## MPI Programming Assignment

# MPI implementations for the Laplace-2D algorithm

Josep Castell & Daniel Salgado

Parallel programming. Course 2017/2018.

## Contents

# 1 Basic MPI implementation

**Task1**. *Implement a working version of this algorithm (remember that you can assume that m is divisible by N). The code must be properly documented. You must include all the necessary information for running your code and indicate the test cases you have used.*

In this section we describe the different parts of the base MPI implementation code in C that we have developed. Then in section 1.1 we will discuss about how to compile, run and test that our code actually works.

We consider a squared 2D grid `A`, $n \times n$. If we have $N \geq 2$ processes, labelled $0, 1, .., N-1$, to which we can distribute computational work, we assume that $n$ is divisible by $N$ [1]. The idea is to divide the grid `A` in $N$ parts and distribute $n/N$ (contiguous) rows to each process. Moreover, the process $i$ needs the last row of the process $i-1$ and the first row of the process $i+1$ (if $i = 0$ or $i = N-1$, then only one additional row is needed). In our approach, each process will have a local matrix `my_A` with dimensions $(n/N+2) \times n$ containing the mentioned set of rows.

Before we analyse our C implementation it may be useful to have a look at the pseudo-code showed in Algorithm 1.

Now we describe the code step by step. The whole code can be found in Appendix A.1:

- First we initialise global and local variables:

```c
// Initalisation of variables
  double t0, tf; /*Initial and final time counters*/

  int n = 4096; /* Size of the grid n x n */
  int iter_max = 1000; /* Number of iterations */
  float *A, *temp; /* Pointers to grid A and temp */

  const float tol = 1.0e-5f; /* Tolerance */
  float error= 1.0f; /* Global error variable */

  int numtasks, rank, tag = 1,rc; /* # of processes, process ID, tag, rc */
  int  my_nrows, my_size; /* Number of rows of my_A, dimension of my_A*/
  float *my_A, *my_temp; /* Portion of A carried by each process*/
  float my_error= 1.0f; /* Error for each process*/
  MPI_Status Stat; /* MPI status variable to control the status*/

  int rowstart, rowend, nrows; /*Auxiliar variables related to rows*/
```

[1] If the grid was not squared, we would require the number of rows to be divisible by $N$.

**Algorithm 1** Basic MPI implementation of the Laplace-2D problem

```
procedure main (args)
    Initialisation of variables. ▷ All processes declare/initialise global and local variables
    Initialise MPI environment. ▷ MPI_init(args), MPI_Comm_size(...,numtasks), MPI_Comm_rank(..., rank)
    if rank = 0 then
        Get runtime arguments
        Allocate memory for A and temp.
        Initialise A and temp (apply boundary conditions).
    end if

    iter ← 0
    Broadcast the global (master) variables needed by other processes ▷ MPI_Bcast
    Initialise auxiliar variables
    Allocate memory for my_A and my_temp
    Distribute the rows of A and temp from the master to the other processes and store in my_A and my_temp, respectively. ▷ MPI_Scatter

    while error > $tol^2$ and iter < iter_max do
        iter ← iter + 1
        if rank > 0 then
            Send the first row of the process rank to the process rank-1 ▷ MPI_Send
            Process rank receives the last row from the process rank-1. ▷ MPI_Recv
        if rank < numtasks-1 then
            Process rank receives the first row from the process rank+1. ▷ MPI_Recv
            Send the last row of the process rank to the process rank+1. ▷ MPI_Send
        Assign to each process the initial and final rows from my_A to be computed. ▷ rowstart, rowend
        my_error= my_laplace_step(my_A, my_temp, nrows, n, rowstart, rowend). ▷ Update the portion of matrix assigned to each process and store the maximum error of this portion in my_error
        Reduction operation: the maximum among all my_error from all processes is calculated and stored in the global variable error ▷ MPI_Reduce
        Swap the roles of my_A and my_temp(double buffer) to be prepared for the next iteration.
    end while

    The master process gathers all the final portions of A stored in my_A of each process to build the matrix A corresponding to the last iteration. ▷ MPI_Gather
    if rank = 0 then
        error ← $\sqrt{error}$
        Print information such as the total final error, number of iterations, time elapsed,...
    Finalize MPI environment. ▷ MPI_Finalize
end procedure
```

- Next the MPI enviroment is initialised. If the number of processes is not equal or greater than 2, the application is aborted.

```
//INIT MPI environment
  rc = MPI_Init (&argc, &argv);
  if (rc != MPI_SUCCESS)
    {
      printf (”Error␣starting␣MPI␣program.␣Terminating.\n”);
      MPI_Abort (MPI_COMM_WORLD, rc);
      return -1;
    }
  MPI_Comm_size (MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  //END basic INIT MPI environment


  // Abort the program if the number of processes is less than 2
  if(numtasks < 2){
    printf (”This␣program␣works␣with␣2␣or␣more␣processes␣(-np␣N␣with␣N␣>=2).\n”);
    MPI_Abort (MPI_COMM_WORLD, 1);
    return -1;
  }
```

- The master process updates, if necessary, the grid size and the total number of iterations; allocates memory for `A` and `temp`, and initialises it applying some particular boundary conditions.

```
//BEGIN MASTER Initialisation of A, temp and initial time
  if(rank == MASTER){
  t0 = MPI_Wtime(); //Record the initial time

  // get runtime arguments
  if (argc>1) {  n        = atoi(argv[1]); }
  if (argc>2) {  iter_max = atoi(argv[2]); }

  // Allocate memory for A and temp
  if( ( A = (float*) malloc(n*n*sizeof(float)) ) == NULL ){
    printf (”Error␣when␣allocating␣memory␣for␣A.\n”);
    MPI_Abort (MPI_COMM_WORLD, 1);
    return -1;
  }
  if( ( temp = (float*) malloc(n*n*sizeof(float)) ) == NULL ){
    printf (”Error␣when␣allocating␣memory␣for␣temp.\n”);
    MPI_Abort (MPI_COMM_WORLD, 1);
    return -1;
  }

  // set boundary conditions
  laplace_init (A, n);
  laplace_init (temp, n);
  A[(n/128)*n+n/128] = 1.0f; // set singular point

  printf(”Jacobi␣relaxation␣Calculation:␣%d␣x␣%d␣mesh,”
         ”␣maximum␣of␣%d␣iterations\n”,
         n, n, iter_max );
  } //END MASTER initialisation
```

- All processes initialise a iterations' counter and the global `error`, grid size `n` and maximum number of iterations `iter_max` is broadcast from the master process.

```
  int iter = 0;

  MPI_Bcast(&error , 1, MPI_FLOAT , MASTER , MPI_COMM_WORLD);
  MPI_Bcast(&n, 1, MPI_INT , MASTER , MPI_COMM_WORLD);
  MPI_Bcast(&iter_max , 1, MPI_INT , MASTER , MPI_COMM_WORLD);
```

- Some auxiliary variables related to the size of the portions of grid of each process are initialised.

```
  my_nrows = n/numtasks;
  nrows = my_nrows +2;
  my_size = n*(my_nrows+2);
```

- Each process allocates memory for its `my_A` and `my_temp`.

```
  if( ( my_A = (float*) malloc( my_size*sizeof(float)) ) == NULL ){
    printf (”Error␣when␣allocating␣memory␣for␣my_A.\n”);
    MPI_Abort (MPI_COMM_WORLD , 1);
    return -1;
  }
  if( ( my_temp = (float*) malloc(my_size*sizeof(float)) ) == NULL
     ){
    printf (”Error␣when␣allocating␣memory␣for␣my_temp.\n”);
    MPI_Abort (MPI_COMM_WORLD , 1);
    return -1;
  }
```

