Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

High-Performance Computing Lab for CSE **2024**

Student: CARLA JUDITH LOPEZ ZURITA

Solution for Project 5

Due date: Monday 13 May 2024, 23:59 (midnight).

Project 5 is about continuing the work on the parallel solution of a nonlinear PDE using MPI and learning about Python for High-Performance Computing.

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

Following the steps of Project 3, we will parallelize the solution of the diffusion equation using MPI. We will also implement parallel I/O to write the solution to disk. We will then evaluate the strong and weak scaling of our parallel implementation.

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

The code should initialize MPI, get the number of processes, and print a welcome message. After adding the header for the library, only a few lines of code were changed to initialize MPI and print the welcome message. The following code snippet shows the changes made to the code:

```
std::cout << "version :: C++ MPI" << std::endl;
std::cout << "processes :: " << size << std::endl;
```

As well as finalizing MPI at the end of the code and adding relevant print conditions for only the master thread.

1.2. Domain decomposition [10 Points]

On the `data.cpp` file, some changes were added to the SubDomain struct very similar to those in Project 3. The strategy implemented is based in determining automatically the number of subdomains in the x and y dimensions using the `MPI_Dims_create` function. The following code snippet shows the changes made to the code:

```
int dims[2] = {0, 0};
MPI_Dims_create(mpi_size, 2, dims);
ndomy = dims[0];
ndomx = dims[1];
```

We also added the `MPI_Cart_create` function to create the Cartesian as well as the `MPI_Cart_coords` function to determine the coordinates of each process and the `MPI_Cart_shift` function to determine the neighbors. We also added some conditions to adjust the cases for when a dimension is not divisible by the number of processes.

The `MPI_Dims_create` function has the eadvnatage of trying to create a grid with the most square-like shape possible, which is a good strategy for domain decomposition. This is important since it helps to minimize the communication overhead between processes. For some number of processes, "slab" decomposition is implemented. Square-like shapes are preferred since they have a better balance between surface to area ration, which improves computational efficiency.

1.3. Linear algebra kernels [5 Points]

Next, we parallelized the relevant linear algebra kernels in `hpc_XXX` in `linalg.cpp` using MPI. Not all funcitons needed to be modified, only the ones that required communication between processes, such as `hpc_dot` and `hpc_norm2`. Both follow the same pattern of using the `MPI_Allreduce`. The code snippet below shows the changes made to the `hpc_dot` function:

```
// computes the inner product of x and y
// x and y are vectors on length N
double hpc_dot(Field const& x, Field const& y) {
    double global_result;
    double local_result = 0.0;
    int N = y.length();
    // compute local result
    for (int i = 0; i < N; i++) {
        local_result += x[i] * y[i];
    }
    MPI_Allreduce(&local_result, &global_result, 1, MPI_DOUBLE, MPI_SUM,
        MPI_COMM_WORLD);
    return global_result;
}
```

The rest of the functions were left unchanged.

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

The next step was to implement the exchange of ghost cells between neighboring processes. This was done by adding the `MPI_Isend` and `MPI_Irecv` functions to the `diffusion` function in `operators.cpp`. This requeired following a series of steps. First, we needed to fill the send buffer with the values of the ghost cells. This was a bit tricky becase the indices needed to be inverted.

```
// Fill buffers
for (int k = 0; k < nx; k++)
{
    // The indices needed to be inverted
    buffN[k] = s_new(k,jend);
    buffS[k] = s_new(k,0);
}
for (int k = 0; k < ny; k++)
{
    buffE[k] = s_new(iend,k);
    buffW[k] = s_new(0,k);
}
```

Then, we needed to send the values and receive the values from the neighbours using non-blocking point-to-pointcommunication. A custom data type was not needed since we were sending contiguous data. The following code snippet shows the changes made to the code:

```
MPI_Request request[8];
int tag1 = 1, tag2 = 2, tag3 = 3, tag4 = 4;
int count = 0;

// SEND
// to top
MPI_Isend(&buffN[0], nx, MPI_DOUBLE, domain.neighbour_north, tag1, domain.
    comm_cart, &request[count++]);
// to bottom
```

```
MPI_Isend(&buffS[0], nx, MPI_DOUBLE, domain.neighbour_south, tag2, domain.
    comm_cart, &request[count++]);
// to right
MPI_Isend(&buffE[0], ny, MPI_DOUBLE, domain.neighbour_east, tag3, domain.
    comm_cart, &request[count++]);
// to left
MPI_Isend(&buffW[0], ny, MPI_DOUBLE, domain.neighbour_west, tag4, domain.
    comm_cart, &request[count++]);

// RECEIVE
// from top
MPI_Irecv(&bndN[0], nx, MPI_DOUBLE, domain.neighbour_north, tag2, domain.
    comm_cart, &request[count++]);
// from bottom
MPI_Irecv(&bndS[0], nx, MPI_DOUBLE, domain.neighbour_south, tag1, domain.
    comm_cart, &request[count++]);
// from right
MPI_Irecv(&bndE[0], ny, MPI_DOUBLE, domain.neighbour_east, tag4, domain.comm_cart
    , &request[count++]);
// from left
MPI_Irecv(&bndW[0], ny, MPI_DOUBLE, domain.neighbour_west, tag3, domain.comm_cart
    , &request[count++]);
```

