Lab Report

Author:
Elias Wachmann (6300421)

November 19, 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:
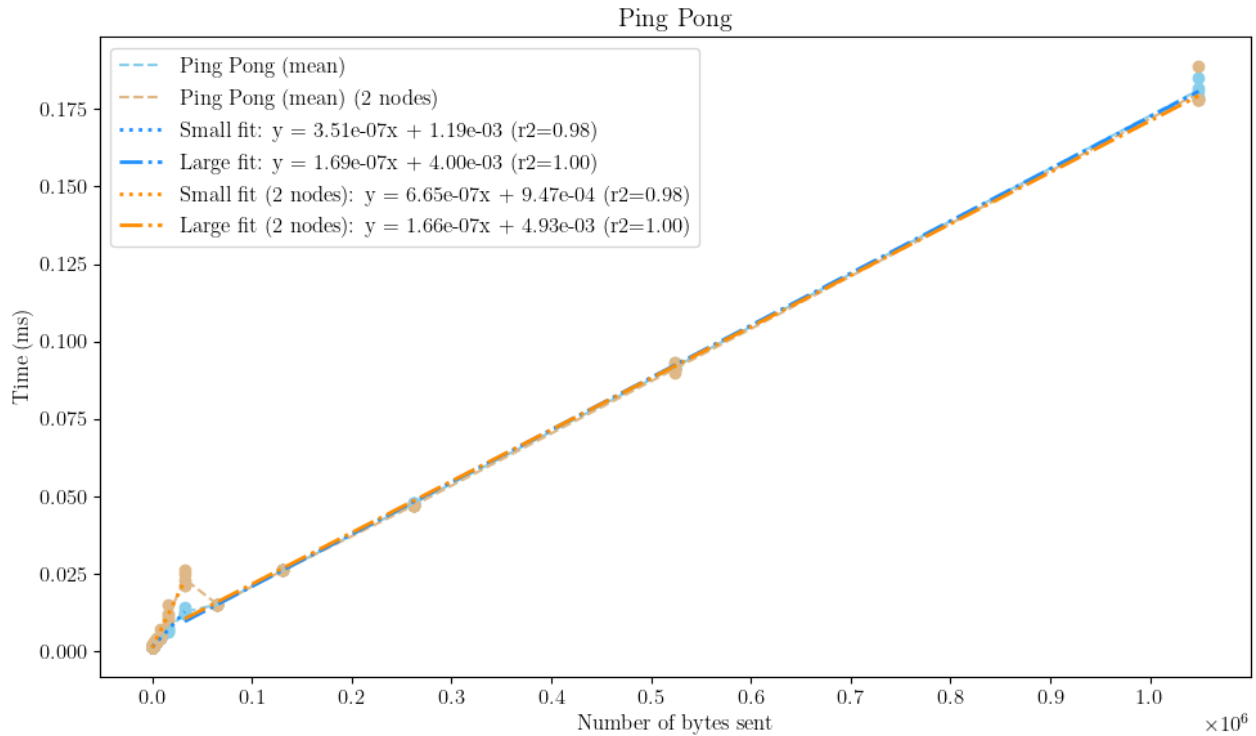
Figure 2: Ping Pong with MPI_SendRecv: Number of bytes sent vs. average time taken from 1000 pairs of send / receive. 5 runs shown for each size as scatter plot. Mean of these 5 runs shown as line. Blue small fit includes all data points up to 32768 bytes, blue large from there. Red small fit includes 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:
  $\texttt{total\_rows} - (\texttt{nr\_workers} - 1) \cdot floor(\frac{\texttt{total\_rows}}{\texttt{nr\_workers}})$.

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

- 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 2000×2000. 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 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.
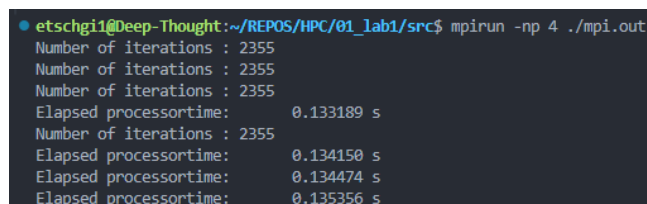
TODO: **Data locality?**

