance of using threads versus MPI processes. Thus the relation between threads t and MPI processes p is $pt = 12$ or $p = 12/t$. The calculations were done with $n = 2^{14} = 16384$. The same problem was repeated on three nodes where all 12 processors were utilized in the same way as above. The results can be found in Figure 4.2. It is clear from this figure that the time varies from each time the code is runned. This is important to take into account later in the

report, where the problem is only run once for each test case. It is clear from both plots that it is preferable to use only MPI processes and no openMP.

While running only one MPI process and 12 threads on one node there is still some overhead from the MPI sending to itself and the work of reordering the data. Therefore the problem was solved on 12 threads without this overhead. The results are the red crosses in Figure 4.2. It is clear that it is a lot faster than with the MPI sending, but it is still slower than running with 12 MPI processes.

The reasons for why pure MPI code is faster than the hybrid and even the pure openMP are most likely that the optimalization done by the compilator is better developed for the MPI library than for openMP. The overhead cost related to creating and closing threads are a lot higher than dividing the node into MPI processes. The advantage openMP has over MPI is the fact that all threads shares memory, instead of distributing memory to each process. Thus openMP is advantageous over MPI if the problem requires that many processors uses the same memory. Since our problem has a relative small memory usage, and the communication needed between the processors is minimal (only two send operations of $8n^2$ bytes is required) pure MPI is the optimal way to parallelize the code.

