Exercise 6
TMA4280 Introduction to Supercomputing

Knut Andreas Skaar Hasund and Emily Siggerud
April 16, 2013

Introduction

In this exercice we have implemented a numerical method for solving the two-dimensional Poisson problem. The problem can be formulated as follows

$$\begin{aligned} -\nabla^2 u &= f \quad \text{in } \Omega = (0,1) \times (0,1) \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned} \tag{1}$$

A regular five-point stencil is used in the discretization. With $(n+1)$ points in each spatial direction and a mesh spacing given as $h = 1/n$, the discrete Laplace operator becomes

$$\nabla^2 u = \frac{u_{i,j+1} + u_{i,j-1} + u_{i+1,j} + u_{i-1,j} - 4u_{i,j}}{h^2} + \mathcal{O}(h^2)$$

Introducing the matrix T:

$$T = \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & & \ddots & & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}$$

the discretized version of (1) can be written:

$$(TU + UT) = h^2 f \tag{2}$$

The studied method applies a diagonalization technique including Discrete Sine Transform (DST) in order to solve system (2). This strategy restricts the problem size n to be defined as a power with base 2. The scheme for performing the entire diagonalization method and the DST function and its inverse were all provided, while the task left to the students was to develop a parallelized method suited for running on a supercomputer. In particular, the major challenge here was to develop a function to transpose matrices, an operation which requires the different threads of the program to exchange data.

Implementation

Given the fact that our program is supposed to run on only one computer cluster (NTNUs Kongull), it is beneficial to exploit the hardware setup of that particular system. The relevant parts of the setup of Kongull is as follows:

- 93 computational nodes
- 1 Gb/s ethernet communicating between nodes
- Each node (Optimist setup):
  - 2 x 6-core 2.4 GHz AMD Opteron 2431 processors
  - 24 GiB RAM
  - 800 MHz bus frequency
  - 149 GiB 15000 RPM SAS system disk
- Each processor:
  - 6 x 512 KiB L1 cache
  - 6 MiB L3-cache (shared between cores)

Each computational node has shared memory between its total of 12 cores, and the different nodes forms a distributed memory system. Given the fact that internal communication in a node (bus) is much faster than the communication between nodes (ethernet), it would be beneficial to capitalize on this. Since each call to the provided `fst`-function is independent of the rest of the system, it allows us to explore some interesting combinations of parallelization tools.

In our implementation we have used two different parallelization toolboxes. The `C` library `MPI` is used to split the system in several smaller parts between nodes, while the language extension `openMP` handles the internal parallelization on each node.

The partitioning of the right hand side of the sytem is done column wise, distributing a subset of neighbouring columns to each available `MPI` thread. Ideally the processors would share the columns equally in order to obtain perfect load balance, though this is seldom possible. Hence, in cases where the number of processors, p, is not a multiple of the matrix dimension, the width of some processors submatrices will be extended by one column each. In practise each submatrix is generated in its respective thread, and such preventing the need to ever generate the entire right hand matrix.

The required communication during a transpose is handled using the `MPI` method `MPI_Alltoallv()`, which allows an efficient way of sending and recieving vector data in between `MPI` threads. We have tried to reduce the amount of

memory use by taking advantage of the allready available memory of the submatrices as send and recieve buffers. In addition we have in one instance used the `C` function `memcpy` to access chunks of memory instead of looking up each separate element when rearranging matrix elements.

In order to take advantage of the multiple threads available on each processor, parallelization within the shared memory units is implemented using `openMP` on all for-loops. Here the static scheduling-option was choosen, as the partial problems to be solved are very close equally sized.

The program is compiled using:

- Intel `C` compiler: icc (ICC) version 11.1 20091012
- Intel `Fortran` compiler: ifort (IFORT) version 11.1 20091012
- CMake version 2.8.7
  - `CMAKE_BUILD_TYPE` is set to Release

The other libraries included are listed in the appendix.

Convergence

To verify the correctness of method, a convergence test was performed for the problem with known analytic solution

$$\begin{aligned} u(x,y) &= \sin(\pi x) \cdot \sin(2\pi y) \\ f(x,y) &= -\nabla^2 u = 5\pi^2 \cdot \sin(\pi x) \cdot \sin(2\pi y) \end{aligned} \tag{3}$$

Solving problems of size $n = 2^k$, where $k \in \{7, 8, \ldots, 14\}$, the largest pointwise deviations towards the analytical solution was collected. The deviations decreased with order $\mathcal{O}(h^2)$. This is consistent with the five-point disctretization applied, and hence, it can be concluded that the method is correct. Figure (1) illustrates the convergence rate.

