Lab Report

Author:
Elias Wachmann (6300421)

October 14, 2024

General Remarks

This final Lab report includes the answers for the exercises (base grad denoted in paranthesis):

0. Introductory exercise (0.5)

1. Poisson solver (1.75)

2. Finite elements simulation (1.0)

3. Eigenvalue solution by Power Method on GPU (1.75)

The optional **shining points** (e.g., performance analysis, optimization, discussion, and clarifying figures) which yield further points are usually marked by a small blue heading in the text or an additional note is added under a figure or table. For example:
This is a shining point.

0   Introductory exercise

In the introductory lab session, we are taking a look at some basic features of MPI.
We start out very simple with a hello world program on two nodes.

Hello World

```
1  #include "mpi.h"
2  #include <stdio.h>
3
4  int np, rank;
5
6  int main(int argc, char **argv)
7  {
8    MPI_Init(&argc, &argv);
9    MPI_Comm_size(MPI_COMM_WORLD, &np);
10   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11
12   printf("Node %d of %d says: Hello world!\n", rank, np);
13
14   MPI_Finalize();
15   return 0;
16 }
```

This program can be compiled with the following command:

```
mpicc -o helloworld1.out helloworld1.c
```

And run with:

```
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:
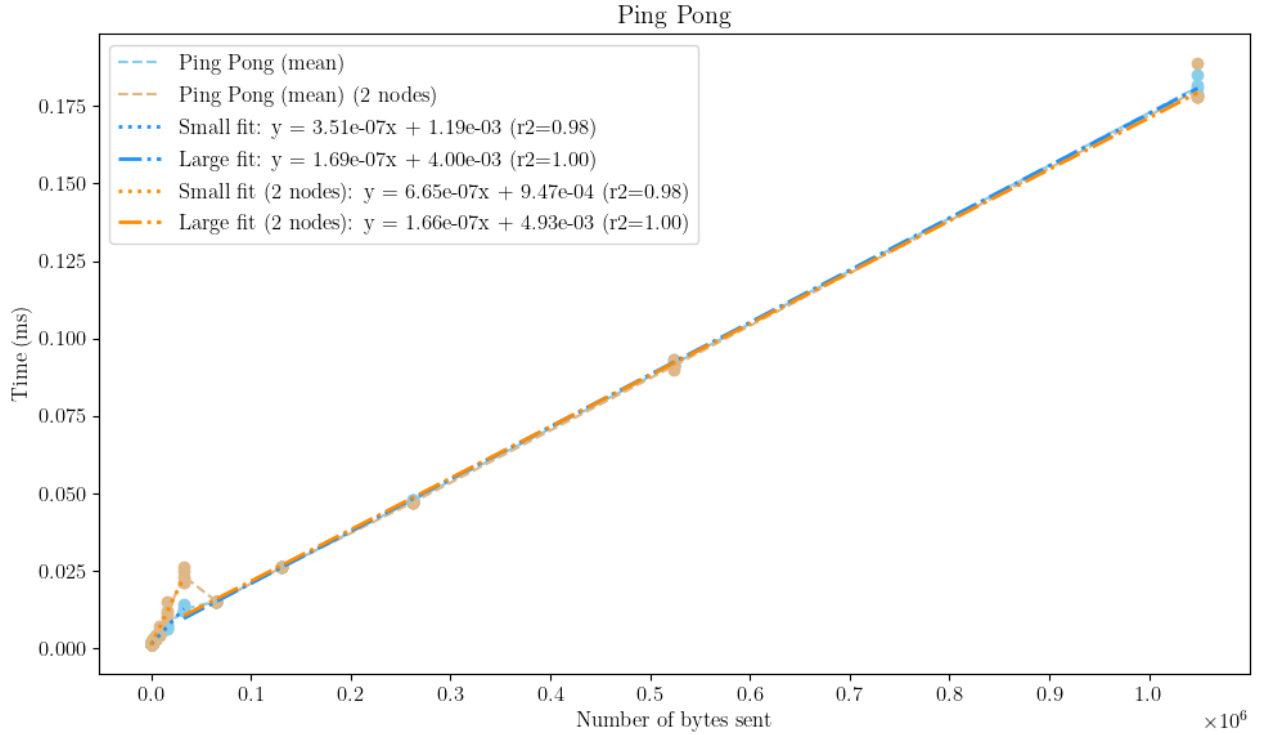
2

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:

```
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.

