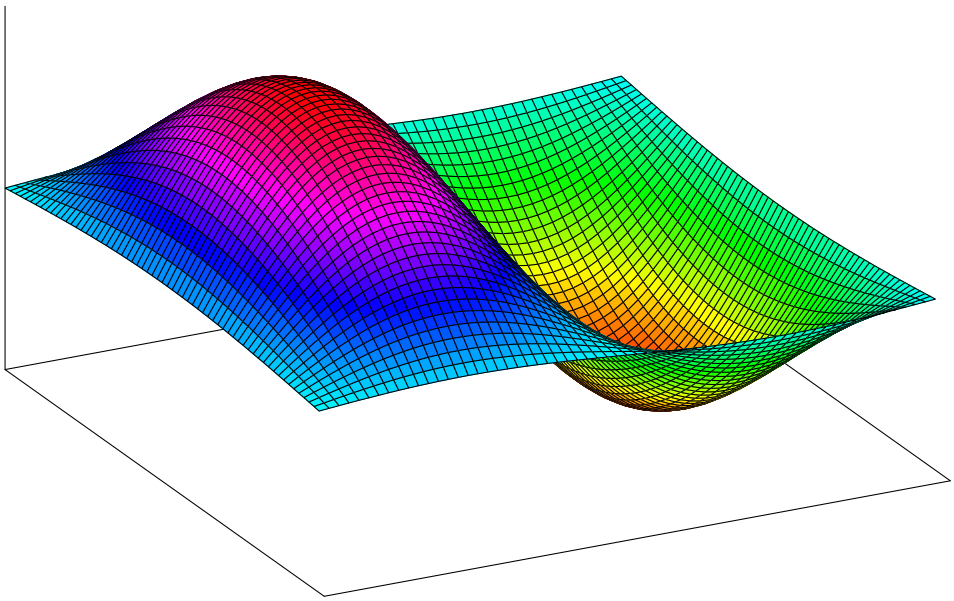
TMA4280 - Introduction to supercomputing
Exercise 6

Vegard Stenhjem Hagen

April 15, 2013

Abstract

In this report the poisson problem is briefly discussed and a strategy for directly solving it numerically on a supercomputer using OpenMP and MPI is presented. The numerical solution is tested for convergence using a readily analytical solvable problem and timed using different combinations MPI-processes and OpenMP threads on the high performance computer *kongull*.



Contents

1 The 2D Poisson problem

1.1 Introduction

The poisson equation on a domain Ω is given as

$$\begin{aligned} -\Delta u &= f \\ u &= 0 \qquad \text{on } \partial\Omega \end{aligned} \tag{1}$$

where f is known, and u is unknown. In this report the main focus will be on solving (1) on the domain $\Omega = (0,1) \times (0,1)$. The problem is discretized on a regular finite difference grid with $n-1$ points in each spatial direction and spacing h between each point. The standard 5-point stencil is used to discretize the Laplace operator $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ (fig.1.1). This yields a roundoff error of $\mathcal{O}(h^2)$. Using the common notation that $u_{i,j} = u(x_i, y_j)$, where $i, j \in \mathbb{Z}^+$ is the discrete coordinates of a point on the grid, equation (1) can be written as

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

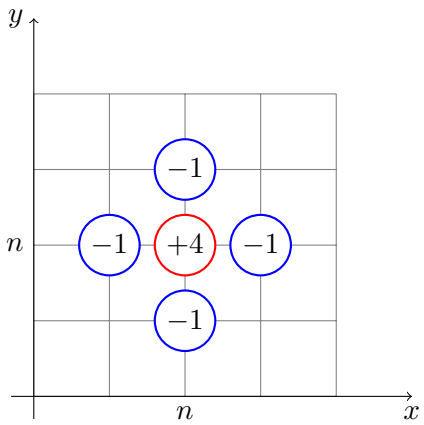


Figure 1.1.1: Illustration of the 5-point stencil on an $n \times n$ grid

1.2 Diagonalization

Let

$$U = \begin{pmatrix} u_{1,1} & \cdots & u_{1,n-1} \\ \vdots & \ddots & \vdots \\ u_{n-1,1} & \cdots & u_{n-1,n-1} \end{pmatrix} \tag{3}$$

and

$$T = \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & & \vdots \\ 0 & & \ddots & & 0 \\ \vdots & & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix} \tag{4}$$

Then

$$(TU)_{i,j} = \begin{cases} 2u_{i,j} - u_{i+1,j} & i = 1, \\ -u_{i-1,j} + 2u_{i,j} - u_{i+1,j} & 2 \le i \le n-2, \\ -u_{i-1,j} + 2u_{i,j} & i = n-1. \end{cases} \tag{5}$$

thus equation (2) can be expressed as

$$\frac{1}{h^2}(TU + UT)_{i,j} = f_{i,j} \quad \text{for} \quad 1 \le i, j \le n-1,$$

or more compact as

$$TU + UT = G, \tag{6}$$

where

$$G = h^2 \begin{pmatrix} f_{1,1} & \cdots & f_{1,n-1} \\ \vdots & \ddots & \vdots \\ f_{n-1,1} & \cdots & f_{n-1,n-1} \end{pmatrix}$$

Since T is a symmetric matrix it can be diagonalized, this means that it can be written on the form

$$T = Q\Lambda Q^T, \tag{7}$$

where $Q = [q_1|q_2|\dots|q_{n-1}]$ is an orthonormal matrix containing the normalized eigenvectors q_i, $1 \leq i \leq n-1$ of T and Λ is a matrix containing the corresponding eigenvalues λ_i along its diagonal.