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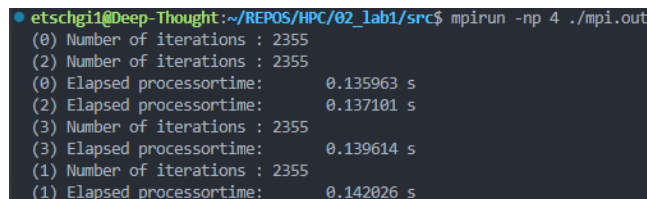
1. **Step:** After adding MPI_Init and MPI_Finalize, we can run the program with multiple processes. We can see that the program runs with 4 processes in Figure 4 via the quadrupeled output.
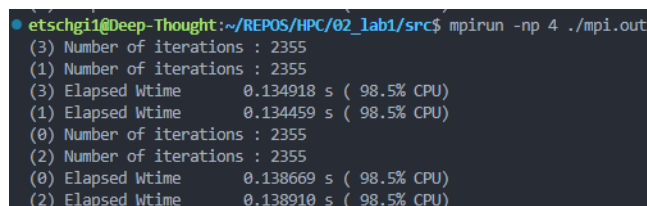

Figure 4: MPI_Poisson after Step 1 - Running with 4 processes

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

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

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

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

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

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

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

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

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

7. **Step:** We overhaul the setup to get a proper local grid for each process. Furthermore, we only save the relevant source fields in the local grid for each process.
With for instance 3 processes you should see that 1 or 2 processes do not do any iteration. Do you understand why?
If we have a look at the input file we see that there are only 3 source fields in the grid. This means that the process that does not have a source field in its local grid will not do any iterations (or only 1). Therefore, if we have 3 processes and the distribution of source fields as given in the input file only 1 process will do iterations if processes are ordered in x-direction and 2 if ordered in y-direction. From this we can conclude that indeed all processes have different local grids and perform different calculations.

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

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

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

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

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

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

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

to

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

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

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

1.2 Exercises, modifications, and performance aspects

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

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 1×4 topology.

1.2.1 Over-relaxation (SOR)

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

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

With the power of a little python scripting we can easily test different values for ω and plot the results as seen in Figure 10.
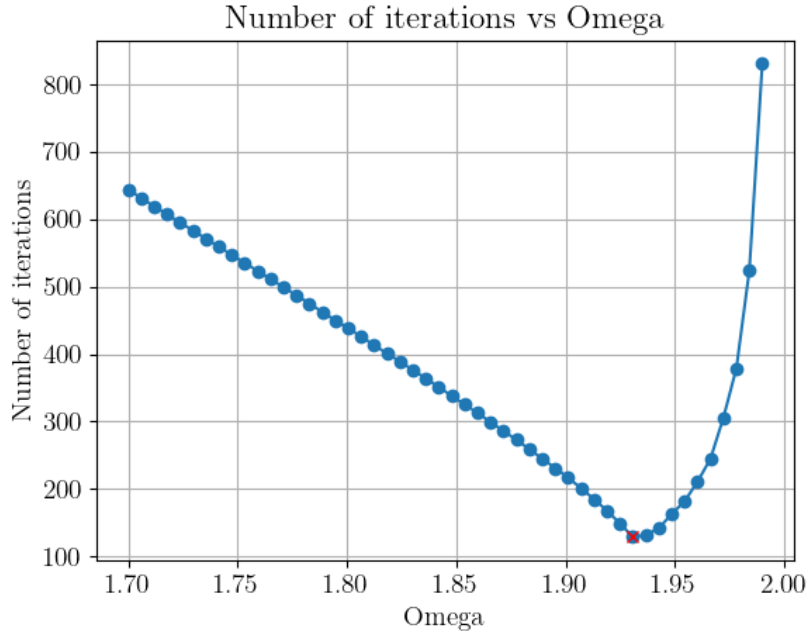

Figure 10: **Optimal ω for 4 processors on a 4x1 grid**

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

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);
```

```
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,
```

```c
            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:

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

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

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

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

Appendix - Poisson solver

The parallel Poisson solver used the following code:

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

```c
65  double Do_Step(int parity);
66  void Solve();
67  void Write_Grid();
68  void Clean_Up();
69  void Debug(char *mesg, int terminate);
70  void start_timer();
71  void resume_timer();
72  void stop_timer();
73  void print_timer();
74
75  void start_timer()
76  {
77      if (!timer_on){
78          MPI_Barrier(grid_comm);
79          ticks = clock();
80          wtime = MPI_Wtime();
81          timer_on = 1;
82      }
83  }
84
85  void resume_timer()
86  {
87      if (!timer_on){
88          ticks = clock()  -  ticks;
89          wtime = MPI_Wtime() - wtime;
90          timer_on = 1;
91      }
92  }
93
94  void stop_timer()
95  {
96      if (timer_on){
97          ticks = clock()  -  ticks;
98          wtime = MPI_Wtime() - wtime;
99          timer_on = 0;
100     }
101 }
102
103 void print_timer()
104 {
105     if (timer_on){
106         stop_timer();
107         printf("(%i) Elapsed Wtime %14.6f s (%5.1f%% CPU)\n", proc_rank, wtime, 100.0 * ticks
    * (1.0 / CLOCKS_PER_SEC) / wtime);
108         resume_timer();
109     }
110     else{
111         printf("(%i) Elapsed Wtime %14.6f s (%5.1f%% CPU)\n", proc_rank, wtime, 100.0 * ticks
    * (1.0 / CLOCKS_PER_SEC) / wtime);
112     }
113 }
114
115 void Debug(char *mesg, int terminate)
116 {
117     if (DEBUG || terminate){
118         printf("%s\n", mesg);
119     }
120     if (terminate){
121         exit(1);
122     }
123 }
124
125 void Setup_Proc_Grid(int argc, char **argv){
126     int wrap_around[2];
127     int reorder;
128
129     Debug("My_MPI_Init",0);
130
131     // num of processes
132     MPI_Comm_size(MPI_COMM_WORLD, &P);
133
134     //calculate the number of processes per column and per row for the grid
135     if(argc>2){
```

```
136        P_grid[X_DIR] = atoi(argv[1]);
137        P_grid[Y_DIR] = atoi(argv[2]);
138        if(P_grid[X_DIR] * P_grid[Y_DIR] != P){
139            Debug("ERROR Proces grid dimensions do not match with P ", 1);
140        }
141        if (argc>3)
142        {
143            // get sor from args
144            sor_omega = atof(argv[3]);
145            printf("Set sor_omega over argv to %5.f\n", sor_omega);
146        }
147    }
148    else{
149        Debug("ERROR Wrong parameter input",1);
150    }
151
152    // Create process topology (2D grid)
153    wrap_around[X_DIR] = 0;
154    wrap_around[Y_DIR] = 0;
155    reorder = 1; //reorder process ranks
156
157    // create grid_comm
158    int ret = MPI_Cart_create(MPI_COMM_WORLD, 2, P_grid, wrap_around, reorder, &grid_comm);
159    if (ret != MPI_SUCCESS){
160        Debug("ERROR: MPI_Cart_create failed",1);
161    }
162    //get new rank and cartesian coords of this proc
163    MPI_Comm_rank(grid_comm, &proc_rank);
164    MPI_Cart_coords(grid_comm, proc_rank, 2, proc_coord);
165    printf("(%i) (x,y)=(%i,%i)\n", proc_rank, proc_coord[X_DIR], proc_coord[Y_DIR]);
166    //calc neighbours
167    // MPI_Cart_shift(grid_comm, Y_DIR, 1, &proc_bottom, &proc_top);
168    MPI_Cart_shift(grid_comm, Y_DIR, 1, &proc_top, &proc_bottom);
169    MPI_Cart_shift(grid_comm, X_DIR, 1, &proc_left, &proc_right);
170    printf("(%i) top %i,  right  %i,  bottom  %i,  left  %i\n", proc_rank, proc_top,
       proc_right, proc_bottom, proc_left);
171 }
172
173 void Setup_Grid()
174 {
175    int x, y, s;
176    double source_x, source_y, source_val;
177    FILE *f;
178
179    Debug("Setup_Subgrid", 0);
180
181    if(proc_rank == 0){
182        f = fopen("input.dat", "r");
183        if (f == NULL){
184            Debug("Error opening input.dat", 1);
185        }
186        fscanf(f, "nx: %i\n", &gridsize[X_DIR]);
187        fscanf(f, "ny: %i\n", &gridsize[Y_DIR]);
188        fscanf(f, "precision goal: %lf\n", &precision_goal);
189        fscanf(f, "max iterations: %i\n", &max_iter);
190    }
