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

```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:

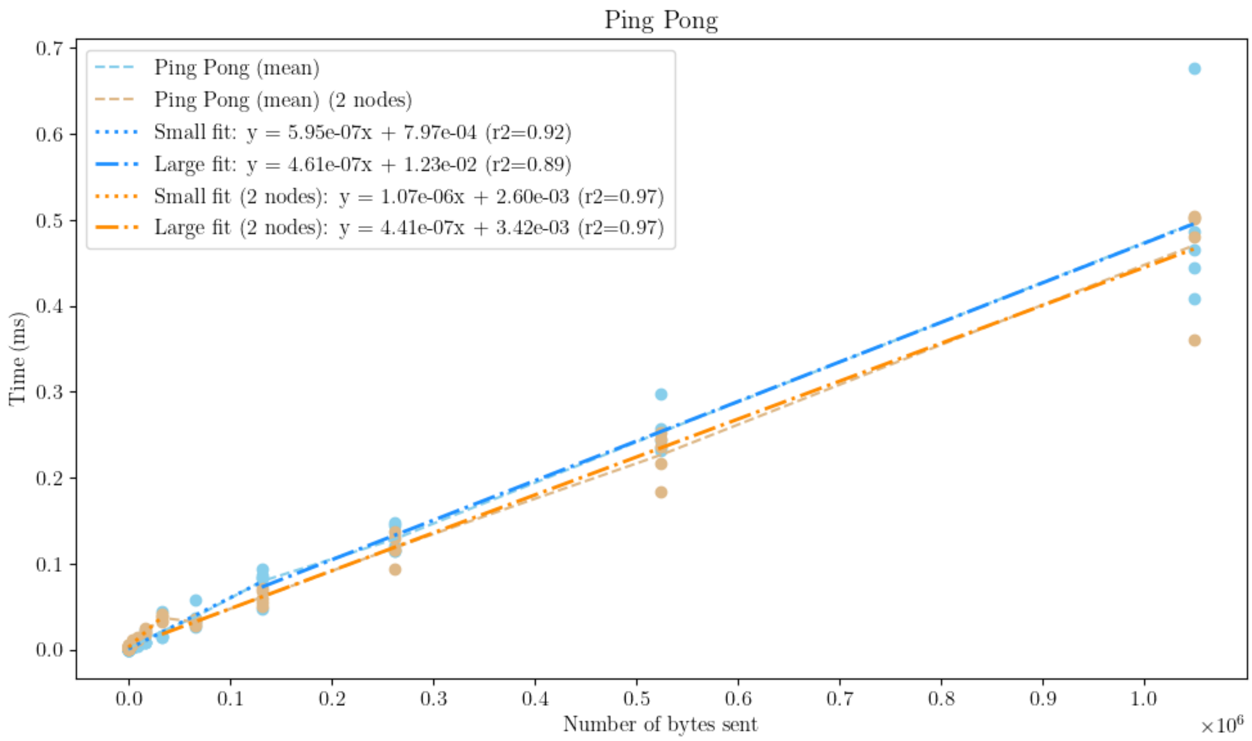


Figure 1: Ping Pong: Number of bytes sent vs. average time taken from 1000 pairs of send / receive. 5 runs shown for each size as scatter plot. Mean of these 5 runs shown as line. Blue small fit includes all data points up to 131072 bytes, blue large from there. Red small fit includes all data points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

| **Run Type** | **Data Size** | **Fit Equation** | **Rš 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 Rš 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 Rš 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:

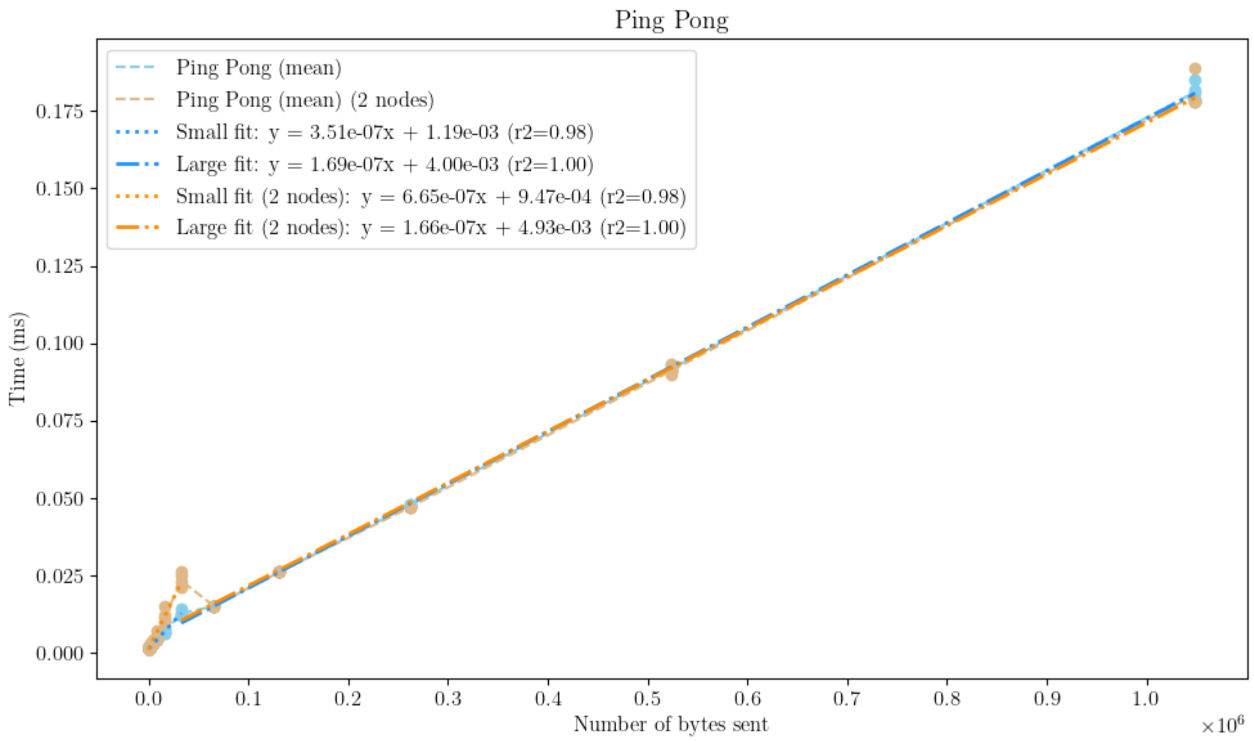


Figure 2: Ping Pong with MPI_SendRecv: Number of bytes sent vs. average time taken from 1000 pairs of send / receive. 5 runs shown for each size as scatter plot. Mean of these 5 runs shown as line. Blue small fit includes all data points up to 32768 bytes, blue large from there. Red small fit includes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

