Delft University of Technology

Introduction to High Performance Computing
WI4049TU

Lab Report

Author:
Elias Wachmann (6300421) [e.wachmann@student.tugraz.at]

January 26, 2025

General Remarks

This final Lab report includes the answers for the exercises (base grad denoted in paranthesis):

0. Introductory exercise (0.5)
1. Poisson solver (1.75)
2. Finite elements simulation (1.0)
3. Eigenvalue solution by Power Method on GPU (1.75)

The optional **shining points** (e.g., performance analysis, optimization, discussion, and clarifying figures) which yield further points are usually marked by a small colored heading in the text or an additional note is added under a figure or table. For example:
This is a shining point.

N.B. The whole repository is available on GitHub. The report would have been far to long if all the scripts used for analysis would have been listed in the appendix. Yet, I've listed all important files, on which MPI / CUDA work has been done and some extra exemplary scripts.

Furthermore, if not stated otherwise the code is run on the **Delft Blue** supercomputer. Due to exessive load on the system some tests (additionally denoted) and especially the development of the code was done on the local machine.

In an attempt to keep the file size of this report relatively low the plots created with matplitlib were saved as pngs.

0 Introductory exercise

In the introductory lab session, we are taking a look at some basic features of MPI.
We start out very simple with a hello world program on two nodes.

Hello World

```c
#include "mpi.h"
#include <stdio.h>

int np, rank;

int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  printf("Node %d of %d says: Hello world!\n", rank, np);

  MPI_Finalize();
  return 0;
}
```

This program can be compiled with the following command:

```
mpicc -o helloworld1.out helloworld1.c
```

And run with:

```
srun -n 2 -c 4 --mem-per-cpu=1GB  ./helloworld1.out
```

We get the following output:

```
Node 0 of 2 says: Hello world!
Node 1 of 2 says: Hello world!
```

From now on I'll skip the compilation and only mention on how many nodes the program is run and what the output is / interpretation of the output.

0.a) Ping Pong

I used the template to check how long `MPI_Send` and `MPI_Recv` take. The code can be found in the appendix for this section.

I've modified the printing a bit to make it easier to gather the information. Then I piped the program output into a textfile for further processing in python. I ran it first on one and then on two nodes as specified in the assignment sheet. Opposed to the averaging over 5 send / receive pairs, I've done 1000 pairs. Furthmore I reran the whole programm 5 times to gather more data. All this data is shown in the following graph:

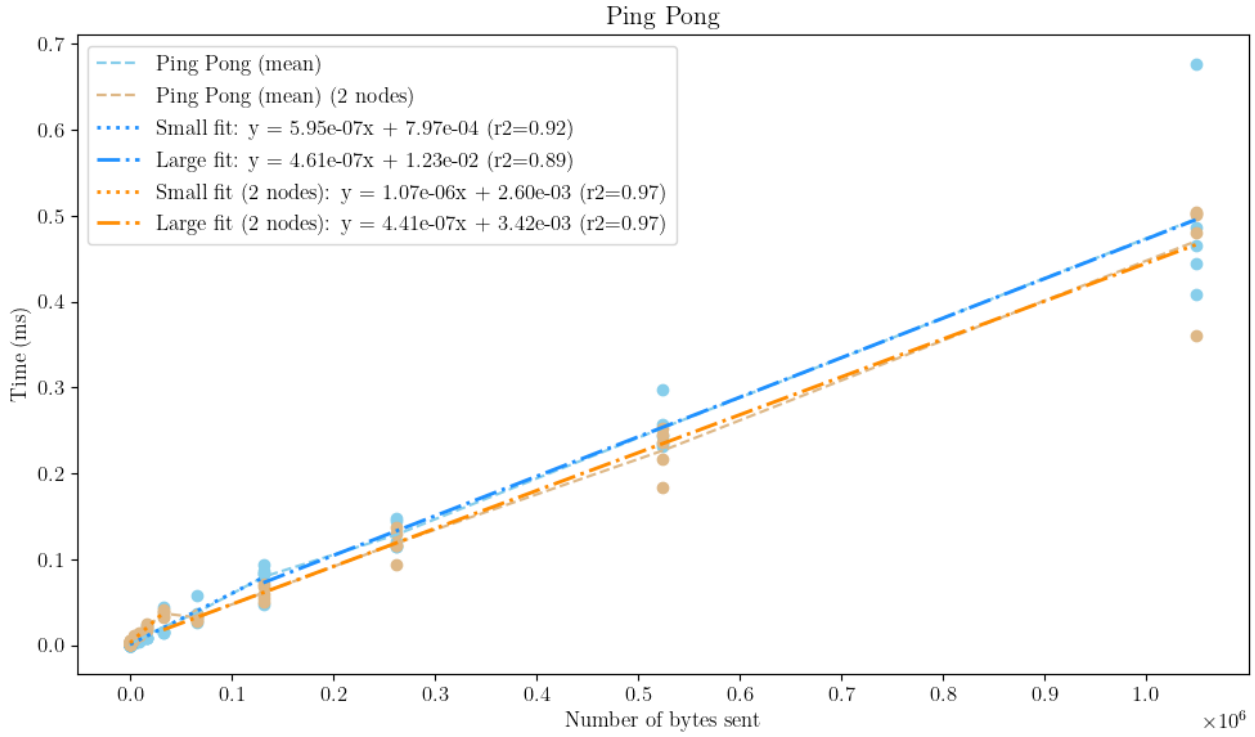


Figure 1: Ping Pong: Number of bytes sent vs. average time taken from 1000 pairs of send / receive. 5 runs shown for each size as scatter plot. Mean of these 5 runs shown as line. Blue small fit includes all data points up to 131072 bytes, blue large from there. Red small fit includes all data points up to 32768 bytes, red large from there.

As can be seen in the data and the fits, there are outliers especially for the larger data sizes.
For our runs we get the following fits and R^2 values:

Run Type	**Data Size**	**Fit Equation**	R^2 **Value**
Single Node	Small (<=131072)	$5.95 \times 10^{-7} \cdot x + 7.97 \times 10^{-4}$	0.92
Single Node	Large (>= 131072)	$4.61 \times 10^{-7} \cdot x + 1.23 \times 10^{-2}$	0.89
Two Node	Small (<=32768)	$1.07 \times 10^{-6} \cdot x + 2.60 \times 10^{-3}$	0.97
Two Node	Large (>=32768)	$4.41 \times 10^{-7} \cdot x + 3.42 \times 10^{-3}$	0.97

Table 1: Fit Equations and R^2 Values for Single Node and Two Node Runs

Note: Each run was performed 5 times (for 1 and 2 nodes) to get a fit on the data and calculate a R^2 value.

Extra: Ping Pong with MPI_SendRecv

We do the same analysis for the changed program utilizing `MPI_SendRecv`. The code can be found in the appendix for this section.
We get the following graph from the measurements which were performed in the same way as for the previous program:

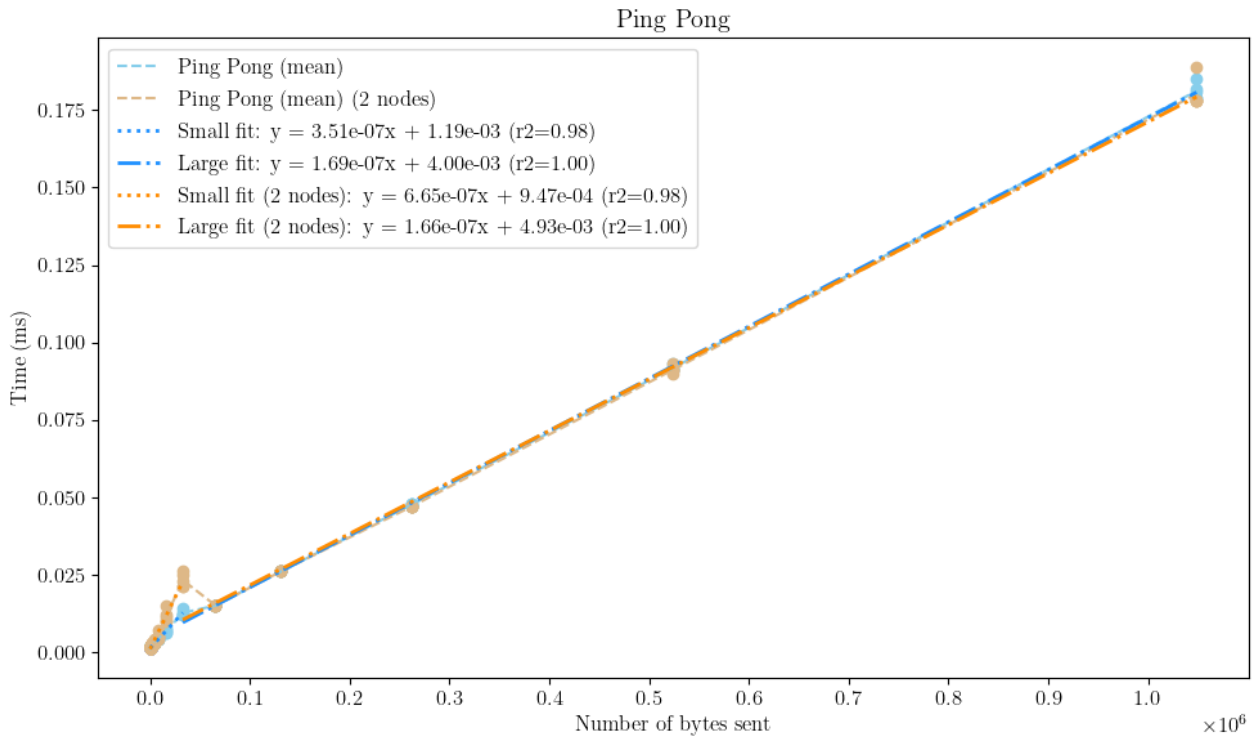


Figure 2: Ping Pong with MPI_SendRecv: Number of bytes sent vs. average time taken from 1000 pairs of send / receive. 5 runs shown for each size as scatter plot. Mean of these 5 runs shown as line. Blue small fit includes all data points up to 32768 bytes, blue large from there. Red small fit includes all data points up to 32768 bytes, red large from there.

We get the following fits and R^2 values for the runs:

Run Type	**Data Size**	**Fit Equation**	R^2 **Value**
Single Node	Small (<=32768)	$3.51 \times 10^{-7} \cdot x + 1.19 \times 10^{-3}$	0.98
Single Node	Large (>=32768)	$1.69 \times 10^{-7} \cdot x + 4.00 \times 10^{-3}$	1.00
Two Node	Small (<=32768)	$6.65 \times 10^{-7} \cdot x + 9.47 \times 10^{-4}$	0.98
Two Node	Large (>=32768)	$1.66 \times 10^{-7} \cdot x + 4.93 \times 10^{-3}$	1.00

Table 2: Fit Equations and R^2 Values for Single Node and Two Node Runs

0.b) MM-product

After an introduction of the matrix-matrix multiplication code in the next section, the measured speedups are discussed in the subsequent section.

Explanation of the code

For this excercise I've used the template provided in the assignment sheet as a base to develop my parallel implementation for a matrix-matrix multiplication. The code can be found in the appendix for this section.

The porgam can be run either in sequential (default) or parallel mode (parallel as a command line argument). For the sequential version, the code is practically unchanged and just refactored into a function for timing purposes. The parallel version is more complex and works as explained bellow:
First, rank 0 computes a sequential reference solution. Then rank 0 distributes the matrices in the following way in `splitwork`:

- Matrix A is split row-wise by dividing the number of rows by the number of nodes.

- The first worker (=rank 1) gets the most rows starting from row 0: $\texttt{total\_rows} - (\texttt{nr\_workers} - 1) \cdot \mathit{floor}(\frac{\texttt{total\_rows}}{\texttt{nr\_workers}})$.
- All other workers and the master (= rank 0) get the same number of rows: $\mathit{floor}(\frac{\texttt{total\_rows}}{\texttt{nr\_workers}})$.
- The master copies the corresponding rows of matrix A and the whole transposed matrix B* into a buffer (for details on `MM_input` buffer see bellow) for each worker and sends them off using `MPI_ISend`.
- The workers receive the data using `MPI_Recv` and then compute their part of the matrix product and send only the rows of the result matrix back to the master using `MPI_Send`.
- In the meanwhile the master computes its part of the matrix product.
- Using `MPI_Waitall` the master waits for all data to be sent to the workers and only afterwards calls `MPI_Recv` to gather the results from the workers.
- Finally all results are gathered by the master in the result matrix.

Assume we have a 5x5 matrix A and 2 workers (rank 1 and rank 2) and master (rank 0). The partitioning is done row-wise as follows:

Partitioning Example

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix} \rightarrow \begin{pmatrix} \text{Worker 1} \\ \text{Worker 1} \\ \text{Worker 1} \\ \text{Master} \\ \text{Master} \end{pmatrix}$$

- **Rank 0 (Master)**: Rows 4 and 5 (last two rows)
- **Rank 1 (Worker 1)**: Rows 1 to 3 (first three rows) - Worker 1 always gets the most rows

This partitioning can be visually represented as:

$$\text{Master (rank 0):} \begin{pmatrix} a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix}$$

$$\text{Worker 1 (rank 1):} \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \end{pmatrix}$$

Each worker computes its part of the matrix product, and the master gathers the results at the end and compiles them into the final matrix.
The `MM_input` buffer is used to store the rows of matrix A and the whole matrix B for each worker. It is implemented using a simple struct:

```
typedef struct MM_input {
    size_t rows;
    double *a;
    double *b;
} MM_input;
```

***[Optimization] Note on transposed matrix B:** It is usually beneficial from a cache perspective to index arrays sequentially or in a row-major order. However, in the matrix-matrix multiplication, we access the elements of matrix B in a column-wise order. This leads to cache misses and is not optimal. To mitigate this, we can transpose matrix B and then access it in a row-wise order. This is done in the code by the master before sending the data to the workers.

Discussion of the speedups

The code was run on Delft's cluster with 1, 2, 4, 8, 16, 24, 32, 48, and 64 nodes. For the experiments the matrix size of A and B was set to 2000×2000. This means that the program has to evaluate 2000 multiplications and 1999 additions for each element of the resulting matrix C. In total this results in $\approx 2000^3 = 8 \times 10^9$ operations. The command looked similar to the following for the different node counts:

```
srun -n 48 --mem-per-cpu=4GB --time=00:02:00 ./MM.out parallel
```

For this experiment, the execution time was measured and the speedup was calculated. The results are shown in Table 3 and Figure 3.

CPU Count	Execution Time / s	Approx. Speedup
1	47.11	1.0
2	10.26	4.6
4	10.30	4.6
8	5.20	9.1
16	2.97	15.9
24	2.54	18.5
32	2.29	20.6
48	2.98	15.8
64	1.72	27.4

Table 3: Execution Time vs CPU Count

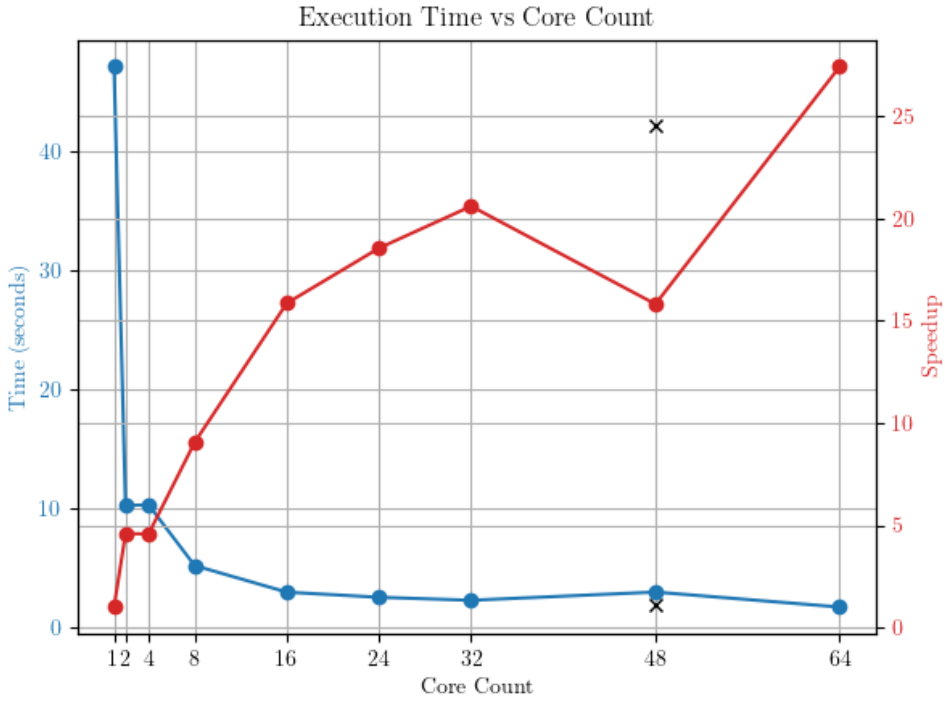


Figure 3: Speedup vs CPU Count
Black × marks the average of the rerun for $n = 48$.

Note: The speedup is calculated as $S = \frac{T_1}{T_p}$, where T_1 is the execution time on 1 node and T_p is the execution time on p nodes.

Discussion:
As one can cleary discern from the data in Table 3 and Figure 3, the speedup increases with the number of nodes (with the exception of $n = 48$). This is expected as the more nodes we have, the more work can be done in parallel. However, the speedup is not linear. This is due to the overhead of communication between the nodes. The more nodes we have, the more communication is needed, and this overhead increases. This is especially visible in the data for $n = 48$. Here the speedup is lower than for $n = 32$. For this run the communication didn't went as smooth as for the other runs. This can potentially be attributed to the fact that one (or more) of the nodes or the network was under heavy load during this task.

[Further investigation] After observing this slower speed for the $n = 48$, I reran the tests multiple times and got a runtime of around $1.9s$ which was to be expected initially. Therefore, this one run is an odd one out, most likely due to the reasons mentioned above! I've also added the averaged data of the reruns as a datapoint in Figure 3.
Another interesting fact can be seen when comparing the time taken for $n = 1$ and $n = 2$. They don't at all scale with the expected factor of 2. This is could be due to the fact, that the resource management system prefers runs with multiple nodes instead of a single node (= sequential).

Additional notes: The flag `-mem-per-cpu=<#>GB` was set depending on the number of nodes used. For 1-24 nodes 8GB was used, for 32-48 nodes 4GB, and for 64 nodes 3GB. This had to be done to comply with QOS policy on the cluster.

1 Poisson solver

In this section of the lab report, we will dicuss a prallel implementation of the Poisson solver. The Poisson solver is a numerical method used to solve the Poisson equation, which is a partial differential equation that is useful in many areas of physics.

Note: For local testing and development I'll run the code with `mpirun` instead of the `srun` command on the cluster.

1.1 Building a parallel Poisson solver

For the first part of the exercise we follow the steps lined out in the assignment sheet. I'll comment on the steps 1 through 10 and related questions bellow. The finished implementation can be found in the appendix for this section.

1. **Step:** After adding MPI_Init and MPI_Finalize, we can run the program with multiple processes. We can see that the program runs with 4 processes in Figure 4 via the quadrupeled output.

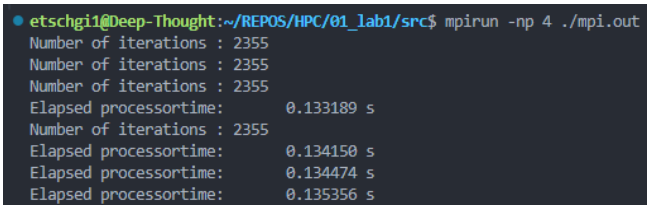


Figure 4: MPI_Poisson after Step 1 - Running with 4 processes

2. **Step:** To see which process is doing what, I included the rank of the process for the print statements as shown in Figure 5.

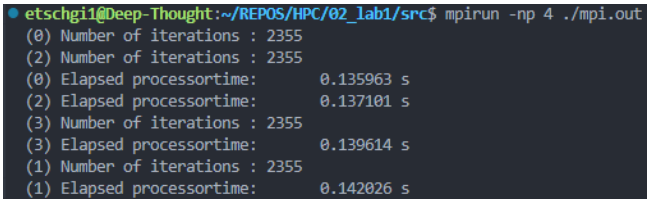


Figure 5: MPI_Poisson after Step 2 - Running with 4 processes

3. **Step:** Next we define `wtime` as a global double and replace the four utility timing functions with the ones given on Brightspace. A quick verification as shown in Figure 6 shows that the program still runs as expected.

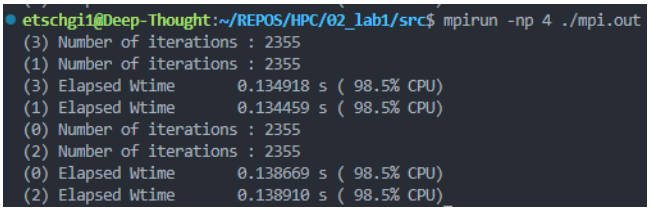


Figure 6: MPI_Poisson after Step 3 - Running with 4 processes

4. **Step:** Next we check if two processes indeed give the same output. Both need 2355 iterations to converge and the `diff` command returned no output, which means that the files content is identical.

5. **Step:** Now only the process with rank 0 will read data from files and subsequently broadcast it to the others. Testing this again with 2 processes, we see an empty diff of the output files and the same number of iterations needed to converge.

6. **Step:** We create a cartesian grid of processes using `MPI_Cart_create` and use `MPI_Cart_shift` to find the neighbors of each process. We can see that the neighbors are correctly identified in Figure 7.

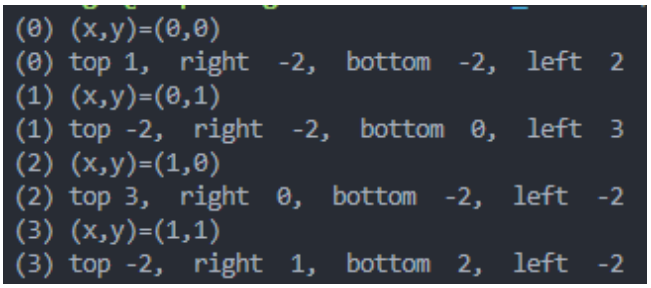


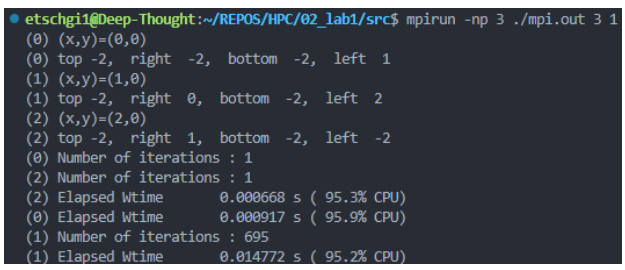
Figure 7: MPI_Poisson after Step 6 - Running with 4 processes on a 2x2 grid

When there is no neighbor in a certain direction, -2 (or `MPI_PROC_NULL`) is returned.

7. **Step:** We overhaul the setup to get a proper local grid for each process. Furthermore, we only save the relevant source fields in the local grid for each process.

