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 7

Due date: 21.12.2022, 23:59

1. Parallel Space Solution of a nonlinear PDE using MPI [in total 35 points]

1.1. Initialize and finalize MPI [5 Points]

We aren't using multiple threads for each process, so we can just use **MPI_Init()** to initialize the environment. **MPI_Comm_rank()** and **MPI_Comm_size()** are used to get the current rank of this process and the number of total processes. At the very end, we use **MPI_Finalize()** to close the multiprocess environment.

1.2. Create a Cartesian topology [5 Points]

To create a Cartesian topology, we can just use the same methods used in project5. **MPI_Dims_create()** tell us the shape of the matrix, we can create the topology using **MPI_Cart_create()** (we can specify the matrix isn't periodic with an array of all zero values). To retrieve the coordinate in the grid of each process, we use **MPI_Cart_coords()** and **MPI_Cart_shift()** can be used to find the neighbours of each process.

1.3. Extend the linear algebra functions [5 Points]

To share the data in the linear algebra methods, we use **MPI_Allreduce()** specifying **MPI_SUM** as the reduction operation. This means that each process shares the partial result it computed, and the sum is shared between all processes. We use **MPI_Allreduce()** only on these two functions because they are the only two that work on the global domain and not only on the current process domain. This means that in order to compute the norm or dot product on the whole domain, we need to share the partial result computed by each process and sum them up.

1.4. Exchange ghost cells [10 Points]

Now we need to exchange ghost cells between the neighbours, 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 neighbours and send its own data to them. So for each neighbour, 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 custom type to send them.
- Send the newly created buffer to the neighbour. 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 neighbours to do that. Now we need to compute the outer points of the grid, so we wait for the results from the neighbours 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.

1.5. Scaling experiments [10 Points]

1.5.1. Strong Scaling

I run a strong scaling experiment to check how well the program scales when the size of the matrix and the number of processes increases. The analysis has been done looking at the time execution and iteration per seconds. All the experiments have been done in two ways: having all the processes on one node and on different nodes. The experiments have been made with sizes $128x128, 256x256, 512x512, 1024x1024$ with $1, 2, \ldots, 20$ processes. For the simulation, I used 0.01 as the last timestamp of the simulation, doing 100 iterations.

