

# High-Performance Computing 2024

Notes on Python for HPC

# Python for HPC?

Not specifically ***designed*** for HPC, but a language widely used in ***many areas***, including HPC.

### Python vs C++

#### Python

- **interpreted:** The code is interpreted by python.
- **untyped:** Variable types are automatically assigned.
- **dynamically typed:** Variable ↔ value associated at runtime.
- **code style:** code blocks by "indentation"!

#### C++


- **compiled:** The code is compiled by a C++ compiler to machine code.
- **typed:** Variable types need to be assigned.
- **statically typed:** Variable ↔ value associated at compile time.
- **code style:** code blocks by {}, don't care about spacing.

What is `python` ? It's a program that runs python code, i.e., an *interpreter*.

The time invested in doing an experiment is short for Python.

*Python can be very efficient way to attain results, and there are tools to make it applicable in a quasi-HPC environment.*

**TIOBE programming community index**

Nov 2024	Nov 2023	Change	Programming Language	Ratings	Change
1	1		 Python	22.85%	+8.69%
2	3	▲	 C++	10.64%	+0.29%
3	4	▲	 Java	9.60%	+1.26%
4	2	▼	 C	9.01%	-2.76%
5	5		 C#	4.98%	-2.67%
6	6		 JavaScript	3.71%	+0.50%
7	13	▲▲	 Go	2.35%	+1.16%
8	12	▲▲	 Fortran	1.97%	+0.67%
9	8	▼	 Visual Basic	1.95%	-0.15%
10	9	▼	 SQL	1.94%	+0.05%

# Parallelism in Python ?

## Global Interpreter Lock

### Global Interpreter Lock (GIL):

(-) The thread within each process cannot perform task simultaneously!

(+) This is intended to improve performance for single-threaded programs.

```
# Thread library
```

```
import threading:
```

```
# Only one thread can execute Python code at once! Not really parallel.
```

```
# Multiprocessing library
```

```
import multiprocessing
```

```
# Allows to perform tasks simultaneously (using processes instead of  
threads).
```

```
# No support for parallelization over multiple compute nodes
```

# Python For HPC

## NUMBA—JIT compiler for Python

**J**ust-**I**n-**T**ime compiler, compiles code during runtime,  
as opposed to GCC, which is *before* execution.

```
// install  
pip3 install numba
```

```
from numba import jit,prange  
  
@jit  
def jit_monte_carlo_pi(nsamples):  
    sum = 0  
    for i in range(nsamples):  
        sum+=calculation(i)  
    return sum  
  
@jit(parallel=True)  
def prl_monte_carlo_pi(nsamples):  
    sum = 0  
    for i in prange(nsamples):  
        sum+=calculation(i)  
    return sum
```

```
//run  
python3 main.py
```

Numba is a **JIT compiler** implemented as a library  
for Python, enabling significant performance  
enhancements with minimal effort!

See [numba.pydata.org](http://numba.pydata.org) for details.

“Bindings for the MPI standard, allowing Python to exploit multiple processors on workstations, clusters and supercomputers.”

```
// install  
pip3 install mpi4py
```

```
from mpi4py import MPI  
import numpy as np  
  
comm = MPI.COMM_WORLD  
rank = comm.Get_rank()  
  
data=np.zeros(10)  
  
if rank == 0:  
    comm.send(np.ones(10), dest=1, tag=100)  
elif rank == 1:  
    data=comm.recv(source=0, tag=100)
```

```
//run  
mpirun -np 4 python3 main.py
```

**mpi4py** is the MPI standard (a library) in python, and it supports *NumPy* !

See [mpi4py.readthedocs.io](https://mpi4py.readthedocs.io) for details.

Leverage C++ **performance** and keep the **simplicity/functionality** of Python.

```
#include <pybind11/pybind11.h>

double bigcalc(double x){
    Return x+1.0;
};

PYBIND11 MODULE(module, m) {
    m.def("bigcalc", &bigcalc,
        "some_info");
}
```

*pybind11*



```
Import module
c=module.bigcalc(2)
```

See [pybind11.readthedocs.io](https://pybind11.readthedocs.io) for details.



# Project 5

## Parallel Nonlinear PDE using MPI and HPC with Python

**Due date:** 27 November 2024 at 23:59 (See iCorsi for updates)

In this project, we will continue learning about parallel programming with MPI which was introduced in project 4. Particularly, we will use the ghost cells exchange between neighboring processes towards building an MPI parallel solver for Fisher's equation that we discussed and parallelized with OpenMP in project 3. Furthermore, we will extend this project with several tasks related to High-Performance Computing with Python. The Python programming language is very popular in scientific computing because of the benefits it offers for fast code development.

As usual, you find all the skeleton source codes for the project on the course [iCorsi](#) page. We highly recommend to review all the provided skeleton codes before starting any serious implementation of the project tasks.

