Università della Svizzera italiana

Institute of Computing
CI

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

Student: INNOCENT KISOKA Discussed with: -

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 both executable package and 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.

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

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

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

1.1.1. MPI Initialization and Finalization

The code initializes the MPI environment and obtains critical process information as shown in the code 1:

- `MPI_Init`: Initializes the MPI execution environment.
- `MPI_Comm_size`: Retrieves the total number of processes.
- `MPI_Comm_rank`: Determines the rank of the current process.

These steps are essential for setting up parallel execution. The code ensures proper cleanup by using:

- `MPI_Comm_free`: Frees communicator resources.
- `MPI_Finalize`: Terminates the MPI environment.

It is shown in code 2

```
// TODO initialize
// use "MPI_Comm_size", "MPI_Comm_rank" and "MPI_Init_thread"
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
```

Figure 1: Code Snippet for MPI Initialization

1.1.2. Modified Welcome Message

```
// TODO finalize it using "MPI_Finalize" and "MPI_Comm_free"
MPI_Comm_free(&domain.comm_cart);
MPI_Finalize();
```

Figure 2: Code Snippet for MPI Finalization

To provide a user-friendly interface, a welcome message is displayed when the simulation starts as shown in code 3 and it's output in 4. The message includes:

- The program's name and version.
- The number of MPI processes being used.
- Mesh details, such as grid dimensions and spatial resolution.
- Time-stepping parameters, including the number of iterations and tolerance.

Key Features

- **One Process Output**: The message is displayed only by the root process (rank 0) to prevent clutter.
- **Formatted Output**: The message is structured for readability using separators (=).

```
//// TODO: Modify welcome message

if( domain.rank == 0 ) {
    std::cout << std::string(80, '=') << std::endl;
    std::cout << "Welcome to mini-stencil!" << std::endl;
    std::cout << "version   :: C++ MPI  "<<std::endl;
    std:: cout<< "Processes :: " <<domain.size << std::endl;
    std::cout << "mesh      :: " << options.nx << " * "
                                   << options.ny << " dx = " << options.dx << std::endl;
    std::cout << "time      :: " << nt << " time steps from 0 .. "
                                       << options.nt*options.dt << std::endl;;
    std::cout << "iteration :: " << "CG "          << max_cg_iters
                                 << ", Newton "    << max_newton_iters
                                 << ", tolerance " << tolerance << std::endl;;
     std::cout << std::string(80, '=') << std::endl;
}
```

Figure 3: Modified Message

Figure 4: Modified Message Output

1.2. Task 2 - Create a Cartesian topology/Domain decomposition [10 Points]

A robust domain decomposition strategy was implemented, compatible with any square grid size and number of processes.

Key Features

Domain decomposition:

- Utilizes a 2D Cartesian decomposition for dividing the grid into subdomains distributed among MPI processes.
- `MPI_Dims_create` ensures optimal and balanced subdomain dimensions.

Topology and Neighbor Identification:

- A non-periodic Cartesian topology is created with `MPI_Cart_create`.
- Processes identify neighbors (North, South, East, West) using `MPI_Cart_shift`.

Grid Boundaries:

- Subdomains compute local grid dimensions, adjusting boundaries for edge subdomains to handle irregular grid sizes.

Implementation Details

- **Initialization:** Calculates subdomain locations and neighbors, storing results in `data::Discretization` and `data::SubDomain`.
- **Bounding Box:** Computes local grid dimensions and indices, adjusting edge subdomains to ensure full grid coverage.
- **Output:** Each process prints its rank, neighbors, and local grid details.
- **Communication Topology:** Neighbor data is stored for efficient halo exchanges, enforcing non-periodic boundaries.

Findings

- `MPI_Dims_create` ensures even distribution of grid points, minimizing idle processes.
- Efficient patterns via `MPI_Cart_shift` limit communication to immediate neighbors, reducing overhead.
- The 2D Cartesian strategy supports strong and weak scaling as grid size and process count change.
- Independent subdomain computations reduce synchronization delays while ensuring no grid points are missed.

Performance Analysis

- **Strengths:**
  - Adaptable to any grid size or process count.
  - Balances load efficiently and simplifies communication for halo exchanges.
- **Challenges:**
  - Increased communication complexity with large grids and high process counts.
  - Slight unevenness in subdomain sizes for prime process counts.

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

Parallelization Strategy

The parallelization process focused on incorporating MPI for distributed computing and SIMD (Single Instruction Multiple Data) techniques for intra-node performance optimization[1]. The key modifications included:

- Use of `MPI_Allreduce` for collective communication to compute global results across all processes.
- Application of loop unrolling to enhance vectorized computations in key kernels.
- Incorporation of efficient memory access patterns to minimize data transfer overheads.

