Delft University of Technology

Introduction to High Performance Computing
WI4049TU

Lab Report

Author:
Elias Wachmann (6300421)

November 11, 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:

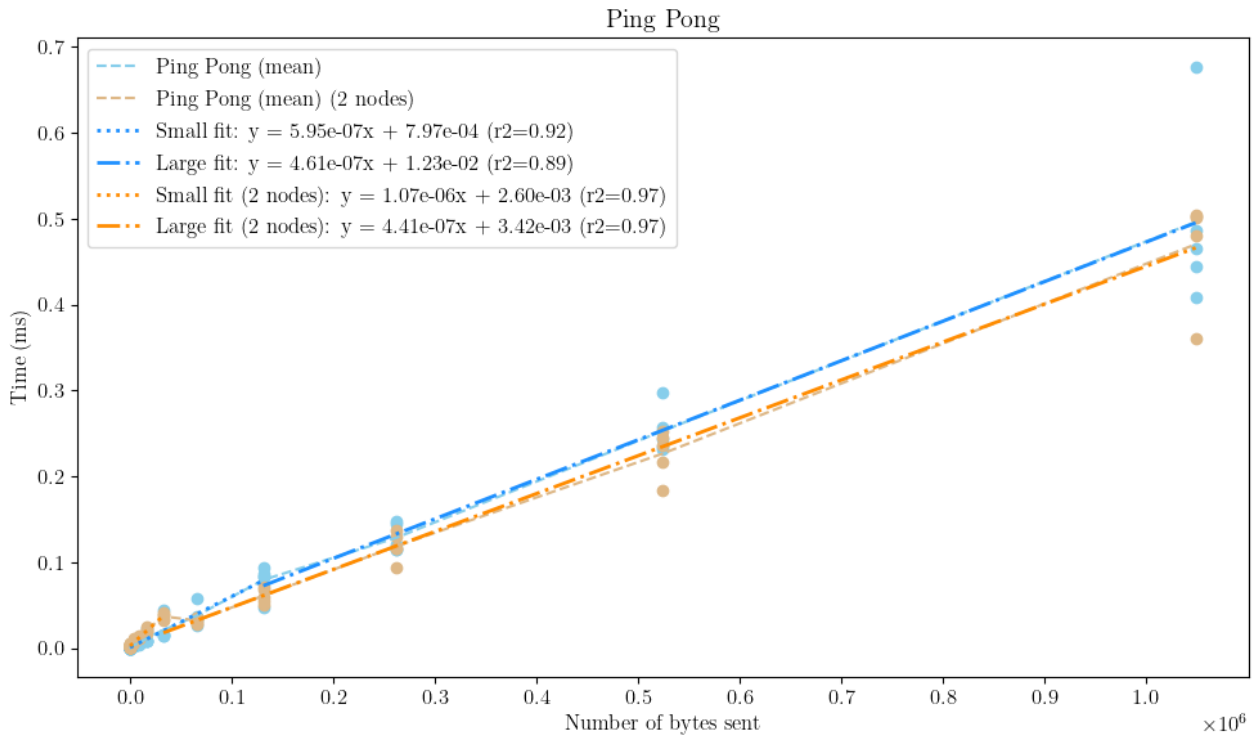


Figure 1: Ping Pong: Number of bytes sent vs. average time taken from 1000 pairs of send / receive. 5 runs shown for each size as scatter plot. Mean of these 5 runs shown as line. Blue small fit includes all data points up to 131072 bytes, blue large from there. Red small fit includes all data points 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 R^2 values:

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

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

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

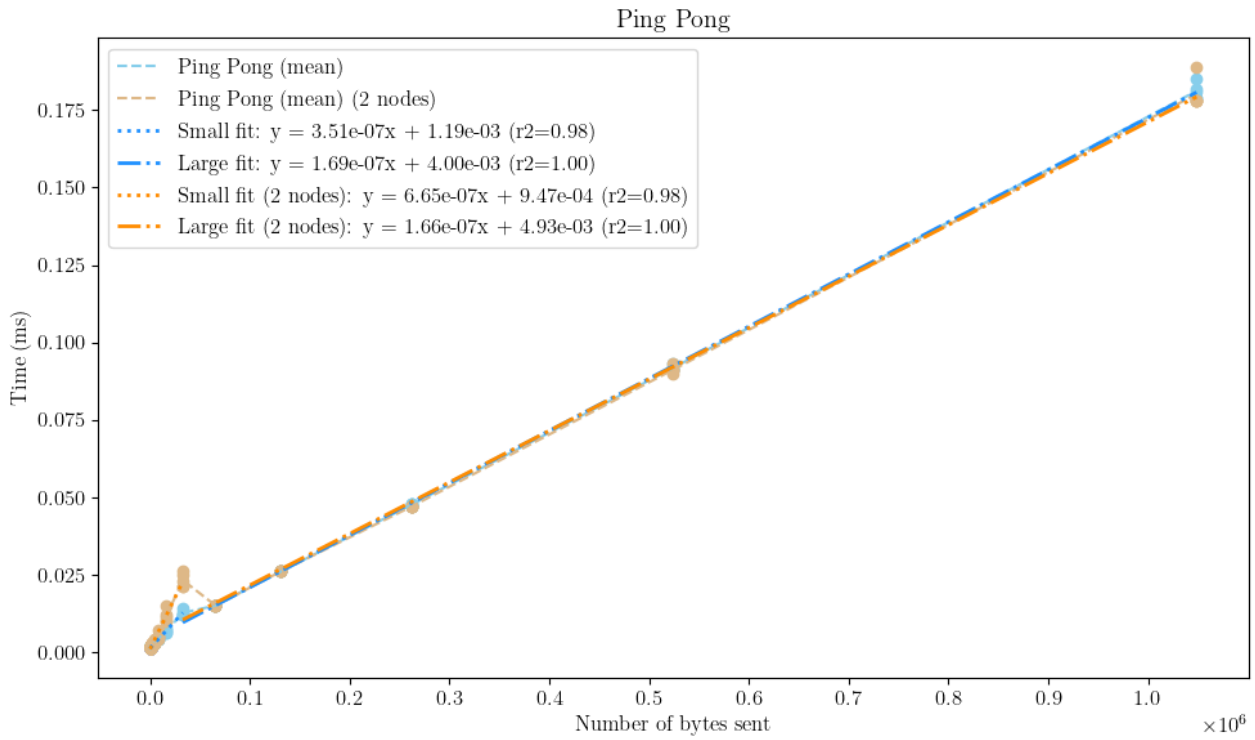


Figure 2: Ping Pong with MPI_SendRecv: Number of bytes sent vs. average time taken from 1000 pairs of send / receive. 5 runs shown for each size as scatter plot. Mean of these 5 runs shown as line. Blue small fit includes all data points up to 32768 bytes, blue large from there. Red small fit includes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

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

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

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

TODO: **Further analysis needed?**

0.b) MM-product

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

Explanation of the code

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

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

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

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

Partitioning Example

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

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

This partitioning can be visually represented as:

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

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

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

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

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

Discussion of the speedups

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

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

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

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

Table 3: Execution Time vs CPU Count

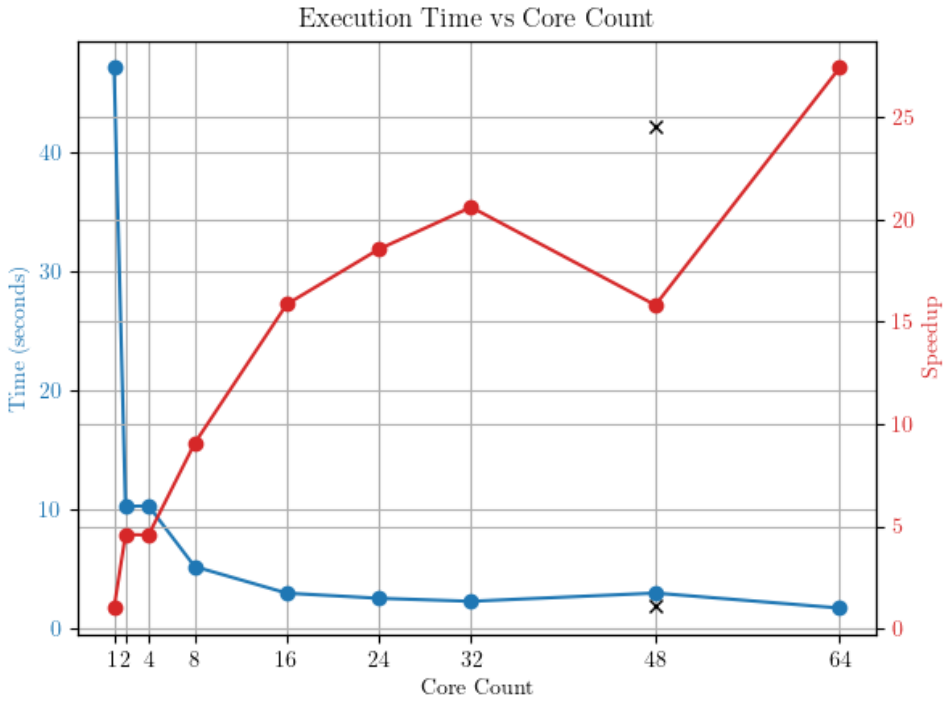


Figure 3: Speedup vs CPU Count
Black × marks the average of the rerun for $n = 48$.

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

Discussion:
As one can cleary discern from the data in Table 3 and Figure 3, the speedup increases with the number of nodes (with the exception of $n = 48$). This is expected as the more nodes we have, the more work can be done in

parallel. However, the speedup is not linear. This is due to the overhead of communication between the nodes. The more nodes we have, the more communication is needed, and this overhead increases. This is especially visible in the data for $n = 48$. Here the speedup is lower than for $n = 32$. For this run the communication didn't went as smooth as for the other runs. This can potentially be attributed to the fact that one (or more) of the nodes or the network was under heavy load during this task.
[Further investigation] After observing this slower speed for the $n = 48$, I reran the tests multiple times and got a runtime of around $1.9s$ which was to be expected initially. Therefore, this one run is an odd one out, most likely due to the reasons mentioned above! I've also added the averaged data of the reruns as a datapoint in Figure 3.
Another interesting fact can be seen when comparing the time taken for $n = 1$ and $n = 2$. They don't at all scale with the expected factor of 2. This is could be due to the fact, that the resource management system prefers runs with multiple nodes instead of a single node (= sequential).

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

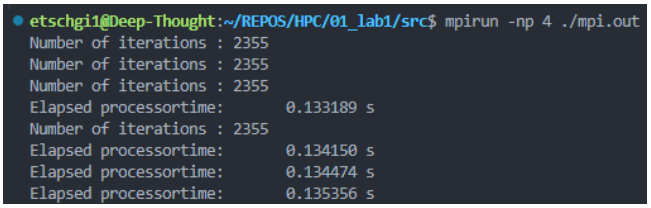


Figure 4: MPI_Poisson after Step 1 - Running with 4 processes

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.

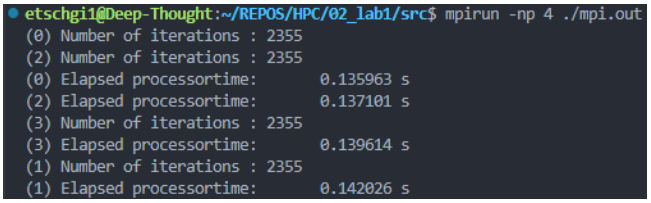


Figure 5: MPI_Poisson after Step 2 - Running with 4 processes

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.

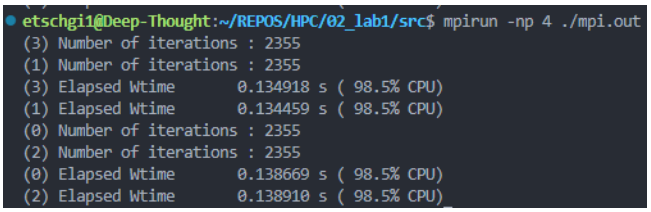


Figure 6: MPI_Poisson after Step 3 - Running with 4 processes

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.