- The rows of A from the master are distributed to all processes, who then store them in the appropriate positions of my_A. At the end, the matrix my_A is copied to my_temp.

```
MPI_Scatter(A, my_nrows*n,  MPI_FLOAT, my_A+n, my_nrows*n, MPI_FLOAT, MASTER, MPI_COMM_WORLD);
float * my_temp_plusn = my_temp+n;
copy_A_to_temp(my_A+n, &my_temp_plusn, my_nrows, n);
```

- Now a while loop is started and run as long as error $>$ tol$^2$ and iter $<$ iter_max, where tol is the tolerance we fix. We describe the different parts of code inside the loop:
  - The iteration counter is updated by all processes.
  - All processes except for the MASTER process send the first row to the previous process and receive the last row from the previous process.
  - All processes except for the LAST one receive the first row from the next process and sent its last row to the next process.

```
    iter++;
    /*Send and Recv calls so that each process obtain two additional rows needed for
    the computation of the new values.
    */

    if(rank > MASTER){
      //For all the processes apart from MASTER, which does not need a previous row

      /* Send the first row of the process 'rank' to the process 'rank-1'*/
      MPI_Send(my_A+n, n, MPI_FLOAT, rank-1, tag ,MPI_COMM_WORLD);
      /* Process 'rank' recieves the last row from the process 'rank-1'*/
      MPI_Recv(my_A   , n, MPI_FLOAT, rank-1, tag ,MPI_COMM_WORLD, &Stat);
    }
    if(rank < numtasks -1 ){
       //For all the processes apart from THE LAST, which does not need a 'last' row

       /* Process 'rank' recieves the first row from the process 'rank+1'*/
       MPI_Recv( (my_A + n*(my_nrows+1) )   , n, MPI_FLOAT, rank+1, tag ,MPI_COMM_WORLD, &Stat);
       /* Send the last row of the process 'rank' to the
          process 'rank+1'*/
       MPI_Send(  (my_A + n*(my_nrows))  , n, MPI_FLOAT,
          rank+1, tag ,MPI_COMM_WORLD);
    }
```

- Only the internal points of the original grid `A` have to be updated, i.e., the values of the boundary points have to remain constant over iterations. Since the MASTER and LAST processes contain whole row of boundary points, we have to consider extra cases to tell from which initial row to which final row the local portion of the grid each process has to update.
- Then, each process performs the classical Jacobi or Laplace step to update the points of `my_A` that are interior points of `A`. The local error of each process' portion is stored in `my_error`.

```
    if(rank == MASTER){
      rowstart =2;
      rowend = nrows-1;
    }
    else if(rank == (numtasks - 1)){
      rowstart = 1;
      rowend = nrows -2;
    }
    else{
      rowstart = 1;
      rowend = nrows-1;
    }

    my_error= my_laplace_step(my_A, my_temp, nrows, n,
       rowstart, rowend);
```

- Then each process calls the `MPI_Reduce` function so that after all processes have finished the computations, a reduction operation is performed to update the value of the global variable `error` in order to contain the maximum among the `my_error` values coming from all processes.
- Finally, the roles of `my_A` and `my_temp` are swaped (double buffer) to be prepared for the next iteration.

```
    MPI_Reduce(&my_error, &error, 1, MPI_FLOAT, MPI_MAX,
       MASTER, MPI_COMM_WORLD);

    float *swap= my_A; my_A=my_temp; my_temp= swap;
```

- After the while loop ends, thee master process gathers all the final portions of `A` stored in `my_A` of each process to build the matrix `A` corresponding to the last iteration performed.

```
MPI_Gather(my_A+n, my_nrows*n ,  MPI_FLOAT, A, my_nrows*n,
   MPI_FLOAT, MASTER, MPI_COMM_WORLD);
```

- Finally the MASTER process computes the final error by taking the square root of `error` and, if desired, can print some running information such as the total elapsed time, the final error,...
- The MPI environment has to be finalized.

```
if(rank == MASTER){
  error = sqrtf( error );
   printf("Total␣Iterations:␣%5d,␣ERROR:␣%0.6f,␣", iter, error);
   printf("A[%d][%d]=␣%0.6f\n", n/128, n/128, A[(n/128)*n+n/128]);
   free(A); free(temp);
 }
 if(rank == MASTER){
  tf = MPI_Wtime();
  printf("Elapsed␣time,␣%2.5lf\n", tf-t0);
 }
 //Finalize the MPI environment.
 MPI_Finalize();
 return 0;
```

## 1.1 Program running and test cases

The detailed description, step by step, of how to compile and execute the code can be found in our GitHub repository [7]. Basically, the commands to compile and run the code are the following:

```
module load gcc/6.1.0
module load mpe2/mpi-1.10.2/2.4.8
mpicc -g -lm -fopenmp -o mpi_lapFusion lapFusion_mpi.c
mpirun -np N mpi_lapFusion n iter_max
```

where $N$ is the number of processes that we want to use (if we run this in a single computer, the maximum $N$ would depend on the number of cores or total number of threads that it can deal with); $n$ is the size of the grid, and `iter_max` the maximum number of iterations.

During the development of our MPI implementation, several tests have been performed to check that the program does what we want and expect. The main test or debugging approaches we have done are the following:

- Put some print statements to see if the broadcast of variables has succeeded, and thus all processes have the correct values of the broadcast variables.

- Put a print statement to see if the `my_A` grid of each process is the appropriate one. This involves to compare all `my_A`s with the original `A` and see that in fact, process $i$ has the $i$-th portion of `A` if we start counting from the first row (from up to down). With this test we have assured that the `MPI_Scatter` is performed properly.
- The next important test was considered once the code for the while loop was completed. The goal was to check if after the `MPI_Gather` call, the final (iteration) matrix `A` contained the correct values. This can be achieved by running the original baseline code `lapFusion.c` using the same input values and comparing the final matrices of both programs (the basic MPI and the baseline).
- Similarly, we compared the final global errors obtained by our MPI implementation and the baseline one.

If one would like to run the mentioned tests, one should have to download our GitHub repository [7] and `checkout` the appropriate commits containing test codes. Concretely, the commits that correspond to initial the debugging and test stages are those from or before the 30th of November (2017).

# 2 Optimizing the basic MPI implementation

**Task2**. *Implement a working version of one of these mechanisms (you can make assumptions about the matrix dimensions [m, n] and the number of processes N if needed). You must justify your selection, which means that you must discuss the perceived pros and cons of all methods. The code must be properly documented. You must include all the necessary information for running your code and indicate the test cases you have used.*

## 2.1 Discussion of possible alternatives

Now we will enumerate the pros and cons of implementing the proposed optimitzations. We pretend to justify our pick. We did not choose the optimitzation that we thought it would work better, we picked the one that seemed easiest to us.

1. **Overlapping communication and computation**

   Pros: Simple algorithm, and does not require great modifications to the core functions.

   Cons: Requires more skill and knowledge of MPI message manipulation, and a good understand of the send/receive actions in order to implement this optimization into the base code.

2. **Block partitioning**

   Pros: Very simple algorithm, the code should to the same than the base code, but instead of working with rectangles it should work with squares.

   Cons: Hard to code, send messages mixing rows and columns seemed hard to us.