Combining equations (6) and (7) and multiplying with Q^T from the right and with Q from the left results in

$$\Lambda \underbrace{Q^T U Q}_{\widetilde{U}} + \underbrace{Q^T U Q}_{\widetilde{U}} \Lambda = \underbrace{Q^T G Q}_{\widetilde{G}}. \tag{8}$$

Hence, (6) may be solved in three steps:

Step 1) Compute

$$\widetilde{G} = Q^T G Q \qquad \mathcal{O}(n^3) \quad \text{operations}$$

Step 2) Solve

$$\Lambda\widetilde{U} + \widetilde{U}\Lambda = \widetilde{G}$$

for $\widetilde{U}$, which boils down to solving

$$\widetilde{u}_{i,j} = \frac{\widetilde{g}_{i,j}}{\lambda_i + \lambda_j} \qquad \mathcal{O}(n^2) \quad \text{operations}$$

Step 3) Compute

$$U = Q\widetilde{U}Q^T \qquad \mathcal{O}(n^3) \quad \text{operations}$$

Here $\widetilde{G}, G, Q, T, \Lambda, \widetilde{U}$ and U are all $(n-1) \times (n-1)$ real matrices.

1.3 Eigenvalues and eigenvectors

Define the vectors $\widetilde{q}_i$ as

$$(\widetilde{q}_j)_i = \sin\left(\frac{ij\pi}{n}\right) \tag{9}$$

and observe that T as defined in (4) operated on $(\widetilde{q}_j)_i$ gives

$$(T\widetilde{q}_j)_i = \underbrace{2\left(1 - \cos\left(\frac{j\pi}{n}\right)\right)}_{\lambda_j} \underbrace{\sin\left(\frac{ij\pi}{n}\right)}_{(\widetilde{q}_j)_i}. \tag{10}$$

Thus the vectors $\widetilde{q}_i$ defined in (9) are eigenvectors for the matrix T with corresponding eigenvalues λ_j.

Normalized eigenvectors q_i can be represented as

$$(q_j)_i = \sqrt{\frac{2}{n}} \sin\left(\frac{ij\pi}{n}\right). \tag{11}$$

Using these vectors as columns in a matrix yields the requsted orthonormal matrix Q in equation (7). Also note that

$$Q = Q^T = Q^{-1} \tag{12}$$

1.4 Discrete Sine transform

The discrete sine transform (DST) of a vector $v = [v_1, v_2, \ldots, v_{n-1}]$ is defined as

$$\tilde{v}_j = \sum_{i=1}^{n-1} v_i \sin\left(\frac{ij\pi}{n}\right), \qquad j \in \{1, 2, \ldots, n\}, \tag{13}$$

this can be expressed as

$$\tilde{v} = S(v) \tag{14}$$

while the inverse discrete sine transform of $\tilde{v}$ can be expressed as

$$v = S^{-1}(\tilde{v}). \tag{15}$$

S and S^{-1} are related as

$$S = \frac{2}{n} S^{-1}$$

Observe that the matrix Q is a scaled version of the DST:

$$Q = \sqrt{\frac{n}{2}} S = \sqrt{\frac{2}{n}} S^{-1} \tag{16}$$

By utilising the Fast Fourier Transform, equation (14) and (15) can be solved in $\mathcal{O}(n \log n)$ instead of $\mathcal{O}(n^2)$ as by conventional matrix vector multiplication.

Considering step 1 computing $\widetilde{G} = Q^T G Q$, taking the transpose and using the properties stated in (12), this can be rewritten as

$$\begin{aligned} \widetilde{G}^T &= Q^T G^T Q \\ &= Q(QG)^T. \end{aligned}$$

Using (16) this can further be rewritten to

$$\widetilde{G}^T = \sqrt{\frac{2}{n}} S^{-1} \left(\left(\sqrt{\frac{n}{2}} S(G) \right)^T \right) = S^{-1} \left((S(G))^T \right). \tag{17}$$

In a similar fashion, step 3 can be rewritten as

$$\begin{aligned} U &= Q\widetilde{U}Q^T \\ &= Q(Q\widetilde{U}^T)^T \\ &= S^{-1}\left((S(\widetilde{U}^T))^T \right) \end{aligned}$$

2 Implementation

2.1 Overview

The new solution strategy can then be summarized as follows

Step 1) Compute

$$\widetilde{G}^T = S^{-1}\left((S(G))^T\right) \qquad \mathcal{O}(n^2 \log n) \quad \text{operations}$$

Step 2) Solve

$$\widetilde{u}_{j,i} = \frac{\widetilde{g}_{j,i}}{\lambda_j + \lambda_i} \qquad \mathcal{O}(n^2) \quad \text{operations}$$

Step 3) Compute

$$U = S^{-1}\left((S(\widetilde{U}^T))^T\right) \qquad \mathcal{O}(n^2 \log n) \quad \text{operations}$$

which is then implemented in C using a black-box fortran script for the DST.

The OpenMP and MPI libraries are used to help parallelize the code for use on a supercomputer. The program architecture (fig. 2.1) is constructed to be load balanced where each MPI-process gets a share of the problem to solve. The shares may be slightly uneven distributed depening on if the problem size is divisible by the amount of MPI-processes or not, e.g. a problem size of $n = 512$ (511^2 grid points) divded among three MPI-processes results in one processor getting 171 columns, while the other two get 170 columns. This is only a difference of 0.6% of the total elements and therefore neglible. The OpenMP threads spawned by each MPI-process is done by calls to the *#pragma* compiler directives.