With for instance 3 processes you should see that 1 or 2 processes do not do any iteration. Do you understand why?

If we have a look at the input file we see that there are only 3 source fields in the grid. This means that the process that does not have a source field in its local grid will not do any iterations (or only 1). Therefore, if we have 3 processes and the distribution of source fields as given in the input file only 1 process will do iterations if processes are ordered in x-direction and 2 if ordered in y-direction. From this we can conclude that indeed all processes have different local grids and perform different calculations.



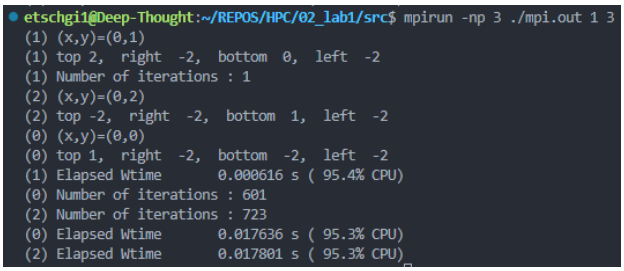


Figure 8: MPI_Poisson after Step 7 - Running with 3 processes on a 3x1 (left) vs. 1x3 (right) grid
For the 3x1 grid, only rank 1 does iterations (> 1), for the 1x3 grid, ranks 0 and 2 do iterations (> 1).

8. **Step:** After defining and commiting two special datatypes for vertical and horizontal communication, we setup the communication logic to exchange the boundary values between the processes. We call our `Exchange_Borders` function after each iteration (for both red / black grid points). Now we face the problem in which some processes may stop instantly (no source in their local grid). They will not supply any data to their neighbors, which will cause the program to hang. We shall fix this in the next step.

9. **Step:** Finally we need to implement the logic to check for convergence (in a global sense). We do this by using a `MPI_Allreduce` call with the `MPI_MAX` operation. This way we aggregate all deltas and choose the biggest one for the global delta which we use in the while-loop-condition to check for convergence. We can see that the program now runs as expected in Figure 9.

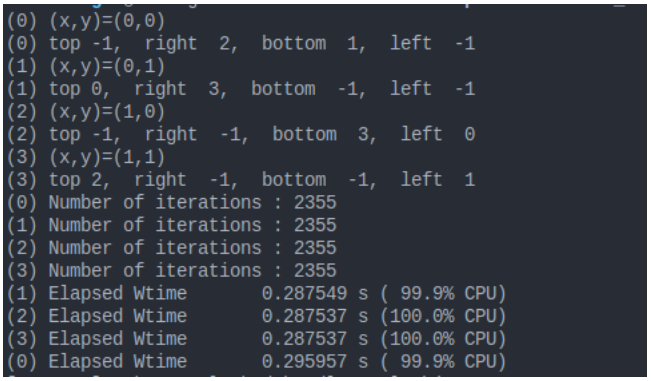


Figure 9: MPI_Poisson after Step 9 - Running with 4 processes on a 2x2 grid

Note that this run in Figure 9 was done with another pc and another MPI implementation. Therefore, we see -1 for cells without a neighbor! However, other than that cosmetic difference it has no impact on the programm.

10. **Step:** Now we only have to fix two remaining things. First we have to make sure that each process uses the right global coordinates for the output file in the end. Therefore, we change the function a bit to include the specific x-/y-offset for each processor. The second thing is the potential problem, that different processors might start with different (red/black) parities. In order to accomplish a global parity we simply have to change the calculation in the if in `Do_Step` from

```
1         if ((x + y) % 2 == parity && source[x][y] != 1)
```

to

```
1         if ((x + offset[X_DIR] + y + offset[Y_DIR]) % 2 == parity && source[x][y] != 1)
```

this guarantees that during a given iteration all processors are using the same parity.

This just leaves one question open: Are the results acutally the same?
Checking the output files of the MPI-implementation with the sequential reference indeed shows identical numerical values for the calculated points. Furthermore, the needed iterationcount is also identical which isn't a big surprise, given that the two programms perform the exact same calculation steps.

1.2 Exercises, modifications, and performance aspects

For this subsection we'll define the following shorthand notation:

Table 4: Notation for this section

n:	the number of iterations
g:	gridsize
t:	time needed in seconds
pt:	processor topology in form pxy, where:
p:	number of processors used
x:	number of processors in x-direction
y:	number of processors in y-direction

$pt = 414$ means 4 processors in a 1×4 topology.

Note on different Versions:
For the following exercises the implementation will be slightly adapted to measure different performance aspects. To facilitate this, we will use defines to switch between different versions of the code at compile time. The final version of the poissonsolver can be found in the appendix for this section.
Note on long scheduling times and work-arounds:
Delft Blue is especially bussy and wait times for jobs can be well over 30 minutes regularly. As we make use of

these resources extensively in this lab, I've created `sbatch` scripts which run mutliple configurations at once. For development of the tests and postprocessing I'll make use of my local machine. As discussed with the tutor of this lab, for some exercises a local run is sufficient to get the desiered insights (e.g. subsubsection 1.2.2 to find the optimal omega). I'll also note this in the respective subsections.

1.2.1 Over-relaxation (SOR)

We start of by rewriting the `Do_Step` routine to facilitate SOR updates. Furthermore, we need h^2, the grid spacing (which is 1 in our case) and the relaxation parameter ω to calculate the updated values. A quick local test shows a speedup of roughly a factor of 10. More systematic tests will be done in the next section.

1.2.2 Optimal ω for 4 proc. on a 4x1 grid

With the power of a little python scripting we can easily test different values for ω and plot the results as seen in Figure 10. This test was performed locally as the results are not dependent on processors because the number of iterations needed for convergence is the same for different configurations and only depends on the algorithm and the specific problem.

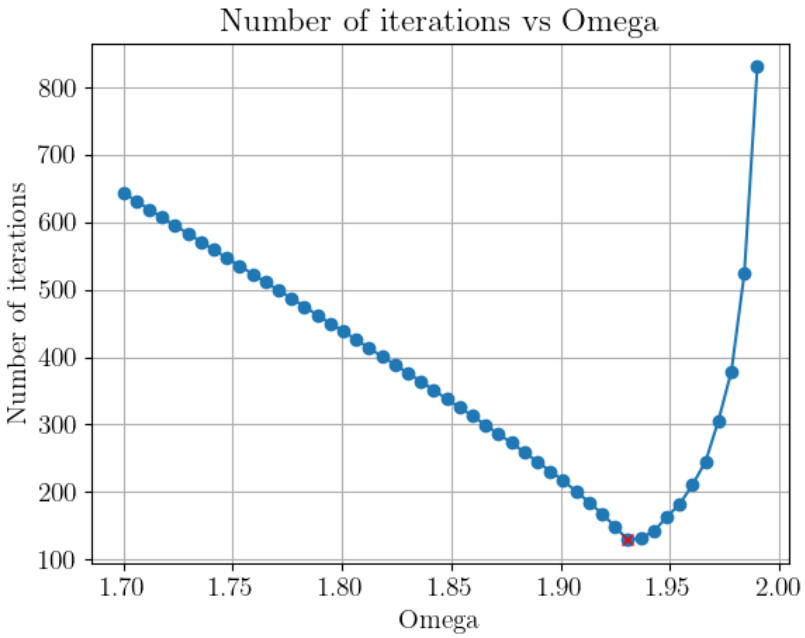


Figure 10: **Optimal ω for 4 processors on a 4x1 grid**

We find that the optimal ω is at about 1.93 for this setup with only 129 iterations. This constitutes a speedup of about 1825% compared to the sequential implementation.

N.B.: If not stated otherwise, we will use $\omega = 1.93$ for the following exercises.

1.2.3 Scaling behavior with larger grids

This investigation is carried out three times: Once with a 4×1 topology (as in the previous section), followed by a 1×4 and a 2×2 topology. We use grid sizes of 200×200, 400×400, 800×800 and 1600×1600 and set $\omega = 1.95$ for all runs. All nine simulations are ran using a `sbatch` script on Delft Blue to change the appropriate input parameters and run the program subseqeuntly (details can be found in the Appendix and online in the repository). The results are shown in Figure 11.

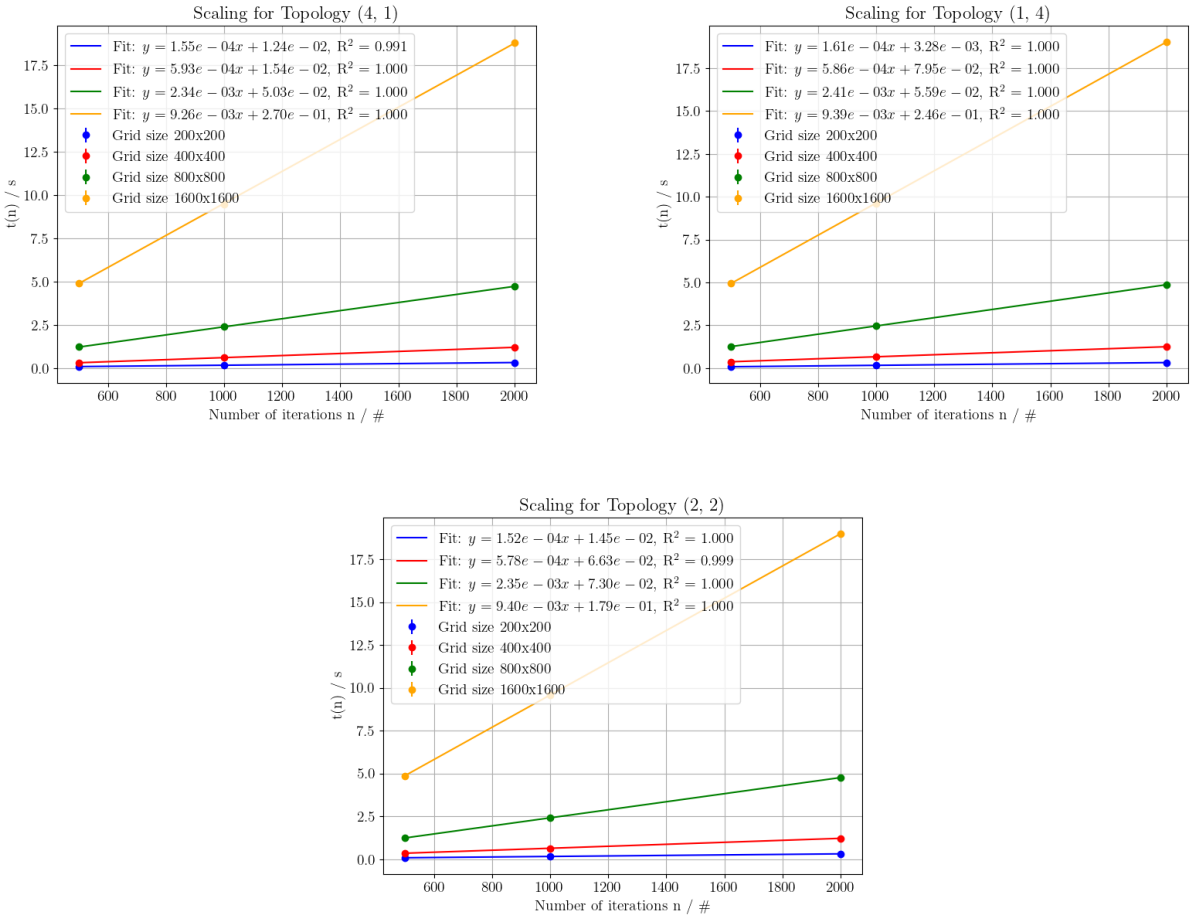


Figure 11: Scaling behavior of the Poisson solver with different grid sizes and processor topologies

As seen by the high R^2 values in the plots, the scaling behavior is very close to linear for the grids. We obtain the following scaling factors for the different grid sizes and topologies from the linear fits:

Table 5: Scaling factors for different processor topologies for the Poisson solver
Using: $t(n) = \alpha + \beta \cdot n$ as a model

Topology	Gridsize	α	β
4×1	200×200	$1.24e-02$	$1.55e-04$
4×1	400×400	$1.54e-02$	$5.93e-04$
4×1	800×800	$5.03e-02$	$2.34e-03$
4×1	1600×1600	$2.70e-01$	$9.26e-03$
1×4	200×200	$3.28e-03$	$1.61e-04$
1×4	400×400	$7.95e-02$	$5.86e-04$
1×4	800×800	$5.59e-02$	$2.41e-03$
1×4	1600×1600	$2.46e-01$	$9.39e-03$
2×2	200×200	$1.45e-02$	$1.52e-04$
2×2	400×400	$6.63e-02$	$5.78e-04$
2×2	800×800	$7.30e-02$	$2.35e-03$
2×2	1600×1600	$1.79e-01$	$9.40e-03$

What can you conclude from the scaling behavior?
We see that the scaling behavior is very close to linear for all topologies. This means that the parallel implementation scales as expected with the number of grid points.
If we compare the scaling factors (β) for the topologies we see that the 2×2 topology scales slightly better than the 4×1 and 1×4 topologies (except for the largest grid, where all three topologies scale with a very similar

factor). This is not surprising, as the 2×2 topology has a more balanced communication workload balance. In the 2×2 topology every processor has two neighbors, while in the 4×1 and 1×4 topologies the processors at the ends only have one neighbor. This is a general trend: A topology which divides the grid into square / square-like parts will scale better than a topology which divides the grid into long and thin parts.
In essence: We want to keep the communication between processors as balanced as possible to achieve the best scaling behavior.

Scaling of different grid sizes:
We see that larger grids take longer for the same amount of iterations. This is also to be expected, as the number of grid points grows quadratically with the grid size. a 800×800 grid has 4 times as many grid points as a 400×400 grid and therefore takes roughly 4 times as long to calculate.

1.2.4 Scaling behavior [Theory - no measurements]

If I could choose between a 16×1, 8×2, 4×4, 2×8, 1×16 topology, I would choose the 4×4 topology. This is because the 4×4 topology has the most balanced communication workload balance, as detailed in the **Shining** in subsubsection 1.2.3.

1.2.5 Iterations needed for convergence scaling

We investigate the number of iterations needed for convergence using the 4×1 topology square grids with sidelength: $10, 25, 50, 100, 200, 400, 800, 1600$. The results for different ω are shown in Figure 12. This test was performed locally as the results are the same for different systems and only depend on the algorithm and the specific problem.

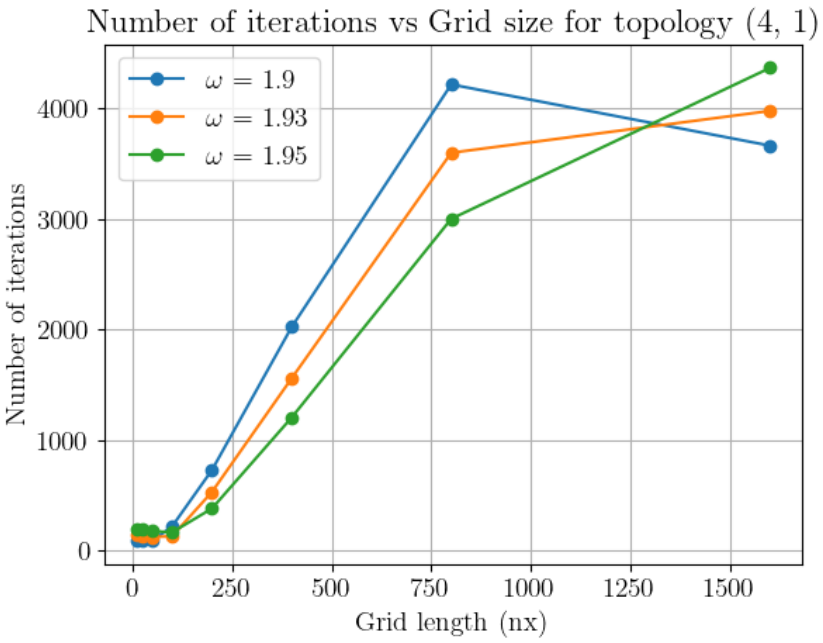


Figure 12: Iterations needed for convergence with different grid side lengths

We can clearly see that the number of iterations till convergence increases with the problem size. At first, I expected linear growth proportional to the number of gridpoints. However, it turns out that the number of iterations actually grow slower and in a square root like fashion. This can be seen by the linear behavior in the plot of grid-side length against iterations.

Why is the number of iterations needed for convergence $\propto \sqrt{g}$?
Our poisson problem is a discretized system in 2D space. The condition number of the matrix we have to solve is proportional to the number of gridpoints in our system. SOR uses the spectral properties of the matrix to solve in a way such that the dominant error mode takes time proportional to the diameter of the domain to

converge. This means it is proportional to $\sqrt{g} = \sqrt{n_x \cdot n_y}$.

Why does omega with the best performance change with the grid size?
As can be seen in Figure 12 $\omega = 1.9$ beats the other two values for very small and the largest gridsize. For different gridsizes we get differnetly sized matrices we have to solve. SOR overrelaxes high-frequency errors and underrelaxes low-frequency errors (the later for stability). The optimal ω is indeed dependent on the gridsize and the error modes present in the system. In our current example, it might be that $\omega = 1.9$ is a good compromise for the grid sizes we are looking at and we are so to say lucky with that specific choice.

1.2.6 Error as a function of the iteration number

With the same 4×1 topology and grid sizes of 800×800 the error for 15000 iterations is tracked using $\omega = 1.93$. The results are shown in Figure 13.

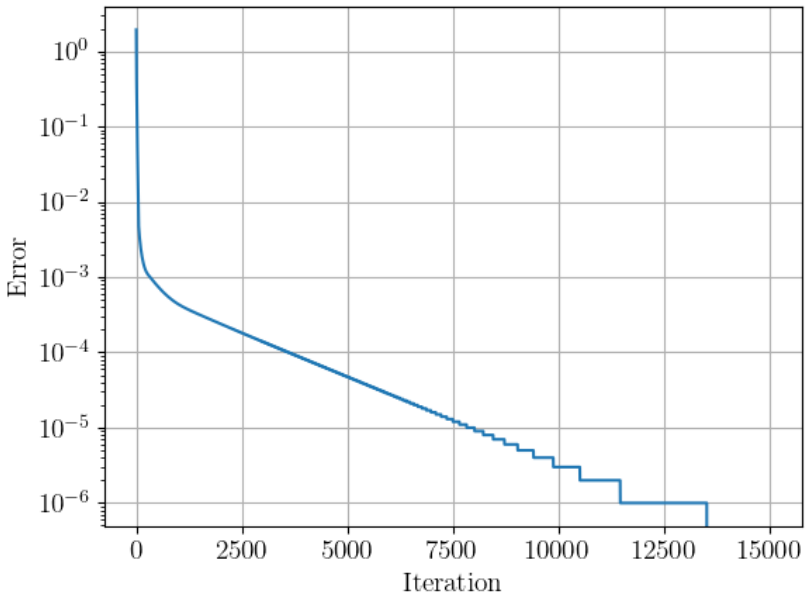


Figure 13: Error as a function of the iteration number

At first the error decreases rapidly in the first few iterations to about 10^{-3} (logarithmic scale!). After that the error decreases more slowly until it is below floating point precision.

Note: All calculations are done using double precision floating point numbers and only the error recording was done using single precision which leaves the step-like artifacts in the plot. Obviously these steps would also be present in the double precision error calculation, but they would be much smaller at comparable iteration numbers and only become visible at much larger iteration numbers.

1.2.7 Optional - Gain performance by reducing `MPI_Allreduce` calls

The last subsection showed us that the error reduces monotonically. We might be able to save some time by leaving out some checks and maybe check the global error every 10th or 100th iteration only.
First, we should benchmark if it is at all wise to optimize here, by measuring how long the `MPI_Allreduce` call takes. We can do this by measuring the time needed for the `MPI_Allreduce` call in the `Do_Step` function and summing up to get the total time spent in `MPI_Allreduce` calls.
We again solve with a 4×1 topology, $\omega = 1.93$ and a 800×800 grid: It takes roughly 20 seconds of which the processors spend around 1 - 2 seconds in the `MPI_Allreduce` call. This is a significant amount of time: $(7.0 \pm 0.4)\%$ of our runtime. This means we would save some time by reducing the number of `MPI_Allreduce` calls and calculating 9 (0.25% of total) more iterations wouldn't hurt us too bad because it takes 3601 to converge!

We run the program three times with `MPI_Allreduce` calls every 1, 10 and 100 iterations and get the speedups in `MPI_Allreduce` calls as shown in Table 6.

Table 6: Speedup in `MPI_Allreduce` calls for different iteration counts and calculated overall speedup (%)

Iterations	`MPI_Allreduce` - speedup (factor)	calculated overall speedup (%)
1 (baseline)	1.00	/
10	6.0 ± 2.0	5.9 ± 0.5
100	62 ± 6	6.9 ± 0.4

As can be clearly seen from the table we can gain around 6 % using `MPI_Allreduce` calls every 10 iterations and around 7 % using `MPI_Allreduce` calls every 100 iterations. This is a significant speedup for a very small change in the code.

Note: The speedup is calculated to account for fluctuations in the runtime of the program, due to other processes running on the same machine / cluster.

1.2.8 Reduce border communication

Another way to reduce communication overhead is to reduce the number of border exchanges. To investigate if this yields a speedup we run the program on a 4×1 topology, $\omega = 1.93$ and different grid sizes and track the iterations and time as seen in Figure 14.

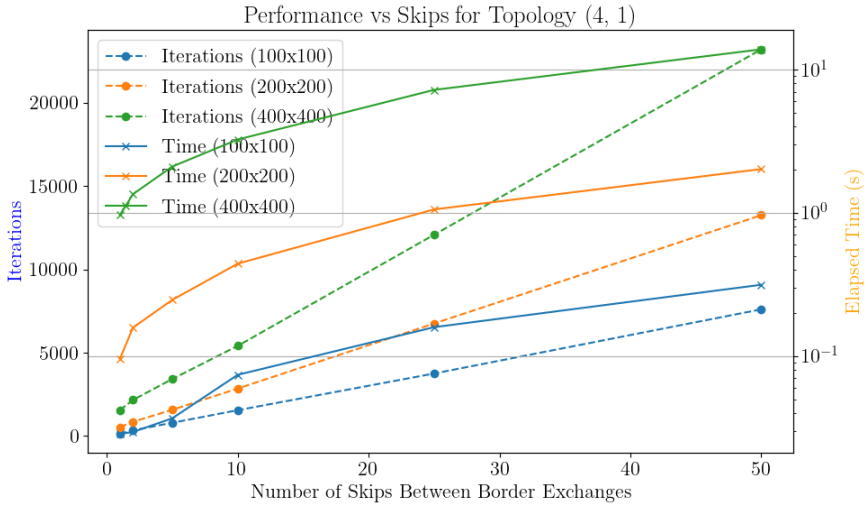


Figure 14: Speedup by reducing border exchanges (4x1 topology)

Running the with different numbers of skipped border exchanges naturally slows down convergence, meaning we need more iterations to reach the same error. For all tested grid sizes the initial SOR version without skipping border exchanges has the fewest iterations needed to convergence and also the fastes runtime.