Modified Functions

1. `hpc_dot()`

Purpose: Computes the inner product of two vectors.
Modification:

- Integrated `MPI_Allreduce` to accumulate local dot products across processes into a global result.
- Applied loop unrolling to process four elements per iteration for better CPU cache utilization and vectorization.
- Resultant global dot product is returned to ensure correctness in a distributed setup.

2. `hpc_norm2()`

Purpose: Computes the 2-norm of a vector.
Modification:

- Similar to `hpc_dot`, this function now uses `MPI_Allreduce` to compute the global norm across all processes.
- Loop unrolling optimizes local computation.
- The global result is computed as the square root of the sum of squared elements.

Unmodified Functions

Certain functions, such as `hpc_fill`, `hpc_copy`, and `hpc_scale`, were not parallelized because:

- They operate independently on local data.
- Parallelization would yield negligible performance gains due to their simplicity and lack of inter-process dependency.

Performance Considerations

- **Load Balancing:** The uniform distribution of data ensures balanced computation across MPI processes.
- **Communication Overhead:** Minimizing MPI communication in performance-critical regions (e.g., within the CG loop) helps maintain scalability.
- **Scalability:** The implementation is designed to scale with the number of processes by ensuring that reductions (`MPI_Allreduce`) operate efficiently on small datasets.

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

I implementated ghost cell exchange using the following steps:

1. Copy the corresponding grid cell data into send buffers (`buffN`, `buffS`, `buffE`, `buffW`).
2. Use `MPI_Isend` to send data to neighboring processes and `MPI_Irecv` to receive ghost cell data into receive buffers (`bndN`, `bndS`, `bndE`, `bndW`).
3. Perform interior grid point computations while communication is ongoing to overlap computation and communication.
4. Use `MPI_Waitall` to ensure all communications are completed before processing boundary points.

The implementation leverages non-blocking communication (`MPI_Isend` and `MPI_Irecv`) to overlap communication with computation. The use of `MPI_Waitall` ensures data consistency by guaranteeing the completion of all communications before proceeding with boundary point computations. This strategy minimizes idle time for processes waiting on communication, thereby improving overall performance[2].

Key Features

- Data is copied into buffers to ensure the consistency of the send/receive operations.
- Non-blocking communication allows interior grid points to be computed while waiting for ghost cell data.
- Communication requests are tracked using `MPI_Request` arrays to handle multiple directions simultaneously.

Performance Considerations

- **Load Balancing:** The uniform grid decomposition ensures balanced computational workloads across processes.
- **Communication Overhead:** The use of non-blocking communication minimizes idle times.
- **Scalability:** The design supports scalability by overlapping communication with computation and minimizing the synchronization overhead.

1.1. Task 5 -Implement parallel I/O [10 Points]

The implementation utilizes MPI-I/O for parallel file output, ensuring that each process writes its data to the correct location in a shared file and the output is shown in figure 5. Key steps include:

- **MPI Initialization**: Processes were initialized and a Cartesian communicator was created for domain decomposition.
- **Domain Decomposition**: Each process handled a subset of the global 2D grid, defined by its subdomain.
- **Defining MPI Subarray**: Each process's local grid was mapped to the global grid using `MPI_Type_create_subarray`, specifying:
  - Global dimensions.
  - Local dimensions.

– Starting indices.

- **Setting File View**: File views were set with `MPI_File_set_view` for correct data placement.
- **Writing Data**: Processes wrote data concurrently using `MPI_File_write_all`.
- **Closing Resources**: File handles and custom MPI types were freed after writing.

Results:

- **Binary Output**: `output.bin` contains the computed solution, written in parallel.
- **Metadata Output**: `output.bov` provides visualization metadata, written by the master process.

Performance:

- The I/O time was measured, and the implementation supports scalable, efficient data writing for large, distributed grids.

