# TMA4280 - Introduction to supercomputing

Compulsory exercise 2

Pål Christie Ryalen and Sindre Eskeland

March 19, 2016

This is a compulsory exercise in the subject 'TMA4280 - Introduction to supercomputing' at NTNU. We will use the celebrated programming language C to calculate numerical solutions the Poisson problem

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

$$u = 0 \qquad \text{on } \partial\Omega \tag{2}$$

Where $f$ is a given function, while $u$ is sought. Our goal is to solve this problem using message passing interface(MPI) and the language extension OpenMP for parallel programming. It is also of interest to compare computation time, as well as speedup and parallel efficiency with different number of MPI and OpenMP threads.

## 1 Discussion of possible solution strategies

It is well known that equation (**??**) and (**??**) yields a linear system

$$\underline{AU} = \underline{b} \tag{3}$$

where $\underline{A}$ was found by using the five-point formula, $\underline{b}$ is associated with $f$, and $\underline{U}$ is a the finite difference approximation of $u$. In our case, we pick an approximation grid which is equidistant in both directions. For the problem (**??**), the length parameter is $h = \frac{1}{n}$. The problem (**??**) is usually solved numerically by one of two solution methods; direct or iterative. We make a short discussion of the two methods in the following.

An iterative solution method finds the solution of (**??**) by splitting the matrix $\underline{A}$ into a sum of matrices and reorganising the problem. More specifically, $\underline{A}$ may be split into matrices $\underline{L} + \underline{D} + \underline{T}$, where $\underline{D}$ is diagonal, $\underline{L}$ is lower diagonal and $\underline{T}$ is upper diagonal. Here we have the common iteration schemes

$$\underline{DU}^{k+1} = b - (\underline{L} + \underline{T})\underline{U}^k \qquad \text{(Gauss-Jacobi)}.$$

$$(\underline{L} + \underline{D})\underline{U}^{k+1} = b - \underline{TU}^k \qquad \text{(Gauss-Seidel)}.$$

While the Gauss-Jacobi method is rather strict - we are only guaranteed that it will work if $\underline{A}$ is strictly diagonally dominant - the Gauss-Seidel will only need $\underline{A}$ to be symmetric positive definite(SPD)[**?**, p. 49]. It turns out that our problem will converge with the Gauss-Jacobi method even though our candidate $\underline{A}$ is not diagonally dominant, that being said, the convergence is slow. On the other hand, it is easy to parallelise. In contrast, the Gauss-Seidel method is hard to parallelise, since each iteration can immediately be reused. However, this problem can be alleviated if we are clever in the way we do the parallelisation.

On a positive remark, for iterative schemes we may choose storage method as we please. This will be useful when performing parallelisation. We may partition our problem in the following two ways; partitioning the matrix, or partitioning the domain $\Omega$. Using the commands

```
MPI_Dims_create()
MPI_Cart_create()
```

we are able to organise the parallelisation. As none of the mentioned methods were chosen, we will not discuss this further. A candidate method is the famous Krylov methods, in particular the conjugate gradient(CG) method, which demands that the matrix $\underline{A}$ is SPD. We will not discuss this further.

Of the direct methods, the method using the discrete sine transform is compelling. Owing to the fact that $\underline{A}$ is SPD, and that we are able to find a tensor product operator, we have an algorithm that scales well in both in terms of calculation and memory. We spell out how the method works in the following. Given a tensor product operator for $\underline{A}$, we may write system (**??**) on the equivalent form

$$\underline{VX} + \underline{XV}^\top = \underline{B}, \tag{4}$$

where $\underline{V}$ is SPD. Since $\underline{V}$ has the eigen decomposition $\underline{V} = \underline{Q}^\top \underline{\Lambda} \underline{Q}$, where $\underline{Q}$ is orthogonal, and $\underline{\Lambda}$ is a diagonal matrix. We may rewrite equation (**??**) as

$$\underline{\Lambda \tilde{X}} + \underline{\tilde{X} \Lambda} = \underline{\tilde{B}}, \tag{5}$$

where $\underline{\tilde{B}} = \underline{Q}^\top \underline{B} \underline{Q}$. Calculating the product $\underline{Q}^\top \underline{B} \underline{Q}$ will usually demand a healthy $\mathcal{O}(N^3)$ operations. However, as luck would have it, we can use the discrete sine transform to improve the performance. Using the orthogonality of $\underline{Q}$ we have the relations

$$\underline{Q} = \sqrt{\frac{N}{2}} \underline{S}.$$
$$\underline{Q}^\top = \sqrt{\frac{2}{N}} \underline{S}^{-1}.$$