What can you conclude from the results?
We can conclude that reducing the number of border exchanges does not yield a speedup. The reason for this is that we have to calculate more iterations to converge to the solution which outweighs the gains from reduced communication overhead. Interestingly, for the 100×100 grid there exists a local minimum in time at 4 skipped border exchanges compared to 3 skipped. This is likely due to our source field distribution and thus specific to our problem.

Running the problem again with another (2×2 topology specifically) on our Delft Blue node we get the same qualitative result as seen in Figure 15.

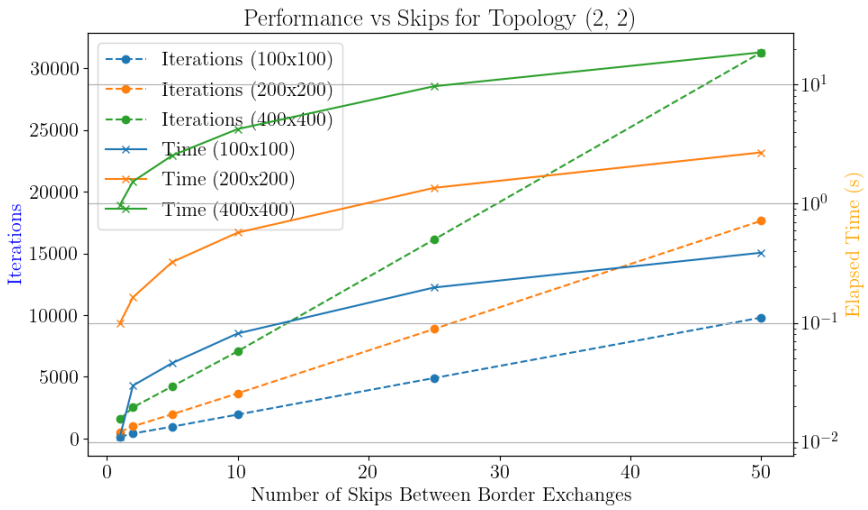


Figure 15: Speedup by reducing border exchanges (2x2 topology)

Taking a closer look at 2×2 vs. 4×1 topology:
While both have qualitatively the same behavior with the fastest time and lowest iterations recorded for the SOR version without skipping border exchanges, the 2×2 topology has a worse convergence behavior in terms of iterations. This is likely due to the fact that the borders of the local grids for the processors in the 2×2 topology are possitioned in closer proximity to the source coordinates compared to the 4×1 topology. This could explain the observed higher iteration count for convergence in all grid sizes in the 2×2 topology.

1.2.9 Optimize `Do_Step` loop

In `Do_Step` we iterate over the whole grid but only update one of the two parities at a time. This means we can split the loop into two loops, one for each parity. We start out with something like this:

```
    for (x = 1; x < dim[X_DIR] - 1; x++){
        for (y = 1; y < dim[Y_DIR] - 1; y++){
            if ((x + offset[X_DIR] + y + offset[Y_DIR]) % 2 == parity && source[x][y] != 1){
                ...
```

and we change it to:

```
    int start_y;
    for (x = 1; x < dim[X_DIR] - 1; x++){
        start_y = ((1 + x + offset[X_DIR] + offset[Y_DIR]) % 2 == parity) ? 1 : 2;
        for (y = start_y; y < dim[Y_DIR] - 1; y += 2){
            if (source[x][y] != 1){
                ...
```

The basic idea is to avoid y-coordinates which are not in the parity we are currently updating. We measure 10 runs for a 800×800 grid and a 4×1 topology with $\omega = 1.93$ and get the following times:

$$t_{\text{no\_improvements}} = (5.59 \pm 0.05)\,\text{s} \qquad \text{and} \qquad t_{\text{loop\_improvements}} = (4.64 \pm 0.07)\,\text{s}$$

So we get a minimal speedup of about 17% by optimizing the loop which is a enormous speedup for such a small change.

Why does this make such a difference
The reason for this is that we avoid unnecessary looping and if statements. This means that we have less overhead in the loop and can therefore calculate faster by skipping the unnecessary loop entries.

1.2.10 Optional - Time spent within `Exchange_Borders`

We can measure the time spent in `Exchange_Borders` by adding a timer to the function. We run the program with $\omega = 1.93$ and different topologies[1] and grid sizes and get the results shown in Figure 16.

[1] {(2,2), (3,3), (4,4), (5,5), (6,6), (2,3), (2,4), (2,5), (3,4)}

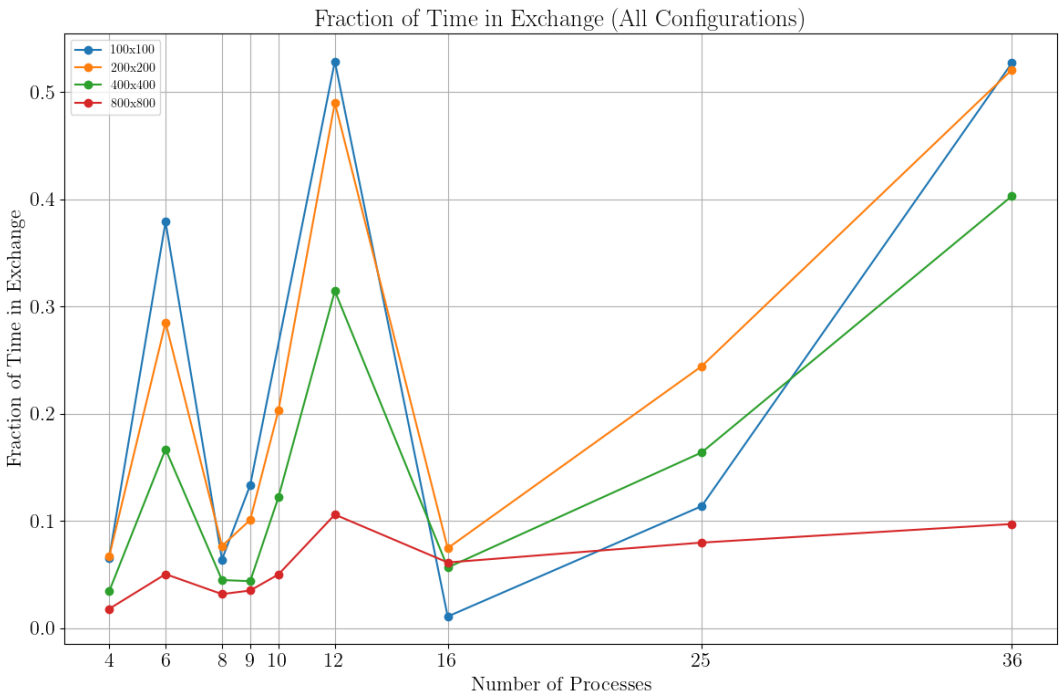


Figure 16: Fraction of total time spent in `Exchange_Borders`

As we can clearly see, the time spent in `Exchange_Borders` is initially smaller and grows with processor count with some remarkable peaks especially for small grids and certain processor topologies. The curves are generally shifted downward for larger gridsizes.

Interpretation:
One would expect larger grid sizes to be more computationally expensive and we have already established that iterations take longer the bigger the grid. Communication obviously also takes longer for a larger grid because we have more data to sent. However, the circumference of a square grows linearly, while the area grows quadratically. The quadratic growth of the area is the reason for the downward shift in the curves for larger grid sizes because the communication overhead grows slower than the computational overhead for larger grids.
Besides the peaks at $p = 6$ and $p = 12$ we see a general trend of increasing time spent in `Exchange_Borders` increasing processor count. This is not surprising as we have more processes which have to communicate with each other and the data locality is worse for larger processor counts.

When is the time spent in `Exchange_Borders` significant / comparable to computation?
As can be seen in Figure 16 the time spent in `Exchange_Borders` is significant for all grid sizes from the start (between 2.5 and 7.5%). Thereafter it peaks for $p = 6$ and again for $p = 12$ to around 5% to 38% and 10% to 54% respectively. This means that the time spent in `Exchange_Borders` is significant for all grid sizes and processor counts, but especially as the processor count grows.

1.2.11 Latency and bandwith in `Exchange_Borders`

We use the configurations from subsubsection 1.2.3: 4×1, 2×2 and 3×3 topologies with grid sizes of 200×200, 400×400 and 800×800 and $\omega = 1.95$ as well as the other settings set to their defaults. We obtain the results in Table 7.

Table 7: Metrics for `Exchange_Borders` latency and bandwith

Topology	Grid Size	Latency (ms)	Latency (%)	Bandwidth ($B\,s^{-1}$)	Total Data (B)
4x1	200x200	2.0 ± 0.6	3.9	2 034 737 840	3 104 896
4x1	400x400	9.8 ± 0.4	1.4	2 005 275 269	19 450 368
4x1	800x800	51 ± 23	0.8	2 112 229 431	96 480 384
2x2	200x200	3.6 ± 1.0	5.0	541 474 183	2 493 696
2x2	400x400	17 ± 7	2.3	668 287 123	15 591 168
2x2	800x800	92 ± 37	1.4	665 489 803	77 261 184
3x3	200x200	161 ± 80	43.9	9 006 796	1 686 912
3x3	400x400	105 ± 53	18.9	56 268 947	10 419 840
3x3	800x800	219 ± 67	6.2	2 369 103 880	51 603 552

Interpretation:
As can be seen the latency is lowest for the 4×1 topology followed by 2×2 and 3×3. This is not surprising as the 4×1 topology has the least amount of neighbors to communicate with per processor. The worst latencies can generally be observed for the 3×3 topology because every processor has to communicate with 2,3 or 8 neighbors. As can be seen by the huge discrepancy in the latency percentage for the 3×3 topology, the latency is strongly dependent on the grid size (problem size). While nearly half of the time is spent in latency for the 200×200 grid, only 6.2% of the time is spent in latency for the 800×800 grid.

1.2.12 Exchange Border potential improvements

Indeed we communicate twice as much as we need after each `Do_Step` call. We shall analyze this considering the following points:

- address of the first point to exchange: This depends on the parity of the processor. It is simply the first or second point in the grid depending on the parity (leaving out the edge case where the processor only has one data-point).
- the number of points to exchange: This normally is the number of points in the grid minus ghost cells divided by two. However, this may change if there is an odd number of points in the grid (then we have to exchange one more or one less point).
- the number of points in between grid points that have to be exchanged: The stride of the data will change. Currently we exchange every point for one direction and every `dim[Y_DIR]`-th point for the other direction. This can be optimized to exchange every second point in one direction and every second `dim[Y_DIR]`-th point in the other direction.

Is it worth it?
For smaller gridsizes it is not worth it to optimize the border exchange. The time spent in the border exchange might be significant in relative terms, but the absolute time is still small. For larger gridsizes it might be worth it to optimize the border exchange. As we have seen, this becomes more significant as the processor count and gridsizes grow. For our current problem and similarly sized problems the effort put into optimizing the border exchange is certainly not worth it.

2 Finite elements simulation

We will now shift our focus to a more general grid which is based on triangulation. In this section we will compare our parallel implementation from the previous section and disect the differences and similarities. For that reason we shall use the same sources in our grid as given in `sources.dat`.
Note that the sections for the exercises will be labeled as (2.1, 2.2, 2.3, ...) corresponding to exercises (4.1, 4.2, 4.3, ...) in the lab manual.

2.1 Code understanding & `Exchange_Borders`

The first step is to read through the code in **MPI_Fempois.c** and to understand it. Furthermore, we have to implement the `Exchange_Borders` function for which only a skeleton is given. The function should exchange the border values of the local grid with the neighboring processes.
The implementation of this function is quite straight forward. We only have to loop over all the neighbors of a process and send out the border values and receive the border values from the neighbors. The function is implemented as follows:

```
void Exchange_Borders(double *vect)
{
    for (size_t i = 0; i < N_neighb; ++i)
    {
        MPI_Sendrecv(vect, 1, send_type[i], proc_neighb[i], 0, //send
                     vect, 1, recv_type[i], proc_neighb[i], 0, //recv
                     grid_comm, &status);
    }
}
```

2.2 Time benchmarking

Next we turn our attention to timing of different sections in the code.
We have to measure:

- Time spent in computation
- Time spent exchanging information with neighbors
- Time spent doing global communication
- Idle time

We setup the following variable to measure / deduce the time spent in the different sections:

```
    double total_time = 0.0;
    double exchange_time_neighbors = 0.0;
    double exchange_time_global = 0.0;
    double compute_time = 0.0;
```

We will measure the time spent in computations by timing the solve function and subtracting the time spent in the `MPI_Allreduce` calls. The time spent in the `MPI_Allreduce` calls is the time spent in global communication. The time spent in exchanging information with neighbors is the time spent in the `Exchange_Borders` function. Finally, the idle time can be determined by summing up the differences between the cores total time and the slowest cores total time. Note that this way of calculating the idle time is an approximation since it is assumed that the slowest core doesn't have any idle time. However, I've discussed this with the TA and he said that this is a valid way of calculating the idle time for this exercise.
The commandline output for runs is as shown in Figure 17 for an example run:

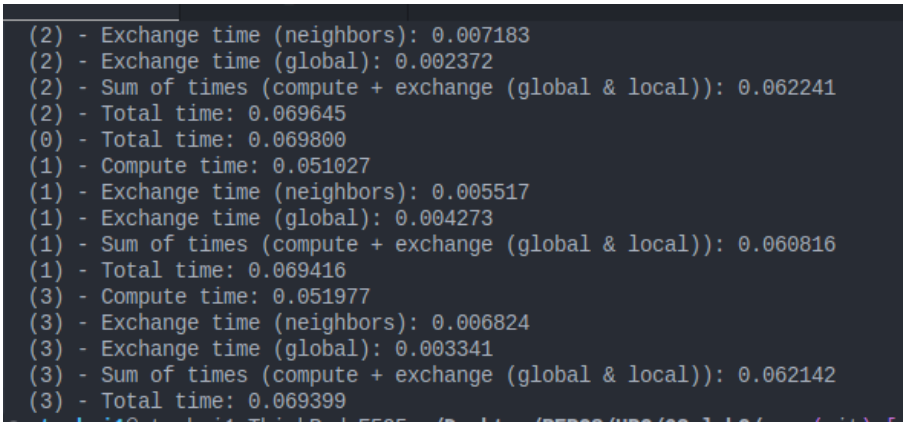


Figure 17: Timing for the different sections.
(x) ... denotes the process rank
Compute time ... time spent in the `solve` function only on computing
Exchange time (neighbors) ... time spent in `Exchange_Borders`
Exchange time (neighbors) ... time spent in `MPI_Allreduce` calls
Sum of times ... total time spent in compute and communication (excluding setup / idle time)
Total time ... total time spent in the program

The idle time is calculated as denoted above and therefore not shown in the output in Figure 17. We get the following results from our Delft Blue runs in Table 8:

Table 8: Rank averaged time benchmark for different grid sizes and topologies.
All times are in milliseconds (ms).
Note: WTime does also include setup and teardown (mallocs, frees, etc.) - therefore the sum of the times is not equal to WTime.

Top.	Grid Size	WTime (avg)	Comp. (avg)	Ex. Neighb. (avg)	Ex. Global (avg)	Idle (avg)
1x4	100x100	82.0	18.6	1.3	1.7	9.0
1x4	200x200	194.5	150.6	3.6	6.8	12.1
1x4	400x400	1498.2	1357.8	10.1	26.4	9.1
2x2	100x100	43.7	18.6	1.6	1.4	9.1
2x2	200x200	184.0	145.2	4.3	8.1	5.7
2x2	400x400	1428.7	1279.2	11.5	22.4	25.6

We see that the total time is comparable between the two topologies for all grid sizes. Furthermore, the total runtime also increases non-surprisingly with the gridsize.

Why does computation time increase faster than linearly?

One could expect, that the grid with 200×200 elements would take 4 times as long as the grid with 100×100 elements. However, the runtime roughly 8 times longer. This happens because additionally to the higher number of grid points the algorithm also takes longer (more iterations) to converge and hence the computation time is longer.

Analysis of the different times

To get a better grasp on the data a stacked bar plot, as shown in Figure 18, is created. We immediately see that the bulk of the time is spent on computing the solution. The second biggest contributor is the global exchange time and the idle time, followed by local exchange time.
That computation takes up the biggest part of the time is of non surprise especially on the bigger grids this is to be expected. Rather surprisingly the global excahnge time takes up a lot of time. This begs the question why two lines of the form:

```
1 MPI_Allreduce(... , ... , 1, MPI_DOUBLE, MPI_SUM, grid_comm);
```

take up this much time. This actually comes down to masked idle time. While the operation itself only sums up the values of 4 double, the operation is blocking and therefore the other cores have to wait for the slow-

est core to finish. If we look back at the definition of idle time, we defined it as the difference of the total time of the slowest core and the total time of the current core. However, as stated above, this does not take waiting time in MPI calls (like `MPI_Allreduce`) into account. Certainly, this explains now that the global exchange time is high, because every core has to wait for the others to synchronize to compute the `MPI_Allreduce`.

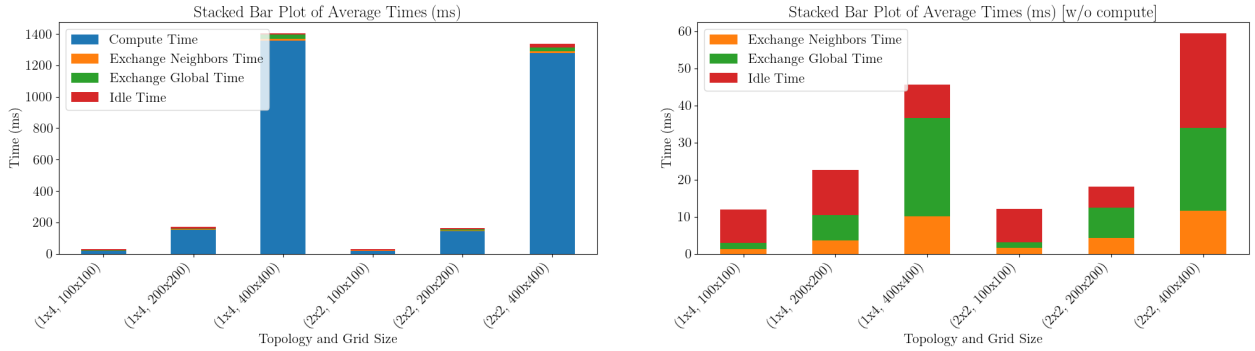


Figure 18: Scaling behavior of the Poisson solver with different grid sizes and processor topologies. with compute time (left) and without compute time (right).

We conclude that most of the time spent on communication in our solver is actually due to the waiting time in the `MPI_Allreduce` calls and the other idle time (waiting for the slowest core to finish). The actual exchange of data only represents a fraction of this time. Biggest, especially for larger grids, contributor for the time is still the computation which takes up north of 95% of the total time on larger grids.

2.3 Data exchange amount

This section is about the amount of data exchanged each iteration among one process with all its neighbors. We assume a uniformly triangulated grid which is partitioned stripe-wise and distribution over P processes. Furthermore, we assume that each process has to send the same amount of data. Indeed, this is not generally the case but for examples with periodic boundary conditions this is for example a valid assumption.
Now we see that every process has to communicate with $2d$ neighbors, where d is the dimension of the grid, under our assumptions there are 2 neighboring processes. With the assumption of a n^2 grid, our process communicates n values with each of these neighbors. For striped partitioning we get:

$$\text{Data exchanged per Process} = 2n$$

or in total:

$$\text{Data exchanged} = 2n \cdot P$$

Let's check for the extreme case 1000×1000 grid and $P = 500$ processes. We get:

$$\text{Data exchanged} = 2 \cdot 1000 \cdot 500 = 1000000$$

Which is the same amount of data as there are datapoints. This makes sense because every process has to communicate the top row upwards and the bottom row downwards.
Note, that we could even do worse if we assigned every process a single row. In this case every process would have to communicate the same data upward and downward. This would result in a data exchange of $2 \cdot 1000 \cdot 1000 = 2000000$ which is double the amount of data as there are datapoints.
For a box partitioning a process has to communicate $n/\sqrt{P}$ (assuming divisibility with all neighbors) with each of its 4 neighbors. We get:

$$\text{Data exchanged per Process} = 4 \cdot n/\sqrt{P}$$

or in total:

$$\text{Data exchanged} = 4 \cdot n \cdot \sqrt{P}$$

This means that a striped partitioning is less efficient compared to a boxed partitioning starting from $P = 4$, as can be seen in Figure 19.

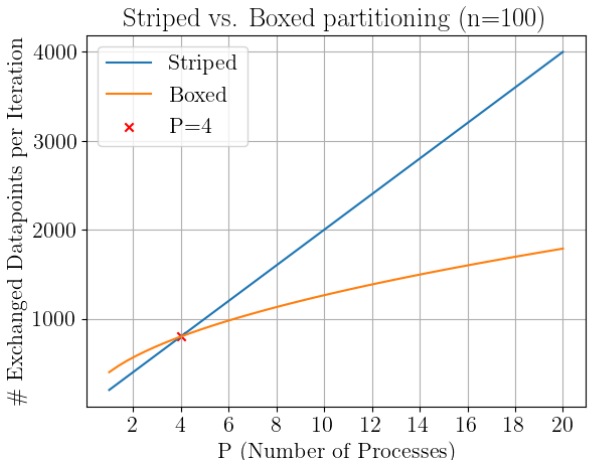


Figure 19: Striped vs. Boxed partitioning on a 100×100 grid using P processors.

2.4 Unbalanced communication

In the last section we assumed that every process has to communicate with the same amount of neighbors in our given scenario. That is actually not entierly true. There is an imbalance, even if just a small one.

But where does this imbalance come from?
Let's take a look at a **uniformly triangulated grid** with $P = 4$ processes. We assume that the grid is box-partitioned as given in Figure 20.

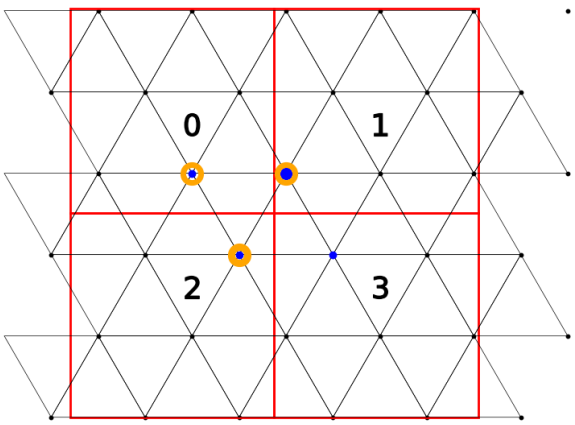


Figure 20: Box partitioning of a uniformly triangulated grid with $P = 4$ processes.
One point (big-blue) in 1 (and similarly in 3) has to communicate its value to 3 neighbors (small-blue).
Another point (big-orange circle) in 2 (and similarly in 4) has to communicate its value to 2 neighbors (small-orange circles).

We see that the geometry of a given uniform triangulated grid leads to an imbalance in communication. This imbalance is usually not a big problem, especially for bigger grids with comparably small amounts of processors because only 2 points have to communicate with 3 neighbors. However, for smaller grids or proportionally high processor counts the imbalance is more significant.