3. **Block communication**

   Pros: Easy to implement into the code once the procedure is understood.

   Cons: Hard to understand.

4. **Hybrid solution**

   Pros: Easiest optimization to implement in the code

   Cons: Executing the code would have required complex requests to the system.

## 2.2 Optimized MPI implementation

Our optimized MPI implementation consists in applying the above mentioned **Block communication** approach. The corresponding algorithm is almost exactly to Algorithm 1, with the only difference being that now we are not sending and receiving only first and last rows from processes, but now we are considering sends and receives for the last and first $k$ rows, with $k \geq 1$ and $k \ll n/N$. Thus, note that the case $k = 1$ coincides with the basic MPI implementation described during section 1. We do not describe the whole code (found in the Appendix A.2), since it is quite the same to the basic code described in the above sections, with minor changes.

An advantage of this implementation is that we can use the same code to run the Basic and the Optimized version just by setting $k = 1$ or $k > 1$, respectively.

## 2.3 Program running and test cases

The detailed description, step by step, of how to compile and execute the code can be found in our GitHub repository [7]. Basically, the commands to compile and run the code are the following:

```
module load gcc/6.1.0
module load mpe2/mpi-1.10.2/2.4.8
mpicc -g -lm -fopenmp -o mpi_lapFusion lapFusion_mpi_opt.c
mpirun -np N mpi_lapFusion_opt n iter_max k
```

where $N$, $n$ and `iter_max` are the same parameters for the basic implementation (see section 1.1), and now we need an extra argument $k \geq 1$ that stands for the number of first and last rows that are interchanged between processes.

During the development of this optimized MPI implementation, we have performed the same tests that we considered for the basic implementation, now varying the value of $k$ (see section 1.1).

# 3 Assesment of the MPI implementations

**Task 3**. *Make a performance anlysis of your program versions using the given hints and the support of the performance analysis tools available in the lab. You must present an organized explanation of this analysis.*

## 3.1 Strong and Weak scalability efficiencies and Speedup

To study the strong and weak scalability we follow the definitions introduced in [6]:

- **Calculating Strong Scaling Efficiency**: *If the amount of time to complete a work unit with 1 processing element is $T_1$, and the amount of time to complete the same unit of work with N processing elements is $T_N$, the strong scaling efficiency (as a percentage of linear) is given as*:
$$\frac{T_1}{N \cdot T_N} \times 100 \tag{1}$$
- **Calculating Weak Scaling Efficiency**: *If the amount of time to complete a work unit with 1 processing element is $T_1$, and the amount of time to complete N of the same work units with N processing elements is $T_N$, the weak scaling efficiency (as a percentage of linear) is given as:*
$$\frac{T_1}{T_N} \times 100 \tag{2}$$

To study the speedup we consider the so called **Amdahl's law** [4], that states a relation to compute the Speedup of a parallel program when compared to the serial version of the same program:

$$\mathsf{Speedup} = \frac{1}{\dfrac{P}{N} + S} \approx \frac{T_1}{T_N} \tag{3}$$

where $N$ is the number of workers that distribute work in parallel regions (cores or threads in our case), $P \in [0, 1]$ de fraction of code that is parallelised, and $S = 1 - P$ the portion of code that is not parallelised (serial). And $T_1$, $T_N$ are the execution times of the serial version and the parallelised version of the program with N workers, respectively. Note that the expected (ideal) speedup, would be given by setting $P = 1$ and $S = 0$, i.e. $\mathsf{Speedup} =$ N, which is linear with the number of workers.

### 3.1.1 Performance of a single node

In this section we analyse the strong and weak scaling efficiencies for the performance of a single node with up to 4 cores, given the properties of the computer we use (`aolin21`). For simplicity we only analyse the case $k = 1$, which is the basic MPI implementation that we presented in section 1. Here we will focus on study the effect of the problem size, in terms of the grid size parameter $n$, on the speedup and the strong and weak scaling efficiencies. In the next section 3.1.2 we will consider both more than one node and more values for $k$.

#### Speedup

Similarly as we did when we studied an OpenMP implementation of the Laplace-2D algorithm, although the processor has only 4 cores, we consider up to 8 threads, since in principle each core can deal with 2 threads. The results for different values of $n$ are shown in Figure 1.

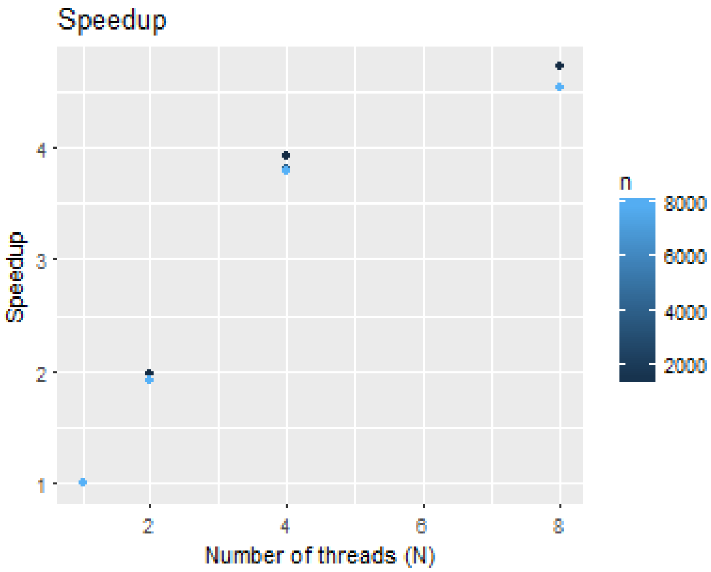

Figure 1: Speedup as a function of the number of threads, $N \in \{1, 2, 4, 8\}$, and for $n \in \{1024, 2048, 4096, 8192\}$, `iter_max` $= 100$.

We observe that for 1, 2 and 4 cores the speedup is almost the ideal one, i.e., `speedup` = N. On the contrary, when $N = 8$ we are exploiting the four cores by using 2 threads from each one, and we observe the same as when we analysed the OpenMP implementation: using 2 threads for each of the 4 cores the speedup is improved, but quite less than what it would be improved if we have had 8 real cores.

Another important result is that the speedup is observed to scale worse as the problem size (in our study in terms of $n$) increases. That is what we can see from Figure 1: smaller sizes lead to greater speedup regardless of the value of $N$.

### Strong scaling

In Figure 2 we have represented the strong scaling efficiency as a function of $N$ and for the same set of $n$ values we used for the speedup study.

It is harder to achieve good strong-scaling at larger process counts since the communication overhead tends to increase in proportion to the number of processes used.

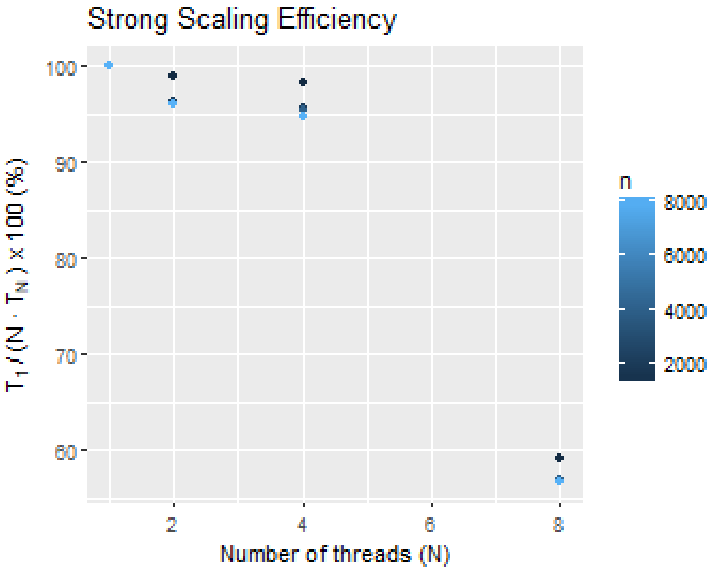

Figure 2: Strong scaling efficiency as a function of the number of threads, $N \in \{1, 2, 4, 8\}$ (see equation (1)), and for $n \in \{1024, 2048, 4096, 8192\}$, `iter_max` $= 100$.