The above relations will allow us the calculate

$$\underline{\tilde{B}}^\top = \underline{S}^{-1} \Big( \big( \underline{SB} \big) \Big)$$

in an improved $\mathcal{O}(N^2 \log(N))$ operations. Similarly, owing to the fact that $\underline{\Lambda}$ is diagonal, we may solve the system (**??**) in $\mathcal{O}(N^2)$ operations, where the $i, j$-th component reads

$$\tilde{x}_{i,j} = \frac{\tilde{b}_{i,j}}{\lambda_i + \lambda_j}.$$

$\Lambda_i$ for $i = 1, \cdots, N$ are the eigenvalues of $\underline{V}$. Lastly, remembering that $\underline{\tilde{X}} = \underline{Q}^\top \underline{X} \underline{Q}$, we solve the system

$$\underline{X} = \underline{S}^{-1} \Big( \underline{S} \big( \underline{\tilde{X}}^\top \big) \Big)^\top$$

in $\mathcal{O}(N^2 \log(N))$ operations. The solution method employed in this project is the discrete sine transform.

## 2 A walkthrough of the cmake-file

We state the relevant code lines in the cmake-file below, and give a short description after that.

1. `cmake_minimum_required(VERSION 2.6)`
2. `enable_language(C)`
3. `enable_language(Fortran)`
4. `option(ENABLE_OPENMP "Enable OpenMP support?" ON)`
5. `option(ENABLE_MPI "Enable MPI support?" ON)`
6. ```
   if(ENABLE_MPI)
     find_package(MPI)
   endif()
   ```
7. ```
   if(MPI_FOUND)
     add_definitions(-DHAVE_MPI=1)
     set(INCLUDES ${INCLUDES} ${MPI_INCLUDE_PATH})
     set(DEPLIBS_C ${DEPLIBS_C} ${MPI_C_LIBRARIES})
     set(DEPLIBS_F ${DEPLIBS_F} ${MPI_Fortran_LIBRARIES})
   endif()
   ```
8. ```
   if(ENABLE_OPENMP)
     find_package(OpenMP)
     if(OPENMP_FOUND)
       add_definitions(-DHAVE_OPENMP=1)
       set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
     endif()
   endif()
   ```
9. `include_directories(${INCLUDES})`
10. `add_library(common fst.f)`
11. `add_executable(poisson poisson.c)`
12. `add_executable(poisson-f poisson.f)`
13. `target_link_libraries(poisson common ${DEPLIBS_C})`
14. `target_link_libraries(poisson-f common ${DEPLIBS_F})`
15. `set_target_properties(poisson PROPERTIES LINKER_LANGUAGE C)`

In the line **??** we impose that the version is not too old. The command **??** enables support for the language `C` in CMake. In **??**, we have also included the support to the `Fortran`-language, since it is employed in the code for the discrete sine transform. In the lines **??** and **??** we allow support for OpenMP and MPI, respectively. In point **??** we load the MPI package if `ENABLE_MPI` is `ON`. In the first place in point **??** we let the compiler know that we wish to compile with MPI. In the places below - the places that begin with `set(` - we tell the compiler where the MPI-libraries are found. The chunk **??** has the same purpose as **??** and **??**, only this time for OpenMP. Point **??** sends the `INCLUDE` directories to the compiler. Point **??** adds the file `fst.f` into a library called `common`. The library will be linked to the executables, as will be decsribed soon. Point **??** and **??** adds executables for the `C`- and `Fortran`-code, respectively. The commands **??** and **??** links the `common`-library to the executables of the `C` and `Fortran` files, respectively. Lastly, the command in **??** reassures that the `poisson`-executable is linked as a `C` program.
   ```

# 3 Regarding kongull

As can be found in the webpage**?**, `kongull` has the following key properties

1. 2 x AMD Opteron model 2431 6-core (Istanbul) processors.
2. 2400 GHz core speed.
3. 667 MHz (48 GiB nodes) or 800 MHz (24 GiB nodes) bus frequency.