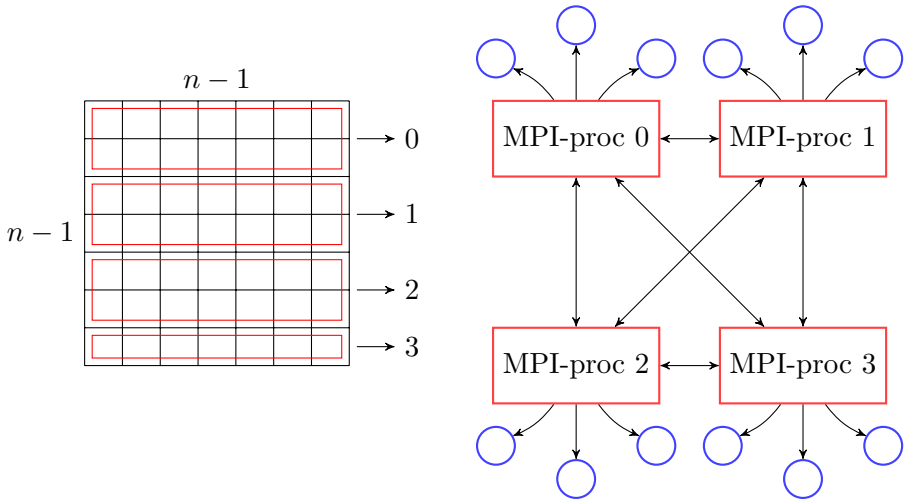


Figure 2.1.1: Illustration of the program architecture. The left shows how the grid is split up among the MPI-processes, notice that there's an uneven distribution since 7^2 is not divisible by 4. The right figure shows how each MPI-process spawns three thread of its own and that each MPI-process communicates with all the other MPI-processes. The number of threads spawned by each MPI-process is arbitrary.

2.1.1 Transpose

In order to perform the transpose operation all the MPI-processes need to communicate with each other, in order to simplify this procedure every MPI-process keep track of who owns which rows and what their displacements are so as to know where to send its data once the transpose operations is called upon. This is accomplished by using the command *MPI_Alltoallv* and packing and unpacking the buffer correctly. A schematical layout of how this is done can be viewed in figure 2.1.1.

The C code listing for the transpose can be found in the appendix.

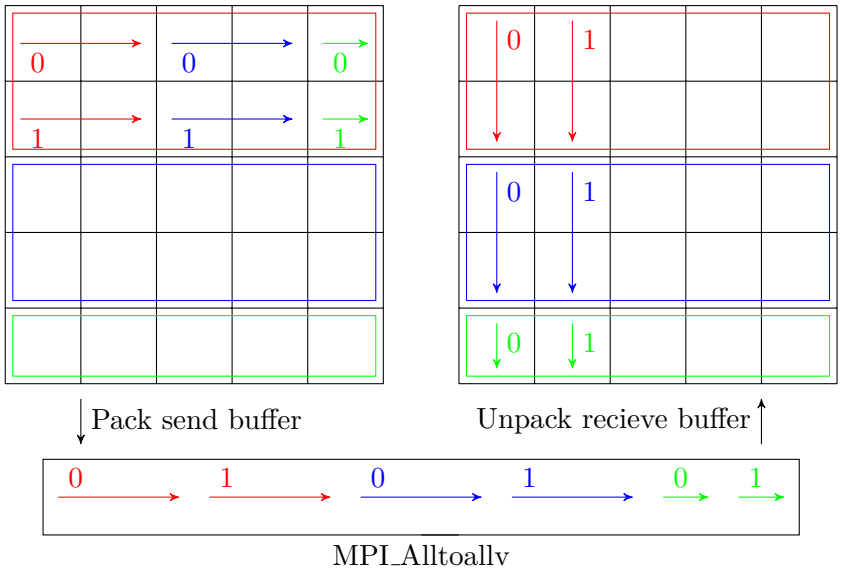


Figure 2.1.2: A schematical layout of how the transpose operation is carried out over the MPI-processes.

3 Numerical results and performance analysis

3.1 Hardware

All the numerical results are obtained by running the program on the high performance computer *kongull*.

Kongull consist of 93 compute nodes, of which 44 nodes have 48 GiB each, while the other 49 nodes have 24 GiB of memory. Each node consists of a HP DL165 G6 server with 2 6-core 2.4 GHz AMD Opteron 2431 (Istanbul) processors, with 6 512 KiB L1 cache and a common 6 MiB L3-cache. The difference in memory is of low concern for this report as the program will not exceed 7 GiB memory on a single node. Assuming a problem size of $n = 16384 = 2^{14}$ the memory requirements for storing the matrix is 8 Bytes $\cdot (2^{14})^2 = 2^{31}$ Bytes $= 2$ GiB. While doing the transpose memory requirement will peak at triple this because both the send and recieve buffers need room to store the matrix. The other variables stored are neglible compared to this as they have memory requirements of at most $\mathcal{O}(n)$. Estimating a generous 1 GiB for the rest of the variables should therefore be more than enough, and hence 24 GiB of memory or more is never obtained.

