Università della Svizzera italiana

Institute of Computing
CI

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

Student: Metehan Dündar Discussed with: Only TAs

Solution for Project 7

1. Parallel Space Solution of a nonlinear PDE using MPI [in total 35 points]

1.1. Initialize and finalize MPI [5 Points]

In the file `main.cpp`, `MPI_Init` is used to initialize MPI. `MPI_Comm_rank` and `MPI_Comm_size` are used to get the current rank and number of ranks, respectively. `MPI_Comm_free` is used to free the Cartesian communicator at the end and then `MPI_Finalize` is used to finalize MPI.

1.2. Create a Cartesian topology [5 Points]

In the file `data.cpp`, a 2D domain decomposition is created in `SubDomain::init` as instructed. `MPI_Dims_create` function is created to determine the number of subdomains in the x and y direction. With the `MPI_Cart_create` function, the Cartesian communicator is created. Then using the `MPI_Cart_coords` function helps to get the coordinates of a given rank in the Cartesian topology. Finally, the `MPI_Cart_shift` function allows us to determine the neighbors of a given rank.

1.3. Extend the linear algebra functions [5 Points]

In the file `linalg.cpp`, The collective operation `MPI_Allreduce` which gathers local results is used both in `hpc_dot` and `hpc_norm` to compute the dot product and the norm, respectively. `hpc_dot` and `hpc_norm` use entries of the whole input vector to compute the output value, and thus communication among processes is necessary. Any entry of other functions' output vector only depends on the entry of the input vector with the same index, which means the local slice of the input vector is sufficient to do the computation.

1.4. Exchange ghost cells [10 Points]

In the file `operators.cpp`,

- After checking the existence of neighbor, the point-to-point communication `MPI_Irecv` and `MPI_Isend` are used to exchange the ghost cells. Though `MPI_Irecv` and `MPI_Isend` accept negative integer as the rank of source and destination, in which cases the communication requests are simply ignored, those statements are put in the `if` block to avoid unnecessary function invocations.
- The overlapping of computation and communication is reflected in several aspects.

  - The interior grid points that don't depend on boundary points are computed while nonblocking `MPI_Irecv` and `MPI_Isend` exchange ghost cells with neighbors.
  - The boundary data is copied to send buffers (`buffN`, `buffS`, `buffE`, `buffW`). Thus the boundary data can be updated while sending its copies. No waiting for sending requests is needed. Since there's always `hpc_dot` or `hpc_norm` which contains blocking communication following `diffusion`, the success of `MPI_Isend` of current process is guaranteed by the success of `MPI_Irecv` of the neighbors as processes synchronize at `MPI_Allreduce` in `hpc_dot` or `hpc_norm`.
  - `MPI_Irecv` is invoked before copying boundary data. It overlaps receiving messages and copying data.

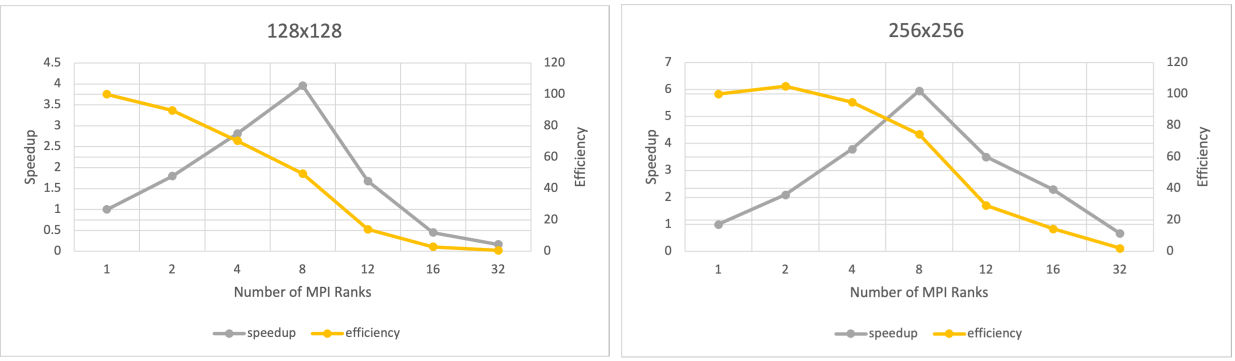
- The number of requests is counted and the `MPI_Waitall` function is used to wait for the communication to finish.

1.5. Scaling experiments [10 Points]

1.5.1. Performance Analysis: Strong Scaling

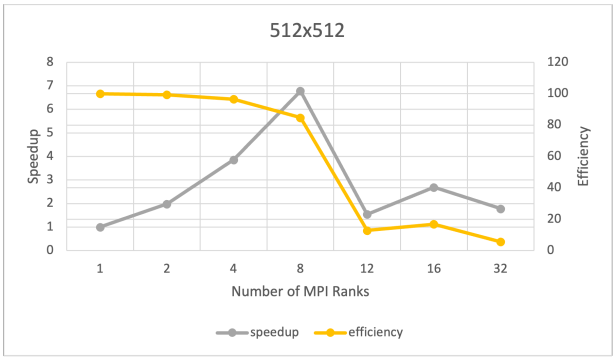
The experiments run on Apple M2 Pro with 12 cores (8 of them are for high performance, 4 of them are for energy-efficient). The performance on different grid sizes is tested and the results (100-time steps and simulation time 0-0.005s) are shown in Figure 1. The parallel speedup is calculated by $t_{serial}(N)/t_p(N)$ and the efficiency is calculated by $t_{serial}(N)/(t_p(N) \times p)$.