Timing Results

Numerous test cases with different combinatitons of problem size ($n \leq 16384$), number of processors, p, and the number of threads, t, revealed varying timing results. The variation seemed to be clustered around two distinctive values. The small variations around each value can, among other things, be explained by varying amount of network traffic by other processes running simultaneously. The gap

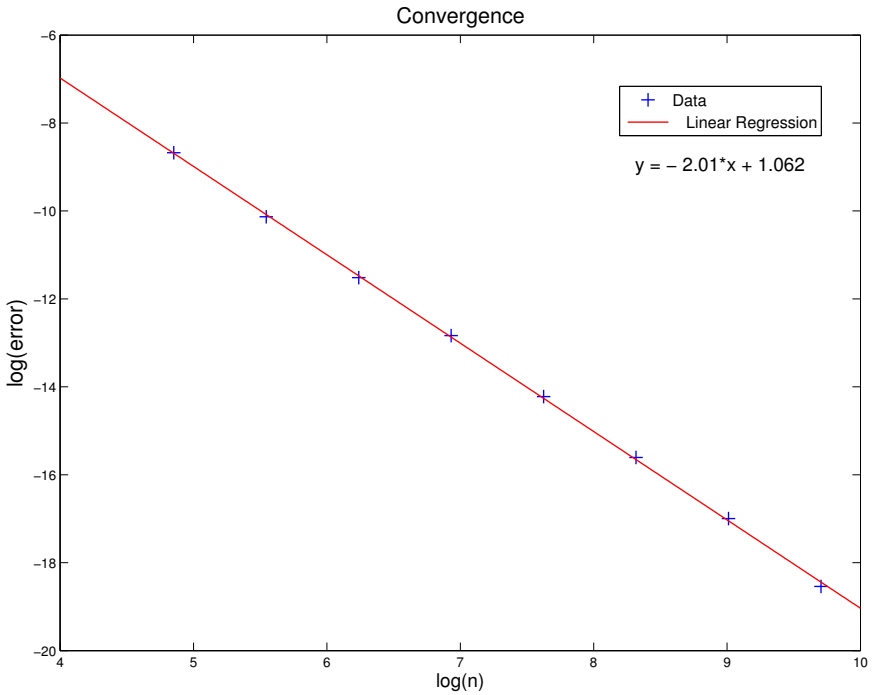

Figure 1: Logarithmic plot of error as function of problem size

between the two clusters were too large to be blamed on communication alone. In the the early stages of the project, we were not aware that parts of the Kongull cluster had been updated with new Intel processors. This should not have any impact on our results, since the upgraded parts of the cluster is off limits for the plebs. In reality it seems that jobs with a low walltime are allowed calculation time when the Intel processors are ideling. The new processors are faster and has a significantly larger cache, which, when randomly allowing passage, had a remarkable impact on our timing results. The execution time when solving our largest discretization on only one thread was observed as low as 439 seconds. By specifying use of the old AMD nodes only in our job scripts, the corresponding result was 798 seconds, and the results were in general more consistent . Normally one would appreciate the utilization of the faster processors, as they allow for faster calculations and a lower power usage per clock cycle. These are all favorable factors in every case where you are not obsessed with consistent timing results, thus the majority of our work is based on calculations on the AMD processors only. A few interesting details about the faster performances are included as well.

Time profile of method

A detailed time profile was investigated for the largest discretization tested ($n = 16384$) to gain insight in the time consumption of different parts of the algorithm with $p = 3$, $t = 12$; $p = t = 6$; $p = 12$, $t = 3$; and $p = 36$, $t = 1$. This was compared to a profile for the corresponding problem solved without any parallelization. The times are listed in the tables below in the order they are executed.

