ItHPC Lab Report

Delft University of Technology

Henrique Dantas, 4172922
H.N.D.M.P.N.Dantas@student.tudelft.nl

February 22, 2014

1. Lab 1: Intro

This lab consists of introductory exercises to MPI. Therefore the answers to each question consist of only of source code, submitted electronically.

2. Lab 2: Poisson's equation

2.1. Part 1

Step 1

It is simple to understand that the program was indeed executed twice since two pairs of statements are written to the terminal in comparison to one pair before the modifications. Since we are now running the same program in two different nodes this behavior is expected.

The result is the following

```
Number of iterations   : 2355
Elapsed processortime : 1.350000 s
Number of iterations   : 2355
Elapsed processortime : 1.360000 s
```

Step 2

After adding the global variable and the necessary call to `MPI_Comm_rank` using the predefined communicator `MPI_COMM_WORLD`.

```
MPI_Comm_rank(MPI_COMM_WORLD, &proc_rank);
```

The following is printed to the standard output:

```
(0)     Elapsed processortime : 1.360000 s
(0)     Number of iterations   : 2355
(1)     Elapsed processortime : 1.360000 s
(1)     Number of iterations   : 2355
```

Step 3

Step 4

Adjusting the code so each process writes to a separate file does not affect the text displayed, so there is no need to repeat it here. In addition by executing the command

```
diff output0.dat output1.dat
```

I was able to confirm the files are indeed identical.

Step 5

On this step, responsible to ensure correct distribution of information originated from an input file, several statements had to be rewritten. Below is a summary of those changes, in particular the parts that were not completely specified in the exercise manual.

To ensure only process 0 opens the file a simple comparison suffices

```
/* only process 0 may execute this if */
if (proc_rank == 0)
{ ... }
```

To broadcast the data read from the file it is first necessary to explain which fields the `MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm))` function requires.

For our situation the `buffer` pointer should refer to the address of the variable we want to broadcast. The `count` relates to the number of entries in the buffer. The `datatype` should describe the type of data the buffer points to, *e.g.* for integers this should be `MPI_INT`. The `root` is the message broadcaster, in our case node 0. Finally we will use the usual predefined communicator for the last argument `comm`.

Thus the broadcast calls are as follows

```
/* broadcast the array gridsize in one call */
MPI_Bcast(&gridsize      , 2, MPI_INT    , 0, MPI_COMM_WORLD
    );
/* broadcast precision_goal */
MPI_Bcast(&precision_goal, 1, MPI_DOUBLE , 0, MPI_COMM_WORLD
    );
/* broadcast max_iter */
MPI_Bcast(&max_iter      , 1, MPI_INT    , 0, MPI_COMM_WORLD
    );
(...)
/* The return value of this scan is broadcast even though
    it is no input data */
MPI_Bcast(&s, 1, MPI_INT, 0, MPI_COMM_WORLD);
(...)
/* broadcast source_x */
MPI_Bcast(&source_x  , 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
/* broadcast source_y */
MPI_Bcast(&source_y  , 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
/* broadcast source_val */
MPI_Bcast(&source_val, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Step 6

Following the same approach as in the previous section, only the finished version of incomplete code from the manual will be shown in the excerpt.

```
MPI_Comm_size(MPI_COMM_WORLD, &P);
(...)
MPI_Cart_create(MPI_COMM_WORLD, 2, P_grid, wrap_around,
    reorder, &grid_comm);
(...)
/* Rank of process in new communicator */
MPI_Comm_rank(grid_comm, &proc_rank);
/* Coordinates of process in new communicator */
MPI_Cart_coords(grid_comm, proc_rank, 2, proc_coord);
(...)
/* rank of processes proc_top and proc_bottom */
MPI_Cart_shift(grid_comm, Y_DIR, 1, &proc_top, &proc_bottom
    );
/* rank of processes proc_left and proc_right */
MPI_Cart_shift(grid_comm, X_DIR, 1, &proc_left, &proc_right
    );
```

There a couple new function calls on this code whose arguments I will explain next. As explained in the exercise description MPI uses the `MPI_Cart_*` function calls to arrange tasks in a virtual process grid.

To create one the API calls needs the previous communicator, in our case we were using `MPI_COMM_WORLD`. `ndims` and `dims` define the number of dimensions of the grid and the number of processors in each, respectively. For our example these are 2 and `P_grid`. Thereafter the periods specifies if the grid is periodic or not per dimension, and finally if the ranking is reordered or not. These are replaced by the self-explanatory variables `wrap_around` and `reorder`. The new communicator is stored in the address of `comm_cart`. From now on all pointers to communicators refer to this one.

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims
    , int *periods, int reorder, MPI_Comm *comm_cart)
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims,
    int *coords)
int MPI_Cart_shift(MPI_Comm comm, int direction, int displ,
     int *source, int *dest)
```

