Università della Svizzera italiana

Institute of Computing
CI

High-Performance Computing Lab **Institute of Computing**

Student: Alessio Barletta

Solution for Project 5

HPC Lab — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

1. Parallel Space Solution of a nonlinear PDE using MPI [in total 60 points]

In this task we are asked to parallelize the solution of a non linear PDE already discussed in project 3, but this time using MPI instead of OpenMP.

1.1. Initialize/finalize MPI and welcome message [5 Points]

Firstly, I added a welcome message that prints the number of processes used. The message is as follows:

```
================================================================================
                        Welcome to mini-stencil!
version   :: C++ MPI
processes :: 8
mesh      :: 1024 * 1024 dx = 0.000977517
.....
```

1.2. Domain decomposition [10 Points]

To implement the domain decomposition, we mainly had to work on the file `data.cpp`. We already had the struct `Discretization` implemented, that contained all the global information about the simulation, such as grid dimension or the number of time steps.

We had to implement the `init` method of the `SubDomain` struct, that is responsible for the local domain decomposition.

I started by initializing the number of sub-domains in the x and y dimensions, to do this, i declared `dims[2] = 0, 0`, and then I called the function `MPI_Dims_create` passing the total number of processes and the `dims` array. Having both dims set to 0, MPI will try to balance the number of processes in both dimensions as much as possible.

This function tries as default to make the subdomain as cubic as possible (square in 2D), allowing us to minimize the perimeter of each subdomain, and thus the amount of data that needs to be exchanged between processes, lowering communication overhead.

Moreover, looking at the provided code, we can see that the data is used as if the (0,0) coordinate is in the bottom left corner of the domain. For this reason, when doing `MPI_cart_shift`, I inverted the north and south directions compared to standard matrix indexing (where (0,0) is at the top left), so that the north direction corresponds to an increase in the y coordinate.

1.3. Linear algebra kernels [5 Points]

The functions that I parallelized are the following:

- `hpc_norm2`
- `hpc_dot`

In both of this function I had to compute a local result and then add the allreduce operation at the end of the computation, to gather the results from all processes and sum them together. This is needed since both functions compute a global result, that depends on the entire domain.

All the other linear algebra kernels were already parallel, since they only involved local operations on each subdomain, so there was no need to modify them.

1.4. The diffusion stencil: Ghost cells exchange [10 Points]

To implement the ghost cell exchange, we had to modify the file `operators.cpp`, The way I implemented it was by essentially following the logic proposed in the hint, so I did four non-blocking receives using `MPI_Irecv` for each direction (north, south, east, west), then I placed my border coordinates in the corresponding buffers (making sure to adhere to the mapping having (0,0) in the bottom left corner) and finally I sent the previously mentioned buffers using `MPI_Isend` (again non-blocking).

Using non-blocking communication allows to compute the inner points of the subdomain while the communication is still ongoing, thus hiding some of the communication latency.

Once the inner points are computed, I waited for all the non-blocking receives to finish using `MPI_Waitall`.

1.5. Implement parallel I/O [10 Points]

To implement the parallel I/O, I followed the provided code and I essentialy had to do just one crucial modification. In the provided demo data is stored this way `data[i*subdomain.ny + j]`, while in out case we do it this way `ptr_[i+j*xdim_]`.

in the function `MPI_Type_create_subarray`, the value `MPI_ORDER_C` indicates that the memory storage order to apply in that of C; row-major order. Since we are using column-major order, I had to invert the order of the dimensions, so instead of passing `{domain.nx, domain.ny}` I passed `{domain.ny, domain.nx}`.