Looking at a 3×3 grid with $P = 9$ processors, we see that the imbalance is more significant as shown in Figure 21.

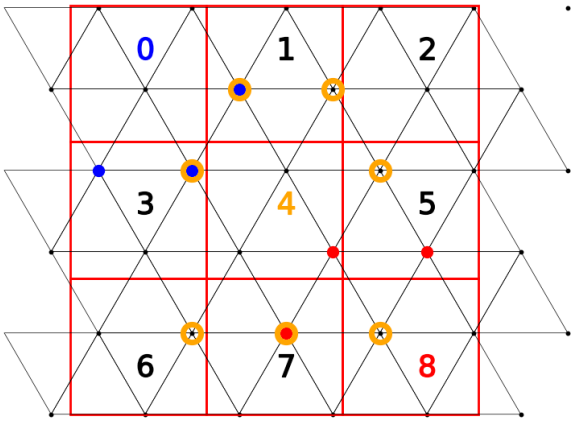


Figure 21: Box partitioning of a uniformly triangulated grid with $P = 9$ processes.
Corner (0/8 - blue/red) processes have to communicate their values to 2 or 3 neighbors respectively (1 and 3 - blue or 4, 5 and 7 - red).
Central (4 - orange) processes have to communicate their value to 6 neighbors (1, 3, 5, 6, 7 and 8).

As we can see the imbalance comes from the partitioning geometry on the uniform triangulated grid. Because our processor grid is rectangular the results will always, in one way or another come out as shown in the schematic above.
For 3×3 grid we have 4 corner processes which have to communicate with 2 or 3 neighbors and 1 central process which has to communicate with 6 neighbors. This is a significant imbalance and compared to the 2×2 grid it is not negligible, because the central process has to communicate more than 2 extra points (whole edges of datapoints) to its neighbors.

2.5 Estimates for computation ≡ communication time

N.B. This section will use two different ways to estimate the equilibrium point.

1) Let's assume that we have 4 processes and want to estimate for which grid size the computation time is equal to the communication time. One way to estimate this is by fitting a polynomial of degree 2 to the computation and a linear fit to the exchange time for our measurements in Table 8 and find the point where the times are equal. Thereafter we can use this information to determine the grid size where this happens. This process gives us a value of $n \approx 84$ for the grid size as shown in Figure 22

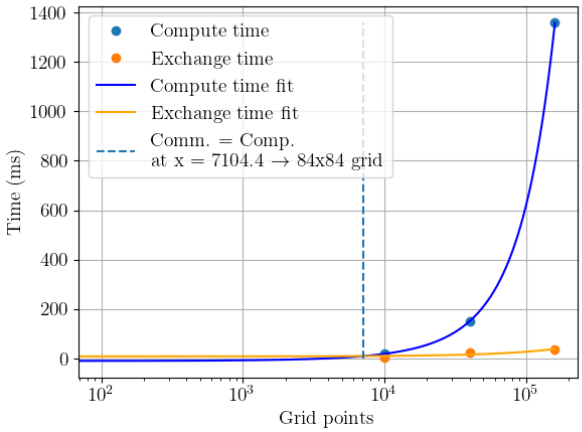


Figure 22: Fitting expected computation time ≡ communication time for 4 processes.

2) We'll also estimate this equilibrium point for a 1000×1000 grid. However, this time around we'll actually use a different way of calculating the number of Processes. We use the definitions from the lecture for stencil type computation:

- Computation time: $T_S = \text{number\_ops} \cdot t_{op} \sim 4t_{op} \cdot n^2$ (for stencil type - overall)
- Communication time: $T_{comm} = \text{number\_comm} \cdot t_{comm} \sim 2n \cdot t_{data}$ (for stencil type - overall)

Note that the definitions above apply for a single iteration and **one** processor!
We can calculate the time for communication from the data in Table 8 as follows:

$$t_{data}(P) = \frac{t_{global\_comm} + t_{neighbor\_comm}}{2 \cdot n \cdot P \cdot \#_{iterations}}$$

Similarly we can calculate the time for computation as follows:

$$t_{op} = \frac{t_{comp.}}{4 \cdot n^2 \cdot \#_{iterations}}$$

Note that the later doesn't depend on the number of processes.
Using the data from Table 8 we determine that one operation takes $t_{op} = (3.3 \pm 0.3) \times 10^{-6}$ ms. From that we can determine the computation time $t_{comp.}$ for $n = 1000$ as follows:

$$t_{comp.} = 4 \cdot n^2 \cdot t_{op} \cdot \#_{iterations} = 4 \cdot 1000^2 \cdot 3.3(3) \cdot 10^{-6} \cdot 1241 = (17.8 \pm 1.5)\,\text{s}$$

where 1241 ($\#_{iterations}$) is the number of iterations to converge. We set $t_{comp.} = t_{data}$ and solve for P. We need values for t_{global_comm} and $t_{neighbor\_comm}$ and take a look at Figure 22 and decide to further investigate the trend for the excahnge time ($= t_{global\_comm} + t_{neighbor\_comm}$). We decide to use a linear extrapolation as shown in Figure 23. We get an estimated communication time of 100 ms.

Now we have everything to solve for P and using 1241 for $\#_{iterations}$ again we get $P = 29 \pm 3$ processes.

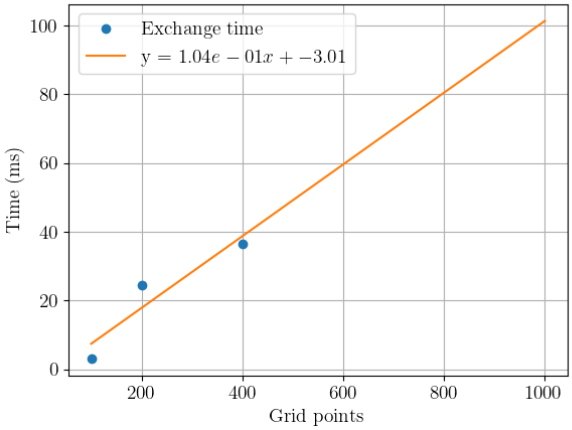


Figure 23: Extrapolating the communication time for $n = 1000$ using a linear fit.

Our predictions are: a 84×84 grid for 4 processes with equal computation and communication times and a 1000×1000 grid for 29 processes (also having equal computation and communication time).
We measure the ratio $t_{\text{comp}}/t_{\text{comm}}$ and get the following results for a 4×1 topology in Table 9:

Table 9: Ratio of computation to communication time for different grid sizes.

Grid Size	Ratio
10x10	0.53 ± 0.06
20x20	0.84 ± 0.15
30x30	0.86 ± 0.13
40x40	1.97 ± 0.24
50x50	2.66 ± 0.61
75x75	4.37 ± 1.00
84x84	3.70 ± 0.85
90x90	6.06 ± 1.94

We see that our initial guess of 84×84 grid for 4 processes is off by a long-shot. However, we also notice that the ratio is highly volatile as can be seen by the high uncertainties on the ratio.
A rerun of the code produced notably different ratios (e.g. 1.35 ± 0.25 for 30×30 grid). It seems that the actual position of the equilibrium point is around $n = 35$. Being a physicis, I have to say that our initial guess was of the same magnitude and therefore actually not too far off by our standards ☺.

Let's see how our prediction for the 1000×1000 grid with 29 processes holds up. Using a 29×1 topology we get 4.04 ± 1.83 for the ratio. It seems we have made a lower prediction in terms of P for the equilibrium point. However, the ratio is in the same ballpark as the ratio for the 84×84 grid which is not surprising since we used the same underlying data to make the prediction.

2.6 Adaptive gird

We will now make our grid denser in the source point areas. `GridDist` already implements an argument (*adapt*) for that purpose. In order to gauge the speed of convergence we let process 0 print the current precision of the solution. We perform runs with a reference and a denser grid to compare the convergence speed, convergence count and amount of computing time for a 2×2 topology and a 100×100, 200×200 and 400×400 grids.
We first take a look at the iterations needed for convergence using the standard percision goal of 0.0001 we get the results in Table 10.
It becomes clear that the adaptive grid needs a few more iterations to converge. This is most likely due to the fact that the dense region near the sources initially lead to a slower ‚spreading' of the solution near the source.

Table 10: Iterations needed for convergence on 2×2 topology for different grid sizes with and without adapt keyword.

Grid Size	100x100	200x200	400x400
Iterations Reference	141	274	529
Iterations Adapt	146	278	532

Does such a distored grid lead to faster convergence?
We can answer this questin with a clear no, at least in terms of iterations. Next we'll look at the time it takes to converge.

A bar-plot of the time till convergence is shown in Figure 24. We see that the time till convergence is roughly the same for the reference and the adaptive grid. This is not surprising since the time till convergence is mainly determined by the number of iterations needed.

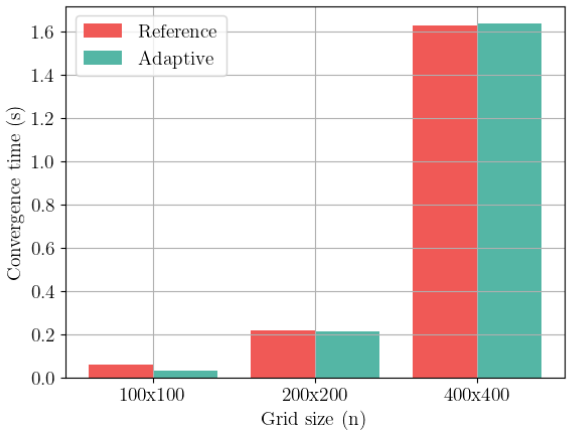


Figure 24: Time till convergence (with setup) on 2×2 topology for different grid sizes with and without adapt keyword.

Does it affect the speed of convergence?
Also no for this one. The time for the reference / adaptive grid is roughly the same with one exception for the 100×100 grid.

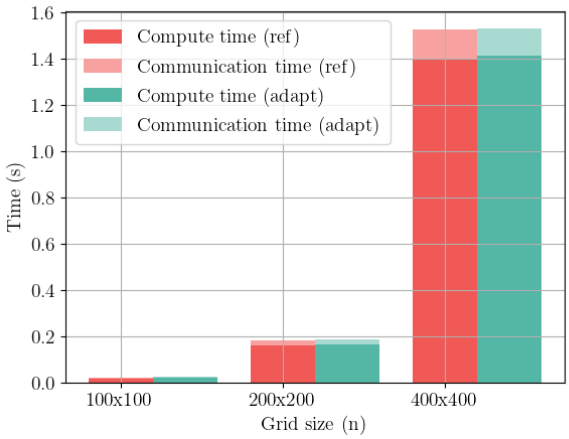


Figure 25: Time till convergence (without setup) on 2×2 topology for different grid sizes with and without adapt keyword.

If we stack the time needed for computation and communication we see that the computation as well as the

communication time is roughly the same for the reference and the adaptive grid as shown in Figure 25.

Does it affect the amount of computing time?
So again no, the amount of computing time is roughly the same for the reference and the adaptive grid.

Why is there hardly any difference between the two?
It seems that in our rather simple example (3 sources) the adaptive grid doesn't lead to a faster convergence. This can likely be attributed to the low complexity of our problem. Due to this low complexity our convergence in iterations is mostly gouverned by the spreading of the solution from the sources to the rest of the grid. Adaptive grids slow down this spreading which is needed to reduce our error in order to converge. The local error in the vicinity of the sources is not the limiting factor as it seems for the convergence in our case.

3 Eigenvalue solution by Power Method on GPU

The last problem concerns the evaluation of eigenvalue using the power method via a paralellized CUDA code. Reference for this implementation is a sequantial CPU-code provided by the course (`power_cpu.cu`).

A scematic overview of the iteration loop for the power method is shown bellow in algorithm 1. The whole implementation details can be found in the appendix for the current section.

Algorithm 1 GPU Power Method

Input: Matrix $\mathbf{A}$ of size $N \times N$, tolerance ϵ, maximum iterations max_iter
Output: Dominant eigenvalue λ
Initialize $\mathbf{v}$ with $v_1 = 1, v_i = 0$ for $i > 1$
$\lambda_{\text{Old}} \leftarrow 0$, $\lambda \leftarrow 0$
Allocate GPU memory for $\mathbf{A}$, $\mathbf{v}$, $\mathbf{w}$, and λ
Copy $\mathbf{A}$ and $\mathbf{v}$ to GPU memory
$\mathbf{w} \leftarrow \mathbf{A} \cdot \mathbf{v}$ ▷ First iteration of $\mathbf{w}$ computation using `Av_Product` kernel
for $i = 0$ to $max\_iter - 1$ **do**
    Compute norm of $\mathbf{w}$: norm $\leftarrow \sqrt{\mathbf{w}^T \cdot \mathbf{w}}$ ▷ Using `FindNormW` kernel
    Normalize $\mathbf{v}$: $\mathbf{v} \leftarrow \mathbf{w}/\text{norm}$ ▷ Using `NormalizeW` kernel
    Compute $\mathbf{w} \leftarrow \mathbf{A} \cdot \mathbf{v}$ ▷ Using `Av_Product` kernel
    Compute eigenvalue: $\lambda \leftarrow \mathbf{v}^T \cdot \mathbf{w}$ ▷ Using `FindNormW` kernel
    if $|\lambda - \lambda_{\text{Old}}| < \epsilon$ **then**
        Break ▷ Convergence achieved
    end if
    $\lambda_{\text{Old}} \leftarrow \lambda$
end for
Copy λ back to host memory
Deallocate GPU memory

Note: A *sqrt*() was added in the `NormalizeW` kernel over `g_NormW[0]`. This way we can use the output of `FindNormW` directly in the `NormalizeW` kernel.

I performed all the following measurements after throwing away the first GPU run (burner-run). The reason beaing, that the first run always took around 10 times longer than the following runs. This is likely due to some GPU initialization and setup overhead. It should also be noted, that I freed the GPU memory after each run to avoid caching effects. The basic setup in the main looks like this schematically:

```
// This is the starting points of GPU
RunGPUPowerMethod(N); // burner run!
// Step 1
printf(">>>Step 1\n");
GLOBAL_MEM = true;
for(int i = 0; i < 10; ++i){
  RunGPUPowerMethod(N);
  CleanGPU();
}
GLOBAL_MEM = false;
printf(">>>Step 1 shared mem\n");
for(int i = 0; i < 10; ++i){
  RunGPUPowerMethod(N);
  CleanGPU();
}
// Step 2:
PRINTLEVEL = 0;
int Ns[] = {50, 500, 2000, 4000, 5000};
for(int i = 0; i < 1; ++i){
 N = Ns[i];
 double time = RunGPUPowerMethod(N);
 printf("%d - GPU: run time = %f secs.\n",N,time);
 CleanGPU();
}
Cleanup();
```

Here `RunGPUPowerMethod` runs the power method on the GPU and on the top we can see the burner run. `CleanGPU` is a function that frees the allocated memory on the GPU as mentioned above.

Iterations to convergence needed

While performing the different measurements on the GPU it stood out to me, that the iterations needed to reach the specified tolerance ($\epsilon = 0.000005$) varied quite a bit. It was evident, that the GPU reaches a value very close to the eigenvalue rather fast (on the second iteration). However, the final convergence seems to depend on the precision of the operations. For the CPU we always perform the same operations in the same order, thus leading to the same results every time. The GPU on the other hand perfroms the calculations in parallel and especially for the accumulation operations the order of operations will certainly vary. Small rounding error introduced by the floating point percision in combination with a small ϵ can now lead to differences: These arise because the rounding in the accumulation functions such as `FindNormW` and `ComputeLamda` is dependant on the order of operations (not associative)! It should also be noted that the T4 GPUs used on Google Colab support only FP32 operations, thus moving to FP64 wasn't an option.

All of the following benchmarks are perfromed in the supplied IPython notebook on Google Collab using T4 GPUs.

3.1 Step 1: Shared vs. global memory for matrix-vector multiplication

As can be seen in the code snippet from above, we perfrom multiple runs of the power method on the GPU. First with global memory and then with shared memory. This is done by using different kernels for the AV-product. The kernel used for shared memory is unchanged. Furthermore, to investigate the performance impact of shared vs. global memory during the matrix vector multiplication we first need an alternative kernel which doesn't use shared memory. This kernel is given bellow:

```
__global__ void Av_ProductGlobal(float* g_MatA, float* g_VecV, float* g_VecW, int N)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row >= N) return;

    float sum = 0.0f;
    for (int col = 0; col < N; col++) {
        sum += g_MatA[row * N + col] * g_VecV[col];
    }
    g_VecW[row] = sum;
}
```

Notice that the kernel above doesn't need to copy data from global to shared memory, this makes it possible to efficiently calculate the AV-product in a striped partitioning fashing. Furthermore, we do not need any syncronisation here in contrast to the original kernel. Given this, at least from a instruction flow perspective, one may think that the global memory kernel is heavily favoured due to it's simplicity. We shall see if this holds up given our measurements.
Subsequently, 10 measurements are perfromed with the original kernel and the changed kernel to determine the mean total runtimes depending on memory usage patterns. We obtain a **scatter plot** seen in Figure 26

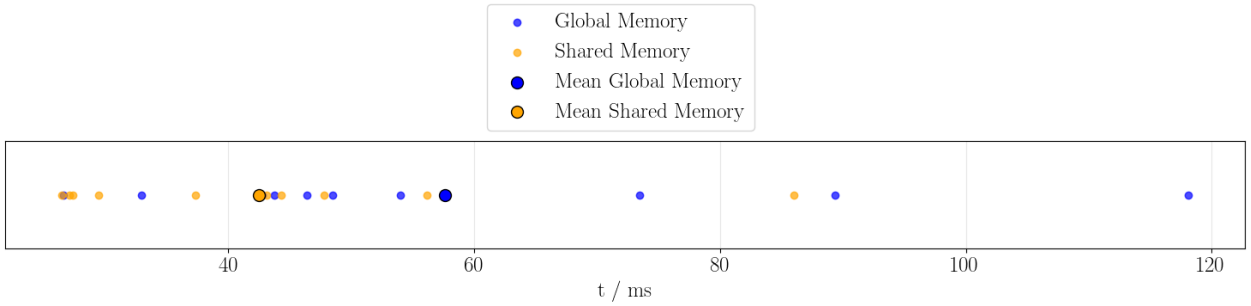


Figure 26: Measurements

Furthermore, we obtain a mean time using global memory of $t_{global} = (57.6 \pm 0.7)\,\text{ms}$ and by using shared memory in this kernel we get $t_{shared} = (42.5 \pm 0.3)\,\text{ms}$. This means we have time savings of $(26 \pm 1)\,\%$ or about

1/4 when using shared memory compared to global memory.

Why is shared memory faster?

First and foremost we shall keep in mind, that the shared memory kernel is at a disadvantage in our comparison with the global kernel. To get insights why it still performs faster we look at benchmark results for the T4 cards to answer this question. According to a benchmark by Liu, et al. [2] not all cards have lower latency for shared memory compared to global memory. However, as their research showed the latency for the T4 architecture is actually lower by a factor of around 2-10 (depending on the number of threads simultaniously accessing the same memory bank). Additional, the bandwidth of the shared memory is much higher (> Tb/s [3]) compared to the global memory (360 Gb/s). A lower latency and higher bandwidth for the shared memory thus makes the difference. In our problem, matrix-vector multiplication, we use the matrix rows multiple time (dimensionality of vector!) and therefore the first copy operation of the data from global to shared memory pays off.
Stated differently, we sacrefice at first by copying to shared memory to gain better data locality and use this to our advantage in the actual calculation.

3.2 Step 2: Execution time for different N and threads per block

I implemented a small loop to run the GPU code for 5 different N with $N \in \{50, 500, 2000, 4000, 5000\}$. The resulting time benchmarks for 32, 64 and 100 threds per block can be seen in Figure 27.

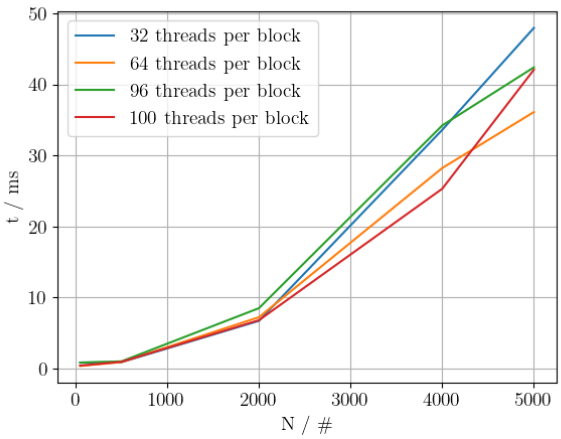


Figure 27: Runtime for $N \in \{50, 500, 2000, 4000, 5000\}$ and for 32, 64 and 100 threads per block respectively.

Looking at the results we do not see an overall fastest option with 64- and 100-threads per block changing places. On the other hand, the 32- & 96-threads per block option seem to be the slowest in terms of total runtime.

Interpretation of the results

The threads of a block are usually further subdivided into different wraps which perform the actual SIMT opertations. NVidia GPUs usually have a warp size of 32 threads which means for our first option (blue line in Figure 27) we only have one warp per block. Since warps are scheduled at the same time for the same operation (SIMT) pipelining within one wrap is not possible. However, multiple wraps in one block might be pipelined in order to hide latency. Therefore, it is plausible that for the second option (orange line in Figure 27) two wrap could hide some latency of each other. While one wrap is computing stuff using the ALU the other one might access memory already and vice versa.
Given the not so clear result between the with 64- and 100-threads case we should focus the multiplicity of 32 first. It would inherently mean that we have 4 wraps and the last has 28 (32*4 - 28 = 100) threads which lay dormant. This is generally inefficient because these threads can't do work due to the software limitation imposed.
Ultimately this problem is a very complex problem of latency and latency hiding and (used/unused) hardware

[2] Liu, Andy T., Yang, Shu-wen, Chi, Po-Han, Hsu, Pei-Hung, and Lee, Hung-yi. "Mockingjay: Unsupervised Speech Representation Learning with Deep Bidirectional Transformer Encoders." *arXiv preprint arXiv:1903.07486*, 2019. https://arxiv.org/abs/1903.07486

[3] I sadly couldn't find information about the exact bandwidth of the shared memory

optimization. There is no clear trend, as can be seen by the 96-threads per block option which is not really noticably better most of the time compared to the 32-threads per block option. In practice it is best to determine the most efficient thread per block size for the given problem and grid size experimentally.

3.3 Step 3: Speedups

Now we really dive into the benefits of GPUs over CPUs in terms of computation speed for paralellizable programs.

We measure two different scenario's computation time:

i excluding time of memory copy from CPU → GPU

ii including time of memory copy from CPU → GPU

After measuring 5 rounds without (i) and with (ii) memory access time we obtain the following **scatter plot** in Figure 28. For these measurements the initial shared memory kernel was used.