The results we have obtained now are quite similar to those we have seen for the speedup: for 1, 2 and 4 cores the strong scaling efficiency is above the 95 %, i.e., very close to the ideal value; and, on the contrary, when considering 8 threads the strong scaling efficiency drastically drops to 60% or below. Again, we see that the strong scaling efficiency scales better for smaller problem sizes.

### Weak scaling

In Figure 3 we have represented the weak scaling efficiency as a function of $N$.

Given the problem size $n^2 \times \texttt{iter}$ we want the quantity

$$\frac{n^2 \times \texttt{iter}}{N} \tag{4}$$

to stay constant regardless of the value of $N$ (we explain what this exactly means in a while). For simplicity we have fixed $\texttt{iter} = 100$ again. Thus, the condition to be satisfied is the following one:

$$\frac{n_1^2 \times 100}{1} = \frac{n_N^2 \times 100}{N} \Longleftrightarrow n_N = \sqrt{N} n_1 \tag{5}$$

The grid size for a single core/thread is chosen to be $n_1 = 2048$. Then $n_2 = \sqrt{2} \times 2048 \approx 2896.309 \approx 2896$ (we get rid of of the decimals so that the number is divisible by 8, 4, 2); $n_4 = \sqrt{4} \times 2048 = 4096$; and $n_8 = \sqrt{8} \times 2048 \approx 5792.619 \approx 5792$, again divisible by 8, 4 and 2.

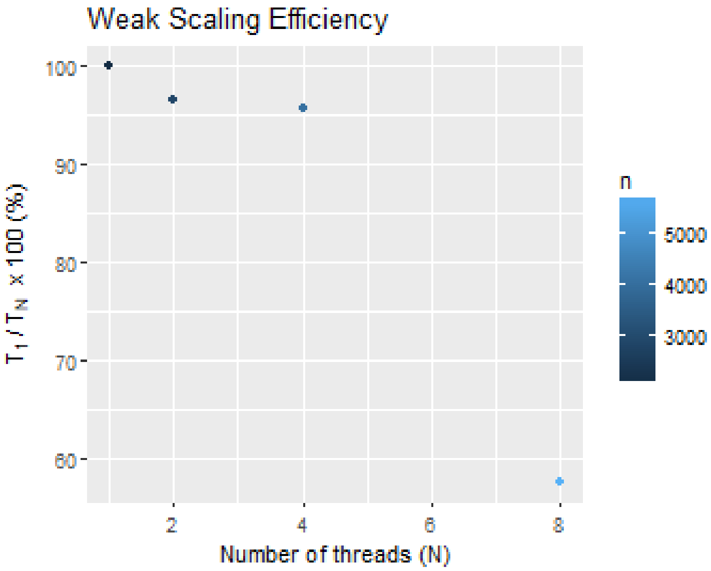

Figure 3: Weak scaling efficiency as a function of the number of threads, $N \in \{1, 2, 4, 8\}$ (see equation (2)), and for $n \in \{2048, 2896, , 4096, 5792\}$, `iter_max` $= 100$.

We observe that for $N \leq 4$ the weak scaling efficiency behaves like the strong scaling efficiency, it stays above the 95% of the ideal value. Moreover, similarly to the strong scaling efficiency, when considering 8 threads the weak scaling efficiency drops to below 60%.

#### 3.1.2 Performance of single and multiple nodes

In this section we analyse the strong and weak scaling efficiencies and speedup for the performance of a single node with up to 4 cores and for the performance of two nodes, also with up to 4 cores. For simplicity we fix the following problem size: $n = 4096$ and `iter_max` $= 1000$. Also, we are going to analyse the optimized version of our MPI implementation for the cases $k \in \{1, 4, 20\}$.

For us the number of resources will be equivalent to the total number of cores used, regardless of the number of nodes we are considering (but we explicitly specify if we are using more than one node). The experimental set-up to study the scaling efficiencies and speedup is the following:

- When considering 1, 2 and 4 cores we are using the same computer (node).
- When considering 8 cores, we are using 2 computers from the LAB cluster, and thus all their cores.
- For one case in the study of the weak scaling efficiency, we consider 4 nodes using its 4 cores each one (so in this way we obtain a total of 16 cores).

The jobs have been submitted with exclusivity.

We can see the results depicted in Figure 4.

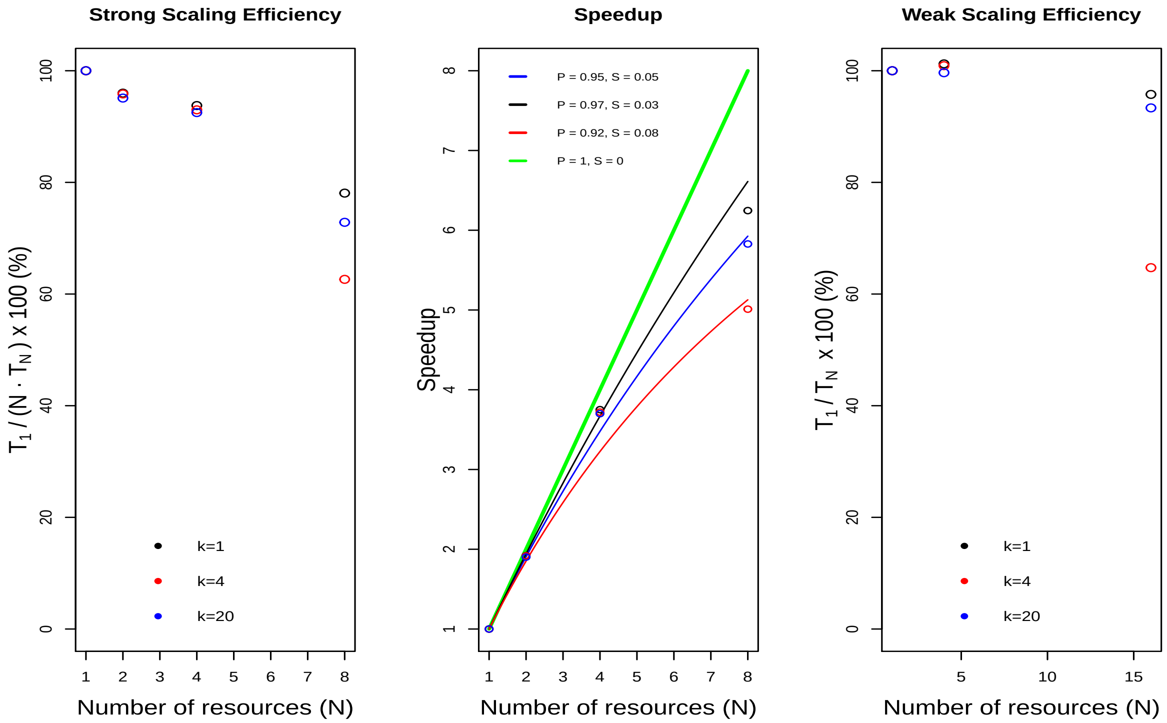

Figure 4: Strong and weak scaling efficiencies and speedup for $n = 4096$, `iter_max` $= 1000$.