The internal network of the kongull cluster is based on a fat tree layout, with slower connections towards the *leaves* of the connection tree. Each login- and I/O-node is connected to a central switch, HP ProCurve 6600-24XG, with 10GbE SPF+ connectors. Two rack switches, ProCurve 2910al-48G are connected to the central switch with 10GbE SPF+ connectors. Each compute node is connected to a rack switch with 1Gb Ethernet network connection to the outside world, via two SR tranceivers on the central switch. [2] Network latency is expected to be a bottleneck for the solver.

The program was compiled using the intel compiler version 11.1.059 and a cmake script with the option *-DCMAKE_BUILD_TYPE=Release*

3.2 Numerical results

The numerical tests were performed on *kongull* using different amount of nodes (N), MPI-processes per node (M) and threads per MPI-process (T_M) which gives a total of $P = N \times M \times T_M$ processors used. The seed function used was

$$f(x, y) = 5\pi^2 \cdot \sin(\pi x) \cdot \sin(2\pi y)$$

which has the analytical solution

$$u(x, y) = \sin(\pi x) \cdot \sin(2\pi y)$$

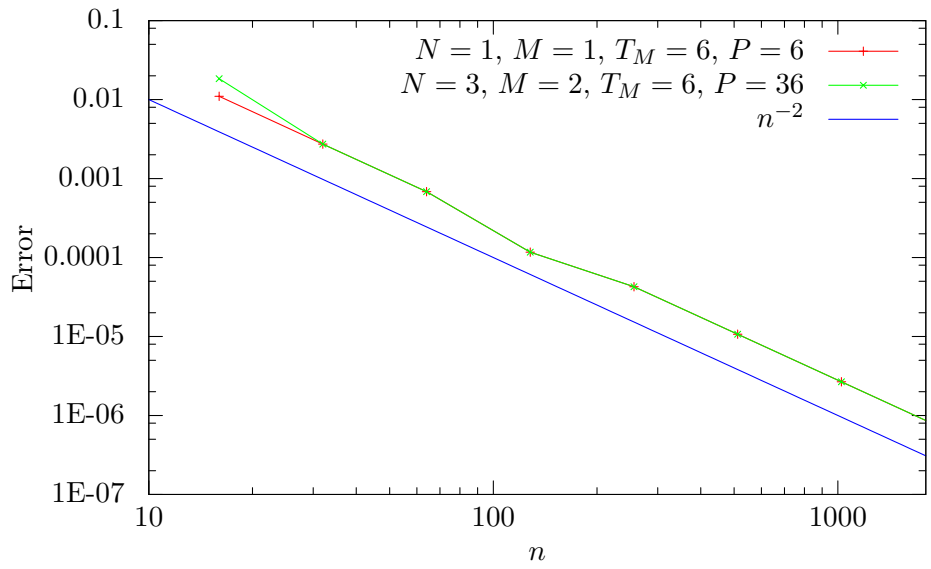


Figure 3.2.1: Pointwise error as a function of grid size for two different processor configurations. Notice that the pointwise error for both configurations is proportional to n^{-2} which indicates that there are no convergence issues since the algorithm predicts convergence on the order of $\mathcal{O}(h^2)$ and h is inversely proportional to n in this case.

for the poisson equation (eq. (1)). This makes it possible to check for convergence issues by comparing the numerical results to the analytical solution. A series of preliminary test runs with different combinations of nodes and MPI-processes per node were made on kongull, some of the results are summarized in table 3.2.1.

Table 3.2.1: Select runs of two different combinations of nodes (N), MPI-processes per node (M) and threads per MPI-process (T_M) for different problem sizes n.

	$N=1,\ M=1,\ T_M=6,\ P=6$			$N=3,\ M=2,\ T_M=6,\ P=36$		
n	Error	τ [s]	$\tau/n^2\log(n)$ [s]	Error	τ [s]	$\tau/n^2\log(n)$ [s]
32	$2.73\cdot10^{-3}$	$3.59\cdot10^{-4}$	$7.01\cdot10^{-8}$	$2.73\cdot10^{-3}$	$4.42\cdot10^{-2}$	$8.64\cdot10^{-6}$
256	$4.27\cdot10^{-5}$	$2.26\cdot10^{-2}$	$4.31\cdot10^{-8}$	$4.27\cdot10^{-5}$	$2.87\cdot10^{-2}$	$5.47\cdot10^{-8}$
2048	$6.67\cdot10^{-7}$	$2.00\cdot10^{+0}$	$4.33\cdot10^{-8}$	$6.67\cdot10^{-7}$	$6.39\cdot10^{-1}$	$1.39\cdot10^{-8}$
16384	$8.89\cdot10^{-9}$	$1.64\cdot10^{+2}$	$4.36\cdot10^{-8}$	$8.89\cdot10^{-9}$	$3.67\cdot10^{+1}$	$9.77\cdot10^{-9}$

3.2.1 Verification of correctness

A plot of the error as a function of problem size (fig. 3.2.1) shows that the error decreases proprotional to n^{-2}, which is consistent with the error decreasing as $\mathcal{O}(h^2)$ since h is inversely proportional to n. It can therefore be concluded that there are no convergence issues with the solver.

3.2.2 Timing

It is expected that the runtime τ per $n^2 \log n$ for large values of n will approach a constant value as the program is estimated to run in $\mathcal{O}(n^2 \log n)$ time. By looking at figure 3.2.2 this appears to be the case. Also note that the addition of more processors actually increases the average time spent calculating each element for small values of n, this is due to the increased overhead when using more processors.

