

DELFT UNIVERSITY OF TECHNOLOGY

INTRODUCTION TO HIGH PERFORMANCE COMPUTING  
WI4049TU

---

# Lab Report

---

*Author:*  
Elias Wachmann (6300421)

November 22, 2024



## 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 blue heading in the text or an additional note is added under a figure or table. For example:

**This is a shining point.**

## 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

```
1 #include "mpi.h"
2 #include <stdio.h>
3
4 int np, rank;
5
6 int main(int argc, char **argv)
7 {
8     MPI_Init(&argc, &argv);
9     MPI_Comm_size(MPI_COMM_WORLD, &np);
10    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11
12    printf("Node %d of %d says: Hello world!\n", rank, np);
13
14    MPI_Finalize();
15    return 0;
16 }
```

This program can be compiled with the following command:

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

And run with:

```
srunk -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. Furthermore I reran the whole program 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 ( $\leq 131072$ )	$5.95 \times 10^{-7} \cdot x + 7.97 \times 10^{-4}$	0.92
Single Node	Large ( $\geq 131072$ )	$4.61 \times 10^{-7} \cdot x + 1.23 \times 10^{-2}$	0.89
Two Node	Small ( $\leq 32768$ )	$1.07 \times 10^{-6} \cdot x + 2.60 \times 10^{-3}$	0.97
Two Node	Large ( $\geq 32768$ )	$4.41 \times 10^{-7} \cdot x + 3.42 \times 10^{-3}$	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.

**TODO: Further analysis needed?**

### 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:

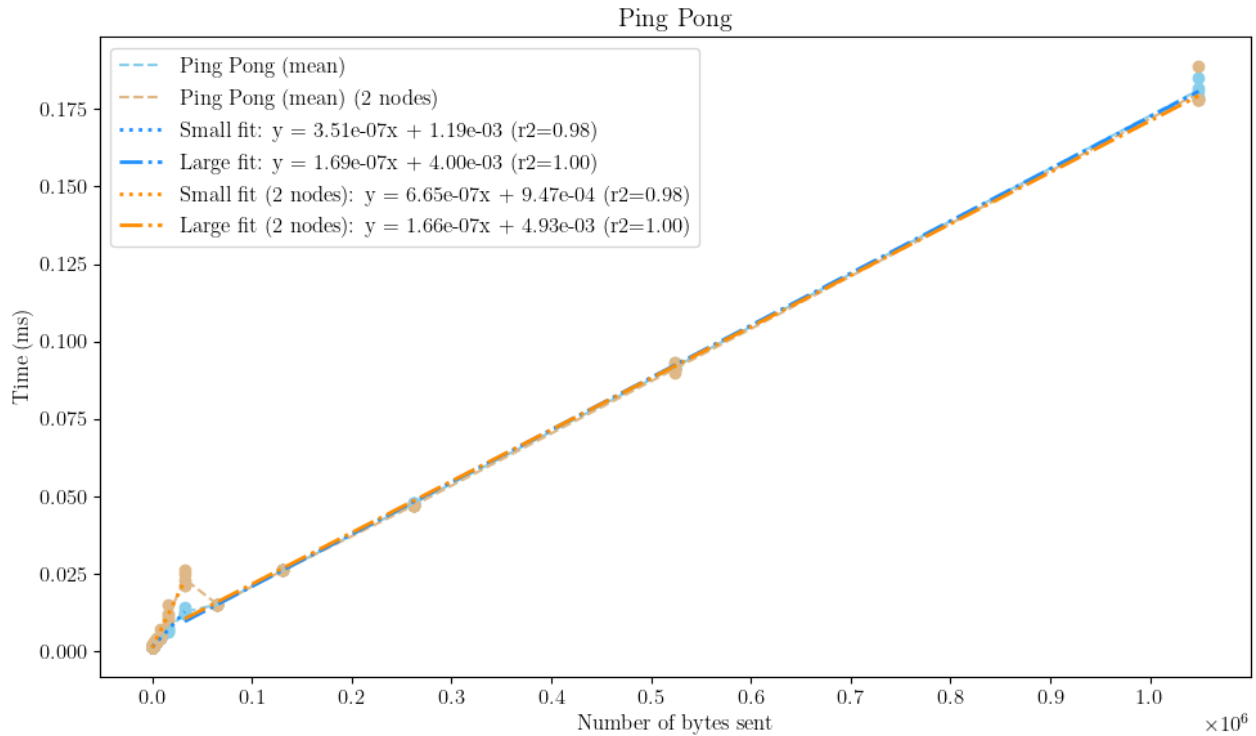


Figure 2: Ping Pong with MPI\_SendRecv: Number of bytes sent vs. average time taken from 1000 pairs of send / receive. 5 runs shown for each size as scatter plot. Mean of these 5 runs shown as line. Blue small fit includes all data points up to 32768 bytes, blue large from there. Red small fit includes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

We get the following fits and Rš values for the runs:

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

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

**TODO: Further analysis needed?**

## 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 exercise 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 program 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 below:

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:  
 $\text{total\_rows} - (\text{nr\_workers} - 1) \cdot \text{floor}(\frac{\text{total\_rows}}{\text{nr\_workers}})$ .
- All other workers and the master (= rank 0) get the same number of rows:  $\text{floor}(\frac{\text{total\_rows}}{\text{nr\_workers}})$ .
- 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 below) 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:

```
1 typedef struct MM_input {
2     size_t rows;
3     double *a;
4     double *b;
5 } 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  $2000 \times 2000$ . 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  $\times$  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 clearly 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.

**TODO: Data locality?**

# 1 Poisson solver

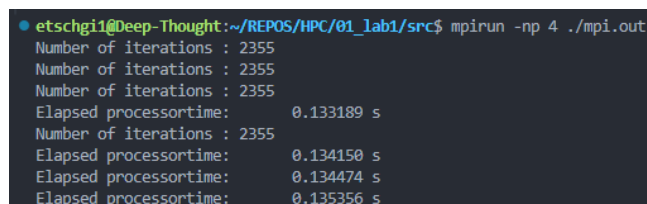
In this section of the lab report, we will discuss a parallel implementation of the Poisson solver. The Poisson solver is a numerical method used to solve the Poisson equation, which is a partial differential equation that is useful in many areas of physics.

**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 below. The finished implementation can be found in the appendix for this section.

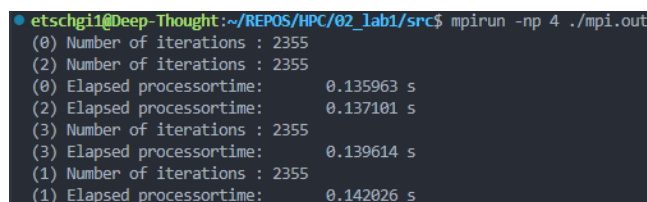
1. **Step:** After adding `MPI_Init` and `MPI_Finalize`, we can run the program with multiple processes. We can see that the program runs with 4 processes in [Figure 4](#) via the quadrupled output.



```
etschgi1@Deep-Thought:~/REPOS/HPC/01_lab1/src$ mpirun -np 4 ./mpi.out
Number of iterations : 2355
Number of iterations : 2355
Number of iterations : 2355
Number of iterations : 2355
Elapsed procestime:      0.133189 s
Number of iterations : 2355
Elapsed procestime:      0.134150 s
Elapsed procestime:      0.134474 s
Elapsed procestime:      0.135356 s
```