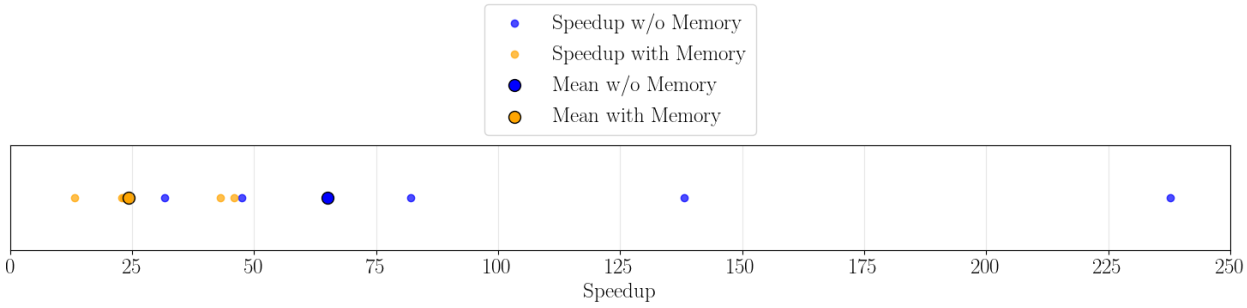


Figure 28: Speedup of GPU implementation vs. CPU with and without memory transfer times.

The mean speedup without memory access times is ×65 and with memory access timed it comes out to ×24.

Why is there such a huge speedup when switching to GPU?
As touched upon above, the SIMD/SIMT nature of GPUs let's them parallize work much more efficiently than a CPU. While it is true that a CPU could also use multiple threads[4] or processes[5], CPUs have way fewer cores (roughly a factor of 100) compared to GPUs. The high number of seperate Cores makes GPUs so fast because they all run simultaniously on different parts of the data.
A short theoretical derivation may give us more insights into the speedup we might expect from a GPU:
Given that a CPU (take a typical Ryzen Server CPU with 64 cores) has around 50 GFLOPS/core and a T4 GPU has 8 TFLOPS in total in single-precision. We pit them against each other (CPU without MPI -> only one core):

$$\text{speedup}_{\text{theory}} = {}^{8000}/_{50} = 160$$

We can expect a speedup of around 160 times without taking transfer time and other overhead of a GPU into account. This seems quite plausible because we indeed measure a speedup of a factor around 65. The reduction from 160 to 65 is likely due to the added complexity in memory operations → copying into shared memory and out again and setup times for the GPU kernels.

3.4 Step 4: Explanation of the Results

The GPU implementation of the power method demonstrates significant performance benefits across multiple aspects. In **Step 1**, comparing global and shared memory usage, we observed a clear advantage for shared memory, with a time saving of around 26 %. This improvement is due to the lower latency and higher bandwidth of shared memory compared to global memory, which allows for efficient reuse of matrix rows during matrix-vector multiplication. Despite the initial cost of copying data into shared memory, the better data locality and reduced global memory accesses outweigh this overhead, leading to faster overall computation.

[4]This is of course only "simulated" or software-level parallelisim since they are scheduled after one another by the OS!
[5]This on the other hand is true parallelisim: e.g. MPI

In **Step 2**, the analysis of execution time for varying numbers of threads per block highlighted the importance of optimizing thread configurations. Blocks with 64 threads performed better for the biggest problem, likely due to efficient warp utilization, allowing latency hiding between multiple warps. Blocks with 32 threads suffered from insufficient pipelining as only one warp could be active at a time, while 96- and 100-thread (for the biggest problem) configurations showed diminishing returns due to inefficiencies such as underutilized threads or increased memory contention. This emphasizes the need to experimentally determine the optimal thread count for a specific problem and grid size.

Finally, in **Step 3**, we observed a significant speedup of $\times 65$ without memory transfer times and $\times 24$ when including memory transfers, compared to the CPU implementation. The GPU's massively parallel architecture allows it to handle matrix-vector multiplications more efficiently than CPUs with limited cores. However, the reduction in speedup when including memory transfers reflects the overhead of data movement between CPU and GPU memory. The theoretical speedup derived from FLOPS capabilities aligns well with these results, validating the practical measurements. Variations in GPU convergence times were attributed to floating-point precision and non-associative operations, which differ from the deterministic order of operations on CPUs.

Overall, the results highlight the importance of leveraging GPU-specific optimizations like shared memory and optimal thread configurations while managing overheads such as memory transfers (especially form CPU to GPU and back).

Appendix - Introductory exercise

The following code was used for the ping pong task:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

// Maximum array size 2^20= 1048576 elements
#define MAX_EXPONENT 20
#define MAX_ARRAY_SIZE (1<<MAX_EXPONENT)
#define SAMPLE_COUNT 1000

int main(int argc, char **argv)
{
    // Variables for the process rank and number of processes
    int myRank, numProcs, i;
    MPI_Status status;

    // Initialize MPI, find out MPI communicator size and process rank
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);


    int *myArray = (int *)malloc(sizeof(int)*MAX_ARRAY_SIZE);
    if (myArray == NULL)
    {
        printf("Not enough memory\n");
        exit(1);
    }
    // Initialize myArray
    for (i=0; i<MAX_ARRAY_SIZE; i++)
        myArray[i]=1;

    int number_of_elements_to_send;
    int number_of_elements_received;

    // PART C
    if (numProcs < 2)
    {
        printf("Error: Run the program with at least 2 MPI tasks!\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    double startTime, endTime;

    // TODO: Use a loop to vary the message size
    for (size_t j = 0; j <= MAX_EXPONENT; j++)
    {
        number_of_elements_to_send = 1<<j;
        if (myRank == 0)
        {
            myArray[0]=myArray[1]+1; // activate in cache (avoids possible delay when sending
    the 1st element)
            startTime = MPI_Wtime();
            for (i=0; i<SAMPLE_COUNT; i++)
            {
                MPI_Send(myArray, number_of_elements_to_send, MPI_INT, 1, 0,
                    MPI_COMM_WORLD);
                MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
                MPI_Get_count(&status, MPI_INT, &number_of_elements_received);

                MPI_Recv(myArray, number_of_elements_received, MPI_INT, 1, 0,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            } // end of for-loop

            endTime = MPI_Wtime();
            printf("Rank %2.1i: Received %i elements: Ping Pong took %f seconds\n", myRank,
    number_of_elements_received,(endTime - startTime)/(2*SAMPLE_COUNT));
        }
        else if (myRank == 1)
        {
            // Probe message in order to obtain the amount of data
            MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

```
            MPI_Get_count(&status, MPI_INT, &number_of_elements_received);

            for (i=0; i<SAMPLE_COUNT; i++)
            {
                MPI_Recv(myArray, number_of_elements_received, MPI_INT, 0, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                MPI_Send(myArray, number_of_elements_to_send, MPI_INT, 0, 0,
                MPI_COMM_WORLD);
            } // end of for-loop
        }
    }

    // Finalize MPI
    MPI_Finalize();

    return 0;
}
```

For the bonus task, the following code was used:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

// Maximum array size 2^20= 1048576 elements
#define MAX_EXPONENT 20
#define MAX_ARRAY_SIZE (1<<MAX_EXPONENT)
#define SAMPLE_COUNT 1000

int main(int argc, char **argv)
{
    // Variables for the process rank and number of processes
    int myRank, numProcs, i;
    MPI_Status status;

    // Initialize MPI, find out MPI communicator size and process rank
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);


    int *myArray = (int *)malloc(sizeof(int)*MAX_ARRAY_SIZE);
    if (myArray == NULL)
    {
        printf("Not enough memory\n");
        exit(1);
    }
    // Initialize myArray
    for (i=0; i<MAX_ARRAY_SIZE; i++)
        myArray[i]=1;

    int number_of_elements_to_send;
    int number_of_elements_received;

    // PART C
    if (numProcs < 2)
    {
        printf("Error: Run the program with at least 2 MPI tasks!\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
    double startTime, endTime;

    // TODO: Use a loop to vary the message size
    for (size_t j = 0; j <= MAX_EXPONENT; j++)
    {
        number_of_elements_to_send = 1<<j;
        if (myRank == 0)
        {
            myArray[0]=myArray[1]+1; // activate in cache (avoids possible delay when sending
    the 1st element)
            startTime = MPI_Wtime();
            for (i=0; i<SAMPLE_COUNT; i++)
            {
                MPI_Sendrecv(myArray, number_of_elements_to_send, MPI_INT, 1,0,myArray,
```

```
    number_of_elements_to_send, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
            }

            endTime = MPI_Wtime();
            printf("Rank %2.1i: Received %i elements: Ping Pong took %f seconds\n", myRank,
    number_of_elements_to_send,(endTime - startTime)/(2*SAMPLE_COUNT));
        }
        else if (myRank == 1)
        {
            for (i=0; i<SAMPLE_COUNT; i++)
            {
                MPI_Sendrecv(myArray, number_of_elements_to_send, MPI_INT, 0,0,myArray,
    number_of_elements_to_send, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
            }
        }
    }

    // Finalize MPI
    MPI_Finalize();

    return 0;
}
```

The matrix multiplication used the following code:

```
/******************************************************************************
 * FILE: mm.c
 * DESCRIPTION:
 *   This program calculates the product of matrix a[nra][nca] and b[nca][ncb],
 *   the result is stored in matrix c[nra][ncb].
 *   The max dimension of the matrix is constraint with static array
 *declaration, for a larger matrix you may consider dynamic allocation of the
 *arrays, but it makes a parallel code much more complicated (think of
 *communication), so this is only optional.
 *
 ******************************************************************************/

#include <math.h>
#include <mpi.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NRA 2000 /* number of rows in matrix A */
#define NCA 2000 /* number of columns in matrix A */
#define NCB 2000 /* number of columns in matrix B */
// #define N 1000
#define EPS 1e-9
#define SIZE_OF_B NCA*NCB*sizeof(double)

bool eps_equal(double a, double b) { return fabs(a - b) < EPS; }

void print_flattened_matrix(double *matrix, size_t rows, size_t cols, int rank) {
    printf("[%d]\n", rank);
    for (size_t i = 0; i < rows; i++) {
        for (size_t j = 0; j < cols; j++) {
            printf("%10.2f ", matrix[i * cols + j]);  // Accessing element in the 1D array
        }
        printf("\n");  // Newline after each row
    }
}

int checkResult(double *truth, double *test, size_t Nr_col, size_t Nr_rows) {
    for (size_t i = 0; i < Nr_rows; ++i) {
        for (size_t j = 0; j < Nr_col; ++j) {
            size_t index = i * Nr_col + j;
            if (!eps_equal(truth[index], test[index])) {
                return 1;
            }
        }
    }
    return 0;
}
```

```
50
51 typedef struct {
52     size_t rows;
53     double *a;
54     double *b;
55 } MM_input;
56
57 char* getbuffer(MM_input *in, size_t size_of_buffer){
58     char* buffer = (char*)malloc(size_of_buffer * sizeof(char));
59     if (buffer == 0)
60     {
61         printf("Buffer couldn't be allocated.");
62         return NULL;
63     }
64     size_t offset = 0;
65     memcpy(buffer + offset, &in->rows, sizeof(size_t));
66     offset += sizeof(size_t);
67     size_t matrix_size = in->rows * NCA * sizeof(double);
68     memcpy(buffer + offset, in->a, matrix_size);
69     offset += matrix_size;
70     memcpy(buffer + offset, in->b, NCA*NCB*sizeof(double));
71     return buffer;
72 }
73
74 MM_input* readbuffer(char* buffer, size_t size_of_buffer){
75     MM_input *mm = (MM_input*)malloc(sizeof(MM_input));
76
77     mm->rows = ((size_t*)buffer)[0];
78     size_t offset = sizeof(size_t);
79     size_t matrix_size =  mm->rows * NCA;
80     mm->a = (double*)malloc(sizeof(double)*matrix_size);
81     mm->b = (double*)malloc(sizeof(double)*matrix_size);
82     memcpy(mm->a, &(buffer[offset]), matrix_size);
83     offset += matrix_size;
84     memcpy(mm->b, &(buffer[offset]), NCA*NCB*sizeof(double));
85     free(buffer);
86     return mm;
87 }
88
89
90 void setupMatrices(double (*a)[NCA], double (*b)[NCB], double (*c)[NCB]){
91     for (size_t i = 0; i < NRA; i++) {
92         for (size_t j = 0; j < NCA; j++) {
93             a[i][j] = i + j;
94         }
95     }
96
97     for (size_t i = 0; i < NCA; i++) {
98         for (size_t j = 0; j < NCB; j++) {
99             b[i][j] = i * j;
100        }
101    }
102
103    for (size_t i = 0; i < NRA; i++) {
104        for (size_t j = 0; j < NCB; j++) {
105            c[i][j] = 0;
106        }
107    }
108 }
109
110 double multsum(double* a,double* b_transposed, size_t size){
111     double acc = 0;
112     for (size_t i = 0; i < size; i++)
113     {
114         acc += a[i]*b_transposed[i];
115     }
116     return acc;
117 }
118
119 double productSequential(double *res) {
120     // dynamically allocate to not run into stack overflow - usually stacks are
121     // 8192 bytes big -> 1024 doubles but we have 1 Mio. per matrix
122     double(*a)[NCA] = malloc(sizeof(double) * NRA * NCA);
```

```
123     double(*b)[NCB] = malloc(sizeof(double) * NCA * NCB);
124     double(*c)[NCB] = malloc(sizeof(double) * NRA * NCB);
125 
126     /*** Initialize matrices ***/
127     setupMatrices(a,b,c);
128 
129     /* Parallelize the computation of the following matrix-matrix
130    multiplication. How to partition and distribute the initial matrices, the
131    work, and collecting final results.
132     */
133     // multiply
134     double start = MPI_Wtime();
135     for (size_t i = 0; i < NRA; i++) {
136         for (size_t j = 0; j < NCB; j++) {
137             for (size_t k = 0; k < NCA; k++) {
138                 res[i * NCB + j] += a[i][k] * b[k][j];
139             }
140         }
141     }
142     /*  perform time measurement. Always check the correctness of the parallel
143        results by printing a few values of c[i][j] and compare with the
144        sequential output.
145     */
146     double time = MPI_Wtime()-start;
147     free(a);
148     free(b);
149     free(c);
150     return time;
151 }
152 
153 double splitwork(double* res, size_t num_workers){
154     if (num_workers == 0) // sadly noone will help me :((
155     {
156         printf("Run sequential!\n");
157         return productSequential(res);
158     }
159 
160     double(*a)[NCA] = malloc(sizeof(double) * NRA * NCA);
161     double(*b)[NCB] = malloc(sizeof(double) * NCA * NCB);
162     double(*c)[NCB] = malloc(sizeof(double) * NRA * NCB);
163     // Transpose matrix b to make accessing columns easier - in row major way - better cache
    performance
164     setupMatrices(a,b,c);
165 
166     double start_time = MPI_Wtime();
167     double (*b_transposed)[NCA] = malloc(sizeof(double) * NCA * NCB);
168     for (size_t i = 0; i < NCA; i++) {
169         for (size_t j = 0; j < NCB; j++) {
170             b_transposed[j][i] = b[i][j];
171         }
172     }
173 
174     /*** Initialize matrices ***/
175     // given number of workers I'll split
176     size_t rows_per_worker = NRA / (num_workers+1); //takes corresponding columns from other
    matrix
177     printf("rows per worker: %zu\n", rows_per_worker);
178     size_t row_end_first = NRA - rows_per_worker*num_workers;
179     printf("first gets most: %zu\n", row_end_first);
180 
181     //setup requests
182     MPI_Request requests[num_workers];
183     MM_input *data_first = (MM_input*)malloc(sizeof(MM_input));
184     data_first->rows = row_end_first;
185     data_first->a = (double*)a; //they both start of with no offset!
186     data_first->b = (double*)b_transposed;
187     size_t total_size = sizeof(size_t) + (data_first->rows * NCA)*sizeof(double)+SIZE_OF_B;
188     char* buffer = getbuffer(data_first, total_size);    //first one
189 
190     // Tag is just nr-cpu -1
191     MPI_Isend(buffer, total_size, MPI_CHAR, 1, 0,MPI_COMM_WORLD, &requests[0]);
192     free(data_first);
193     total_size = sizeof(size_t) + (rows_per_worker * NCA)*sizeof(double) + SIZE_OF_B; //size
```

```
    is the same for all other - just compute once!
    size_t i;
    for (i = 0; i < (num_workers-1); ++i)
    {
        MM_input *data = (MM_input*)malloc(sizeof(MM_input));
        data->rows = rows_per_worker;
        data->a = (double*)(a + (row_end_first + rows_per_worker*i));
        data->b = (double*)(b_transposed); // send everyting - all needed
        buffer = getbuffer(data, total_size);
        printf("nr_worker - %zu\n", i);
        MPI_Isend(buffer, total_size, MPI_CHAR, i+2, i+1,MPI_COMM_WORLD, &requests[i+1]);
        free(data);
    }
    double* my_a = (double*)(a + (row_end_first + rows_per_worker*i));

    //I multiply the rest
    size_t offset = 0;
    for (size_t row = (NRA-rows_per_worker); row < NRA; row++)
    {
        for (size_t col = 0; col < NCB; col++)
        {
            res[row * NCB + col] = multsum(my_a+offset, (((double*)b_transposed)+col*NCA), NCA
    );
        }
        offset += NCA;
    }
    printf("My c: \n");
    //wait for rest
    MPI_Status stats[num_workers];
    if(MPI_Waitall(num_workers, requests, stats) == MPI_ERR_IN_STATUS){
        printf("Communication failed!!! - abort\n");
    }
    printf(">>>Everything sent and recieved\n");

    // reviece rest
    size_t buf_size = sizeof(double)*row_end_first*NCB;
    double* revbuf;
    offset = 0;
    for (size_t worker = 0; worker < num_workers; worker++)
    {
        revbuf = (double*)malloc(buf_size); //first gets largest buffer
        MPI_Recv(revbuf, buf_size/sizeof(double), MPI_DOUBLE, worker+1, worker, MPI_COMM_WORLD
    ,&stats[worker]);
        memcpy(&res[offset/sizeof(double)], revbuf, buf_size);
        free(revbuf);
        offset += buf_size;
        buf_size = sizeof(double)*rows_per_worker*NCB;
    }
    double time = MPI_Wtime()-start_time;
    //free all pointers!
    free(a);
    free(b);
    free(b_transposed);
    free(c);
    return time;
}



double work(int rank, size_t num_workers){
    size_t rows_per_worker = NRA / (num_workers+1);
    char* buffer;
    MPI_Status status;
    if (rank == 1) // first always get's most work
    {
        rows_per_worker = NRA - rows_per_worker*num_workers;
    }
    size_t size_of_meta = sizeof(size_t);
    size_t size_of_a = sizeof(double)*rows_per_worker*NCA;
    size_t buffersize = size_of_meta+size_of_a + SIZE_OF_B;
    buffer = (char*)malloc(buffersize);

    MPI_Recv(buffer, buffersize, MPI_CHAR, 0, rank-1, MPI_COMM_WORLD, &status);
```

```
264     double start = MPI_Wtime();
265     int count;
266     MPI_Get_count(&status, MPI_CHAR, &count);
267     printf("I'm rank %d and I got %d bytes (%ld doubles) of data from %d with tag %d.\n", rank
        , count, (count-sizeof(size_t))/sizeof(double), status.MPI_SOURCE, status.MPI_TAG);
268
269     MM_input *mm = (MM_input*)malloc(sizeof(MM_input));
270     mm->a = (double*)&buffer[size_of_meta];
271     mm->b = (double*)&buffer[size_of_meta+size_of_a];
272
273     double *res =(double*)malloc(sizeof(double)*rows_per_worker*NCB);
274
275     size_t offset = 0;
276     for (size_t row = 0; row < rows_per_worker; row++)
277     {
278         for (size_t col = 0; col < NCB; col++)
279         {
280             res[row * NCB + col] = multsum(mm->a+offset, (((double*)mm->b)+col*NCA), NCA);
281         }
282         offset += NCA;
283     }
284     MPI_Send(res, rows_per_worker*NCB, MPI_DOUBLE, 0,rank-1, MPI_COMM_WORLD);
285     printf("[%d] sent res home\n",rank);
286     free(res);
287     return MPI_Wtime() - start;
288 }
289
290 int main(int argc, char *argv[]) {
291     int tid, nthreads;
292     /* for simplicity, set NRA=NCA=NCB=N  */
293     // Initialize MPI, find out MPI communicator size and process rank
294     int myRank, numProcs;
295     MPI_Status status;
296     MPI_Init(&argc, &argv);
297     MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
298     MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
299     int num_Workers = numProcs-1;
300     if (argc > 1 && strcmp(argv[1], "parallel") == 0) {
301         // Variables for the process rank and number of processes
302        if (myRank == 0) {
303             printf("Run parallel!\n");
304             double *truth = malloc(sizeof(double) * NRA * NCB);
305             double time = productSequential(truth);
306             printf("Computed reference results in %.6f s\n", time);
307             printf("Hello from master! - I have %d workers!\n", num_Workers);
308             // send out work
309             double *res = malloc(sizeof(double)*NRA*NCB);
310             time = splitwork(res, num_Workers);
311             if (checkResult(res, truth, NCB, NRA)) {
312                 printf("Matrices do not match!!!\n");
313                 return 1;
314             }
315             printf("Matrices match (parallel [eps %.10f])! - took: %.6f s\n", EPS, time);
316             free(truth);
317             free(res);
318         } else {
319             double time = work(myRank, num_Workers);
320             printf("Worker bee %d took %.6f s (after recv) for my work\n", myRank, time);
321         }
322
323     } else  // run sequantial
324     {
325         printf("Run sequantial!\n");
326         double *res = malloc(sizeof(double) * NRA * NCB);
327         double time = productSequential(res);
328         if (checkResult(res, res, NCB, NRA)) {
329             printf("Matrices do not match!!!\n");
330             return 1;
331         }
332         printf("Matrices match (sequantial-trivial)! - took: %.6f s\n", time);
333         free(res);
334     }
335
```

```
336     MPI_Finalize();
337     return 0;
338 }
```

Appendix - Poisson solver

The parallel Poisson solver used the following code:
Note: Sbatch scripts used for the exercises will be included after the Poisson-solver code.

```c
/*
 * MPI_Poisson.c
 * 2D Poison equation solver (parallel version)
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <mpi.h>
#include <assert.h>

#define DEBUG 0

#define max(a,b) ((a)>(b)?a:b)


// defines for Exercises!

