Università della Svizzera italiana

**Institute of Computing**
**CI**

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

Student: Paolo Deidda Discussed with: Lino Candian

# Solution for Project 5

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

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

## Contents

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

Image solution for this exercise can seen in Figure 3.

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

Output:

```
================================================================================
                      Welcome to mini-stencil!
version   :: C++ MPI
threads   :: 1
mesh      :: 10 * 10 dx = 0.111111
time      :: 10 time steps from 0 .. 10
iteration :: CG 300, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
simulation took 0.000143 seconds
301 conjugate gradient iterations, at rate of 2.1049e+06 iters/second
1 newton iterations
--------------------------------------------------------------------------------
### 1, 10, 10, 301, 1, 0.000143 ###
Goodbye!
```

## 1.2. Domain decomposition [10 Points]

I use a two–dimensional Cartesian domain decomposition:

- **Process grid creation.** I obtain a balanced $d_x \times d_y$ grid using `MPI_Dims_create`, which guarantees that the process grid is as close to square as possible for any number of processes $P$.
- **Cartesian topology.** I build the topology with `MPI_Cart_create` so that each process can identify its coordinates and its four neighbors through the apposite `MPI_Cart_coords` and `MPI_Cart_shift` functions.
- **Subdomain sizes.** The global domain $n \times n$ is partitioned into rectangular subdomains. Local sizes $(n_x, n_y)$ are computed from $(d_x, d_y)$, and any remainder is distributed across the first processes in each direction so that subdomain sizes differ by at most one cell.
- **Load balancing.** With this decomposition all processes receive almost the same number of grid points $N = n_x n_y$.
- **Communication overhead.** In a 2D layout, the amount of halo data exchanged with neighbors grows only as $\mathcal{O}(n_x + n_y)$, while work grows as $\mathcal{O}(n_x n_y)$. This results in a lower communication-to-computation ratio than a 1D decomposition.

## 1.3. Linear algebra kernels [5 Points]

Only two kernels required MPI parallelization:

- **hpc_dot**: computes a global inner product. I added `MPI_Allreduce` to combine the partial sums from all ranks.
- **hpc_norm2**: computes the global 2-norm. I used `MPI_Allreduce` to sum the local squared values before taking the square root.

All other `hpc_xxx` functions operate purely on local data (vector updates, axpy, copy, scale) and therefore do not require communication or any MPI modification.

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

I implemented the ghost-cell exchange using **non-blocking point-to-point communication** so that communication and computation can overlap. The steps and MPI calls used are:

- **Packing send buffers.** Before communication, each process copies its local boundary data into four buffers: `buffN`, `buffS`, `buffE`, `buffW`.
- **Posting non-blocking receives: `MPI_Irecv`.** For every existing neighbor (north, south, east, west), I post an `MPI_Irecv` on the corresponding receive buffer (`bndN`, `bndS`, `bndE`, `bndW`). Since `MPI_Irecv` is non-blocking, control returns immediately without waiting for the message.
- **Posting non-blocking sends: `MPI_Isend`.** After the receives, each process posts the matching `MPI_Isend` using the packed buffers. Being non-blocking, these sends also do not stop execution.
- **Overlap of communication and computation.** Thanks to the non-blocking operations, the stencil computation on all **interior points** (i.e. grid points that do not require ghost data) can proceed while data is in transit. Only boundary points depend on the incoming ghost values, so they are computed later.
- **Waiting for completion: `MPI_Waitall`.** After finishing the interior region, all pending communications are completed with a single `MPI_Waitall`. Once the ghost buffers are updated, the stencil is applied safely to the boundary and corner points.

With blocking functions like `MPI_Send`, `MPI_Recv`, or `MPI_Sendrecv`, every process has to wait for its neighbors to finish before it can move on, which really slows things down. But by using `MPI_Isend`, `MPI_Irecv`, and `MPI_Waitall`, the solver can work on the calculations while it's waiting for communication, which helps boost efficiency.

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

In the `write_binary` function, I handle parallel output using MPI-I/O file views. Instead of figuring out byte offsets for each row by hand, I create a derived datatype that aligns the local subdomain straight onto the global grid structure.