Property **??** allows us to use 12 processors per node, and maximum 4 nodes. To compile and execute we needed to run the following commands

```
module load intel/compilers/11.1.059.                          (6)
module load cmake.                                             (7)
CC=icc FC=ifort cmake .  -DCMAKE_BUILD_TYPE=Release.           (8)
make                                                           (9)
qsub job.sh                                                    (10)
```

To compile with `Fortran` on `kongull` it is important to use command (**??**) to load a `Fortran` compiler, the number specifies a version with better performance just `module load intel`. Command (**??**) creates the `Makefile` which compiles with command (**??**). To submit a job, a `shell`-script needs to be written. The `shell`-script contains information about number nodes, of MPI threads and OpenMP threads to be used. When using 1 MPI thread `kongull` can only use 1 node. So the maximum number of OpenMP threads with 1 MPI thread is 12, this is because one can only specify the number of MPI threads per node on `kongull`. For more information about the `shell` script, see **?**.

# 4 A short explanation of the finished program

We created a program solving equation (**??**), with boundary conditions (**??**) using the method described in section **??**, and called it `poisson`. The program itself is made by combining several functions, the most important ones are presented in table **??**. We were given functions for discrete sine transform, but this will not be discussed further. `poisson` takes no argument, and will calculate

| | |
|---|---|
| `double solution(double x, double y)` | Takes coordinates, `x`, `y`, returns the value of `u` at that coordinate. |
| `double b_func(double x,double y)` | Takes coordinates, `x`, `y`, returns the value of `f` at that coordinate. |
| `void domDivide(int size, int n, int **vec)` | Divides `n` points on the number of processors, `size`. Returns values to the matrix `vec`. `vec[rank][0]`, `vec[rank][1]` gives the start- and end- point for processor `rank`. Size of `vec` should be `size`×2. |
| `void paralleltransp(int size,int rank, int m,int l,int **vec,double **b)` | Transposes the matrix `b`, with size `m`×`l` globally. |
| `void solvepoisson(int n,int size,int rank)` | Solves Poisson's equation, and compares the result to the analytical solution. |

Table 1: Explanation of important functions in `poisson`.

a numerical approximation of the solution, and compare it to the analytical solution for problem sizes $2^k, k \in \{2, \cdots, 14\}$. The problem sizes are not arbitrarily chosen, discrete sine transform needs to work on numbers of size $2^k - 1$, thus $n - 1$ is the number of internal nodes. To change which problem to solve, one can simply change the output of the `solution` and `b_func` functions. `poisson`

```
/dir$ make
/dir$ OMP_NUM_THREADS=2 mpirun -np 3 poisson
 numpts   numprs   numtrd   maxerr         time
 4        3        2        4.870972e-03   0.012758
 8        3        2        1.189740e-03   0.005394
 ⋮
```

Figure 1: Example of how to run `poisson`, and what will be returned.

will abort if number of MPI-threads are larger than the problem size. Figure **??** shows how `poisson` is used in the `terminal`, and what it returns.
For an explanation of output-data, see table **??**. It is important to note that `poisson` is written

| | |
|---|---|
| `numpts` | Number of interior points pluss one. |
| `numprs` | Number of MPI-threads used. |
| `numtrd` | Number of OpenMP-threads used. |
| `maxerr` | Largest difference between numerical solution, and analytical solution. |
| `time` | Total computation time in second. |

Table 2: Explanation of output data from `poisson`.

to run with several MPI and OpenMP threads. A program written to use only MPI, OpenMP, or neither would outperform `poisson` when using the same number of threads.

# 5 Numerical results

All numerical results, both convergence and speedup analysis was found using `kongull`, with analytical solution $u = x(1-x)\sin(\pi y)$, and $f = \sin(\pi y)(2 + \pi^2 x(1-x))$. The speedup, and parallel efficiency where also calculated. Speedup is defined as

$$S_p = \frac{t_1}{t_p}$$

and parallel efficiency as

$$\eta_p = \frac{S_p}{p}$$

where $t_p$ is run time with $p$ processors, $S_p$ is speedup with $p$ processors, and $\eta_p$ is parallel efficiency for $p$ processors. Speedup measures how much faster a program runs with $p$ processors, ideal speedup is when $S_p = p$. Parallel efficiency measures how well each processor is utilized. Perfect parallel efficiency is when $\eta_p = 1$.
`poisson` has a great weakness, it gets time results from one run through. The time results should be averaged over many runs. The results enclosed may therefore have considerable interference in run time.

## 5.1 Convergence analysis

To show convergence, `poisson` was run with $p$ MPI-threads and $t$ OpenMP -threads, with $p \cdot t = 36$. The result is shown in figure **??**. As can be seen from figure **??**, the convergence is near quadratic for all combinations of $p$ and $t$. It seems as convergence nearly stops with the largest problem size, $n = 16384$. This might be due to the round-off errors. We suspect the discrete sine transform function is the major cause of this incongruity. It also seems as if the error is generally smaller with larger $p$, thus OpenMP must have larger round off error than MPI. For $p = 12, 9$, $t = 3, 4$ the error converges in a serpentine. It is interesting to observe that some of the convergence rates have steeper slopes than that of the Helpline.