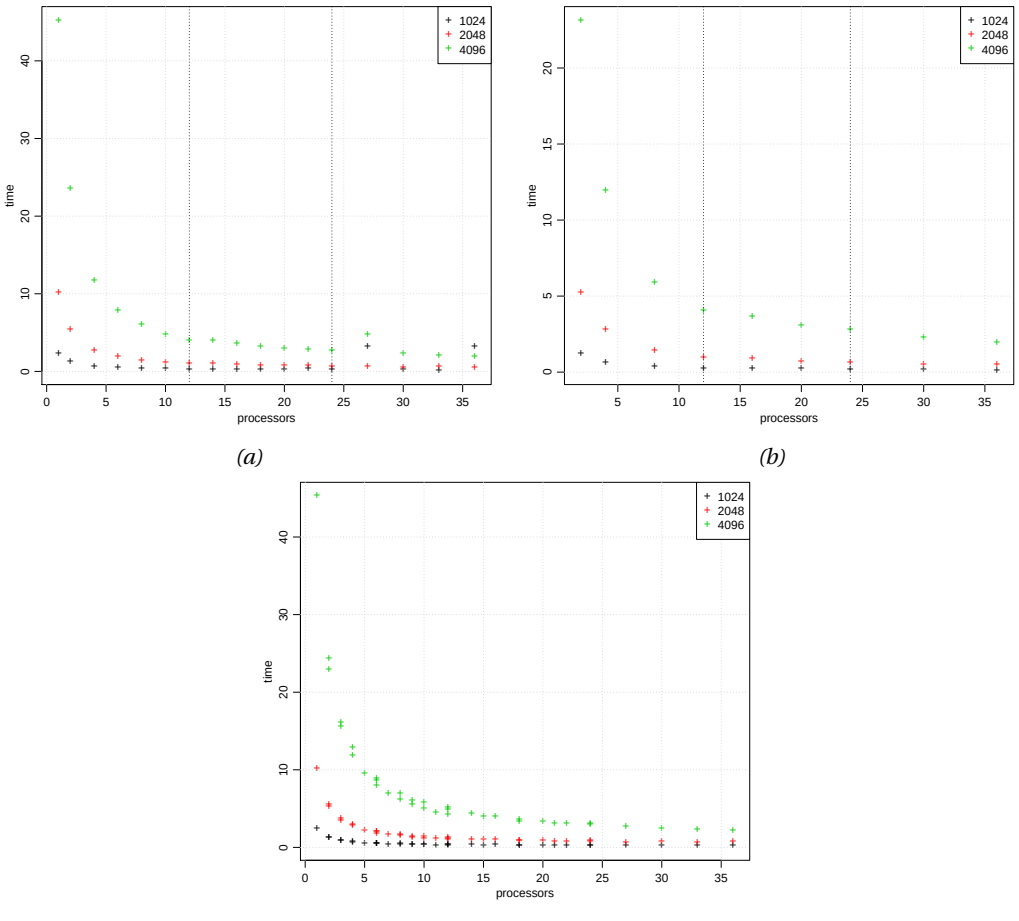

Figure 4.3: Times for running problem with different amount of MPI processes. (a) each MPI process has one thread. (b) each MPI process has two threads. (c) One MPI process is run per node, and the number of threads varies from 1 to 12. (all) The problem is run on as few nodes as possible, and the MPI processes are identically distributed among the nodes. The problem size n is specified in the plots. The vertical lines separate the domain to show when 1, 2 and 3 nodes are used.

4.3 Timing results

4.3.1 Time as a function of p

Timing results were obtained by running the problem on different number of MPI processes with one thread. Here up to three nodes were used, and a new node was only used when there were no more free processors in the previous nodes. This was done for different problem sizes and plotted in Figure 4.3a. Sending between MPI processes on a single node should be faster than sending between nodes. This trend is clear in the figure, as the first points after utilizing an extra node (after the vertical lines), clearly do not follow the trend of the previous points.

In Figure 4.3b, the same experiment is done, but with two threads instead of one. So here only up to 6 MPI processes were used on each node. Although it is clear from Figure 4.2 that this should be worse than