Both the two scaling efficiencies and the speedup for the cases when $N \leq 4$, i.e., when we are making use of a single computer, the values of these properties are very close to the ideal values: the strong scaling efficiency has almost not decreased and stays above the 90%; the speedup is almost equal to the identity function ($f(N) = N$); and the weak scaling efficiency is constant over $N$.

However, for the case $N = 8$ (and $N = 16$ for the weak scaling efficiency case) we are using two different computers (4 for the case of the weak scaling efficiency) and we see that the values obtained are far from the theoretical. We have to assume that the communication time between processes when we are using more nodes increases. Therefore, as we expected, working with cores or with independent nodes will deeply change our performance results.

There is something that remains not well-understood and it is the influence of the parameter $k$ on the performance metrics. Our initial expectations were that increasing the value of $k$, the communication overhead will be reduced considerably so that the performance metrics would improve for larger $k$. However, with the three values of $k$ we have considered, we can not see a clear tendency so a further study taking into account more values for $k$ should have to be done in the future.

Perhaps the communication overhead is large due to the speed of the cluster network and the improvement that the Block communication ($k$) optimization would supply is not significant in this case.

## 3.2 Analysing profiles and traces with TAU and Jumpshot

Now we will proceed to analyse the profiles and the traces of our MPI applications, the optimized and the not-optimized. We will also compare both applications and study if the optimized version is in fact, and optimization.

### 3.2.1 Not Optimized Application

Studying the profiles an execution we can check how much time is invested in the communication between cores, and how much time is invested in computing.

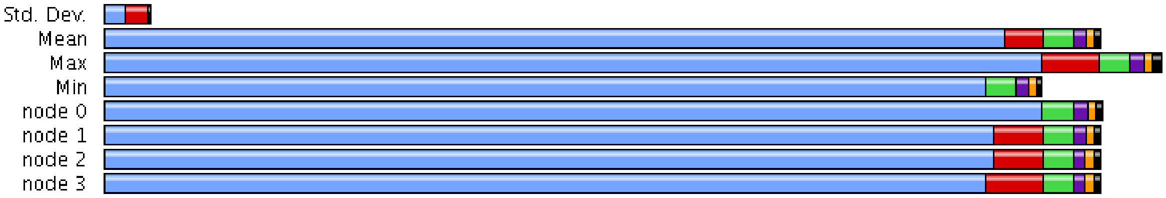

Figure 5: Graphical visualization of a profile using the tool paraprof of an execution for $n = 2048$, `iter_max` $= 100$.

We can see that most of the time is invested in computing (blue bar), and a smaller part of the time is invested in the communication (red bar, corresponding to the send function and black bar, corresponding to recieve). The other bars correspond to the MPI initialization and finalization and the initialization part of the code. Puting numbers, about 8.4 seconds are invested in computing and 0.4 seconds are invested in communication.

Analysing the traces we can study in a more detailed way the execution of our code.

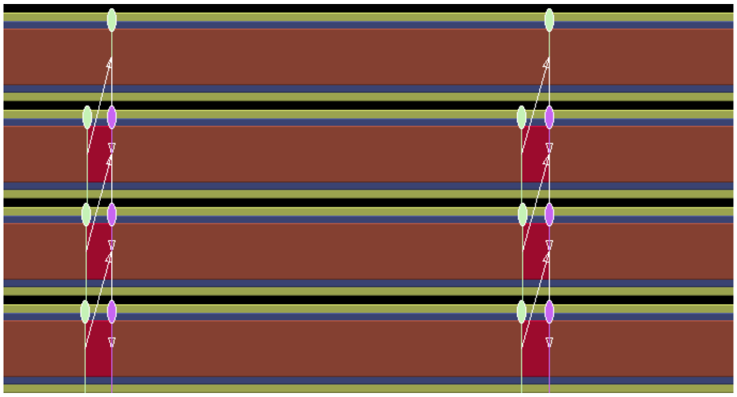

Figure 6: Graphical visualization of a trace using jumpshot. We can see an interation of our code, were the computation is done (brown bars) and then the correspoding communication takes place (red bars).

In figure 6 we can also observe how small is the amount of time invested in communication compared with the time invested in computing. Furthermore, the time invested in sending the messages is much larger than the time invested in receiving them, as we can see in figure 7.

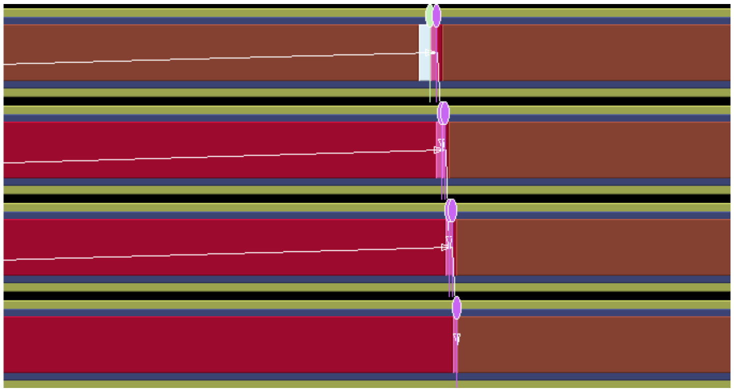

Figure 7: Graphical visualization of a trace using jumpshot. We can see the communication part of an interaction (left side). The red bars correspond to the time spent sending messages, and the pink bars, much smaller, correspond to the time spent in receiving them, in terms of the functions `MPI_Send` and `MPI_Recv`.

However this study have been made using a single node with 4 cores, therefore, we can expect that the communication time is smaller that if we used 4 different nodes. Therefore, since the communication time is that small, we can expect that using a single node, our optimization will not be perceptible.

### 3.2.2 Optimized application

Analysing the profiles of an execution of the same characteristics and $k = 4$ we check that there is almost no difference in the execution time between both applications.

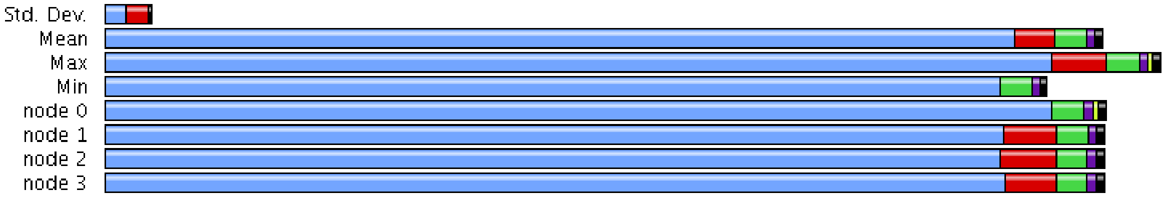

Figure 8: Graphical visualization of a profile using the tool paraprof of an execution for $n = 2048$, `iter_max` $= 100$ and $k = 4$

Comparing figures 5 and 8, we check that there is almost no difference. In fact, in the optimized case, the computation time is 8.42 seconds and the communication time is 0.35 seconds. However a bit more time is spend in the initializations. In the end, there is no appreciable difference in the performance of both applications, and the differences could be due to the statistical fluctuations of the initialization parts and so.

