Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing **2022**

Student: Fabio Marchesi Discussed with: Gioele De Pianto, Raffaele Morganti

Solution for Project 5

Due date: 30.11.2022, 23:59

This project will introduce you a parallel space solution of a nonlinear PDE using MPI.

1. Task 1 - Initialize and finalize MPI [5 Points]

Firstly, we need to initialize MPI, in order to do that is enough to use the standard **MPI_Init**. In my first solution, I used **MPI_Init_thread**, because we are combining the use of processes (MPI) and threads (OpenMP), but it wasn't the right decision because that type of initialization needs to be used when different threads do MPI calls, but it isn't the case for this project. And after some tests I also discovered that using **MPI_Init_thread** and certain set of parameters makes the execution crash and, if the execution was schedule on the cluster with **sbatch**, then the cluster tries to execute it again and again, but in the end this resulted in the crash of the cluster's node that was doing the execution.We can just use **MPI_Comm_rank** and **MPI_Comm_size** to obtain the rank of the current process and the total number of processes. Before the end of the execution, we just need to call **MPI_Finalize** to shut down the MPI library.

```
MPI_Init(&argc, &argv);
int mpi_rank, mpi_size;
MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

..do_work..

MPI_Finalize();
```

2. Task 2 - Create a Cartesian topology [10 Points]

Now we want to create a Cartesian non-periodic topology with our processes. Firstly, we need to understand how to divide the domain into a grid, so we need to compute the number of processes for each row and column. In order to do that we can use **MPI_Dims_create** which, given the number of processes and the number of dimensions required, calculate the grid sizes. In order to create the grid of domains, we used **MPI_Cart_create**, this function has a parameter `const int* periods` that can be used to specify that our grid shouldn't be periodic. After that, using the rank of the current process, we can use **MPI_Cart_coords** to retrieve its coordinates in the grid. To find the neighbors of the current process, we can use **MPI_Cart_shift**. This function requires a direction (in this case we have a 2-dimensional grid, so we have two directions) and a stride (we want to know the immediate neighbors, so the stride will be equal to one).

```
// Computing the dimensions of the grid
int dims[2] = {0, 0};     // No limitation on the number of rows/cols
MPI_Dims_create(mpi_size, 2, dims);
ndomy = dims[0];
ndomx = dims[1];

// Creating the grid
MPI_Comm comm_cart;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm_cart);

// Retrieve coordinates of the process in the cartesian grid
int coords[2];
ierr = MPI_Cart_coords(comm_cart, mpi_rank, 2, coords);
domy = coords[0]+1;
domx = coords[1]+1;

// Set neighbours for all directions
MPI_Cart_shift(comm_cart, 0, 1, &neighbour_south, &neighbour_north);
MPI_Cart_shift(comm_cart, 1, 1, &neighbour_west, &neighbour_east);
```

3. Task 3 - Change linear algebra functions [5 Points]

The functions `ss_dot` and `ss_norm2` are computing, respectively, the dot product of two vectors and the norm of a vector. Here, each function is computed by all the processes, and we need somehow to merge the result of these processes in a way that, in the end, each process will have the merged result. In order to do that, we use **MPI_Allreduce**. This MPI function allows each process to perform a reduction calculation (in this case a sum) and let the result be available to all MPI processes involved (it's a combination of a reduction and a broadcast). We need to use this operation only in these two linear algebra functions because all the other are void functions, so each "void" function will change some values, but there isn't the need to share those modifications. Instead, **ss_norm** and **ss_dot** computes values that need to be known by all the processes (for example the norm is used to check the convergence of the simulation), for this reason we need to share their results. When we use **MPI_Allreduce** we need to specify: a pointer to the send buffer, a pointer to the receive buffer, the number of elements to share, the type of those elements, the reduction operation and the communicator in which the reduction takes place.