Figure 4: MPI\_Poisson after Step 1 - Running with 4 processes

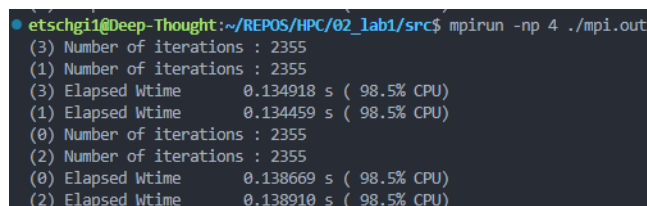
2. **Step:** To see which process is doing what, I included the rank of the process for the print statements as shown in [Figure 5](#).



```
etschgi1@Deep-Thought:~/REPOS/HPC/02_lab1/src$ mpirun -np 4 ./mpi.out
(0) Number of iterations : 2355
(2) Number of iterations : 2355
(0) Elapsed procestime:      0.135963 s
(2) Elapsed procestime:      0.137101 s
(3) Number of iterations : 2355
(3) Elapsed procestime:      0.139614 s
(1) Number of iterations : 2355
(1) Elapsed procestime:      0.142026 s
```

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.



```
etschgi1@Deep-Thought:~/REPOS/HPC/02_lab1/src$ mpirun -np 4 ./mpi.out
(3) Number of iterations : 2355
(1) Number of iterations : 2355
(3) Elapsed Wtime      0.134918 s ( 98.5% CPU)
(1) Elapsed Wtime      0.134459 s ( 98.5% CPU)
(0) Number of iterations : 2355
(2) Number of iterations : 2355
(0) Elapsed Wtime      0.138669 s ( 98.5% CPU)
(2) Elapsed Wtime      0.138910 s ( 98.5% CPU)
```

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](#).

```
(0) (x,y)=(0,0)
(0) top 1, right -2, bottom -2, left 2
(1) (x,y)=(0,1)
(1) top -2, right -2, bottom 0, left 3
(2) (x,y)=(1,0)
(2) top 3, right 0, bottom -2, left -2
(3) (x,y)=(1,1)
(3) top -2, right 1, bottom 2, left -2
```

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.

```
etschgi@Deep-Thought:~/REPOS/HPC/02_lab1/src$ mpirun -np 3 ./mpi.out 3 1
(0) (x,y)=(0,0)
(0) top -2, right -2, bottom -2, left 1
(1) (x,y)=(1,0)
(1) top -2, right 0, bottom -2, left 2
(2) (x,y)=(2,0)
(2) top -2, right 1, bottom -2, left -2
(0) Number of iterations : 1
(2) Number of iterations : 1
(2) Elapsed Wtime 0.000668 s ( 95.3% CPU)
(0) Elapsed Wtime 0.000917 s ( 95.9% CPU)
(1) Number of iterations : 695
(1) Elapsed Wtime 0.014772 s ( 95.2% CPU)
```

```
etschgi@Deep-Thought:~/REPOS/HPC/02_lab1/src$ mpirun -np 3 ./mpi.out 1 3
(1) (x,y)=(0,1)
(1) top 2, right -2, bottom 0, left -2
(1) Number of iterations : 1
(2) (x,y)=(0,2)
(2) top -2, right -2, bottom 1, left -2
(0) (x,y)=(0,0)
(0) top 1, right -2, bottom -2, left -2
(1) Elapsed Wtime 0.000616 s ( 95.4% CPU)
(0) Number of iterations : 601
(2) Number of iterations : 723
(0) Elapsed Wtime 0.017636 s ( 95.3% CPU)
(2) Elapsed Wtime 0.017801 s ( 95.3% CPU)
```

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 committing 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](#).

```

(0) (x,y)=(0,0)
(0) top -1, right 2, bottom 1, left -1
(1) (x,y)=(0,1)
(1) top 0, right 3, bottom -1, left -1
(2) (x,y)=(1,0)
(2) top -1, right -1, bottom 3, left 0
(3) (x,y)=(1,1)
(3) top 2, right -1, bottom -1, left 1
(0) Number of iterations : 2355
(1) Number of iterations : 2355
(2) Number of iterations : 2355
(3) Number of iterations : 2355
(1) Elapsed Wtime      0.287549 s ( 99.9% CPU)
(2) Elapsed Wtime      0.287537 s (100.0% CPU)
(3) Elapsed Wtime      0.287537 s (100.0% CPU)
(0) Elapsed Wtime      0.295957 s ( 99.9% CPU)

```

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 programmes perform the exact same calculation steps.

## 1.2 Exercises, modifications, and performance aspects

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

$n$ :	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

Table 4: Notation for this section

$pt = 414$  means 4 processors in a  $1 \times 4$  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.

### 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  $\omega$  to calculate the updated values. A quick test shows a speedup of roughly a factor of 10. More systematic tests will be done in the next section.

### 1.2.2 Optimal $\omega$ for 4 proc. on a 4x1 grid

With the power of a little python scripting we can easily test different values for  $\omega$  and plot the results as seen in [Figure 10](#).

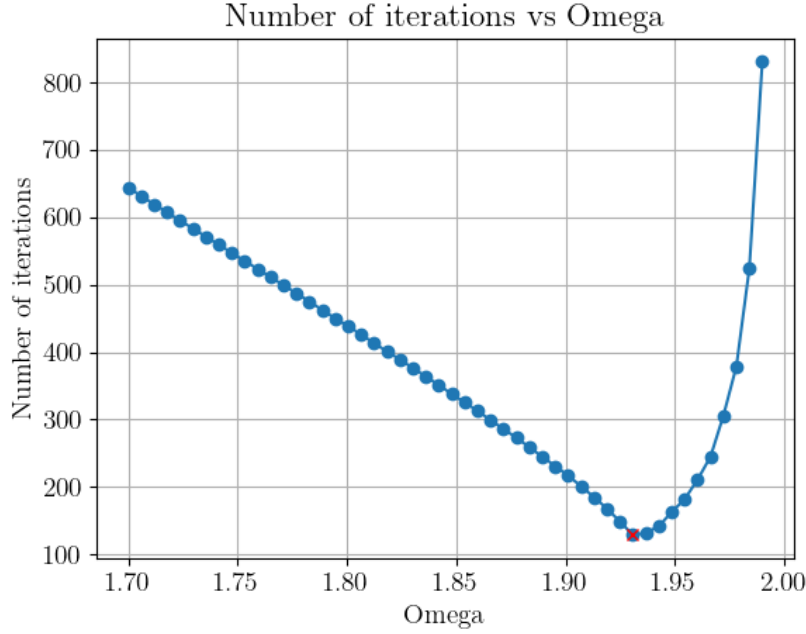


Figure 10: [Optimal  \$\omega\$  for 4 processors on a 4x1 grid](#)

We find that the optimal  $\omega$  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 twice: Once with a  $4 \times 1$  topology (as in the previous section) and once with a  $2 \times 2$  topology. We use grid sizes of  $10 \times 10$ ,  $25 \times 25$ ,  $50 \times 50$ ,  $100 \times 100$ ,  $200 \times 200$ ,  $400 \times 400$ ,  $800 \times 800$  and  $1600 \times 1600$  and set  $\omega = 1.95$  for all runs. The results are shown in [Figure 11](#).

