

DELFT UNIVERSITY OF TECHNOLOGY

INTRODUCTION TO HIGH PERFORMANCE COMPUTING  
WI4049TU

---

# Lab Report

---

*Author:*  
Elias Wachmann (6300421)

November 15, 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:

```
srunc -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<sup>2</sup> values:

Run Type	Data Size	Fit Equation	R <sup>2</sup> 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<sup>2</sup> 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<sup>2</sup> 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:

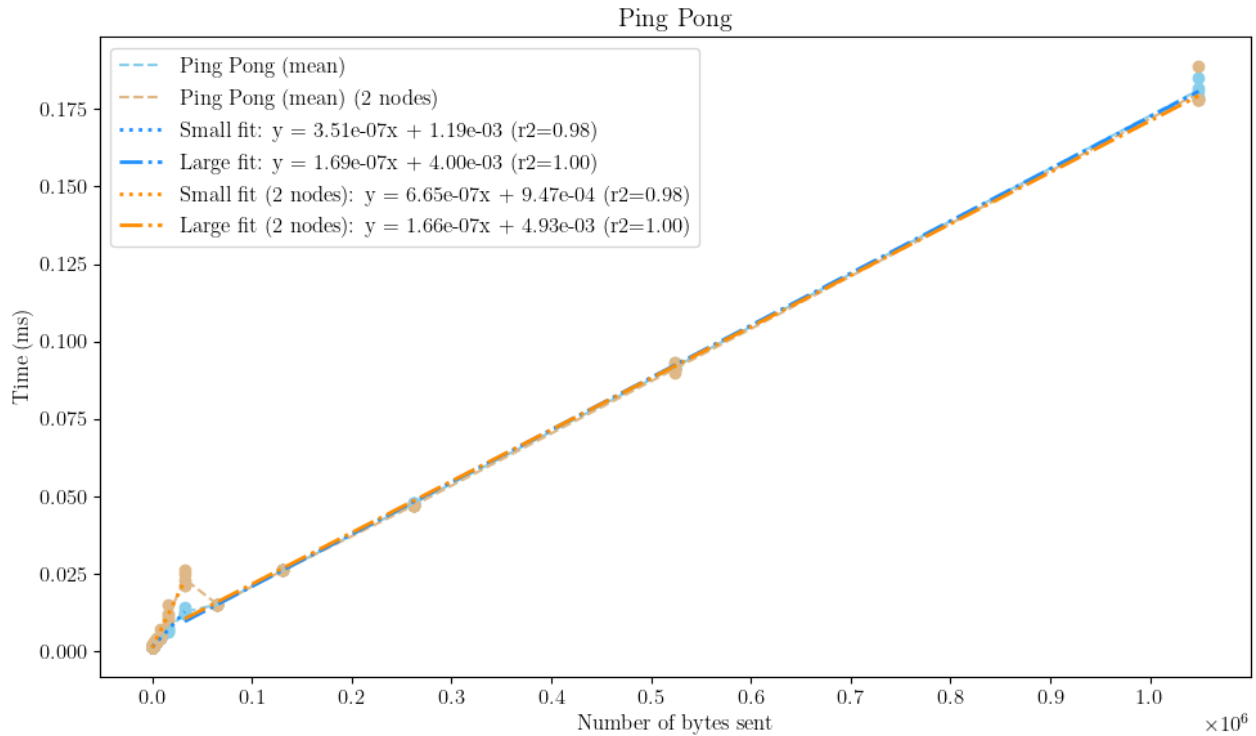


Figure 2: Ping Pong with MPI\_SendRecv: Number of bytes sent vs. average time taken from 1000 pairs of send / receive. 5 runs shown for each size as scatter plot. Mean of these 5 runs shown as line. Blue small fit includes all data points up to 32768 bytes, blue large from there. Red small fit includes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

#### 10. Step:

## 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     offset += NCA;
217 }
218 printf("My c: \n");
219 //wait for rest
220 MPI_Status stats[num_workers];
221 if(MPI_Waitall(num_workers, requests, stats) == MPI_ERR_IN_STATUS){
222     printf("Communication failed!!! - abort\n");
223 }
224 printf(">>>Everything sent and recieved\n");
225
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 enum
18 {
19     X_DIR, Y_DIR
20 };
21
22 /* global variables */
23 int gridsize[2];
24 double precision_goal; /* precision_goal of solution */
25 int max_iter; /* maximum number of iterations allowed */
26 int P; //total number of processes
27 int P_grid[2]; // process grid dimensions
28 MPI_Comm grid_comm; //grid communicator
29 MPI_Status status;
30
31 /* process specific globals*/
32 int proc_rank;
33 double wtime;
34 int proc_coord[2]; // coords of current process in processgrid
35 int proc_top, proc_right, proc_bottom, proc_left; // ranks of neighboring procs
36 // step 7
37 int offset[2] = {0,0};
38 // step 8
39 MPI_Datatype border_type[2];
40
41 /* benchmark related variables */
42 clock_t ticks; /* number of systemticks */
43 int timer_on = 0; /* is timer running? */
44
45 /* local grid related variables */
46 double **phi; /* grid */
47 int **source; /* TRUE if subgrid element is a source */
48 int dim[2]; /* grid dimensions */
49
50 void Setup_Grid();
51 double Do_Step(int parity);
52 void Solve();
53 void Write_Grid();
54 void Clean_Up();
55 void Debug(char *mesg, int terminate);
56 void start_timer();
57 void resume_timer();
58 void stop_timer();
59 void print_timer();
60
61 void start_timer()
62 {
63     if (!timer_on){
64         MPI_Barrier(grid_comm);