4. Task 4 - Exchange ghost cells [45 Points]

Now we need to exchange ghost cells between the neighbors, so every process has all the data it needs for the stencil computation. In order to do that, each process will receive data from its neighbors and send its own data to them. So for each neighbor, we can do something like this:

- Start to receive the boundary points from him. But we want to do this in a non-blocking way, so, we use **MPI_Irecv**. This function let us receive the message without blocking the computation. Later, we need to have a function that waits for the end of this operation.

- Copy the boundary points that we need to send him into a buffer, so we don't need to define a costum type to send them.

- Send the newly created buffer to the neighbor. Also, this time we want to use a non-blocking call, this time we use the counterpart of **MPI_Irecv** that is **MPI_Isend**.

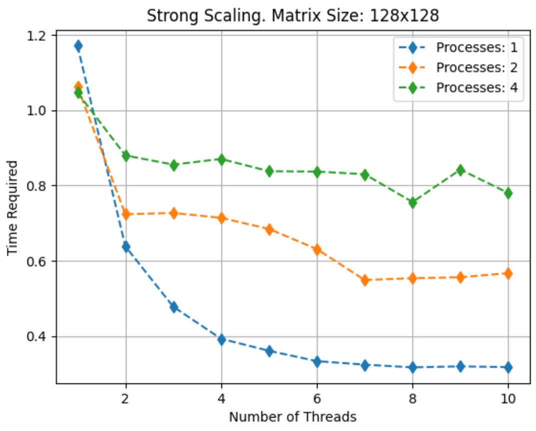
After this, we want to overlap computation and communication, so, after the start of send/receive operations, we start to compute the inner grid points of the current process because we don't need any information from the neighbors to do that. Now we need to compute the outer points of the grid, so we wait for the results from the neighbors to the east, west, north, south. We don't wait for all of them simultaneously, when the east border arrives we use it, after that when the west border arrives we use it and so on. Only now that the current process has completed the stencil computation, we wait for the send operations to finish. If we didn't do it, we could encounter problems generated by the fact that different processes could be at different iterations.

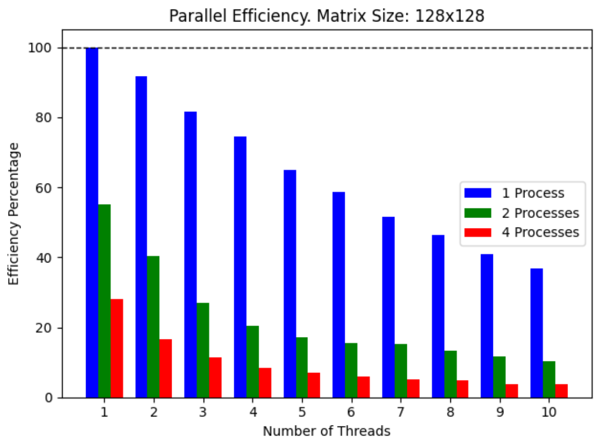
For each process, we have at most 4 neighbors (the reported tests use at most 4 processes, in this case each process has at most two neighbors), so before sending/receiving data in some direction we need to make sure that the neighbor exists. In the program, this is done with checking the condition $domain.neighbour\_X \geq 0$. And obviously each process waits only the data from its existing neighbors, otherwise the corresponding wait function wouldn't terminate.

5. Task 5 - Testing [20 Points]

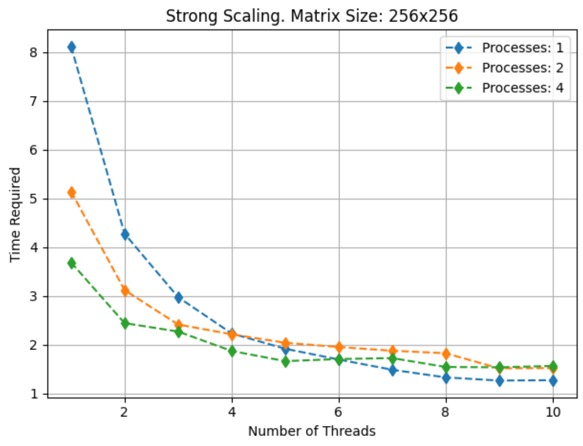
In order to test the performance of the application and see how well it scales, I computed its execution time for different combinations of parameters. In particular, we put N (the matrix side) equal to $128, 256, 512, 1024$, p (the number of processes used by MPI) to $1, 2, 4$ and t (the number of threads) equal to $1, 2, \ldots, 10$. For each combination of parameters, we computed the execution time and the parallel efficiency. The parallel efficiency can be used to estimate how much of the computing capacity is actually used to carry out a calculation and is computed as $\frac{Speed\_up}{Number\_of\_cores} \times 100$, where in this case the number of cores is computed a $p \cdot t$