The resulting image is shown below:

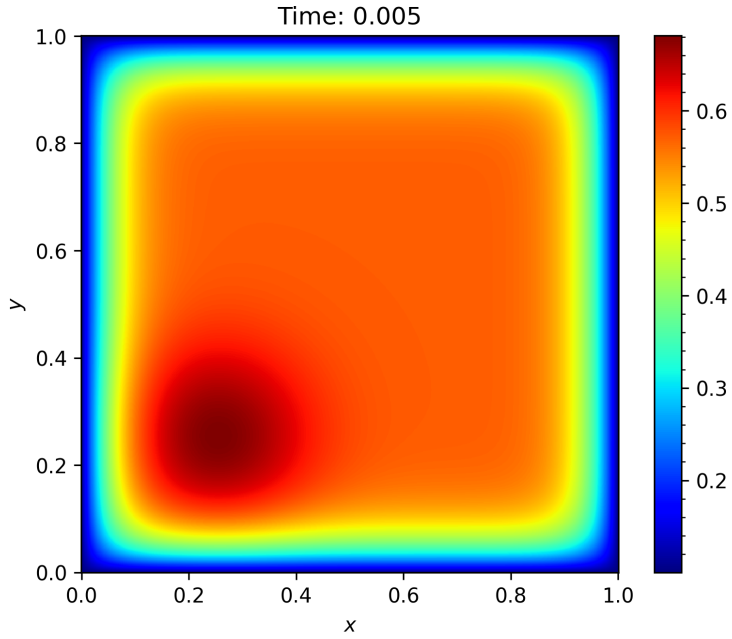

Figure 1: Output image generated using parallel I/O with MPI.

1.6. Strong scaling [10 Points]

The strong scaling results are shown in the following plot, I run the scaling test using both one node and 4 nodes for MPI, I compared all the result to the OpenMP implementation using the same number of threads and processes. The first six graphs show the MPI implementation using one node, the last six graphs show the MPI implementation using 4 nodes.

The first thing we can observe is that the MPI implementation is just slightly faster than the OpenMP one when using only one node, and that the biggest difference can be observed when we have a higher number of processes/threads. This is probably due to the fact that the communication overhead on a single node is lower than the overhead introduced by OpenMP thread management, and this difference can be observed more clearly when we have a higher number of processes/threads.

When using 4 nodes, we can see that the results are essentially the same up to four processes/threads, this is because I set the SLURM configuration to use 4 nodes with 4 tasks each, so up to 4 processes/threads we are still using only one node. When going beyond 4 processes/threads, we can very clearly see how the communication overhead is much higher than before, and this causes the MPI implementation to be significantly slower than the OpenMP one, especially for smaller sizes.

However, as the size increases, the performance gap between MPI and OpenMP decreases, this is because the computation time becomes more significant compared to the communication time, thus reducing the impact of communication overhead on the overall performance. Eventually, for the largest size tested (1024x1024) and for 16 processes/threads, the MPI implementation is able to slightly outperform both the MPI using only one node and the OpenMP implementation, showing that for large enough problems, MPI can scale better than OpenMP even when using multiple nodes. This might be because with larger problems, the advantage of having distributed memory and the ability to use more total memory across nodes outweighs the communication overhead.