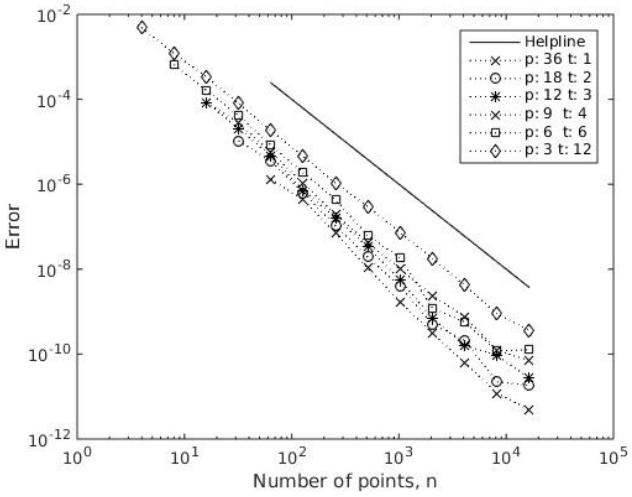


Figure 2: A loglogplot of the convergence of `poisson`. The 'Helpline' shows the ideal convergence rate. The dotted lines shows the convergence rate for a certain number of MPI and OpenMP threads.

# 6 Timing results

Figure **??** shows how the run time scales with varying problem sizes. The initial drop in run time is

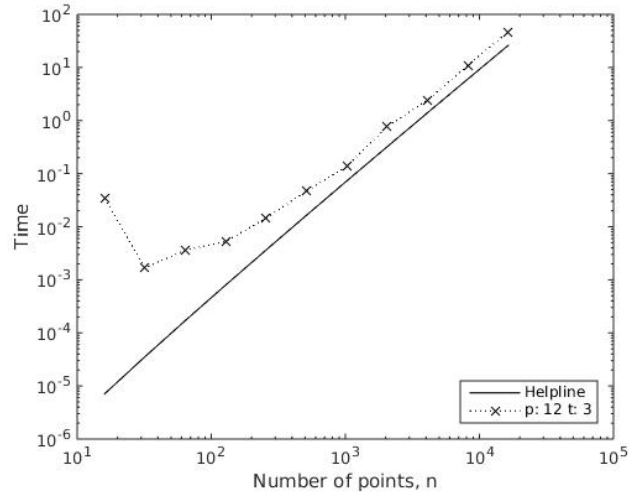


Figure 3: A loglogplot of `poisson`'s run time. The 'Helpline' shows the expected timescale, $n^2 \log_2(n)$. The dotted line is actual run time with $p = 12$, $t = 3$.

probably due to start up time and initializing. For small problem sizes the time increase is smaller than expected, but converges toward the expected timescale. The reason for the small increase in run time is due to the fact that MPI preforms better with medium problem sizes. For even lager problem sizes the run time would probably increase even faster, due to increased need to send and receive data across threads.

## 6.1 Speedup analysis: MPI and OpenMP

To compare run time, `poisson` was run with different combinations of $p$ and $t$, where $p \cdot t = 36$, and $n = 16384$. The results are shown in figure **??**. From figure **??** we see that a combination of MPI

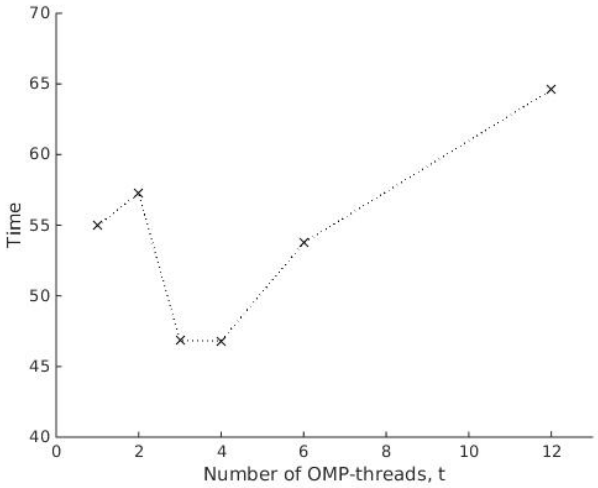


Figure 4: A plot of the time used to run `poisson` with $n = 16384$ and with different numbers of MPI and OpenMP threads so that $p \cdot t = 36$.

and OpenMP threads are the optimal for solving this problem. This may be because the program was written to use both MPI and OpenMP, and without using both there are considerable extra work to do, without any gain.