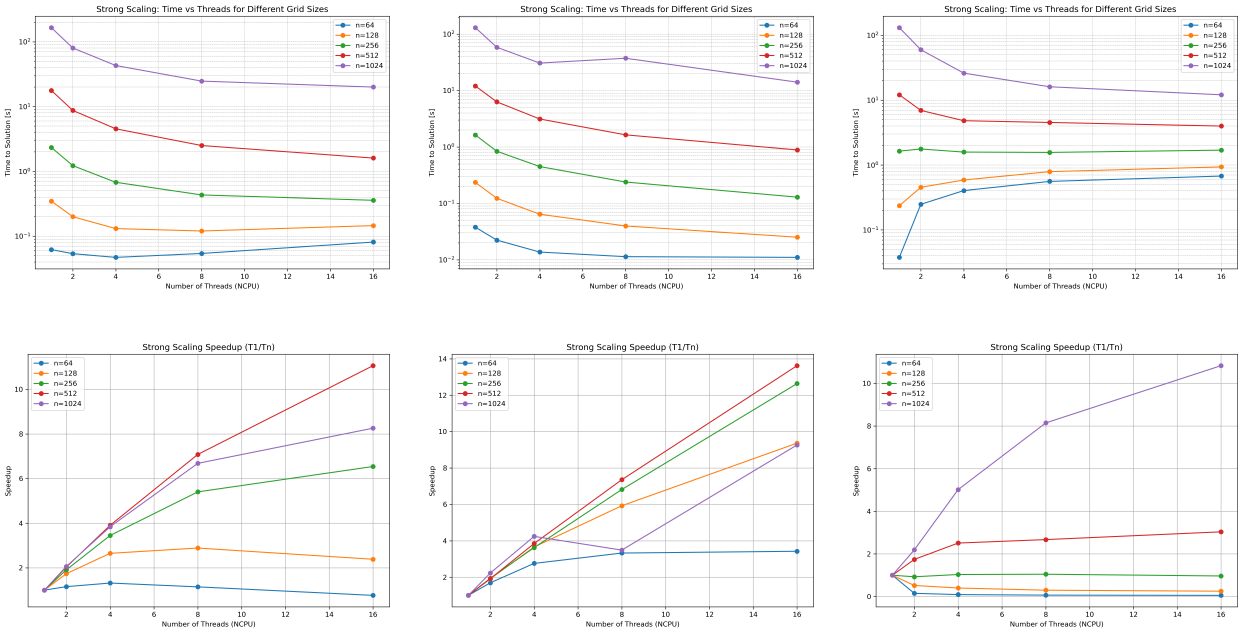
The main feature here is `MPI_Type_create_subarray`, which sets up a non-contiguous data layout. With the global dimensions of $(N \times N)$ and the local coordinates $(\text{start}_x, \text{start}_y)$, MPI takes care of calculating the displacement in the file automatically. In simple terms, this determines where we'll write in the global offset:

$$\text{Offset} = (\text{start}_y \times N_{\text{global}} + \text{start}_x) \times \text{sizeof(double)}$$

By using this datatype with `MPI_File_set_view`, processes can call `MPI_File_write_all` to write their whole block together. This setup lets the MPI library fine-tune disk access while keeping the global order of the matrix intact. As a result, we end up with one binary array that holds the complete global solution, eliminating the need for any extra processing or temporary files.

## 1.6. Strong scaling [10 Points]

Higher resulution images can be found in their respective folders in mini_app




(a) openMP (project 3) (b) MPI multi task over 1 node (c) MPI 1 task over multiple nodes

Figure 1: *Strong* scaling results: **Above** time over number of threads - **Below** efficiency over number of threads.

Figure 1 shows the results for strong scaling, comparing three setups: the OpenMP implementation 1a MPI on a single node 1b, and MPI spread across multiple nodes 1c. The top row presents the time it took to reach a solution, while the bottom row illustrates the speedup ($T_1/T_N$).

**Comparison with OpenMP (Single Node):** Looking at MPI on a single node: The MPI setup running on one node 1b has a scaling profile that's pretty similar to the OpenMP version 1a. When we work with larger grid sizes ($n = 1024$), both setups manage to achieve almost linear speedup up to 16 processes, with MPI showing a slight edge in performance (reaching a speedup of $\approx 13\times$ vs. $\approx 11\times$ for OpenMP). This indicates that the overhead from MPI's internal memory copying either compares or is better managed than the thread synchronization that come with the OpenMP shared-memory model.