```

```

65         ticks = clock();
66         wtime = MPI_Wtime();
67         timer_on = 1;
68     }
69 }
70
71 void resume_timer()
72 {
73     if (!timer_on){
74         ticks = clock() - ticks;
75         wtime = MPI_Wtime() - wtime;
76         timer_on = 1;
77     }
78 }
79
80 void stop_timer()
81 {
82     if (timer_on){
83         ticks = clock() - ticks;
84         wtime = MPI_Wtime() - wtime;
85         timer_on = 0;
86     }
87 }
88
89 void print_timer()
90 {
91     if (timer_on){
92         stop_timer();
93         printf("(i) Elapsed Wtime %14.6f s (%5.1f%% CPU)\n", proc_rank, wtime, 100.0 * ticks
94 * (1.0 / CLOCKS_PER_SEC) / wtime);
95         resume_timer();
96     }
97     else{
98         printf("(i) Elapsed Wtime %14.6f s (%5.1f%% CPU)\n", proc_rank, wtime, 100.0 * ticks
99 * (1.0 / CLOCKS_PER_SEC) / wtime);
100     }
101 }
102
103 void Debug(char *mesg, int terminate)
104 {
105     if (DEBUG || terminate){
106         printf("%s\n", mesg);
107     }
108     if (terminate){
109         exit(1);
110     }
111 }
112
113 void Setup_Proc_Grid(int argc, char **argv){
114     int wrap_around[2];
115     int reorder;
116
117     Debug("My_MPI_Init",0);
118
119     // num of processes
120     MPI_Comm_size(MPI_COMM_WORLD, &P);
121
122     //calculate the number of processes per column and per row for the grid
123     if(argc>2){
124         P_grid[X_DIR] = atoi(argv[1]);
125         P_grid[Y_DIR] = atoi(argv[2]);
126         if(P_grid[X_DIR] * P_grid[Y_DIR] != P){
127             Debug("ERROR Proces grid dimensions do not match with P ", 1);
128         }
129     }
130     else{
131         Debug("ERROR Wrong parameter input",1);
132     }
133
134     // Create process topology (2D grid)
135     wrap_around[X_DIR] = 0;
136     wrap_around[Y_DIR] = 0;
137     reorder = 1; //reorder process ranks