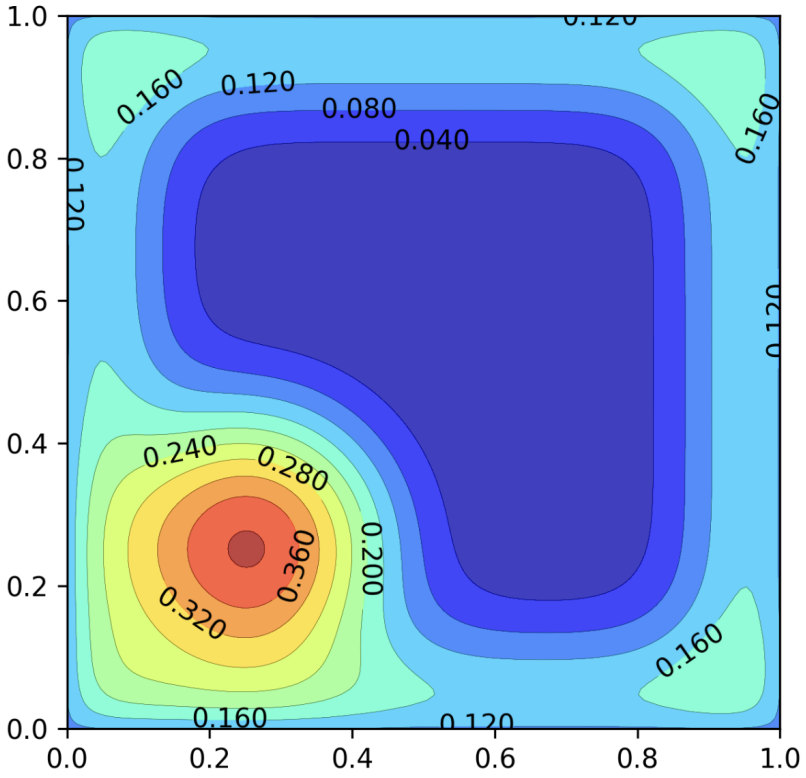

Figure 5: Results of the Stencil Computation

1.6 Strong scaling [10 Points]

Definition

Strong scaling measures how the execution time of a fixed problem size decreases as the number of CPUs (NCPU) increases. Ideally, the execution time should decrease linearly as more CPUs are added. The time recorded can be seen in table 1 together with the strong scaling plot in figure 6

Grid Size ($n \times n$)	**Time (s)** N=1	**Time (s)** N=2	**Time (s)** N=4	**Time (s)** N=8	**Time (s)** N=16
64×64	0.0398729	0.0229523	0.0147691	0.012604	0.0153703
128×128	0.259799	0.130442	0.0686735	0.0404601	0.0315571
256×256	1.71422	0.878382	0.454705	0.230752	0.134358
512×512	13.1958	6.31247	3.25356	1.67867	0.84599
1024×1024	153.788	83.1634	54.1292	42.5365	21.9804

Table 1: Performance times for different grid sizes and number of CPUs.

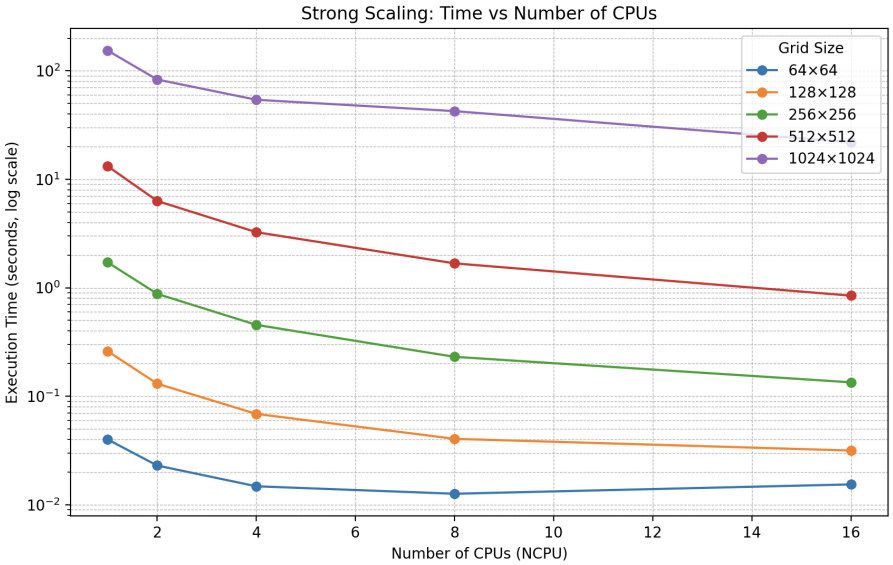

Figure 6: Strong Scaling Performance Plot

Observations and Interpretation

Smaller Grid Sizes (64×64, 128×128)

- For smaller grids, the execution time decreases initially but flattens or even increases slightly as the number of CPUs rises (e.g., for 64×64, the time for NCPU = 16 is higher than NCPU = 8).
- **Reason:** Parallelization overhead (e.g., communication and synchronization) dominates the actual computation time for small grids. This makes it inefficient to use a large number of CPUs for such small problems.

Medium Grid Sizes (256×256, 512×512)

- Medium-sized grids exhibit better scaling, with execution time decreasing consistently as more CPUs are used.
- However, for NCPU = 16, the improvement diminishes compared to earlier CPU increments.
- **Reason:** The problem becomes communication-bound as the ratio of computation per CPU decreases.

Larger Grid Sizes (1024×1024)

- Larger grids show the best scaling behavior, with significant reductions in execution time as CPUs increase.

- **Reason:** Larger grids involve more computation, which outweighs the parallelization overhead. This aligns with Amdahl's Law, where scaling is more efficient for problems with a high computation-to-communication ratio.

Performance Bottlenecks

- **Communication Overhead:** At higher CPU counts, communication and synchronization among processes increase. This is evident in the smaller grids where the performance improvement diminishes or even reverses.
- **Load Balancing:** For smaller grids, the work per CPU becomes minimal, leading to inefficiencies in utilizing all CPUs effectively.