## 1 Parallel Space Solution of a nonlinear PDE using MPI [60 points]

This sub-project discusses domain decomposition for an MPI parallel solver of a nonlinear PDE that we discussed in detail in project 3. In project 3, we have added OpenMP to the parallel space solution of a nonlinear PDE mini-application, so that we could use all cores on one compute node on the Rosa cluster. The goal of this exercise is now to utilize MPI (Message Passing Interface), enabling the use of multiple compute nodes. Unlike the serial and OpenMP versions, where a single process handles all the data, the MPI version distributes the computational domain across multiple processes (ranks). Each rank handles a sub-domain, allowing for *domain decomposition*. Each process can access only its sub-domain's data and not the data from other processes. For computations using the five-point finite-difference stencil, each process needs data from neighboring grid points. If these points lie on the boundary of a process's sub-domain, the necessary values must be obtained from adjacent processes. Therefore, before each iteration, all MPI processes exchange these boundary values — known as *ghost*, *guard* or *halo cells* — storing them in boundary buffers. This exchange ensures that each process has the necessary data to compute the next iteration.

You can find an initial incomplete version of the MPI code in the directory `mini_app`. The source code is almost equivalent to the serial/OpenMP version that you have already implemented in project 3. There are some comments below and in the code that will guide you through the implementation.

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

Initialize and finalize the MPI environment in `main.cpp`. Replace the welcome message with one that (i) informs the user that the code is using MPI and (ii) indicates the number of processes:

```
[user@icslogin01 ~]$ srun --ntasks=4 --nodes=1 --time=00:05:00
↪ --reservation=hpc-tuesday --pty bash -i
srun: job 13542 queued and waiting for resources
srun: job 13542 has been allocated resources
[user@icsnodeXX mini_app]$ module load openmpi
[user@icsnodeXX mini_app]$ make
[user@icsnodeXX mini_app]$ mpirun ./main 128 100 0.005
=====
Welcome to mini-stencil!
version   :: C++ MPI
processes :: 4
```

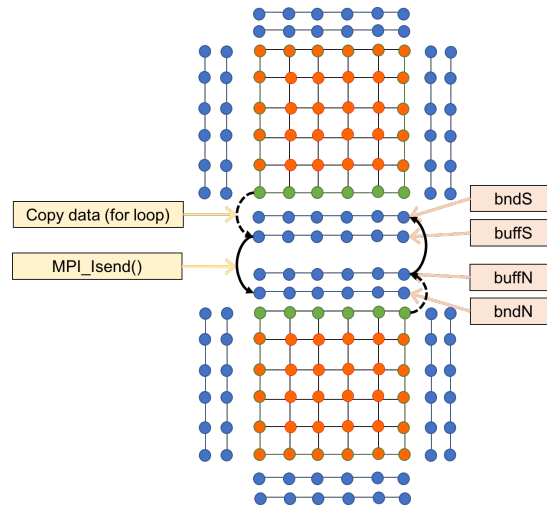


Figure 1: Ghost cell exchange: The bottom process copies the north row (green) into buffer `buffN`, and sends it to its north neighbor (top process). The top process receives the ghost cells from its south neighbor (bottom process) into `bndS`. Likewise, the top process copies the south row (green) into buffer `buffS` and sends it to its south neighbor (bottom process). The bottom process receives the ghost cells from its north neighbor (top process) into `bndN`.

```
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration  :: CG 300, Newton 50, tolerance 1e-06
=====
-----
simulation took 0.0614173 seconds
1513 conjugate gradient iterations, at rate of 24634.8 iters/second
300 newton iterations
-----
### 4, 128, 100, 1513, 300, 0.0614173 ###
Goodbye!
```

For readability, only one process should output this message.

**Note:** You can use the `-reservation=hpc-tuesday` or `-reservation=hpc-wednesday` for better priority.

## 1.2 Domain decomposition [10 Points]

Design a domain decomposition strategy that is compatible with any square grid size and any number of MPI processes<sup>1</sup>. Implement this strategy by storing the decomposition details in the `data::Discretization` and `data::SubDomain` structs. In your report, explain your chosen method of domain decomposition. Discuss why this method was selected and analyze its implications on the performance of the application. Consider aspects such as load balancing, communication overhead, and computational efficiency in your analysis.

**Hint:** Consider the number of data that must be communicated by a generic process with its neighbors. See the short discussion in [1, Sec. 10.4.1].

## 1.3 Linear algebra kernels [5 Points]

Parallelize the relevant linear algebra kernels `hpc_XXX` in `linalg.cpp` using MPI. Please briefly explain in your report which functions you modified and which you did not.

<sup>1</sup>We naturally exclude the practically irrelevant case where the number of MPI processes exceeds the grid size.

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

Implement the ghost cell exchange between neighboring processes. See Fig. 1 for an illustration. Use *non-blocking* point-to-point communication with the objective to overlap computation and communication. Explain your approach in your report.

**Hint:** Copy the corresponding cell data into the *send buffers* `buffN`, `buffS`, `buffE` and `buffW`. Similarly, receive the ghost cell data into the *receive buffers* `bndN`, `bndS`, `bndE` and `bndW`.