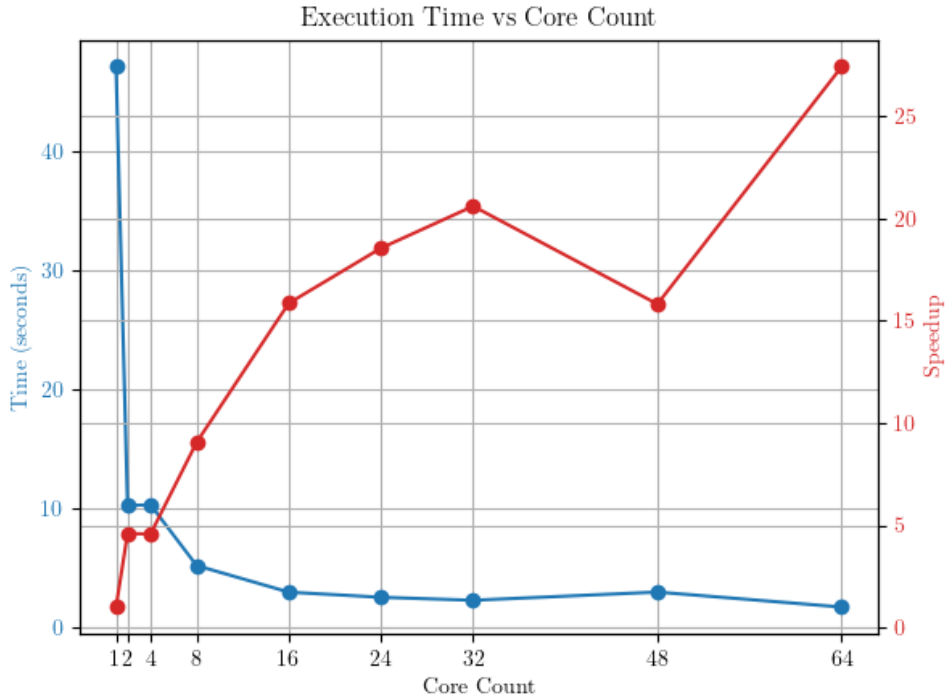| CPU Count	Execution Time / s	Approx. Speedup
1	47.11	1.0
2	10.26	4.6
4	10.30	4.6
8	5.20	9.1
16	2.97	15.9
24	2.54	18.5
32	2.29	20.6
48	2.98	15.8
64	1.72	27.4

Table 3: Execution Time vs CPU Count

Figure 3: Speedup vs CPU Count

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.

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.

1 Poisson solver

2 Finite elements simulation

3 Eigenvalue solution by Power Method on GPU

Appendix - Introductory exercise

The following code was used for the ping pong task:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4
5  // Maximum array size 2^20= 1048576 elements
6  #define MAX_EXPONENT 20
7  #define MAX_ARRAY_SIZE (1<<MAX_EXPONENT)
8  #define SAMPLE_COUNT 1000
9
10 int main(int argc, char **argv)
11 {
12     // Variables for the process rank and number of processes
13     int myRank, numProcs, i;
14     MPI_Status status;
15
16     // Initialize MPI, find out MPI communicator size and process rank
17     MPI_Init(&argc, &argv);
18     MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
19     MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
20
21
22     int *myArray = (int *)malloc(sizeof(int)*MAX_ARRAY_SIZE);
23     if (myArray == NULL)
24     {
25         printf("Not enough memory\n");
26         exit(1);
27     }
28     // Initialize myArray
29     for (i=0; i<MAX_ARRAY_SIZE; i++)
30         myArray[i]=1;
31
32     int number_of_elements_to_send;
33     int number_of_elements_received;
34
35     // PART C
36     if (numProcs < 2)
37     {
38         printf("Error: Run the program with at least 2 MPI tasks!\n");
39         MPI_Abort(MPI_COMM_WORLD, 1);
40     }
41     double startTime, endTime;
42
43     // TODO: Use a loop to vary the message size
44     for (size_t j = 0; j <= MAX_EXPONENT; j++)
45     {
```