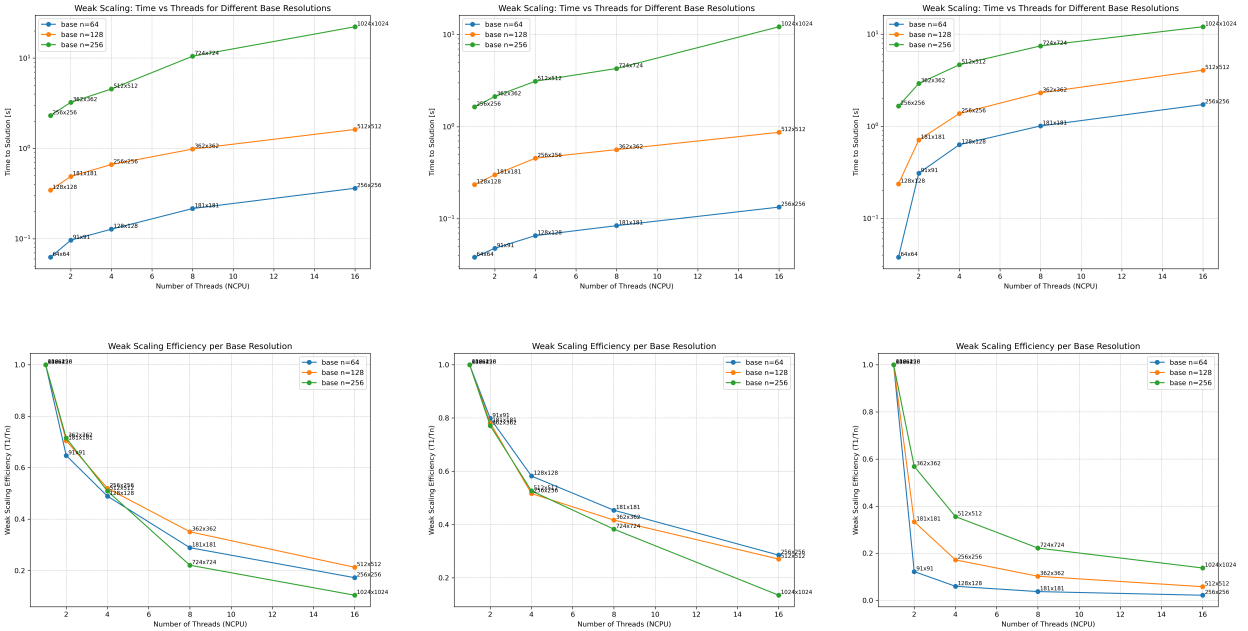
**Distributed Memory Performance (Multi-Node):** The multi-node MPI configuration (c) highlights the critical impact of network latency.

- **Large Grids ($n = 1024$):** The computations scale impressively, hitting the highest linearity among the three cases tested. The effective computation-to-communication ratio does a great job of masking the overhead from the network.
- **Small Grids ($n \leq 128$):** Performance takes a significant hit. When $n = 64$, the speedup drops below 1.0 right away, indicating that parallel execution ends up slower than doing it serially. This happens because the sub-domains are too small; the fixed costs associated with network latency and halo exchanges overshadow the small amount of computational work available per rank.

In summary, while MPI performs just as well as OpenMP within a single node, its scalability across multiple nodes really rely on the size of the problem. Distributed parallelism proves advantageous only when the local domain size is large enough to offset the costs associated with communication between nodes.

## 1.7. Weak scaling [10 Points]

Higher resulution images can be found in their respective folders in mini_app




(a) openMP (project 3) (b) MPI multi task over 1 node (c) MPI 1 task over multiple nodes

Figure 2: *Weak* scaling results: **Above** time over number of threads - **Below** efficiency over number of threads.

Figure 2 shows how weak scaling performs when the problem size grows along with the number of processing elements ($N$), which helps keep a consistent workload for each process. Ideally, we want the time it takes to reach a solution to stay the same as a flat line; an efficiency of $\approx 1$.