191    MPI_Bcast(&gridsize, 2, MPI_INT, 0, grid_comm);
192    MPI_Bcast(&precision_goal, 1, MPI_DOUBLE, 0, grid_comm);
193    MPI_Bcast(&max_iter, 1, MPI_INT, 0, grid_comm);
194    h = 1;
195    /* Calculate dimensions of local subgrid */ //! We do that later now!
196    // dim[X_DIR] = gridsize[X_DIR] + 2;
197    // dim[Y_DIR] = gridsize[Y_DIR] + 2;
198
199    //! Step 7
200    int upper_offset[2] = {0,0};
201    // Calculate top left corner cordinates of local grid
202    offset[X_DIR] = gridsize[X_DIR] * proc_coord[X_DIR] / P_grid[X_DIR];
203    offset[Y_DIR] = gridsize[Y_DIR] * proc_coord[Y_DIR] / P_grid[Y_DIR];
204    upper_offset[X_DIR] = gridsize[X_DIR] * (proc_coord[X_DIR] + 1) / P_grid[X_DIR];
205    upper_offset[Y_DIR] = gridsize[Y_DIR] * (proc_coord[Y_DIR] + 1) / P_grid[Y_DIR];
206
207    // dimensions of local grid
```

```
208        dim[X_DIR] = upper_offset[X_DIR] - offset[X_DIR];
209        dim[Y_DIR] = upper_offset[Y_DIR] - offset[Y_DIR];
210        // Add space for rows/columns of neighboring grid
211        dim[X_DIR] += 2;
212        dim[Y_DIR] += 2;
213        //! Step 7 end
214
215        /* allocate memory */
216        if ((phi = malloc(dim[X_DIR] * sizeof(*phi))) == NULL){
217            Debug("Setup_Subgrid : malloc(phi) failed", 1);
218        }
219        if ((source = malloc(dim[X_DIR] * sizeof(*source))) == NULL){
220            Debug("Setup_Subgrid : malloc(source) failed", 1);
221        }
222        if ((phi[0] = malloc(dim[Y_DIR] * dim[X_DIR] * sizeof(**phi))) == NULL){
223            Debug("Setup_Subgrid : malloc(*phi) failed", 1);
224        }
225        if ((source[0] = malloc(dim[Y_DIR] * dim[X_DIR] * sizeof(**source))) == NULL){
226            Debug("Setup_Subgrid : malloc(*source) failed", 1);
227        }
228        for (x = 1; x < dim[X_DIR]; x++)
229        {
230            phi[x] = phi[0] + x * dim[Y_DIR];
231            source[x] = source[0] + x * dim[Y_DIR];
232        }
233
234        /* set all values to '0' */
235        for (x = 0; x < dim[X_DIR]; x++){
236            for (y = 0; y < dim[Y_DIR]; y++)
237            {
238                phi[x][y] = 0.0;
239                source[x][y] = 0;
240            }
241        }
242        /* put sources in field */
243        do{
244            if (proc_rank==0)
245            {
246                s = fscanf(f, "source: %lf %lf %lf\n", &source_x, &source_y, &source_val);
247            }
248            MPI_Bcast(&s, 1, MPI_INT, 0, grid_comm);
249            if (s==3){
250                MPI_Bcast(&source_x, 1, MPI_DOUBLE, 0, grid_comm);
251                MPI_Bcast(&source_y, 1, MPI_DOUBLE, 0, grid_comm);
252                MPI_Bcast(&source_val, 1, MPI_DOUBLE, 0, grid_comm);
253                x = source_x * gridsize[X_DIR];
254                y = source_y * gridsize[Y_DIR];
255                x = x + 1 - offset[X_DIR]; // Step 7 --> local grid transform
256                y = y + 1 - offset[Y_DIR]; // Step 7 --> local grid transform
257                if(x > 0 && x < dim[X_DIR] -1 && y > 0 && y < dim[Y_DIR]-1){ // check if in local
       grid
258                    phi[x][y] = source_val;
259                    source[x][y] = 1;
260                }
261            }
262        }
263        while (s==3);
264
265        if(proc_rank==0){
266            fclose(f);
267        }
268 }
269
270 void Setup_MPI_Datatypes()
271 {
272        Debug("Setup_MPI_Datatypes",0);
273
274        // vertical data exchange (Y_Dir)
275        MPI_Type_vector(dim[X_DIR] - 2, 1, dim[Y_DIR], MPI_DOUBLE, &border_type[Y_DIR]);
276        // horizontal data exchange (X_Dir)
277        MPI_Type_vector(dim[Y_DIR] - 2, 1, 1, MPI_DOUBLE, &border_type[X_DIR]);
278
279        MPI_Type_commit(&border_type[Y_DIR]);
```

```
280        MPI_Type_commit(&border_type[X_DIR]);
281 }
282
283 void Exchange_Borders()