Finally, we added a `MPI_Waitall` function to wait for all the communications to finish.

1.5. Implement parallel I/O [10 Points]

Finally, we implemented parallel I/O to write the solution to disk. We used as base the code provided in the assignment. The code in `main.cpp` was modified to use a new version of `write_binary` that uses MPI I/O. The function is based on the `MPI_File_write_at_all` function, calling on each subdomain to write its own data. One of the main changes was to change the order to `MPI_ORDER_FORTRAN` to match the order of the data in the file. An example of the output image is shown in Figure 1, which was generated using 2048×2048 grid points and 32 processes.

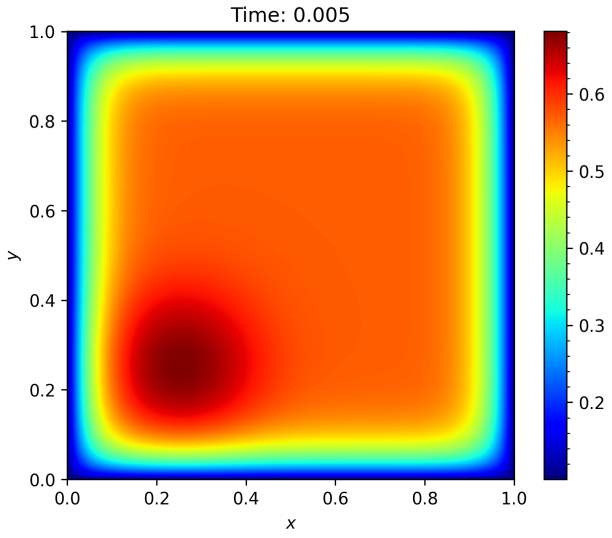

Figure 1: Example of the output image using 2048×2048 grid points and 32 processes.

1.6. Strong scaling [10 Points]

Each configuration was run fifty times in order to get a more representative time. The distribution of the data is shown in Figure 2. Warm up runs were included in the data, since they didn't seem to fall outside the distribution. We can see that for low number of processes, the data is not very