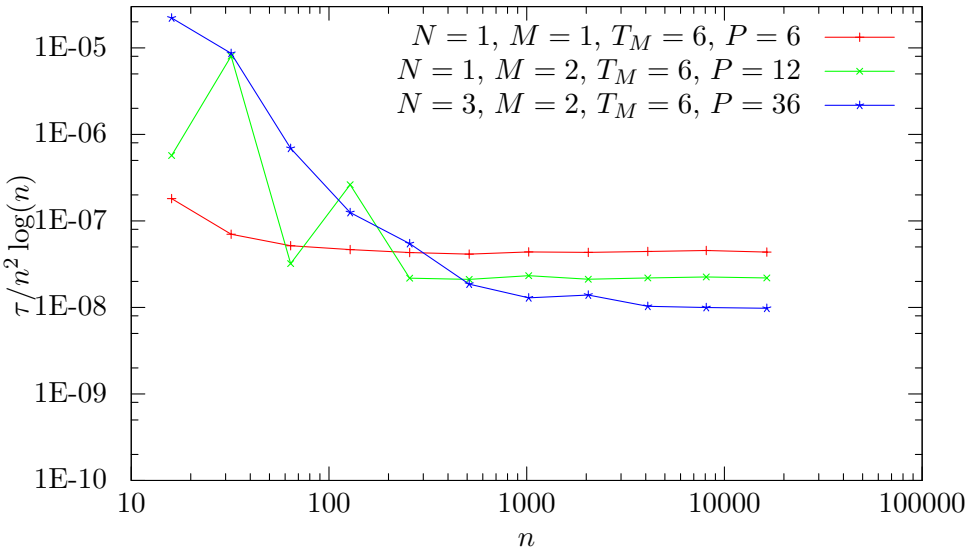


Figure 3.2.2: Total runtime τ divided by $n^2 \log n$ as a function of n. Observe that the runtime per $n^2 \log n$ appear to converge to a constant for large n which is a strong indicator that the program runs in $\mathcal{O}(n^2 \log n)$ time. The increased value of $\tau/n^2 \log n$ for $P = 36$ for small n is due to increased overhead and network latency when having to communicate between more nodes. This overhead is however neglible when the problem size increases.

3.2.3 Speedup and parallel efficency

The speedup S_P of an algorithm run on more processors is defined as the time it takes to run on one processor τ_1 divided by the time it takes to run on P processors τ_p,

$$S_P = \frac{\tau_1}{\tau_P}. \tag{18}$$

The ideal speedup of an algorithm is directly proportional to the number of processors it is run on. Because of increased overhead when utilizing several processors ideal speedup is not obtainable. A measure of parallel efficency η_P is defined as

$$\eta_P = \frac{S_P}{P} \tag{19}$$

which generally decreases as P is increased because of the extra overhead and network latency.

A simple numerical analysis of the speedup and parallel efficency done on a single node on Kongull using only one MPI-process is summarized in table 3.2.3. A plot of the table can be found in figure 3.2.3. Notice that η_P and S_P decreases as P increases

Using multiple MPI-processes on a single node offers a better speedup since there is much less overhead and communication time is small. This is backed up by the numerical results as listed in table 3.2.3. A plot of the parallel efficency can be seen in figure 3.2.3. Observe that the parallel efficency is close to unity when all the processors on the node are run as MPI-processes as compared to when all the processors are used as OpenMP threads when the parallel efficency is down to 0.65.

When the number of nodes is increased one would expect the the parallel efficency to drop substantially as there is network latency involved. By looking at timing results obtained from running the program over three nodes in table 3.2.3 a decreased efficency is observed, as the number of processors is tripled, the run time is only about halved as compared to table 3.2.3. Again as with the single node case, running the algorithm with only MPI-processes yields the fastest result.

From figure 3.2.3 it can be observed that for small problem sizes ($n < 1024$) there is very little payoff in using many processors as the prallel efficency is shown to be very low. The hybrid model has a somewhat better payoff for smaller problems, but it is still inefficient.

Table 3.2.2: Timing results on a single node ($N = 1$) using one MPI-process ($M = 1$) and different amounts of threads T_M with a problem size n of 16384. Recall that $P = N \times M \times T_M$

P	τ_P [s]	$S_p = \tau_1/\tau_p$	$\eta_p = S_p/P$
1	790.7	1.00	1.00
2	416.1	1.90	0.95
3	289.9	2.73	0.91
4	226.3	3.49	0.87
6	164.6	4.80	0.80
8	132.6	5.96	0.75
10	114.6	6.90	0.69
12	100.9	7.84	0.65