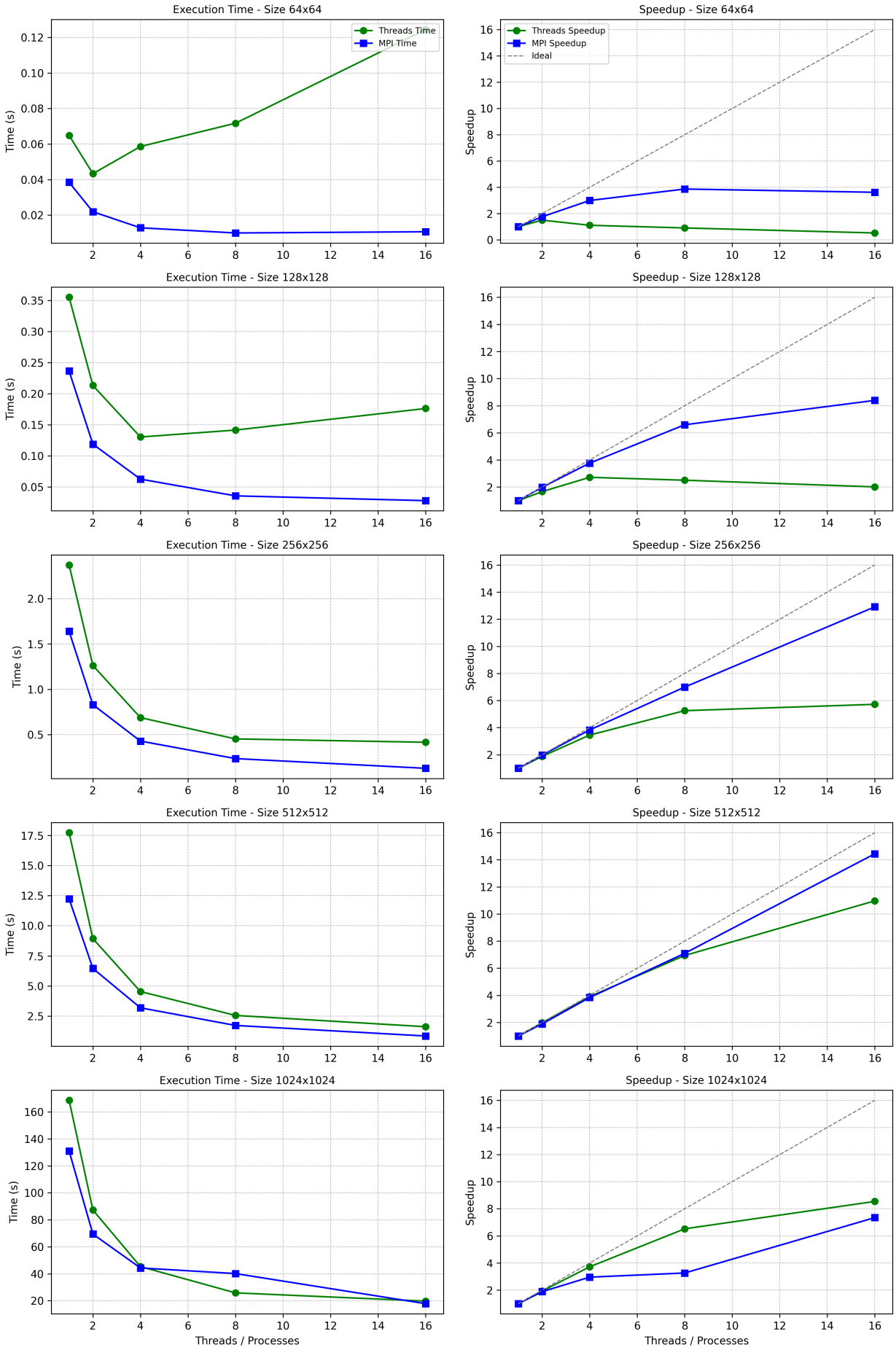

Figure 2: Strong scaling results for the MPI implementation using 1 node.