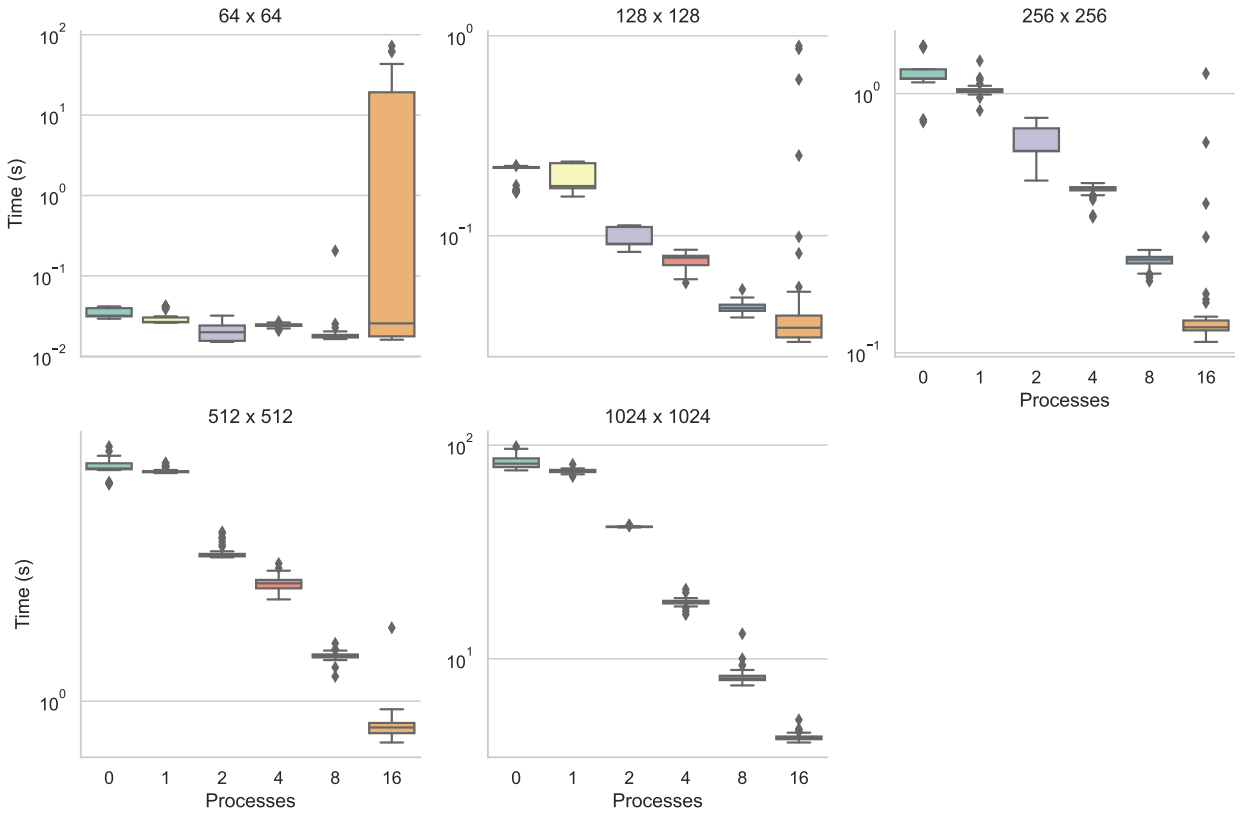

Figure 2: Data distribution of the runtimes for different configurations of the parallel diffusion equation solver. Zero processes indicates the serial version.

spread out, but as the number of processes increases, the distribution is also wider, specially for a low number of grid points. This is expected since the communication overhead increases with the number of processes, and is specially noticeable for a low number of grid points where parallelization is not as effective. We can see that for a high number of grid points, the distribution is consistent. As serial version, I used the version of the code that doesn't use MPI, and is shown in the plot using the label 0 processes. Interestingly, we can see some cases where the parallel version using one thread is faster than the serial version. Both versions were both compiled using the same flags but different compilers, namely `c++` for the serial version and `mpicxx` for the parallel version. I am not sure if this affects the results. The results of the strong scaling are shown in the Figure 3. The data point used to generate the plot is the median of the fifty runs. We can see consistent speedup for a high number of grid points, but the speedup is not as good for a low number of grid points. The case of grid size 64×64 is the one with the worst speedup, showing almost no speedup at all. This is expected due to the communication overhead as discussed previously. The case of grid size 1024×1024 is the one with the best speedup, showing beyond ideal speedup, which could be a result of the times taken from the serial version being slower due to some unknown reason.

1.7. Weak scaling [10 Points]

Again the data was run fifty times for each configuration. The distribution of the data is shown in Figure 4. Warm up runs were included in the data, since they didn't seem to fall outside the distribution. This plot is inverted with respect to the strong scaling data visualization, since the needs for the weak scaling require different configurations. We can see that for each number of