## 1.5 Implement parallel I/O [10 Points]

Implement the output of the computed solution with MPI-I/O.

**Hint:** Study the MPI-I/O demo in the provided skeleton codes.

## 1.6 Strong scaling [10 Points]

How does your code scale for different resolutions? Plot the time to solution for  $N_{\text{CPU}} = 1, 2, 4, 8, 16$  processes across resolutions of  $n \times n$ , where  $n = 64, 128, 256, 512, 1024$ . Interpret your results and compare them to the OpenMP implementation of Project 3 in your report.

## 1.7 Weak scaling [10 Points]

How does code scale for constant work by process ratio? Plot the time to solution as the total problem size and the number of processes  $N_{\text{CPU}} = 1, 2, 4, 8, 16$  increase proportionally, to maintain a constant workload per process, and the base resolutions  $n \times n$  and  $n = 64, 128, 256$ . Interpret your results and compare them to the OpenMP implementation of Project 3 in your report.

# 2 Python for High-Performance Computing [25 points]

Python is increasingly used in High-Performance Computing (HPC) projects, serving various roles such as a high-level interface to existing applications and libraries, an embedded interpreter, or even for direct implementation. Its popularity in scientific computing has surged due to its flexibility and ease of use. Users now commonly use Python not only to prototype codes at small scales but also to develop parallel production codes. This shift is partly replacing traditional compiled HPC languages such as C/C++ and Fortran for certain applications. However, when adopting Python for such purposes, it is crucial to monitor performance to meet the rigorous demands of HPC environments. Similar bindings exist for other popular languages such as [Julia](https://julialang.org/)<sup>2</sup> (see [MPI.jl](https://juliaparallel.org/MPI.jl/stable/)<sup>3</sup>).

We highly recommend to (partly) watch the course [High-Performance Computing with Python](#)<sup>4</sup> held July 02–04, 2019 at CSCS. In particular, we will use the package MPI for Python (`mpi4py`) for using MPI within Python. To get started, please watch the [Introduction to MPI](#)<sup>5</sup> lesson of the CSCS course. Although the lessons use mostly IPython/Jupyter notebooks, we will use plain Python scripts.

**ICS Cluster environment setup instructions:** In order to run `mpi4py` on the ICS Cluster you will need to install a custom environment using `anaconda` that has all of the libraries that you will need to run the code. To set up this environment you will need the text file `project5_conda_env.txt` that is located in the `hpc-python` directory.

To set up the environment navigate to the directory containing the `project5_conda_env.txt` file and run the following commands:

```
[user@icslogin01 hpc_python]$ source /apps/miniconda3/bin/activate
[user@icslogin01 hpc_python]$ conda create --name project5_env --file
→ project5_conda_env.txt
[user@icslogin01 hpc_python]$ conda init bash
[user@icslogin01 hpc_python]$ exit
[uname@personal_computer]$ ssh rosa
(base) [user@icslogin01 hpc_python]$ conda activate project5_env
```

<sup>2</sup><https://julialang.org/>

<sup>3</sup><https://juliaparallel.org/MPI.jl/stable/>

<sup>4</sup>[https://www.youtube.com/watch?v=JYX4TQ\\_fCqY&list=PL1tk5lGm7zvQ-EzsiTZ6Xv1SxZs74epzg](https://www.youtube.com/watch?v=JYX4TQ_fCqY&list=PL1tk5lGm7zvQ-EzsiTZ6Xv1SxZs74epzg)

<sup>5</sup><https://www.youtube.com/watch?v=XeyspDaKjMM>

**Please note** that if you have the OpenMPI module loaded in your current session on the cluster you will have problems running the code. Please start a new session for running Python MPI code and do not load the OpenMPI module.

For Python, we refer to the documentation

- <https://docs.python.org/3/>

The documentation for `mpi4py` can be found here

- <https://mpi4py.readthedocs.io/en/stable/index.html>

Remember to use the help function within a Python interpreter:

```
>>> from mpi4py import MPI
>>> help(MPI)
```

In order to get started, we begin with a simple Python MPI program `hello.py`:

```
from mpi4py import MPI

# get comm, size, rank & host name
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
proc = MPI.Get_processor_name()

# hello
print(f"Hello world from processor {proc}, rank {rank} out of {size} processes")
```

Run the script as follows:

```
(base) [user@icslogin01 hpc_python]$ salloc --nodes=4 --ntasks=8 --ntasks-per-node=2
(base) [user@icslogin01 hpc_python]$ conda activate project5_env
(project5_env) [user@icslogin01 hpc_python]$ mpiexec -n 8 python hello.py
Hello world from processor icsnode33, rank 0 out of 8 processes
Hello world from processor icsnode33, rank 1 out of 8 processes
Hello world from processor icsnode34, rank 2 out of 8 processes
Hello world from processor icsnode34, rank 3 out of 8 processes
Hello world from processor icsnode36, rank 7 out of 8 processes
Hello world from processor icsnode35, rank 4 out of 8 processes
Hello world from processor icsnode36, rank 6 out of 8 processes
Hello world from processor icsnode35, rank 5 out of 8 processes
```

Now that everything is set up and working, we can get started!