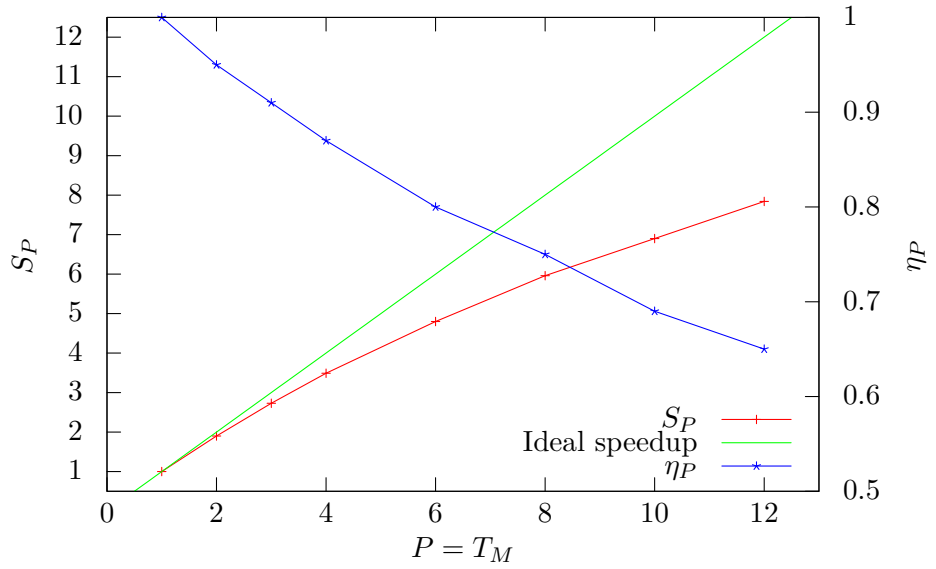


Figure 3.2.3: A plot of the speedup (S_P) and parallel efficency (η_P) for different amounts of processors (P) utilized. Take note that the second axis is different for η_P and S_P.

Table 3.2.3: Listing of timing results as a comparison between MPI (M) and OpenMP processes per thread (T_M). Results are obtained using only one node and a problem size of $n = 16384$.

M	T_M	P	τ_P [s]	$S_p = \tau_1/\tau_p$	$\eta_p = S_p/P$
1	12	12	100.9	7.84	0.65
2	6	12	82.7	9.56	0.80
3	4	12	76.2	10.38	0.86
4	3	12	72.8	10.86	0.91
6	2	12	69.6	11.36	0.95
12	1	12	66.4	11.91	0.99

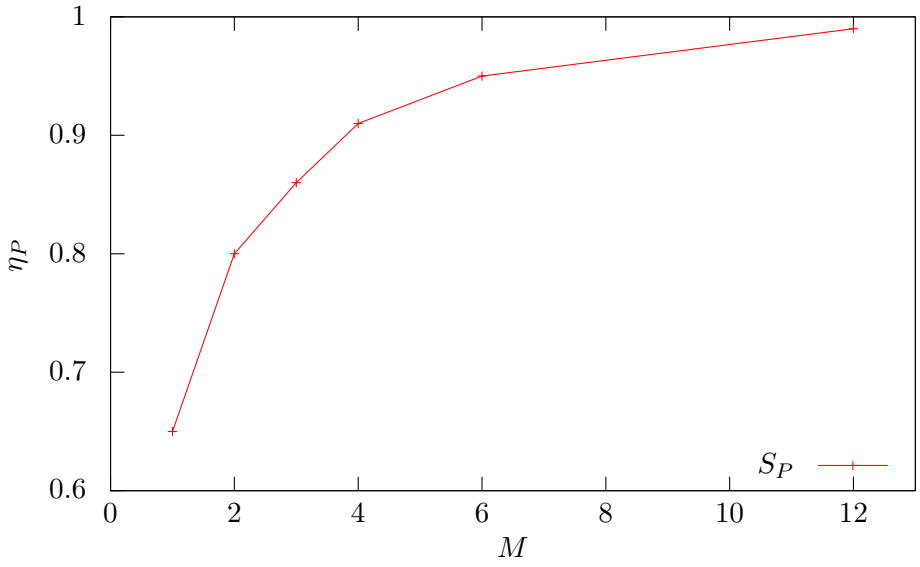


Figure 3.2.4: A plot of the parallel efficency η_P as a function of MPI-processes (M) on a single node. The total number of utilized processors is always $P = 12$ as the processors not used as a MPI-process is utilized as a thread.

Table 3.2.4: Timing results on three nodes ($N = 3$) with different combinations of MPI-processes per node (M) and OpenMP threads per MPI-process (T_M). The total number of processors utilized is 36 in each case (recall that $P = N \times M \times T_M$).

M	T_M	τ_P [s]	$S_p = \tau_1/\tau_p$	$\eta_p = S_p/P$
1	12	42.9	18.43	0.51
2	6	36.6	21.60	0.60
3	4	34.4	22.99	0.64
4	3	33.3	23.74	0.66
6	2	32.6	24.25	0.67
12	1	32.0	24.71	0.69