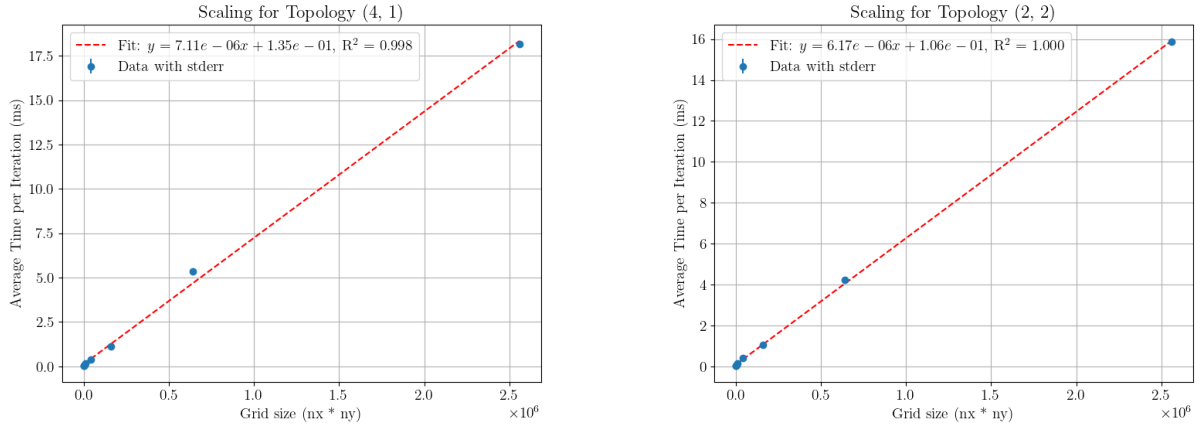


Figure 11: Scaling behavior of the Poisson solver with different grid sizes and processor topologies

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

Topology	$\alpha$	$\beta$
$4 \times 1$	$1.35 + 10^{-1}$	$7.11 + 10^{-6}$
$2 \times 2$	$1.06 + 10^{-1}$	$6.17 + 10^{-6}$

Table 5: Scaling factors for different processor topologies for the Poisson solver

Using:  $t(n) = \alpha + \beta \cdot n$  as a model

### What can you conclude from the scaling behavior?

We see that the scaling behavior is very close to linear for both topologies. This means that the parallel implementation scales as expected with the number of grid points.

If we compare the scaling factors ( $\beta$ ) for the two topologies we see that the  $2 \times 2$  topology scales slightly better than the  $4 \times 1$  topology. This is not surprising, as the  $2 \times 2$  topology has a more balanced communication workload balance. In the  $2 \times 2$  topology every processor has two neighbors, while in the  $4 \times 1$  topology 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.

#### 1.2.4 Scaling behavior [Theory - no measurements]

If I could choose between a  $16 \times 1$ ,  $8 \times 2$ ,  $4 \times 4$ ,  $2 \times 8$ ,  $1 \times 16$  topology, I would choose the  $4 \times 4$  topology. This is because the  $4 \times 4$  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  $4 \times 1$  topology square grids with sidelength: 10, 25, 50, 100, 200, 400, 800, 1600. The results for different  $\omega$  are shown in [Figure 12](#).

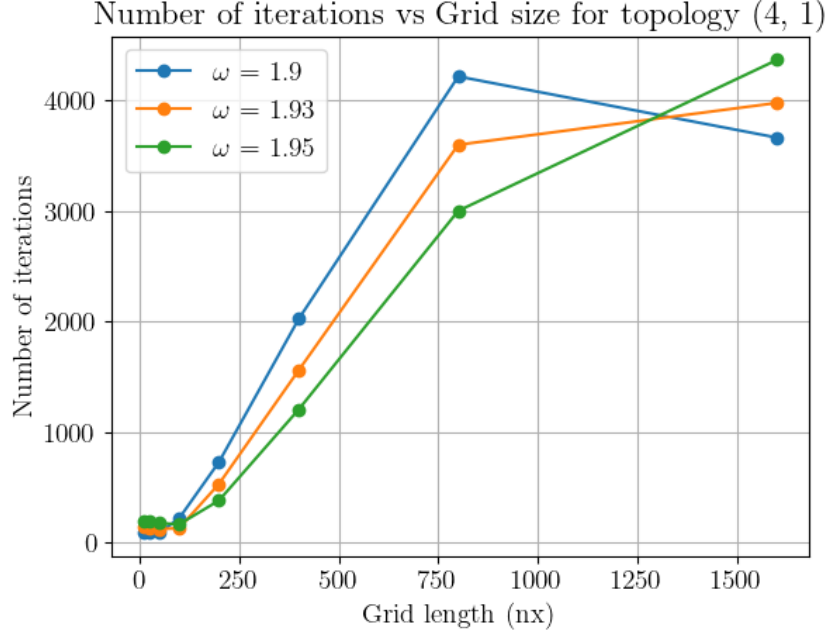


Figure 12: Iterations needed for convergence with different grid side lengths

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 differently sized matrices we have to solve. SOR overrelaxes high-frequency errors and underrelaxes low-frequency errors (the later for stability). The optimal  $\omega$  is therefore 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.

### 1.2.6 Error as a function of the iteration number

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

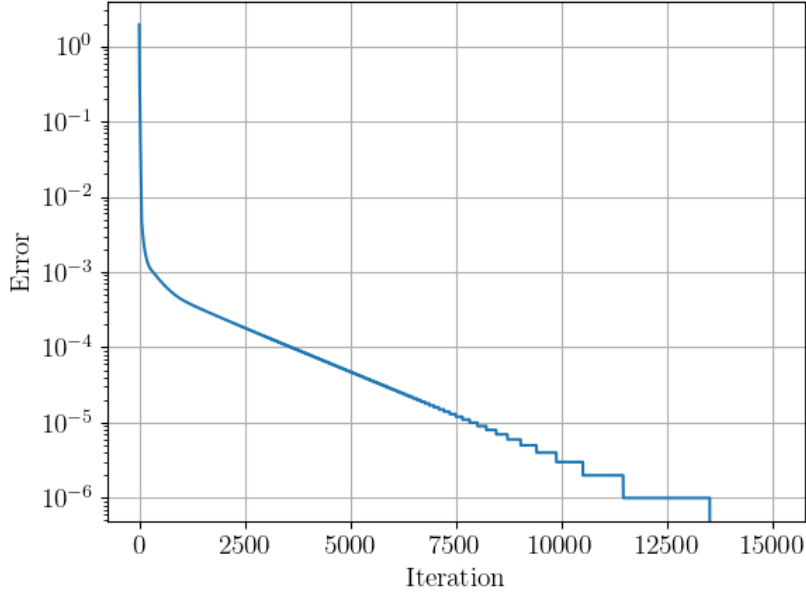


Figure 13: Error as a function of the iteration number

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.

### 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  $4 \times 1$  topology,  $\omega = 1.93$  and a  $800 \times 800$  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 ( $(7.0 \pm 0.4)\%$ ). 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.

Iterations	MPI_Allreduce - speedup (factor)	calculated overall speedup (%)
1	1.00	0
10	$6.0 \pm 2.0$	$5.9 \pm 0.5$
100	$62 \pm 6$	$6.9 \pm 0.4$

Table 6: Speedup in MPI\_Allreduce calls for different iteration counts and calculated overall speedup (%)

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  $4 \times 1$  topology,  $\omega = 1.93$  and different grid sizes and track the iterations and time as seen in Figure 14.