**Note:** If the warning about `-ntasks-per-node` appears but the tasks are distributed correctly across nodes, it can be safely ignored. This warning does not impact the performance or execution of the job.

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

With MPI for Python's collective communication methods, write a script that computes the sum of all ranks:

- using the pickle-based communication of generic Python objects, i.e. the *all-lowercase* methods;
- using the fast, near C-speed, direct array data communication of buffer-provider objects, i.e. the method names starting with an *uppercase* letter.

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

Write a script that creates a two-dimensional, periodic Cartesian topology and implements ghost cell exchange (as in Project 4):

- use the method `MPI.Compute_dims`, a convenience function similar to MPI's `MPI_Dims_create`;

- create a Cartesian topology using MPI for Python;
- determine the neighboring processes;
- output the topology: rank, Cartesian coordinates in decomposition, East/West/North/South neighbors;
- for each process, exchange its rank with the four east/west/north/south neighbors.

Verify that you obtain the expected result.

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

In this task, you are asked to implement one of the most common parallel patterns: the *manager-worker* pattern. The basic idea is that one process, known as the manager, is responsible for delegating work to other processes, known as the workers. This is particularly useful in problems where the amount of work per worker is difficult to estimate and the workers don't have to communicate with each other in order to do their work. As a particular example, we again consider the Mandelbrot set. Note that this is only meant as an illustration of this fundamental type of parallel algorithm, and not really as the best way to parallelize the computation of the Mandelbrot set.

The manager decomposes the Mandelbrot set into a number of (rectangular) patches. Computing the Mandelbrot (sub)set on a particular patch will be called a task. The manager then delegates these tasks to the workers. Once a worker is done computing a particular task, he sends the patch back to the manager. Therewith, the worker signals to the manager that he is available to work on a new task. The manager then sends the worker another task to work on. This process is repeated until no more tasks remain, i.e. all the patches of the Mandelbrot set have been computed. Finally, the manager combines all the patches from the workers and outputs the Mandelbrot set.

The skeleton codes for this sub-project are located in the folder `hpc_python/ManagerWorker` available through the course iCorsi page. Begin by familiarizing yourself with the `mandelbrot_task.py` module. It contains two classes. First, the class `mandelbrot`, which decomposes the Mandelbrot set computation in a series of subsets or patches, produces a list of tasks, and combines the tasks' patches together. Second, the `mandelbrot_patch` class, which holds a subset or patch of the Mandelbrot set and contains a method `do_work` that performs the actual computation. This part is already fully implemented for your convenience. However, feel free to try out different implementations, e.g., domain decompositions, etc.

Complete the following:

- Implement the manager-worker algorithm in the skeleton code `manager_worker.py`.
- Add a scaling study using 2-16 workers for a 4001x4001 domain and split the workload into 50 and 100 tasks.

The program can be called as follows:

```
(project5-env) [user@icslogin01 ManagerWorker]$ mpiexec -n 4 python manager_worker.py
→ 4001 4001 100
```

## 3 Quality of the Report [15 Points]

Each project will have 100 points (out of 15 point will be given to the general written quality of the 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.
  - Build files and scripts. If you have modified the provided build files or scripts, make sure they still build the sources and run correctly. We will use them to grade your submission.
  - `project_number_lastname_firstname.pdf`, your write-up with your name.
  - Follow the provided guidelines for the report.
- Submit your `.tgz` through iCorsi.

## Code of Conduct and Policy

- Do not use or otherwise access any on-line source or service other than the iCorsi system for your submission. In particular, you may not consult sites such as GitHub Co-Pilot or ChatGPT.
- You must acknowledge any code you obtain from any source, including examples in the documentation or course material. Use code comments to acknowledge sources.
- Your code must compile with a standard-configuration C/C++ compiler.

Please follow these instructions and naming conventions. Failure to comply results in additional work for the TAs, which makes the TAs sad...

## References

- [1] Georg Hager and Gerhard Wellein. Introduction to high performance computing for scientists and engineers. *Chapman & Hall/CRC Computational Science*, July 2010. URL: <http://dx.doi.org/10.1201/EBK1439811924>, doi:10.1201/ebk1439811924.

## Solution for Project 5

---

# 1. Parallel Space Solution of a nonlinear PDE using MPI

## 1.1. Domain decomposition

The domain was decomposed using the non-periodic 2D Cartesian domain decomposition strategy, where the grid is divided into smaller rectangular subgrids, which are then assigned to a MPI processes. This method is suited for problems where you have grid and update the values on said grid with a stencil. Furthermore in our case the work is evenly distributed over the grid, which makes this method ideal, because it is simple and results in a good work balancing. The topology created by the 2D Cartesian domain, also leads to efficient exchanging of neighboring ghost cells which reduces the communication overhead significantly. Another plus point is that it is very scalable and can be easily used with varying number of processes.