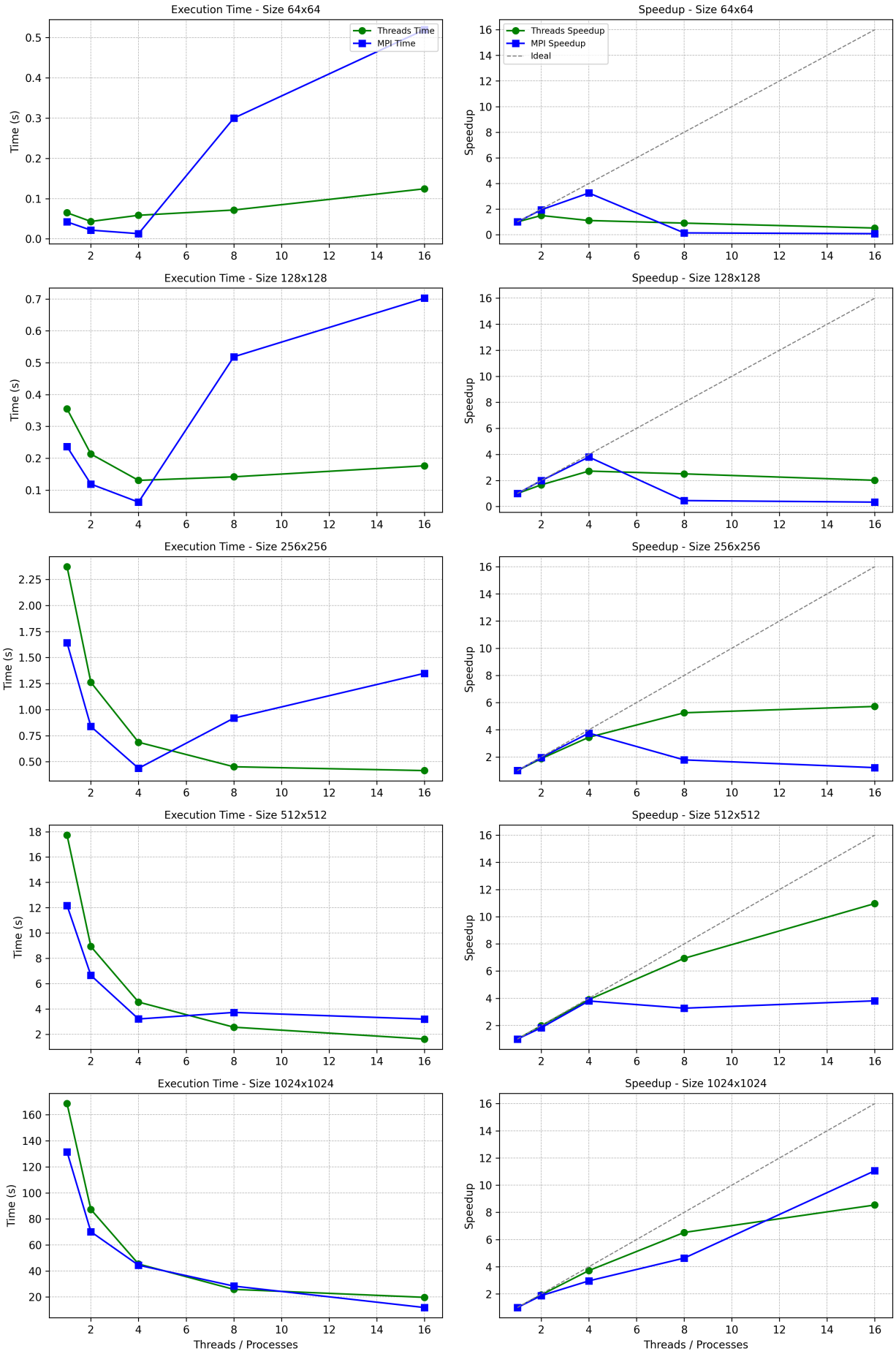

Figure 3: Strong scaling results for the MPI implementation using 4 nodes.

1.7. Weak scaling [10 Points]

As for strong scaling, I run the weak scaling test using both one node and 4 nodes for MPI and compared all the result to the OpenMP implementation using the same number of threads and processes. The results are shown in the following plots, the first two graphs show the MPI implementation using one node, the last two graphs show the MPI implementation using 4 nodes.