```c
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 $2000 \times 2000$. 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

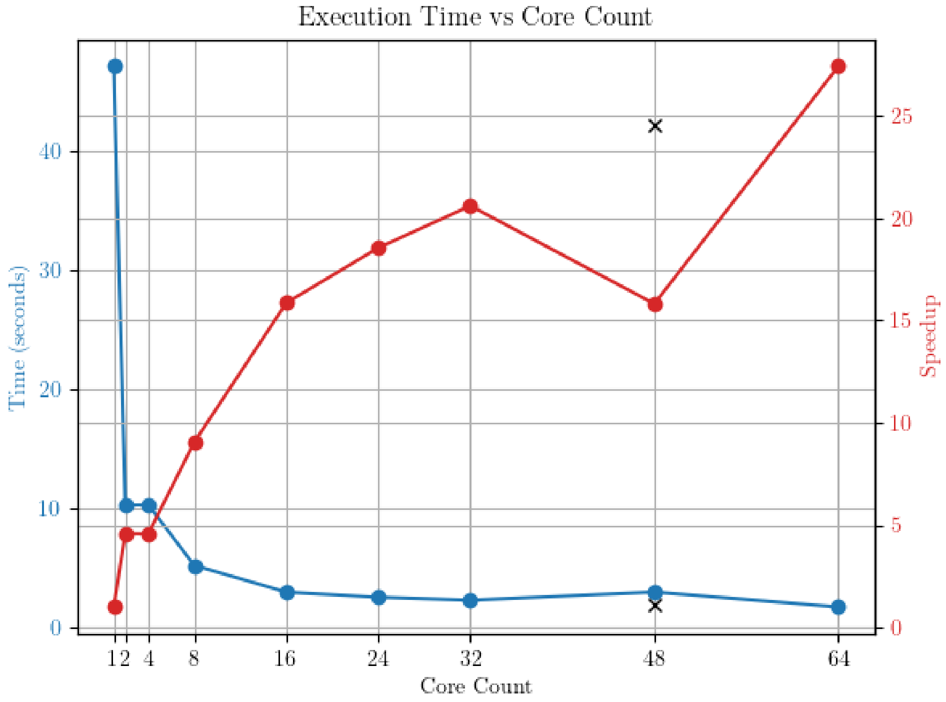


Figure 3: Speedup vs CPU Count
Black × marks the average of the rerun for $n = 48$.

**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.
TODO: **Data locality?**

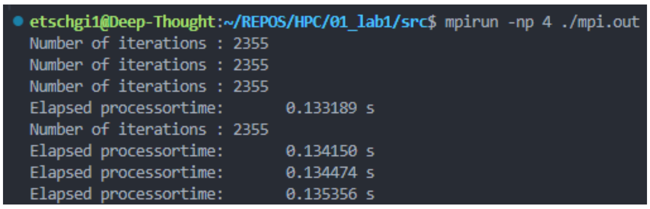
# 1 Poisson solver

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.

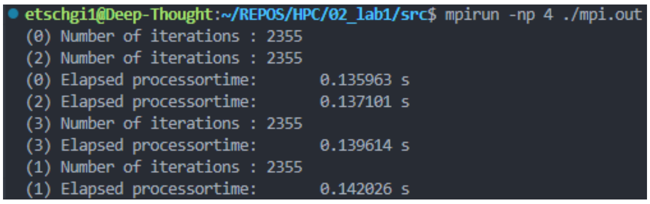
1. **Step:** After adding MPI_Init and MPI_Finalize, we can run the program with multiple processes. We can see that the program runs with 4 processes in Figure 4 via the quadrupeled output.



```
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
 Elapsed processortime:       0.133189 s
 Number of iterations : 2355
 Elapsed processortime:       0.134150 s
 Elapsed processortime:       0.134474 s
 Elapsed processortime:       0.135356 s
```

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

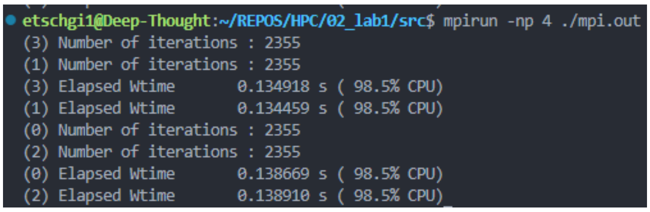
2. **Step:** To see which process is doing what, I included the rank of the process for the print statements as shown in Figure 5.



```
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 processortime:       0.135963 s
 (2) Elapsed processortime:       0.137101 s
 (3) Number of iterations : 2355
 (3) Elapsed processortime:       0.139614 s
 (1) Number of iterations : 2355
 (1) Elapsed processortime:       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.

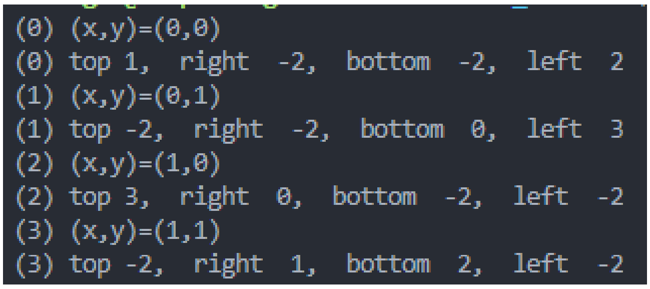


