Lab Report

Author:
Elias Wachmann (6300421)

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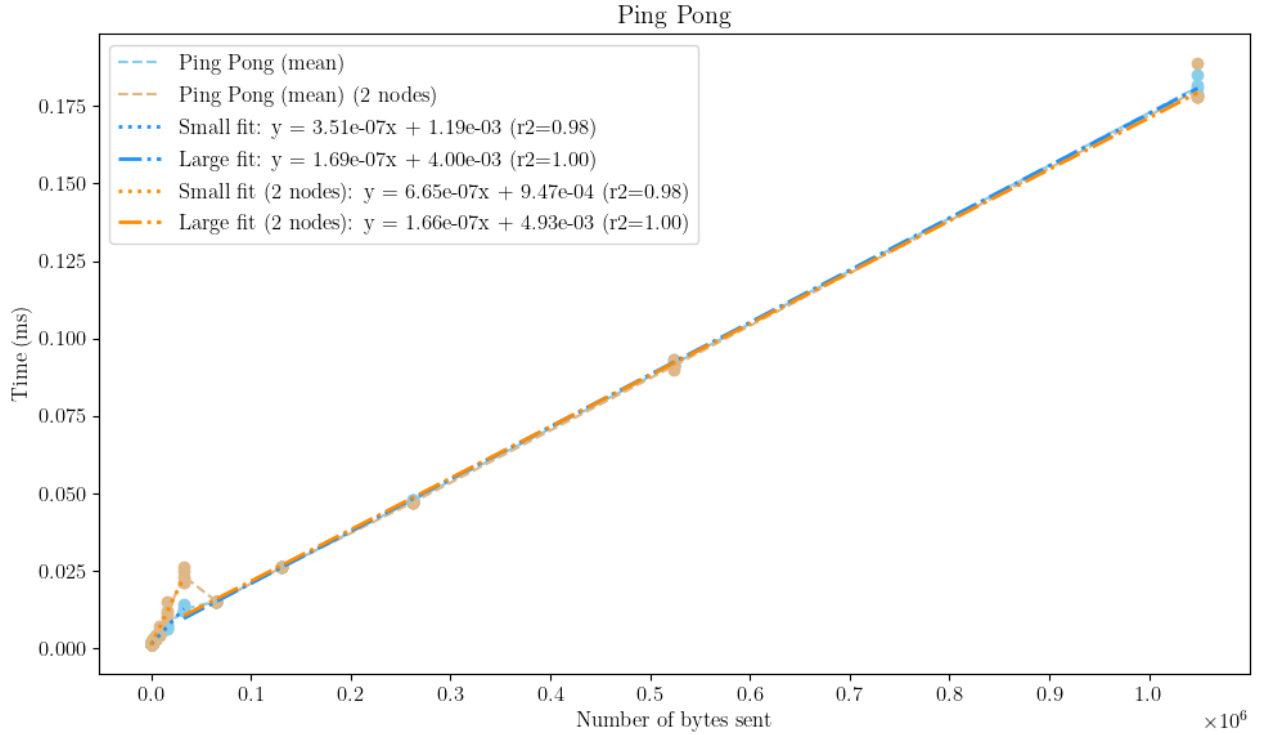
Figure 2: Ping Pong with MPI_SendRecv: Number of bytes sent vs. average time taken from 1000 pairs of send / receive. 5 runs shown for each size as scatter plot. Mean of these 5 runs shown as line. Blue small fit includes all data points up to 32768 bytes, blue large from there. Red small fit includes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

- 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

5

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.
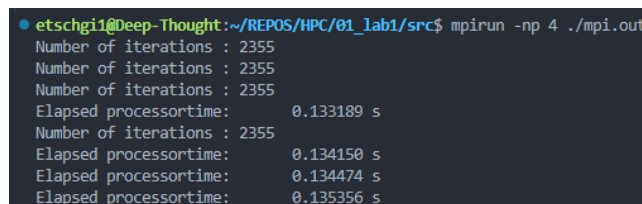
TODO: **Data locality?**