only using MPI processes, it is hard to distinguish Figure 4.3a and Figure 4.3b. To investigate how the time changed with only using threads on each node, the problem was run on 1 to 12 threads on 1 to three nodes. In Figure 4.3c this is plotted and we see the behavior is the same as in the other figures. It is reasonable to assume that for if run on more nodes, the time would eventually start to grow because of the overhead. There are some larger times at the end of Figure 4.3a, but this is probably just noise.

4.3.2 Time as a function of n

The time of the algorithm should be of order $O(n^2 \log(n))$. To investigate how the time scales with n, $time/n^2$ was plotted as a function of n. This is displayed in the left plot in Figure 4.4. The data used is the same as in Figure 4.1. One can clearly see an upward trend for larger problem sizes due to the factor $log(n)$. Therefore $time/(n^2 \log(n))$ was also plotted in Figure 4.4. Here the ratio is constant and it can be concluded that the time is of order $O(n^2 \log(n))$. For small times, the overhead becomes significant, and that is probably why the first point is so much higher for 3 and 6 MPI processes.

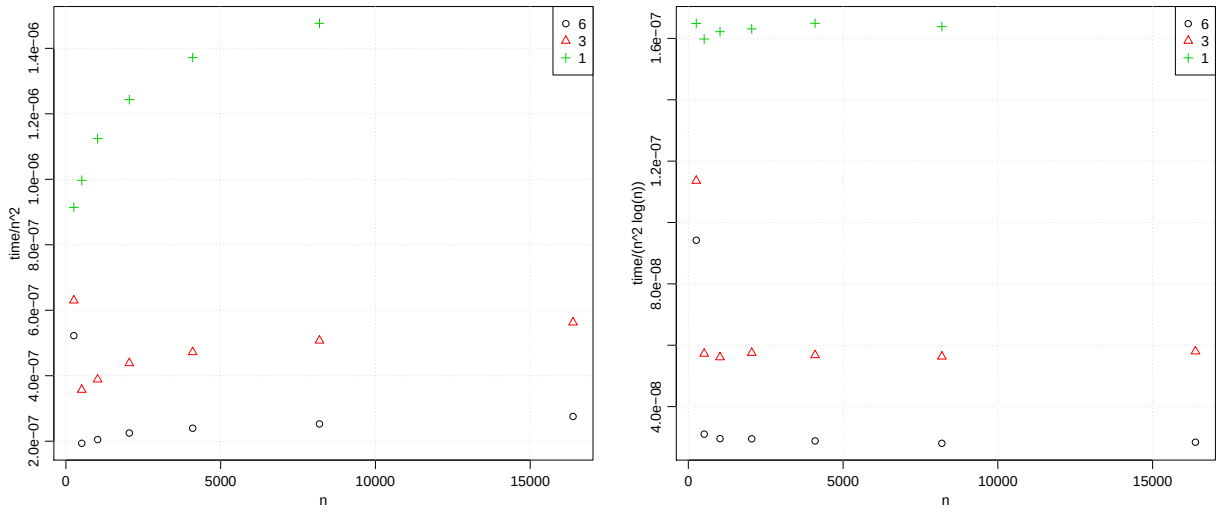

Figure 4.4: Left plot is $Time/n^2$ as function of n. The right plot is $Time/(n^2 \log(n))$. The data is the same as in Figure 4.1.