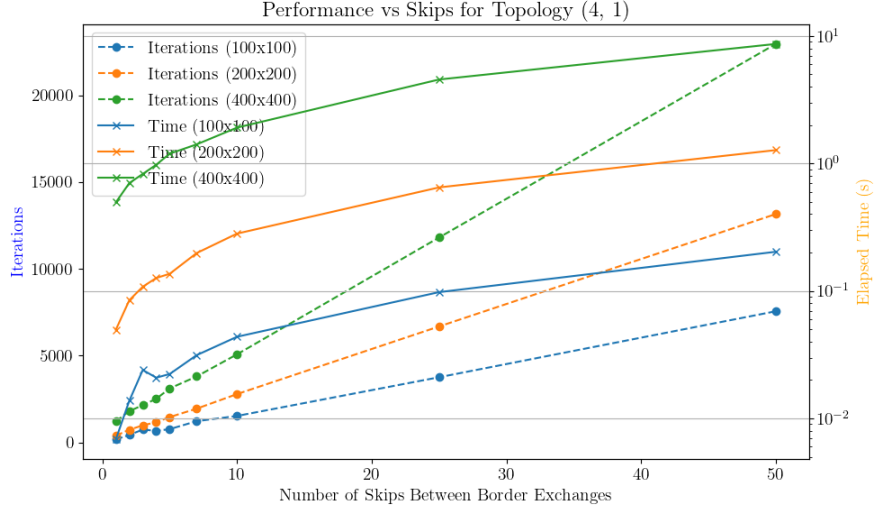


Figure 14: Speedup by reducing border exchanges

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 fastest 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  $100 \times 100$  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.

### 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:

```
1  for (x = 1; x < dim[X_DIR] - 1; x++){
2      for (y = 1; y < dim[Y_DIR] - 1; y++){
3          if ((x + offset[X_DIR] + y + offset[Y_DIR]) % 2 == parity && source[x][y] != 1){
4              ...
```

and we change it to:

```
1  for (x = 1; x < dim[X_DIR] - 1; x++){
2      for (y = (x + offset[X_DIR]) % 2 == 0 ? 1 : 2; y < dim[Y_DIR] - 1; y += 2){
3          if (source[x][y] != 1){
4              ...
```

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

$$t_{\text{no\_improvements}} = (5.66 \pm 0.09) \text{ s} \quad \text{and} \quad t_{\text{loop\_improvements}} = (5.63 \pm 0.08) \text{ s}$$

So we get a minimal speedup of about 0.5% by optimizing the loop.

#### Why is the speedup relatively small?

We witness the sheer speed of c-loops here. There are certainly multiple factors at play here. Manual optimization of the loops is often not as effective as one might think, because the compiler is already very good

at optimizing loops and has probably more tricks up its sleeve than we do. We could test that hypothesis by comparing against versions without any optimization flags (-O0): We perform 5 runs with the same setup and -O0 and get the following times:

$$t_{\text{no\_improvements}} = (5.664 \pm 0.020) \text{ s} \quad \text{and} \quad t_{\text{loop\_improvements}} = (5.69 \pm 0.06) \text{ s}$$

Funily enough, now the optimized loop is actually slower than the unoptimized one. Did we do something wrong? No, but probably the compiler doesn't optimize the loop as well as we thought and we were just lucky with the first run. Notice that the two times are actually within the errorbars of each other!

#### What else could it be?

Well maybe we have to think hardware and branch prediction. The loop is very simple and the branch prediction of the CPU might be very good at predicting the next iteration. This means that the CPU can already start fetching the next instruction before the branch is resolved. In our case the CPU's branch predictor will learn that only every second iteration is actually executed and will be able to predict the next iteration with high accuracy. This means that the speedup we get from optimizing the loop is very small to non existent.

#### 1.2.10 Optional - Time spent within Exchange\_Borders



## 2 Finite elements simulation

### 3 Eigenvalue solution by Power Method on GPU

## Appendix - Introductory exercise

The following code was used for the ping pong task:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4
5 // Maximum array size 2^20= 1048576 elements
6 #define MAX_EXPONENT 20
7 #define MAX_ARRAY_SIZE (1<<MAX_EXPONENT)
8 #define SAMPLE_COUNT 1000
9
10 int main(int argc, char **argv)
11 {
12     // Variables for the process rank and number of processes
13     int myRank, numProcs, i;
14     MPI_Status status;
15
16     // Initialize MPI, find out MPI communicator size and process rank
17     MPI_Init(&argc, &argv);
18     MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
19     MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
20
21
22     int *myArray = (int *)malloc(sizeof(int)*MAX_ARRAY_SIZE);
23     if (myArray == NULL)
24     {
25         printf("Not enough memory\n");
26         exit(1);
27     }
28     // Initialize myArray
29     for (i=0; i<MAX_ARRAY_SIZE; i++)
30         myArray[i]=1;
31
32     int number_of_elements_to_send;
33     int number_of_elements_received;
34
35     // PART C
36     if (numProcs < 2)
37     {
38         printf("Error: Run the program with at least 2 MPI tasks!\n");
39         MPI_Abort(MPI_COMM_WORLD, 1);
40     }
41     double startTime, endTime;
42
43     // TODO: Use a loop to vary the message size
44     for (size_t j = 0; j <= MAX_EXPONENT; j++)
45     {
46         number_of_elements_to_send = 1<<j;
47         if (myRank == 0)
48         {
49             myArray[0]=myArray[1]+1; // activate in cache (avoids possible delay when sending
the 1st element)
50             startTime = MPI_Wtime();
51             for (i=0; i<SAMPLE_COUNT; i++)
52             {
53                 MPI_Send(myArray, number_of_elements_to_send, MPI_INT, 1, 0,
54                     MPI_COMM_WORLD);
55                 MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
56                 MPI_Get_count(&status, MPI_INT, &number_of_elements_received);
57
58                 MPI_Recv(myArray, number_of_elements_received, MPI_INT, 1, 0,
59                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
60             } // end of for-loop
61
62             endTime = MPI_Wtime();
63             printf("Rank %2.1i: Received %i elements: Ping Pong took %f seconds\n", myRank,
number_of_elements_received, (endTime - startTime)/(2*SAMPLE_COUNT));
64         }
65         else if (myRank == 1)
66         {
67             // Probe message in order to obtain the amount of data
68             MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

```

69     MPI_Get_count(&status, MPI_INT, &number_of_elements_received);
70
71     for (i=0; i<SAMPLE_COUNT; i++)
72     {
73         MPI_Recv(myArray, number_of_elements_received, MPI_INT, 0, 0,
74                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
75         MPI_Send(myArray, number_of_elements_to_send, MPI_INT, 0, 0,
76                 MPI_COMM_WORLD);
77     } // end of for-loop
78 }
79 }
80
81 // Finalize MPI
82 MPI_Finalize();
83
84 return 0;
85 }

```

For the bonus task, the following code was used:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4
5  // Maximum array size 2^20= 1048576 elements
6  #define MAX_EXPONENT 20
7  #define MAX_ARRAY_SIZE (1<<MAX_EXPONENT)
8  #define SAMPLE_COUNT 1000
9
10 int main(int argc, char **argv)
11 {
12     // Variables for the process rank and number of processes
13     int myRank, numProcs, i;
14     MPI_Status status;
15
16     // Initialize MPI, find out MPI communicator size and process rank
17     MPI_Init(&argc, &argv);
18     MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
19     MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
20
21
22     int *myArray = (int *)malloc(sizeof(int)*MAX_ARRAY_SIZE);
23     if (myArray == NULL)
24     {
25         printf("Not enough memory\n");
26         exit(1);
27     }
28     // Initialize myArray
29     for (i=0; i<MAX_ARRAY_SIZE; i++)
30         myArray[i]=1;
31
32     int number_of_elements_to_send;
33     int number_of_elements_received;
34
35     // PART C
36     if (numProcs < 2)
37     {
38         printf("Error: Run the program with at least 2 MPI tasks!\n");
39         MPI_Abort(MPI_COMM_WORLD, 1);
40     }
41     double startTime, endTime;
42
43     // TODO: Use a loop to vary the message size
44     for (size_t j = 0; j <= MAX_EXPONENT; j++)
45     {
46         number_of_elements_to_send = 1<<j;
47         if (myRank == 0)
48         {
49             myArray[0]=myArray[1]+1; // activate in cache (avoids possible delay when sending
the 1st element)
50             startTime = MPI_Wtime();
51             for (i=0; i<SAMPLE_COUNT; i++)
52             {
53                 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);
54     }
55
56     endTime = MPI_Wtime();
57     printf("Rank %2.i: Received %i elements: Ping Pong took %f seconds\n", myRank,
    number_of_elements_to_send, (endTime - startTime)/(2*SAMPLE_COUNT));
58     }
59     else if (myRank == 1)
60     {
61         for (i=0; i<SAMPLE_COUNT; i++)
62         {
63             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);
64         }
65     }
66 }
67
68 // Finalize MPI
69 MPI_Finalize();
70
71 return 0;
72 }

```

The matrix multiplication used the following code:

