Università della Svizzera italiana

Institute of Computing
CI

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

Student: Paolo Deidda Discussed with: FULL NAME

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.

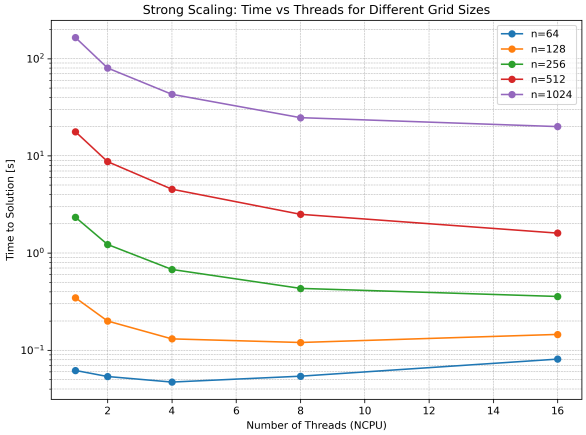
1.5. Implement parallel I/O [10 Points]

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:

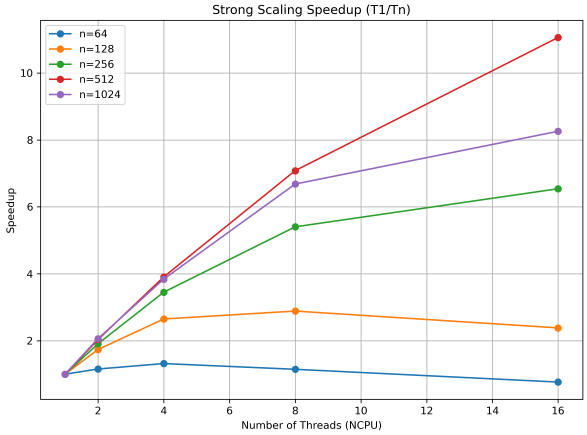
- **MPI_File_open:** The file is opened collectively by all ranks in write-only mode.
- **Offset computation:** Each process computes its byte offset inside the global $n \times n$ array using its global start indices $(\text{start}_x, \text{start}_y)$ provided by the domain decomposition.
- **Derived 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.

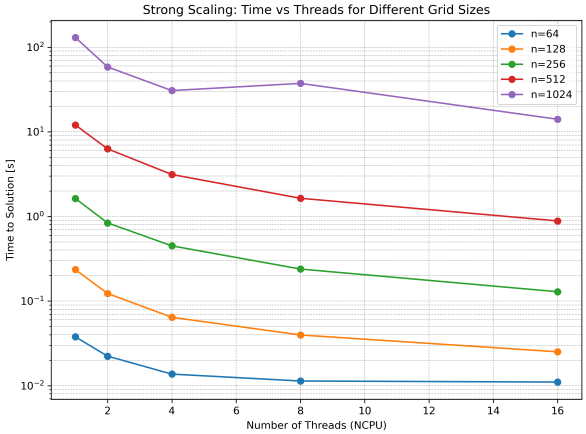
1.6. Strong scaling [10 Points]



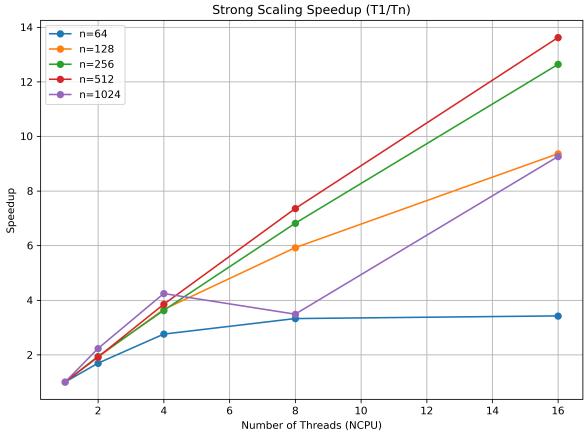
(a) strong scaling time vs threads with openMP