4.4 Speedup and Efficiency

While Figure 4.3 displays the times, it is hard to get a lot of information from the plots. A useful diagnostics tool is the speedup $S_p = T_1/T_p$, where T_p is the time used with p processors. If there is perfect speedup, i.e. for p processors the time $T_p = T_1/p$, then the speedup should be a linear function of the nr. of processors, with slope

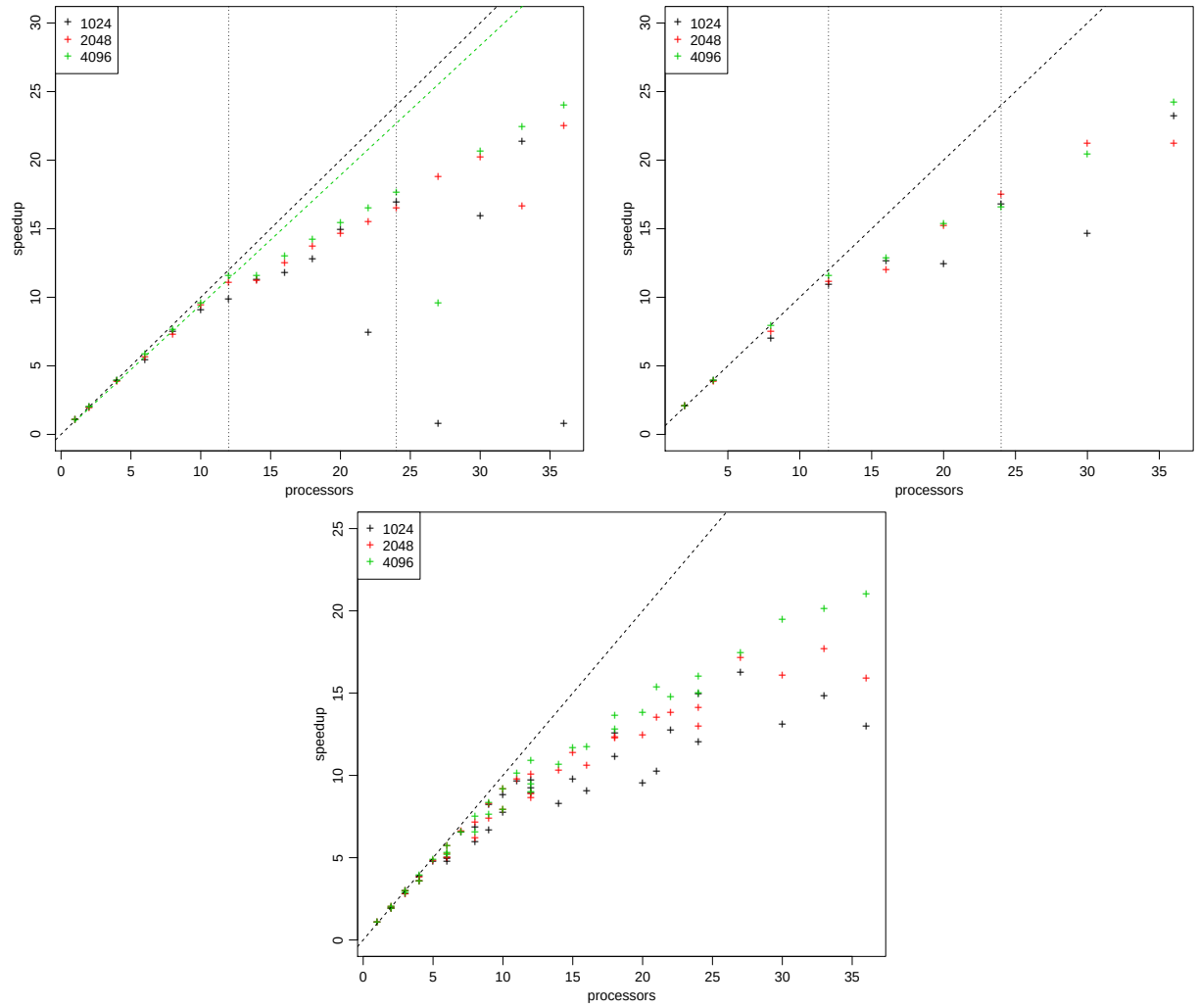

Figure 4.5: Speedup for running the problem with different amount of MPI processes. In the upper left figure each process has one thread. In the upper right figure each process has two threads. In the bottom figure only one MPI process is run per node and the number of threads varies between 1 and 12. The problem is run on as few nodes as possible, and the MPI processes are identically distributed among the nodes. There are drawn vertical lines to show when a new node is utilized, and a line with slope 1. The green line is the expected speedup from the linear network model (4.3) for $n = 4096$. The problem size n is specified in the plots.

1. The speedup, along with a reference line of slope 1, is plotted in Figure 4.5.The setup is exacly the same as in Figure 4.3.

From this plot it is clear that the speedup is better for larger problem sizes. This makes sense as the overhead becomes less significant as the time increases. Also note that there are some noisy points, especially for the smallest problem size. This was addressed in the beginning of this section. What is clear here is that the noise has a higher impact on small problems as the time is shorter, which makes perfect sense. Also here it is clear that there are more overhead when problem is done on one more node (see points after vertical lines in upper left figure). After the last vertical line the time gets really small and the data gets quite noisy. The trend is nevertheless clear, an increasing amount of nodes decreases the speedup.

When sending data between processes with MPI, the speedup is not expected to be 1. By using the *linear net-*

work model (4.3), a better estimate can be obtained (though still not very accurate).

$$\begin{aligned} \tau_s(b) &= \tau_c + \gamma b, \\ T_p &= \frac{T_1}{p} + \tau_s\left(8\frac{n^2}{p}\right), \\ S_p &= \frac{T_1}{T_p}. \end{aligned} \tag{4.3}$$

Here τ_s is the time to communicate b bytes and τ_c is the latency of the network. In Figure 4.5, the green line is from (4.3), with $\tau_c = 10^{-6}$ and $\gamma = 5 \cdot 10^{-9}$. The lines seems to be a good approximation for one node, but not for more than one. This makes sense as it takes longer time to send data between nodes than internally in a node. The efficiency $\eta = S_p/p$ is another diagnostics tool. this variable will take the value 1 if the speedup is perfect. This is shown in Figure 4.6. The plots show pretty much the same as Figure 4.5, but the scale is different, so it is easier to see that there is no case of perfect speedup. It is valuable do note that on 1 node the efficiency is around 0.9 and when 2 nodes is used the effiency immediately drops to right below 0.8 and from there it keeps sinking steadily.