**Single-Node Bottlenecks (OpenMP vs. MPI):** Both the OpenMP implementation 2a and the single-node MPI run 2b show a notable drop from the ideal weak scaling. As $N$ increases, the runtime goes up steadily, causing the efficiency to drop below 0.4 at $N = 16$. This suggests that the system is **memory-bound**. Even though the computational workload per core stays constant, the total memory bandwidth needed increases with N. Since all processes on a single node are competing for the same shared memory bus, the bandwidth fills up, leading to a bottleneck that slows down all processes at once. The similarity between 2a and 2b indicates that this hardware limitation impacts both parallel models equally.

**Multi-Node Scaling and Network Overhead:** The multi-node configuration 2c presents a dual behavior dependent on the base resolution:

- **Small Base ($n = 64$):** The performance is quite poor here. Right off the bat, the efficiency drops to $\approx 0.1$ immediately. The local sub-domain is just too small to make the fixed overhead of network latency (inter-node communication) worthwhile, which results in execution being limited by latency.
- **Large Base ($n = 256$):** Interestingly, the multi-node configuration performs better than the single-node setup. The runtime slope for the green line in 2c is less steep than in 2b. This is because spreading tasks across multiple nodes means access to combined memory bandwidth, as each node has its own memory bus. Unlike the single-node scenario, the computation isn't held back by memory contention, provided the local problem size is large enough to offset the costs of network communication.

In conclusion, while single-node scaling is held back by hardware memory bandwidth, multi-node scaling is mainly limited by network latency. So, the multi-node approach is definitely better for large-scale problems that need high memory throughput.

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

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

Two different communication approaches: generic *object serialization* vs *direct buffer handling*.

- **Pickle-based communication (`sum_ranks_pickle.py`):** I used the lowercase `comm.allreduce` method, which lets us send any Python objects, like simple integers. However, it needs the `pickle` module for both serialization and deserialization. While it offers great flexibility, it does come with some considerable overhead compared to native C operations.
- **Buffer-based communication (`sum_ranks_buffer.py`):** I used the uppercase `comm.Allreduce` method along with `numpy` arrays that are typed as integers (`dtype='i'`). This to avoid Python's overhead by directly passing memory pointers to the MPI C implementation beneath, leading to a much higher performance that's perfect for high-performance computing workloads.

**Verification:** with $P = 8$ processes the expected result is the sum of integers from 0 to 7:

$$S = \frac{n(n-1)}{2} = \frac{8 \times 7}{2} = 28$$

The output confirms the correctness of both implementations:

```
[buffer] Sum of ranks over 8 processes = 28
[pickle] Sum of ranks over 8 processes = 28
```

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

In this task, I recreated the 2D Cartesian topology and the ghost cell exchange mechanism using the `mpi4py` library. This implementation is built on four important functions that help manage the domain decomposition and communication:

- `MPI.Compute_dims(size, 2)`: Automatically calculates the optimal grid dimensions based on the number of available processes. In my test case with $P = 8$, it generated a $4 \times 2$ grid.
- `comm.Create_cart(dims, periods=[True, True])`: Creates a new communicator with a Cartesian topology. The `periods` argument enables periodic boundaries (torus) in both vertical and horizontal directions.
- `cart_comm.Shift(direction, disp=1)`: Queries the topology to determine the ranks of neighbors. It returns the source (neighbor in the negative direction) and destination (neighbor in the positive direction) for data shifting.
- `cart_comm.sendrecv(...)`: Performs a blocking send and receive operation simultaneously. This is safer than separate blocking send/recv calls as it automatically handles buffering to prevent deadlocks during the cyclic exchange.