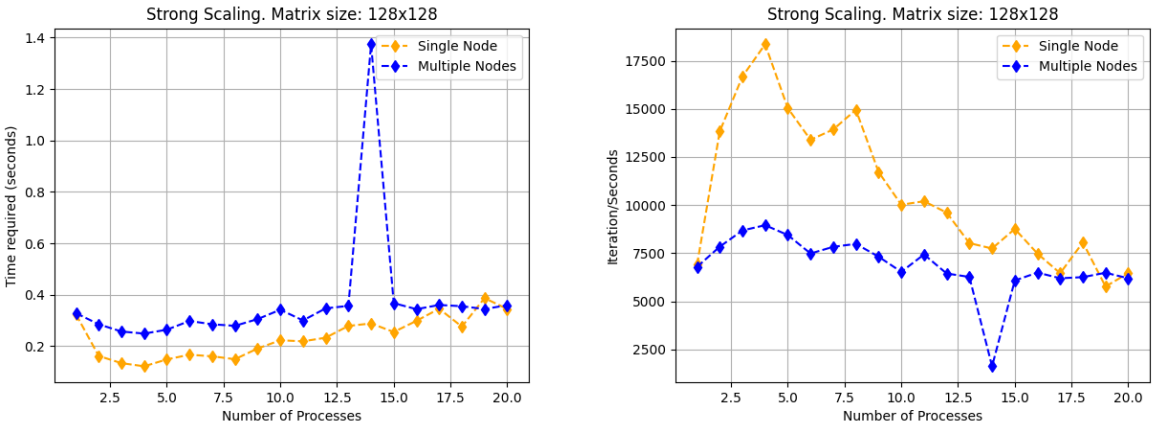

Figure 1: Strong Scaling Experiment, Matrix Size 128x128

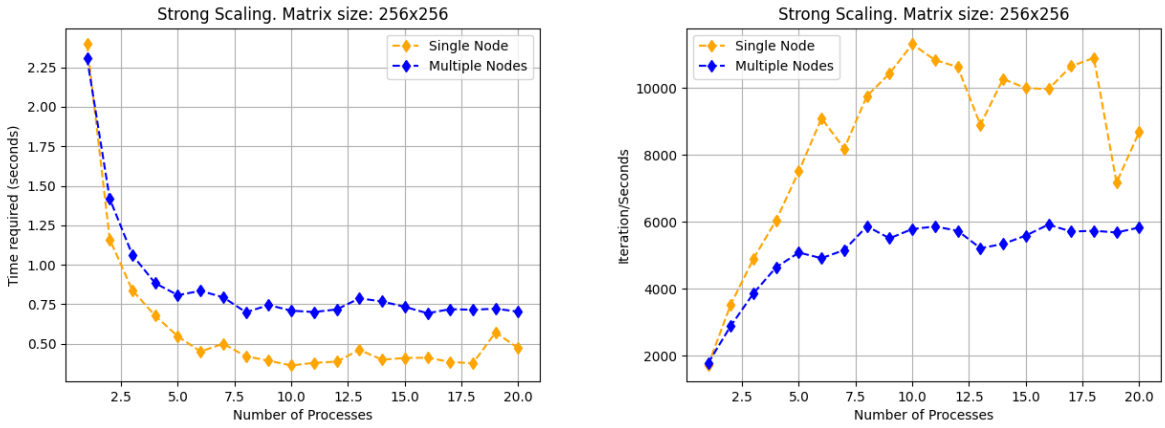

Figure 2: Strong Scaling Experiment, Matrix Size 256x256

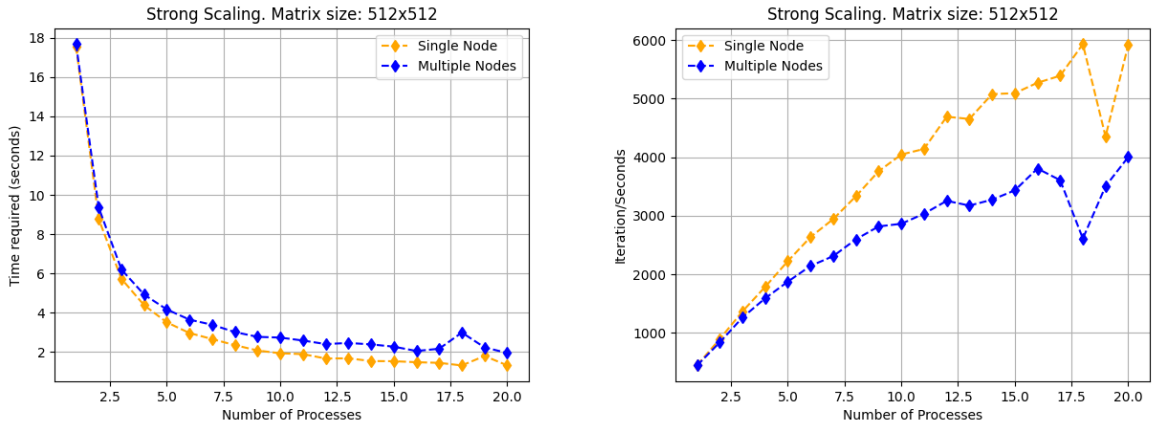

Figure 3: Strong Scaling Experiment, Matrix Size 512x512

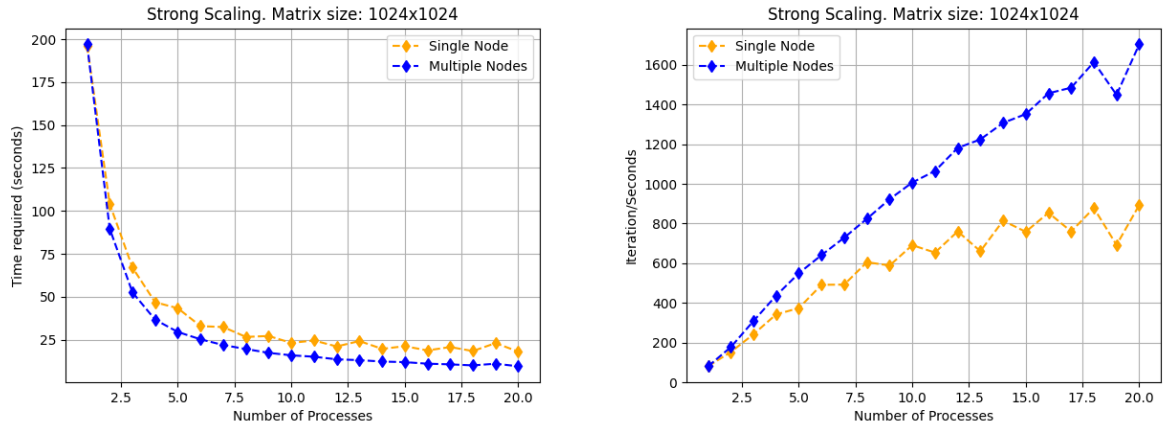

Figure 4: Strong Scaling Experiment, Matrix Size 1024x1024

The first thing we can see is that the plots about time required and iterations per second show the same results. When we look at the plots on matrices of side 128, 256 and 512 we see that the performance on a single node is actually faster than having one process per node. This is due to the fact that when we have multiple nodes, we have a communication overhead that is higher than having all the processes on the same node of the cluster. Instead, the advantage of having multiple

nodes is that each node has to work with matrices of smaller size, this make the program more cache friendly, this factor seems to overcome the communication overhead on the biggest matrix. The only size where the program doesn't scale well is on the matrix $128x128$, instead in all the other plots we see how the time required decreases and the iterations per second ration increases when the number of processes increase. There is a strange spike on the matrix $128x128$ with 14 processes, but being the simulation that short, I believe it's just an accurate measurement.