```
46        number_of_elements_to_send = 1<<j;
47        if (myRank == 0)
48        {
49            myArray[0]=myArray[1]+1; // activate in cache (avoids possible delay when sending
   the 1st element)
50            startTime = MPI_Wtime();
51            for (i=0; i<SAMPLE_COUNT; i++)
52            {
53                MPI_Send(myArray, number_of_elements_to_send, MPI_INT, 1, 0,
54                    MPI_COMM_WORLD);
55                MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
56                MPI_Get_count(&status, MPI_INT, &number_of_elements_received);
57
58                MPI_Recv(myArray, number_of_elements_received, MPI_INT, 1, 0,
59                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
60            } // end of for-loop
61
62            endTime = MPI_Wtime();
63            printf("Rank %2.1i: Received %i elements: Ping Pong took %f seconds\n", myRank,
   number_of_elements_received,(endTime - startTime)/(2*SAMPLE_COUNT));
64        }
65        else if (myRank == 1)
66        {
67            // Probe message in order to obtain the amount of data
68            MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
69            MPI_Get_count(&status, MPI_INT, &number_of_elements_received);
70
71            for (i=0; i<SAMPLE_COUNT; i++)
72            {
73                MPI_Recv(myArray, number_of_elements_received, MPI_INT, 0, 0,
74                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
75                MPI_Send(myArray, number_of_elements_to_send, MPI_INT, 0, 0,
76                MPI_COMM_WORLD);
77            } // end of for-loop
78        }
79    }
80
81    // Finalize MPI
82    MPI_Finalize();
83
84    return 0;
85 }
```

For the bonus task, the following code was used:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4
5 // Maximum array size 2^20= 1048576 elements
6 #define MAX_EXPONENT 20
7 #define MAX_ARRAY_SIZE (1<<MAX_EXPONENT)
8 #define SAMPLE_COUNT 1000
9
10 int main(int argc, char **argv)
11 {
12     // Variables for the process rank and number of processes
13     int myRank, numProcs, i;
14     MPI_Status status;
15
16     // Initialize MPI, find out MPI communicator size and process rank
17     MPI_Init(&argc, &argv);
18     MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
19     MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
20
21
22     int *myArray = (int *)malloc(sizeof(int)*MAX_ARRAY_SIZE);
23     if (myArray == NULL)
24     {
25         printf("Not enough memory\n");
26         exit(1);
27     }
28     // Initialize myArray
29     for (i=0; i<MAX_ARRAY_SIZE; i++)
```

```
30          myArray[i]=1;
31
32      int number_of_elements_to_send;
33      int number_of_elements_received;
34
35      // PART C
36      if (numProcs < 2)
37      {
38          printf("Error: Run the program with at least 2 MPI tasks!\n");
39          MPI_Abort(MPI_COMM_WORLD, 1);
40      }
41      double startTime, endTime;
42
43      // TODO: Use a loop to vary the message size
44      for (size_t j = 0; j <= MAX_EXPONENT; j++)
45      {
46          number_of_elements_to_send = 1<<j;
47          if (myRank == 0)
48          {
49              myArray[0]=myArray[1]+1; // activate in cache (avoids possible delay when sending
    the 1st element)
50              startTime = MPI_Wtime();
51              for (i=0; i<SAMPLE_COUNT; i++)
52              {
53                  MPI_Sendrecv(myArray, number_of_elements_to_send, MPI_INT, 1,0,myArray,
    number_of_elements_to_send, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
54              }
55
56              endTime = MPI_Wtime();
57              printf("Rank %2.1i: 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      }
67
68      // Finalize MPI
69      MPI_Finalize();
70
71      return 0;
72 }
```