284 {
285        Debug("Exchange_Borders",0);
286        // top direction
287        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);
288        // bottom direction
289        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);
290        // left direction
291        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);
292        // right direction
293        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);
294 }
295
296 double Do_Step(int parity)
297 {
298   int x, y;
299   double old_phi, c_ij;
300   double max_err = 0.0;
301
302   /* calculate interior of grid */
303      for (x = 1; x < dim[X_DIR] - 1; x++){
304          for (y = 1; y < dim[Y_DIR] - 1; y++){
305              if ((x + offset[X_DIR] + y + offset[Y_DIR]) % 2 == parity && source[x][y] != 1){
306                  old_phi = phi[x][y];
307                  #ifndef SOR
308                  phi[x][y] = (phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1]) *
           0.25;
309                  #endif
310                  #ifdef SOR //! I'm not quite sure about the h and source parts here
311                  c_ij = (phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1] + h*h*
           source[x][y]) * 0.25 - phi[x][y];
312                  phi[x][y] += sor_omega*c_ij;
313                  #endif
314                  if (max_err < fabs(old_phi - phi[x][y])){
315                      max_err = fabs(old_phi - phi[x][y]);
316                  }
317              }
318          }
319      }
320
321   return max_err;
322 }
323
324 void Solve()
325 {
326      int count = 0;
327      double delta;
328      double global_delta;
329      double delta1, delta2;
330
331      Debug("Solve", 0);
332
333      /* give global_delta a higher value then precision_goal */
334      global_delta = 2 * precision_goal;
335
336      while (global_delta > precision_goal && count < max_iter)
337      {
338          Debug("Do_Step 0", 0);
339          delta1 = Do_Step(0);
340          Exchange_Borders();
341          Debug("Do_Step 1", 0);
342          delta2 = Do_Step(1);
343          Exchange_Borders();
344          delta = max(delta1, delta2);
345          MPI_Allreduce(&delta, &global_delta, 1, MPI_DOUBLE, MPI_MAX, grid_comm);
346          count++;
```

```
347        }
348
349        printf("(%i) Number of iterations : %i\n", proc_rank, count);
350 }
351
352 double* get_Global_Grid()
353 {
354        Debug("get_Global_Grid", 0);
355        //!! DEBUG only
356        for (size_t i = 0; i < dim[X_DIR]; i++)
357        {
358            for (size_t j = 0; j < dim[Y_DIR]; j++)
359            {
360                phi[i][j] = proc_rank;
361            }
362
363        }
364
365        // only process 0 needs to store all data!
366        double* global_phi = NULL;
367        if (proc_rank == 0) {
368            global_phi = malloc(gridsize[X_DIR] * gridsize[Y_DIR] * sizeof(double));
369            if (global_phi == NULL) {
370                Debug("get_Global_Grid : malloc(global_phi) failed", 1);
371            }
372        }
373
374        // copy own part into buffer - flatten!
375        size_t buf_size = (dim[X_DIR] - 2) * (dim[Y_DIR] - 2) * sizeof(double);
376        double* local_phi = malloc(buf_size);
377        int idx = 0;
378        for (int x = 1; x < dim[X_DIR] - 1; x++) {
379            for (int y = 1; y < dim[Y_DIR] - 1; y++) {
380                local_phi[idx++] = phi[x][y];
381            }
382        }
383        printf("I'm proc %d and i have a buffer of size %zu\n", proc_rank, buf_size);
384
385
386        // only proc 0 needs sendcounts and displacements for the gatherv operation
387        int* sendcounts = NULL;
388        int* displs = NULL;
389        if (proc_rank == 0) {
390            sendcounts = malloc(P * sizeof(int));
391            displs = malloc(P * sizeof(int));