```
etschgi1@Deep-Thought:~/REPOS/HPC/02_lab1/src$ mpirun -np 4 ./mpi.out
 (3) Number of iterations : 2355
 (1) Number of iterations : 2355
 (3) Elapsed Wtime       0.134918 s ( 98.5% CPU)
 (1) Elapsed Wtime       0.134459 s ( 98.5% CPU)
 (0) Number of iterations : 2355
 (2) Number of iterations : 2355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

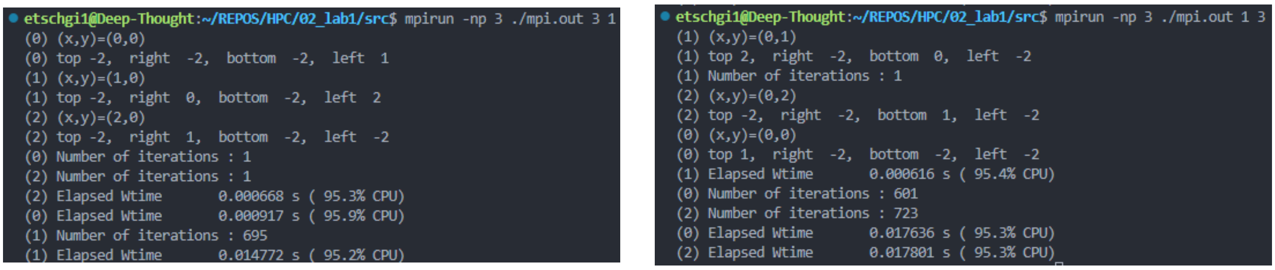


```
(0) (x,y)=(0,0)
(0) top 1,  right  -2,  bottom  -2,  left  2
(1) (x,y)=(0,1)
(1) top -2,  right  -2,  bottom  0,  left  3
(2) (x,y)=(1,0)
(2) top 3,  right  0,  bottom  -2,  left  -2
(3) (x,y)=(1,1)
(3) top -2,  right  1,  bottom  2,  left  -2
```

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.



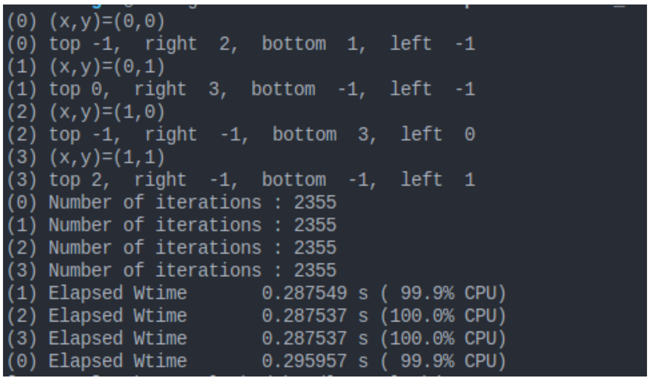
```
● etschgi1@Deep-Thought:~/REPOS/HPC/02_lab1/src$ mpirun -np 3 ./mpi.out 3 1
  (0) (x,y)=(0,0)
  (0) top -2,  right  -2,  bottom  -2,  left  1
  (1) (x,y)=(1,0)
  (1) top -2,  right  0,  bottom  -2,  left  2
  (2) (x,y)=(2,0)
  (2) top -2,  right  1,  bottom  -2,  left  -2
  (0) Number of iterations : 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)
```

```
● etschgi1@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 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.