The matrix multiplication used the following code:

```
 1 /*******************************************************************************
 2  * FILE: mm.c
 3  * DESCRIPTION:
 4  *   This program calculates the product of matrix a[nra][nca] and b[nca][ncb],
 5  *   the result is stored in matrix c[nra][ncb].
 6  *   The max dimension of the matrix is constraint with static array
 7  *declaration, for a larger matrix you may consider dynamic allocation of the
 8  *arrays, but it makes a parallel code much more complicated (think of
 9  *communication), so this is only optional.
10  *
11  ******************************************************************************/
12
13 #include <math.h>
14 #include <mpi.h>
15 #include <stdbool.h>
16 #include <stdio.h>
17 #include <stdlib.h>
18 #include <string.h>
19
20 #define NRA 2000 /* number of rows in matrix A */
21 #define NCA 2000 /* number of columns in matrix A */
22 #define NCB 2000 /* number of columns in matrix B */
23 // #define N 1000
24 #define EPS 1e-9
```

```c
25 #define SIZE_OF_B NCA*NCB*sizeof(double)
26
27 bool eps_equal(double a, double b) { return fabs(a - b) < EPS; }
28
29 void print_flattened_matrix(double *matrix, size_t rows, size_t cols, int rank) {
30     printf("[%d]\n", rank);
31     for (size_t i = 0; i < rows; i++) {
32         for (size_t j = 0; j < cols; j++) {
33             printf("%10.2f ", matrix[i * cols + j]);  // Accessing element in the 1D array
34         }
35         printf("\n");  // Newline after each row
36     }
37 }
38
39 int checkResult(double *truth, double *test, size_t Nr_col, size_t Nr_rows) {
40     for (size_t i = 0; i < Nr_rows; ++i) {
41         for (size_t j = 0; j < Nr_col; ++j) {
42             size_t index = i * Nr_col + j;
43             if (!eps_equal(truth[index], test[index])) {
44                 return 1;
45             }
46         }
47     }
48     return 0;
49 }
50
51 typedef struct {
52     size_t rows;
53     double *a;
54     double *b;
55 } MM_input;
56
57 char* getbuffer(MM_input *in, size_t size_of_buffer){
58     char* buffer = (char*)malloc(size_of_buffer * sizeof(char));
59     if (buffer == 0)
60     {
61         printf("Buffer couldn't be allocated.");
62         return NULL;
63     }
64     size_t offset = 0;
65     memcpy(buffer + offset, &in->rows, sizeof(size_t));
66     offset += sizeof(size_t);
67     size_t matrix_size = in->rows * NCA * sizeof(double);
68     memcpy(buffer + offset, in->a, matrix_size);
69     offset += matrix_size;
70     memcpy(buffer + offset, in->b, NCA*NCB*sizeof(double));
71     return buffer;
72 }
73
74 MM_input* readbuffer(char* buffer, size_t size_of_buffer){
75     MM_input *mm = (MM_input*)malloc(sizeof(MM_input));
76
77     mm->rows = ((size_t*)buffer)[0];
78     size_t offset = sizeof(size_t);
79     size_t matrix_size =  mm->rows * NCA;
80     mm->a = (double*)malloc(sizeof(double)*matrix_size);
81     mm->b = (double*)malloc(sizeof(double)*matrix_size);
82     memcpy(mm->a, &(buffer[offset]), matrix_size);
83     offset += matrix_size;
84     memcpy(mm->b, &(buffer[offset]), NCA*NCB*sizeof(double));
85     free(buffer);
86     return mm;
87 }
88
89
90 void setupMatrices(double (*a)[NCA], double (*b)[NCB], double (*c)[NCB]){
91     for (size_t i = 0; i < NRA; i++) {
92         for (size_t j = 0; j < NCA; j++) {
93             a[i][j] = i + j;
94         }
95     }
96
97     for (size_t i = 0; i < NCA; i++) {
```

```
 98         for (size_t j = 0; j < NCB; j++) {
 99             b[i][j] = i * j;
100         }
101     }
102
103     for (size_t i = 0; i < NRA; i++) {
104         for (size_t j = 0; j < NCB; j++) {
105             c[i][j] = 0;
106         }
107     }
108 }
109
110 double multsum(double* a,double* b_transposed, size_t size){
111     double acc = 0;
112     for (size_t i = 0; i < size; i++)
113     {
114         acc += a[i]*b_transposed[i];
115     }
116     return acc;
117 }
118
119 double productSequential(double *res) {
120     // dynamically allocate to not run into stack overflow - usually stacks are
121     // 8192 bytes big -> 1024 doubles but we have 1 Mio. per matrix
122     double(*a)[NCA] = malloc(sizeof(double) * NRA * NCA);
123     double(*b)[NCB] = malloc(sizeof(double) * NCA * NCB);
124     double(*c)[NCB] = malloc(sizeof(double) * NRA * NCB);
125
126     /*** Initialize matrices ***/
127     setupMatrices(a,b,c);
128
129     /* Parallelize the computation of the following matrix-matrix
130     multiplication. How to partition and distribute the initial matrices, the
131     work, and collecting final results.
132     */
133     // multiply
134     double start = MPI_Wtime();
135     for (size_t i = 0; i < NRA; i++) {
136         for (size_t j = 0; j < NCB; j++) {
137             for (size_t k = 0; k < NCA; k++) {
138                 res[i * NCB + j] += a[i][k] * b[k][j];
139             }
140         }
141     }
142     /*  perform time measurement. Always check the correctness of the parallel
143         results by printing a few values of c[i][j] and compare with the
144         sequential output.
145     */
146     double time = MPI_Wtime()-start;
147     free(a);
148     free(b);
149     free(c);
150     return time;
151 }
152
153 double splitwork(double* res, size_t num_workers){
154     if (num_workers == 0) // sadly noone will help me :((
155     {
156         printf("Run sequential!\n");
157         return productSequential(res);
158     }
159
160     double(*a)[NCA] = malloc(sizeof(double) * NRA * NCA);
161     double(*b)[NCB] = malloc(sizeof(double) * NCA * NCB);
162     double(*c)[NCB] = malloc(sizeof(double) * NRA * NCB);
163     // Transpose matrix b to make accessing columns easier - in row major way - better cache
       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