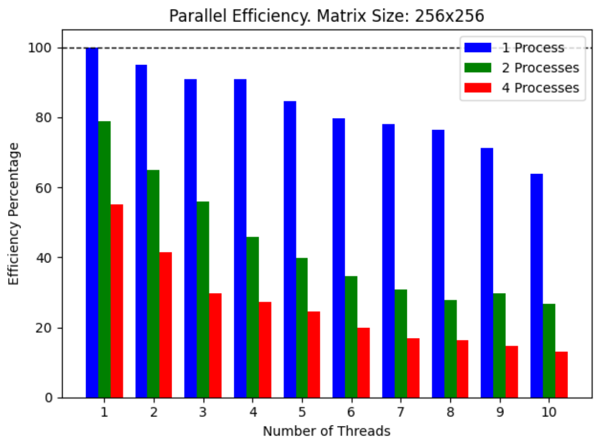
- $N = 128$: The matrix here is tiny, so we don't actually exploit the potential multiprocessing. Indeed, we see how using only one process we obtain a better execution time than using two or four of them. The reason behind this counterintuitive result is that the communication overhead of having multiple processes on different nodes is higher than the advantage of splitting the computation. At the same time, we can see how increasing the number of threads (especially with only one process) actually improves the execution time. The fact that the application doesn't scale well with the number of processes on this matrix can also be seen looking at the parallel efficiency plot. Indeed, we can see how the green and red bars are far away from 100% for each possible number of threads.



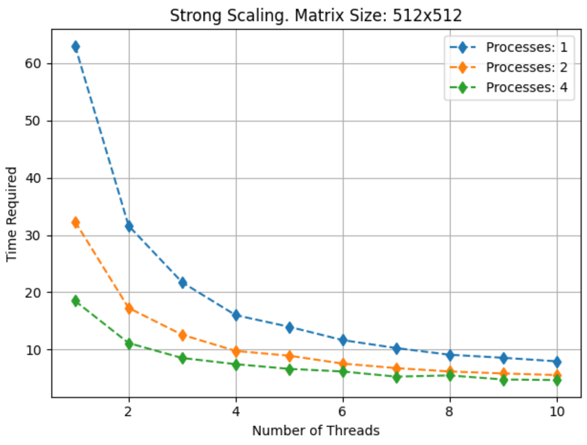


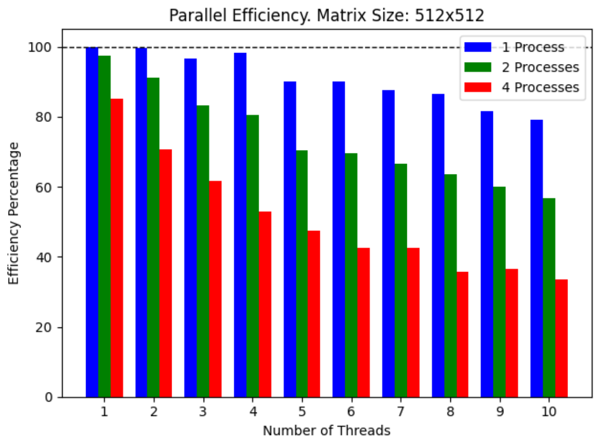
- $N = 256$: We can start to see the improvement given by the multiprocessing. Indeed, we can see how the green line is under the other two for each $t < 6$, after that, when we start to have more threads the execution with one process is, again, the fastest one. Also, the parallel efficiency graph shows a better use of parallel computation than before, indeed the bars are definitely higher of each possible number of threads.