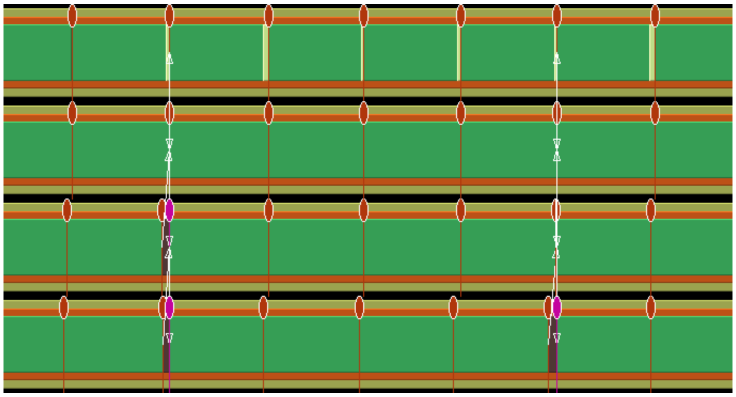

Figure 9: Graphical visualization of a trace using jumpshot. We can see now that that there is only communication every 4 iterations, since $k = 4$. In this case green is calculing.

However, analysing the traces of our optimized application, we see that the algorithm working is different, and it is doing what it is expected: It only communicates every 4 iterations. Note that between communications there are exactly four red bars which correspond to the call and performance of the `MPI_Reduce` operation after each iteration

### 3.2.3 Comparison between both applications

We have analysed the performance of a concrete execution done for both applications. However, these examples are not conclusive tests of how much our optimized version optimizes our application. In the studied case, we saw what we expected: The communication is reduced, increasing the computations time.

This is the expected since we are communicating less frequently paying a price: Each process computes extra rows (so the computation time must increase) and the messages, despite being less frequent become bigger (so the communication time does not decrease too much).

Then, the question is: Is our application faster with the optimization? We think that it depends on the situation. When we work with a single node and multiple cores, the communication time is about the 5% of the total time of the execution, so we can not reduce it too much.

However, when using several nodes, the communication time should increase, while the time invested in calculating should be the same. Therefore, since the proportion of time invested in communication will be larger than for a single node, the performance of the optimized version should be much better than the not optimized version. The point is that in this case the percentage of time corresponding to communication will be larger and thus the reduction of time achieved by the optimized version will be also larger.

We would have like to do this comparison between two executions done in several nodes, but during all the Christmas holidays we only had been able to do 7 executions using more than one node, submitting jobs using slurm. We could not been able to do a TAU analysis of executions of our applications with slurm, since we did not had enough resources (we have been trying for more than 10 days).

# 4 Conclusions

We wrote our version of the code that solves the Laplace problem using MPI, and we checked that worked properly, getting the same results that we obtained with the sequential version. We checked that the error obtained and the final matrix were the same in the MPI version than in the sequential version.

Then we tried to optimized it, reducing the communication time. We chose the Block communication because we thought that it was the simplest to implement. We also checked that the error and the final matrix were almost the same that in the sequential version for a lot of values of $k$ [2]. Later, analysing the traces we also checked that the code was doing what we expected: only communications every $k$ iterations.

Then we studied the strong and weak scalability efficiencies and the speedup. For the not optimized MPI version using up to 4 cores of the same node, taking into account that the cores of the lab can do 2 processes each. We found that our code's speedup and weak and strong scaling efficiencies almost matched with the theoretical values (above 95%) for 4 processes or less, and it dropped to about 60% for 8 processes. We think that it dropped for 8 processes due to the fact that the computer has 4 real cores, and when they do 2 processes each they reduce their performance as we saw for the OpenMP assignment. We also found that the strong scaling efficiency and the speedup were bigger for smaller sizes of the problem.

We also did a more superficial study using multiple nodes and the optimized version and we obtained similar results. When we used multiple nodes we found a drop of the efficiencies and the speedup. We think that it dropped due to an increase of the communication times when using two or more different computers. We could not do a deeper study of the dependence of the computation time depending on the $k$.

Comparing both versions, the optimized and the not optimized code, we found that the communication time using only one node was about the 5% of the execution time. Therefore, our optimization, that it only reduces the communication time, could not improve too much the performance. In fact that is what we found: the application was optimized, but only about a 1% in general.

We expect that using several nodes, the communication time should increase, while the computational time should remain the same (if all the cores have the same properties). Hence, using multiple nodes our optimized version should be significantly or much faster than the not optimized version.

However, we could not do this study since jobs submitted by slurm were not working properly. We tried to do a TAU analysis for executions done in multiple nodes, but since the Christmas holidays started, the cluster did not do almost none of the jobs submitted.

[2] We have observed that as the number of $k$ increases, there are little deviations of the total final error and of some components of the matrix, probably due to propagation of errors and due to the fact that each process is computing more rows each iteration.

# References

[1] OpenMPI Slides, UAB.

[2] MPI tutorial and general information https://computing.llnl.gov/tutorials/mpi/.

[3] Theoretical video-tutorials of MPI, High Performance Computing by Prof. Matthew Jacob, Department of Computer Science and Automation, IISC Bangalore. For more details on NPTEL visit http://nptel.iitm.ac.in. The video-tutorials can be found in YouTube: https://youtu.be/mzfVimVbguQ, https://youtu.be/mb5wV4AqXso.

[4] Amdahl's law (definition of speedup). (2017, July 16). In Wikipedia, The Free Encyclopedia. Retrieved 22:19, November 25, 2017, from https://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=790799480.

[5] Speedup Ratio and Parallel Efficiency, http://www.bu.edu/tech/support/research/training-consulting/online-tutorials/matlab-pct/scalability/.

[6] Measuring parallel scaling performance, https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance.

[7] GitHub repository https://github.com/dsalgador/Laplace2D_MPI.