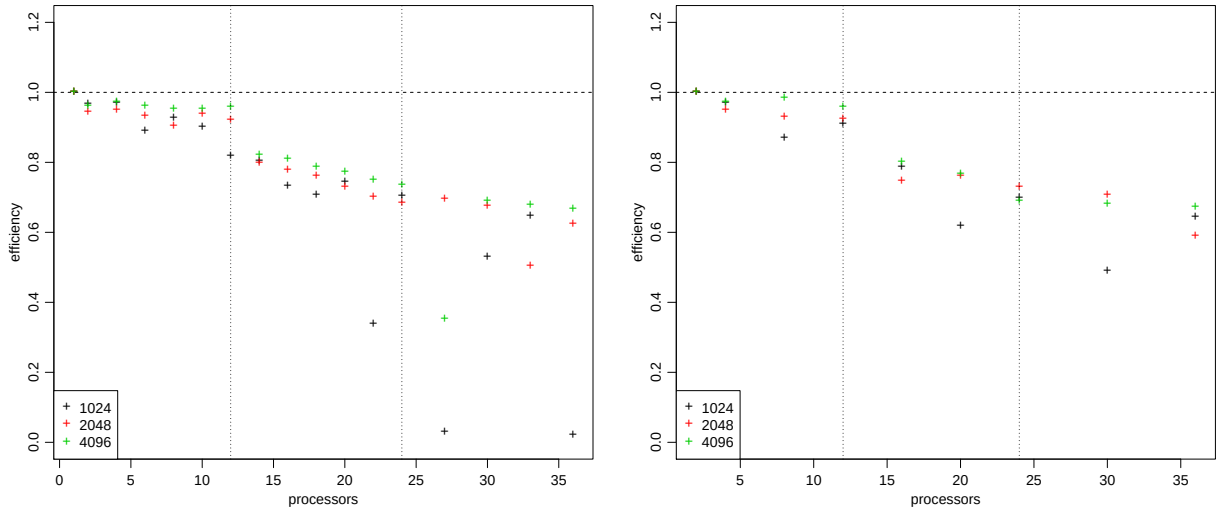

Figure 4.6: Efficiency for running problem with different amount of MPI processes. In the left figure each process has one thread. In the right figure each process has two threads. The problem is run on as few nodes as possible, and the MPI processes are identically distributed among the nodes. There are drawn vertical lines to show when a new node is utilized, and a horizontal line at $y = 1$. The problem size n is specified in the plots.

5 Discussion

5.1 The non-homogeneous case

If one were to solve the Poisson problem with non-homogeneous boundary conditions ie. $u = g(x)$ on $\partial\Omega$, some minor modifications would have to be done. The matrix system that is generated through this algorithm does not include the effect of the boundary elements, and in practice this is the same as assuming that they are equal zero. The second order differential operator on the elements closest to the boundary are on the form

$$-\frac{\partial^2 u_{i,j}}{\partial x^2} = -\frac{u_{i-1,j} - 2u_{i,j}}{h^2} \tag{5.1}$$

The incrementations of i, j will depend on which boundary the element is close to, but the form of the operator is nevertheless the same. The effect on equation 1.3 is that the boundary element is added to the left side. In the

homogeneous case this does not affect the numerical algorithm, but in the non-homogeneous case the boundary value needs to be added to the right hand side as well. This is implemented before the FST is done and does not have any impact on the parallel implementation.

5.2 Variations in the loading function f

The loading function simply needs to be evaluated in all the grid points and multiplied by the steplength, before the FST-procedure starts. The only thing that will differ from the serial code is the displacement each MPI process have. In Figure 5.1 three solutions to the poisson problem have been plotted, all with different loading functions.

5.3 Variations in the domain Ω

By defining the domain $\Omega = (0, L_x) \times (0, L_y)$ but keeping the same number of grid points in each direction the discretization of the laplacian operator would be changed. In the unit square the discrete laplacian is defined as

$$\Delta u_{i,j} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h^2}. \tag{5.2}$$

In the new domain the operator would be defined as

$$\Delta u_{i,j} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h_x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h_y^2}. \tag{5.3}$$

Where $h_x = L_x/n$ and $h_y = L_y/n$ defines the new steplengths in each direction. Notice that the infamous 5-point formula now takes quite a different form and some rewriting is necessary. The diagonalized matrix system will now end up looking as

$$\left(\frac{1}{h_x^2} \underline{T}\, \underline{U} + \frac{1}{h_y^2} \underline{U}\, \underline{T} \right)_{i,j} = f_{i,j} \tag{5.4}$$

By continuing the diagonalization procedure st. $\underline{T} = \underline{Q}\, \underline{\Lambda}\, \underline{Q}^T$ and defining $\underline{\tilde{U}} = \underline{Q}^T\, \underline{U}\, \underline{Q}$ one is left with the matrix system

$$\frac{1}{h_x^2} \underline{\Lambda}\, \underline{\tilde{U}} + \frac{1}{h_y^2} \underline{\tilde{U}}\, \underline{\Lambda} = \underline{\tilde{F}}. \tag{5.5}$$

Notice that the right hand side can not be initially multiplied with the steplength h^2 since the two terms on the left hand side now have different coefficients. This has to be done in step 2 of the algorithm, and would be implemented in the following way;

$$\tilde{u}_{i,j} = \frac{\tilde{f}_{i,j}}{\lambda_i / h_x^2 + \lambda_j / h_y^2}. \tag{5.6}$$

The last part of the algorithm would not need any further changing.