```
(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

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

to

```
        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:

| | |
|---|---|
| $n$: | 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.

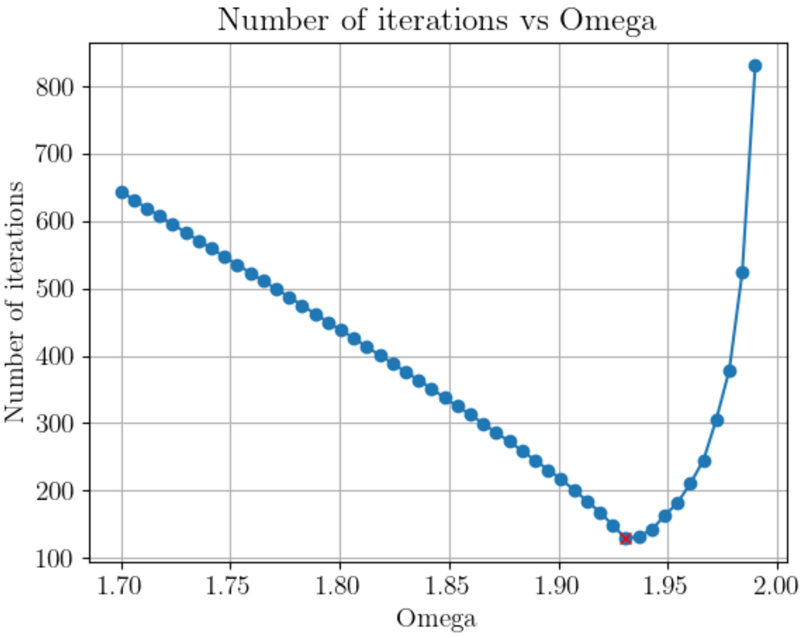


Figure 10: **Optimal $\omega$ for 4 processors on a 4x1 grid**

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.

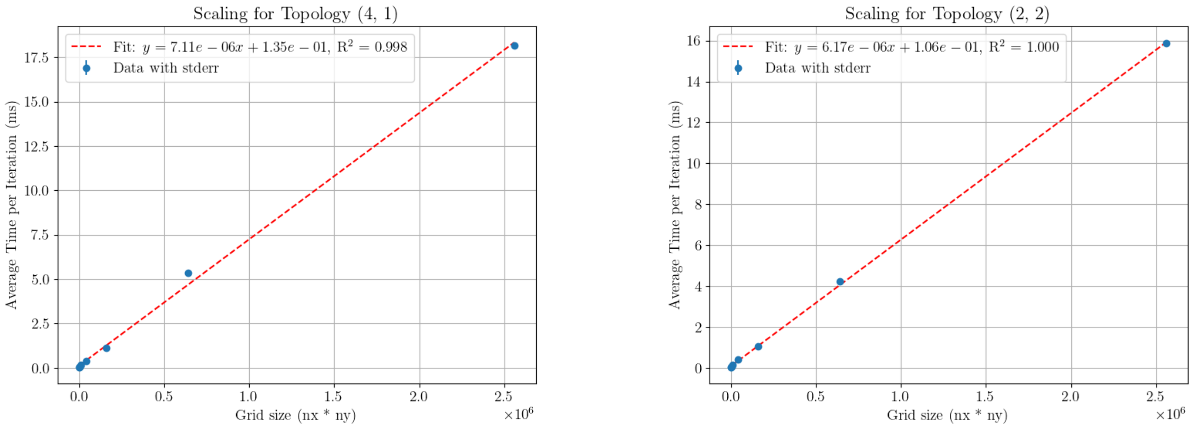


Figure 11: Scaling behavior of the Poisson solver with different grid sizes and processor topologies

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.

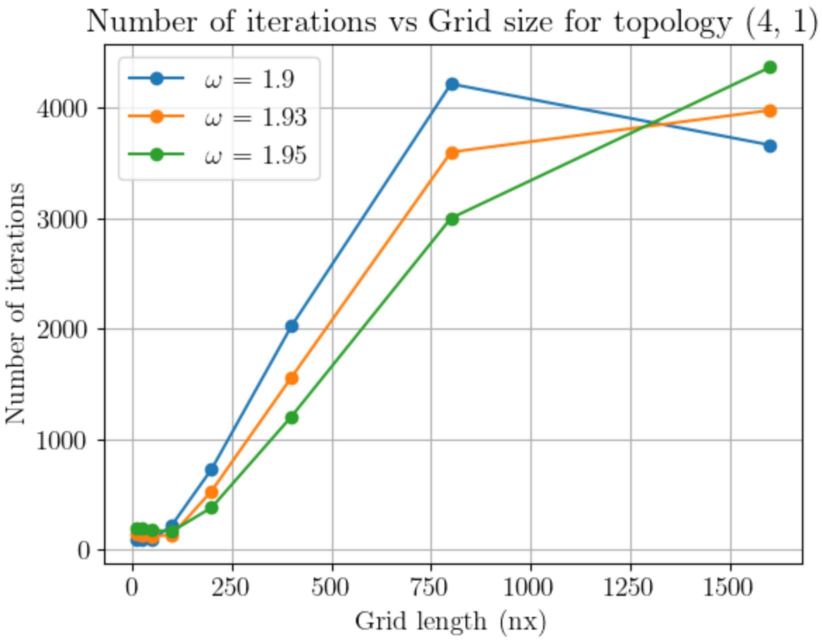


Figure 12: Iterations needed for convergence with different grid side lengths

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 differnetly 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.

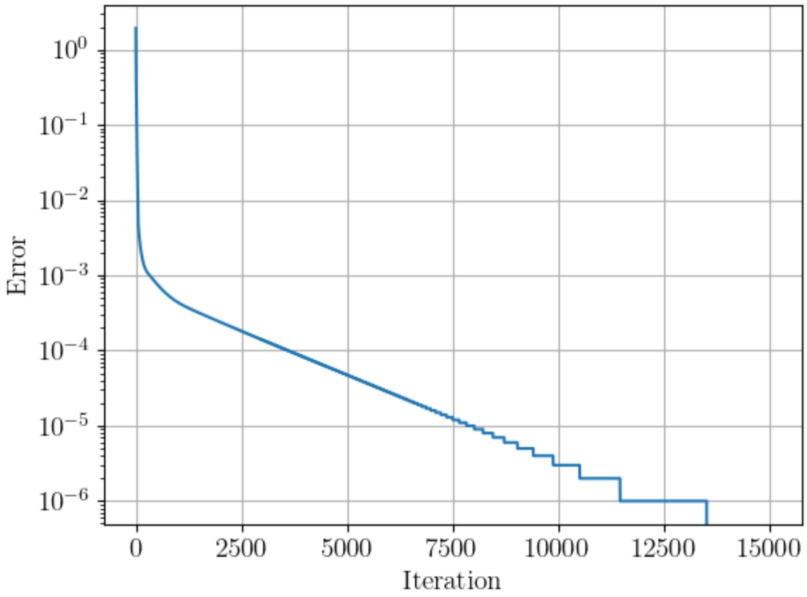


Figure 13: Error as a function of the iteration number

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

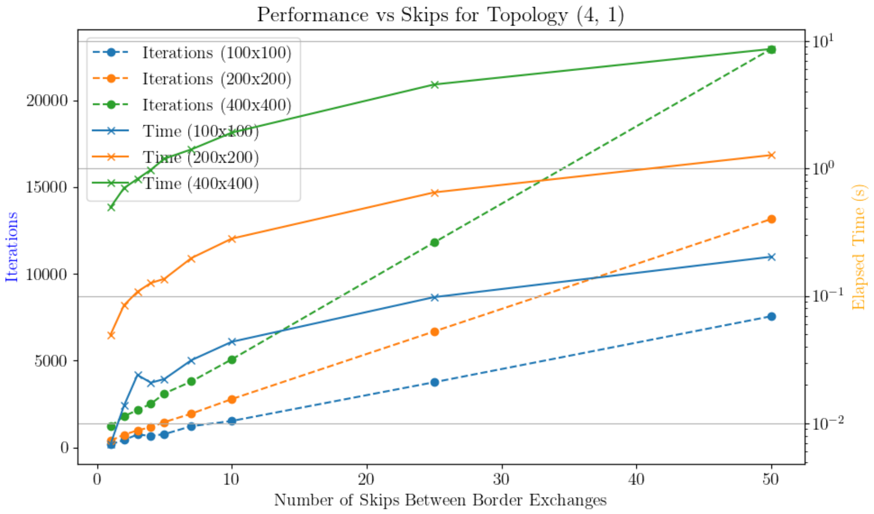


Figure 14: Speedup by reducing border exchanges

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 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.

### 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 a 4 × 1 topology, $\omega = 1.93$ and a $800 \times 800$ grid and get the following times:

# 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:

```c
#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:

```c
#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;
}
```

```c

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

char* getbuffer(MM_input *in, size_t size_of_buffer){
    char* buffer = (char*)malloc(size_of_buffer * sizeof(char));
    if (buffer == 0)
    {
        printf("Buffer couldn’t be allocated.");
        return NULL;
    }
    size_t offset = 0;
    memcpy(buffer + offset, &in->rows, sizeof(size_t));
    offset += sizeof(size_t);
    size_t matrix_size = in->rows * NCA * sizeof(double);
    memcpy(buffer + offset, in->a, matrix_size);
    offset += matrix_size;
    memcpy(buffer + offset, in->b, NCA*NCB*sizeof(double));
    return buffer;
}

MM_input* readbuffer(char* buffer, size_t size_of_buffer){
    MM_input *mm = (MM_input*)malloc(sizeof(MM_input));

    mm->rows = ((size_t*)buffer)[0];
    size_t offset = sizeof(size_t);
    size_t matrix_size =  mm->rows * NCA;
    mm->a = (double*)malloc(sizeof(double)*matrix_size);
    mm->b = (double*)malloc(sizeof(double)*matrix_size);
    memcpy(mm->a, &(buffer[offset]), matrix_size);
    offset += matrix_size;
    memcpy(mm->b, &(buffer[offset]), NCA*NCB*sizeof(double));
    free(buffer);
    return mm;
}


void setupMatrices(double (*a)[NCA], double (*b)[NCB], double (*c)[NCB]){
    for (size_t i = 0; i < NRA; i++) {
        for (size_t j = 0; j < NCA; j++) {
            a[i][j] = i + j;
        }
    }

    for (size_t i = 0; i < NCA; i++) {
        for (size_t j = 0; j < NCB; j++) {
            b[i][j] = i * j;
        }
    }

    for (size_t i = 0; i < NRA; i++) {
        for (size_t j = 0; j < NCB; j++) {
            c[i][j] = 0;
        }
    }
}

double multsum(double* a,double* b_transposed, size_t size){
    double acc = 0;
    for (size_t i = 0; i < size; i++)
    {
        acc += a[i]*b_transposed[i];
    }
    return acc;
}

double productSequential(double *res) {
    // dynamically allocate to not run into stack overflow - usually stacks are
    // 8192 bytes big -> 1024 doubles but we have 1 Mio. per matrix
    double(*a)[NCA] = malloc(sizeof(double) * NRA * NCA);
    double(*b)[NCB] = malloc(sizeof(double) * NCA * NCB);
    double(*c)[NCB] = malloc(sizeof(double) * NRA * NCB);

    /*** Initialize matrices ***/
    setupMatrices(a,b,c);

    /* Parallelize the computation of the following matrix-matrix
   multiplication. How to partition and distribute the initial matrices, the
   work, and collecting final results.
    */
    // multiply
    double start = MPI_Wtime();
    for (size_t i = 0; i < NRA; i++) {
        for (size_t j = 0; j < NCB; j++) {
            for (size_t k = 0; k < NCA; k++) {
                res[i * NCB + j] += a[i][k] * b[k][j];
            }
        }
    }
    /*  perform time measurement. Always check the correctness of the parallel
       results by printing a few values of c[i][j] and compare with the
       sequential output.
    */
    double time = MPI_Wtime()-start;
    free(a);
    free(b);
    free(c);
    return time;
}

double splitwork(double* res, size_t num_workers){
    if (num_workers == 0) // sadly noone will help me :((
    {
        printf("Run sequential!\n");
        return productSequential(res);
    }

    double(*a)[NCA] = malloc(sizeof(double) * NRA * NCA);
    double(*b)[NCB] = malloc(sizeof(double) * NCA * NCB);
    double(*c)[NCB] = malloc(sizeof(double) * NRA * NCB);
    // Transpose matrix b to make accessing columns easier - in row major way - better cache performance
    setupMatrices(a,b,c);

    double start_time = MPI_Wtime();
    double (*b_transposed)[NCA] = malloc(sizeof(double) * NCA * NCB);
    for (size_t i = 0; i < NCA; i++) {
        for (size_t j = 0; j < NCB; j++) {
            b_transposed[j][i] = b[i][j];
        }
    }

    /*** Initialize matrices ***/
    // given number of workers I'll split
    size_t rows_per_worker = NRA / (num_workers+1); //takes corresponding columns from other matrix
    printf("rows per worker: %zu\n", rows_per_worker);
    size_t row_end_first = NRA - rows_per_worker*num_workers;
    printf("first gets most: %zu\n", row_end_first);

    //setup requests
    MPI_Request requests[num_workers];
    MM_input *data_first = (MM_input*)malloc(sizeof(MM_input));
    data_first->rows = row_end_first;
    data_first->a = (double*)a; //they both start of with no offset!
    data_first->b = (double*)b_transposed;
    size_t total_size = sizeof(size_t) + (data_first->rows * NCA)*sizeof(double)+SIZE_OF_B;
    char* buffer = getbuffer(data_first, total_size);     //first one

    // Tag is just nr-cpu -1
    MPI_Isend(buffer, total_size, MPI_CHAR, 1, 0,MPI_COMM_WORLD, &requests[0]);
    free(data_first);
    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);
```

```c
    double start = MPI_Wtime();
    int count;
    MPI_Get_count(&status, MPI_CHAR, &count);
    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);

    MM_input *mm = (MM_input*)malloc(sizeof(MM_input));
    mm->a = (double*)&buffer[size_of_meta];
    mm->b = (double*)&buffer[size_of_meta+size_of_a];

    double *res =(double*)malloc(sizeof(double)*rows_per_worker*NCB);

    size_t offset = 0;
    for (size_t row = 0; row < rows_per_worker; row++)
    {
        for (size_t col = 0; col < NCB; col++)
        {
            res[row * NCB + col] = multsum(mm->a+offset, (((double*)mm->b)+col*NCA), NCA);
        }
        offset += NCA;
    }
    MPI_Send(res, rows_per_worker*NCB, MPI_DOUBLE, 0,rank-1, MPI_COMM_WORLD);
    printf("[%d] sent res home\n",rank);
    free(res);
    return MPI_Wtime() - start;
}

int main(int argc, char *argv[]) {
    int tid, nthreads;
    /* for simplicity, set NRA=NCA=NCB=N  */
    // Initialize MPI, find out MPI communicator size and process rank
    int myRank, numProcs;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    int num_Workers = numProcs-1;
    if (argc > 1 && strcmp(argv[1], "parallel") == 0) {
        // Variables for the process rank and number of processes
       if (myRank == 0) {
            printf("Run parallel!\n");
            double *truth = malloc(sizeof(double) * NRA * NCB);
            double time = productSequential(truth);
            printf("Computed reference results in %.6f s\n", time);
            printf("Hello from master! - I have %d workers!\n", num_Workers);
            // send out work
            double *res = malloc(sizeof(double)*NRA*NCB);
            time = splitwork(res, num_Workers);
            if (checkResult(res, truth, NCB, NRA)) {
                printf("Matrices do not match!!!\n");
                return 1;
            }
            printf("Matrices match (parallel [eps %.10f])! - took: %.6f s\n", EPS, time);
            free(truth);
            free(res);
        } else {
            double time = work(myRank, num_Workers);
            printf("Worker bee %d took %.6f s (after recv) for my work\n", myRank, time);
        }

    } else  // run sequantial
    {
        printf("Run sequantial!\n");
        double *res = malloc(sizeof(double) * NRA * NCB);
        double time = productSequential(res);
        if (checkResult(res, res, NCB, NRA)) {
            printf("Matrices do not match!!!\n");
            return 1;
        }
        printf("Matrices match (sequantial-trivial)! - took: %.6f s\n", time);
        free(res);
    }

    MPI_Finalize();
    return 0;
}
```