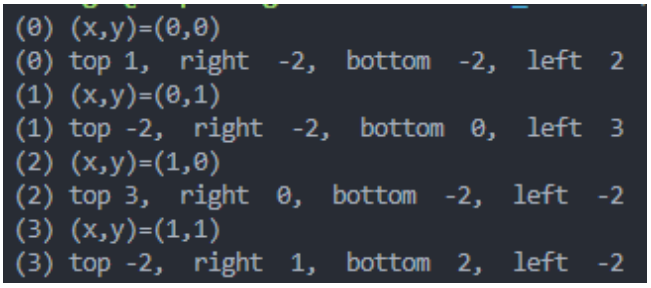
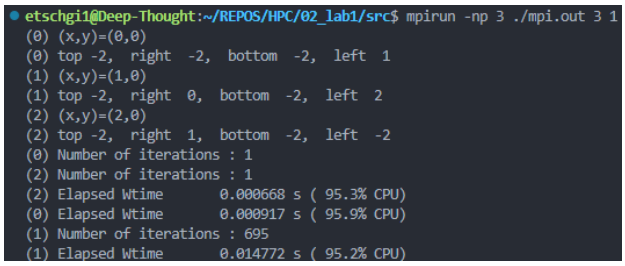


Figure 7: MPI_Poisson after Step 6 - Running with 4 processes on a 2x2 grid

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.



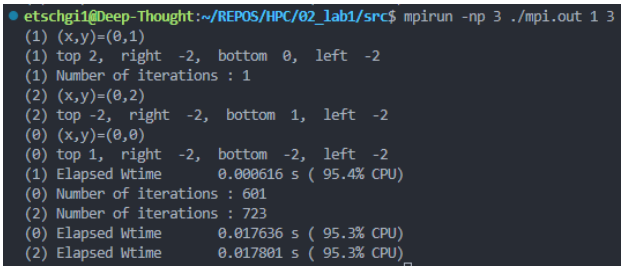


Figure 8: MPI_Poisson after Step 7 - Running with 3 processes on a 3x1 (left) vs. 1x3 (right) grid
For the 3x1 grid, only rank 1 does iterations (> 1), for the 1x3 grid, ranks 0 and 2 do iterations (> 1).

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_SUM` operation. This way we aggregate all deltas into one 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.

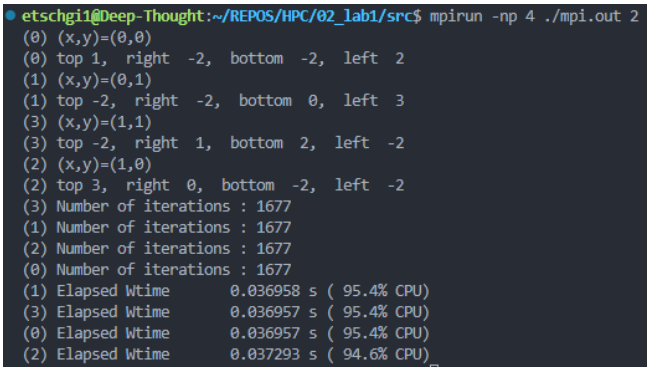


Figure 9: MPI_Poisson after Step 9 - Running with 4 processes on a 2x2 grid

TODO: **Different iteration count between seq and parallel version?! - also change pic if fixed.**

10.

2 Finite elements simulation

3 Eigenvalue solution by Power Method on GPU

Appendix - Introductory exercise

The following code was used for the ping pong task:

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

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

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

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


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

    int number_of_elements_to_send;
    int number_of_elements_received;

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

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

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

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

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

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

    // Finalize MPI
    MPI_Finalize();

    return 0;
}
```

For the bonus task, the following code was used:

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

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

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

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


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

    int number_of_elements_to_send;
    int number_of_elements_received;

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

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

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

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

    // Finalize MPI
    MPI_Finalize();

    return 0;
}
```

The matrix multiplication used the following code:

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

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

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

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

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

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

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

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

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

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

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



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

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

```
    double start = MPI_Wtime();
    int count;
    MPI_Get_count(&status, MPI_CHAR, &count);
    printf("I'm rank %d and I got %d bytes (%ld doubles) of data from %d with tag %d.\n", rank
    , count, (count-sizeof(size_t))/sizeof(double), status.MPI_SOURCE, status.MPI_TAG);

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

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

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

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

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