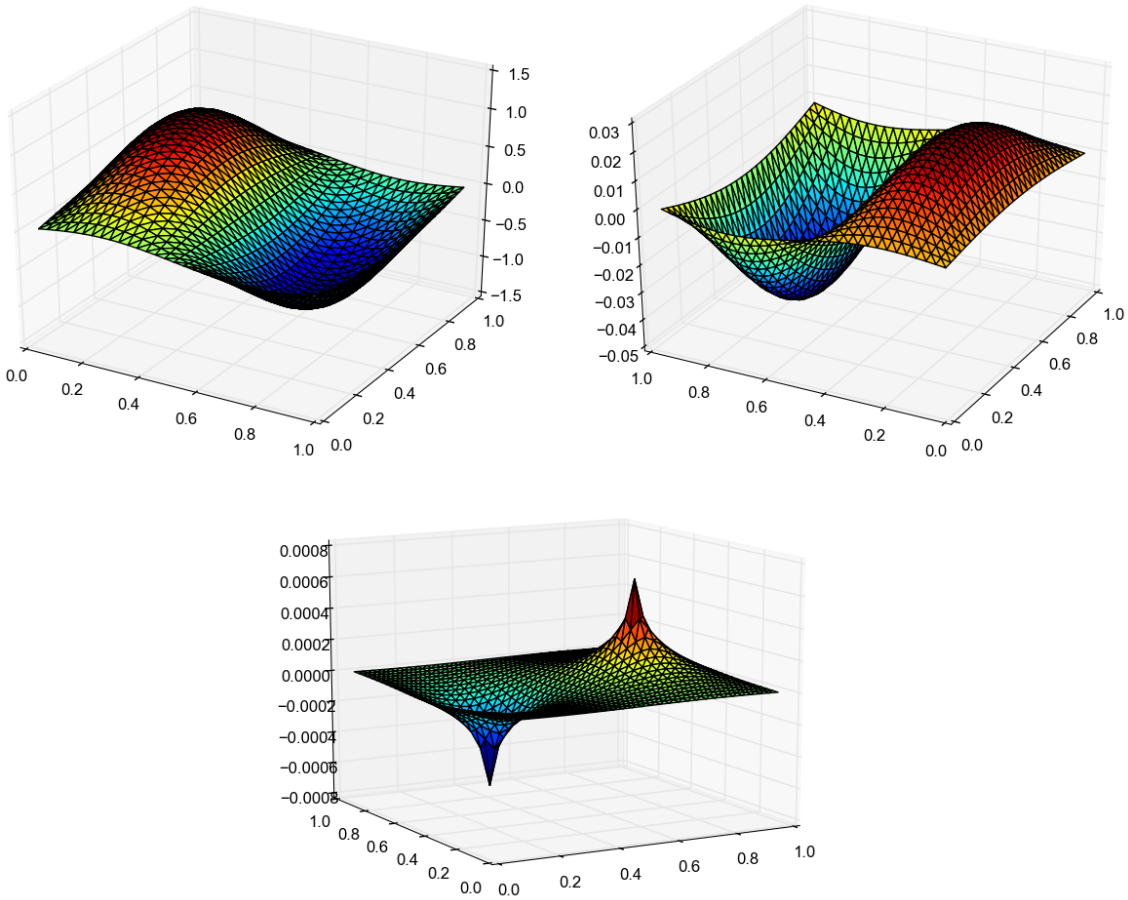

Figure 5.1: The obtained solutions to Poisson's equation using different source functions. The upper left, right and lower subfigure show the solutions for $f(x, y) = 5\pi^2 \sin(\pi x) \sin(2\pi y)$, $f(x, y) = e^x \sin(2\pi x) \sin(\pi y)$ *and a sink-source-function, respectively.*

REFERENCES

[1] Einar M. Rønquist, *The Poisson problem in* $\mathbb{R}^2$*: Diagonalization methods*, Department of Mathematical Sciences, NTNU,N-7491 Trondheim, Norway, Revised by Arne Morten Kvarving, 2014.