task	$p = 1, t = 1$	$p = 3, t = 12$	$p = 6, t = 6$	$p = 12, t = 3$	$p = 36, t = 1$
Initialization	1.377e+01	1.569e+00	6.408e-01	8.881e-01	4.878e-01
1st fst call	1.870e+02	5.398e+00	5.209e+00	5.201e+00	5.200e+00
1st transpose	1.380e+01	6.257e+00	5.822e+00	5.834e+00	5.401e+00
1st fstinv call	1.885e+02	5.255e+00	5.451e+00	5.247e+00	5.247e+00
Eigenvalue div.	2.076e+00	1.401e-01	8.660e-02	7.994e-02	5.882e-02
2nd fst call	1.868e+02	5.201e+00	5.209e+00	5.200e+00	5.200e+00
2nd transpose	1.376e+01	6.164e+00	5.580e+00	6.109e+00	5.220e+00
2nd fstinv call	1.885e+02	5.246e+00	5.257e+00	5.243e+00	5.254e+00
Error calculation	3.742e+00	1.175e-01	3.825e-01	1.583e-01	1.816e-01
Total Walltime	7.980e+02	3.535e+01	3.364e+01	3.396e+01	3.225e+01

Table 1: Wall time profiles (AMD only)

In table (1), we have only used AMD processors. One can see that its beneficial to parallelize the algorithm, as the runtime goes down from a total of 798 seconds on one thread to around 32 seconds on 36 threads.

task	$p = 12,\ t = 3$	$p = t = 6$	$p = 3,\ t = 12$	$p = t = 1$
Initialization	1.253e+00	2.212e+00	3.900e+00	1.141e+01
Set eigenvalues	1.389e-04	1.411e-04	1.800e-04	1.862e-04
1st fst	2.854e+00	2.853e+00	3.042e+00	1.025e+02
1st transpose	9.237e+00	9.993e+00	1.144e+01	1.200e+02
1st fst inverse	2.842e+00	2.840e+00	3.016e+00	1.021e+02
divide by eigenvalues	5.541e-02	5.538e-02	5.549e-02	1.985e+00
2nd fst	2.855e+00	2.853e+00	2.856e+00	1.025e+02
2nd transpose	5.287e+00	5.006e+00	4.609e+00	6.101e+00
2nd fst inverse	2.841e+00	2.840e+00	3.002e-01	1.021e+02
Total time	2.320e+01	2.368e+01	2.589e+01	4.390e+02

Table 2: Wall time profiles (IntelXenon)

By removing the AMD restriction on our job scripts, we got quite different and faster results, as can be seen in table (2). Here, transposing is the most time

consuming part of the parallel method due to communication. Even though the transposed matrix has constant dimensions, the 2nd transpose is performed in half the time. This might be because of a high number of successfull cache hits, due to the much larger cache available on the Intel processors. This facilitates the reuse of element location stored in the cache after the 1st transpose.

Hybrid vs. pure distributed memory

Table 2 and 1 shows how different combinations of shared and distributed memory affects the run time when $n = 16384$. Using AMD only, all the parallized versions are more than twice as fast as the sequential, but with litte variation among each other. Still, it seems slightly faster to use pure distributed memory only for this problem size.

Utilizing the IntelXenon processors, the versions with 6 and 12 processors combined with 6 and 3 threads per processor respectively were the fastest options. Only the case $p = 36$, having purely distributed memory, is distinctively slower than the others. A reason for this might be that the increasing amount of communication between the 36 processors is slower than reading and writing to larger structures of shared memory. The case $p = 3$, $t = 12$ is slightly slower than the fastest. With a larger shared memory required for this case, it might be necessary to involve more of the slower accessible memory.

For the IntelXenon-results, it can be concluded that a pure distributed memory version is slower than a hybrid in terms of speed, as the hybrid balances the time spent in communication and accessing shared memory best. The AMD-results are obtained with a smaller cache available, which makes a purely distributed memory stucture the better option. For other problem sizes other results might occur, as the partitioning of the problem may fit the computer architecture differently.

Speed up

The speedup of the parallel algorithm, $S_p = \frac{T_1}{T_p}$, was investigated for increasing number of processors in two cases; $n = 16384$ and $n = 512$. With only one processor available, the time consume for each case is $T_1 = 798s$ and $T_1 = 0.493s$ respectively. Figure (2) shows the fastest versions (pure distributed memory) at each point. The parallel efficiency, $\eta_p = \frac{S_p}{p}$, corresponding to the speedup found is shown in figure (3). As expected, S_p increases with the number of processors, but the growth rate of the speedup declines. The parallel efficiency illustrates the same tendency. The reason for this is that communication between distributed memories takes more time when the number of processors utilized increases.