```

```

136
137 // create grid_comm
138 int ret = MPI_Cart_create(MPI_COMM_WORLD, 2, P_grid, wrap_around, reorder, &grid_comm);
139 if (ret != MPI_SUCCESS){
140     Debug("ERROR: MPI_Cart_create failed",1);
141 }
142 //get new rank and cartesian coords of this proc
143 MPI_Comm_rank(grid_comm, &proc_rank);
144 MPI_Cart_coords(grid_comm, proc_rank, 2, proc_coord);
145 printf("(i) (x,y)=(i,i)\n", proc_rank, proc_coord[X_DIR], proc_coord[Y_DIR]);
146 //calc neighbours
147 // MPI_Cart_shift(grid_comm, Y_DIR, 1, &proc_bottom, &proc_top);
148 MPI_Cart_shift(grid_comm, Y_DIR, 1, &proc_top, &proc_bottom);
149 MPI_Cart_shift(grid_comm, X_DIR, 1, &proc_left, &proc_right);
150 printf("(i) top i, right i, bottom i, left i\n", proc_rank, proc_top,
proc_right, proc_bottom, proc_left);
151 }
152
153 void Setup_Grid()
154 {
155     int x, y, s;
156     double source_x, source_y, source_val;
157     FILE *f;
158
159     Debug("Setup_Subgrid", 0);
160
161     if(proc_rank == 0){
162         f = fopen("input.dat", "r");
163         if (f == NULL){
164             Debug("Error opening input.dat", 1);
165         }
166         fscanf(f, "nx: %i\n", &gridsize[X_DIR]);
167         fscanf(f, "ny: %i\n", &gridsize[Y_DIR]);
168         fscanf(f, "precision goal: %lf\n", &precision_goal);
169         fscanf(f, "max iterations: %i\n", &max_iter);
170     }
171     MPI_Bcast(&gridsize, 2, MPI_INT, 0, grid_comm);
172     MPI_Bcast(&precision_goal, 1, MPI_DOUBLE, 0, grid_comm);
173     MPI_Bcast(&max_iter, 1, MPI_INT, 0, grid_comm);
174
175     /* Calculate dimensions of local subgrid */
176     // dim[X_DIR] = gridsize[X_DIR] + 2;
177     // dim[Y_DIR] = gridsize[Y_DIR] + 2;
178
179     ///! Step 7
180     int upper_offset[2] = {0,0};
181     // Calculate top left corner coordinates of local grid
182     offset[X_DIR] = gridsize[X_DIR] * proc_coord[X_DIR] / P_grid[X_DIR];
183     offset[Y_DIR] = gridsize[Y_DIR] * proc_coord[Y_DIR] / P_grid[Y_DIR];
184     upper_offset[X_DIR] = gridsize[X_DIR] * (proc_coord[X_DIR] + 1) / P_grid[X_DIR];
185     upper_offset[Y_DIR] = gridsize[Y_DIR] * (proc_coord[Y_DIR] + 1) / P_grid[Y_DIR];
186
187     // dimensions of local grid
188     dim[X_DIR] = upper_offset[X_DIR] - offset[X_DIR];
189     dim[Y_DIR] = upper_offset[Y_DIR] - offset[Y_DIR];
190     // Add space for rows/columns of neighboring grid
191     dim[X_DIR] += 2;
192     dim[Y_DIR] += 2;
193     ///! Step 7 end
194
195     /* allocate memory */
196     if ((phi = malloc(dim[X_DIR] * sizeof(*phi))) == NULL){
197         Debug("Setup_Subgrid : malloc(phi) failed", 1);
198     }
199     if ((source = malloc(dim[X_DIR] * sizeof(*source))) == NULL){
200         Debug("Setup_Subgrid : malloc(source) failed", 1);
201     }
202     if ((phi[0] = malloc(dim[Y_DIR] * dim[X_DIR] * sizeof(**phi))) == NULL){
203         Debug("Setup_Subgrid : malloc(*phi) failed", 1);
204     }
205     if ((source[0] = malloc(dim[Y_DIR] * dim[X_DIR] * sizeof(**source))) == NULL){
206         Debug("Setup_Subgrid : malloc(*source) failed", 1);
207     }