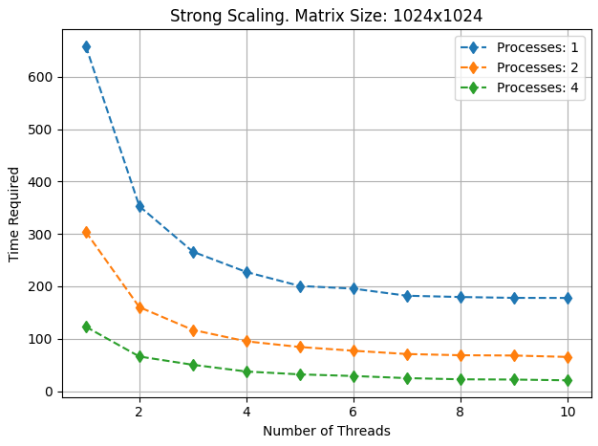


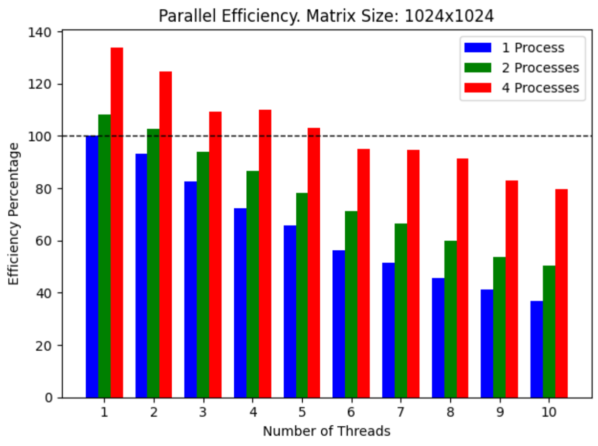
- $N = 512$: The matrix here starts to become big, indeed, in this case we have 16 times more grid cells than the first test. And here, for the first time, we have that the execution times with 4 processes are faster than the ones with 2 processes, which are faster than the ones with only one process for each number of threads. The multithreading seems to work as well, indeed all the three lines are (almost) always decreasing. The parallel efficiency is good as well, indeed we can see how the bars corresponding to the four processes executions are quite high and definitely higher than the ones with had with $N = 128$.





- $N = 1024$: The matrix here is massive. And the improvements of multiprocessing are even easier to see. The application scales with the number of threads and with the number of processes. An interesting fact that I notice looking at the parallel efficiency plot is for some combination of t and p we have a parallel efficiency higher than 100%. Indeed, with $p = 1$ and $t = 1$ the execution time is $658s$ while with $p = 4$ and $t = 1$ the execution time is $122s$. This means that increasing the computation power by a factor of 4 we get results that are better by a factor that is actually higher than 4, getting a parallel efficiency higher than 130%. The first thing that I checked was if the number of total iterations remained the same between the two tests. It turned out that the number of newton iterations and the number of conjugate gradient iterations is basically the same. What is changed is the rate of conjugate gradient iterations. In the first case (with $p = 1, t = 1$) we have 78.5183 iters/second while in the second case (with $p = 4$, $t = 1$) we have a rate of rate of 414.441 iters/second (also here the difference factor is greater than 4). The best idea that I had that justifies this difference is that a big single matrix exploits the cache terribly, while dividing the matrix in four smaller sub-matrices allows them to use better the cache of the four different nodes of the cluster that are storing them. This would mean that the number of iterations and accesses to the matrix remains the same, but in the second case the accesses have a faster average time.





To obtain these results, I used the flag **–cpus-per-proc** which bind each process to the specified number of cpus.

6. Task 6 - Quality of the Report [15 Points]

Additional notes and submission details

Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.) and summarize your results and the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to iCorsi.

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:
  - all the source codes of your MPI solutions.
  - your write-up with your name project_number_lastname_firstname.pdf,
- Submit your .tgz through Icorsi.