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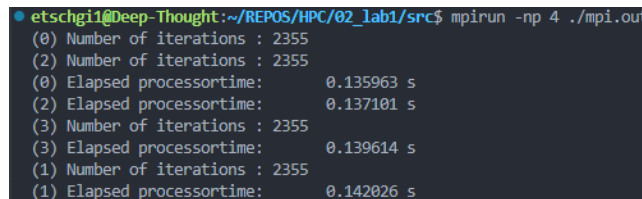
1. **Step:** After adding MPI_Init and MPI_Finalize, we can run the program with multiple processes. We can see that the program runs with 4 processes in Figure 4 via the quadrupeled output.
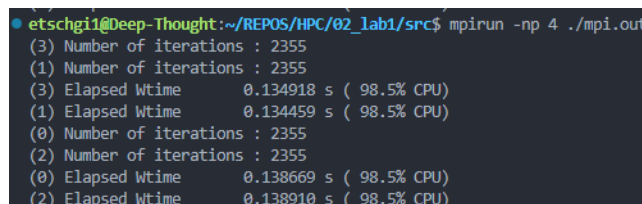

Figure 4: MPI_Poisson after Step 1 - Running with 4 processes

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.

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,
```

```
              number_of_elements_to_send, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
54              }

56              endTime = MPI_Wtime();
57              printf("Rank %2.1i: Received %i elements: Ping Pong took %f seconds\n", myRank,
        number_of_elements_to_send,(endTime - startTime)/(2*SAMPLE_COUNT));
58          }
59          else if (myRank == 1)
60          {
61              for (i=0; i<SAMPLE_COUNT; i++)
62              {
63                  MPI_Sendrecv(myArray, number_of_elements_to_send, MPI_INT, 0,0,myArray,
        number_of_elements_to_send, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
64              }
65          }
66      }

68      // Finalize MPI
69      MPI_Finalize();

71      return 0;
72 }
```

The matrix multiplication used the following code:

```
1 /*****************************************************************************
2  * FILE: mm.c
3  * DESCRIPTION:
4  *   This program calculates the product of matrix a[nra][nca] and b[nca][ncb],
5  *   the result is stored in matrix c[nra][ncb].
6  *   The max dimension of the matrix is constraint with static array
7  *declaration, for a larger matrix you may consider dynamic allocation of the
8  *arrays, but it makes a parallel code much more complicated (think of
9  *communication), so this is only optional.
10  *
11  *****************************************************************************/

13 #include <math.h>
14 #include <mpi.h>
15 #include <stdbool.h>
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <string.h>

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

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

29 void print_flattened_matrix(double *matrix, size_t rows, size_t cols, int rank) {
30     printf("[%d]\n", rank);
31     for (size_t i = 0; i < rows; i++) {
32         for (size_t j = 0; j < cols; j++) {
33             printf("%10.2f ", matrix[i * cols + j]);  // Accessing element in the 1D array
34         }
35         printf("\n");  // Newline after each row
36     }
37 }

39 int checkResult(double *truth, double *test, size_t Nr_col, size_t Nr_rows) {
40     for (size_t i = 0; i < Nr_rows; ++i) {
41         for (size_t j = 0; j < Nr_col; ++j) {
42             size_t index = i * Nr_col + j;
43             if (!eps_equal(truth[index], test[index])) {
44                 return 1;
45             }
46         }
47     }
48     return 0;
49 }
```

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