To define the coordinates of the process in the new communicator we use `MPI_Cart_coords`. Here the rank is necessarily the processor rank (`proc_rank`), the maximum dimensions is 2. The coordinates of each specified process are stored in the `proc_coord` array.

Finally the shift operation returns the shifted source and destination ranks. The direction and displacement quantity arguments are self-evident and are replaced in the program by `X_DIR` or `Y_DIR` and 1 respectively for horizontal and vertical displacements. In accordance to the outputs source and destination are stored in `proc_top` and `proc_bottom` or `proc_left` and `proc_right` depending on the direction.

The text written to the standard output now features the coordinate of each processor.

```
(0) Number of iterations : 2355
(0) (x,y)=(0,0)
(0) Elapsed Wtime:       1.414062 s ( 96.9% CPU)
(1) Number of iterations : 2355
(1) (x,y)=(1,0)
(1) Elapsed Wtime:       1.464844 s ( 95.6% CPU)
```

As one can observe since there are two processor, number 0 is allocated the left half of the grid, and processor with rank 1 deals with the right half.

Step 7

When there are three processors the work can not be evenly split between them. This can be confirmed by inspecting the `x` and `y` variables in the `Setup_grid function`.

For example for processor 2 (in a 3 processor configuration) it is visible that `x` is always negative.

```
(2) x = -30, dim[X_DIR] = 36
(2) y =  71,  dim[Y_DIR] = 102

(2) x = -3, dim[X_DIR] = 36
(2) y = 76, dim[Y_DIR] = 102

(2) x = -28, dim[X_DIR] = 36
(2) y =  26, dim[Y_DIR] = 102
```

Step 8

Step 9

After implementing the collective reduction operation the total number of iterations is indeed the same, as confirmed by the program's output.

```
(0) Number of iterations : 2355
(0) Elapsed Wtime:       1.507812 s ( 96.2% CPU)
(1) Number of iterations : 2355
(1) Elapsed Wtime:       1.496094 s ( 97.6% CPU)
(2) Number of iterations : 2355
(2) Elapsed Wtime:       1.507812 s ( 96.8% CPU)
```

Step 10

2.2. Part 2

2.1

2.2

After changing the code to accommodate for the algorithmic improvement, 5 tests were performed to compare different values for the relaxation parameter ω. The results are summarized in table 2.1. From this data we concluded that 1.93 is the optimal value for the relaxation parameter, accomplishing almost 18 times less iterations than the original with ω equal to one.

ω	`Wtime`[1] (s)	n	Reduction
1.00	1.500000	2355	1.00
1.90	0.222656	220	10.7
1.92	0.199219	165	14.3
1.93	**0.175781**	**131**	**18.0**
1.94	0.289062	142	16.6
1.98	0.351562	419	5.62

Table 2.1: Time, number of iterations obtained and respective iteration reduction for different ω values. The topology used was `pt:`441 with a grid size of `g:`100x100.

2.3

The goal of this exercise is to investigate the scaling behavior of the code with a fixed relaxation parameter. To accomplish that analysis several runs

[1]This value was computed as the maximum `Wtime` over the four individual processor times for each ω value. In addition, in this report all reported time are determined this way, *i.e.* the maximum reported value over the four nodes.

were measured with various grid sizes. In addition different *slices* were also tested.