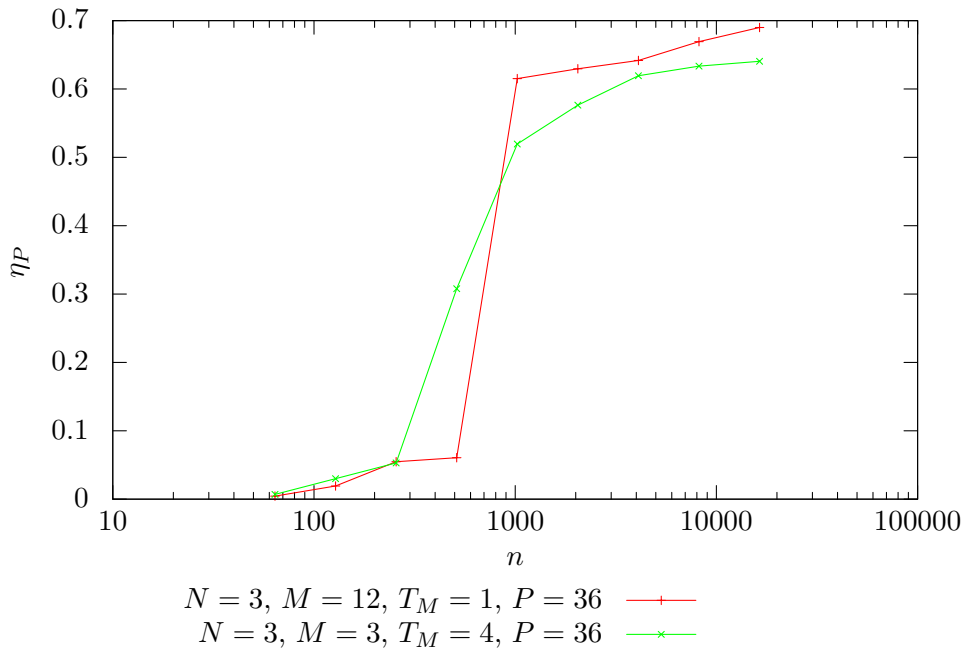


Figure 3.2.5: Parallel efficency η_p for two configurations of MPI-processes (M) and OpenMP threads per MPI-process (T_M) as a function of problem size.

3.3 Summary

The solver is found to give correct results within the error estimate for an analytical solvable seed function. When on a single node the best speedup is acheieved by using all available processors as MPI-processes instead of spawning OpenMP threads. This is because of increased overhead and iterations in the transpose iteration when using OpenMP-threads compared to MPI-processes. On three nodes the best result is also obtained when all processors are utilized as MPI-processes, as can be seen in table 3.2.3. Parallel efficency however suffers when using more than one node because of network latency. Parallel efficency also suffer when the problem size is relatively small, this is due to the added overhead when using multiple processors drowning the time used for actually solving the problem.

Tests were made using more than 36 processors with $n = 16384$, the results indicate that the parallel efficency η_p drops slowly when increasing the number of processors used. Using three nodes the best η_P obtained was 0.69, when using 6 nodes this was down to 0.62 using only the MPI model. On 12 nodes the best efficency was obtained using a hybrid model with $M = 6$ and $T_M = 2$ with $\eta_P = 0.56$, compared to $\eta_p = 0.46$ using the pure MPI model. This behaviour can be explained using figure 3.2.3 where the problem size for each node is small enough that the hybrid model offers advantage.

The transpose operation uses a triple nested for-loop that is executed approximately $M \cdot (n/M) \cdot (n/M) = n^2/M$ times each time the transpose operations is called.

The transpose operation contains a triple nested for-loop that can not be split up among OpenMP threads. This triple nested for-loop is is executed approximately $M \cdot (n/M) \cdot (n/M) = n^2/M$ times per MPI-process each time the transpose operation is called upon. with a problem size of $n = 16384 = 2^{14}$ and on a signle node, using only one MPI-process and 12 threads this amount to once processor having to execute the triple nested for-loop approximately 268 million times while the 11 others are idle. Using a pure MPI-model on the other hand, each processor only need to run 22 million iterations and it can be done in parallel, so the MPI-model is much more load balanced for this operation. However on a distributed system there will be network latency to counter the effect parallelisation of the triple nested loop since more MPI-processes means more network traffic. By looking at table 3.2.3 the hybrid model does not lie far behind the pure MPI-model, which indicates that

network latency is a substantial factor when it comes to parallel computing.

4 Solver capabilities

The solver can solve for different seed functions $f(x, y)$ by changing the *return* statement in the *evalFunc* function in the code. This will change the G matrix in step one to correspond with the given function. The max pointwise error will show false information unless changed as well. Some plots of solutions for different seed functions can be found in figure 4.0.1 to 4.0.3. Axis desciption is dropped since there is no real information in the plots and they look prettier that way.

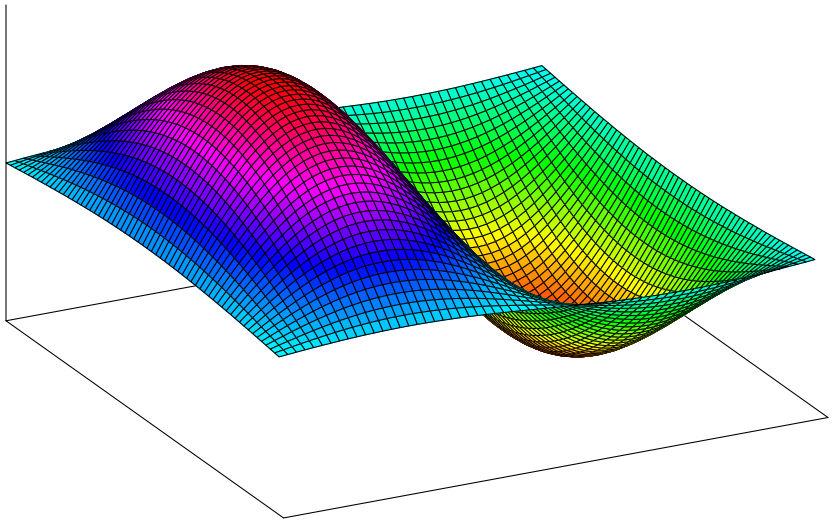


Figure 4.0.1: Plot of the solution u for $f(x, y) = 5\pi^2 \sin(\pi x) \sin(2\pi y)$. Problem size $n = 64$.

4.1 Extending capabilities and improving the solver

The current algorithm only support homogeneous Dirichlet boundary conditions ($u = 0$ on $\partial\Omega$. The algorithm can be further generalized by adding support for non-homogeneous boundary conditions. This can be done by adding a lifting function to the right hand side of equation (1) and storing all boundary elements. This would require some change to the code as the boundary is not stored and assumed equal to zero.