Comparing these result with the one I obtained with the project three (there we used threads instead of processes) we have that using multiple processes the program scales much better, probably this is due to the fact that each process can work on its part of the domain without having to worry about other processes, instead using threads the data is shared between all of them so managing their access take more time.

1.5.2. Weak Scaling

For the weak scaling experiments, I analysed the time required and the iterations per second ratio for matrices of different sizes and different number of processes. In particular, when I have 20 processes the matrix is of size $1024x1024$, for each other test the matrix size is calculated in a way that for each test the number of grid cells for each process is the same. Here we can see that when the number of processes isn't that high (let's say less than 15) both versions (on a single node and on multiple nodes) scale in the same way, which isn't perfect, but it isn't so bad. Instead, when the matrices are big, we see again that having multiple nodes is much better than having only one node. This is again because on a single node you are limited by the size of the level three cache, instead when the matrix is split on smaller matrices for different nodes this isn't a problem any more.

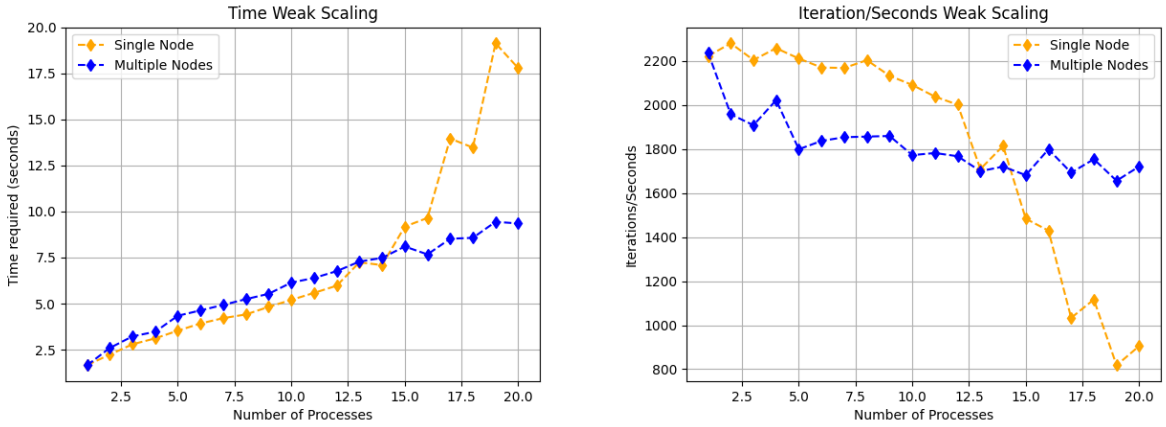

Figure 5: Weak Scaling Experiment, Matrix Size 1024x1024

2. Python for High-Performance Computing (HPC) [in total 50 points]

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

We can sum the ranks of the different processes by using a reduction and specifying the sum operation. We can do that in both pickle-based communication (**allreduce()**) and direct array data communication of buffer-provider objects (**Allreduce()**).