# Appendix - Poisson solver

The parallel Poisson solver used the following code:

```
/*
 * 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 exchange_time = 0;
#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;
```

```c
double wtime;
int proc_coord[2]; // coords of current process in processgrid
int proc_top, proc_right, proc_bottom, proc_left; // ranks of neighboring procs
// step 7
int offset[2] = {0,0};
// step 8
MPI_Datatype border_type[2];

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

/* local grid related variables */
double **phi;     /* grid */
int **source;     /* TRUE if subgrid element is a source */
int dim[2];     /* grid dimensions */

void Setup_Grid();
double Do_Step(int parity);
void Solve();
void Write_Grid();
void Clean_Up();
void Debug(char *mesg, int terminate);
void start_timer();
void resume_timer();
void stop_timer();
void print_timer();

void start_timer()
{
    if (!timer_on){
        MPI_Barrier(grid_comm);
        ticks = clock();
        wtime = MPI_Wtime();
        timer_on = 1;
    }
}

void resume_timer()
{
    if (!timer_on){
        ticks = clock()  -  ticks;
        wtime = MPI_Wtime() - wtime;
        timer_on = 1;
    }
}

void stop_timer()
{
    if (timer_on){
        ticks = clock()  -  ticks;
        wtime = MPI_Wtime() - wtime;
        timer_on = 0;
    }
}

void print_timer()
{
    if (timer_on){
        stop_timer();
        printf("(%i) Elapsed Wtime %14.6f s (%5.1f%% CPU)\n", proc_rank, wtime, 100.0 * ticks * (1.0 / CLOCKS_PER_SEC) / wtime);
        resume_timer();
    }
    else{
        printf("(%i) Elapsed Wtime %14.6f s (%5.1f%% CPU)\n", proc_rank, wtime, 100.0 * ticks * (1.0 / CLOCKS_PER_SEC) / wtime);
    }
}

void Debug(char *mesg, int terminate)
{
    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;
```

```c
    FILE *f;

    Debug("Setup_Subgrid", 0);

    if(proc_rank == 0){
        f = fopen("input.dat", "r");
        if (f == NULL){
            Debug("Error opening input.dat", 1);
        }
        fscanf(f, "nx: %i\n", &gridsize[X_DIR]);
        fscanf(f, "ny: %i\n", &gridsize[Y_DIR]);
        fscanf(f, "precision goal: %lf\n", &precision_goal);
        fscanf(f, "max iterations: %i\n", &max_iter);
    }
    MPI_Bcast(&gridsize, 2, MPI_INT, 0, grid_comm);
    MPI_Bcast(&precision_goal, 1, MPI_DOUBLE, 0, grid_comm);
    MPI_Bcast(&max_iter, 1, MPI_INT, 0, grid_comm);
    hx = 1 / (double)gridsize[X_DIR];
    hy = 1 / (double)gridsize[Y_DIR];

    /* Calculate dimensions of local subgrid */ //! We do that later now!
    // dim[X_DIR] = gridsize[X_DIR] + 2;
    // dim[Y_DIR] = gridsize[Y_DIR] + 2;

    //! Step 7
    int upper_offset[2] = {0,0};
    // Calculate top left corner cordinates of local grid
    offset[X_DIR] = gridsize[X_DIR] * proc_coord[X_DIR] / P_grid[X_DIR];
    offset[Y_DIR] = gridsize[Y_DIR] * proc_coord[Y_DIR] / P_grid[Y_DIR];
    upper_offset[X_DIR] = gridsize[X_DIR] * (proc_coord[X_DIR] + 1) / P_grid[X_DIR];
    upper_offset[Y_DIR] = gridsize[Y_DIR] * (proc_coord[Y_DIR] + 1) / P_grid[Y_DIR];

    // dimensions of local grid
    dim[X_DIR] = upper_offset[X_DIR] - offset[X_DIR];
    dim[Y_DIR] = upper_offset[Y_DIR] - offset[Y_DIR];
    // Add space for rows/columns of neighboring grid
    dim[X_DIR] += 2;
    dim[Y_DIR] += 2;
    //! Step 7 end

    /* allocate memory */
    if ((phi = malloc(dim[X_DIR] * sizeof(*phi))) == NULL){
        Debug("Setup_Subgrid : malloc(phi) failed", 1);
    }
    if ((source = malloc(dim[X_DIR] * sizeof(*source))) == NULL){
        Debug("Setup_Subgrid : malloc(source) failed", 1);
    }
    if ((phi[0] = malloc(dim[Y_DIR] * dim[X_DIR] * sizeof(**phi))) == NULL){
        Debug("Setup_Subgrid : malloc(*phi) failed", 1);
    }
    if ((source[0] = malloc(dim[Y_DIR] * dim[X_DIR] * sizeof(**source))) == NULL){
        Debug("Setup_Subgrid : malloc(*source) failed", 1);
    }
    for (x = 1; x < dim[X_DIR]; x++)
    {
        phi[x] = phi[0] + x * dim[Y_DIR];
        source[x] = source[0] + x * dim[Y_DIR];
    }

    /* set all values to ’0’ */
    for (x = 0; x < dim[X_DIR]; x++){
        for (y = 0; y < dim[Y_DIR]; y++)
        {
            phi[x][y] = 0.0;
            source[x][y] = 0;
        }
    }
    /* put sources in field */
    do{
        if (proc_rank==0)
        {
            s = fscanf(f, "source: %lf %lf %lf\n", &source_x, &source_y, &source_val);
        }
        MPI_Bcast(&s, 1, MPI_INT, 0, grid_comm);
        if (s==3){
            MPI_Bcast(&source_x, 1, MPI_DOUBLE, 0, grid_comm);
            MPI_Bcast(&source_y, 1, MPI_DOUBLE, 0, grid_comm);
            MPI_Bcast(&source_val, 1, MPI_DOUBLE, 0, grid_comm);