```
123    double(*b)[NCB] = malloc(sizeof(double) * NCA * NCB);
124    double(*c)[NCB] = malloc(sizeof(double) * NRA * NCB);
125
126    /*** Initialize matrices ***/
127    setupMatrices(a,b,c);
128
129    /* Parallelize the computation of the following matrix-matrix
130    multiplication. How to partition and distribute the initial matrices, the
131    work, and collecting final results.
132    */
133    // multiply
134    double start = MPI_Wtime();
135    for (size_t i = 0; i < NRA; i++) {
136        for (size_t j = 0; j < NCB; j++) {
137            for (size_t k = 0; k < NCA; k++) {
138                res[i * NCB + j] += a[i][k] * b[k][j];
139            }
140        }
141    }
142    /*  perform time measurement. Always check the correctness of the parallel
143        results by printing a few values of c[i][j] and compare with the
144        sequential output.
145    */
146    double time = MPI_Wtime()-start;
147    free(a);
148    free(b);
149    free(c);
150    return time;
151 }
152
153 double splitwork(double* res, size_t num_workers){
154    if (num_workers == 0) // sadly noone will help me :((
155    {
156        printf("Run sequential!\n");
157        return productSequential(res);
158    }
159
160    double(*a)[NCA] = malloc(sizeof(double) * NRA * NCA);
161    double(*b)[NCB] = malloc(sizeof(double) * NCA * NCB);
162    double(*c)[NCB] = malloc(sizeof(double) * NRA * NCB);
163    // Transpose matrix b to make accessing columns easier - in row major way - better cache
       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!
194     size_t i;
195     for (i = 0; i < (num_workers-1); ++i)
196     {
197         MM_input *data = (MM_input*)malloc(sizeof(MM_input));
198         data->rows = rows_per_worker;
199         data->a = (double*)(a + (row_end_first + rows_per_worker*i));
200         data->b = (double*)(b_transposed); // send everyting - all needed
201         buffer = getbuffer(data, total_size);
202         printf("nr_worker - %zu\n", i);
203         MPI_Isend(buffer, total_size, MPI_CHAR, i+2, i+1,MPI_COMM_WORLD, &requests[i+1]);
204         free(data);
205     }
206     double* my_a = (double*)(a + (row_end_first + rows_per_worker*i));
207
208     //I multiply the rest
209     size_t offset = 0;
210     for (size_t row = (NRA-rows_per_worker); row < NRA; row++)
211     {
212         for (size_t col = 0; col < NCB; col++)
213         {
214             res[row * NCB + col] = multsum(my_a+offset, (((double*)b_transposed)+col*NCA), NCA
    );
215         }
216         offset += NCA;
217     }
218     printf("My c: \n");
219     //wait for rest
220     MPI_Status stats[num_workers];
221     if(MPI_Waitall(num_workers, requests, stats) == MPI_ERR_IN_STATUS){
222         printf("Communication failed!!! - abort\n");
223     }
224     printf(">>>Everything sent and recieved\n");
225
226     // reviece rest
227     size_t buf_size = sizeof(double)*row_end_first*NCB;
228     double* revbuf;
229     offset = 0;
230     for (size_t worker = 0; worker < num_workers; worker++)
231     {
232         revbuf = (double*)malloc(buf_size); //first gets largest buffer
233         MPI_Recv(revbuf, buf_size/sizeof(double), MPI_DOUBLE, worker+1, worker, MPI_COMM_WORLD
    ,&stats[worker]);
234         memcpy(&res[offset/sizeof(double)], revbuf, buf_size);
235         free(revbuf);
236         offset += buf_size;
237         buf_size = sizeof(double)*rows_per_worker*NCB;
238     }
239     double time = MPI_Wtime()-start_time;
240     //free all pointers!
241     free(a);
242     free(b);
243     free(b_transposed);
244     free(c);
245     return time;
246 }
247
248
249
250 double work(int rank, size_t num_workers){
251     size_t rows_per_worker = NRA / (num_workers+1);
252     char* buffer;
253     MPI_Status status;
254     if (rank == 1) // first always get's most work
255     {
256         rows_per_worker = NRA - rows_per_worker*num_workers;
257     }
258     size_t size_of_meta = sizeof(size_t);
259     size_t size_of_a = sizeof(double)*rows_per_worker*NCA;
260     size_t buffersize = size_of_meta+size_of_a + SIZE_OF_B;
261     buffer = (char*)malloc(buffersize);
262
263     MPI_Recv(buffer, buffersize, MPI_CHAR, 0, rank-1, MPI_COMM_WORLD, &status);
```

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