Key Comparisons with Project 3

A graph 7 for strong scaling from project 3 was used for comparison purposes.

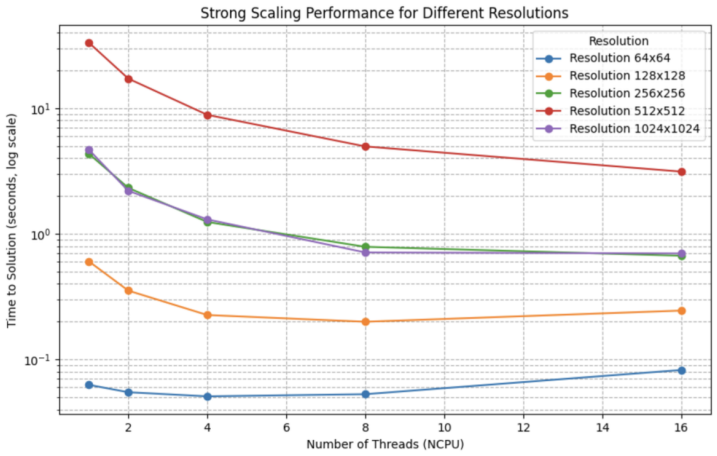

Figure 7: Project 3 Strong Scaling Performance

Performance for Smaller Grids

OpenMP (Project 3):

- The time-to-solution curve for smaller grid sizes (e.g., 64×64 and 128×128) shows better scaling with increased threads.
- OpenMP exhibits efficient performance for smaller grids due to its shared-memory architecture, which avoids the communication overhead of MPI.

MPI (Current Project):

- MPI struggles with smaller grids, as seen in the 64×64 case where execution time increases for higher NCPU (e.g., at 16 CPUs).
- This is likely due to communication overhead outweighing the benefits of parallel computation.

Performance for Larger Grids

OpenMP:

- OpenMP continues to scale for larger grids (e.g., 512×512 and 1024×1024), but the improvement flattens as the number of threads increases.

- This is due to shared memory contention and limitations of the memory bandwidth in multi-threaded environments.

MPI:

- MPI demonstrates superior performance for larger grids, with better scalability and reduced execution time as CPUs increase.
- Its distributed-memory model allows handling larger workloads more effectively, with reduced contention compared to OpenMP.

Scaling Efficiency

OpenMP:

- The scaling efficiency diminishes as the number of threads increases, especially for smaller and medium grid sizes.
- This is due to the overhead associated with thread synchronization and shared memory contention.

MPI:

- MPI maintains better scaling efficiency for larger grid sizes, though it also suffers from diminishing returns at higher CPU counts due to communication overhead.

Time-to-Solution Trends

- For larger grids (1024×1024), the time-to-solution with MPI decreases significantly compared to OpenMP, showcasing its suitability for distributed workloads.
- MPI's performance gain becomes more evident as grid size increases.

Suitability for Grid Sizes

- **OpenMP:** Better suited for smaller grids and cases where communication overhead would dominate the computational cost.
- **MPI:** Better suited for larger grids, where distributed memory systems can reduce the computational burden on individual processors.

1.6 Weak scaling [10 Points]

Weak scaling refers to how the performance of the code changes as both the problem size and the number of processes increase proportionally, ensuring that each process gets the same amount of work.

In my scaling tests, I've increased both the global problem size (n_{global}) and the number of processes (N_{CPU}) in a proportional manner while keeping the workload per process constant. The time recorded is shown in table2 and the plot is shown in figure8

Base n($n_{global} = n_{base} \times \sqrt{NCPU}$)

(n_{base})	(NCPU)	(n_{global})	Time (s)
64	1	64x64	0.0392514
	2	90x90	0.0498204
	4	128x128	0.0676643
	8	181x181	0.0885274
	16	256x256	0.134214
128	1	128x128	0.262674
	2	181x181	0.331769
	4	256x256	0.452449
	8	362x362	0.613271
	16	512x512	0.846193
256	1	256x256	1.71919
	2	362x362	2.32069
	4	512x512	3.25216
	8	724x724	10.9703
	16	1024x1024	21.7381