- Pickel Based:

```
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
total_sum = comm.allreduce(rank, op=MPI.SUM)
```

- Direct array data communication of buffer-provider objects:

```
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = np.array([comm.Get_rank()])
total_sum = np.array([0])
comm.Allreduce(rank, total_sum, op=MPI.SUM)
```

2.2. Domain decomposition: Create a Cartesian topology [5 Points]

Using the methods suggested in the assignment, we can create a Cartesian topology in the following way.

```
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
dims = [0, 0]
dims = MPI.Compute_dims(size, dims)
periods = [True, True]
comm_cart = comm.Create_cart(dims, periods=periods)
coords = comm_cart.Get_coords(rank)
neigh_west, neigh_east = comm_cart.Shift(1, -1)
neigh_south, neigh_north = comm_cart.Shift(0, -1)
```

dims specifies that we don't have any requirements on the form of the grid, while **periods** specifies that we want the topology to be periodic. In order to get the rank of the neighbours of the current process, we use the **Shift()** method. Here the coordinates are assigned in a way that process 0 is the one in the bottom left corner of the grid, process 1 is on its right and so on.

2.3. Exchange rank with neighbours [5 Points]

Once we have the coordinates of the neighbours, we can just use the method **sendrecv()** to exchange ranks between neighbours. This method allows us to specify the rank of the process that will receive our rank and the rank of the process that will send us its rank.

```
west_rank = comm_cart.sendrecv(rank, neigh_east, neigh_west)
east_rank = comm_cart.sendrecv(rank, neigh_west, neigh_east)
south_rank = comm_cart.sendrecv(rank, neigh_north, neigh_south)
north_rank = comm_cart.sendrecv(rank, neigh_south, neigh_north)
```

2.4. Change linear algebra functions [5 Points]

Each process has information about only a portion of the whole domain, so, each one will compute the desired function on the data it has, but then it needs to share it with the other processes (each process needs to have the data about the whole domain), in order to do that we can use the **allreduce()** method again. To compute the information required about its domain, each process needs to iterate over some values. In order to do that, in python you can use standard loops, but this is really slow considering python is an interpreted language. A faster alternative is represented by the **numpy** module. From the tests made using **numpy** we reach an execution time that is more than ten times faster than the one obtained using standard loops.

```
def hpc_dot(x, y):
    return x.domain.comm.allreduce(np.dot(x._inner.flatten(), y._inner.flatten()))

def hpc_norm2(x):
    return x.domain.comm.allreduce(np.dot(x._inner.flatten(), x._inner.flatten())) ** 0.5
```

2.5. Exchange ghost cells [5 Points]

We want the processes to share their boundaries, in order to do that we use two functions: the first one, **exchange_startall()** which starts the sharing process and **exchange_waitall()** that waits for the end of the sharing process. Both methods are called on all the processes. The first method starts the communication with the neighbours, to do that it does the following steps:

- It starts to receive the boundary from the neighbour using **Irecv()** which is a non-blocking receive call.
- It copies the data to send to the neighbour into a buffer.
- It starts to send the content of the buffer using a non-blocking send operation: **Isend()**.

We use these two methods, so the process can compute the inner-points of the grid while the communication takes place. The **exchange_waitall()** method just waits for the completion of communication between the processes. To each send and receive operation there is a **request** associated, we can use that **request** to know when an operation is completed.

2.6. Scaling experiments [5 Points]

For both strong and weak scaling, I repeated all the experiments presented for the C version of the application, the only difference is that, being python slower the tests took more time, so I decided to run the version with one process per node only for a maximum of 10 nodes.