```

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

Appendix - Poisson solver

The parallel Poisson solver used the following code:

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

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

#define DEBUG 0

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

enum
{
    X_DIR, Y_DIR
};

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

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

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

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

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

void start_timer()
{
    if (!timer_on){
        MPI_Barrier(grid_comm);
        ticks = clock();
```

```
        wtime = MPI_Wtime();
        timer_on = 1;
    }
}

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

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

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

void Debug(char *mesg, int terminate)
{
    if (DEBUG || terminate){
        printf("%s\n", mesg);
    }
    if (terminate){
        exit(1);
    }
}

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

    Debug("My_MPI_Init",0);

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

    //calculate the number of processes per column and per row for the grid
    if(argc>2){
        P_grid[X_DIR] = atoi(argv[1]);
        P_grid[Y_DIR] = atoi(argv[2]);
        if(P_grid[X_DIR] * P_grid[Y_DIR] != P){
            Debug("ERROR Proces grid dimensions do not match with P ", 1);
        }
    }
    else{
        Debug("ERROR Wrong parameter input",1);
    }

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

```

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

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

```
276   int x, y;
277   double old_phi;
278   double max_err = 0.0;
279
280   /* calculate interior of grid */
281     for (x = 1; x < dim[X_DIR] - 1; x++){
282         for (y = 1; y < dim[Y_DIR] - 1; y++){
283             if ((x + y) % 2 == parity && source[x][y] != 1){
284                 old_phi = phi[x][y];
285                 phi[x][y] = (phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1]) *
    0.25;
286                 if (max_err < fabs(old_phi - phi[x][y])){
287                     max_err = fabs(old_phi - phi[x][y]);
288                 }
289             }
290         }
291     }
292
293   return max_err;
294 }
295
296 void Solve()
297 {
298     int count = 0;
299     double delta;
300     double global_delta;
301     double delta1, delta2;
302
303     Debug("Solve", 0);
304
305     /* give global_delta a higher value then precision_goal */
306     global_delta = 2 * precision_goal;
307
308     while (global_delta > precision_goal && count < max_iter)
309     {
310         Debug("Do_Step 0", 0);
311         delta1 = Do_Step(0);
312         Exchange_Borders();
313         Debug("Do_Step 1", 0);
314         delta2 = Do_Step(1);
315         Exchange_Borders();
316         delta = max(delta1, delta2);
317         MPI_Allreduce(&delta, &global_delta, 1, MPI_DOUBLE, MPI_SUM, grid_comm);
318         count++;
319     }
320
321     printf("(%i) Number of iterations : %i\n", proc_rank, count);
322 }
323
324 void Write_Grid()
325 {
326     int x, y;
327     FILE *f;
328     char filename[40]; //seems danagerous to use a static buffer but let's go with the steps
329     sprintf(filename, "output%i.dat", proc_rank);
330     if ((f = fopen(filename, "w")) == NULL){
331         Debug("Write_Grid : fopen failed", 1);
332     }
333
334     Debug("Write_Grid", 0);
335
336     for (x = 1; x < dim[X_DIR] - 1; x++){
337         for (y = 1; y < dim[Y_DIR] - 1; y++){
338             fprintf(f, "%i %i %f\n", x, y, phi[x][y]);
339         }
340     }
341     fclose(f);
342 }
343
344 void Clean_Up()
345 {
346     Debug("Clean_Up", 0);
347
```

```
348     free(phi[0]);
349     free(phi);
350     free(source[0]);
351     free(source);
352 }
353 
354 int main(int argc, char **argv)
355 {
356     MPI_Init(&argc, &argv);
357     Setup_Proc_Grid(argc,argv); // was earlier MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank);
358     start_timer();
359 
360     Setup_Grid();
361     Setup_MPI_Datatypes();
362 
363     Solve();
364 
365     Write_Grid();
366 
367     print_timer();
368 
369     Clean_Up();
370     MPI_Finalize();
371     return 0;
372 }
```