Table 2: Weak Scaling Results

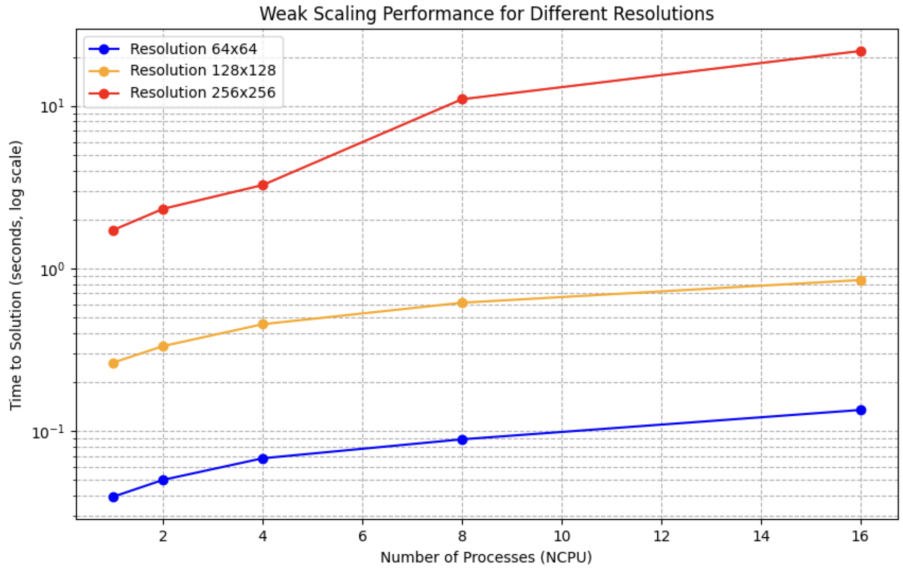

Figure 8: Weak Scaling Plot

Key Observations from the Results

Small Grid Size (64×64) Results:

- With $N_{CPU} = 1$, the time to solution is 0.0392 seconds for a 64×64 grid.
- As the number of CPUs increases to 2, 4, 8, and 16, the time increases proportionally as well (to 0.1342 seconds for $N_{CPU} = 16$).
- The increase in time as more processes are added indicates a higher communication overhead, which is expected for MPI-based parallelization.

Medium Grid Size (128×128) Results:

- For $N_{CPU} = 1$, the time is 0.2627 seconds for a 128×128 grid.
- Again, as N_{CPU} increases, the time increases, with the time reaching 0.8462 seconds at $N_{CPU} = 16$.

- The time increase per process is more noticeable than for the 64×64 grid, suggesting that with larger problem sizes, the communication overhead becomes more pronounced.

Large Grid Size (256×256) Results:

- For $N_{CPU} = 1$, the time to solution is 1.7192 seconds for a 256×256 grid.
- As N_{CPU} increases, the time grows significantly, reaching 21.7381 seconds at $N_{CPU} = 16$.
- This shows that as the problem size increases, the system struggles more with scaling efficiently, especially when the number of CPUs is large.

General Trends

Proportional Increase in Time: In weak scaling, the time to solution ideally increases slowly as both the number of processes and the problem size increase. However, the time increases quite a bit as N_{CPU} grows, which indicates that the system's ability to handle larger grids efficiently diminishes as the number of CPUs increases.

Communication Overhead: As the problem size increases, the relative communication time between processes becomes more significant. This could be due to higher data exchange requirements, which are more challenging for MPI implementations than for OpenMP-based shared memory parallelism.

Scaling Efficiency: The results show sub-optimal scaling efficiency. For perfect weak scaling, the time to solution should ideally remain constant as both the problem size and number of processes increase. However, the time increases noticeably for each increase in N_{CPU}, particularly for the larger grid sizes.

Comparison with OpenMP Implementation (Project 3)

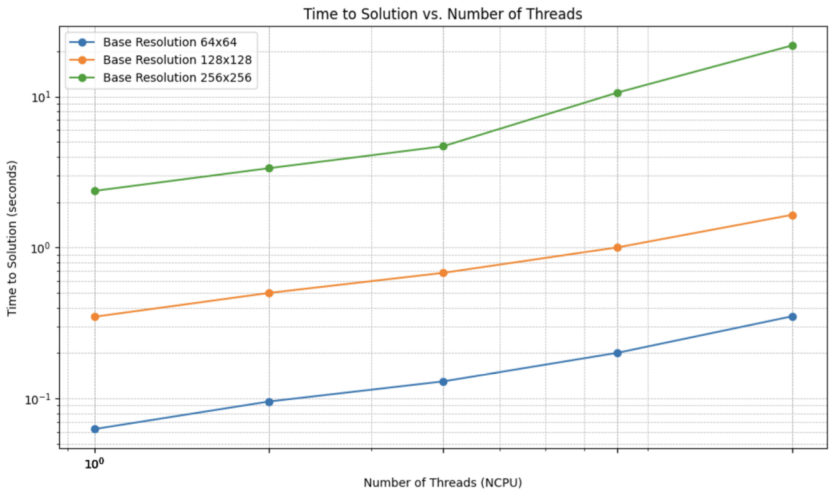

Figure 9: Project 3 Weak Scaling Plot

In Project 3 the graph is shown in figure 9, while OpenMP is used.The shared memory model resulted in more efficient communication and scaling for smaller grid sizes. However, as the grid size grows, the inefficiencies of OpenMP could also surface due to memory contention or synchronization issues. In comparison:

- MPI generally has higher communication overhead due to the distributed memory model, and as the problem size grows, this overhead becomes more pronounced.