```c
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
12 #define DEBUG 0
13
14 #define max(a,b) ((a)>(b)?a:b)
15
16 enum
17 {
18     X_DIR, Y_DIR
19 };
20
21 /* global variables */
22 int gridsize[2];
23 double precision_goal;    /* precision_goal of solution */
24 int max_iter;     /* maximum number of iterations alowed */
25 int proc_rank;
26 double wtime;
27
28 /* benchmark related variables */
29 clock_t ticks;       /* number of systemticks */
30 int timer_on = 0;    /* is timer running? */
31
32 /* local grid related variables */
33 double **phi;     /* grid */
34 int **source;      /* TRUE if subgrid element is a source */
35 int dim[2];      /* grid dimensions */
36
37 void Setup_Grid();
38 double Do_Step(int parity);
39 void Solve();
40 void Write_Grid();
41 void Clean_Up();
42 void Debug(char *mesg, int terminate);
43 void start_timer();
44 void resume_timer();
45 void stop_timer();
46 void print_timer();
47
48 void start_timer()
49 {
50     if (!timer_on){
51         MPI_Barrier(MPI_COMM_WORLD);
52         ticks = clock();
53         wtime = MPI_Wtime();
54         timer_on = 1;
55     }
56 }
57
58 void resume_timer()
59 {
60     if (!timer_on){
61         ticks = clock()  -  ticks;
62         wtime = MPI_Wtime() - wtime;
63         timer_on = 1;
64     }
```

```c
65  }
66
67  void stop_timer()
68  {
69      if (timer_on){
70          ticks = clock()  -  ticks;
71          wtime = MPI_Wtime() - wtime;
72          timer_on = 0;
73      }
74  }
75
76  void print_timer()
77  {
78      if (timer_on){
79          stop_timer();
80          printf("(%i) Elapsed Wtime %14.6f s (%5.1f%% CPU)\n", proc_rank, wtime, 100.0 * ticks
      * (1.0 / CLOCKS_PER_SEC) / wtime);
81          resume_timer();
82      }
83      else{
84          printf("(%i) Elapsed Wtime %14.6f s (%5.1f%% CPU)\n", proc_rank, wtime, 100.0 * ticks
      * (1.0 / CLOCKS_PER_SEC) / wtime);
85      }
86  }
87
88  void Debug(char *mesg, int terminate)
89  {
90      if (DEBUG || terminate){
91          printf("%s\n", mesg);
92      }
93      if (terminate){
94          exit(1);
95      }
96  }
97
98  void Setup_Grid()
99  {
100     int x, y, s;
101     double source_x, source_y, source_val;
102     FILE *f;
103
104     Debug("Setup_Subgrid", 0);
105
106     f = fopen("input.dat", "r");
107     if (f == NULL){
108         Debug("Error opening input.dat", 1);
109     }
110     fscanf(f, "nx: %i\n", &gridsize[X_DIR]);
111     fscanf(f, "ny: %i\n", &gridsize[Y_DIR]);
112     fscanf(f, "precision goal: %lf\n", &precision_goal);
113     fscanf(f, "max iterations: %i\n", &max_iter);
114
115     /* Calculate dimensions of local subgrid */
116     dim[X_DIR] = gridsize[X_DIR] + 2;
117     dim[Y_DIR] = gridsize[Y_DIR] + 2;
118
119     /* allocate memory */
120     if ((phi = malloc(dim[X_DIR] * sizeof(*phi))) == NULL){
121         Debug("Setup_Subgrid : malloc(phi) failed", 1);
122     }
123     if ((source = malloc(dim[X_DIR] * sizeof(*source))) == NULL){
124         Debug("Setup_Subgrid : malloc(source) failed", 1);
125     }
126     if ((phi[0] = malloc(dim[Y_DIR] * dim[X_DIR] * sizeof(**phi))) == NULL){
127         Debug("Setup_Subgrid : malloc(*phi) failed", 1);
128     }
129     if ((source[0] = malloc(dim[Y_DIR] * dim[X_DIR] * sizeof(**source))) == NULL){
130         Debug("Setup_Subgrid : malloc(*source) failed", 1);
131     }
132     for (x = 1; x < dim[X_DIR]; x++)
133     {
134         phi[x] = phi[0] + x * dim[Y_DIR];
135         source[x] = source[0] + x * dim[Y_DIR];
```