## 1.2. Linear algebra kernels

The primary rational used in order to decide which functions needed to be updated with MPI code was if the function needed to compute a result from distributed data across the processes or if the computation are done independently on each process. The two functions that modified were:

- `hpc_dot`: Added a `MPI_Allreduce` in order to compute the global sum of the inner local product values.
- `hpc_norm2`: Similar to the dot product the same MPI function was used to computed the global squared norm.

All the other functions can be used locally on the subdomain without needing to share the result, therefore no modification was needed.

## 1.3. The diffusion stencil

The non-blocking point-to-point MPI communication was used to exchange the ghost cells between processes. The non-blocking approach was used to overlap communication and therefore improve performance. As the initial step the ghost buffers are filled with the boundary values from the local domain, corresponding to the direction they are sent. Furthermore the buffers for the receiving ghost cells are initialized. Next the ghost are exchange using a non-blocking communication using `MPI_Isend` and `MPI_Irecv`, the order in which the send and received are executed does not matter due to the communication being non-blocking, but it is important that there is for each send there is a corresponding receive command. I decided to group them by direction as seen in Listing 1.



```

1  // Send/recv east
2  MPI_Isend(buffE.data(), buffE.xdim(), MPI_DOUBLE, domain.neighbour_east, 0,
3           domain.comm_cart, &send_req[0]);
4  MPI_Irecv(bndE.data(), bndE.xdim(), MPI_DOUBLE, domain.neighbour_east, 1,
5           domain.comm_cart, &recv_req[1]);
6  // Send/recv west
7  MPI_Isend(buffW.data(), buffW.xdim(), MPI_DOUBLE, domain.neighbour_west, 1,
8           domain.comm_cart, &send_req[1]);
9  MPI_Irecv(bndW.data(), bndW.xdim(), MPI_DOUBLE, domain.neighbour_west, 0,
10          domain.comm_cart, &recv_req[0]);
11
12 // Send/recv north
13 MPI_Isend(buffN.data(), buffN.xdim(), MPI_DOUBLE, domain.neighbour_north, 2,
14          domain.comm_cart, &send_req[2]);
15 MPI_Irecv(bndN.data(), bndN.xdim(), MPI_DOUBLE, domain.neighbour_north, 3,
16          domain.comm_cart, &recv_req[3]);
17
18 // Send/recv south
19 MPI_Isend(buffS.data(), buffS.xdim(), MPI_DOUBLE, domain.neighbour_south, 3,
20          domain.comm_cart, &send_req[3]);
21 MPI_Irecv(bndS.data(), bndS.xdim(), MPI_DOUBLE, domain.neighbour_south, 2,
22          domain.comm_cart, &recv_req[2]);
23
24 // the interior grid points
25 for (int j = 1; j < jend; j++) {
26     for (int i = 1; i < iend; i++) {
27         f(i, j) = -(4. + alpha) * s_new(i, j)           // central point
28                 + s_new(i - 1, j) + s_new(i + 1, j)    // east and west
29                 + s_new(i, j - 1) + s_new(i, j + 1)    // north and south
30                 + alpha * s_old(i, j) +
31                 beta * s_new(i, j) * (1.0 - s_new(i, j));
32     }
33 }
34 MPI_Waitall(4, recv_req, MPI_STATUSES_IGNORE);

```

Listing 1: Non-blocking send and receive pattern

The main reason why we use non-blocking communication is to overlap communication with computation, hence it is of upmost importance where the wait clause is placed. Because for computing the inner grid points the ghost cells are not necessary, we only need to ensure that all messages were received after the inner grid points calculation and before the boundary grid points are computed. In Listing 1 the `MPI_Waitall` for all the receiving requests is placed after the interior grid points for computed.