- OpenMP would likely perform better for smaller grid sizes due to lower communication overhead, but for very large grid sizes (like 256×256), the scaling might be comparable or even worse due to memory bottlenecks.

Therefore, the MPI implementation, although it maintains constant work per process, faces significant challenges in scaling efficiently as the number of processes increases, especially with large problem sizes.

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

I implemented the whole set up and it worked perfectly achieved the expected output below in figure 10:

Figure 10: Set up Implementation

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

In this section, I coded a file called `sum_of_ranks.py`, where passing the argument "`pickle`" will yield pickle-based (serialized) point-to-point communication of `mpi4py`, whereas the argument "`buff`" will yield the buffered, faster version.

I present an implementation of the sum of ranks computation using MPI collective communication methods in Python with the `mpi4py` library. The project explores two communication modes:

1. Pickle-based communication for serialized point-to-point message passing.
2. Buffered communication for faster, low-level array data transfer.

The goal of the methodology is to compute the sum of all MPI process ranks in two distinct ways, highlighting the differences in performance and communication mechanisms between the two modes.

2.1.1. Pickle-based Communication

In this mode, I use Python's `pickle` serialization method as shown in figure 11 to send and receive Python objects between processes. Each process sends its rank to the next process in a ring topology, and the received rank is added to a running sum. I obtained the following output :

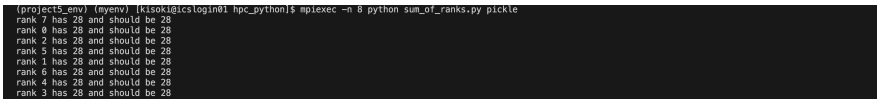

Figure 11: Pickle-based output

2.1.2. Buffered Communication

In this mode, communication is performed using MPI's direct array data transfer method as shown in figure 12 , which is more efficient than the pickle-based communication. The process uses `comm.Send()` and `comm.Recv()` to send and receive numpy arrays, allowing faster communication between processes. This method reduces the overhead of serialization, offering faster communication speeds compared to the pickle-based method.

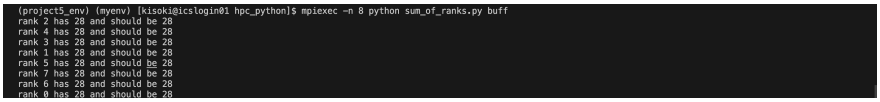

Figure 12: Buffered-based output

Results

The following results were observed when running the script with the arguments `pickle` and `buf`:

2.1. Pickle-based Communication

- The ranks were successfully passed from one process to the next using serialized data.
- The final sum of ranks was correctly computed and matched the expected value.
- The communication speed was slower due to the serialization overhead, as expected.

2.2. Buffered Communication

- The communication was faster than the pickle-based method due to the use of direct memory buffers.
- The final sum of ranks was again computed correctly and matched the expected value.
- Buffered communication demonstrated scalability and efficiency, especially for larger process counts.

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

2.3.1. create a Cartesian topology using MPI for Python

To implement this, I wrote a file named "`cart_decomposition.py` "that uses pickle-based functions.

2.3.2. Determine the neighboring processes

I made use of a cartesian decomposition and buffered methods and the output is as shown in figure 13. The file implementing this is "`ghost_exchange.py`".

In this task, a two-dimensional periodic Cartesian topology was created using MPI for Python, and ghost cell exchange was implemented. The goal was to set up a grid where neighboring processes can communicate with each other to exchange data, ensuring that the grid behaves as if it is "wrapped" in both the horizontal and vertical directions (periodic boundaries).

Output:

The output displayed the grid configuration for each process, showing its rank, coordinates, and the neighboring processes in the four directions. Here is an example of the output for one of the processes:

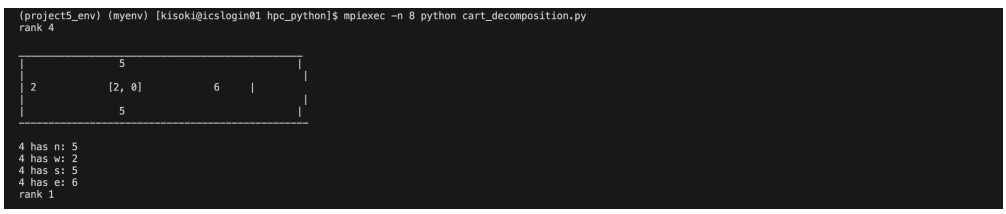

Figure 13: Expected result for a 2D periodic grid

Command Used to Run the Script:

The script was executed using the following command:

`mpiexec -n 8 python cart_decomposition.py`

This command ran the script with 8 processes, as required.

Verification:

The output was consistent with the expected result for a 2D periodic grid. Each process correctly identified its neighbors in all four directions, and the Cartesian coordinates aligned with the process rank in the grid.