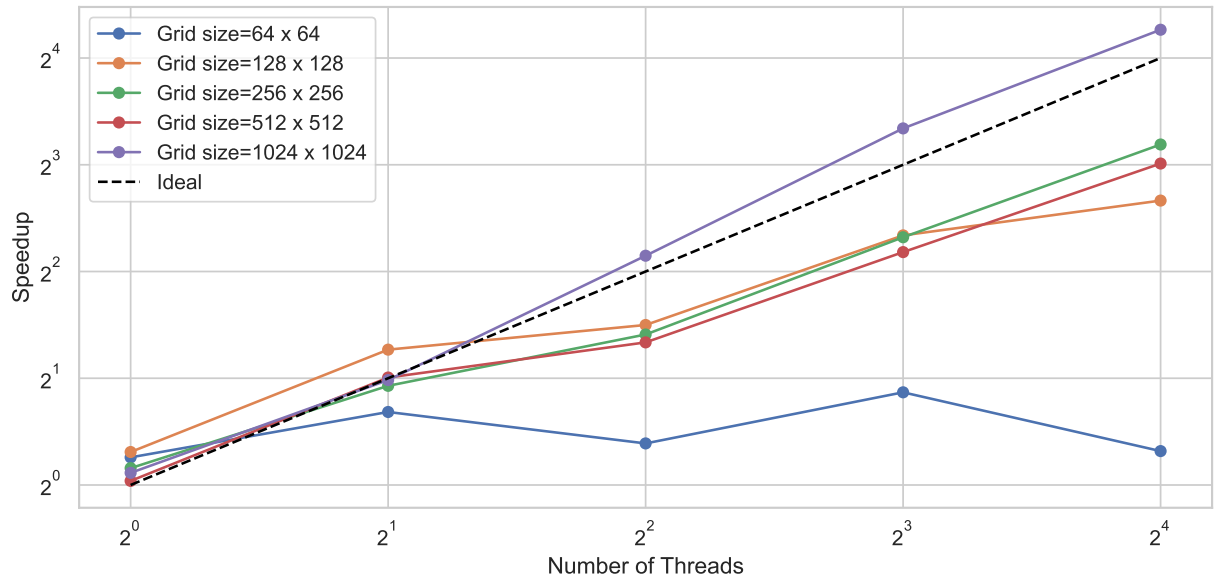

Figure 3: Strong scaling of the parallel diffusion equation solver using different grid sizes compared against serial version. Representative data point is the mean of 50 runs.

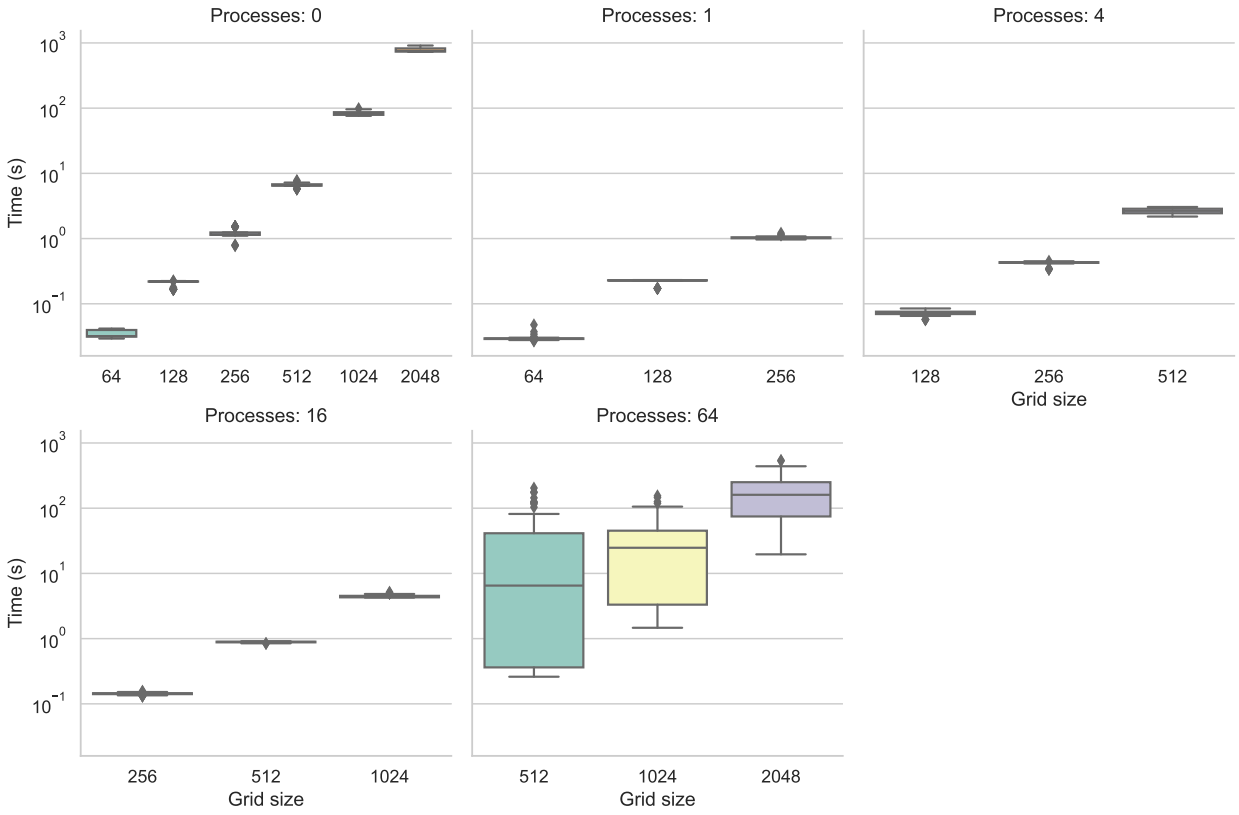

Figure 4: Data distribution of the runtimes for different configurations of the parallel diffusion equation solver.

processes, the data increases as the number of grid points increases. This is expected since the computation time increases with the number of grid points. The data is not very spread out, which is a good indication of low variability in runtimes, except for the case of 64 processes, where the distribution is significantly wider. This is expected since it correlates with the overhead in communication.