```

1  /*****
2  * FILE: mm.c
3  * DESCRIPTION:
4  *   This program calculates the product of matrix a[nra][nca] and b[nca][ncb],
5  *   the result is stored in matrix c[nra][ncb].
6  *   The max dimension of the matrix is constraint with static array
7  *   declaration, for a larger matrix you may consider dynamic allocation of the
8  *   arrays, but it makes a parallel code much more complicated (think of
9  *   communication), so this is only optional.
10  *
11  *****/
12
13 #include <math.h>
14 #include <mpi.h>
15 #include <stdbool.h>
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <string.h>
19
20 #define NRA 2000 /* number of rows in matrix A */
21 #define NCA 2000 /* number of columns in matrix A */
22 #define NCB 2000 /* number of columns in matrix B */
23 // #define N 1000
24 #define EPS 1e-9
25 #define SIZE_OF_B NCA*NCB*sizeof(double)
26
27 bool eps_equal(double a, double b) { return fabs(a - b) < EPS; }
28
29 void print_flattened_matrix(double *matrix, size_t rows, size_t cols, int rank) {
30     printf("[%d]\n", rank);
31     for (size_t i = 0; i < rows; i++) {
32         for (size_t j = 0; j < cols; j++) {
33             printf("%10.2f ", matrix[i * cols + j]); // Accessing element in the 1D array
34         }
35         printf("\n"); // Newline after each row
36     }
37 }
38
39 int checkResult(double *truth, double *test, size_t Nr_col, size_t Nr_rows) {
40     for (size_t i = 0; i < Nr_rows; ++i) {
41         for (size_t j = 0; j < Nr_col; ++j) {
42             size_t index = i * Nr_col + j;
43             if (!eps_equal(truth[index], test[index])) {
44                 return 1;
45             }
46         }
47     }
48     return 0;
49 }

```

```

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
164     // performance
165     setupMatrices(a,b,c);
166
167     double start_time = MPI_Wtime();
168     double (*b_transposed)[NCA] = malloc(sizeof(double) * NCA * NCB);
169     for (size_t i = 0; i < NCA; i++) {
170         for (size_t j = 0; j < NCB; j++) {
171             b_transposed[j][i] = b[i][j];
172         }
173     }
174
175     /** Initialize matrices */
176     // given number of workers I'll split
177     size_t rows_per_worker = NRA / (num_workers+1); //takes corresponding columns from other
178     // matrix
179     printf("rows per worker: %zu\n", rows_per_worker);
180     size_t row_end_first = NRA - rows_per_worker*num_workers;
181     printf("first gets most: %zu\n", row_end_first);
182
183     //setup requests
184     MPI_Request requests[num_workers];
185     MM_input *data_first = (MM_input*)malloc(sizeof(MM_input));
186     data_first->rows = row_end_first;
187     data_first->a = (double*)a; //they both start of with no offset!
188     data_first->b = (double*)b_transposed;
189     size_t total_size = sizeof(size_t) + (data_first->rows * NCA)*sizeof(double)+SIZE_OF_B;
190     char* buffer = getbuffer(data_first, total_size); //first one
191
192     // Tag is just nr-cpu -1
193     MPI_Isend(buffer, total_size, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &requests[0]);
194     free(data_first);
195     total_size = sizeof(size_t) + (rows_per_worker * NCA)*sizeof(double) + SIZE_OF_B; //size

```

```

194 is the same for all other - just compute once!
195 size_t i;
196 for (i = 0; i < (num_workers-1); ++i)
197 {
198     MM_input *data = (MM_input*)malloc(sizeof(MM_input));
199     data->rows = rows_per_worker;
200     data->a = (double*)(a + (row_end_first + rows_per_worker*i));
201     data->b = (double*)(b_transposed); // send everything - all needed
202     buffer = getbuffer(data, total_size);
203     printf("nr_worker - %zu\n", i);
204     MPI_Isend(buffer, total_size, MPI_CHAR, i+2, i+1, MPI_COMM_WORLD, &requests[i+1]);
205     free(data);
206 }
207 double* my_a = (double*)(a + (row_end_first + rows_per_worker*i));
208 //I multiply the rest
209 size_t offset = 0;
210 for (size_t row = (NRA-rows_per_worker); row < NRA; row++)
211 {
212     for (size_t col = 0; col < NCB; col++)
213     {
214         res[row * NCB + col] = multsum(my_a+offset, (((double*)b_transposed)+col*NCA), NCA
215 );
216     }
217     offset += NCA;
218 }
219 printf("My c: \n");
220 //wait for rest
221 MPI_Status stats[num_workers];
222 if(MPI_Waitall(num_workers, requests, stats) == MPI_ERR_IN_STATUS){
223     printf("Communication failed!!! - abort\n");
224 }
225 printf(">>>Everything sent and recieved\n");
226 // reviece rest
227 size_t buf_size = sizeof(double)*row_end_first*NCB;
228 double* revbuf;
229 offset = 0;
230 for (size_t worker = 0; worker < num_workers; worker++)
231 {
232     revbuf = (double*)malloc(buf_size); //first gets largest buffer
233     MPI_Recv(revbuf, buf_size/sizeof(double), MPI_DOUBLE, worker+1, worker, MPI_COMM_WORLD
234 ,&stats[worker]);
235     memcpy(&res[offset/sizeof(double)], revbuf, buf_size);
236     free(revbuf);
237     offset += buf_size;
238     buf_size = sizeof(double)*rows_per_worker*NCB;
239 }
240 double time = MPI_Wtime()-start_time;
241 //free all pointers!
242 free(a);
243 free(b);
244 free(b_transposed);
245 free(c);
246 return time;
247 }
248
249
250 double work(int rank, size_t num_workers){
251     size_t rows_per_worker = NRA / (num_workers+1);
252     char* buffer;
253     MPI_Status status;
254     if (rank == 1) // first always get's most work
255     {
256         rows_per_worker = NRA - rows_per_worker*num_workers;
257     }
258     size_t size_of_meta = sizeof(size_t);
259     size_t size_of_a = sizeof(double)*rows_per_worker*NCA;
260     size_t buffersize = size_of_meta+size_of_a + SIZE_OF_B;
261     buffer = (char*)malloc(buffersize);
262
263     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     } else // run sequential
323     {
324         printf("Run sequential!\n");
325         double *res = malloc(sizeof(double) * NRA * NCB);
326         double time = productSequential(res);
327         if (checkResult(res, res, NCB, NRA)) {
328             printf("Matrices do not match!!!\n");
329             return 1;
330         }
331         printf("Matrices match (sequential-trivial)! - took: %.6f s\n", time);
332         free(res);
333     }
334 }
335

```

```

336     MPI_Finalize();
337     return 0;
338 }

```

## Appendix - Poisson solver

The parallel Poisson solver used the following code:

```

1  /*
2  * MPI_Poisson.c
3  * 2D Poisson equation solver (parallel version)
4  */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <math.h>
9  #include <time.h>
10 #include <mpi.h>
11 #include <assert.h>
12
13 #define DEBUG 0
14
15 #define max(a,b) ((a)>(b)?a:b)
16
17
18 // defines for Exercises!
19
20 #define SOR 1
21 #define MONITOR_ERROR 1
22 #define FAST_DO_STEP_LOOP
23 // #define MONITOR_ALLREDUCE 1
24 // #define ALLREDUCE_COUNT 100
25 #define SKIP_EXCHANGE
26
27 #define DEFINES_ON (SOR || MONITOR_ERROR || 0)
28 //defines end
29
30 enum
31 {
32     X_DIR, Y_DIR
33 };
34
35 // only needed for certain configs!
36 #ifdef SOR
37 double sor_omega = 1.9;
38 #endif
39 #ifdef MONITOR_ERROR
40 double *errors=NULL;
41 #endif
42 #ifdef MONITOR_ALLREDUCE
43 double all_reduce_time = 0;
44 #endif
45 #ifdef SKIP_EXCHANGE
46 size_t skip_exchange;
47 #endif
48
49 /* global variables */
50 int gridsize[2];
51 double precision_goal; /* precision_goal of solution */
52 int max_iter; /* maximum number of iterations allowed */
53 int P; //total number of processes
54 int P_grid[2]; // process grid dimensions
55 MPI_Comm grid_comm; //grid communicator
56 MPI_Status status;
57 double hx, hy;
58
59 /* process specific globals*/
60 int proc_rank;
61 double wtime;
62 int proc_coord[2]; // coords of current process in processgrid
63 int proc_top, proc_right, proc_bottom, proc_left; // ranks of neighboring procs
64 // step 7

```

```

65 int offset[2] = {0,0};
66 // step 8
67 MPI_Datatype border_type[2];
68
69 /* benchmark related variables */
70 clock_t ticks; /* number of systemticks */
71 int timer_on = 0; /* is timer running? */
72
73 /* local grid related variables */
74 double **phi; /* grid */
75 int **source; /* TRUE if subgrid element is a source */
76 int dim[2]; /* grid dimensions */
77
78 void Setup_Grid();
79 double Do_Step(int parity);
80 void Solve();
81 void Write_Grid();
82 void Clean_Up();
83 void Debug(char *mesg, int terminate);
84 void start_timer();
85 void resume_timer();
86 void stop_timer();
87 void print_timer();
88
89 void start_timer()
90 {
91     if (!timer_on){
92         MPI_Barrier(grid_comm);
93         ticks = clock();
94         wtime = MPI_Wtime();
95         timer_on = 1;
96     }
97 }
98
99 void resume_timer()
100 {
101     if (!timer_on){
102         ticks = clock() - ticks;
103         wtime = MPI_Wtime() - wtime;
104         timer_on = 1;
105     }
106 }
107
108 void stop_timer()
109 {
110     if (timer_on){
111         ticks = clock() - ticks;
112         wtime = MPI_Wtime() - wtime;
113         timer_on = 0;
114     }
115 }
116
117 void print_timer()
118 {
119     if (timer_on){
120         stop_timer();
121         printf("(%) Elapsed Wtime %14.6f s (%5.1f%% CPU)\n", proc_rank, wtime, 100.0 * ticks
122             * (1.0 / CLOCKS_PER_SEC) / wtime);
123         resume_timer();
124     }
125     else{
126         printf("(%) Elapsed Wtime %14.6f s (%5.1f%% CPU)\n", proc_rank, wtime, 100.0 * ticks
127             * (1.0 / CLOCKS_PER_SEC) / wtime);
128     }
129 }
130
131 void Debug(char *mesg, int terminate)
132 {
133     if (DEBUG || terminate){
134         printf("%s\n", mesg);
135     }
136     if (terminate){
137         exit(1);
138     }
139 }

```

```

136 }
137 }
138
139 void Setup_Proc_Grid(int argc, char **argv){
140     int wrap_around[2];
141     int reorder;
142
143     Debug("My_MPI_Init",0);
144
145     // num of processes
146     MPI_Comm_size(MPI_COMM_WORLD, &P);
147
148     //calculate the number of processes per column and per row for the grid
149     if(argc>2){
150         P_grid[X_DIR] = atoi(argv[1]);
151         P_grid[Y_DIR] = atoi(argv[2]);
152         if(P_grid[X_DIR] * P_grid[Y_DIR] != P){
153             Debug("ERROR Proces grid dimensions do not match with P ", 1);
154         }
155         #ifdef SOR
156         if (argc>3)
157         {
158             // get sor from args
159             sor_omega = atof(argv[3]);
160             printf("Set sor_omega over argv to %.4f\n", sor_omega);
161         }
162         #endif
163         #ifdef SKIP_EXCHANGE
164         if (argc > 4)
165         {
166             skip_exchange = atoi(argv[4]);
167             printf("Set skip_exchange over argv to %zu\n", skip_exchange);
168         }
169         else{
170             skip_exchange = 1;
171             printf("Set skip_exchange to default value 1\n");
172         }
173         #endif
174     }
175     else{
176         Debug("ERROR Wrong parameter input",1);
177     }
178
179     // Create process topology (2D grid)
180     wrap_around[X_DIR] = 0;
181     wrap_around[Y_DIR] = 0;
182     reorder = 1; //reorder process ranks
183
184     // create grid_comm
185     int ret = MPI_Cart_create(MPI_COMM_WORLD, 2, P_grid, wrap_around, reorder, &grid_comm);
186     if (ret != MPI_SUCCESS){
187         Debug("ERROR: MPI_Cart_create failed",1);
188     }
189     //get new rank and cartesian coords of this proc
190     MPI_Comm_rank(grid_comm, &proc_rank);
191     MPI_Cart_coords(grid_comm, proc_rank, 2, proc_coord);
192     printf("(i) (x,y)=(%i,%i)\n", proc_rank, proc_coord[X_DIR], proc_coord[Y_DIR]);
193     //calc neighbours
194     // MPI_Cart_shift(grid_comm, Y_DIR, 1, &proc_bottom, &proc_top);
195     MPI_Cart_shift(grid_comm, Y_DIR, 1, &proc_top, &proc_bottom);
196     MPI_Cart_shift(grid_comm, X_DIR, 1, &proc_left, &proc_right);
197     printf("(i) top %i, right %i, bottom %i, left %i\n", proc_rank, proc_top,
198     proc_right, proc_bottom, proc_left);
199 }
200 void Setup_Grid()
201 {
202     int x, y, s;
203     double source_x, source_y, source_val;
204     FILE *f;
205
206     Debug("Setup_Subgrid", 0);
207

```