Ghost Exchange Verification

I ran the code using this command "`mpiexec -n 8 python ghost_exchange.py`"

Before Exchange: The grid consists of 8 rows and 8 columns, where all values are initially set to 1 and the output is as shown in figure 14:

Figure 14: Ghost exchange output

Key Observations:

- The outer edges of the grid have changed, with some rows filled with different values (7s and 3s) after the ghost cell exchange.
- The inner cells (those not on the border) remain unchanged with values of 1. This suggests that the ghost cell exchange correctly updated the boundary cells (the ghost cells) with data from neighboring processes.
- The 7 and 3 values on the boundaries indicate that the neighboring processes exchanged rank information (or ghost cells) as expected.

Therefore, the ghost cell exchange works perfectly and verified by updating the boundary cells of each process's local grid with values from its neighbors.

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

In this section I implemented and analyzed the parallel computation of the Mandelbrot set using the manager-worker pattern. The task involves:

- Implementing the parallelized Mandelbrot set computation.
- Conducting a scalability study with 2 to 16 workers for a 4001x4001 domain.
- Decomposing the workload into 50 and 100 tasks and analyzing the performance.

2.5. Parallel Mandelbrot Set Computation

The Mandelbrot set computation is divided into rectangular patches, each representing a sub-task. The manager process:

- Decomposes the Mandelbrot set into a set of tasks (patches).
- Assigns these tasks to the worker processes.
- Once a worker completes a task, it sends the result back to the manager and requests another task.

- This cycle continues until all tasks are completed.
- Finally, the manager combines the results of the patches to produce the full Mandelbrot set.

The worker processes independently compute the Mandelbrot set for their assigned patches using the `mandelbrot_patch` class, which performs the actual computation. The `mandelbrot` class handles the task decomposition and result combination.

2.6. Scalability Study

The performance is evaluated with 2, 4, 8 and 16 worker processes (ranks) .The workload is split into 50 and 100 tasks for each scenario.

The experiments were executed using MPI with the following command:

```
mpiexec -n 4 python manager_worker.py 4001 4001 100
```

In this setup, 4001 4001 100 specifies the domain size, which is a grid of dimensions 4001 × 4001 4001×4001. The number of tasks, set to 100 and the number of ranks (n n) is varied to assess scalability and performance under different configurations.

2.7. Data Collected

The collected data for different domain sizes and worker configurations is summarized in the following tables:

Example: Execution Time for nx = 4001

Ranks	Tasks	Time (s)	Speedup	Efficiency
2	50	24.5874	1	0.50
2	100	24.8438	0.9897	0.4949
4	50	8.6955	2.8277	0.7069
4	100	8.4655	2.9044	0.7261
8	50	4.4085	5.5773	0.6972
8	100	3.9515	6.2222	0.7778
16	50	2.2157	11.0969	0.6936
16	100	1.8918	12.9968	0.8123

Table 3: The table showing execution time, Speedup and Efficiency for the Mandelbrot set computation

2.8. Plots

Plot 1: Time vs. Ranks for 50 and 100 Tasks The first plot, which shown in figure .15 illustrates the execution time for varying ranks (2 to 16) and domain size (nx=4001). The key observation here is that execution time decreases as the number of ranks increases.

Plot 3: Speedup vs. Number of Ranks This plot in figure 16 show the speedup achieved by varying different ranks illustrating how well the parallelization scales up.

Plot 4: Efficiency vs. Number of Ranks This plot displayed in figure 17shows parallel efficiency, which is the ratio of observed speedup to ideal speedup, as the number of ranks increases.