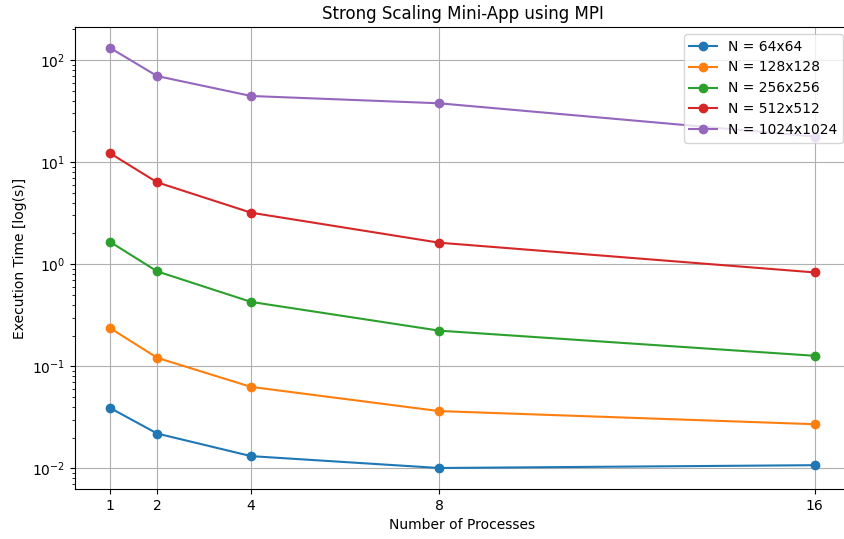
## 1.4. Strong scaling

The graphs seen in illustration 1 show the strong scaling behavior of the mini application using MPI 1a and OpenMP 1b for its implementation. For both implementation were run on a single node on the Rosa cluster. Analyzing the MPI implementation we can clearly see that the execution time decreases for all grid sizes. Larger grid sizes benefit more from the parallelization, due to the computation outweighing the communication overhead. For smaller grid sizes on the other hand the reduction of execution flattens out when the communication overhead becomes large in relation to computation.

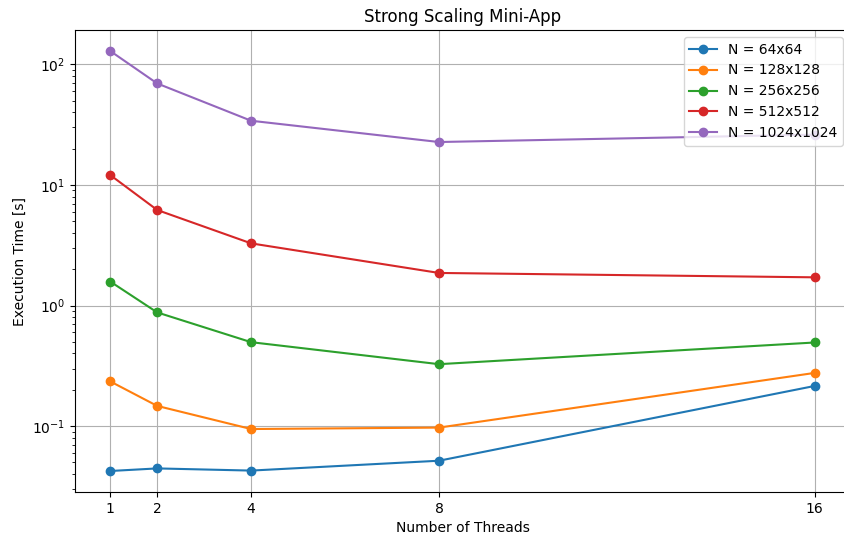
Comparing this result to the results from Project 3, we can observe that MPI achieves better across all resolutions, especially for larger grids size due to its distributed memory model. OpenMP on the other hand struggles when the thread size increases, especially for smaller grids. Furthermore, by enabling overlapping communication with computation, the MPI version reduces idle time and improves scaling, compared to the OpenMP, where the memory is shared and this lead to potential bottlenecks, especially for larger thread count. In conclusion for small and medium grid sizes the OpenMP is more suited due to it performing very similar to the MPI version and it is a lot easier and quicker to implement. For large grid sizes on the other hand it is advised to take the time and



implement the MPI version.