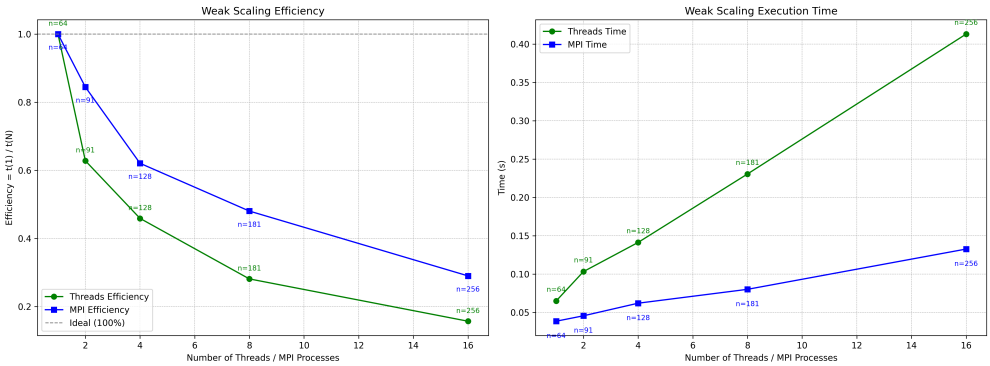

Figure 4: Weak scaling results for the MPI implementation using 1 node.

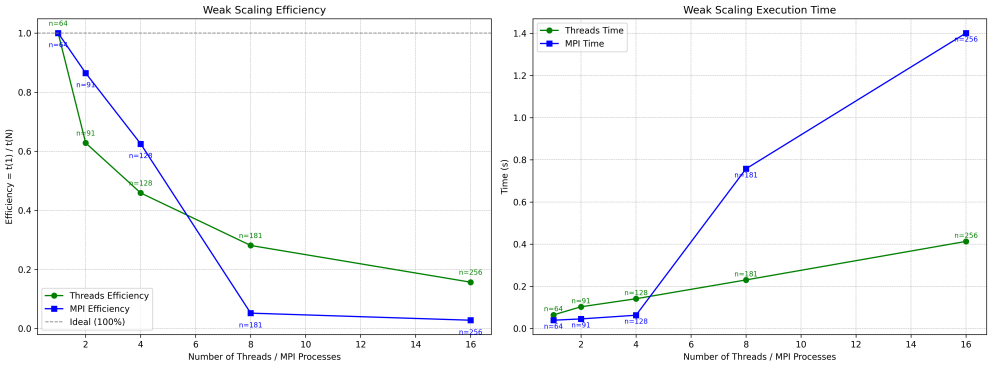

Figure 5: Weak scaling results for the MPI implementation using 4 nodes.

We can see that for one node, the MPI implementation is slightly faster than the OpenMP one, consistently with what we observed in the strong scaling test. When using 4 nodes, we can see that the MPI implementation is slower than the OpenMP one, again consistently with the strong scaling test, in particular, going from 4 processes to 8, we can observe a significant increase in execution time (again due to the communication overhead introduced by using multiple nodes), but then the execution not increases as much when going from 8 to 16 processes, suggesting that the communication overhead is not increasing significantly with more processes. This tells us that the MPI implementation scales relatively well as more processes (and nodes) are added.

2. Python for High-Performance Computing [in total 25 points]

In this task we are introduced to the use of MPI in Python and asked to implement three small examples.

2.1. Sum of ranks: MPI collectives [5 Points]

For this task I implemented a simple MPI program that computes the sum of the ranks of all processes using the `Allreduce` collective operation. I used both the pickle and the buffer implementations.

- For the **pickle** implementation, I used the `comm.allreduce` method, passing the rank of each process and the operation `MPI.SUM`, here serialization of the data is handled automatically by mpi4py.
- For the **buffer** implementation, I used the `comm.Allreduce` method, passing numpy arrays as buffers for both send and receive data.

2.2. Ghost cell exchange between neighboring processes [5 Points]

For this exercise I implemented a 2D grid decomposition using MPI Cartesian topology and performed ghost cell exchange between neighboring processes. I did so taking inspiration from the C++ implementation done in the project 4, but this time exchanging only the rank of neighboring processes.