The results of the weak scaling are shown in the Figure 5. It shows a somewhat expected result,

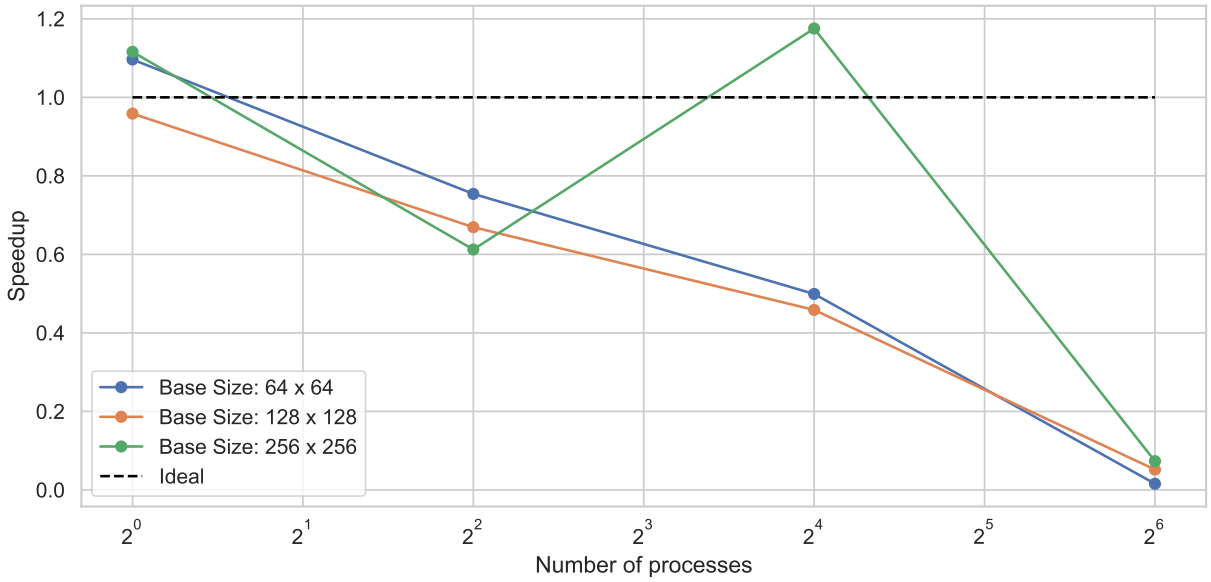

Figure 5: Weak scaling of the parallel diffusion equation solver compared against serial version, using different base sizes. Representative data point is the mean of 50 runs.

except for the case of 256×256 grid points as base case, where the speedup seems to be better than expected. The behaviour of the rest of the data points is consistent with the expected results, showing a decrease in speedup as the number of processes increases, even though the number of grid points also increases. Interestingly, the decrease in speedup is consistent for all the cases of starting grid points, which tells us that for this condigurations, using 32 processes is not good practice. Combined with the strong scaling results, we can see that the best number of processes to use is 16, since it shows a good speedup for all the cases. I am not sure why the speedup for the case of 256×256 grid ppoints using 16 processes is better than expected, but it could be related to hardware being optimal at that number of processes and grid points.

2. Python for High-Performance Computing [in total 40 points]

In this section of the project, we will learn about MPI for Python and use it to parallelize the computation of the Mandelbrot set. We will use a manager-worker pattern to distribute the computation of the Mandelbrot set among the workers.

Every python code using mpi4py is initialized with the following lines:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
proc = MPI.Get_processor_name()
```

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

Using MPI for Python's collective communication methods, we wrote a script that computes the sum of all ranks. This was done using the pickle-based communication of generic Python objects as well as the fast, near C-speed, direct array data communication of buffer-provider objects. The following code snippets show the changes made to the code using lower case functions:

```
global_sum = comm.gather(rank)
if rank==0:
print(f"Sum: {sum(global_sum)}")
```

as well as using the uppercase functions:

```
rank_array = np.array(rank, dtype=np.int32)
total_sum = np.zeros(1, dtype=np.int32)
comm.Allreduce(rank_array, total_sum, op=MPI.SUM)

if rank==0:
print(f"Sum: {sum(total_sum)}")
```

As the description of the assignment suggests, the uppercase functions are more efficient and also need to use arrays to store the data, otherwise the code will not work. Running the code with `salloc -n 8 --nodes=4 --ntasks-per-node=2`, the output for both was:

```
Sum: 28
```

Both methods worked as expected (7+6+5+4+2+3+1+0).

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

Now, we implemented the exchange of ghost cells between neighboring processes using MPI for Python. This replicates the same functionality as in the C++ code, done in project 4. First, we need to determine the number of subsections in the x and y dimensions, depending on the number of processes. This was done using the `Compute_dims` function. Then, we created a periodic Cartesian grid using the `Create_cart` function. We also used the `Shift` function to determine the coordinates of the neighbors. The following code snippet shows the implemented code:

```
# Create dims depending on number of ranks
dims = [0, 0]
dims = MPI.Compute_dims(size, dims)

# Create cartesian grid
periods = [True, True]
cart2d = comm.Create_cart(dims, periods, reorder=False)
coords = cart2d.Get_coords(rank)

# Get coordinates of the neighbours
west, east = cart2d.Shift(direction=1, disp=1)
north, south = cart2d.Shift(direction=0, disp=1)
```

After filling a numpy array with the value of the rank, we excahnge the ghost cells with the neighbors using the `Sendrecv` function. I decideded to use this function instead of the `sendrecv` for python objects, since it is more efficient and made sense with the description of the project since we are already dealing with data arrays. Also, the `Sendrecv` function is non-blocking, in contrast to the separate `Send` and `Recv` functions. Similarly to the C++ code, we needed send contiguous data; we didn't need to create a custom data type, but we did need to convert the columns into contiguous numpy arrays. Then, we assigned the received data to the corresponding columns. The following code snippet shows the implemented code:

```
# Contiguous arrays to store columns
col_east = np.empty(SUBDOMAIN, dtype=np.int32)
col_west = np.empty(SUBDOMAIN, dtype=np.int32)

# Send and receive
```

```
comm.Sendrecv(data[0, 1:-1], north, 1, new_data[DOMAINSIZE-1, 1:-1], south, 1)
comm.Sendrecv(data[DOMAINSIZE-1, 1:-1], south, 2, new_data[0, 1:-1], north, 2)
comm.Sendrecv(np.array(data[1:-1, DOMAINSIZE-1].data,
            dtype=np.int32), east, 4, col_west, west, 4)
comm.Sendrecv(np.array(data[1:-1, 0].data,
            dtype=np.int32), west, 3, col_east, east, 3)

new_data[1:-1, DOMAINSIZE-1] = col_east.data
new_data[1:-1, 0] = col_west.data
```

I added a Barrier and printed the results. I have to admit, doing this in Python was significantly easier than in C++.

2.3. A self-scheduling example: Parallel Mandelbrot [30 Points]

The final task was the most fun for me. I implemented the manager-worker pattern to distribute the computation of the Mandelbrot set among the workers. The manager was responsible for distributing the work and collecting the results, while the workers were responsible for computing the Mandelbrot set for a specific region of the complex plane. The manager was also responsible for plotting the final image. The modifications were done on the code provided in the assignment.

The following code snippet shows the implementation of the worker:

```
def worker(comm):
"""
The worker.

Parameters
----------
comm : mpi4py.MPI communicator
    MPI communicator
"""
while True:
    # receive task
    task = comm.recv(source=MANAGER, tag=TAG_TASK)
    if task is None:
        break
    # do task
    task.do_work()
    # send task done
    comm.send(task, dest=MANAGER, tag=TAG_TASK_DONE)
```

We can see from the code that the worker is in an infinite loop, waiting for a task to be sent by the manager. Once the task is received, the worker computes the Mandelbrot set for the given region and sends the task back to the manager. When it receives a `None` object, the worker breaks the loop and exits.

The following code snippet shows the implementation of the manager:

```
def manager(comm, tasks, TasksDoneByWorker):
"""
The manager.

Parameters
----------
comm : mpi4py.MPI communicator
    MPI communicator
```

```
tasks : list of objects with a do_task() method performing the task
    List of tasks to accomplish
TasksDoneByWorker : dict
    Dictionary of number tasks done by worker

Returns
-------
tasks_done_ : list of objects with _i_start, _nx_local, and _patch
attributes
    List of tasks done
"""
ntasks = len(tasks)
nworkers = comm.Get_size() - 1
ntasks_sent = 0
ntasks_done = 0
task_index = 0
tasks_done_ = []
while ntasks_done < ntasks:
    # send new tasks
    while task_index < ntasks and ntasks_sent < nworkers:
        comm.send(tasks[task_index], dest=ntasks_sent + 1, tag=TAG_TASK)
        ntasks_sent += 1
        task_index += 1

    # receive task done
    status = MPI.Status()
    task = comm.recv(source=MPI.ANY_SOURCE,
                     tag=TAG_TASK_DONE, status=status)
    tasks_done_.append(task)
    TasksDoneByWorker[status.source] += 1
    ntasks_done += 1
    ntasks_sent -= 1