2.6.1. Strong Scaling

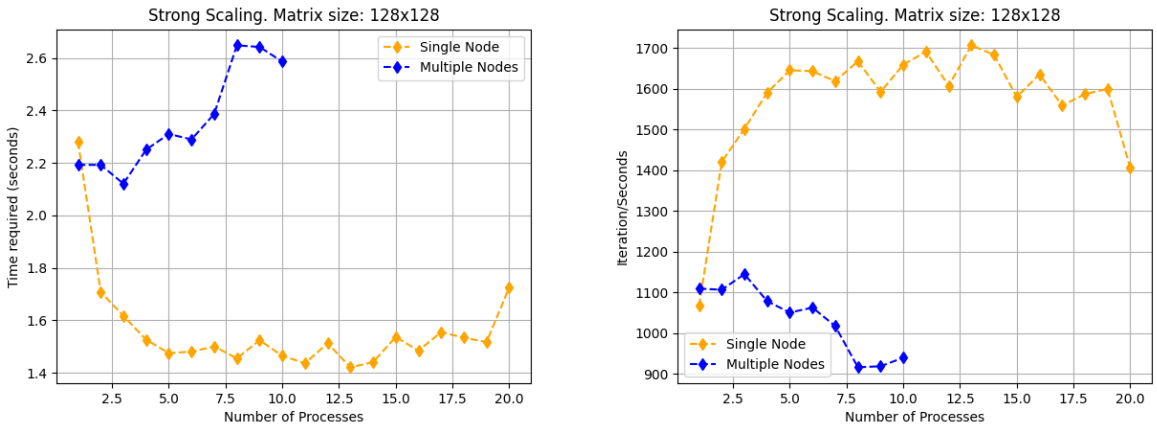

Figure 6: Strong Scaling Experiment, Matrix Size 128x128

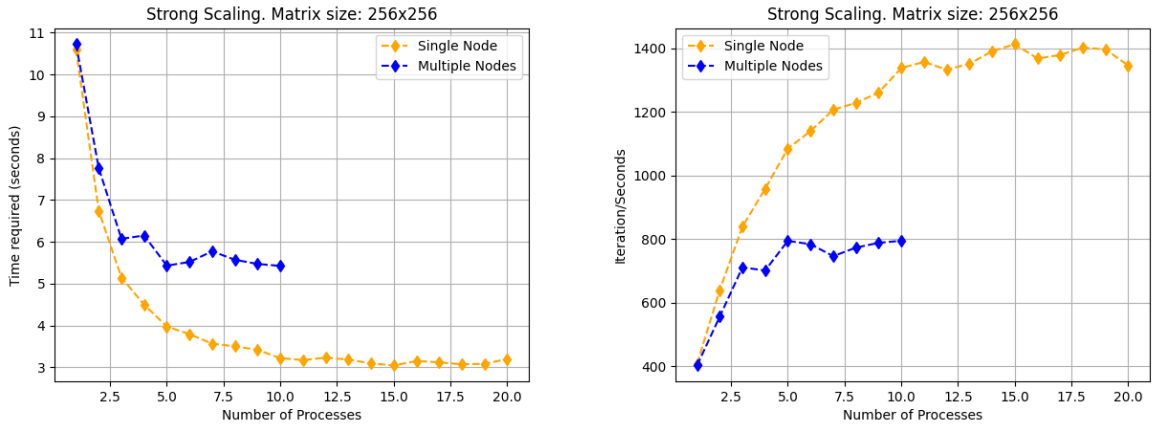

Figure 7: Strong Scaling Experiment, Matrix Size 256x256

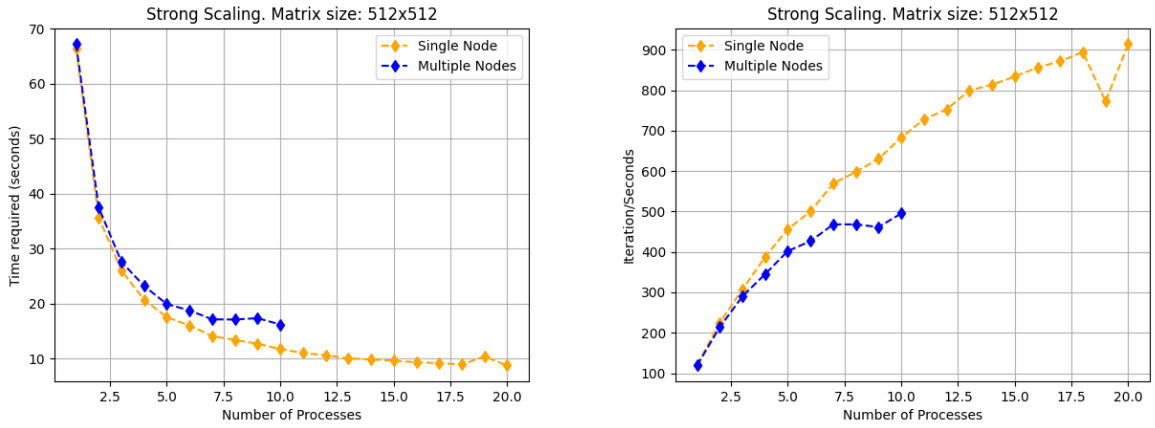

Figure 8: Strong Scaling Experiment, Matrix Size 512x512

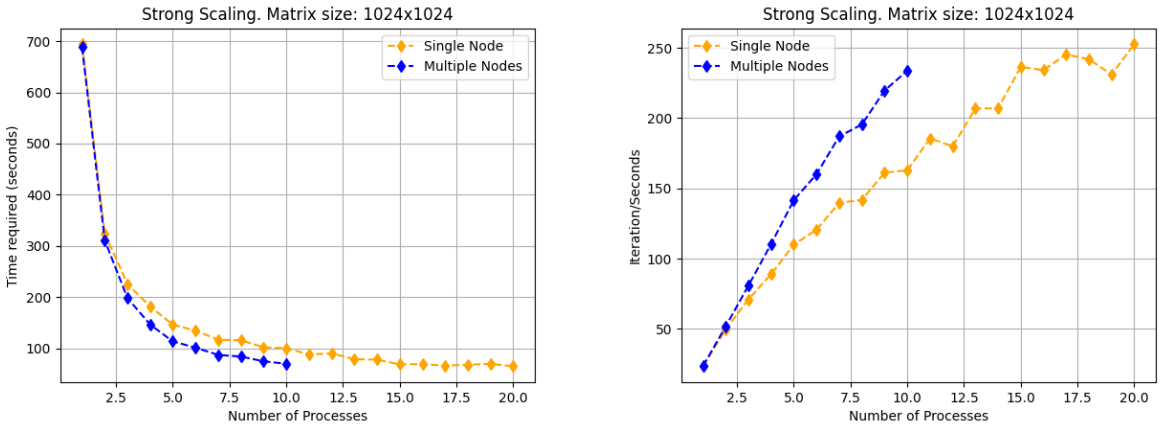

Figure 9: Strong Scaling Experiment, Matrix Size 1024x1024

Like it happened for the C version, the tests with all the processes on the same node have better results than the ones executed on multiple nodes for matrix sizes: $128x128$, $256x256$ and $512x512$, instead for the biggest matrix, having the computations split into different nodes improves the performance. For all sizes apart from the smallest one, we have a good strong scaling, this can be noted by the fact that increasing the number of processes, the lines on the plots that show the

time required are all decreasing while the iterations per second ratio increases. Here we can see the difference in time between the C and python implementation (using one node).