## 6.2 Speedup analysis: MPI

To measure the speedup MPI causes, `poisson` was run with $p, n$ changing and $t = 1$. Figure **??** shows run time for varying $p$ when $n = 16384$. Speedup and parallel efficiency is posted in table **??**. From table **??** it is clear that using several threads on a small problem size gives little or negative

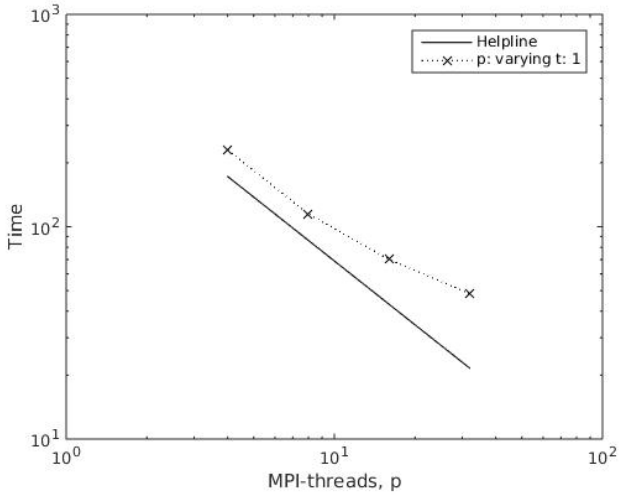


Figure 5: A plot of the time used to run `poisson` with $n = 16384$, $t = 1$, and $p$ varying.The 'Helpline' shows ideal speedup.

| $n$ | 32 | 128 | 1024 | 4096 | 16384 |
|---|---|---|---|---|---|
| $S_4$ | 0.8137 | 1.1686 | 3.9479 | 3.9684 | 3.8170 |
| $S_8$ | 0.8595 | 1.7880 | 7.4630 | 7.5402 | 7.6457 |
| $S_{16}$ | 0.1052 | 1.5999 | 11.1927 | 11.2353 | 11.7548 |
| $S_{32}$ | 0.0610 | 1.7339 | 14.5749 | 17.0748 | 18.6038 |
| $\eta_4$ | 0.2034 | 0.2921 | 0.9870 | 0.9921 | 0.9543 |
| $\eta_8$ | 0.1074 | 0.2235 | 0.9329 | 0.9425 | 0.9557 |
| $\eta_{16}$ | 0.0066 | 0.1000 | 0.6995 | 0.7022 | 0.7347 |
| $\eta_{32}$ | 0.0019 | 0.0542 | 0.4555 | 0.5336 | 0.5814 |

Table 3: Speedup $S_p$ and parallel efficiency $\eta_p$ for different $n, p$ with $t = 1$.

gain. With larger problem sizes the speedup seems to be near perfect for few processors, and a significant increase for several processors. There also seems to be a negative speedup for $p = 4$ when $n$ goes from 4096 to 16384, but the numbers are to close to say anything specific, other than random interference. Figure **??** also shows the increasing speedup, and decreasing parallel efficiency.

## 6.3 Speedup analysis: OpenMP

A speedup analysis was also preformed for OpenMP. `poisson` was now run with $t, n$ changing and $p = 1$. Figure **??** shows run time for varying $t$ when $n = 16384$. Speedup and parallel efficiency is posted in table **??**. From table **??** it is clear that OpenMP gives a considerable speedup for both

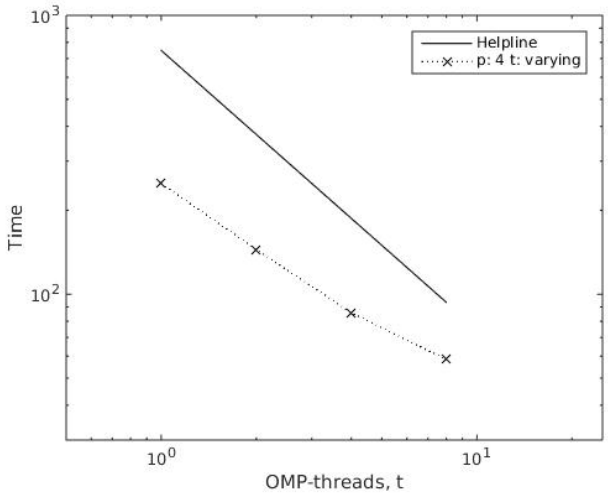


Figure 6: A plot of the time used to run `poisson` with $n = 16384$, $t$ varying, and $p = 1$.The 'Helpline' shows ideal speedup.