Support for rectangular grids can be implemented by having two different step sizes, one for the x-direction, h_x, and one for the y-direction, h_y. There is already support for different grid length, but only for square domains $(0, L) \times (0, L)$, if the different step sizes are implemented it should be a short leap to be able to work on rectangular $(0, L_x) \times (0, L_y)$ domains.

It's possible to modify the FST to allow for grid sizes different from $n = 2^k, \quad k \in \mathbb{Z}^+$, but this comes at the cost of speed.

A bottleneck of the solver is believed to lie in the transpose operation, if this can be improved, running the hybrid model will give better results on kongull than using the pure MPI model in more cases as there is less communication overhead needed in the hybrid model compared to the pure MPI model.

Another improvement to the program would be to implement parallel I/O instead of the current solution where each MPI-process writes to its own file.

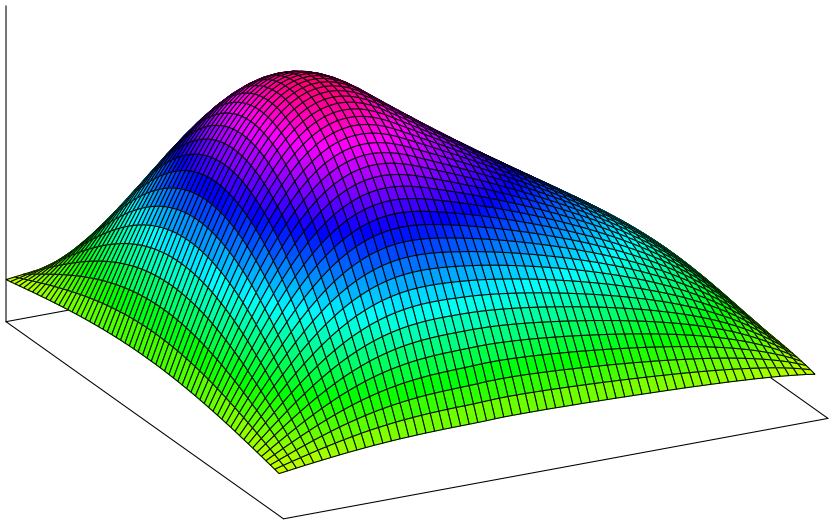


Figure 4.0.2: Plot of the solution u for $f(x, y) = 15$ if $x > 0.4$ and $y > 0.4$ and 0 elsewhere. Problem size $n = 64$.

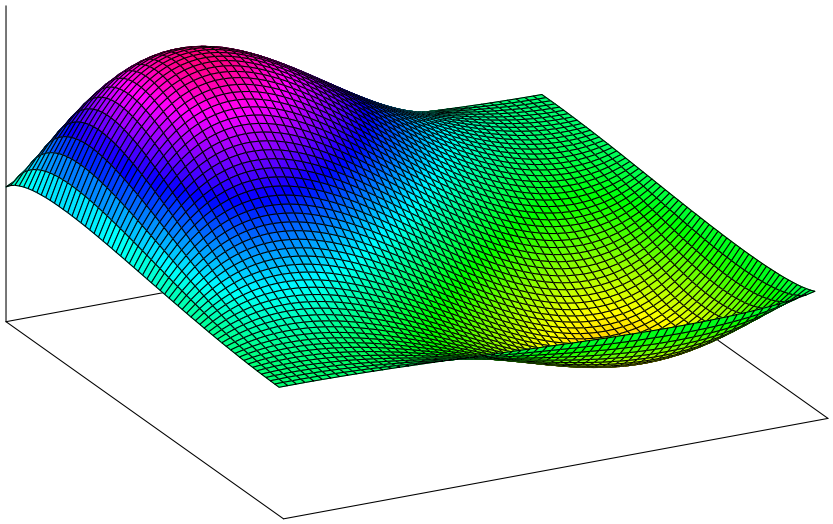


Figure 4.0.3: Plot of the solution u for $f(x, y) = \exp(1 - x - y) - 1$. Problem size $n = 64$.

References

[1] Einar M. Rønquist. Lecture notes in TMA4280, 2011,2012.

[2] Einar Næss Jensen. Hardware. http://docs.notur.no/Members/hrn/kongull.hpc.ntnu.no/kongull-hardware-1/hardware, May 2010.

A Appendix

A.1 C printout of the transpose operation

```
void transposeMPI(Matrix ut, Matrix u){
        int len = ut.displ[ut.comm_size-1]+ut.count[ut.comm_size-1];
        double* sendbuff = (double*)malloc(len*sizeof(double));
        double* recvbuff = (double*)malloc(len*sizeof(double));
        int l = 0;
        for (int i = 0,count = 0; i < ut.comm_size; i++){
                for (int j = 0; j < ut.sizes[ut.comm_rank]; j++){
                        for (int k = 0; k < ut.sizes[i]; k++){
                                sendbuff[count++] = u.data[j][k+l];
                        }
                }
                l += ut.sizes[i];
        }
        MPI_Alltoallv(sendbuff,u.count,u.displ,MPI_DOUBLE,recvbuff,ut.count,ut.displ,MPI_DOUBL
        free(sendbuff);
        for (int i = 0,count = 0; i < ut.m; i++){
                for (int j = 0; j < ut.sizes[ut.comm_rank]; j++){
                        ut.data[j][i] = recvbuff[count++];
                }
        }
        free(recvbuff);
}
```