(a) MPI



(b) OpenMP

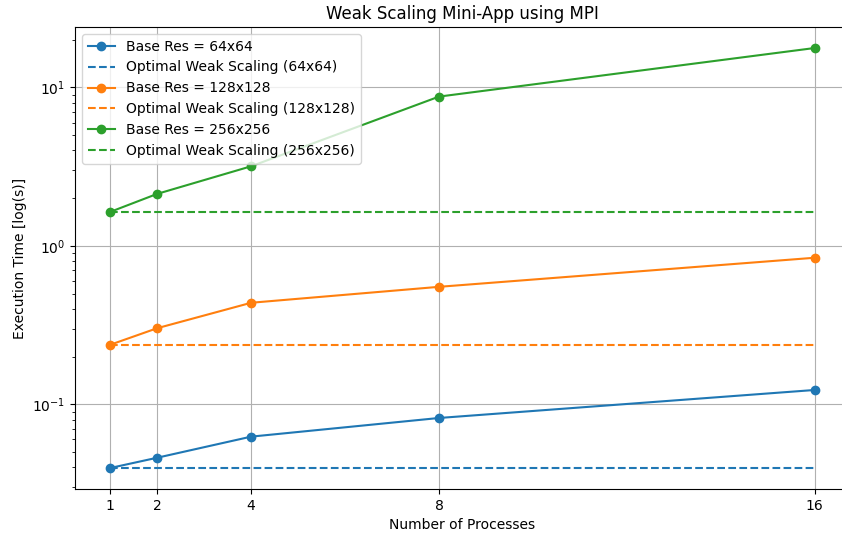
Figure 1: Strong Scaling

### 1.5. Weak scaling

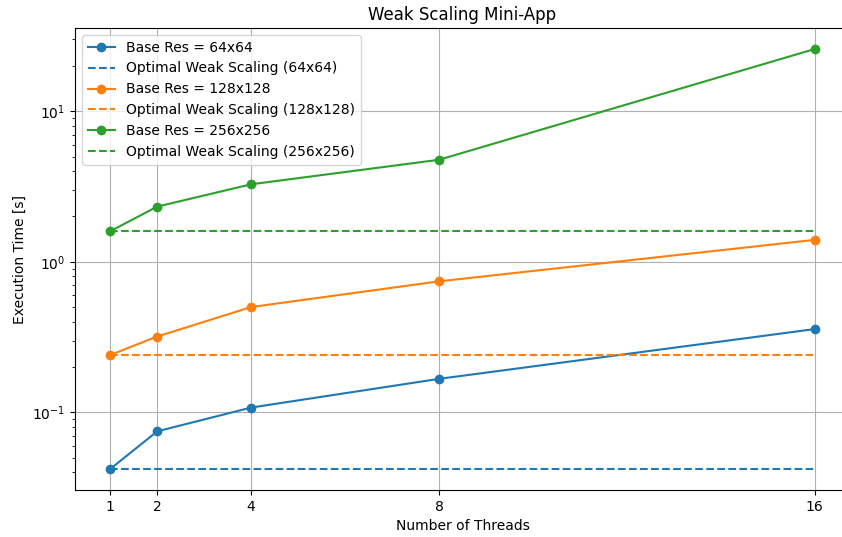
The weak scaling of the mini-app for the MPI (2a) implementation, the efficiency decreases by a lot when increasing the base resolution, which is increased by multiplying the base resolution by the square root of the number processes. Larger base resolutions have stronger increase of execution time compared to lower base resolution, this indicates inefficiencies as the problem size increases per process.

Comparing it to the result obtained in Project 3 for the OpenMP implementation, which can be seen in Subfigure 2b. In terms of scaling efficiency the MPI outperforms the OpenMP version especially for larger problem sizes with higher number of processes. This shows that the MPI implementation handles large grid size more effectively, while the OpenMP's performance degrades. One of the factors attested to that is the higher overhead usually encountered in application that use OpenMP. Even though the mini-app is rather inefficient, we can clearly see that the MPI version is closer

to the optimal scaling indicated by the dashed lines. Overall the execution times are consistently lower for the MPI implementation. In conclusion MPI is the preferred choice especially if you have a larger workload, but it also comes with the trade off of being more time intensive to implement.



(a) MPI



(b) OpenMP

Figure 2: Weak Scaling

## 2. Python in High-Performance Computing

### 2.1. Sum of ranks

The sum of ranks code performs a ring-based communication among the processes using MPI. In my implementation each process sends or receives data from or to its neighbors based on its rank. The process are divided in two disjointed groups, based on if their rank number is even or odd. This two groups are then alternating between sending and receiving. Each process has a local sum variable that sums up all the values and sends the last received rank to its neighbor with the `dest=(rank + 1) % size`. After the number of rounds equal to the number of processes, each process sum variable holds the sum of all ranks. Because this version of sum of ranks was implemented with

mpi4py, there are two possible ways to implement this, one using the pickle-based communication of generic Python objects and the other the near C-speed direct array communication. These are the differences in implementation between the two:

- **Pickle-based (lowercase):** In this version you can send generic python object making the implementation straight forward. One can just put the generic python object and the function call and then save the return value of the receive function in an other variable.
- **Direct-array (uppercase):** In this version the communication is done by using a buffer-like object, where the arguments need to be specified explicitly by using a tuple for example [data, MPI.DOUBLE] and also returns a tuple by using references and not a specific return statement.

## 2.2. Ghost cells

This program is an reimplementaion of the ghost cells in Project 4 in mpi4py. We construct a cartesian grid dynamically using the `MPI.Compute_dims` function. Each process sends its ghost cells to its neighbours (north, east, south, west). The communication is done using the `(Sendrecv)`, where the ghost cells are send in one direction and at the same time received from the opposite one.

```

1 # S:left/R:right
2 recv_temp = np.empty(SUBDOMAIN, dtype=np.float64)
3 domain_send = DOMAINSIZ + 1
4 domain_recv = DOMAINSIZ + (SUBDOMAIN + 1)
5 comm_cart.Sendrecv(
6     [data[domain_send:], 1, T_col],
7     dest=rank_left,
8     sendtag=0,
9     recvbuf=[recv_temp, MPI.DOUBLE],
10    source=rank_right,
11    recvtag=0,
12 )
13
14 data[domain_recv : domain_recv + SUBDOMAIN * DOMAINSIZ : DOMAINSIZ] = recv_temp

```

Listing 2: Ghost cell Sendrecv example

In Listing 2 an example `Sendrecv` is shown, where we assign a temporary variable for the incoming buffer, set the starting point of the domain of the continuously stored data for both the receiving and sending ghost cells. Subsequently the data is send and received using a costume type named `MPI.Datype T_col`. Finally we put the received data into the local matrix.

