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]

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 logical topology with `MPI_Cart_create`, so that each process can identify its coordinates and its four neighbors through `MPI_Cart_coords` and `MPI_Cart_shift`.
- **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.

A blocking version (`MPI_Send`, `MPI_Recv`, or `MPI_Sendrecv`) would force each process to wait for neighbors before continuing, preventing overlap and reducing performance. Using `MPI_Irecv`, `MPI_Isend`, and `MPI_Waitall` allows the solver to hide communication costs behind the interior computation, improving overall efficiency.

1.5. Implement parallel I/O [10 Points]

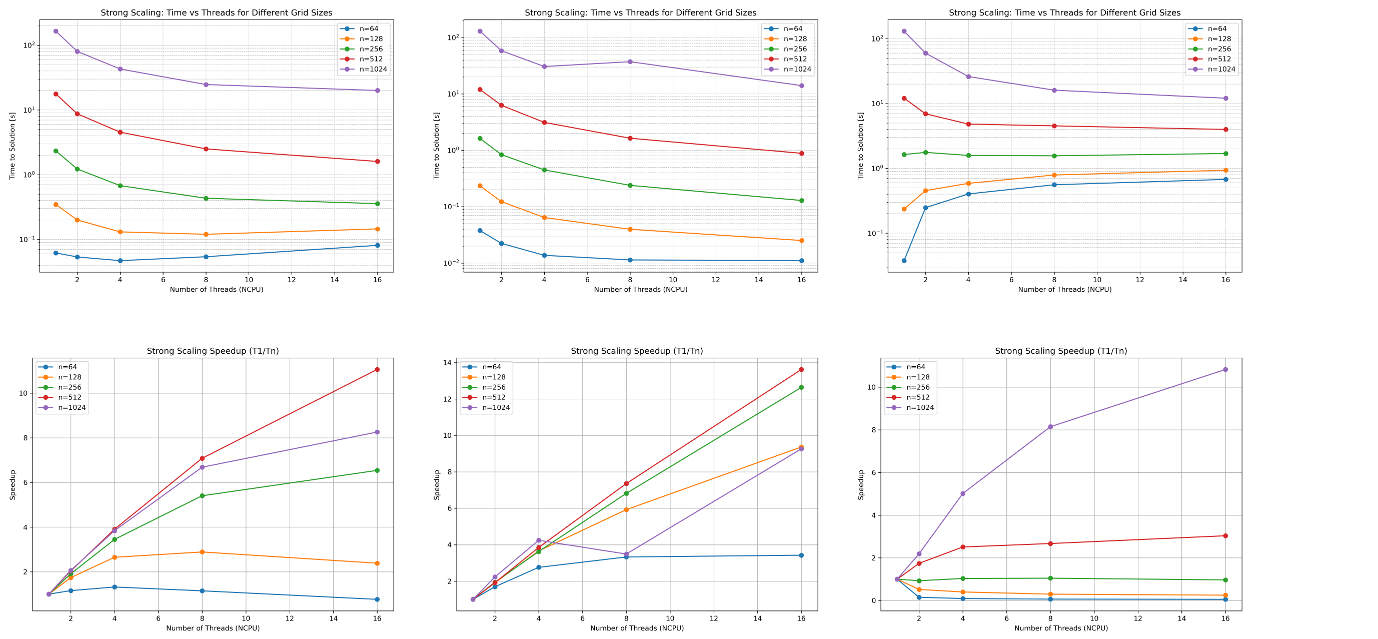
I implemented the final output using **MPI-IO** so that all processes write their local subdomains into a single global binary file. The steps are:

- **MPI_File_open:** The file is opened collectively by all ranks in write-only mode.
- **Offset computation:** Each process computes its byte offset inside the global $n \times n$ array using its global start indices $(\text{start}_x, \text{start}_y)$ provided by the domain decomposition.
- **Derived datatype:** I created a derived MPI datatype using `MPI_Type_contiguous` to represent a full local row of n_x doubles, simplifying the write calls.
- **MPI_File_write_at:** Each process writes its n_y local rows into the correct position in the global file without any file locking or coordination between processes.
- **Metadata:** Only rank 0 writes the accompanying `.bov` visualization header.

This method produces a single binary array that contains the full global solution, with no need for post-processing or intermediate 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 presents the strong scaling results, comparing the OpenMP implementation (a), MPI on a single node (b), and MPI distributed across multiple nodes (c). The top row displays the time to solution, while the bottom row illustrates the speedup (T_1/T_N).

Comparison with OpenMP (Single Node): The MPI implementation executed on a single node (b) exhibits a scaling profile remarkably similar to the OpenMP version (a). For large grid sizes ($n = 1024$), both implementations achieve near-linear speedup up to 16 processes, with the MPI version showing slightly superior performance (reaching a speedup of $\approx 13\times$ vs. $\approx 11\times$ for OpenMP). This suggests that the overhead of MPI internal memory copying is comparable to, or potentially handled better than, the thread synchronization and false sharing overheads inherent in the OpenMP shared-memory model.