| $n$ | 32 | 128 | 1024 | 4096 | 16384 |
|---|---|---|---|---|---|
| $S_4$ | 2.9523 | 3.5120 | 3.4007 | 3.8034 | 3.7721 |
| $S_8$ | 1.3490 | 2.1090 | 5.8543 | 6.9255 | 6.4845 |
| $\eta_4$ | 0.7381 | 0.8780 | 0.8502 | 0.9508 | 0.9430 |
| $\eta_8$ | 0.1686 | 0.2636 | 0.7318 | 0.8657 | 0.8106 |

Table 4: Speedup $S_t$ and parallel efficiency $\eta_t$ for different $n, t$ with $p = 1$.

small and large problem sizes, but also seems to give a performance decrease when $n$ goes from 4096

to 16384. Again, this may be due to random interference, but the decrease is considerably larger than in the MPI-case.

### 6.4 Comparing MPI and OpenMP

When comparing speedup and parallel efficiency it is clear that MPI outperforms OpenMP for large problem sizes, and OpenMp outperforms MPI for small problem sizes. This matches the result from section **??**, and it seems to be best to use a small number of OpenMP-threads, and a large number of MPI-threads. When comparing figure **??** and figure **??** it may seem as OpenMP has a speedup closer to the ideal speedup, but keep in mind that the largest number of threads in figure **??** is 32, while the largest number of threads in figure **??** is 8. Due to random interference, the results may vary, and does not give a clear answer, just indications.

## 7 Changing the problem

Solving Poisson's equation on the unit square with zero boundary values is all well and good, but what happens if we want to change any of this. Let's first see what needs to change if the domain was any rectangle, that is equation (**??**) becomes

$$-\nabla^2 u = f \qquad \text{in } \Omega = (0, L_x) \times (0, L_y) \tag{11}$$

Since we can change our reference system, any rectangular domain can be the domain in equation (**??**). The necessary changes to `poisson` would be to create new step sizes $h_x = \frac{L_x}{n}$, and $h_y = \frac{L_y}{n}$, and change $h$ to $h_x$ and $h_y$ in the correct places. We assume that the number of interior grid points $(n-1)$ are kept equal in both directions.
Changing the boundary values from equation (**??**) to

$$u = g(x, y) \qquad \text{on } \partial\Omega \tag{12}$$

is a little more tricky. First we need to change $n-1$ internal points to $n+1$ nodes. The additional points will contain the boundary values. If $\underline{B_{old}}$ is the matrix solving (**??**) with conditions (**??**) then $\underline{B_{new}}$ as defined in (**??**) is the matrix solving (**??**) with conditions (**??**).

$$\underline{B_{new}} = \begin{bmatrix} g(x_0, y_0) & \dots & g(x_n, y_0) \\ \vdots & \underline{B_{old}} & \vdots \\ g(x_0, y_n) & \dots & g(x_n, y_n) \end{bmatrix} \tag{13}$$

The two changes discussed here can easily be implemented together.

## 8 Conclusion

We have investigated the effectiveness of MPI and OpenMP on different problem sizes, with different combinations of the two. Concepts like speedup, relative efficiency and run time were measured. The convergence was good. We found that OpenMP worked well when the number of points was small, while MPI performed better when the number of points were large. When combining OpenMP with MPI we found that using a large number of MPI threads with a small number of OpenMP threads was best. `poisson` also scaled well with time vs number of points. Possible bottlenecks occur when transposing the matrix $\underline{\mathrm{B}}$ due to the large exchange of data between threads.

# References

Kongull hardware
`https://www.hpc.ntnu.no/display/hpc/Kongull+Hardware`

Einar M. Rønquist, *Introduction to supercomputing*, Problem set 4, 2015

Arne Morten Kvarving, *Introduction to supercomuting*, Solving the linear system resulting from the Poisson problem, lecture slides, 2015

Einar M. Rønquist, *Introduction to supercomputing*, Introduction, 2011

Einar M. Rønquist, *Introduction to supercomputing*, Computer Architecture: Single Processor Systems, 2009

Arne Morten Kvarving, *Using the Kongull cluster*, 2012