# A Code

## A.1 Basic MPI implementation code

```
// Libraries
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

//Definitions
#define MASTER 0        /* task ID of master task */

float stencil ( float v1, float v2, float v3, float v4)
{
  return (v1 + v2 + v3 + v4) * 0.25f;
}

float max_error ( float prev_error, float old, float new )
{
  float t= fabsf( new - old );
  return t>prev_error? t: prev_error;
}

/*Initialisation of the grid: internal points set to 0
  and boundary conditions initialised according to the PDF of
  this assignemnt*/
void laplace_init(float *in, int n)
{
  int i;
  const float pi  = 2.0f * asinf(1.0f);
  memset(in, 0, n*n*sizeof(float));
  for (i=0; i<n; i++) {
    float V = in[i*n] = sinf(pi*i / (n-1));
    in[ i*n+n-1 ] = V*expf(-pi);
  }
}


/*Given a matrix * in with nrows rows and ncols columns
  prints it on the console in a representative way*/
void print_matrix(float * in, int nrows, int ncols){
    int i,j;
    for ( j=0; j < nrows; j++ ){
      for ( i=0; i < ncols; i++ )
    {
      printf(”%0.3f|”, in[j*ncols+i]);
    }
    printf(”\n”);
    }
    printf(”\n”);
}

/*
 This is quite the same as the laplace_step function from the original code
 lapFusion.c. Now my_laplace_step is a function called by each Process to update
 its part of the matrix A (stored in ’my_A’). The part of matrix for each process,
 my_A, has ’nrows’ rows and ’ncols’ columns. We add also the two parameters
 ’rowstart’ and ’rowend’ that allow us to decide from which row to wich row my_A is updated
 by the Process that is calling my_laplace_step() function.
 */
float my_laplace_step(float *in, float *out, int nrows, int ncols, int rowstart, int rowend)
{
  int i, j;
  float my_error=0.0f;
  for ( j=rowstart; j < rowend; j++ )
    #pragma omp simd reduction(max:my_error)
    for ( i=1; i < ncols-1; i++ )
    {
      out[j*ncols+i]= stencil(in[j*ncols+i+1], in[j*ncols+i-1], in[(j-1)*ncols+i], in[(j+1)*ncols+i]);
      my_error = max_error( my_error, out[j*ncols+i], in[j*ncols+i] );
    }
  return my_error;
}

void copy_A_to_temp(float *A, float * *temp, int nrows, int ncols){
    for(int j = 0; j < nrows; j++){
      for(int i = 0; i < ncols; i++){
        (*temp)[j*ncols +i] = A[j*ncols +i];
      }
    }
}

/*
Commands to run the code:
module load gcc/6.1.0
module load mpe2/mpi-1.10.2/2.4.8
mpicc -g -lm -fopenmp -o mpi_lapFusion lapFusion_mpi.c
```

```

mpirun -np N mpi_lapFusion n iter_max
*/

int main(int argc, char** argv)
{
  // Initalisation of variables
  double t0, tf; /*Initial and final time counters*/

  int n = 4096; /* Size of the grid n x n */
  int iter_max = 1000; /* Number of iterations */
  float *A, *temp; /* Pointers to grid A and temp */

  const float tol = 1.0e-5f; /* Tolerance */
  float error= 1.0f; /* Global error variable */

  int numtasks, rank, tag = 1,rc; /* # of processes, process ID, tag, rc */
  int  my_nrows, my_size; /* Number of rows of my_A, dimension of my_A*/
  float *my_A, *my_temp; /* Portion of A carried by each process*/
  float my_error= 1.0f; /* Error for each process*/
  MPI_Status Stat; /* MPI status variable to control the status*/

  int rowstart, rowend, nrows; /*Auxiliar variables related to rows*/

  //INIT MPI environment
  rc = MPI_Init (&argc, &argv);
  if (rc != MPI_SUCCESS)
    {
      printf ("Error␣starting␣MPI␣program.␣Terminating.\n");
      MPI_Abort (MPI_COMM_WORLD, rc);
      return -1;
    }
  MPI_Comm_size (MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  //END basic INIT MPI environment


  // Abort the program if the number of processes is less than 2
  if(numtasks < 2){
    printf ("This␣program␣works␣with␣2␣or␣more␣processes␣(-np␣N␣with␣N␣>=2).\n");
    MPI_Abort (MPI_COMM_WORLD, 1);
    return -1;
  }


  //BEGIN MASTER Initialisation of A, temp and initial time
  if(rank == MASTER){
  t0 = MPI_Wtime(); //Record the initial time

  // get runtime arguments
  if (argc>1) {  n        = atoi(argv[1]); }
  if (argc>2) {  iter_max = atoi(argv[2]); }

  // Allocate memory for A and temp
  if( ( A = (float*) malloc(n*n*sizeof(float)) ) == NULL ){
    printf (”Error␣when␣allocating␣memory␣for␣A.\n”);
    MPI_Abort (MPI_COMM_WORLD, 1);
    return -1;
  }
  if( ( temp = (float*) malloc(n*n*sizeof(float)) ) == NULL ){
    printf (”Error␣when␣allocating␣memory␣for␣temp.\n”);
    MPI_Abort (MPI_COMM_WORLD, 1);
    return -1;
  }

  // set boundary conditions
  laplace_init (A, n);
  laplace_init (temp, n);
  A[(n/128)*n+n/128] = 1.0f; // set singular point

  printf(”Jacobi␣relaxation␣Calculation:␣%d␣x␣%d␣mesh,”
         ”␣maximum␣of␣%d␣iterations\n”,
         n, n, iter_max );
  } //END MASTER initialisation

  //All processes initialise iter to 0
  int iter = 0;

  //Broadcast de global (MASTER) error, n and iter_max
  MPI_Bcast(&error, 1, MPI_FLOAT, MASTER, MPI_COMM_WORLD);
  MPI_Bcast(&n, 1, MPI_INT, MASTER, MPI_COMM_WORLD);
  MPI_Bcast(&iter_max, 1, MPI_INT, MASTER, MPI_COMM_WORLD);

  //Initialise some auxiliar variables
  my_nrows = n/numtasks;
  nrows = my_nrows +2;
  my_size = n*(my_nrows+2);

  //Allocate memory for my_A and my_temp
  if( ( my_A = (float*) malloc( my_size*sizeof(float)) ) == NULL ){
    printf (”Error␣when␣allocating␣memory␣for␣my_A.\n”);
    MPI_Abort (MPI_COMM_WORLD, 1);
    return -1;
  }
  if( ( my_temp = (float*) malloc(my_size*sizeof(float)) ) == NULL ){
    printf (”Error␣when␣allocating␣memory␣for␣my_temp.\n”);
    MPI_Abort (MPI_COMM_WORLD, 1);
    return -1;
  }

  //Distribute the rows of A and temp among all the processes --> store in my_A, my_temp
  MPI_Scatter(A, my_nrows*n,  MPI_FLOAT, my_A+n, my_nrows*n, MPI_FLOAT, MASTER, MPI_COMM_WORLD);
  float * my_temp_plusn = my_temp+n;
  copy_A_to_temp(my_A+n, &my_temp_plusn, my_nrows, n);
  //MPI_Scatter(temp, my_nrows*n,  MPI_FLOAT, my_temp+n, my_nrows*n, MPI_FLOAT, MASTER, MPI_COMM_WORLD);


 while ( error > tol*tol && iter < iter_max )
  {
    iter++;
    /*Send and Recv calls so that each process obtain two additional rows needed for
    the computation of the new values.
    */

    if(rank > MASTER){
      //For all the processes apart from MASTER, which does not need a previous row

      /* Send the first row of the process ’rank’ to the process ’rank-1’*/
      MPI_Send(my_A+n, n, MPI_FLOAT, rank-1, tag ,MPI_COMM_WORLD);
      /* Process ’rank’ recieves the last row from the process ’rank-1’*/
      MPI_Recv(my_A  , n, MPI_FLOAT, rank-1, tag ,MPI_COMM_WORLD, &Stat);
    }
    if(rank < numtasks -1 ){
       //For all the processes apart from THE LAST, which does not need a ’last’ row

       /* Process ’rank’ recieves the first row from the process ’rank+1’*/
       MPI_Recv( (my_A + n*(my_nrows+1) )  , n, MPI_FLOAT, rank+1, tag ,MPI_COMM_WORLD, &Stat);
       /* Send the last row of the process ’rank’ to the process ’rank+1’*/
       MPI_Send(  (my_A + n*(my_nrows))  , n, MPI_FLOAT, rank+1, tag ,MPI_COMM_WORLD);
    }