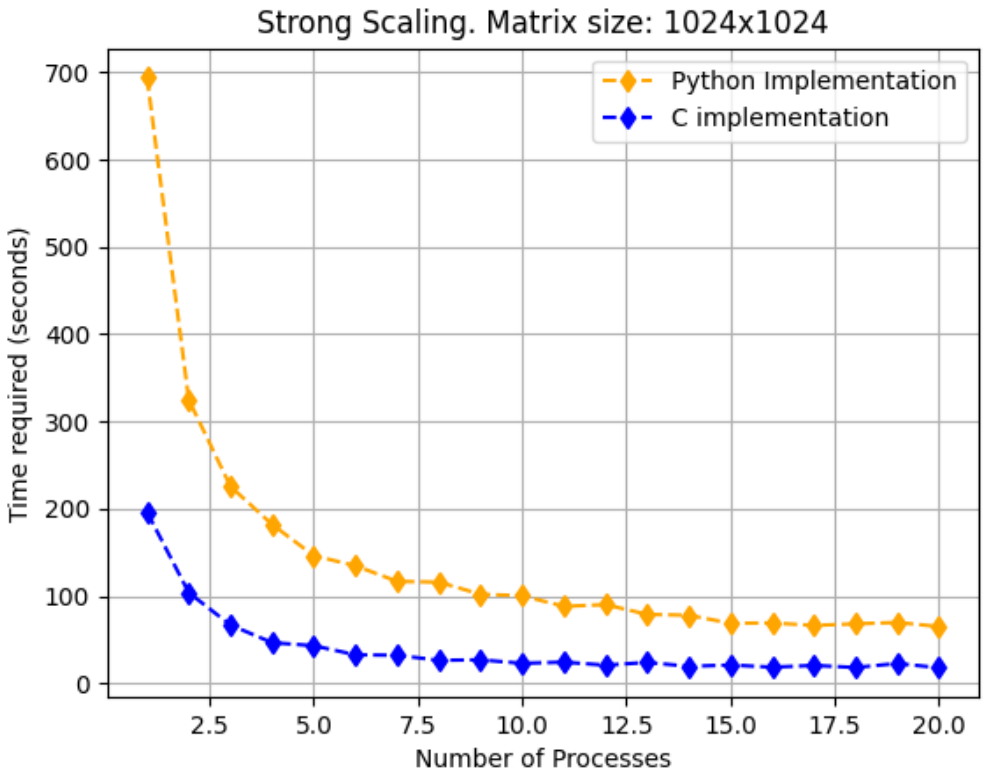

Figure 10: Comparison between python and C

C is much faster than python for each number of processes. This is due to the fact that C is a compiled language, while python needs to be interpreted during runtime.

2.6.2. Weak Scaling

For the weak scaling experiments, I analysed the time required and the iterations per second ratio for matrices of different sizes and different number of processes. In particular, when I have 20 processes the matrix is of size $512x512$, for each other test the matrix size is calculated in a way that for each test the number of grid cells for each process is the same. Firstly, we can see how the program weakly scales better when run on a single node. We can also notice how the python weakly scales better than the C implementation of the application. Looking at the lines for the multiple nodes, we can also assume that if the tests were run on more than 10 nodes, we would have a much worse scaling on those instances.

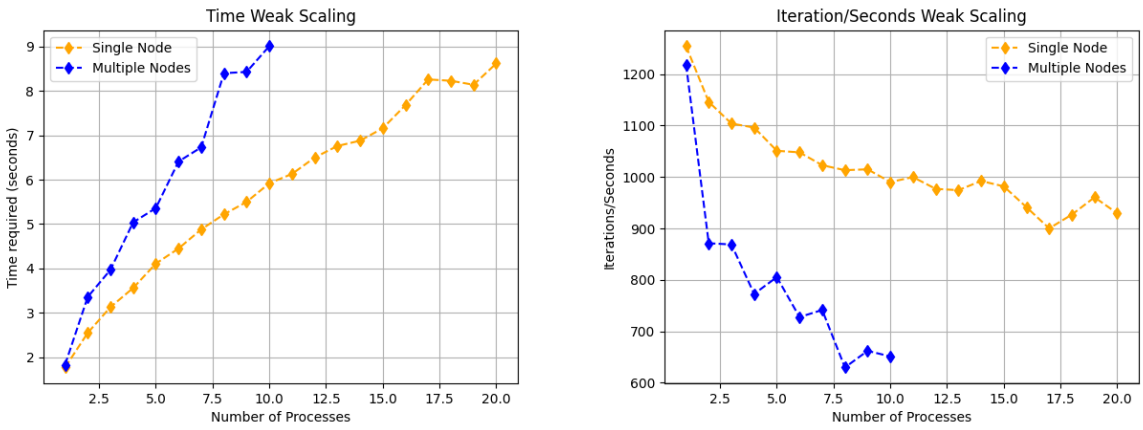

Figure 11: Strong Scaling Experiment, Matrix Size 1024x1024

2.7. A self-scheduling example: Parallel Mandelbrot [20 Points]

The objective here is to compute the Mandelbrot set using the manager-worker paradigm. In this paradigm, there are two possible roles for each process: the manager that coordinates the work and the worker which actually carry out part of the whole computation. A single manager manages all the workers. For the Mandelbrot set, we can divide the whole domain in a set of patches that divides the global structure in multiple vertical patches. Once we have these patches, we can compute the Mandelbrot set on their subdomains and combine the results in the end. This paradigm is used because there are portions of the Mandelbrot set domain that are heavier to compute than other, so, if we tried to divide the computation in few partitions (for example one partition for each process) the workload would be unbalanced.
The manager has the list of tasks (patches) that need to be computed and starts to give to each process a task to compute. After that, it waits for the first message from a worker and, if there are any tasks left to do, it sends the worker a new task. In this way, especially dividing the domain in many tasks, each worker will end up with roughly the same amount of computation to perform.
An elegant way to implement the communication of this type of paradigm is to define different type of messages (different tags).

- **TAG_TASK**: it's used when the manager is sending a new task to a worker.
- **TAG_DONE**: it's used by the manager to tell a worker that there isn't any more computation to perform (in this way, the worker can just stop executing).
- **TAG_TASK_DONE**: it's used by the workers to tell the manager that they completed the current task and that they can start to perform the next one.