```
136        }
137
138        /* set all values to '0' */
139        for (x = 0; x < dim[X_DIR]; x++){
140            for (y = 0; y < dim[Y_DIR]; y++)
141            {
142                phi[x][y] = 0.0;
143                source[x][y] = 0;
144            }
145        }
146        /* put sources in field */
147        do{
148            s = fscanf(f, "source: %lf %lf %lf\n", &source_x, &source_y, &source_val);
149            if (s==3){
150                x = source_x * gridsize[X_DIR];
151                y = source_y * gridsize[Y_DIR];
152                x += 1;
153                y += 1;
154                phi[x][y] = source_val;
155                source[x][y] = 1;
156            }
157        }
158        while (s==3);
159
160        fclose(f);
161 }
162
163 double Do_Step(int parity)
164 {
165   int x, y;
166   double old_phi;
167   double max_err = 0.0;
168
169   /* calculate interior of grid */
170     for (x = 1; x < dim[X_DIR] - 1; x++){
171         for (y = 1; y < dim[Y_DIR] - 1; y++){
172             if ((x + y) % 2 == parity && source[x][y] != 1){
173                 old_phi = phi[x][y];
174                 phi[x][y] = (phi[x + 1][y] + phi[x - 1][y] + phi[x][y + 1] + phi[x][y - 1]) *
175     0.25;
176                 if (max_err < fabs(old_phi - phi[x][y])){
176                     max_err = fabs(old_phi - phi[x][y]);
177                 }
178             }
179         }
180     }
181
182   return max_err;
183 }
184
185 void Solve()
186 {
187     int count = 0;
188     double delta;
189     double delta1, delta2;
190
191     Debug("Solve", 0);
192
193     /* give global_delta a higher value then precision_goal */
194     delta = 2 * precision_goal;
195
196     while (delta > precision_goal && count < max_iter)
197     {
198         Debug("Do_Step 0", 0);
199         delta1 = Do_Step(0);
200
201         Debug("Do_Step 1", 0);
202         delta2 = Do_Step(1);
203
204         delta = max(delta1, delta2);
205         count++;
206     }
207
```

```
208         printf("(%i) Number of iterations : %i\n", proc_rank, count);
209 }
210
211 void Write_Grid()
212 {
213     int x, y;
214     FILE *f;
215     char filename[40]; //seems danagerous to use a static buffer but let's go with the steps
216     sprintf(filename, "output%i.dat", proc_rank);
217     if ((f = fopen(filename, "w")) == NULL){
218         Debug("Write_Grid : fopen failed", 1);
219     }
220
221     Debug("Write_Grid", 0);
222
223     for (x = 1; x < dim[X_DIR] - 1; x++){
224         for (y = 1; y < dim[Y_DIR] - 1; y++){
225             fprintf(f, "%i %i %f\n", x, y, phi[x][y]);
226         }
227     }
228     fclose(f);
229 }
230
231 void Clean_Up()
232 {
233     Debug("Clean_Up", 0);
234
235     free(phi[0]);
236     free(phi);
237     free(source[0]);
238     free(source);
239 }
240
241 int main(int argc, char **argv)
242 {
243     MPI_Init(&argc, &argv);
244     MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank);
245     start_timer();
246
247     Setup_Grid();
248
249     Solve();
250
251     Write_Grid();
252
253     print_timer();
254
255     Clean_Up();
256     MPI_Finalize();
257     return 0;
258 }
```