Keeping $p = t = 6$ fixed, the speedup was also investigated for increasing number of grid nodes, $n \in \{64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384\}$. Figure

(4) shows that the speedup increases with system size. This is as expected, as time spent on overhead decreases compared to actual calculation time. The speedup and parallel efficiency when the IntelXenon processors were accessed is not plotted here, but we got to observe their behavior. In despite of the best performance, the speedup and parallel efficiency for these tests did not look as promising, due to a relatively fast sequential run time (439 seconds).

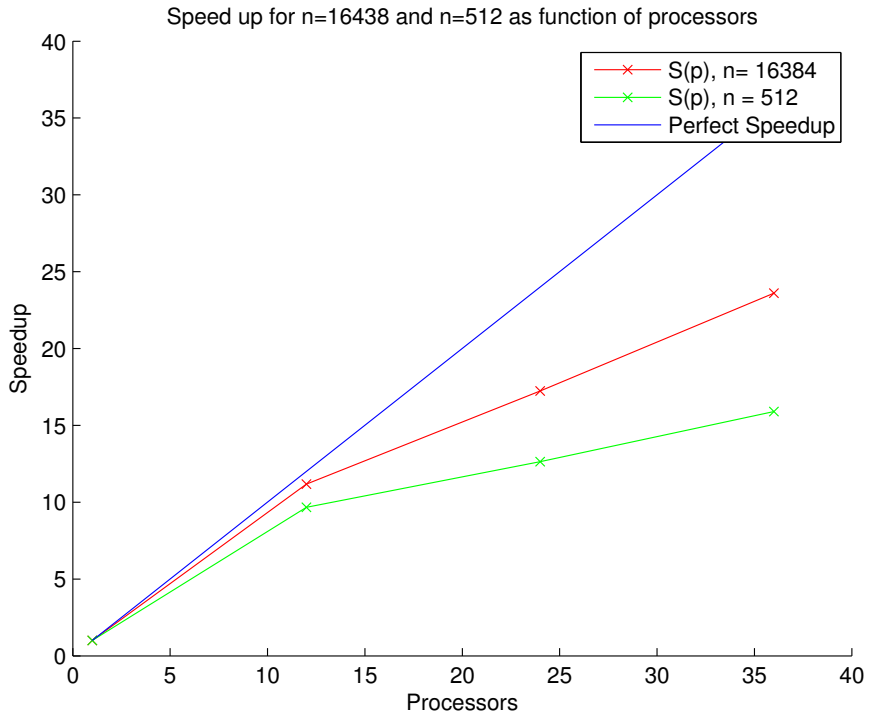

Figure 2: Speed up when the number of processors increases

Other problems

Another right hand side

The method was also tested for a slightly modified right hand side

$$f(x,y) = -\nabla^2 u = 25\pi^2 \cdot \sin(3\pi x) \cdot \sin(4\pi y) \tag{4}$$

with solution

$$u(x,y) = \sin(3\pi x) \cdot \sin(4\pi y) \tag{5}$$

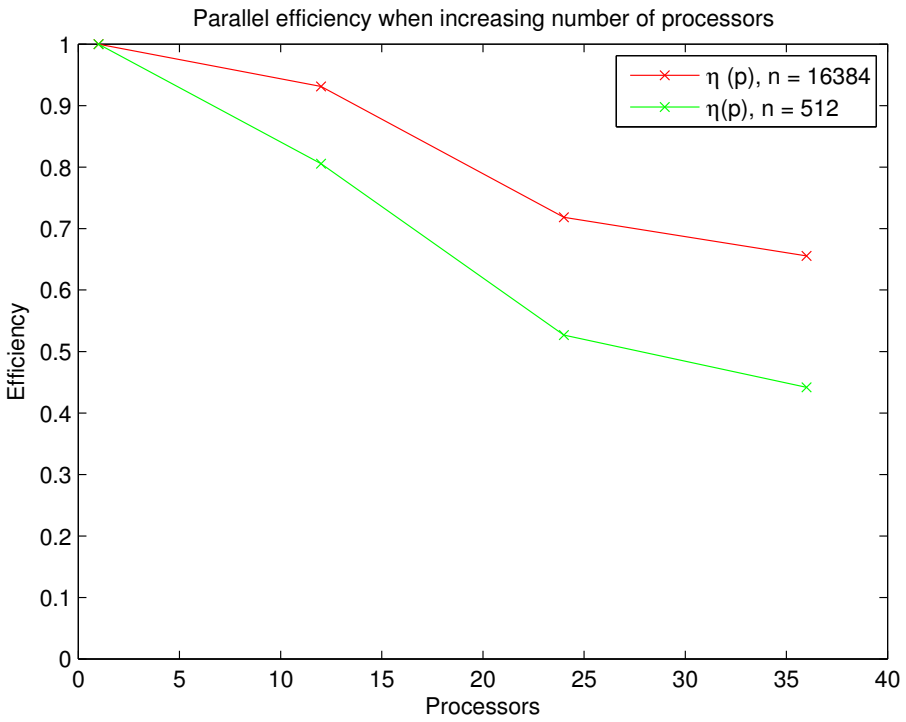

Figure 3: Parallel efficiency up when the number of processors increases

The method solved this problem with about the same performance as for (1). A plot of the solution is shown in figure (5).