The communication was implemented using blocking send/receive operations because the communication time it's a small fraction of the total execution time. Indeed, I didn't observe major improvements using non-blocking call to send tasks to the workers. When the computation of each task is done, the manager can combine the different patches to obtain the final image.

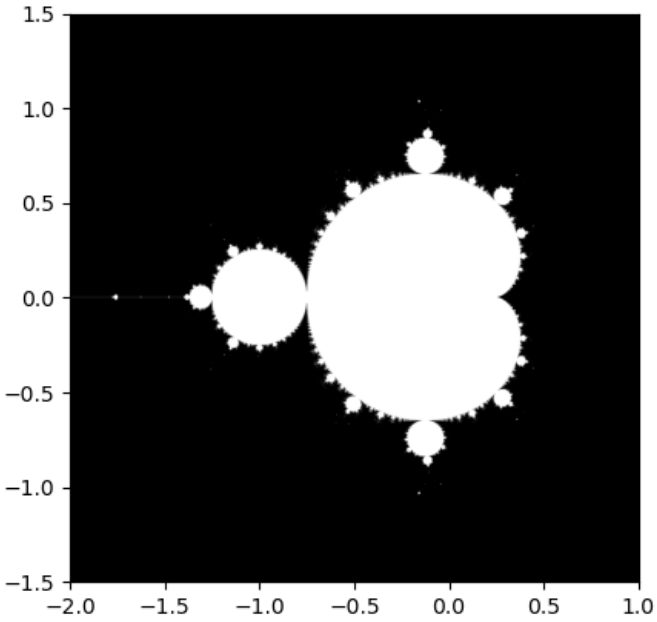

Figure 12: Mandelbrot Set Visualization

We divided the domain in 50 and 100 tasks and tested the performance using $1, 2, 4, 8, 16$ workers, obtaining the following execution times:

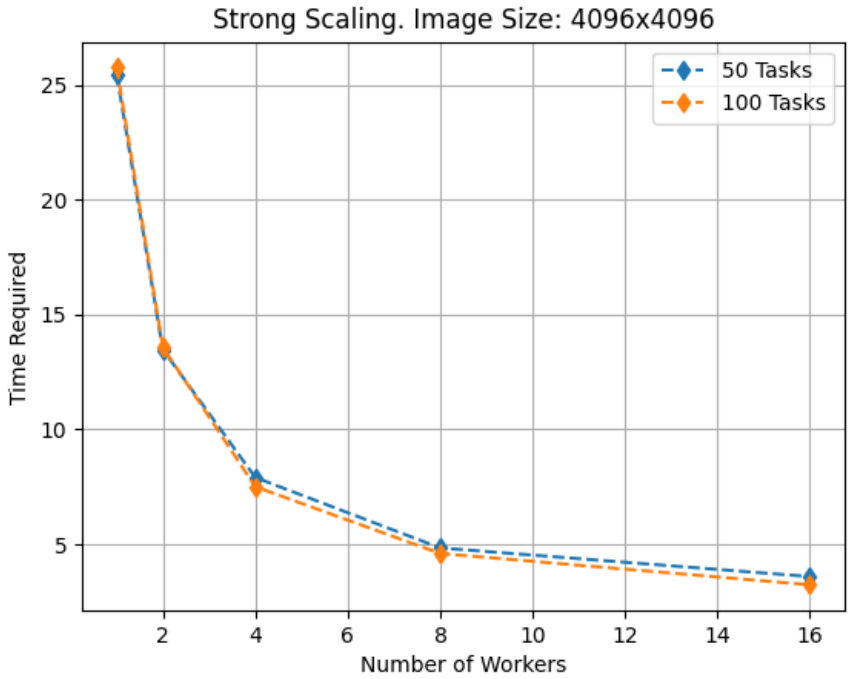

Figure 13: Strong Staling of Manager-Worker paradigm

The execution time scales really well with the number of workers. We can also see that the number of tasks doesn't make a big difference in terms of performance, using 100 tasks instead of 50 is slightly better, but the difference is minimal. I also tried to store into a vector the execution

time of each process, and in the end compute the variance of this vector. The variance was really similar using 50 and 100 tasks.

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

Each project will have 100 points (out of which 15 points will be given to the general quality of the written report).

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 solutions;
  - your write-up with your name project_number_lastname_firstname.pdf.
- Submit your .tgz through Icorsi.