		`Wtime` (s)		
Grid Size	n	`pt:441`	`pt:422`	δ (%)
200	50	0.164	0.164	0.00
	100	0.226	0.246	8.62
	200	0.343	0.335	-2.27
	300	0.464	0.449	-3.36
400	100	0.437	0.460	5.36
	300	1.054	1.054	0.00
	500	1.746	1.667	-4.47
	1000	3.308	3.304	-0.12
800	100	1.414	1.445	2.21
	300	3.742	3.726	-0.42
	500	5.976	5.988	0.20
	1000	11.625	11.617	-0.07
2000	100	8.218	8.234	0.19
	300	21.351	21.328	-0.11
	500	34.933	35.492	1.60
	1000	68.304	67.824	-0.70

Table 2.2: The maximum time for different grid sizes and different *slicing* arrangements. The δ column compares the relative difference in performance between the two previous columns. The number of iterations for each run was fixed to enable a more accurate comparison. The 200x200 grid size measurements were performed in different conditions in regards to the iteration count since the program converges after 382 iterations.

The results of the aforementioned experiment are shown in table 2.2. As one can obverse the different domain partitions have little impact in the overall performance of the program, even as the grid size increases.

As explained in the exercise manual the (average) time t per iteration n can be parametrized as follows

$$t(n) = \alpha + \beta \cdot n \tag{2.1}$$

Where α and β are arbitrary constants. To determine these constants we

will use the least-squares method. The results of applying this technique[2] to the available dataset are shown in table 2.3

	pt:441		pt:422	
Grid Size	α	β	α	β
200	0.105	0.001	0.120	0.001
400	0.116	0.003	0.115	0.003
800	0.310	0.011	0.329	0.011
2000	1.449	0.067	1.721	0.066

Table 2.3: Using least-squares method we estimated α and β as defined in equation 2.1. The dataset adopted for this computation is present in table 2.2.

In order to improve the quality of this an estimation a thorougher study should be performed with more runs. In addition considering other (non-linear) parametrization functions, *e.g.* exponential, may also yield interesting results.

To conclude this question we plotted the estimated functions along with the actual data points to facilitate a visual comparison of the estimation.

As figure 2.1 shows the estimated α and β can accurately determine the time necessary to complete the operation. This can be observed by noticing that the measured data points (depicted as -△- and -□- for the two different partitions) are located close to the curves (which represent the predicted values).

2.4

Based on the results from the previous question it is expected that the division of the domains does not result is significantly different running times. Thus, the choice can be done arbitrarily. In order to estimate the constants without performing further experiments one can extrapolate based on the already calculated values.

For 8 processors the area allocated to each processor is reduced 4 times in comparison with 4 processors. Therefore it is reasonable to assume that the time per iteration (β)is reduced by the same ratio. On the other the communication overhead (α) is not expected to experience drastic changes. After identifying this assumptions it is trivial to estimate the new constants.

[2]The Google Drive implementation of least-squares, *LINEST* function, was used for this computation. For more information please see `https://support.google.com/drive/answer/3094249`.