```

```

208     for (x = 1; x < dim[X_DIR]; x++)
209     {
210         phi[x] = phi[0] + x * dim[Y_DIR];
211         source[x] = source[0] + x * dim[Y_DIR];
212     }
213
214     /* set all values to '0' */
215     for (x = 0; x < dim[X_DIR]; x++){
216         for (y = 0; y < dim[Y_DIR]; y++){
217             {
218                 phi[x][y] = 0.0;
219                 source[x][y] = 0;
220             }
221         }
222     /* put sources in field */
223     do{
224         if (proc_rank==0)
225         {
226             s = fscanf(f, "source: %lf %lf %lf\n", &source_x, &source_y, &source_val);
227         }
228         MPI_Bcast(&s, 1, MPI_INT, 0, grid_comm);
229         if (s==3){
230             MPI_Bcast(&source_x, 1, MPI_DOUBLE, 0, grid_comm);
231             MPI_Bcast(&source_y, 1, MPI_DOUBLE, 0, grid_comm);
232             MPI_Bcast(&source_val, 1, MPI_DOUBLE, 0, grid_comm);
233             x = source_x * gridsize[X_DIR];
234             y = source_y * gridsize[Y_DIR];
235             x = x + 1 - offset[X_DIR]; // Step 7 --> local grid transform
236             y = y + 1 - offset[Y_DIR]; // Step 7 --> local grid transform
237             if(x > 0 && x < dim[X_DIR] - 1 && y > 0 && y < dim[Y_DIR] - 1){ // check if in local
grid
238                 phi[x][y] = source_val;
239                 source[x][y] = 1;
240             }
241         }
242     }
243     while (s==3);
244
245     if(proc_rank==0){
246         fclose(f);
247     }
248 }
249
250 void Setup_MPI_Datatypes()
251 {
252     Debug("Setup_MPI_Datatypes",0);
253
254     // vertical data exchange (Y_Dir)
255     MPI_Type_vector(dim[X_DIR] - 2, 1, dim[Y_DIR], MPI_DOUBLE, &border_type[Y_DIR]);
256     // horizontal data exchange (X_Dir)
257     MPI_Type_vector(dim[Y_DIR] - 2, 1, 1, MPI_DOUBLE, &border_type[X_DIR]);
258
259     MPI_Type_commit(&border_type[Y_DIR]);
260     MPI_Type_commit(&border_type[X_DIR]);
261 }
262
263 void Exchange_Borders()
264 {
265     Debug("Exchange_Borders",0);
266     // top direction
267     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);
268     // bottom direction
269     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);
270     // left direction
271     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);
272     // right direction
273     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);
274 }
275

```

```

276 double Do_Step(int parity)
277 {
278     int x, y;
279     double old_phi;
280     double max_err = 0.0;
281
282     /* calculate interior of grid */
283     for (x = 1; x < dim[X_DIR] - 1; x++){
284         for (y = 1; y < dim[Y_DIR] - 1; y++){
285             if ((x + offset[X_DIR] + y + offset[Y_DIR]) % 2 == parity && source[x][y] != 1){
286                 old_phi = phi[x][y];
287                 phi[x][y] = (phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1]) *
0.25;
288                 if (max_err < fabs(old_phi - phi[x][y])){
289                     max_err = fabs(old_phi - phi[x][y]);
290                 }
291             }
292         }
293     }
294
295     return max_err;
296 }
297
298 void Solve()
299 {
300     int count = 0;
301     double delta;
302     double global_delta;
303     double delta1, delta2;
304
305     Debug("Solve", 0);
306
307     /* give global_delta a higher value then precision_goal */
308     global_delta = 2 * precision_goal;
309
310     while (global_delta > precision_goal && count < max_iter)
311     {
312         Debug("Do_Step 0", 0);
313         delta1 = Do_Step(0);
314         Exchange_Borders();
315         Debug("Do_Step 1", 0);
316         delta2 = Do_Step(1);
317         Exchange_Borders();
318         delta = max(delta1, delta2);
319         MPI_Allreduce(&delta, &global_delta, 1, MPI_DOUBLE, MPI_MAX, grid_comm);
320         count++;
321     }
322
323     printf("(%) Number of iterations : %i\n", proc_rank, count);
324 }
325
326 double* get_Global_Grid()
327 {
328     Debug("get_Global_Grid", 0);
329     // Nur Prozess 0 benötigt Speicherplatz für das gesamte Gitter
330     double* global_phi = NULL;
331     if (proc_rank == 0) {
332         global_phi = malloc(gridsize[X_DIR] * gridsize[Y_DIR] * sizeof(double));
333         if (global_phi == NULL) {
334             Debug("get_Global_Grid : malloc(global_phi) failed", 1);
335         }
336     }
337
338     // Jeder Prozess muss seinen Teil in einen linearen Puffer kopieren
339     double* local_phi = malloc((dim[X_DIR] - 2) * (dim[Y_DIR] - 2) * sizeof(double));
340     int idx = 0;
341     for (int x = 1; x < dim[X_DIR] - 1; x++) {
342         for (int y = 1; y < dim[Y_DIR] - 1; y++) {
343             local_phi[idx++] = phi[x][y];
344         }
345     }
346
347     // Definiere die Sendcounts und Displacements nur auf Prozess 0