Other domains

The method itself will only solve problems on the unit square with constant mesh spacing, h. Yet, problems on other domains can be solved by introducing a mapping that associates the new domain with the unit square. Going from $\Omega_{x,y} = (0, L) \times (0, L)$ to the domain $\Omega_{\tilde{x},\tilde{y}} = (0, 1) \times (0, 1)$ the mapping becomes

$$\tilde{x} = x/L \text{ and } \tilde{y} = y/L$$

Hence, the right hand side, f, must be modified to take the form

$$f(x, y) = L^2 f(\tilde{x}, \tilde{y})$$

The method will then return $u(\tilde{x}, \tilde{y})$, from which the solution to the original problem is found by mapping back.

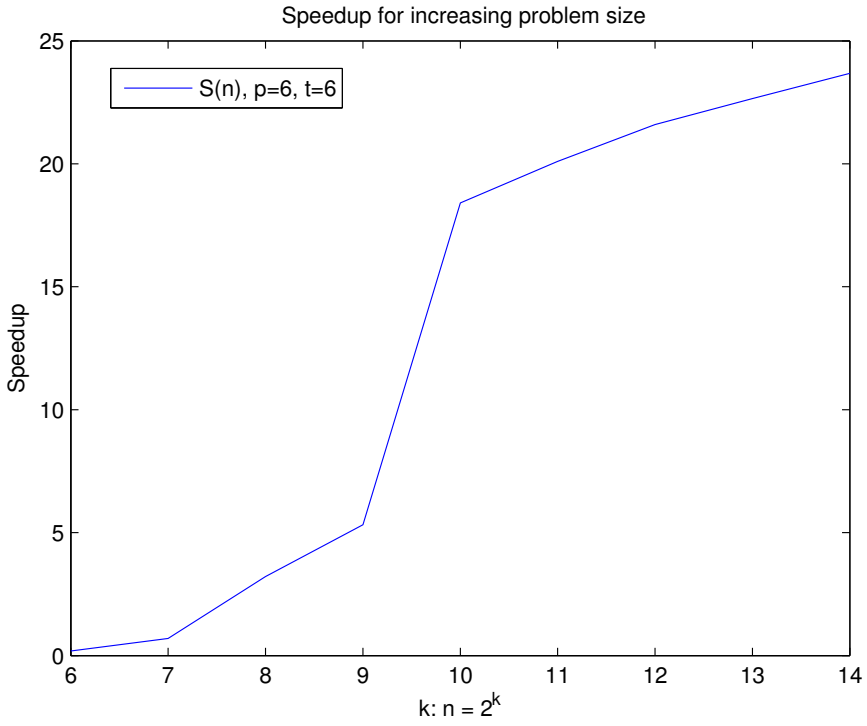

Figure 4: Speed up when the number grid points in the discretization

On domains like $\Omega = (0, L_x) \times (0, L_y)$, where $L_x \neq L_y$, more extensive modifications to the algorithm are necessary. Keeping the number of grid points equal to n in each direction, the mesh spacings are no longer equal.

$$h_x = L_x/n \text{ and } h_y = L_y/n$$

Hence, the discretized system can be written

$$\frac{TU_{i,j}}{h_x^2} + \frac{UT_{i,j}}{h_y^2} = f_{i,j}$$

Which leads to the transformed system

$$\frac{\lambda_i \overline{U}_{i,j}}{h_x^2} + \frac{\lambda_j \overline{U}_{i,j}}{h_y^2} = \overline{f}_{i,j}$$

introducing $\tilde{\lambda}_i = \lambda_i/h_x^2$ and $\tilde{\lambda}_j = \lambda_j/h_y^2$ we get