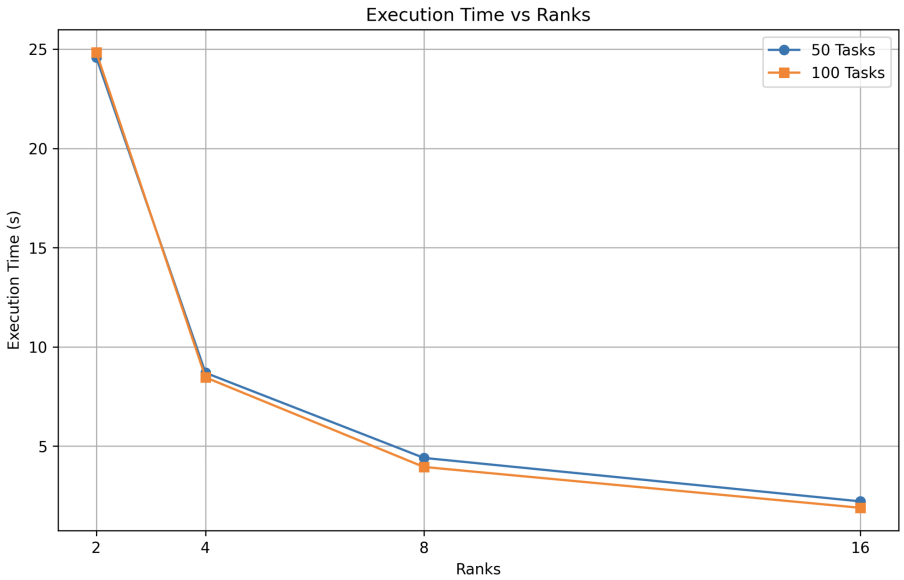

Figure 15: Time vs Number of Ranks Plot

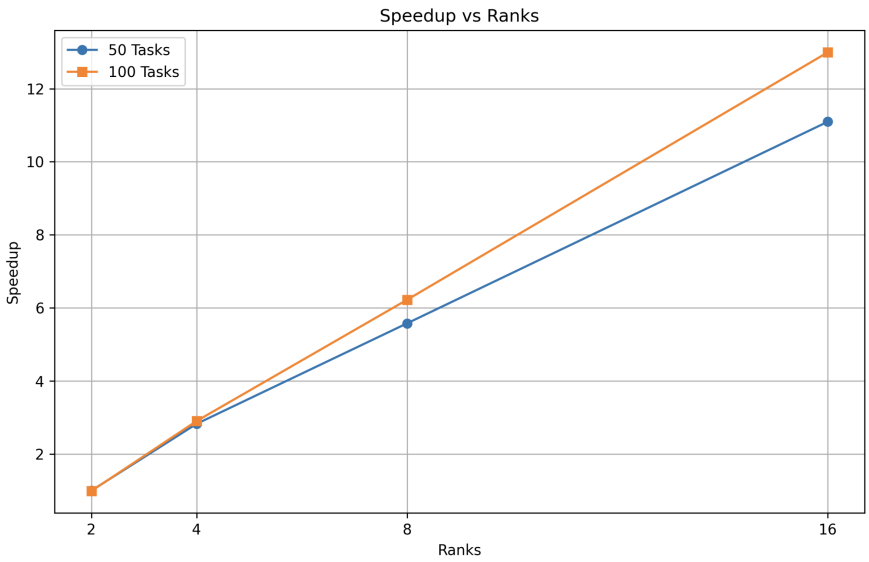

Figure 16: Speedup vs Number of Ranks Plot

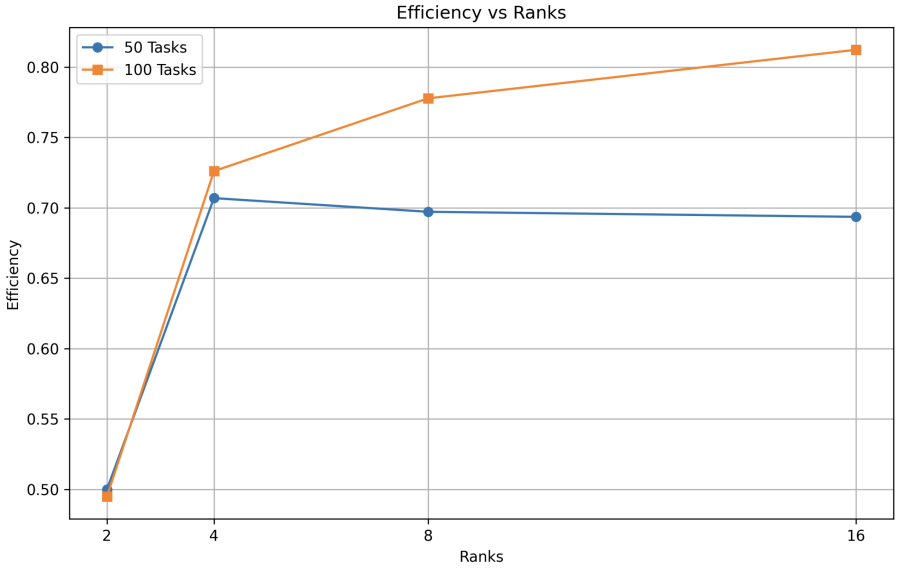

Figure 17: Efficiency vs Number of Ranks Plot

2.9. Scalability

As the number of ranks increases, I observed a decrease in execution time.

2.10. Parallel Efficiency

Parallel efficiency tends to decrease as the number of ranks increases.

2.11. Task Decomposition

The decomposition of the Mandelbrot set into 50 and 100 tasks was effective in balancing the workload across ranks. However, for very small domains, fewer tasks may provide better performance by reducing communication overhead.

References

[1] Dong Zhong, Qinglei Cao, George Bosilca, and Jack Dongarra. Using long vector extensions for mpi reductions. *Parallel Computing*, 109, 3 2022.

[2] Taher Saif and Manish Parashar. Understanding the behavior and performance of non-blocking communications in mpi. pages 173–182, 08 2004.

2.1. 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.