The main steps I followed are:

- Initialize MPI and create a Cartesian communicator using `comm.Create_cart`.
- Determine the ranks of neighboring processes using `comm.Shift`.
- Sending to north and receiving from south using `comm.sendrecv`, and vice versa for all other directions.
- Print the received ranks from neighboring processes, I used `cart_comm.Barrier()` just to be sure that all processes printed the desired information in order.

Again I used both the pickle and buffer implementations, similarly to the previous exercise. For the buffer implementation, I used numpy arrays to hold the data to be sent and received.

2.3. A self-scheduling example: Parallel Mandelbrot [15 Points]

- The manager takes as input the communicator and the number of tasks, and assigns a task to each worker until all tasks are completed. Firstly, the manager sends an initial task to each worker, then it enters a loop where it waits for a worker to signal that it has completed its task using `comm.recv`, then it assigns a new task to that worker using `comm.send`, this continues until all tasks are assigned. Finally, the manager sends a termination signal to each worker and returns the list of completed tasks.
- In the worker implementation, each worker enters a loop where it waits for a task from the manager using `comm.recv`, then it computes the Mandelbrot set for the assigned task and signals the manager that it has completed the task using `comm.send`.
- In the main function, if I am rank 0, I call the manager function and then combine all tasks, otherwise I call the worker function.

The resulting Mandelbrot set image is shown below:

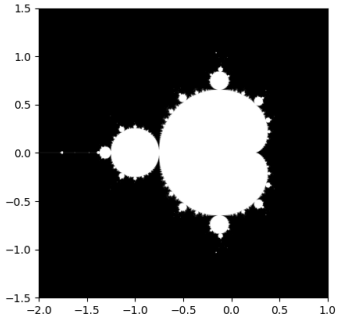

Figure 6: Mandelbrot set generated using manager_worker.py

The strong scaling result using 2 to 16 workers and a 4001x4001 domain of the manager-worker implementation is shown below:

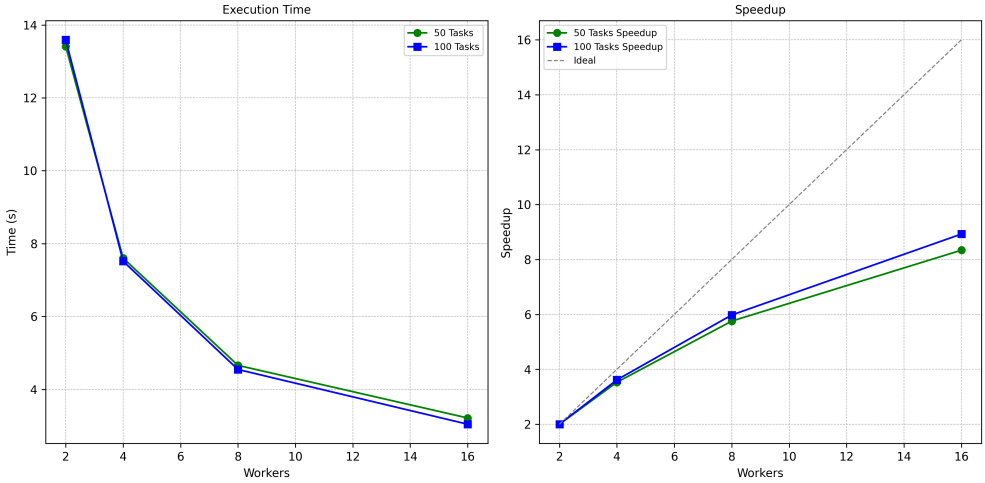

Figure 7: Strong scaling results for the manager-worker implementation of the Mandelbrot set.

We can see that dividing the work in 50 vs 100 tasks does not have a significant impact on the performance, as the 100 tasks implementation is only slightly faster than the 50 tasks one. The best result are obtained using 16 processes, where the 100 task implementation is around 6% faster than the 50 task one. This indicates that the mandelbrot computation is already well balanced using 50 tasks, and increasing the number of tasks does not significantly improve performance.

3. Task: Quality of the Report [15 Points]