    # send new task
    if task_index < ntasks:
        comm.send(tasks[task_index], dest=status.source, tag=TAG_TASK)
        ntasks_sent += 1
        task_index += 1
    # send done
    else:
        comm.send(None, dest=status.source, tag=TAG_TASK)
return tasks_done_
```

We can see that the maneger has a more complex structure than the worker. It needs to keep track of the tasks that have been sent and the tasks that have been done. It also needs to keep track of the number of tasks done by each worker. The manager sends tasks to the workers and receives the results. Once all the tasks have been sent, the manager sends a `None` object to the workers to signal that there are no more tasks to be done. The manager then waits for the workers to finish their tasks and collects the results. The results are stored in a list and returned to the main function. The main function then plots the results. I also added the TasksDoneByWorker dictionary that keeps track of the number of tasks done by each worker. Since it is a dictionary, it is passed by reference and the changes are reflected in the main function.

The following code snippet shows the changes done on the main function:

```
...
if my_rank == MANAGER:
    tasks_done_ = manager(comm, tasks, TasksDoneByWorker)
else:
    worker(comm)

comm.Barrier()
if my_rank == MANAGER:
    im = M.combine_tasks(tasks_done_)
    plt.imshow(im.T, cmap="gray", extent=[x_min, x_max, y_min, y_max])
    plt.savefig("mandelbrot.png")
...
```

We can see that the manager and worker functions are called depending on the rank of the process. As a sidenote, I didn't find it necessary to use the `TAG_DONE` tag.