```



```

348     int* sendcounts = NULL;
349     int* displs = NULL;
350     if (proc_rank == 0) {
351         sendcounts = malloc(P * sizeof(int));
352         displs = malloc(P * sizeof(int));
353
354         // Berechne die Größen und Offsets für jedes Subgitter
355         int subgrid_width = gridsize[X_DIR] / P_grid[X_DIR];
356         int subgrid_height = gridsize[Y_DIR] / P_grid[Y_DIR];
357         for (int px = 0; px < P_grid[X_DIR]; px++) {
358             for (int py = 0; py < P_grid[Y_DIR]; py++) {
359                 int rank = px * P_grid[Y_DIR] + py;
360                 sendcounts[rank] = subgrid_width * subgrid_height;
361                 displs[rank] = (px * subgrid_width * gridsize[Y_DIR]) + (py * subgrid_height);
362             }
363         }
364     }
365     Debug("get_Global_Grid : MPI_Gatherv", 0);
366     // Sammle alle lokalen Gitter in das globale Gitter auf Prozess 0
367     MPI_Gatherv(local_phi, (dim[X_DIR] - 2) * (dim[Y_DIR] - 2), MPI_DOUBLE,
368                 global_phi, sendcounts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
369
370     free(local_phi);
371     if (proc_rank == 0) {
372         free(sendcounts);
373         free(displs);
374     }
375
376     return global_phi;
377 }
378
379 void Write_Grid()
380 {
381     double* global_phi = get_Global_Grid();
382     if(proc_rank != 0){
383         assert (global_phi == NULL);
384         return;
385     }
386     int x, y;
387     FILE *f;
388     char filename[40]; //seems danagerous to use a static buffer but let's go with the steps
389     sprintf(filename, "output_MPI%i.dat", proc_rank);
390     if ((f = fopen(filename, "w")) == NULL){
391         Debug("Write_Grid : fopen failed", 1);
392     }
393
394     Debug("Write_Grid", 0);
395
396     for (x = 0; x < gridsize[X_DIR]; x++){
397         for (y = 0; y < gridsize[Y_DIR]; y++){
398             fprintf(f, "%i %i %f\n", x+1, y+1, global_phi[x*gridsize[Y_DIR] + y]);
399         }
400     }
401     fclose(f);
402     free(global_phi);
403 }
404
405 void Clean_Up()
406 {
407     Debug("Clean_Up", 0);
408
409     free(phi[0]);
410     free(phi);
411     free(source[0]);
412     free(source);
413 }
414
415 int main(int argc, char **argv)
416 {
417     MPI_Init(&argc, &argv);
418     Setup_Proc_Grid(argc,argv); // was earlier MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank);
419     start_timer();
420

```

```
421     Setup_Grid();
422     Setup_MPI_Datatypes();
423
424     Solve();
425
426     Write_Grid();
427
428     print_timer();
429
430     Clean_Up();
431     MPI_Finalize();
432     return 0;
433 }
```