**Verification:** The script was executed with 8 processes, which produced a $4 \times 2$ grid. The output shows that the topological setup and data exchange are correct. For instance, **Rank 0** (coordinates $[0, 0]$) accurately recognizes its North neighbor as **Rank 6** (coordinates $[3, 0]$), demonstrating that the vertical boundary wraps around periodically. Additionally, the verification step confirms that the data received matches the rank of the topological neighbors.

```
Rank 00 | Coords [0, 0] | Neighbors (N, S, W, E): 06, 02, 01, 01
Rank 01 | Coords [0, 1] | Neighbors (N, S, W, E): 07, 03, 00, 00
Rank 02 | Coords [1, 0] | Neighbors (N, S, W, E): 00, 04, 03, 03
Rank 03 | Coords [1, 1] | Neighbors (N, S, W, E): 01, 05, 02, 02
Rank 04 | Coords [2, 0] | Neighbors (N, S, W, E): 02, 06, 05, 05
Rank 05 | Coords [2, 1] | Neighbors (N, S, W, E): 03, 07, 04, 04
Rank 06 | Coords [3, 0] | Neighbors (N, S, W, E): 04, 00, 07, 07
Rank 07 | Coords [3, 1] | Neighbors (N, S, W, E): 05, 01, 06, 06
```

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

In this project, I implemented a load-balancing system using `mpi4py` for the *Manager-Worker* model. This approach works really well for the Mandelbrot set, where the computational load per pixel can differ a lot—like, pixels inside the set need maximum iterations to calculate, but those outside can diverge in no time.

**Implementation Details:** The logic is divided based on the MPI rank:

- **Manager (Rank 0):** The system keeps a queue of tasks. It starts by sending one task to each worker with `comm.send`. Then, it gets into a loop, waiting for *any* worker to finish a task. As soon as it gets a result back, it sends a new task to that worker right away. When there are no more tasks left in the queue, it sends a termination signal `TAG_DONE`.
- **Workers (Rank $>$ 0):** Execute an infinite loop waiting for messages. Upon receiving a task, they compute the Mandelbrot patch and send the object back. The loop terminates only upon receiving `TAG_DONE`.

**Scaling Analysis:** As requested the study was done on a $4001 \times 4001$ area using between 2 and 16 workers, looking at two different granularities: 50 tasks and 100 tasks. The results are all in Table 1.

| Workers | Total Procs | Time (50 Tasks) | Time (100 Tasks) |
|---|---|---|---|
| 2 | 3 | 13.49 s | 13.43 s |
| 4 | 5 | 8.00 s | 7.51 s |
| 8 | 9 | 4.78 s | 4.56 s |
| 16 | 17 | 3.56 s | **3.21 s** |

Table 1: Runtime comparison for different task granularities ($4001^2$ grid).

**Observations:**

1. **Load Balancing Benefits:** The configuration with 100 tasks consistently beats the one with just 50. With 16 workers, breaking it down into 100 tasks gives you $\approx 10\%$ boost in performance 3.21 seconds compared to 3.56 seconds.
2. **Granularity Impact:** With only 50 tasks for those 16 workers, each is handling $\approx 3$ patches on average. That can lead to a few workers getting overloaded with tougher tasks, which we call 'stragglers.' But when you bump it up to 100 tasks, with each worker managing avg $\approx 6$, it balances things out. Faster workers can tackle easier patches, while others deal with the harder ones, reducing the chances of everyone just sitting around waiting at the end of the simulation.

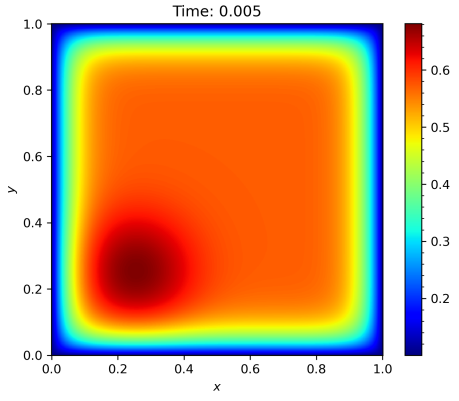



Figure 3: ex1 - Nonlinear PDE

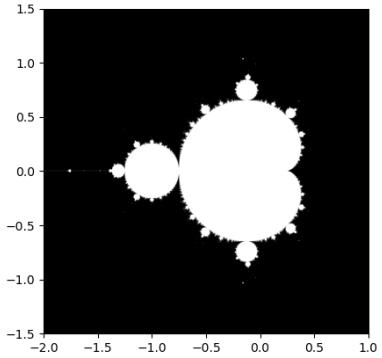



Figure 4: ex 2.3 - Mandelbrot set

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