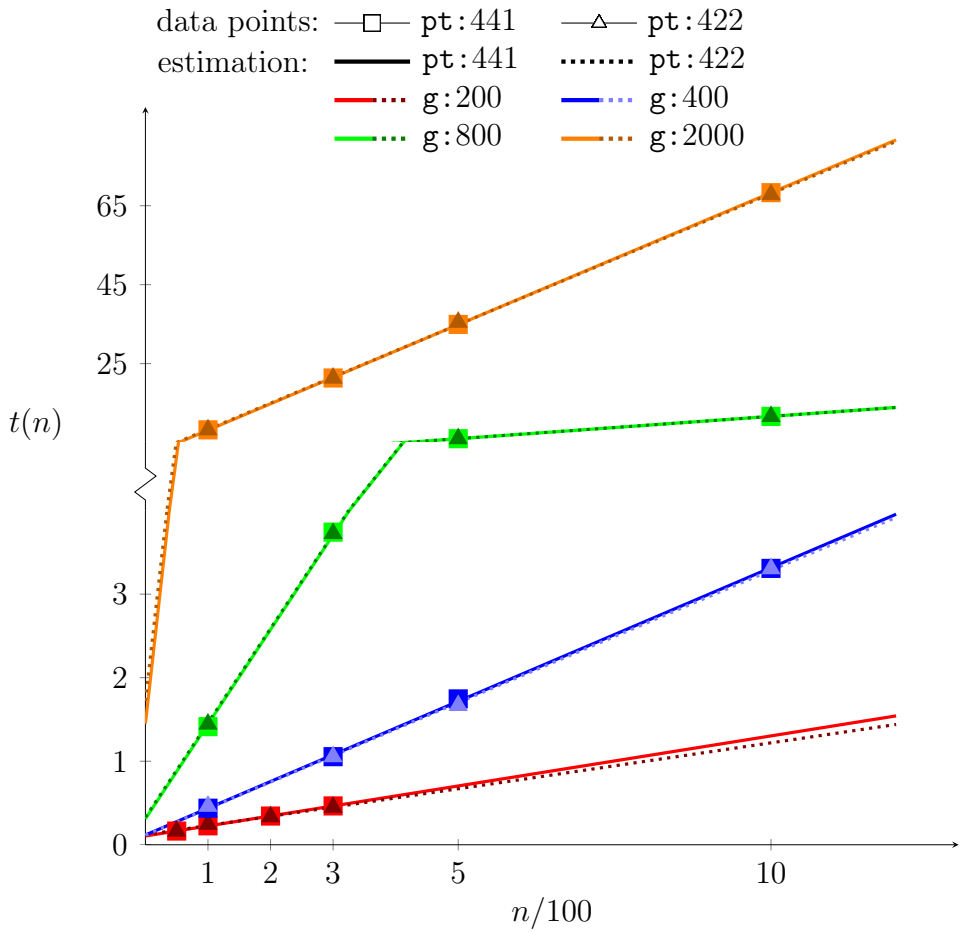

Figure 2.1: Visual comparison between the experimental data and respective linear estimation using the least-squares method.

$$\alpha_8 = \alpha_4 \tag{2.2}$$
$$\beta_8 = \beta_4/4 \tag{2.3}$$

Table 2.4 incorporates equations 2.2 and 2.3 to predict the new constants.

	pt:441		pt:422	
Grid Size	α_8	β_8	α_8	β_8
200	0.105	0.00029	0.120	0.00027
400	0.116	0.00080	0.115	0.00079
800	0.310	0.00283	0.329	0.00282
2000	1.449	0.01671	1.721	0.01657

Table 2.4: The estimated α and β constants for an 8 processor configuration. These results are based on the data from table 2.3 and equations 2.2 and 2.3.

2.5

Table 2.5 features the number of iterations necessary for several grid sizes.

g	n
200	382
300	771
400	1206
500	1664

Table 2.5: Number of iterations necessary to solve the Poisson equation for various grid sizes. The topology used was `pt:441`.

Using the same technique as before we can parametrize the iteration evolution according to grid size to a linear equation. For the aforementioned data we obtained the following constants $\alpha = -492.6$ and $\beta = 4.281$. Thus equation 2.5 can be used to estimate the number iterations for different grid sizes.

$$n(g) = \alpha + g \cdot \beta = -492.6 + 4.281 \cdot g \tag{2.4}$$
$$\tag{2.5}$$

To finish table 2.6 highlights the errors between the predictions and the measured values, and estimates the number of iterations for higher grid sizes.

g	n	n_{est}	δ (%)
200	382	364	4.82
300	771	792	-2.68
400	1206	1220	-1.14
500	1664	1648	0.97
1000	-	3788	-
5000	-	20912	-
10000	-	42317	-

Table 2.6: Comparing empirical and estimated data for iteration evolution with increasing grid sizes.

The conclusion that can be drawn from table 2.6 is that the iteration evolution is approximately linear with the grid size. This claim is supported by the low relative error between predicted and actual data points, in particular as dimensions increase. Nonetheless in order to make a stronger claim it is important to perform a more refined study (*i.e.* increase the data set) as we only consider four point.

2.6

Figure 2.2 depicts the evolution of the error with increased iterations for a 500x500 grid size. It is worthy to mention that the error reduces drastically until approximately the hundredth iteration. From that point on the rate of descent is decreased

2.7

2.8

Changing the number of red/black sweeps between border exchanges has a significant impact both on the total number of iterations and the total running time, as exhibited in table 2.7. As the work between communication steps increases the number of iterations to converge declines, however the time per iteration augments. The issue, is that the latter rate is higher than the former which results in a global execution time penalty.