```

208     if(proc_rank == 0){
209         f = fopen("input.dat", "r");
210         if (f == NULL){
211             Debug("Error opening input.dat", 1);
212         }
213         fscanf(f, "nx: %i\n", &gridsize[X_DIR]);
214         fscanf(f, "ny: %i\n", &gridsize[Y_DIR]);
215         fscanf(f, "precision goal: %lf\n", &precision_goal);
216         fscanf(f, "max iterations: %i\n", &max_iter);
217     }
218     MPI_Bcast(&gridsize, 2, MPI_INT, 0, grid_comm);
219     MPI_Bcast(&precision_goal, 1, MPI_DOUBLE, 0, grid_comm);
220     MPI_Bcast(&max_iter, 1, MPI_INT, 0, grid_comm);
221     hx = 1 / (double)gridsize[X_DIR];
222     hy = 1 / (double)gridsize[Y_DIR];
223
224     /* Calculate dimensions of local subgrid */ ///! We do that later now!
225     // dim[X_DIR] = gridsize[X_DIR] + 2;
226     // dim[Y_DIR] = gridsize[Y_DIR] + 2;
227
228     ///! Step 7
229     int upper_offset[2] = {0,0};
230     // Calculate top left corner coordinates of local grid
231     offset[X_DIR] = gridsize[X_DIR] * proc_coord[X_DIR] / P_grid[X_DIR];
232     offset[Y_DIR] = gridsize[Y_DIR] * proc_coord[Y_DIR] / P_grid[Y_DIR];
233     upper_offset[X_DIR] = gridsize[X_DIR] * (proc_coord[X_DIR] + 1) / P_grid[X_DIR];
234     upper_offset[Y_DIR] = gridsize[Y_DIR] * (proc_coord[Y_DIR] + 1) / P_grid[Y_DIR];
235
236     // dimensions of local grid
237     dim[X_DIR] = upper_offset[X_DIR] - offset[X_DIR];
238     dim[Y_DIR] = upper_offset[Y_DIR] - offset[Y_DIR];
239     // Add space for rows/columns of neighboring grid
240     dim[X_DIR] += 2;
241     dim[Y_DIR] += 2;
242     ///! Step 7 end
243
244     /* allocate memory */
245     if ((phi = malloc(dim[X_DIR] * sizeof(*phi))) == NULL){
246         Debug("Setup_Subgrid : malloc(phi) failed", 1);
247     }
248     if ((source = malloc(dim[X_DIR] * sizeof(*source))) == NULL){
249         Debug("Setup_Subgrid : malloc(source) failed", 1);
250     }
251     if ((phi[0] = malloc(dim[Y_DIR] * dim[X_DIR] * sizeof(**phi))) == NULL){
252         Debug("Setup_Subgrid : malloc(*phi) failed", 1);
253     }
254     if ((source[0] = malloc(dim[Y_DIR] * dim[X_DIR] * sizeof(**source))) == NULL){
255         Debug("Setup_Subgrid : malloc(*source) failed", 1);
256     }
257     for (x = 1; x < dim[X_DIR]; x++)
258     {
259         phi[x] = phi[0] + x * dim[Y_DIR];
260         source[x] = source[0] + x * dim[Y_DIR];
261     }
262
263     /* set all values to '0' */
264     for (x = 0; x < dim[X_DIR]; x++){
265         for (y = 0; y < dim[Y_DIR]; y++)
266         {
267             phi[x][y] = 0.0;
268             source[x][y] = 0;
269         }
270     }
271     /* put sources in field */
272     do{
273         if (proc_rank==0)
274         {
275             s = fscanf(f, "source: %lf %lf %lf\n", &source_x, &source_y, &source_val);
276         }
277         MPI_Bcast(&s, 1, MPI_INT, 0, grid_comm);
278         if (s==3){
279             MPI_Bcast(&source_x, 1, MPI_DOUBLE, 0, grid_comm);
280             MPI_Bcast(&source_y, 1, MPI_DOUBLE, 0, grid_comm);

```

```

281     MPI_Bcast(&source_val, 1, MPI_DOUBLE, 0, grid_comm);
282     x = source_x * gridsize[X_DIR];
283     y = source_y * gridsize[Y_DIR];
284     x = x + 1 - offset[X_DIR]; // Step 7 --> local grid transform
285     y = y + 1 - offset[Y_DIR]; // Step 7 --> local grid transform
286     if(x > 0 && x < dim[X_DIR] - 1 && y > 0 && y < dim[Y_DIR] - 1){ // check if in local
grid
287         phi[x][y] = source_val;
288         source[x][y] = 1;
289     }
290 }
291 }
292 while (s==3);
293
294 if(proc_rank==0){
295     fclose(f);
296 }
297 }
298
299 void Setup_MPI_Datatypes()
300 {
301     Debug("Setup_MPI_Datatypes",0);
302
303     // vertical data exchange (Y_Dir)
304     MPI_Type_vector(dim[X_DIR] - 2, 1, dim[Y_DIR], MPI_DOUBLE, &border_type[Y_DIR]);
305     // horizontal data exchange (X_Dir)
306     MPI_Type_vector(dim[Y_DIR] - 2, 1, 1, MPI_DOUBLE, &border_type[X_DIR]);
307
308     MPI_Type_commit(&border_type[Y_DIR]);
309     MPI_Type_commit(&border_type[X_DIR]);
310 }
311
312 int Exchange_Borders()
313 {
314     Debug("Exchange_Borders",0);
315     // top direction
316     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);
317     // bottom direction
318     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);
319     // left direction
320     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);
321     // right direction
322     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);
323     return 1;
324 }
325
326 double Do_Step(int parity)
327 {
328     int x, y;
329     double old_phi, c_ij;
330     double max_err = 0.0;
331
332     #ifdef FAST_DO_STEP_LOOP
333     for (x = 1; x < dim[X_DIR] - 1; x++){
334         for (y = (x + offset[X_DIR]) % 2 == 0 ? 1 : 2; y < dim[Y_DIR] - 1; y += 2){
335             //assert ((x + offset[X_DIR] + y + offset[Y_DIR]) % 2 == parity);
336             if (source[x][y] != 1){
337                 old_phi = phi[x][y];
338                 #ifndef SOR
339                 phi[x][y] = (phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1]) *
0.25;
340                 #endif
341                 #ifdef SOR
342                 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];
343                 phi[x][y] += sor_omega*c_ij;
344                 #endif
345                 if (max_err < fabs(old_phi - phi[x][y])){
346                     max_err = fabs(old_phi - phi[x][y]);

```

```

347     }
348   }
349 }
350 }
351 return max_err;
352 #endif
353
354 #ifndef FAST_DO_STEP_LOOP
355 /* calculate interior of grid */
356 for (x = 1; x < dim[X_DIR] - 1; x++){
357   for (y = 1; y < dim[Y_DIR] - 1; y++){
358     if ((x + offset[X_DIR] + y + offset[Y_DIR]) % 2 == parity && source[x][y] != 1){
359       old_phi = phi[x][y];
360       #ifndef SOR
361         phi[x][y] = (phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1]) *
0.25;
362       #endif
363       #ifdef SOR
364         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];
365         phi[x][y] += sor_omega*c_ij;
366       #endif
367       if (max_err < fabs(old_phi - phi[x][y])){
368         max_err = fabs(old_phi - phi[x][y]);
369       }
370     }
371   }
372 }
373 return max_err;
374 #endif
375 }
376
377 void Solve()
378 {
379   int count = 0;
380   double delta;
381   double global_delta;
382   double delta1, delta2;
383
384   Debug("Solve", 0);
385
386   /* give global_delta a higher value then precision_goal */
387   global_delta = 2 * precision_goal;
388
389   while (global_delta > precision_goal && count < max_iter)
390   {
391     Debug("Do_Step 0", 0);
392     delta1 = Do_Step(0);
393     #ifndef SKIP_EXCHANGE
394     if (count % skip_exchange == 0 && Exchange_Borders()) // use short circuit evaluation
395     #endif
396     #ifndef SKIP_EXCHANGE
397     Exchange_Borders();
398     #endif
399     Debug("Do_Step 1", 0);
400     delta2 = Do_Step(1);
401     #ifndef SKIP_EXCHANGE
402     if (count % skip_exchange == 0 && Exchange_Borders())
403     #endif
404     #ifndef SKIP_EXCHANGE
405     Exchange_Borders();
406     #endif
407     delta = max(delta1, delta2);
408     #ifdef MONITOR_ALLREDUCE
409     double time_ = MPI_Wtime();
410     #endif
411     #ifdef ALLREDUCE_COUNT
412     if (count % ALLREDUCE_COUNT == 0){
413       MPI_Allreduce(&delta, &global_delta, 1, MPI_DOUBLE, MPI_MAX, grid_comm);
414     }
415     #endif
416     #ifndef ALLREDUCE_COUNT
417     MPI_Allreduce(&delta, &global_delta, 1, MPI_DOUBLE, MPI_MAX, grid_comm);

```