392
393            // size and offset of different subgrids
394            //! Note that this only works if every process has the same subgrid
395            if (gridsize[X_DIR] % P_grid[X_DIR] != 0 || gridsize[Y_DIR] % P_grid[Y_DIR] != 0)
396            {
397                Debug("!!!A grid dimension is not a multiple of the P_grid in this direction!", 1)
       ;
398            }
399
400            int subgrid_width = gridsize[X_DIR] / P_grid[X_DIR];
401            int subgrid_height = gridsize[Y_DIR] / P_grid[Y_DIR];
402            for (int px = 0; px < P_grid[X_DIR]; px++) {
403                for (int py = 0; py < P_grid[Y_DIR]; py++) {
404                    int rank = px * P_grid[Y_DIR] + py;
405                    sendcounts[rank] = subgrid_width * subgrid_height;
406                    displs[rank] = (px * subgrid_width * gridsize[Y_DIR]) + (py * subgrid_height);
407                }
408            }
409        }
410        Debug("get_Global_Grid : MPI_Gatherv", 0);
411        //! TODO this Gatherv does something wrong - all local grids are alright!!!
412        MPI_Gatherv(local_phi, (dim[X_DIR] - 2) * (dim[Y_DIR] - 2), MPI_DOUBLE, global_phi,
       sendcounts, displs, MPI_DOUBLE, 0, MPI_COMM_WORLD);
413
414        free(local_phi);
415        if (proc_rank == 0) {
416            free(sendcounts);
417            free(displs);
```

```
418        }
419
420        return global_phi;
421    }
422
423    void Write_Grid_global(){
424        int x, y;
425        FILE *f;
426        char filename[40]; //seems danagerous to use a static buffer but let's go with the steps
427        sprintf(filename, "output_MPI_global_%i.dat", proc_rank);
428        if ((f = fopen(filename, "w")) == NULL){
429            Debug("Write_Grid : fopen failed", 1);
430        }
431
432        Debug("Write_Grid", 0);
433
434        for (x = 1; x < dim[X_DIR]-1; x++){
435            for (y = 1; y < dim[Y_DIR]-1; y++){
436                int x_glob = x + offset[X_DIR];
437                int y_glob = y + offset[Y_DIR];
438                fprintf(f, "%i %i %f\n", x_glob, y_glob, phi[x][y]);
439            }
440        }
441        fclose(f);
442    }
443
444    void Write_Grid()
445    {
446        double* global_phi = get_Global_Grid();
447        if(proc_rank != 0){
448            assert (global_phi == NULL);
449            return;
450        }
451        int x, y;
452        FILE *f;
453        char filename[40]; //seems danagerous to use a static buffer but let's go with the steps
454        sprintf(filename, "output_MPI%i.dat", proc_rank);
455        if ((f = fopen(filename, "w")) == NULL){
456            Debug("Write_Grid : fopen failed", 1);
457        }
458
459        Debug("Write_Grid", 0);
460
461        for (x = 0; x < gridsize[X_DIR]; x++){
462            for (y = 0; y < gridsize[Y_DIR]; y++){
463                fprintf(f, "%i %i %f\n", x+1, y+1, global_phi[x*gridsize[Y_DIR] + y]);
464            }
465        }
466        fclose(f);
467        free(global_phi);
468    }
469
470    void Clean_Up()
471    {
472        Debug("Clean_Up", 0);
473
474        free(phi[0]);
475        free(phi);
476        free(source[0]);
477        free(source);
478    }
479
480    int main(int argc, char **argv)
481    {
482        MPI_Init(&argc, &argv);
483        Setup_Proc_Grid(argc,argv); // was earlier MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank);
484        start_timer();
485
486        Setup_Grid();
487        Setup_MPI_Datatypes();
488
489        #ifdef SOR
490        if (proc_rank == 0)
```

26

```
491        {
492            printf("SOR using omega: %.5f\n", sor_omega);
493        }
494    #endif
495
496    Solve();
497
498    // Write_Grid();
499    Write_Grid_global();
500    print_timer();
501
502    Clean_Up();
503    MPI_Finalize();
504    return 0;
505 }
```