#define SOR 1
#define MONITOR_ERROR 1
#define FAST_DO_STEP_LOOP
// #define MONITOR_ALLREDUCE 1
// #define ALLREDUCE_COUNT 100
#define MONITOR_EXCHANGE_BORDERS
#define SKIP_EXCHANGE

#define DEFINES_ON (SOR || MONITOR_ERROR || 0)
//defines end

enum
{
    X_DIR, Y_DIR
};

// only needed for certain configs!
#ifdef SOR
double sor_omega = 1.9;
#endif
#ifdef MONITOR_ERROR
double *errors=NULL;
#endif
#ifdef MONITOR_ALLREDUCE
double all_reduce_time = 0;
#endif
#ifdef MONITOR_EXCHANGE_BORDERS
double total_exchange_time = 0.0;   // Total time spent in exchanges
double total_latency = 0.0;         // Total latency
double total_data_transferred = 0.0; // Total data transferred
int num_exchanges = 0;              // Number of exchanges
#endif
#ifdef SKIP_EXCHANGE
size_t skip_exchange;
#endif

/* global variables */
int gridsize[2];
double precision_goal;    /* precision_goal of solution */
int max_iter;     /* maximum number of iterations alowed */
int P; //total number of processes
int P_grid[2]; // process grid dimensions
MPI_Comm grid_comm; //grid communicator
MPI_Status status;
double hx, hy;

/* process specific globals*/
int proc_rank;
```

```
68 double wtime;
69 int proc_coord[2]; // coords of current process in processgrid
70 int proc_top, proc_right, proc_bottom, proc_left; // ranks of neighboring procs
71 // step 7
72 int offset[2] = {0,0};
73 // step 8
74 MPI_Datatype border_type[2];
75
76 /* benchmark related variables */
77 clock_t ticks;      /* number of systemticks */
78 int timer_on = 0;   /* is timer running? */
79
80 /* local grid related variables */
81 double **phi;     /* grid */
82 int **source;     /* TRUE if subgrid element is a source */
83 int dim[2];     /* grid dimensions */
84
85 void Setup_Grid();
86 double Do_Step(int parity);
87 void Solve();
88 void Write_Grid();
89 void Clean_Up();
90 void Debug(char *mesg, int terminate);
91 void start_timer();
92 void resume_timer();
93 void stop_timer();
94 void print_timer();
95
96 void start_timer()
97 {
98     if (!timer_on){
99         MPI_Barrier(grid_comm);
100         ticks = clock();
101         wtime = MPI_Wtime();
102         timer_on = 1;
103     }
104 }
105
106 void resume_timer()
107 {
108     if (!timer_on){
109         ticks = clock()  -  ticks;
110         wtime = MPI_Wtime() - wtime;
111         timer_on = 1;
112     }
113 }
114
115 void stop_timer()
116 {
117     if (timer_on){
118         ticks = clock()  -  ticks;
119         wtime = MPI_Wtime() - wtime;
120         timer_on = 0;
121     }
122 }
123
124 void print_timer()
125 {
126     if (timer_on){
127         stop_timer();
128         printf("(%i) Elapsed Wtime %14.6f s (%5.1f%% CPU)\n", proc_rank, wtime, 100.0 * ticks
    * (1.0 / CLOCKS_PER_SEC) / wtime);
129         resume_timer();
130     }
131     else{
132         printf("(%i) Elapsed Wtime %14.6f s (%5.1f%% CPU)\n", proc_rank, wtime, 100.0 * ticks
    * (1.0 / CLOCKS_PER_SEC) / wtime);
133     }
134 }
135
136 void Debug(char *mesg, int terminate)
137 {
138     if (DEBUG || terminate){
```

```
        printf("%s\n", mesg);
    }
    if (terminate){
        exit(1);
    }
}

void Setup_Proc_Grid(int argc, char **argv){
    int wrap_around[2];
    int reorder;

    Debug("My_MPI_Init",0);

    // num of processes
    MPI_Comm_size(MPI_COMM_WORLD, &P);

    //calculate the number of processes per column and per row for the grid
    if(argc>2){
        P_grid[X_DIR] = atoi(argv[1]);
        P_grid[Y_DIR] = atoi(argv[2]);
        if(P_grid[X_DIR] * P_grid[Y_DIR] != P){
            Debug("ERROR Proces grid dimensions do not match with P ", 1);
        }
        #ifdef SOR
        if (argc>3)
        {
            // get sor from args
            sor_omega = atof(argv[3]);
            printf("Set sor_omega over argv to %1.4f\n", sor_omega);
        }
        #endif
        #ifdef SKIP_EXCHANGE
        if (argc > 4)
        {
            skip_exchange = atoi(argv[4]);
            printf("Set skip_exchange over argv to %zu\n", skip_exchange);
        }
        else{
            skip_exchange = 1;
            printf("Set skip_exchange to default value 1\n");
        }
        #endif
    }
    else{
        Debug("ERROR Wrong parameter input",1);
    }

    // Create process topology (2D grid)
    wrap_around[X_DIR] = 0;
    wrap_around[Y_DIR] = 0;
    reorder = 1; //reorder process ranks

    // create grid_comm
    int ret = MPI_Cart_create(MPI_COMM_WORLD, 2, P_grid, wrap_around, reorder, &grid_comm);
    if (ret != MPI_SUCCESS){
        Debug("ERROR: MPI_Cart_create failed",1);
    }
    //get new rank and cartesian coords of this proc
    MPI_Comm_rank(grid_comm, &proc_rank);
    MPI_Cart_coords(grid_comm, proc_rank, 2, proc_coord);
    printf("(%i) (x,y)=(%i,%i)\n", proc_rank, proc_coord[X_DIR], proc_coord[Y_DIR]);
    //calc neighbours
    // MPI_Cart_shift(grid_comm, Y_DIR, 1, &proc_bottom, &proc_top);
    MPI_Cart_shift(grid_comm, Y_DIR, 1, &proc_top, &proc_bottom);
    MPI_Cart_shift(grid_comm, X_DIR, 1, &proc_left, &proc_right);
    printf("(%i) top %i,  right  %i,  bottom  %i,  left  %i\n", proc_rank, proc_top,
    proc_right, proc_bottom, proc_left);
}

void Setup_Grid()
{
    int x, y, s;
    double source_x, source_y, source_val;
```

```
211     FILE *f;
212
213     Debug("Setup_Subgrid", 0);
214
215     if(proc_rank == 0){
216         f = fopen("input.dat", "r");
217         if (f == NULL){
218             Debug("Error opening input.dat", 1);
219         }
220         fscanf(f, "nx: %i\n", &gridsize[X_DIR]);
221         fscanf(f, "ny: %i\n", &gridsize[Y_DIR]);
222         fscanf(f, "precision goal: %lf\n", &precision_goal);
223         fscanf(f, "max iterations: %i\n", &max_iter);
224     }
225     MPI_Bcast(&gridsize, 2, MPI_INT, 0, grid_comm);
226     MPI_Bcast(&precision_goal, 1, MPI_DOUBLE, 0, grid_comm);
227     MPI_Bcast(&max_iter, 1, MPI_INT, 0, grid_comm);
228     hx = 1 / (double)gridsize[X_DIR];
229     hy = 1 / (double)gridsize[Y_DIR];
230
231     /* Calculate dimensions of local subgrid */ //! We do that later now!
232     // dim[X_DIR] = gridsize[X_DIR] + 2;
233     // dim[Y_DIR] = gridsize[Y_DIR] + 2;
234
235     //! Step 7
236     int upper_offset[2] = {0,0};
237     // Calculate top left corner cordinates of local grid
238     offset[X_DIR] = gridsize[X_DIR] * proc_coord[X_DIR] / P_grid[X_DIR];
239     offset[Y_DIR] = gridsize[Y_DIR] * proc_coord[Y_DIR] / P_grid[Y_DIR];
240     upper_offset[X_DIR] = gridsize[X_DIR] * (proc_coord[X_DIR] + 1) / P_grid[X_DIR];
241     upper_offset[Y_DIR] = gridsize[Y_DIR] * (proc_coord[Y_DIR] + 1) / P_grid[Y_DIR];
242
243     // dimensions of local grid
244     dim[X_DIR] = upper_offset[X_DIR] - offset[X_DIR];
245     dim[Y_DIR] = upper_offset[Y_DIR] - offset[Y_DIR];
246     // Add space for rows/columns of neighboring grid
247     dim[X_DIR] += 2;
248     dim[Y_DIR] += 2;
249     //! Step 7 end
250
251     /* allocate memory */
252     if ((phi = malloc(dim[X_DIR] * sizeof(*phi))) == NULL){
253         Debug("Setup_Subgrid : malloc(phi) failed", 1);
254     }
255     if ((source = malloc(dim[X_DIR] * sizeof(*source))) == NULL){
256         Debug("Setup_Subgrid : malloc(source) failed", 1);
257     }
258     if ((phi[0] = malloc(dim[Y_DIR] * dim[X_DIR] * sizeof(**phi))) == NULL){
259         Debug("Setup_Subgrid : malloc(*phi) failed", 1);
260     }
261     if ((source[0] = malloc(dim[Y_DIR] * dim[X_DIR] * sizeof(**source))) == NULL){
262         Debug("Setup_Subgrid : malloc(*source) failed", 1);
263     }
264     for (x = 1; x < dim[X_DIR]; x++)
265     {
266         phi[x] = phi[0] + x * dim[Y_DIR];
267         source[x] = source[0] + x * dim[Y_DIR];
268     }
269
270     /* set all values to '0' */
271     for (x = 0; x < dim[X_DIR]; x++){
272         for (y = 0; y < dim[Y_DIR]; y++)
273         {
274             phi[x][y] = 0.0;
275             source[x][y] = 0;
276         }
277     }
278     /* put sources in field */
279     do{
280         if (proc_rank==0)
281         {
282             s = fscanf(f, "source: %lf %lf %lf\n", &source_x, &source_y, &source_val);
283         }
```

```
284         MPI_Bcast(&s, 1, MPI_INT, 0, grid_comm);
285         if (s==3){
286             MPI_Bcast(&source_x, 1, MPI_DOUBLE, 0, grid_comm);
287             MPI_Bcast(&source_y, 1, MPI_DOUBLE, 0, grid_comm);
288             MPI_Bcast(&source_val, 1, MPI_DOUBLE, 0, grid_comm);
289             x = source_x * gridsize[X_DIR];
290             y = source_y * gridsize[Y_DIR];
291             x = x + 1 - offset[X_DIR]; // Step 7 --> local grid transform
292             y = y + 1 - offset[Y_DIR]; // Step 7 --> local grid transform
293             if(x > 0 && x < dim[X_DIR] -1 && y > 0 && y < dim[Y_DIR]-1){ // check if in local
    grid
294                 phi[x][y] = source_val;
295                 source[x][y] = 1;
296             }
297         }
298     }
299     while (s==3);
300 
301     if(proc_rank==0){
302         fclose(f);
303     }
304 }
305 
306 void Setup_MPI_Datatypes()
307 {
308     Debug("Setup_MPI_Datatypes",0);
309 
310     // vertical data exchange (Y_Dir)
311     MPI_Type_vector(dim[X_DIR] - 2, 1, dim[Y_DIR], MPI_DOUBLE, &border_type[Y_DIR]);
312     // horizontal data exchange (X_Dir)
313     MPI_Type_vector(dim[Y_DIR] - 2, 1, 1, MPI_DOUBLE, &border_type[X_DIR]);
314 
315     MPI_Type_commit(&border_type[Y_DIR]);
316     MPI_Type_commit(&border_type[X_DIR]);
317 }
318 
319 int Exchange_Borders()
320 {
321     #ifdef MONITOR_EXCHANGE_BORDERS
322     double start_time, latency_start, latency;
323     double data_size_top, data_size_left;
324     double exchange_time;
325 
326     // Measure latency with a small dummy message
327     latency_start = MPI_Wtime();
328     double dummy;
329     MPI_Sendrecv(&dummy, 1, MPI_DOUBLE, proc_top, 0, &dummy, 1, MPI_DOUBLE, proc_bottom, 0,
    grid_comm, &status);
330     latency = MPI_Wtime() - latency_start;
331     total_latency += latency;
332 
333     // Calculate data sizes
334     data_size_top = dim[X_DIR] * sizeof(double);  // Top and bottom rows
335     data_size_left = dim[Y_DIR] * sizeof(double); // Left and right columns
336     double data_transferred = 2 * (data_size_top + data_size_left); // Total data for this
    exchange
337     total_data_transferred += data_transferred;
338     #endif
339 
340     Debug("Exchange_Borders",0);
341     #ifdef MONITOR_EXCHANGE_BORDERS
342     start_time = MPI_Wtime();
343     #endif
344     // top direction
345     MPI_Sendrecv(&phi[1][1], 1, border_type[Y_DIR], proc_top, 0, &phi[1][dim[Y_DIR] - 1], 1,
    border_type[Y_DIR], proc_bottom, 0, grid_comm, &status);
346     // bottom direction
347     MPI_Sendrecv(&phi[1][dim[Y_DIR] - 2], 1, border_type[Y_DIR], proc_bottom, 0, &phi[1][0],
    1, border_type[Y_DIR], proc_top, 0, grid_comm, &status);
348     // left direction
349     MPI_Sendrecv(&phi[1][1], 1, border_type[X_DIR], proc_left, 0, &phi[dim[X_DIR]-1][1], 1,
    border_type[X_DIR], proc_right, 0, grid_comm, &status);
350     // right direction
```

```
351     MPI_Sendrecv(&phi[dim[X_DIR]-2][1], 1, border_type[X_DIR], proc_right, 0, &phi[0][1], 1,
        border_type[X_DIR], proc_left, 0, grid_comm, &status);
352
353     #ifdef MONITOR_EXCHANGE_BORDERS
354     exchange_time = MPI_Wtime() - start_time;
355     total_exchange_time += exchange_time;
356     num_exchanges++;
357     #endif
358     return 1;
359 }
360
361 double Do_Step(int parity)
362 {
363     int x, y;
364     double old_phi, c_ij;
365     double max_err = 0.0;
366
367     #ifdef FAST_DO_STEP_LOOP
368     int start_y;
369     for (x = 1; x < dim[X_DIR] - 1; x++){
370         start_y = ((1 + x + offset[X_DIR] + offset[Y_DIR]) % 2 == parity) ? 1 : 2;
371         for (y = start_y; y < dim[Y_DIR] - 1; y += 2){
372             if (source[x][y] != 1){
373                 old_phi = phi[x][y];
374                 #ifndef SOR
375                 phi[x][y] = (phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1]) *
        0.25;
376                 #endif
377                 #ifdef SOR
378                 c_ij = (phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1] + hx*hy*
        source[x][y]) * 0.25 - phi[x][y];
379                 phi[x][y] += sor_omega*c_ij;
380                 #endif
381                 if (max_err < fabs(old_phi - phi[x][y])){
382                     max_err = fabs(old_phi - phi[x][y]);
383                 }
384             }
385         }
386     }
387     return max_err;
388     #endif
389
390     #ifndef FAST_DO_STEP_LOOP
391     /* calculate interior of grid */
392     for (x = 1; x < dim[X_DIR] - 1; x++){
393         for (y = 1; y < dim[Y_DIR] - 1; y++){
394             if ((x + offset[X_DIR] + y + offset[Y_DIR]) % 2 == parity && source[x][y] != 1){
395                 old_phi = phi[x][y];
396                 #ifndef SOR
397                 phi[x][y] = (phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1]) *
        0.25;
398                 #endif
399                 #ifdef SOR
400                 c_ij = (phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1] + hx*hy*
        source[x][y]) * 0.25 - phi[x][y];
401                 phi[x][y] += sor_omega*c_ij;
402                 #endif
403                 if (max_err < fabs(old_phi - phi[x][y])){
404                     max_err = fabs(old_phi - phi[x][y]);
405                 }
406             }
407         }
408     }
409   return max_err;
410   #endif
411 }
412
413 void Solve()
414 {
415     int count = 0;
416     double delta;
417     double global_delta;
418     double delta1, delta2;
```

```
419
420     Debug("Solve", 0);
421
422     /* give global_delta a higher value then precision_goal */
423     global_delta = 2 * precision_goal;
424
425     while (global_delta > precision_goal && count < max_iter)
426     {
427         Debug("Do_Step 0", 0);
428         delta1 = Do_Step(0);
429         #ifdef SKIP_EXCHANGE
430         if (count % skip_exchange == 0 && Exchange_Borders()) // use short circuit evaluation
431         #endif
432         #ifndef SKIP_EXCHANGE
433         Exchange_Borders();
434         #endif
435         Debug("Do_Step 1", 0);
436         delta2 = Do_Step(1);
437         #ifdef SKIP_EXCHANGE
438         if (count % skip_exchange == 0 && Exchange_Borders())
439         #endif
440         #ifndef SKIP_EXCHANGE
441         Exchange_Borders();
442         #endif
443         delta = max(delta1, delta2);
444         #ifdef MONITOR_ALLREDUCE
445         double time_ = MPI_Wtime();
446         #endif
447         #ifdef ALLREDUCE_COUNT
448         if(count % ALLREDUCE_COUNT == 0){
449             MPI_Allreduce(&delta, &global_delta, 1, MPI_DOUBLE, MPI_MAX, grid_comm);
450         }
451         #endif
452         #ifndef ALLREDUCE_COUNT
453         MPI_Allreduce(&delta, &global_delta, 1, MPI_DOUBLE, MPI_MAX, grid_comm);
454         #endif
455         #ifdef MONITOR_ALLREDUCE
456         all_reduce_time += MPI_Wtime() - time_;
457         #endif
458         #ifdef MONITOR_ERROR
459         if (proc_rank == 0)
460         {
461             errors[count] = global_delta;
462         }
463         #endif
464         count++;
465     }
466
467     printf("(%i) Number of iterations : %i\n", proc_rank, count);
468     #ifdef MONITOR_ALLREDUCE
469     printf("(%i) Allreduce time: %14.6f\n", proc_rank, all_reduce_time);
470     #endif
471     #ifdef MONITOR_EXCHANGE_BORDERS
472     printf("(%i) Exchange time: %14.6f\n", proc_rank, total_exchange_time);
473     #endif
474 }
475
476 double* get_Global_Grid()
477 {
478     Debug("get_Global_Grid", 0);
479     //!! DEBUG only
480     for (size_t i = 0; i < dim[X_DIR]; i++)
481     {
482         for (size_t j = 0; j < dim[Y_DIR]; j++)
483         {
484             phi[i][j] = proc_rank;
485         }
486
487     }
488
489     // only process 0 needs to store all data!
490     double* global_phi = NULL;
491     if (proc_rank == 0) {
```

```
492         global_phi = malloc(gridsize[X_DIR] * gridsize[Y_DIR] * sizeof(double));
493         if (global_phi == NULL) {
494             Debug("get_Global_Grid : malloc(global_phi) failed", 1);
495         }
496     }
497 
498     // copy own part into buffer - flatten!
499     size_t buf_size = (dim[X_DIR] - 2) * (dim[Y_DIR] - 2) * sizeof(double);
500     double* local_phi = malloc(buf_size);
501     int idx = 0;
502     for (int x = 1; x < dim[X_DIR] - 1; x++) {
503         for (int y = 1; y < dim[Y_DIR] - 1; y++) {
504             local_phi[idx++] = phi[x][y];
505         }
506     }
507     printf("I'm proc %d and i have a buffer of size %zu\n", proc_rank, buf_size);
508 
509 
510     // only proc 0 needs sendcounts and displacements for the gatherv operation
511     int* sendcounts = NULL;
512     int* displs = NULL;
513     if (proc_rank == 0) {
514         sendcounts = malloc(P * sizeof(int));
515         displs = malloc(P * sizeof(int));
516 
517         // size and offset of different subgrids
518         //! Note that this only works if every process has the same subgrid
519         if (gridsize[X_DIR] % P_grid[X_DIR] != 0 || gridsize[Y_DIR] % P_grid[Y_DIR] != 0)
520         {
521             Debug("!!!A grid dimension is not a multiple of the P_grid in this direction!", 1)
    ;
522         }
523 
524         int subgrid_width = gridsize[X_DIR] / P_grid[X_DIR];
525         int subgrid_height = gridsize[Y_DIR] / P_grid[Y_DIR];
526         for (int px = 0; px < P_grid[X_DIR]; px++) {
527             for (int py = 0; py < P_grid[Y_DIR]; py++) {
528                 int rank = px * P_grid[Y_DIR] + py;
529                 sendcounts[rank] = subgrid_width * subgrid_height;
530                 displs[rank] = (px * subgrid_width * gridsize[Y_DIR]) + (py * subgrid_height);
531             }
532         }
533     }
534     Debug("get_Global_Grid : MPI_Gatherv", 0);
535     //! TODO this Gatherv does something wrong - all local grids are alright!!!
536     MPI_Gatherv(local_phi, (dim[X_DIR] - 2) * (dim[Y_DIR] - 2), MPI_DOUBLE, global_phi,
    sendcounts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
537 
538     free(local_phi);
539     if (proc_rank == 0) {
540         free(sendcounts);
541         free(displs);
542     }
543 
544     return global_phi;
545 }
546 
547 void Write_Grid_global(){
548     int x, y;
549     FILE *f;
550     char filename[40]; //seems danagerous to use a static buffer but let's go with the steps
551     sprintf(filename, "output_MPI_global_%i.dat", proc_rank);
552     if ((f = fopen(filename, "w")) == NULL){
553         Debug("Write_Grid : fopen failed", 1);
554     }
555 
556     Debug("Write_Grid", 0);
557 
558     for (x = 1; x < dim[X_DIR]-1; x++){
559         for (y = 1; y < dim[Y_DIR]-1; y++){
560             int x_glob = x + offset[X_DIR];
561             int y_glob = y + offset[Y_DIR];
562             fprintf(f, "%i %i %f\n", x_glob, y_glob, phi[x][y]);
```

```
563         }
564     }
565     fclose(f);
566 }
567 
568 void Write_Grid()
569 {
570     double* global_phi = get_Global_Grid();
571     if(proc_rank != 0){
572         assert (global_phi == NULL);
573         return;
574     }
575     int x, y;
576     FILE *f;
577     char filename[40]; //seems danagerous to use a static buffer but let's go with the steps
578     sprintf(filename, "output_MPI%i.dat", proc_rank);
579     if ((f = fopen(filename, "w")) == NULL){
580         Debug("Write_Grid : fopen failed", 1);
581     }
582 
583     Debug("Write_Grid", 0);
584 
585     for (x = 0; x < gridsize[X_DIR]; x++){
586         for (y = 0; y < gridsize[Y_DIR]; y++){
587             fprintf(f, "%i %i %f\n", x+1, y+1, global_phi[x*gridsize[Y_DIR] + y]);
588         }
589     }
590     fclose(f);
591     free(global_phi);
592 }
593 
594 void Clean_Up()
595 {
596     Debug("Clean_Up", 0);
597 
598     free(phi[0]);
599     free(phi);
600     free(source[0]);
601     free(source);
602     #ifdef MONITOR_ERROR
603     free(errors);
604     #endif
605 }
606 void setup_error_monitor(){
607     if (proc_rank != 0)
608     {
609         return;
610     }
611 
612     errors = malloc(sizeof(double)*max_iter);
613 }
614 void write_errors(){
615     if(proc_rank != 0){
616         return;
617     }
618     FILE *f;
619     char filename[40]; //seems danagerous to use a static buffer but let's go with the steps
620     sprintf(filename, "errors_MPI.dat");
621     if ((f = fopen(filename, "w")) == NULL){
622         Debug("Write_Errors : fopen failed", 1);
623     }
624 
625     Debug("Write_Errors", 0);
626 
627     for (size_t i = 0; i < max_iter; ++i)
628     {
629         fprintf(f, "%f\n", errors[i]);
630     }
631     fclose(f);
632 }
633 
634 void Print_Aggregated_Metrics()
635 {
```

```
    #ifdef MONITOR_EXCHANGE_BORDERS
    if (num_exchanges > 0) {
        double avg_exchange_time = total_exchange_time / num_exchanges;
        double avg_latency = total_latency / num_exchanges;
        double avg_bandwidth = total_data_transferred / total_exchange_time;

        printf("\n--- Aggregated Metrics ---\n");
        printf("Total Exchanges: %d\n", num_exchanges);
        printf("Total Data Transferred: %.2f bytes\n", total_data_transferred);
        printf("Total Exchange Time: %.9f s\n", total_exchange_time);
        printf("Average Exchange Time per Call: %.9f s\n", avg_exchange_time);
        printf("Average Latency per Call: %.9f s\n", avg_latency);
        printf("Average Bandwidth: %.2f bytes/s\n", avg_bandwidth);
    } else {
        printf("No exchanges recorded.\n");
    }
    #endif
}

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    Setup_Proc_Grid(argc,argv); // was earlier MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank);
    start_timer();

    Setup_Grid();
    Setup_MPI_Datatypes();

    #ifdef SOR
    if (proc_rank == 0)
    {
        printf("SOR using omega: %.5f\n", sor_omega);
    }
    #endif
    #ifdef MONITOR_ERROR
    setup_error_monitor();
    #endif

    Solve();
    #ifdef MONITOR_ERROR
    write_errors();
    #endif
    // Write_Grid();
    Write_Grid_global();
    Print_Aggregated_Metrics();
    print_timer();

    Clean_Up();
    MPI_Finalize();
    return 0;
}
```

As an example for a launch-script for DB I'll present the following script used for subsubsection 1.2.3 bellow. For a exhaustive reference of all scripts / files used go to GitHub.