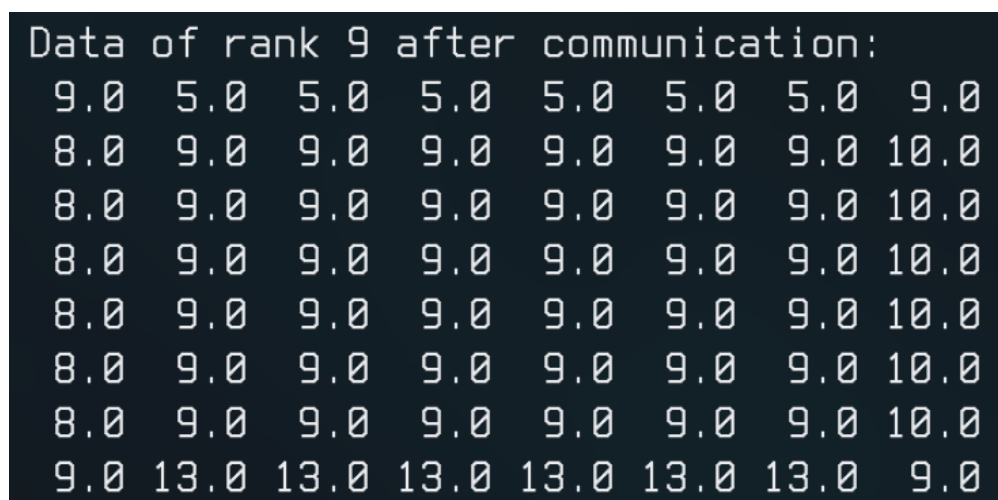


Figure 3: Output for ghost cells exchange for rank 9 with 16 processes

### 2.3. A self-scheduling Parallel Mandelbrot

The manager-worker paradigm handles the workload by dynamically efficiently distributing it among the process. This is especially useful when we deal with irregular workloads, such as the Mandelbrot set. My implementation uses a queue where all the subbatches are stored and as soon a process is done with the subbatch the next one in the queue is assigned to the worker, which in our case is just a process. This ensures efficient load is dynamically redistributed depending on availability of the worker. This comes at a cost of having higher communication overhead due to having more data exchanges and the manager can potentially act as a bottleneck.

For our scaling analysis the Mandelbrot domain is fixed and of sizes  $4001 \times 4001$ . We split the workload into 50 and 100 tasks respectively. In Figure 4 we can observe that execution time decreases significantly when the number of workers is increased. Beginning from 8 workers onwards we can see that the efficiency starts to decrease to do increased communication overhead, compared to workload per worker. Looking at the different number of tasks, the blue line indicating a 100 tasks scales slightly better in this case the granularity results in a better load balancing which is counter acted by the previously mentioned costs. This result shows that the load balancing and the cost pretty much offset and there is not a significant difference between the number of tasks in the case of the mandelbrot set.

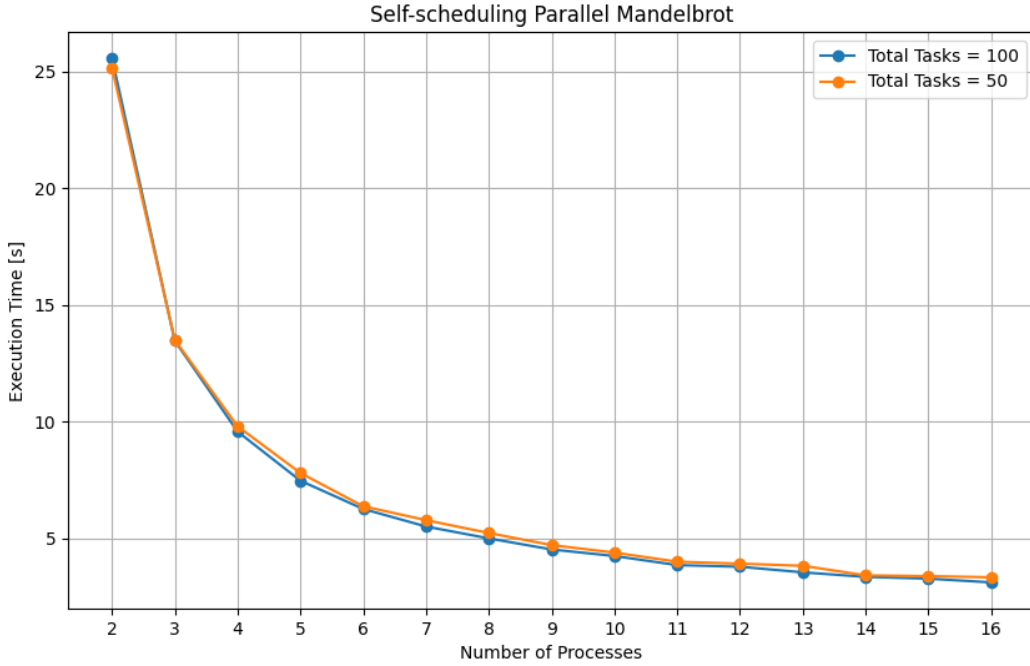


Figure 4: Manager Worker scheme scaling with different number of tasks