```

```
    /*Set values for rowstart and rowend in order to make all processes modify only the internal
    points of A. We have to distinguish between the process that has the first block of rows of A
    (the MASTER) and the one that has the last block of rows (the process numtasks-1).
    */
    if(rank == MASTER){
      rowstart =2;
      rowend = nrows-1;
    }
    else if(rank == (numtasks - 1)){
      rowstart = 1;
      rowend = nrows -2;
    }
    else{
      rowstart = 1;
      rowend = nrows-1;
    }
    //Each process performs the laplace_step updating the points of my_A that are interior points of A
    my_error= my_laplace_step(my_A, my_temp, nrows, n, rowstart, rowend);

    /*Reduction operation: the maximum among all my_error from all processes is calculated and stored
    in the variable error, which is the global error and originally stored in the MASTER process*/
    MPI_Reduce(&my_error, &error, 1, MPI_FLOAT, MPI_MAX, MASTER, MPI_COMM_WORLD);
    //Swap the roles of my_A and my_temp (double buffer) to be prepared for the next iteration.
    float *swap= my_A; my_A=my_temp; my_temp= swap;
  }
  /*The master process gathers all the final portions of A stored in my_A of each process to build
   the matrix A corresponding to the last iteration.
  */
  MPI_Gather(my_A+n, my_nrows*n ,  MPI_FLOAT, A, my_nrows*n, MPI_FLOAT, MASTER, MPI_COMM_WORLD);

  /*The MASTER process computes the final error as the sqrt of the variable error
    and some information is printed onto the screen*/
  if(rank == MASTER){
    error = sqrtf( error );
    printf("Total␣Iterations:␣%5d,␣ERROR:␣%0.6f,␣", iter, error);
    printf("A[%d][%d]=␣%0.6f\n", n/128, n/128, A[(n/128)*n+n/128]);
    free(A); free(temp);
   }
   /*The master process prints the execution time*/
   if(rank == MASTER){
    tf = MPI_Wtime();
    printf(”Elapsed␣time,␣%2.5lf\n”, tf-t0);
   }
   //Finalize the MPI environment.
   MPI_Finalize();
   return 0;
}
```

## A.2 Optimized MPI implementation code

```
/*
 .
 .
 .
 The same as the Base MPI implementation code
 .
 .
 .
*/

 //Broadcast de global (MASTER) error, n, iter_max and k
  MPI_Bcast(&error, 1, MPI_FLOAT, MASTER, MPI_COMM_WORLD);
  MPI_Bcast(&n, 1, MPI_INT, MASTER, MPI_COMM_WORLD);
  MPI_Bcast(&iter_max, 1, MPI_INT, MASTER, MPI_COMM_WORLD);
  MPI_Bcast(&k, 1, MPI_INT, MASTER, MPI_COMM_WORLD);

  //Initialise some auxiliar variables
  my_nrows = n/numtasks;
  nrows = my_nrows +2*k;
  my_size = n*nrows;

  //Allocate memory for my_A and my_temp
  if( ( my_A = (float*) malloc( my_size*sizeof(float)) ) == NULL ){
    printf (”Error␣when␣allocating␣memory␣for␣my_A.\n”);
    MPI_Abort (MPI_COMM_WORLD, 1);
    return -1;
  }
  if( ( my_temp = (float*) malloc(my_size*sizeof(float)) ) == NULL ){
    printf (”Error␣when␣allocating␣memory␣for␣my_temp.\n”);
    MPI_Abort (MPI_COMM_WORLD, 1);
    return -1;
  }

  int starting = n*k;
  //int ending = n*k;
  //Distribute the rows of A and temp among all the processes --> store in my_A, my_temp
  MPI_Scatter(A, my_nrows*n,  MPI_FLOAT, my_A+starting, my_nrows*n, MPI_FLOAT, MASTER, MPI_COMM_WORLD);
  float * my_temp_plusnk = my_temp+starting;
  copy_A_to_temp(my_A+starting, &my_temp_plusnk, my_nrows, n);
  //MPI_Scatter(temp, my_nrows*n,  MPI_FLOAT, my_temp+starting, my_nrows*n, MPI_FLOAT, MASTER, MPI_COMM_WORLD);


 while ( error > tol*tol && iter < iter_max )
  {
    iter++;
    /*Send and Recv calls so that each process obtain two additional rows needed for
    the computation of the new values.
    */

    if( ( (iter-1) % k) == 0 )
    {
      if(rank > MASTER){
        //For all the processes apart from MASTER, which does not need a previous row

        /* Send the first k rows of the process 'rank' to the process 'rank-1'*/
        MPI_Send(my_A+n*k, n*k, MPI_FLOAT, rank-1, tag ,MPI_COMM_WORLD);
        /* Process 'rank' recieves the last k rows from the process 'rank-1'*/
        MPI_Recv(my_A  , n*k, MPI_FLOAT, rank-1, tag ,MPI_COMM_WORLD, &Stat);
      }
      if(rank < numtasks -1 ){
         //For all the processes apart from THE LAST, which does not need a 'last' row

         /* Process 'rank' recieves the first k rows from the process 'rank+1'*/
         MPI_Recv( (my_A + n*(nrows-k) )  , n*k, MPI_FLOAT, rank+1, tag ,MPI_COMM_WORLD, &Stat);
         /* Send the last k rows of the process 'rank' to the process 'rank+1'*/
         MPI_Send(  (my_A + n*(my_nrows))  , n*k, MPI_FLOAT, rank+1, tag ,MPI_COMM_WORLD);
      }
```

```

    } //endif iter % k

    /*Set values for rowstart and rowend in order to make all processes modify only the internal
    points of A. We have to distinguish between the process that has the first block of rows of A
    (the MASTER) and the one that has the last block of rows (the process numtasks-1).
    */
    if(rank == MASTER){
      rowstart = 1+k;//2     1+k
      rowend = nrows-1;//nrows -1        nrows-k
    }
    else if(rank == (numtasks - 1)){
      rowstart = 1; //1
      rowend = nrows - (1+k); // nrows -2     nrows - (1+k)
    }
    else{
      rowstart = 1; //1      k
      rowend = nrows-1; //nrows-1       nrows-k
    }
    //Each process performs the laplace_step updating the points of my_A that are interior points of A
    my_error= my_laplace_step(my_A, my_temp, nrows, n, rowstart, rowend);

    /*Reduction operation: the maximum among all my_error from all processes is calculated and stored
    in the variable error, which is the global error and originally stored in the MASTER process*/

    /*if( ( (iter-1) % k) == 0 )
    {  */
    MPI_Reduce(&my_error, &error, 1, MPI_FLOAT, MPI_MAX, MASTER, MPI_COMM_WORLD);
    // }

    //Swap the roles of my_A and my_temp (double buffer) to be prepared for the next iteration.
    float *swap= my_A; my_A=my_temp; my_temp= swap;
  }
  /*The master process gathers all the final portions of A stored in my_A of each process to build
   the matrix A corresponding to the last iteration.
  */
  //MPI_Reduce(&my_error, &error, 1, MPI_FLOAT, MPI_MAX, MASTER, MPI_COMM_WORLD);

  MPI_Gather(my_A+starting, my_nrows*n ,  MPI_FLOAT, A, my_nrows*n,
     MPI_FLOAT, MASTER, MPI_COMM_WORLD);

  /*The MASTER process computes the final error as the sqrt of the
     variable error
    and some information is printed onto the screen*/
  if(rank == MASTER){
    error = sqrtf( error );
    printf("Total␣Iterations:␣%5d,␣ERROR:␣%0.6f,␣", iter, error);
    printf("A[%d][%d]=␣%0.6f\n", n/128, n/128, A[(n/128)*n+n/128]);
    //print_matrix(A, n, n);
    free(A); free(temp);
   }
   /*The master process prints the execution time*/
   if(rank == MASTER){
    tf = MPI_Wtime();
    printf("Elapsed␣time,%2.5lf\n", tf-t0);
   }
   //Finalize the MPI environment.
   MPI_Finalize();
   return 0;
}
```