```
#!/bin/bash
#SBATCH --job-name="scaling_123"
#SBATCH --time=00:03:00
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=1
#SBATCH --partition=compute
#SBATCH --mem=2GB  # Increased memory
#SBATCH --account=Education-EEMCS-Courses-WI4049TU

if [ -z "$1" ] || [ -z "$2" ]; then
    echo "Usage: $0 <topology_x> <topology_y>"
    exit 1
fi

# Move to the src directory
cd ../src || { echo "Error: ../src directory not found"; exit 1; }
basefolder="123/${1}_${2}"
mkdir -p ../scripts/output/$basefolder || { echo "Error creating output directory"; exit 1; }

```

```
# Define maximum iterations and grid sizes
maxiters=("500" "1000" "2000")
#grids=("50 50" "100 100" "200 200")
grids=("1600 1600" "3200 3200")

for grid in "${grids[@]}"; do
    nx=$(echo $grid | cut -d' ' -f1)
    ny=$(echo $grid | cut -d' ' -f2)

    mkdir -p ../scripts/output/$basefolder/${nx}x${ny} || { echo "Error creating grid
    directory"; exit 1; }

    for maxiter in "${maxiters[@]}"; do
        python3 -c "
import util
util.update_input_file(nx=$nx, ny=$ny, precision_goal=0.000000000000001, max_iter=$maxiter)
" || { echo "Python script failed for nx=$nx, ny=$ny, max_iter=$maxiter"; exit 1; }

        echo "Running with nx=$nx, ny=$ny, maxiter=$maxiter"
        srun ./MPI_Poisson.out $1 $2 1.95 > ../scripts/output/$basefolder/${nx}x${ny}/
    maxiter_${maxiter}.txt || {
            echo "srun failed for nx=$nx, ny=$ny, max_iter=$maxiter"; exit 1;
        }
        echo "Finished maxiter=$maxiter for grid ${nx}x${ny}"
    done
done

python3 -c "
import util
util.reset_input_file()
" || { echo "Error resetting input file"; exit 1; }

echo "Job completed successfully."
```

Appendix - Finite elements simulation

The following code was used to solve the tasks in section 2:

```c
/*
 * MPI_Fempois.c
 * 2D Poisson equation solver with MPI and FEM
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "mpi.h"

#define DEBUG 0

#define TYPE_GHOST 1
#define TYPE_SOURCE 2

#define MAXCOL 20

typedef struct
{
  int type;
  double x, y;
}
Vertex;

typedef int Element[3];

typedef struct
{
  int Ncol;
  int *col;
  double *val;
}
Matrixrow;

/* globals related to timing*/
double total_time = 0.0;
double exchange_time_neighbors = 0.0;
double exchange_time_global = 0.0;
double compute_time = 0.0;

/* global variables */
double precision_goal;    /* precision_goal of solution */
int max_iter;     /* maximum number of iterations alowed */
int P;        /* total number of processes */
int P_grid[2];      /* processgrid dimensions */
MPI_Comm grid_comm;   /* grid COMMUNICATOR */
MPI_Status status;

/* benchmark related variables */
clock_t ticks;      /* number of systemticks */
double wtime;     /* wallclock time */
int timer_on = 0;   /* is timer running? */

/* local process related variables */
int proc_rank;      /* rank of current process */
int proc_coord[2];    /* coordinates of current procces in processgrid */
int N_neighb;     /* Number of neighbouring processes */
int *proc_neighb;   /* ranks of neighbouring processes */
MPI_Datatype *send_type;  /* MPI Datatypes for sending */
MPI_Datatype *recv_type;  /* MPI Datatypes for receiving */

/* local grid related variables */
Vertex *vert;     /* vertices */
double *phi;      /* vertex values */
int N_vert;     /* number of vertices */
Matrixrow *A;     /* matrix A */

```

```
69  void Setup_Proc_Grid();
70  void Setup_Grid();
71  void Build_ElMatrix(Element el);
72  void Sort_MPI_Datatypes();
73  void Setup_MPI_Datatypes(FILE *f);
74  void Exchange_Borders(double *vect);
75  void Solve();
76  void Write_Grid();
77  void Clean_Up();
78  void Debug(char *mesg, int terminate);
79  void start_timer();
80  void resume_timer();
81  void stop_timer();
82  void print_timer();
83
84  void start_timer()
85  {
86    if (!timer_on)
87    {
88      MPI_Barrier(MPI_COMM_WORLD);
89      ticks = clock();
90      wtime = MPI_Wtime();
91      timer_on = 1;
92    }
93  }
94
95  void resume_timer()
96  {
97    if (!timer_on)
98    {
99      ticks = clock()-ticks;
100     wtime = MPI_Wtime()-wtime;
101     timer_on = 1;
102   }
103 }
104
105 void stop_timer()
106 {
107   if (timer_on)
108   {
109     ticks = clock() - ticks;
110     wtime = MPI_Wtime() - wtime;
111     timer_on = 0;
112   }
113 }
114
115 void print_timer()
116 {
117   if (timer_on)
118   {
119     stop_timer();
120     printf("(%i) Elapsed Wtime: %14.6f s (%5.1f%% CPU)\n",
121      proc_rank, wtime, 100.0 * ticks * (1.0 / CLOCKS_PER_SEC) / wtime);
122     resume_timer();
123   }
124   else
125     printf("(%i) Elapsed Wtime: %14.6f s (%5.1f%% CPU)\n",
126      proc_rank, wtime, 100.0 * ticks * (1.0 / CLOCKS_PER_SEC) / wtime);
127 }
128
129 void Debug(char *mesg, int terminate)
130 {
131   if (DEBUG || terminate)
132     printf("(%i) %s\n", proc_rank, mesg);
133   if (terminate)
134   {
135     MPI_Abort(MPI_COMM_WORLD, 1);
136     exit(1);
137   }
138 }
139
140 void Setup_Proc_Grid()
141 {
```

```
142   FILE *f = NULL;
143   char filename[25];
144   int i;
145   int N_nodes = 0, N_edges = 0;
146   int *index, *edges, reorder;
147
148   MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank);
149   Debug("My_MPI_Init", 0);
150
151   /* Retrieve the number of processes and current process rank */
152   MPI_Comm_size(MPI_COMM_WORLD, &P);
153
154   /* Create process topology (Graph) */
155   if (proc_rank == 0)
156   {
157     sprintf(filename, "mapping%i.dat", P);
158     if ((f = fopen(filename, "r")) == NULL)
159       Debug("My_MPI_Init : Can't open mapping inputfile", 1);
160
161     /* after reading N_nodes, a line is skipped */
162     fscanf(f, "N_proc : %i\n%*[^\n]\n", &N_nodes);
163     if (N_nodes != P)
164       Debug("My_MPI_Init : Mismatch of number of processes in mapping inputfile", 1);
165   }
166   else
167     N_nodes = P;
168
169   if ((index = malloc(N_nodes * sizeof(int))) == NULL)
170       Debug("My_MPI_Init : malloc(index) failed", 1);
171
172   if (proc_rank == 0)
173   {
174     for (i = 0; i < N_nodes; i++)
175       fscanf(f, "%i\n", &index[i]);
176   }
177
178   MPI_Bcast(index, N_nodes, MPI_INT, 0, MPI_COMM_WORLD);
179
180   N_edges = index[N_nodes - 1];
181   if (N_edges>0)
182   {
183     if ((edges = malloc(N_edges * sizeof(int))) == NULL)
184       Debug("My_MPI_Init : malloc(edges) failed", 1);
185   }
186   else
187     edges = index; /* this is actually nonsense,
188                       but 'edges' needs to be a non-null pointer */
189
190   if (proc_rank == 0)
191   {
192     fscanf(f, "%*[^\n]\n");   /* skip a line of the file */
193     for (i = 0; i < N_edges; i++)
194       fscanf(f, "%i\n", &edges[i]);
195
196     fclose(f);
197   }
198
199   MPI_Bcast(edges, N_edges, MPI_INT, 0, MPI_COMM_WORLD);
200
201   reorder = 1;
202   MPI_Graph_create(MPI_COMM_WORLD, N_nodes, index, edges, reorder, &grid_comm);
203
204   /* Retrieve new rank of this process */
205   MPI_Comm_rank(grid_comm, &proc_rank);
206
207   if (N_edges>0)
208     free(edges);
209   free(index);
210 }
211
212 void Setup_Grid()
213 {
214   int i, j, v;
```

```
215   Element element;
216   int N_elm;
217   char filename[25];
218   FILE *f;
219
220   Debug("Setup_Grid", 0);
221
222   /* read general parameters (precision/max_iter) */
223   if (proc_rank==0)
224   {
225     if ((f = fopen("input.dat", "r")) == NULL)
226       Debug("Setup_Grid : Can't open input.dat", 1);
227     fscanf(f, "precision goal: %lf\n", &precision_goal);
228     fscanf(f, "max iterations: %i", &max_iter);
229     fclose(f);
230   }
231   MPI_Bcast(&precision_goal, 1, MPI_DOUBLE, 0, grid_comm);
232   MPI_Bcast(&max_iter, 1, MPI_INT, 0, grid_comm);
233
234   /* read process specific data */
235   sprintf(filename, "input%i-%i.dat", P, proc_rank);
236   if ((f = fopen(filename, "r")) == NULL)
237     Debug("Setup_Grid : Can't open data inputfile", 1);
238   fscanf(f, "N_vert: %i\n%*[^\n]\n", &N_vert);
239
240   /* allocate memory for phi and A */
241   if ((vert = malloc(N_vert * sizeof(Vertex))) == NULL)
242     Debug("Setup_Grid : malloc(vert) failed", 1);
243   if ((phi = malloc(N_vert * sizeof(double))) == NULL)
244     Debug("Setup_Grid : malloc(phi) failed", 1);
245
246   if ((A = malloc(N_vert * sizeof(*A))) == NULL)
247     Debug("Setup_Grid : malloc(*A) failed", 1);
248   for (i=0; i<N_vert; i++)
249   {
250     if ((A[i].col=malloc(MAXCOL*sizeof(int)))==NULL)
251       Debug("Setup_Grid : malloc(A.col) failed", 1);
252     if ((A[i].val=malloc(MAXCOL*sizeof(double)))==NULL)
253       Debug("Setup_Grid : malloc(A.val) failed", 1);
254   }
255
256   /* init matrix rows of A */
257   for (i = 0; i < N_vert; i++)
258       A[i].Ncol = 0;
259
260   /* Read all values */
261   for (i = 0; i < N_vert; i++)
262   {
263     fscanf(f, "%i", &v);
264     fscanf(f, "%lf %lf %i %lf\n", &vert[v].x, &vert[v].y,
265      &vert[v].type, &phi[v]);
266   }
267
268   /* build matrix from elements */
269   fscanf(f, "N_elm: %i\n%*[^\n]\n", &N_elm);
270   for (i = 0; i < N_elm; i++)
271   {
272     fscanf(f, "%*i");  /* we are not interested in the element-id */
273     for (j = 0; j < 3; j++)
274     {
275       fscanf(f, "%i", &v);
276       element[j] = v;
277     }
278     fscanf(f, "\n");
279     Build_ElMatrix(element);
280   }
281
282   Setup_MPI_Datatypes(f);
283
284   fclose(f);
285 }
286
287 void Add_To_Matrix(int i, int j, double a)
```

```
{
  int k;
  k=0;

  while ( (k<A[i].Ncol) && (A[i].col[k]!=j) )
    k++;
  if (k<A[i].Ncol)
    A[i].val[k]+=a;
  else
  {
    if (A[i].Ncol>=MAXCOL)
      Debug("Add_To_Matrix : MAXCOL exceeded", 1);
    A[i].val[A[i].Ncol]=a;
    A[i].col[A[i].Ncol]=j;
    A[i].Ncol++;
  }
}

void Build_ElMatrix(Element el)
{
  int i, j;
  double e[3][2];
  double s[3][3];
  double det;

  e[0][0] = vert[el[1]].y - vert[el[2]].y;  /* y1-y2 */
  e[1][0] = vert[el[2]].y - vert[el[0]].y;  /* y2-y0 */
  e[2][0] = vert[el[0]].y - vert[el[1]].y;  /* y0-y1 */
  e[0][1] = vert[el[2]].x - vert[el[1]].x;  /* x2-x1 */
  e[1][1] = vert[el[0]].x - vert[el[2]].x;  /* x0-x2 */
  e[2][1] = vert[el[1]].x - vert[el[0]].x;  /* x1-x0 */

  det = e[2][0] * e[0][1] - e[2][1] * e[0][0];
  if (det == 0.0)
    Debug("One of the elements has a zero surface", 1);

  det = fabs(2 * det);

  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
      s[i][j] = (e[i][0] * e[j][0] + e[i][1] * e[j][1]) / det;

  for (i = 0; i < 3; i++)
    if (!((vert[el[i]].type & TYPE_GHOST) |
    (vert[el[i]].type & TYPE_SOURCE)))
      for (j = 0; j < 3; j++)
        Add_To_Matrix(el[i],el[j],s[i][j]);
}

void Sort_MPI_Datatypes()
{
  int i, j;
  MPI_Datatype data2;
  int proc2;

  for (i=0;i<N_neighb-1;i++)
    for (j=i+1;j<N_neighb;j++)
      if (proc_neighb[j]<proc_neighb[i])
      {
        proc2 = proc_neighb[i];
        proc_neighb[i] = proc_neighb[j];
        proc_neighb[j] = proc2;
        data2 = send_type[i];
        send_type[i] = send_type[j];
        send_type[j] = data2;
        data2 = recv_type[i];
        recv_type[i] = recv_type[j];
        recv_type[j] = data2;
      }
}

void Setup_MPI_Datatypes(FILE * f)
{
```

```
  int i, s;
  int count;
  int *indices;
  int *blocklens;

  Debug("Setup_MPI_Datatypes", 0);

  fscanf(f, "Neighbours: %i\n", &N_neighb);

  /* allocate memory */

  if (N_neighb>0)
  {
    if ((proc_neighb = malloc(N_neighb * sizeof(int))) == NULL)
        Debug("Setup_MPI_Datatypes: malloc(proc_neighb) failed", 1);
    if ((send_type = malloc(N_neighb * sizeof(MPI_Datatype))) == NULL)
      Debug("Setup_MPI_Datatypes: malloc(send_type) failed", 1);
    if ((recv_type = malloc(N_neighb * sizeof(MPI_Datatype))) == NULL)
      Debug("Setup_MPI_Datatypes: malloc(recv_type) failed", 1);
  }
  else
  {
    proc_neighb = NULL;
    send_type = NULL;
    recv_type = NULL;
  }

  if ((indices = malloc(N_vert * sizeof(int))) == NULL)
      Debug("Setup_MPI_Datatypes: malloc(indices) failed", 1);
  if ((blocklens = malloc(N_vert * sizeof(int))) == NULL)
      Debug("Setup_MPI_Datatypes: malloc(blocklens) failed", 1);

  for (i = 0; i < N_vert; i++)
    blocklens[i] = 1;

  /* read vertices per neighbour */
  for (i = 0; i < N_neighb; i++)
  {
    fscanf(f, "from %i :", &proc_neighb[i]);
    s = 1;
    count = 0;
    while (s == 1)
    {
      s = fscanf(f, "%i", &indices[count]);
      if ((s == 1) && !(vert[indices[count]].type & TYPE_SOURCE))
  count++;
    }
    fscanf(f, "\n");
    MPI_Type_indexed(count, blocklens, indices, MPI_DOUBLE, &recv_type[i]);
    MPI_Type_commit(&recv_type[i]);

    fscanf(f, "to %i :", &proc_neighb[i]);
    s = 1;
    count = 0;
    while (s == 1)
    {
      s = fscanf(f, "%i", &indices[count]);
      if ((s == 1) && !(vert[indices[count]].type & TYPE_SOURCE))
  count++;
    }
    fscanf(f, "\n");
    MPI_Type_indexed(count, blocklens, indices, MPI_DOUBLE, &send_type[i]);
    MPI_Type_commit(&send_type[i]);
  }

  Sort_MPI_Datatypes();

  free(blocklens);
  free(indices);
}
```

```
434
435
436 void Exchange_Borders(double *vect)
437 {
438     // Please finsih this part to realize the purpose of data communication among neighboring
    processors. (Tip: the function "MPI_Sendrecv" needs to be used here.)
439     double temp = MPI_Wtime();
440     for (size_t i = 0; i < N_neighb; ++i)
441     {
442         MPI_Sendrecv(vect, 1, send_type[i], proc_neighb[i], 0, //send
443                      vect, 1, recv_type[i], proc_neighb[i], 0, //recv
444                      grid_comm, &status);
445     }
446     exchange_time_neighbors += MPI_Wtime() - temp;
447 }
448
449 void Solve()
450 {
451     // Formating is a mess here - but the most striking thing is that there are no curly
    braces for the if / for statements.
452   int count = 0;
453   int i, j;
454   double *r, *p, *q;
455   double a, b, r1, r2 = 1;
456
457   double sub;
458
459   Debug("Solve", 0);
460
461   if ((r = malloc(N_vert * sizeof(double))) == NULL)
462       Debug("Solve : malloc(r) failed", 1);
463   if ((p = malloc(N_vert * sizeof(double))) == NULL)
464       Debug("Solve : malloc(p) failed", 1);
465   if ((q = malloc(N_vert * sizeof(double))) == NULL)
466       Debug("Solve : malloc(q) failed", 1);
467
468   /* Implementation of the CG algorithm : */
469
470   Exchange_Borders(phi);
471
472   /* r = b-Ax */
473   for (i = 0; i < N_vert; i++)
474   {
475     r[i] = 0.0;
476     for (j = 0; j < A[i].Ncol; j++)
477       r[i] -= A[i].val[j] * phi[A[i].col[j]];
478   }
479
480   r1 = 2 * precision_goal;
481   while ((count < max_iter) && (r1 > precision_goal))
482   {
483     if(proc_rank==0){
484       printf("<%i> Precision: %.6f\n", count, r1);
485     }
486     /* r1 = r' * r */
487     sub = 0.0;
488     for (i = 0; i < N_vert; i++)
489       if (!(vert[i].type & TYPE_GHOST))
490   sub += r[i] * r[i];
491     // measure gloabl com
492     double temp = MPI_Wtime();
493     MPI_Allreduce(&sub, &r1, 1, MPI_DOUBLE, MPI_SUM, grid_comm);
494     exchange_time_global += MPI_Wtime() - temp;
495
496     if (count == 0)
497     {
498       /* p = r */
499       for (i = 0; i < N_vert; i++)
500   p[i] = r[i];
501     }
502     else
503     {
504       b = r1 / r2;
```

```
505
506       /* p = r + b*p */
507       for (i = 0; i < N_vert; i++)
508   p[i] = r[i] + b * p[i];
509     }
510     Exchange_Borders(p);
511
512     /* q = A * p */
513     for (i = 0; i < N_vert; i++)
514     {
515       q[i] = 0;
516       for (j = 0; j < A[i].Ncol; j++)
517         q[i] += A[i].val[j] * p[A[i].col[j]];
518     }
519
520     /* a = r1 / (p' * q) */
521     sub = 0.0;
522     for (i = 0; i < N_vert; i++)
523       if (!(vert[i].type & TYPE_GHOST))
524   sub += p[i] * q[i];
525     temp = MPI_Wtime();
526     MPI_Allreduce(&sub, &a, 1, MPI_DOUBLE, MPI_SUM, grid_comm);
527     exchange_time_global += MPI_Wtime() - temp;
528     a = r1 / a;
529
530     /* x = x + a*p */
531     for (i = 0; i < N_vert; i++)
532       phi[i] += a * p[i];
533
534     /* r = r - a*q */
535     for (i = 0; i < N_vert; i++)
536       r[i] -= a * q[i];
537
538     r2 = r1;
539
540     count++;
541   }
542   free(q);
543   free(p);
544   free(r);
545
546   if (proc_rank == 0)
547     printf("Number of iterations : %i\n", count);
548 }
549
550 void Write_Grid()
551 {
552   int i;
553   char filename[25];
554   FILE *f;
555
556   Debug("Write_Grid", 0);
557
558   sprintf(filename, "output%i-%i.dat", P, proc_rank);
559   if ((f = fopen(filename, "w")) == NULL)
560     Debug("Write_Grid : Can't open data outputfile", 1);
561
562   for (i = 0; i < N_vert; i++)
563     if (!(vert[i].type & TYPE_GHOST))
564       fprintf(f, "%f %f %f\n", vert[i].x, vert[i].y, phi[i]);
565
566   fclose(f);
567 }
568
569 void Clean_Up()
570 {
571   int i;
572   Debug("Clean_Up", 0);
573
574   if (N_neighb>0)
575   {
576     free(recv_type);
577     free(send_type);
```

```
    free(proc_neighb);
  }

  for (i=0;i<N_vert;i++)
  {
    free(A[i].col);
    free(A[i].val);
  }
  free(A);
  free(vert);
  free(phi);
}
void report_times()
{
  printf("(%i) - Compute time: %f\n", proc_rank, compute_time);
  printf("(%i) - Exchange time (neighbors): %f\n", proc_rank, exchange_time_neighbors);
  printf("(%i) - Exchange time (global): %f\n", proc_rank, exchange_time_global);
  double total_ex = exchange_time_global + exchange_time_neighbors;
  double ratio = compute_time / total_ex;
  printf("(%i) - Exchange time (total): %f\n", proc_rank, total_ex);
  printf("(%i) - Exchange time - comp ratio: %f\n", proc_rank, ratio);
  printf("(%i) - Sum of times (compute + exchange (global & local)): %f\n", proc_rank,
    compute_time + exchange_time_neighbors + exchange_time_global);
  printf("(%i) - Total time: %f\n", proc_rank, total_time);
}

int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  total_time = MPI_Wtime();

  start_timer();