```

418     #endif
419     #ifdef MONITOR_ALLREDUCE
420     all_reduce_time += MPI_Wtime() - time_;
421     #endif
422     #ifdef MONITOR_ERROR
423     if (proc_rank == 0)
424     {
425         errors[count] = global_delta;
426     }
427     #endif
428     count++;
429 }
430
431 printf("(%i) Number of iterations : %i\n", proc_rank, count);
432 #ifdef MONITOR_ALLREDUCE
433 printf("(%i) Allreduce time: %14.6f\n", proc_rank, all_reduce_time);
434 #endif
435 }
436
437 double* get_Global_Grid()
438 {
439     Debug("get_Global_Grid", 0);
440     //!! DEBUG only
441     for (size_t i = 0; i < dim[X_DIR]; i++)
442     {
443         for (size_t j = 0; j < dim[Y_DIR]; j++)
444         {
445             phi[i][j] = proc_rank;
446         }
447     }
448 }
449
450 // only process 0 needs to store all data!
451 double* global_phi = NULL;
452 if (proc_rank == 0) {
453     global_phi = malloc(gridsize[X_DIR] * gridsz[e[Y_DIR] * sizeof(double));
454     if (global_phi == NULL) {
455         Debug("get_Global_Grid : malloc(global_phi) failed", 1);
456     }
457 }
458
459 // copy own part into buffer - flatten!
460 size_t buf_size = (dim[X_DIR] - 2) * (dim[Y_DIR] - 2) * sizeof(double);
461 double* local_phi = malloc(buf_size);
462 int idx = 0;
463 for (int x = 1; x < dim[X_DIR] - 1; x++) {
464     for (int y = 1; y < dim[Y_DIR] - 1; y++) {
465         local_phi[idx++] = phi[x][y];
466     }
467 }
468 printf("I'm proc %d and i have a buffer of size %zu\n", proc_rank, buf_size);
469
470
471 // only proc 0 needs sendcounts and displacements for the gather operation
472 int* sendcounts = NULL;
473 int* displs = NULL;
474 if (proc_rank == 0) {
475     sendcounts = malloc(P * sizeof(int));
476     displs = malloc(P * sizeof(int));
477
478     // size and offset of different subgrids
479     //!! Note that this only works if every process has the same subgrid
480     if (gridsize[X_DIR] % P_grid[X_DIR] != 0 || gridsz[e[Y_DIR] % P_grid[Y_DIR] != 0)
481     {
482         Debug("!!!A grid dimension is not a multiple of the P_grid in this direction!", 1)
483     }
484
485     int subgrid_width = gridsz[e[X_DIR] / P_grid[X_DIR];
486     int subgrid_height = gridsz[e[Y_DIR] / P_grid[Y_DIR];
487     for (int px = 0; px < P_grid[X_DIR]; px++) {
488         for (int py = 0; py < P_grid[Y_DIR]; py++) {
489             int rank = px * P_grid[Y_DIR] + py;

```



```

490         sendcounts[rank] = subgrid_width * subgrid_height;
491         displs[rank] = (px * subgrid_width * gridsize[Y_DIR]) + (py * subgrid_height);
492     }
493 }
494 }
495 Debug("get_Global_Grid : MPI_Gatherv", 0);
496 ///! TODO this Gatherv does something wrong - all local grids are alright!!!
497 MPI_Gatherv(local_phi, (dim[X_DIR] - 2) * (dim[Y_DIR] - 2), MPI_DOUBLE, global_phi,
498             sendcounts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
499
500 free(local_phi);
501 if (proc_rank == 0) {
502     free(sendcounts);
503     free(displs);
504 }
505
506 return global_phi;
507 }
508
509 void Write_Grid_global(){
510     int x, y;
511     FILE *f;
512     char filename[40]; //seems danagerous to use a static buffer but let's go with the steps
513     sprintf(filename, "output_MPI_global%i.dat", proc_rank);
514     if ((f = fopen(filename, "w")) == NULL){
515         Debug("Write_Grid : fopen failed", 1);
516     }
517
518     Debug("Write_Grid", 0);
519
520     for (x = 1; x < dim[X_DIR]-1; x++){
521         for (y = 1; y < dim[Y_DIR]-1; y++){
522             int x_glob = x + offset[X_DIR];
523             int y_glob = y + offset[Y_DIR];
524             fprintf(f, "%i %i %f\n", x_glob, y_glob, phi[x][y]);
525         }
526     }
527     fclose(f);
528 }
529
530 void Write_Grid()
531 {
532     double* global_phi = get_Global_Grid();
533     if(proc_rank != 0){
534         assert (global_phi == NULL);
535         return;
536     }
537     int x, y;
538     FILE *f;
539     char filename[40]; //seems danagerous to use a static buffer but let's go with the steps
540     sprintf(filename, "output_MPI%i.dat", proc_rank);
541     if ((f = fopen(filename, "w")) == NULL){
542         Debug("Write_Grid : fopen failed", 1);
543     }
544
545     Debug("Write_Grid", 0);
546
547     for (x = 0; x < gridsize[X_DIR]; x++){
548         for (y = 0; y < gridsize[Y_DIR]; y++){
549             fprintf(f, "%i %i %f\n", x+1, y+1, global_phi[x*gridsize[Y_DIR] + y]);
550         }
551     }
552     fclose(f);
553     free(global_phi);
554 }
555
556 void Clean_Up()
557 {
558     Debug("Clean_Up", 0);
559
560     free(phi[0]);
561     free(phi);
562     free(source[0]);

```

```

562     free(source);
563     #ifdef MONITOR_ERROR
564     free(errors);
565     #endif
566 }
567 void setup_error_monitor(){
568     if (proc_rank != 0)
569     {
570         return;
571     }
572
573     errors = malloc(sizeof(double)*max_iter);
574 }
575 void write_errors(){
576     if(proc_rank != 0){
577         return;
578     }
579     FILE *f;
580     char filename[40]; //seems dangerous to use a static buffer but let's go with the steps
581     sprintf(filename, "errors_MPI.dat");
582     if ((f = fopen(filename, "w")) == NULL){
583         Debug("Write_Errors : fopen failed", 1);
584     }
585
586     Debug("Write_Errors", 0);
587
588     for (size_t i = 0; i < max_iter; ++i)
589     {
590         fprintf(f, "%f\n", errors[i]);
591     }
592     fclose(f);
593 }
594 int main(int argc, char **argv)
595 {
596     MPI_Init(&argc, &argv);
597     Setup_Proc_Grid(argc,argv); // was earlier MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank);
598     start_timer();
599
600     Setup_Grid();
601     Setup_MPI_Datatypes();
602
603     #ifdef SOR
604     if (proc_rank == 0)
605     {
606         printf("SOR using omega: %.5f\n", sor_omega);
607     }
608     #endif
609     #ifdef MONITOR_ERROR
610     setup_error_monitor();
611     #endif
612
613     Solve();
614     #ifdef MONITOR_ERROR
615     write_errors();
616     #endif
617     // Write_Grid();
618     Write_Grid_global();
619     print_timer();
620
621     Clean_Up();
622     MPI_Finalize();
623     return 0;
624 }

```