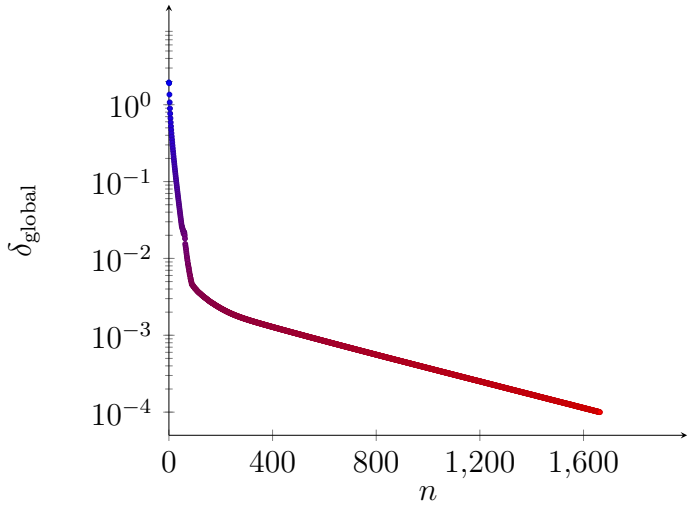

Figure 2.2: The evolution of δ_{global} for a grid size of 500 and a 41 processor topology. The blue color represents high error while red illustrates errors close to the precision goal, 10^{-4}. When this value is reached no more iterations occur. It is important to note that the y axis is logarithmic.

Sweeps	n	`Wtime` (s)
1	1664	7.945312
2	927	8.125000
3	732	9.191406
5	624	12.808594

Table 2.7: Analyzing the execution time as the number of sweeps between communication steps evolves. n portrays the number of iterations necessary for convergence. These results were obtained with the following configuration `g:`500x500 and `pt:`441.

2.9

To incorporate the optimization indicated in the question an extra variable, `aux`, was added. The respective code, in particular the inner `for` loop was updated to the following.

```
for (x = 1; x < dim[X_DIR] - 1; x++) {
  aux = (x + offset[X_DIR] + offset[Y_DIR] + parity + 1) % 2;
  for (y = 1 + aux; y < dim[Y_DIR] - 1; y += 2)
    if (source[x][y] != 1)
```

```
    {
      ...
    }
  }
}
```

The added variable along with the adjustment to the increment of `y` obviates the need to explicitly check the parity in the subsequent `if` statement. More importantly it reduces the number of iterations of the inner loop. Table 2.8 compares the running times foe before and after the optimization.

gs	n	$\texttt{Wtime}_{\texttt{before}}$ (s)	$\texttt{Wtime}_{\texttt{after}}$ (s)	δ (%)
200	382	0.56	0.54	-4.17
400	1206	3.80	3.54	-6.69
800	3003	33.32	29.44	-11.65
2000	3770	246.54	210.63	-14.56

Table 2.8: Results before and after implementing the inner loop optimization for various grid sizes. The same topology as before was used for this measurements: `pt:`441.

The conclusion from the results presented in table 2.8 is that the optimization has a moderate but important impact on the overall performance. This difference becomes more evident as the grid size increases.

For the remaining of this document the "optimized" version of the code will be used.

2.10

2.11

To investigate the impact of the `Exchange_Borders` function the code on the performance the total time spent on this function as well as the amount of data transmitted were studied. To account for the time we added calls to the `resume_timer` and `stop_time` functions at the beginning and end, respectively of `Exchange_Borders`. On the other hand to measure the amount of data send we simply added the appropriate value to a global counter for each time the function under studied is called. This value is easy to obtain as each `MPI_Send` command sends `dim[X_DIR] - 2` or `dim[Y_DIR] - 2` `MPI_DOUBLE`s.

Table 2.9 summarizes the empirical results. As the number of iterations for the two 4-node topologies studied (*i.e.* `tp:`441 and `tp:`422) is the same,

the amount of data transmitted is also the same. Thus the timing results are also similar and consequently only the former is considered on the table.

`gs`	Comm time per node (s)	Total data sent (MB)	Bandwidth (MB/s)
200	0.20	11.66	59.69
400	0.72	73.61	102.41
800	2.92	366.58	125.46
2000	6.53	1 150.51	176.16

Table 2.9: Total amount of data sent[3] and individual communication time in the `Exchange_Borders` function.

2.12

The impact of the communication overhead is limited. Even for large grid sizes halving the amount of data sent is likely to cause, at best, moderate improvements. Given the code changes necessary to accomplish that goal we conclude it is not worth the effort. In addition the maintainability of the code would decrease rapidly as such modifications would hamper readability and comprehension of the code base.