  Setup_Proc_Grid();

  Setup_Grid();
  double temp = MPI_Wtime();
  Solve();
  compute_time = MPI_Wtime() - temp - exchange_time_global;

  Write_Grid();

  Clean_Up();

  print_timer();

  Debug("MPI_Finalize", 0);

  total_time = MPI_Wtime() - total_time;
  MPI_Finalize();
  report_times();
  return 0;
}
```

An example of a sbatch script used for the timing of the finite elements simulation in subsection 2.2 is shown bellow:

```
#!/bin/bash
#SBATCH --job-name="time_fem"
#SBATCH --time=00:03:00
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=1
#SBATCH --partition=compute
#SBATCH --mem=2GB  # Increased memory
#SBATCH --account=Education-EEMCS-Courses-WI4049TU

if [ -z "$1" ] || [ -z "$2" ]; then
    echo "Usage: $0 <topology_x> <topology_y>"
    exit 1
fi

# Move to the src directory
cd ../src || { echo "Error: ../src directory not found"; exit 1; }
basefolder="22/${1}_${2}"
```

```
mkdir -p ../scripts/output/$basefolder || { echo "Error creating output directory"; exit 1; }

grids=("100 100" "200 200" "400 400")

for grid in "${grids[@]}"; do
    nx=$(echo $grid | cut -d' ' -f1)
    ny=$(echo $grid | cut -d' ' -f2)

    mkdir -p ../scripts/output/$basefolder/${nx}x${ny} || { echo "Error creating grid
    directory"; exit 1; }

    echo "Running pre-script: ./GridDist.out"
    ./GridDist.out $1 $2 $nx $ny || { echo "Pre-script failed"; exit 1; }
    echo "Pre-script completed successfully."

    echo "Running with nx=$nx, ny=$ny, maxiter=5000"
    srun ./MPI_Fempois.out > ../scripts/output/$basefolder/${nx}x${ny}/output.txt || {
        echo "srun failed for nx=$nx, ny=$ny, max_iter=5000"; exit 1;
        }
    echo "Finished maxiter=5000 for grid ${nx}x${ny}"
done

echo "Job completed successfully."
```

Appendix - Eigenvalue solution by Power Method on GPU

The code for the CUDA assignment was mostly given and adapted in the main function and some kernels as detailed in section 3. The full code is given bellow:

```
%%cuda

// the subroutine for GPU code can be found in several separated text file from the
    Brightspace.
// You can add these subroutines to this main code.
////////////////////////////////////////////


#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "cuda.h"
#include <stdarg.h> // For variable argument handling
#include <assert.h>


const int BLOCK_SIZE =32;  // number of threads per block
int PRINTLEVEL = 1;
bool GLOBAL_MEM = true;
bool WITHOUT_MEMCPY = false;

// Input Array Variables
float* h_MatA = NULL;
float* d_MatA = NULL;

// Output Array
float* h_VecV = NULL;
float* d_VecV = NULL;
float* h_VecW = NULL;
float* d_VecW = NULL;
float* h_NormW = NULL;
float* d_NormW = NULL;

// Variables to change
int GlobalSize = 5000;          // this is the dimension of the matrix, GlobalSize*GlobalSize
int BlockSize = 32;             // number of threads in each block
const float EPS = 0.000005;     // tolerence of the error
int max_iteration = 100;        // the maximum iteration steps


// Functions
void Cleanup(void);
void InitOne(float*, int);
void UploadArray(float*, int);
float CPUReduce(float*, int);
void  Arguments(int, char**);
void checkCardVersion(void);
void CleanGPU();

// Kernels
__global__ void Av_Product(float* g_MatA, float* g_VecV, float* g_VecW, int N);
__global__ void Av_ProductGlobal(float* g_MatA, float* g_VecV, float* g_VecW, int N);
__global__ void FindNormW(float* g_VecW, float * g_NormW, int N);
__global__ void NormalizeW(float* g_VecW, float* g_NormW, float* g_VecV, int N);
__global__ void ComputeLamda( float* g_VecV,float* g_VecW, float * g_Lamda,int N);
__global__ void Av_Product_1D(const float* A, const float* v, float* w, int N);

//dummy for args!
void ParseArguments(int argc, char** argv) {
    // If you want to parse some arguments
}

void CPU_AvProduct()
{
  int N = GlobalSize;
  int matIndex =0;
```

```
    for(int i=0;i<N;i++)
  {
    h_VecW[i] = 0;
    for(int j=0;j<N;j++)
    {
      matIndex = i*N + j;
      h_VecW[i] += h_MatA[matIndex] * h_VecV[j];

    }
  }
}

void CPU_NormalizeW()
{
  int N = GlobalSize;
  float normW=0;
  for(int i=0;i<N;i++)
    normW += h_VecW[i] * h_VecW[i];

  normW = sqrt(normW);
  for(int i=0;i<N;i++)
    h_VecV[i] = h_VecW[i]/normW;
}

float CPU_ComputeLamda()
{
  int N = GlobalSize;
  float lamda =0;
  for(int i=0;i<N;i++)
    lamda += h_VecV[i] * h_VecW[i];

  return lamda;
}

void RunCPUPowerMethod()
{
  printf("************************************\n");
  float oldLamda =0;
  float lamda=0;

  //AvProduct
  CPU_AvProduct();

  //power loop
  for (int i=0;i<max_iteration;i++)
  {
    CPU_NormalizeW();
    CPU_AvProduct();
    lamda= CPU_ComputeLamda();
    printf("CPU lamda at %d: %f \n", i, lamda);
    // If residual is lass than epsilon break
    if(abs(oldLamda - lamda) < EPS)
      break;
    oldLamda = lamda;

  }
  printf("************************************\n");

}



void printfD(int threshold, const char* format, ...) {
    if (PRINTLEVEL >= threshold) {
        va_list args;
        va_start(args, format);
        vprintf(format, args); // Use vprintf to handle the variable arguments
        va_end(args);
    }
}

typedef struct {
    struct timespec start_time;
```

```
    struct timespec end_time;
    double elapsed; // Cumulative elapsed time
    int running;    // Flag to track if the timer is running
} Timer;

// Function to initialize the timer
void timer_init(Timer* timer) {
    timer->elapsed = 0.0;
    timer->running = 0;
}

// Function to start the timer
void timer_start(Timer* timer) {
    if (!timer->running) {
        clock_gettime(CLOCK_REALTIME, &timer->start_time);
        timer->running = 1;
    }
}

// Function to stop the timer
void timer_stop(Timer* timer) {
    if (timer->running) {
        clock_gettime(CLOCK_REALTIME, &timer->end_time);
        double start_sec = timer->start_time.tv_sec + timer->start_time.tv_nsec / 1e9;
        double end_sec = timer->end_time.tv_sec + timer->end_time.tv_nsec / 1e9;
        timer->elapsed += end_sec - start_sec;
        timer->running = 0;
    }
}

// Function to get the total elapsed time in seconds
double timer_get_elapsed(Timer* timer) {
    if (timer->running) {
        struct timespec now;
        clock_gettime(CLOCK_REALTIME, &now);
        double start_sec = timer->start_time.tv_sec + timer->start_time.tv_nsec / 1e9;
        double now_sec = now.tv_sec + now.tv_nsec / 1e9;
        return timer->elapsed + (now_sec - start_sec);
    }
    return timer->elapsed;
}


double RunGPUPowerMethod(int N){
    double runtime;
    // timers
    Timer total_time_GPU;
    Timer time_GPU_mem;
    Timer time_GPU_kernels;

    timer_init(&total_time_GPU);
    timer_init(&time_GPU_mem);
    timer_init(&time_GPU_kernels);

    printfD(0, "Power method in GPU starts\n");
    checkCardVersion();

    size_t vec_size = N * sizeof(float);
    size_t mat_size = N * N * sizeof(float);
    size_t norm_size = sizeof(float);


    timer_start(&total_time_GPU);
    // Set the kernel arguments
    int threadsPerBlock = BlockSize;
    int sharedMemSize = threadsPerBlock * threadsPerBlock * sizeof(float); // in per block,
    the memory is shared
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    timer_start(&total_time_GPU);
    // Allocate matrix and vectors in device memory
    cudaMalloc((void**)&d_MatA, mat_size);
    cudaMalloc((void**)&d_VecV, vec_size);
```

```
213     cudaMalloc((void**)&d_VecW, vec_size); // This vector is only used by the device
214     cudaMalloc((void**)&d_NormW, norm_size);
215
216
217     //Copy from host memory to device memory
218     timer_start(&time_GPU_mem);
219     cudaMemcpy(d_MatA, h_MatA, mat_size, cudaMemcpyHostToDevice);
220     cudaMemcpy(d_VecV, h_VecV, vec_size, cudaMemcpyHostToDevice);
221     timer_stop(&time_GPU_mem);
222
223    //Power method loops
224     float OldLamda = 0;
225     float Lambda = 0;
226     h_NormW = 0;
227
228     if(GLOBAL_MEM == true){
229       Av_ProductGlobal<<<blocksPerGrid, threadsPerBlock, sharedMemSize>>>(d_MatA, d_VecV,
      d_VecW, N); //first w
230     }
231     else{
232       Av_Product<<<blocksPerGrid, threadsPerBlock, sharedMemSize>>>(d_MatA, d_VecV, d_VecW, N)
      ; //first w
233     }
234     cudaThreadSynchronize(); //Needed, kind of barrier to sychronize all threads
235
236     // This part is the main code of the iteration process for the Power Method in GPU.
237     // Please finish this part based on the given code. Do not forget the command line
238     // cudaThreadSynchronize() after callig the function every time in CUDA to synchoronize
      the threads
239     ////////////////////////////////////////////
240     for(int i=0; i < max_iteration; ++i){
241       timer_start(&time_GPU_mem);
242       cudaMemcpy(d_NormW, h_NormW, norm_size, cudaMemcpyHostToDevice);
243       timer_stop(&time_GPU_mem);
244       cudaThreadSynchronize();
245
246       // Get norm
247       cudaMemset(d_NormW, 0, norm_size);
248       FindNormW<<<blocksPerGrid, threadsPerBlock, sharedMemSize>>>(d_VecW, d_NormW, N);
249       cudaThreadSynchronize(); //Needed, kind of barrier to sychronize all threads
250
251       // Normalize vec
252       NormalizeW<<<blocksPerGrid, threadsPerBlock, sharedMemSize>>>(d_VecW, d_NormW, d_VecV, N
      );
253       cudaThreadSynchronize(); //Needed, kind of barrier to sychronize all threads
254
255       // get new w vec
256       if(GLOBAL_MEM == true){
257         Av_ProductGlobal<<<blocksPerGrid, threadsPerBlock, sharedMemSize>>>(d_MatA, d_VecV,
      d_VecW, N); //first w
258       }
259       else{
260         Av_Product<<<blocksPerGrid, threadsPerBlock, sharedMemSize>>>(d_MatA, d_VecV, d_VecW,
      N); //first w
261       }
262       cudaThreadSynchronize(); //Needed, kind of barrier to sychronize all threads
263
264       // get eigenvalue
265       cudaMemset(d_NormW, 0, norm_size);
266       ComputeLamda<<<blocksPerGrid, threadsPerBlock, sharedMemSize>>>(d_VecV, d_VecW, d_NormW,
       N);
267       cudaThreadSynchronize(); //Needed, kind of barrier to sychronize all threads
268       timer_start(&time_GPU_mem);
269       cudaMemcpy(&Lambda, d_NormW, sizeof(float), cudaMemcpyDeviceToHost);
270       timer_stop(&time_GPU_mem);
271
272       printfD(2,"GPU lamda at %d: %f \n", i, Lambda);
273       // If residual is lass than epsilon break
274       if(fabs(OldLamda - Lambda) < EPS){
275         printfD(1,"Early exit at %d\n", i);
276         break;
277       }
278       OldLamda = Lambda;
```

```
279     }
280
281
282     timer_stop(&total_time_GPU);
283     runtime = timer_get_elapsed(&total_time_GPU);
284     double memtime = timer_get_elapsed(&time_GPU_mem);
285     printf("Lambda: %f\n", Lambda);
286     printfD(1,"GPU: run time = %f secs.\n",runtime);
287     if(WITHOUT_MEMCPY == true){
288         runtime -= memtime;
289         assert (runtime > 0);
290     }
291     return runtime;
292 }
293
294 // Host code
295 int main(int argc, char** argv)
296 {
297     struct timespec t_start,t_end;
298     double runtime;
299     ParseArguments(argc, argv);
300
301     int N = GlobalSize;
302     printf("Matrix size %d X %d \n", N, N);
303     size_t vec_size = N * sizeof(float);
304     size_t mat_size = N * N * sizeof(float);
305     size_t norm_size = sizeof(float);
306
307     // Allocate normalized value in host memory
308     h_NormW = (float*)malloc(norm_size);
309     // Allocate input matrix in host memory
310     h_MatA = (float*)malloc(mat_size);
311     // Allocate initial vector V in host memory
312     h_VecV = (float*)malloc(vec_size);
313     // Allocate W vector for computations
314     h_VecW = (float*)malloc(vec_size);
315
316
317     // Initialize input matrix
318     UploadArray(h_MatA, N);
319     InitOne(h_VecV,N);
320
321     printf("Power method in CPU starts\n");
322     clock_gettime(CLOCK_REALTIME,&t_start);
323     RunCPUPowerMethod();   // the lamda is already solved here
324     clock_gettime(CLOCK_REALTIME,&t_end);
325     runtime = (t_end.tv_sec - t_start.tv_sec) + 1e-9*(t_end.tv_nsec - t_start.tv_nsec);
326     printf("CPU: run time = %f secs.\n",runtime);
327     printf("Power method in CPU is finished\n");
328
329
330     ////////////////////////////////////////////////
331     // This is the starting points of GPU
332     // burner run!
333     RunGPUPowerMethod(N);
334     // Step 1
335     printf(">>>Step 1\n");
336     GLOBAL_MEM = true;
337     for(int i = 0; i < 5; ++i){
338       RunGPUPowerMethod(N);
339       CleanGPU();
340     }
341     GLOBAL_MEM = false;
342     printf(">>>Step 1 shared mem\n");
343     for(int i = 0; i < 5; ++i){
344       RunGPUPowerMethod(N);
345       CleanGPU();
346     }
347
348     // Step 2:
349     //PRINTLEVEL = 0;
350     //int Ns[] = {5000};
351     //for(int i = 0; i < 1; ++i){
```

```
352     //  N = Ns[i];
353     //  double time = RunGPUPowerMethod(N);
354     //  printf("%d - GPU: run time = %f secs.\n",N,time);
355     //  CleanGPU();
356     //}
357     ////////////////////////////////////////////////
358 
359     Cleanup();
360 }
361 void CleanGPU(){
362        // Free device memory
363     if (d_MatA)
364         cudaFree(d_MatA);
365     if (d_VecV)
366         cudaFree(d_VecV);
367     if (d_VecW)
368         cudaFree(d_VecW);
369     if (d_NormW)
370         cudaFree(d_NormW);
371 }
372 
373 void Cleanup(void)
374 {
375     // Free device memory
376     if (d_MatA)
377         cudaFree(d_MatA);
378     if (d_VecV)
379         cudaFree(d_VecV);
380     if (d_VecW)
381         cudaFree(d_VecW);
382     if (d_NormW)
383         cudaFree(d_NormW);
384 
385     // Free host memory
386     if (h_MatA)
387         free(h_MatA);
388     if (h_VecV)
389         free(h_VecV);
390     if (h_VecW)
391         free(h_VecW);
392     if (h_NormW)
393         free(h_NormW);
394 
395     exit(0);
396 }
397 
398 // Allocates an array with zero value.
399 void InitOne(float* data, int n)
400 {
401     for (int i = 0; i < n; i++)
402         data[i] = 0;
403   data[0]=1;
404 }
405 
406 void UploadArray(float* data, int n)
407 {
408    int total = n*n;
409    int value=1;
410     for (int i = 0; i < total; i++)
411     {
412       data[i] = (int) (rand() % (int)(101));//1;//value;
413       value ++; if(value>n) value =1;
414       // data[i] = 1;
415     }
416 }
417 
418 // Obtain program arguments
419 void Arguments(int argc, char** argv)
420 {
421     for (int i = 0; i < argc; ++i)
422     {
423         if (strcmp(argv[i], "--size") == 0 || strcmp(argv[i], "-size") == 0)
424         {
```

```
            GlobalSize = atoi(argv[i+1]);
        i = i + 1;
        }
        if (strcmp(argv[i], "--max_iteration") == 0 || strcmp(argv[i], "-max_iteration") == 0)
        {
            max_iteration = atoi(argv[i+1]);
        i = i + 1;
        }
    }
}


void checkCardVersion()
{
   cudaDeviceProp prop;

   cudaGetDeviceProperties(&prop, 0);

   printfD(2, "This GPU has major architecture %d, minor %d \n",prop.major,prop.minor);
   if(prop.major < 2)
   {
      fprintf(stderr,"Need compute capability 2 or higher.\n");
      exit(1);
   }
}

//   _     _ _______ _____ _       _____ _________      __      __  __ ______ _______ _     _  ____  _____   _____
//  | |   | |__   __|_   _| |     |_   _|__   __\ \    / /     |  \/  |  ____|__   __| |   | |/ __ \|  __ \ / ____|
//  | |   | |  | |    | | | |       | |    | |   \ \_/ /      | \  / | |__     | |  | |__| | |  | | |  | | (___
//  | |   | |  | |    | | | |       | |    | |    \   /       | |\/| |  __|    | |  |  __  | |  | | |  | |\___ \
//  | |__| |  | |   _| |_| |____ _| |_   | |     | |         | |  | | |____   | |  | |  | | |__| | |__| |____) |
//   \____/   |_|  |_____|______|_____|  |_|     |_|         |_|  |_|______|  |_|  |_|  |_|\____/|_____/|_____/


// This is a list of utility methods that should you use in your code

/*****************************************************************************
This function finds the product of Matrix A and vector V
*****************************************************************************/

//
    ************************************************************************************************

// parallelization method for the Matrix-vector multiplication as follows:

// each thread handle a multiplication of each row of Matrix A and vector V;

// The share memory is limited for a block, instead of reading an entire row of matrix A or
    vector V from global memory to share memory,
// a square submatrix of A is shared by a block, the size of square submatrix is BLOCK_SIZE*
    BLOCK_SIZE; Thus, a for-loop is used to
// handle a multiplication of each row of Matrix A and vector V step by step. In eacg step,
    two subvectors with size BLOCK_SIZE is multiplied.
//
    ************************************************************************************************


__global__ void Av_ProductGlobal(float* g_MatA, float* g_VecV, float* g_VecW, int N)
{
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row >= N) return;

    float sum = 0.0f;
    for (int col = 0; col < N; col++) {
        sum += g_MatA[row * N + col] * g_VecV[col];
    }
    g_VecW[row] = sum;
```

```
}
__global__ void Av_Product(float* g_MatA, float* g_VecV, float* g_VecW, int N)
{
    // Block index
    int bx = blockIdx.x;

    // Thread index
    int tx = threadIdx.x;

    int aBegin = N * BLOCK_SIZE * bx;

    int aEnd   = aBegin + N - 1;
    int step  = BLOCK_SIZE;

    int bBegin = 0;//BLOCK_SIZE * bx;
    int bIndex=0;
    int aIndex =0;
    float Csub = 0;

    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += step, b += step)
    {

        __shared__ float As[BLOCK_SIZE*BLOCK_SIZE];

        __shared__ float bs[BLOCK_SIZE];


        for (int aa = 0; aa < BLOCK_SIZE;aa+= 1)
        {
            aIndex = a+tx+aa*N;
            if( aIndex < N*N)
              As[tx+aa*BLOCK_SIZE] = g_MatA[aIndex];
            else
              As[tx+aa*BLOCK_SIZE] = 0;
        }

        bIndex = b+tx;
        if(bIndex<N)
          bs[tx] = g_VecV[bIndex];
        else
          bs[tx] = 0;

        __syncthreads();

        for (int k = 0; k < BLOCK_SIZE; ++k)
        {
            Csub += As[k+tx*BLOCK_SIZE] * bs[k];
        }//}
        __syncthreads();
    }

    g_VecW[ BLOCK_SIZE * bx + tx] = Csub;
}

/*************************************************
Normalizes vector W : W/norm(W)
*************************************************/
__global__ void FindNormW(float* g_VecW, float * g_NormW, int N)
{
  // shared memory size declared at kernel launch
  extern __shared__ float sdata[];
  unsigned int tid = threadIdx.x;
  unsigned int globalid = blockIdx.x*blockDim.x + threadIdx.x;

  // For thread ids greater than data space
  if (globalid < N) {
     sdata[tid] =  g_VecW[globalid];
  }
  else {
     sdata[tid] = 0;  // Case of extra threads above N
  }
```

```
558
559   // each thread loads one element from global to shared mem
560   __syncthreads();
561
562   sdata[tid] = sdata[tid] * sdata[tid];
563   __syncthreads();
564
565   // do reduction in shared mem
566   for (unsigned int s=blockDim.x / 2; s > 0; s = s >> 1) {
567      if (tid < s) {
568          sdata[tid] = sdata[tid] + sdata[tid+ s];
569      }
570      __syncthreads();
571   }
572    // atomic operations:
573   if (tid == 0) atomicAdd(g_NormW,sdata[0]);
574 }
575
576 __global__ void NormalizeW(float* g_VecW, float* g_NormW, float* g_VecV, int N)
577 {
578   // shared memory size declared at kernel launch
579   extern __shared__ float sNormData[];
580   unsigned int tid = threadIdx.x;
581   unsigned int globalid = blockIdx.x*blockDim.x + threadIdx.x;
582
583   if(tid==0) sNormData[0] =  sqrt(g_NormW[0]);
584   __syncthreads();
585
586   // For thread ids greater than data space
587   if (globalid < N) {
588      g_VecV[globalid] = g_VecW[globalid]/sNormData[0];
589   }
590
591 }
592
593 __global__ void ComputeLamda( float* g_VecV, float* g_VecW, float * g_Lamda,int N)
594 {
595   // shared memory size declared at kernel launch
596   extern __shared__ float sdataVW[];
597   unsigned int tid = threadIdx.x;
598   unsigned int globalid = blockIdx.x*blockDim.x + threadIdx.x;
599
600   // For thread ids greater than data space
601   if (globalid < N) {
602      sdataVW[tid] =  g_VecV[globalid] * g_VecW[globalid];
603   }
604   else {
605      sdataVW[tid] = 0;  // Case of extra threads above N
606   }
607
608   // each thread loads one element from global to shared mem
609   __syncthreads();
610
611   // do reduction in shared mem
612   for (unsigned int s=blockDim.x / 2; s > 0; s = s >> 1) {
613      if (tid < s) {
614          sdataVW[tid] = sdataVW[tid] + sdataVW[tid+ s];
615      }
616      __syncthreads();
617   }
618    // atomic operations:
619   if (tid == 0) atomicAdd(g_Lamda,sdataVW[0]);
620 }
```