            x = source_x * gridsize[X_DIR];
            y = source_y * gridsize[Y_DIR];
            x = x + 1 - offset[X_DIR]; // Step 7 --> local grid transform
            y = y + 1 - offset[Y_DIR]; // Step 7 --> local grid transform
            if(x > 0 && x < dim[X_DIR] -1 && y > 0 && y < dim[Y_DIR]-1){ // check if in local grid
                phi[x][y] = source_val;
                source[x][y] = 1;
            }
        }
    }
    while (s==3);

    if(proc_rank==0){
        fclose(f);
    }
}

void Setup_MPI_Datatypes()
{
    Debug("Setup_MPI_Datatypes",0);

    // vertical data exchange (Y_Dir)
    MPI_Type_vector(dim[X_DIR] - 2, 1, dim[Y_DIR], MPI_DOUBLE, &border_type[Y_DIR]);
    // horizontal data exchange (X_Dir)
    MPI_Type_vector(dim[Y_DIR] - 2, 1, 1, MPI_DOUBLE, &border_type[X_DIR]);

    MPI_Type_commit(&border_type[Y_DIR]);
    MPI_Type_commit(&border_type[X_DIR]);
}

int Exchange_Borders()
{
    #ifdef MONITOR_EXCHANGE_BORDERS
    double time_ = MPI_Wtime();
    #endif
    Debug("Exchange_Borders",0);
    // top direction
    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);
    // bottom direction
    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);
    // left direction
    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);
    // right direction
    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);
    #ifdef MONITOR_EXCHANGE_BORDERS
    exchange_time += MPI_Wtime() - time_;
    #endif
    return 1;
}

double Do_Step(int parity)
{
    int x, y;
    double old_phi, c_ij;
    double max_err = 0.0;

    #ifdef FAST_DO_STEP_LOOP
    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){
                old_phi = phi[x][y];
                #ifndef SOR
                phi[x][y] = (phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1]) * 0.25;
                #endif
                #ifdef SOR
                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];
                phi[x][y] += sor_omega*c_ij;
                #endif
                if (max_err < fabs(old_phi - phi[x][y])){
                    max_err = fabs(old_phi - phi[x][y]);
                }
            }
        }
    }
    return max_err;
    #endif

    #ifndef FAST_DO_STEP_LOOP
    /* calculate interior of grid */
    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){
                old_phi = phi[x][y];
                #ifndef SOR
                phi[x][y] = (phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1]) * 0.25;
                #endif
                #ifdef SOR
                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];
                phi[x][y] += sor_omega*c_ij;
                #endif
                if (max_err < fabs(old_phi - phi[x][y])){
                    max_err = fabs(old_phi - phi[x][y]);
                }
            }
        }
    }
  return max_err;
  #endif
}

void Solve()
{
    int count = 0;
    double delta;
    double global_delta;
    double delta1, delta2;

    Debug("Solve", 0);

    /* give global_delta a higher value then precision_goal */
    global_delta = 2 * precision_goal;

    while (global_delta > precision_goal && count < max_iter)
    {
        Debug("Do_Step 0", 0);
        delta1 = Do_Step(0);
        #ifdef SKIP_EXCHANGE
        if (count % skip_exchange == 0 && Exchange_Borders()) // use short circuit evaluation
        #endif
        #ifndef SKIP_EXCHANGE
        Exchange_Borders();
        #endif
        Debug("Do_Step 1", 0);
        delta2 = Do_Step(1);
        #ifdef SKIP_EXCHANGE
        if (count % skip_exchange == 0 && Exchange_Borders())
        #endif
        #ifndef SKIP_EXCHANGE
        Exchange_Borders();
        #endif
        delta = max(delta1, delta2);
        #ifdef MONITOR_ALLREDUCE
        double time_ = MPI_Wtime();
        #endif
        #ifdef ALLREDUCE_COUNT
        if(count % ALLREDUCE_COUNT == 0){
            MPI_Allreduce(&delta, &global_delta, 1, MPI_DOUBLE, MPI_MAX, grid_comm);
        }
        #endif
        #ifndef ALLREDUCE_COUNT
        MPI_Allreduce(&delta, &global_delta, 1, MPI_DOUBLE, MPI_MAX, grid_comm);
        #endif
        #ifdef MONITOR_ALLREDUCE
        all_reduce_time += MPI_Wtime() - time_;
        #endif
        #ifdef MONITOR_ERROR
        if (proc_rank == 0)
        {
            errors[count] = global_delta;
        }
        #endif
        count++;
    }

    printf("(%i) Number of iterations : %i\n", proc_rank, count);
    #ifdef MONITOR_ALLREDUCE
    printf("(%i) Allreduce time: %14.6f\n", proc_rank, all_reduce_time);
    #endif
    #ifdef MONITOR_EXCHANGE_BORDERS
    printf("(%i) Exchange time: %14.6f\n", proc_rank, exchange_time);