(b) strong scaling speedup with openMP



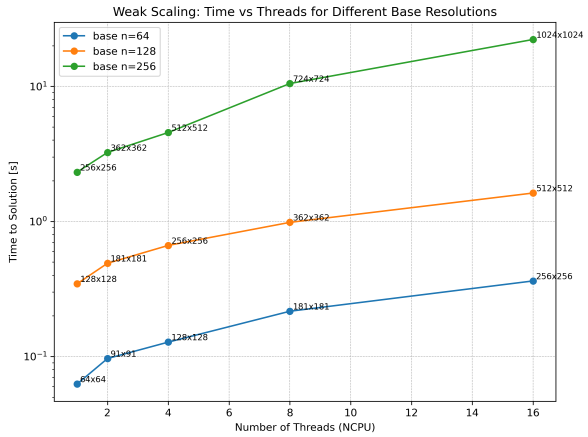
(c) strong scaling time vs threads without MPI



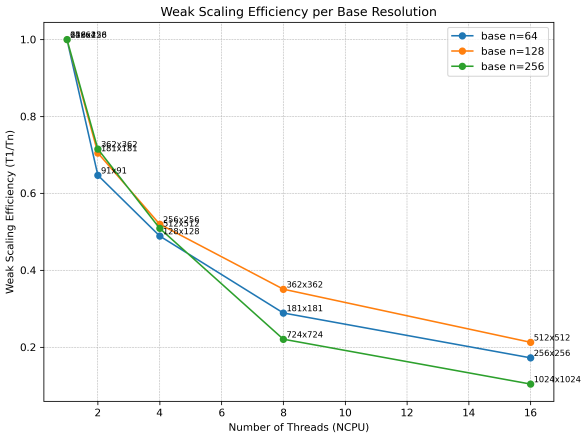
(d)
strong scaling speedup without MPI

Figure 1: Above results with OpenMP (from project 3), below results without MPI.

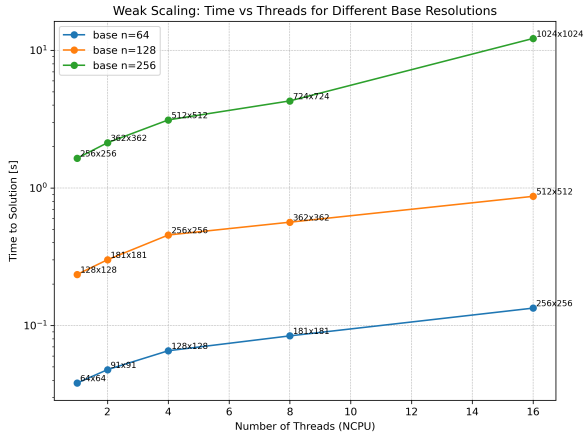
1.7. Weak scaling [10 Points]



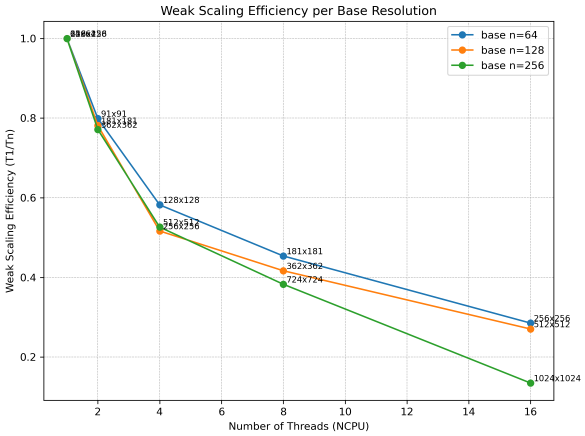
(a) weak scaling time vs threads with openMP



(b) weak scaling speedup with openMP



(c) strong scaling time vs threads without MPI



(d) weak scaling efficiency without MPI

Figure 2: Above results with OpenMP (from project 3), below results without MPI.

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

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

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

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

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