Now, we will analyze the strong scaling of the parallel Mandelbrot set computation, for different numbers of processes and tasks. Each configuration was run fifty times and the median was used to generate the plot. The results are shown in the Figure 6. The reference was the case of 2 processes

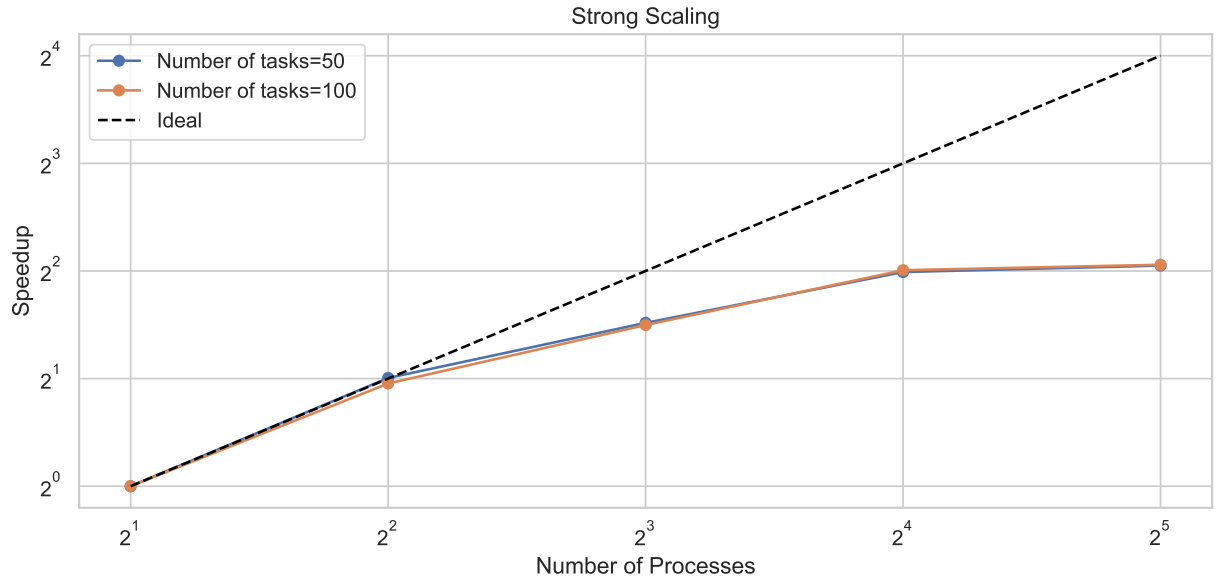

Figure 6: Strong scaling of the parallel Mandelbrot set computation with manager-worker implementation on a 4001×4001 grid with 50 and 100 tasks. Representative data point is the mean of 50 runs.

(1 worker and 1 manager) and 1 task. We can see that the speedup is consistent for all both cases of number of tasks, it is a good result that matches expected behaviour. For a high number of processes, the speedup declines a bit due to the communication overhead, but this is expected. Finally, we plot the bar plot of the number of tasks done by each worker for each configuration. The results are shown in the Figure 7. We can see that the number of tasks done by each worker is distributed evenly for all configurations, which is a good result. This is expected since the tasks are distributed by the manager as soon as they are completed. For the plot, I took the mean of the 50 runs, and that is why they are very even through the workers, since by the nature of the Mandelbrot set, some regions are more computationally expensive than others. We can also see that the behaviour is very similar for 50 and 100 tasks, which is a good indicator of the consistency of the implementation.

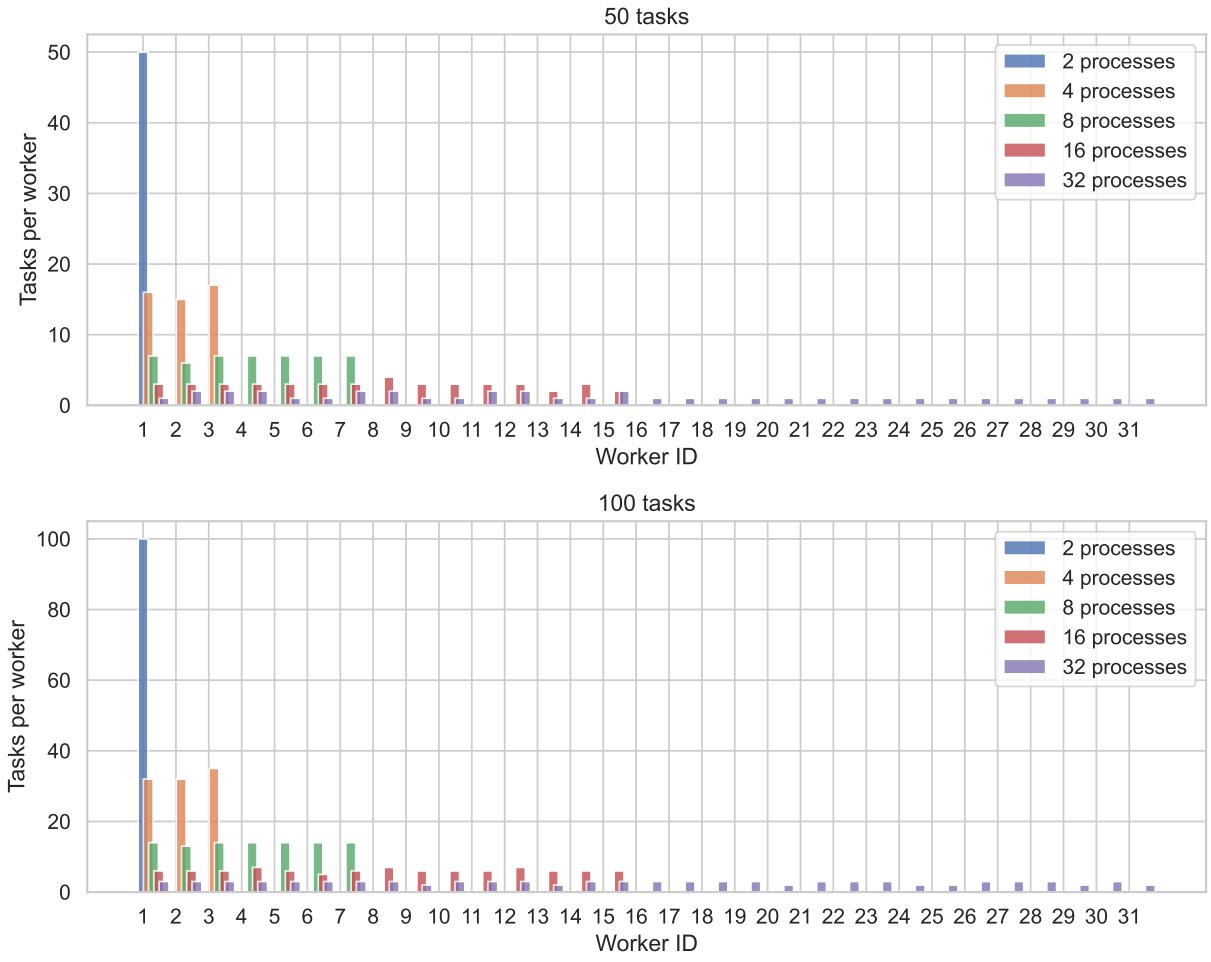

Figure 7: Number of tasks done by each worker for each configuration of the parallel Mandelbrot set computation with manager-worker implementation. Maanager process is ommited from the plot since it doesn't do any tasks.