    #endif
}

double* get_Global_Grid()
{
    Debug("get_Global_Grid", 0);
    //!! DEBUG only
    for (size_t i = 0; i < dim[X_DIR]; i++)
    {
        for (size_t j = 0; j < dim[Y_DIR]; j++)
        {
            phi[i][j] = proc_rank;
        }

    }

    // only process 0 needs to store all data!
    double* global_phi = NULL;
    if (proc_rank == 0) {
        global_phi = malloc(gridsize[X_DIR] * gridsize[Y_DIR] * sizeof(double));
        if (global_phi == NULL) {
            Debug("get_Global_Grid : malloc(global_phi) failed", 1);
        }
    }

    // copy own part into buffer - flatten!
    size_t buf_size = (dim[X_DIR] - 2) * (dim[Y_DIR] - 2) * sizeof(double);
    double* local_phi = malloc(buf_size);
    int idx = 0;
    for (int x = 1; x < dim[X_DIR] - 1; x++) {
        for (int y = 1; y < dim[Y_DIR] - 1; y++) {
            local_phi[idx++] = phi[x][y];
        }
    }
    printf("I'm proc %d and i have a buffer of size %zu\n", proc_rank, buf_size);


    // only proc 0 needs sendcounts and displacements for the gatherv operation
    int* sendcounts = NULL;
    int* displs = NULL;
    if (proc_rank == 0) {
        sendcounts = malloc(P * sizeof(int));
        displs = malloc(P * sizeof(int));
```

```c

        // size and offset of different subgrids
        //! Note that this only works if every process has the same subgrid
        if (gridsize[X_DIR] % P_grid[X_DIR] != 0 || gridsize[Y_DIR] % P_grid[Y_DIR] != 0)
        {
            Debug("!!!A grid dimension is not a multiple of the P_grid in this direction!", 1);
        }

        int subgrid_width = gridsize[X_DIR] / P_grid[X_DIR];
        int subgrid_height = gridsize[Y_DIR] / P_grid[Y_DIR];
        for (int px = 0; px < P_grid[X_DIR]; px++) {
            for (int py = 0; py < P_grid[Y_DIR]; py++) {
                int rank = px * P_grid[Y_DIR] + py;
                sendcounts[rank] = subgrid_width * subgrid_height;
                displs[rank] = (px * subgrid_width * gridsize[Y_DIR]) + (py * subgrid_height);
            }
        }
    }
    Debug("get_Global_Grid : MPI_Gatherv", 0);
    //! TODO this Gatherv does something wrong - all local grids are alright!!!
    MPI_Gatherv(local_phi, (dim[X_DIR] - 2) * (dim[Y_DIR] - 2), MPI_DOUBLE, global_phi, sendcounts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    free(local_phi);
    if (proc_rank == 0) {
        free(sendcounts);
        free(displs);
    }

    return global_phi;
}

void Write_Grid_global(){
    int x, y;
    FILE *f;
    char filename[40]; //seems danagerous to use a static buffer but let's go with the steps
    sprintf(filename, "output_MPI_global_%i.dat", proc_rank);
    if ((f = fopen(filename, "w")) == NULL){
        Debug("Write_Grid : fopen failed", 1);
    }

    Debug("Write_Grid", 0);

    for (x = 1; x < dim[X_DIR]-1; x++){
        for (y = 1; y < dim[Y_DIR]-1; y++){
            int x_glob = x + offset[X_DIR];
            int y_glob = y + offset[Y_DIR];
            fprintf(f, "%i %i %f\n", x_glob, y_glob, phi[x][y]);
        }
    }
    fclose(f);
}

void Write_Grid()
{
    double* global_phi = get_Global_Grid();
    if(proc_rank != 0){
        assert (global_phi == NULL);
        return;
    }
    int x, y;
    FILE *f;
    char filename[40]; //seems danagerous to use a static buffer but let's go with the steps
    sprintf(filename, "output_MPI%i.dat", proc_rank);
    if ((f = fopen(filename, "w")) == NULL){
        Debug("Write_Grid : fopen failed", 1);
    }

    Debug("Write_Grid", 0);

    for (x = 0; x < gridsize[X_DIR]; x++){
        for (y = 0; y < gridsize[Y_DIR]; y++){
            fprintf(f, "%i %i %f\n", x+1, y+1, global_phi[x*gridsize[Y_DIR] + y]);
        }
    }
    fclose(f);
    free(global_phi);
}

void Clean_Up()
{
    Debug("Clean_Up", 0);

    free(phi[0]);
    free(phi);
    free(source[0]);
    free(source);
    #ifdef MONITOR_ERROR
    free(errors);
    #endif
}
void setup_error_monitor(){
    if (proc_rank != 0)
    {
        return;
    }

    errors = malloc(sizeof(double)*max_iter);
}
void write_errors(){
    if(proc_rank != 0){
        return;
    }
    FILE *f;
    char filename[40]; //seems danagerous to use a static buffer but let's go with the steps
    sprintf(filename, "errors_MPI.dat");
    if ((f = fopen(filename, "w")) == NULL){
        Debug("Write_Errors : fopen failed", 1);
    }

    Debug("Write_Errors", 0);

    for (size_t i = 0; i < max_iter; ++i)
    {
        fprintf(f, "%f\n", errors[i]);
    }
    fclose(f);
}
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_timer();

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