Nonetheless to accommodate this optimization the address of the first point to exchange would have to be calculated differently, taking into account the parity value. Moreover the size of the "jump" between transmittable points would also need to be modified. Such a change would best be done in the `Setup_MPI_Datatypes()` where the custom data types are defined. The greatest advantage would be the 50% reduction number of points to exchange.

2.13

[3]The shown values only consider sent data. To determine the total traffic (*i.e.* data sent and data received) one should simply double those numbers.

3. Lab 5: Matmul

This section is dedicated to the implementation of the general matrix multiplication (GEMM). Equation 3.1 describes the operation. A, B and C are the matrices to be multiplied and the result respectively, while α and β are scalar coefficients. The developed code was built upon the provided sequential implementation (`matmul.c`). It uses the Message Passing Interface (MPI) for inter-node communication and Open Multi-Processing (OpenMP) for intra-node computations. Unless, specifically specified the code is running on 4 nodes from DAS3. This assumption facilitates the explanation of the proposed solution although in the C code implementation the number of nodes can be chosen arbitrarily as long as the grid can be evenly distributed among them. Finally the compiler used was `gcc 4.3.2` invoked with several optimization flags that shall be enumerated in 3.2.

$$C = \alpha AB + \beta C \tag{3.1}$$

This section is organized as follows: in subsection 3.1 the higher level architecture of the code is explained. In essence it covers how the matrices are partitioned and divided among processors to enable parallel computation and what communication steps are necessary top obtain the final result. Subsequently subsection 3.2 will be focused on the performance analysis of this implementation.

3.1. Architecture

The first step is to divide the matrices evenly among the 4 nodes available. The subscripts in equations 3.3-3.7 indicate the node responsible for the subset.

$$C = \alpha AB + \beta C \tag{3.2}$$

$$\begin{bmatrix} C_0 & C_1 \\ C_2 & C_3 \end{bmatrix} = \alpha \begin{bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{bmatrix} \cdot \begin{bmatrix} B_0 & B_1 \\ B_2 & B_3 \end{bmatrix} + \beta \begin{bmatrix} C_0 & C_1 \\ C_2 & C_3 \end{bmatrix} \tag{3.3}$$

$$C_0 = \alpha(A_0B_0 + A_1B_2) + \beta C_0 \tag{3.4}$$
$$C_1 = \alpha(A_0B_1 + A_1B_3) + \beta C_1 \tag{3.5}$$
$$C_2 = \alpha(A_2B_0 + A_3B_2) + \beta C_2 \tag{3.6}$$
$$C_3 = \alpha(A_2B_1 + A_3B_3) + \beta C_3 \tag{3.7}$$

Therefore each node requires two parts from each of A and B to calculate its share of C. Moreover it is not surprising to find that the way A and B are allocated is different, while A is partitioned horizontally, B is sliced vertically. For simplicity purposes, in the actual implementation each node initializes the matrices independently but coherently although it only uses the subsets it needs. Alternatively this could be done using MPI. For example node zero would initialize all matrices and send the respective parts to the remaining three nodes.

The next step involves three operations per node: two multiplications and one addition. The multiplication are realized in a similar fashion to the sequential program, *i.e.* with three nested loops. However an OpenMP directive is used to parallelize the outer `for` loop. After some experimentation it was concluded that applying a single directive to the outer loop yielded the best performance. The following OpenMP directive is responsible for that task.

```
#pragma omp parallel for default(none) private(...)
  shared(...)
```

Other options for parallelizing the loop, shown below, were studied but with less interesting results.

```
#pragma omp for schedule(dynamic) nowait
#pragma omp for schedule(dynamic, chunk) nowait
#pragma omp for schedule(runtime) nowait
```

The major final step is to aggregate all values back into a single node. This is accomplished through calls to `MPI_Send()` and `MPI_Recv()`. Node zero was elected to collect the data from the remaining nodes. Thereafter it compiles all of it, including its own share, in a single 2D array and writes the result to a text file.

In summary three main steps were necessary to fulfill the task at end:

1. Divide evenly the input matrices A and B between nodes.
2. Each node computes the required operations over its designated subsets.
3. Aggregate all results in a single node and write result to a file.

Although the high level overview is strikingly simple the implementation was not so straightforward. As always the devil is in the details and in particular properly handling the indices of each matrix in an abstract way proved to be quite cumbersome.

3.2. Results

In the text that follows the results of the previously described implementation will be presented and analyzed.