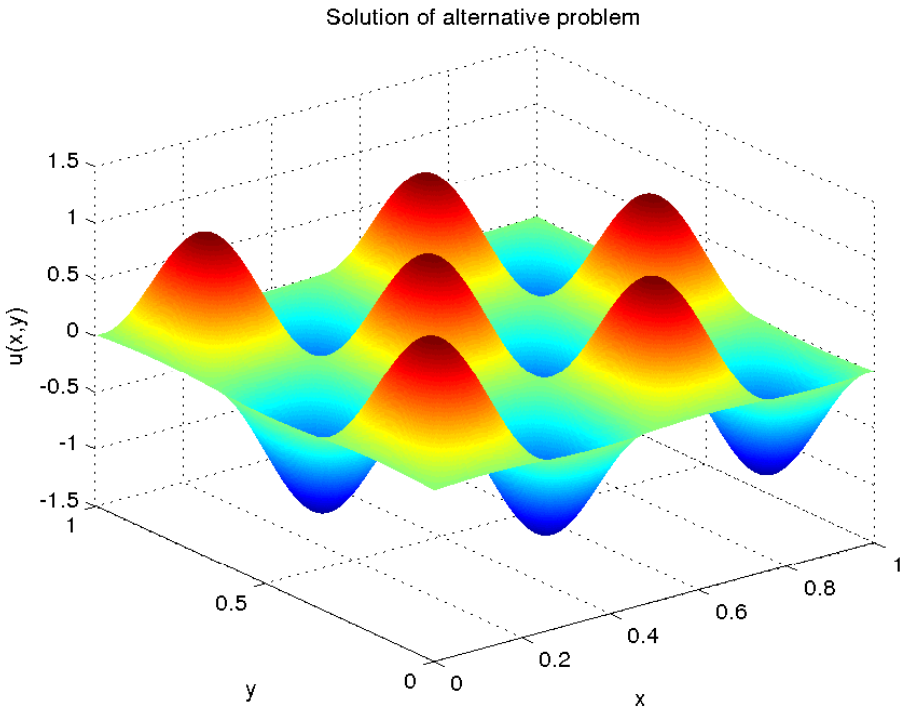

Figure 5: Solution of (4). One can see that the plot is consistent with the exact solution given by (5).

$$\overline{U}_{i,j} = \frac{\overline{f}_{i,j}}{\tilde{\lambda}_i + \tilde{\lambda}_j} \tag{6}$$

From equation (6) one can see that the right hand side must be initialized without multiplying by h^2. i.e. $b_{ij} = f_{ij}$ in stead of $b_{ij} = h^2 f_{ij}$. h_x^2 and h_y^2 have to be included when dividing by the eigenvalues in the step after the call to the first fst inverse-function.

Non-homogeneous boundary conditions

In the case of problems with non-homogeneous Dirichlet boundary conditions, it is necessary to introduce a lifting function. Set $u = v - v_0$ and let v have homogeneous boundary conditions so that

$$\nabla^2 v = f + \nabla^2 v_0.$$

Now, the implemented algorithm can be used to compute v, and u can be found through the relation $u = v - v_0$.

Conclusions and reflections

In this project we have experienced how the advantages of parallel algorithms can differ for different problem sizes, implementation approaches and computer architectures. In terms of speeding up the execution of the algorithm, two major bottlenecs have been found; network communication and cache capacity. In particular, we have found that the Kongull updates evokes the largest improvement. Running on the older nodes of Kongull only, no time was gained when introducing the hybrid version of shared and distributed memory. However, such a hybrid prooved to be beneficial when accessing the newer nodes of Kongull. Thus, this might be a piece of mind for the future.

From our experiences, we do believe that calculations on computer specific cache- and problem sizes may be a key to success when aiming to optimize an algorithm. On the other hand, as system size increases, it will at some point be too large anyways, and cache optimization will not matter that much. It might be hard to reduce network traffic during the transpose, since this information has to be exchanged one way or another. That being said, if one could do the communication in place, the total memory requirement could be reduced almost by half. Since `MPI_Alltoallv` requires both a send buffer and a recieve buffer, one would have to find or develop some other method to do this.

As a final note regarding parallelization alternatives, the most time consuming part of this project has been to implement the distributed memory transpose function. Even though a pure `MPI` is slightly faster, the convinience of implementing parallelization through `openMP` would probably cause the largest "total speedup" when working on this exercise.

Appendix

Libraries included in the code
stddef.h
stdlib.h
stdio.h
memory.h
math.h
mpi.h
omp.h