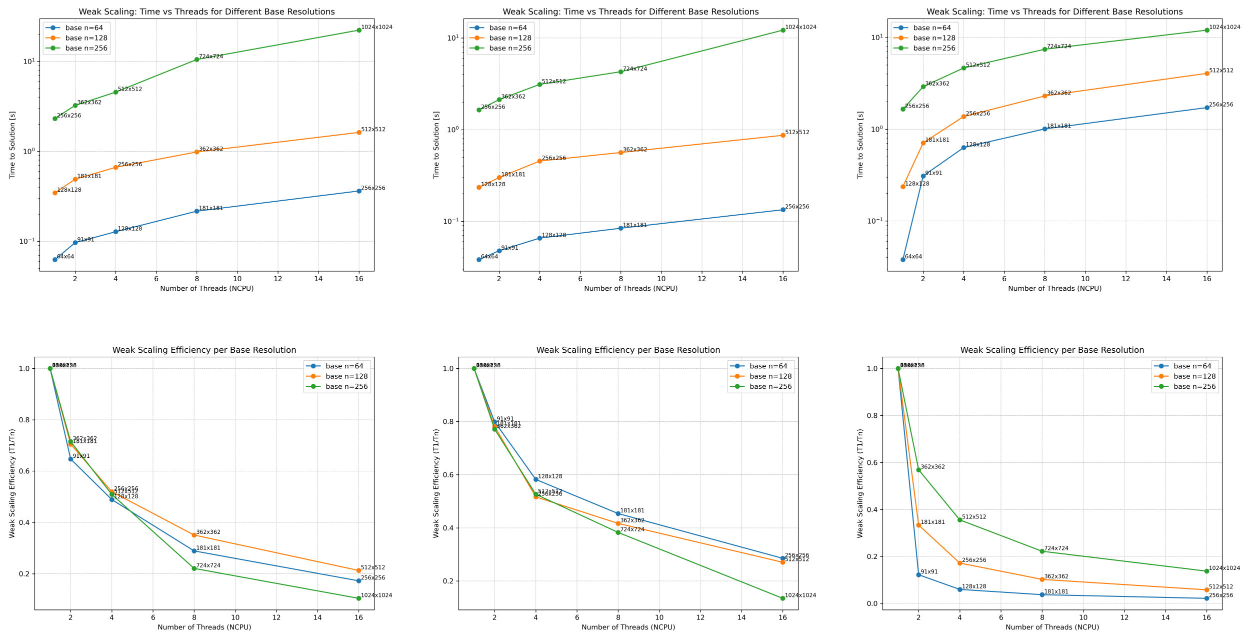
Distributed Memory Performance (Multi-Node): The multi-node MPI configuration (c) highlights the critical impact of network latency.

- **Large Grids ($n = 1024$):** The computation scales excellently, achieving the highest linearity among the three test cases. The high computation-to-communication ratio effectively hides the network overhead.
- **Small Grids ($n \leq 128$):** Performance degrades significantly. For $n = 64$, the speedup drops below 1.0 immediately, meaning the parallel execution is slower than the serial one. This occurs because the sub-domains are too small; the fixed cost of network latency and halo exchange dominates the negligible computational work available per rank.

In summary, while MPI matches OpenMP efficiency within a node, the multi-node scalability is strictly bound by the problem size. Distributed parallelism is only beneficial when the local domain size remains large enough to amortize the cost of inter-node communication.

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 illustrates the weak scaling performance, where the problem size increases proportionally with the number of processing elements (N) to maintain a constant workload per process. Ideally, the time to solution should remain constant (flat line), corresponding to an efficiency of 1.0.

Single-Node Bottlenecks (OpenMP vs. MPI): Both the OpenMP implementation (a) and the single-node MPI execution (b) exhibit a significant deviation from ideal weak scaling. As N increases, the runtime rises steadily, causing efficiency to drop below 0.4 for $N = 16$. This behavior indicates that the system is **memory-bound**. Although the computational work per core is constant, the *aggregate* memory bandwidth requirement increases with N. Since all processes on a single node compete for the same shared memory bus, the bandwidth saturates, creating a bottleneck that slows down all processes simultaneously. The similarity between (a) and (b) confirms that this hardware limitation affects both parallelization models equally.

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

- **Small Base ($n = 64$):** Performance is poor. The efficiency drops sharply to ≈ 0.1 immediately. The local sub-domain is too small to justify the fixed overhead of network latency (inter-node communication), making the execution latency-bound.
- **Large Base ($n = 256$):** Interestingly, the multi-node setup scales *better* than the single-node setup. The runtime slope for the green line in (c) is flatter than in (b). This is because distributing tasks across multiple nodes provides access to aggregated memory bandwidth (each node has its own memory bus). Unlike the single-node case, the computation is not throttled by memory contention, provided the local problem size is large enough to hide the network communication costs.

In conclusion, while single-node scaling is limited by hardware memory bandwidth, multi-node scaling is limited only by network latency. Consequently, the multi-node approach is superior for large-scale problems where high memory throughput is required.

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 utilized the lowercase `comm.allreduce` method. This approach allows sending arbitrary Python objects (in this case, simple integers) but relies on the `pickle` module for serialization and deserialization. While flexible, this introduces significant overhead compared to native C operations.
- **Buffer-based communication (`sum_ranks_buffer.py`):** I utilized the uppercase `comm.Allreduce` method combined with `numpy` arrays explicitly typed as integers (`dtype='i'`). This method bypasses Python's overhead by passing memory pointers directly to the underlying MPI C implementation, resulting in significantly higher performance suitable for HPC workloads.

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

$$S = \frac{n(n-1)}{2} = \frac{8\times7}{2} = 28$$

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, we replicated the 2D Cartesian topology and ghost cell exchange mechanism using the `mpi4py` library. The implementation relies on four key functions to 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 our test case with $P = 8$, it generated a 4×2 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 and generated a 4×2 grid. The output confirms the correct topological setup and data exchange. For example, **Rank 0** (coordinates $[0, 0]$) correctly identifies its North neighbor as **Rank 6** (coordinates $[3, 0]$), confirming that the vertical boundary wraps around periodically. 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]

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