- As the number of MPI ranks increases, the parallel execution time decreases, leading to better speedup and efficiency, the speedup generally improves, indicating that parallelization is beneficial for the given problem.
- For the smallest problem size (128), the speedup and efficiency increase as the number of MPI ranks increases, up to 8 ranks. Beyond that, efficiency starts to decrease.
- For larger problem sizes (256, 512, 1024), the trend is similar, but efficiency starts to drop even earlier (around 4-8 ranks).
- The largest problem size (1024) shows lower efficiency overall, and there is a case where the efficiency is less than 100%, indicating diminishing returns with more MPI ranks.
- For the smallest problem size (128), the speedup and efficiency increase as the number of MPI ranks increases, up to 8 ranks. Beyond that, efficiency starts to decrease.
- For some cases, especially with a higher number of MPI ranks (e.g., 128 with 32 ranks), efficiency drops. This could be due to communication overhead or other factors that limit the scalability of the parallel implementation.

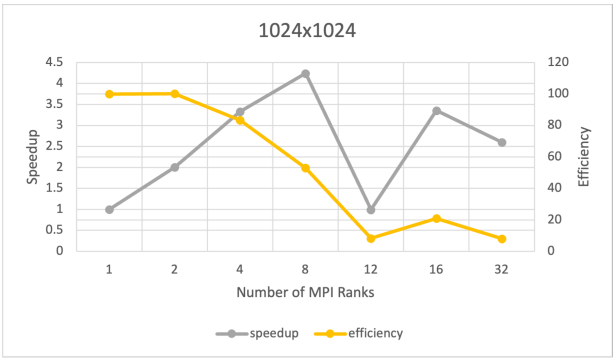


(a) 128×128 grid

(b) 256×256 grid



(c) 512×512 grid



(d) 1024×1024 grid

Figure 1: Strong scaling result

1.5.2. Performance Analysis: Weak Scaling

A complementary weak scaling study was conducted, maintaining a fixed grid size per MPI Rank while varying the number of threads. The experiments run on Apple M2 Pro with 12 cores (8 of them are for high performance, 4 of them are for energy-efficient). The performance with different workloads per MPI rank is tested and the results (100-time steps and simulation time 0-0.005s) are shown in Figure 2. The parallel efficiency is calculated by $t_{serial}(N_0)/t_p(N_p) * 100$ and speedup is calculated by $p \times t_{serial}(N_0)/t_p(N_p)$. In particular to quadruple the amount of MPI Ranks when doubling the grid size.

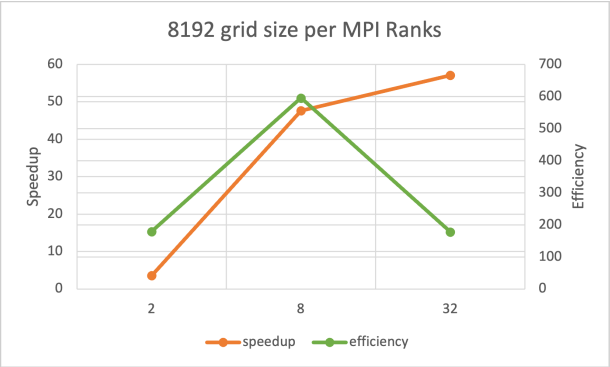
1. In the Figure 2a 8192 points per MPI Rank by running 128x128 with 2 MPI Ranks, 256x256 with 8 MPI Ranks and 512x512 with 32 threads is calculated.

2. In the Figure 2b 16384 points per MPI Rank by running 128x128 with 1 MPI Ranks, 256x256 with 4 MPI Ranks and 512x512 with 16 threads is calculated.

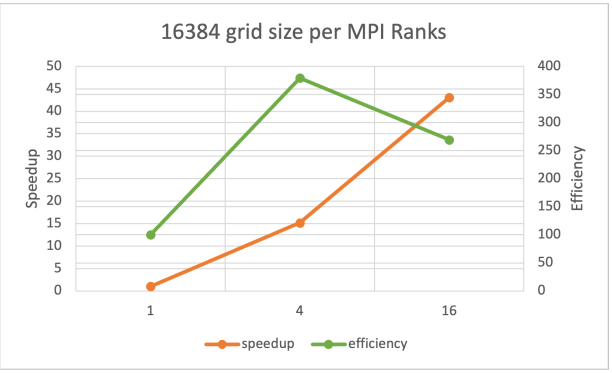
3. In the Figure 2c 32768 points per MPI Rank by running 256x256 with 2 MPI Ranks, 512x512 with 8 MPI Ranks and 1024x1024 with 32 threads is calculated.

4. In the Figure 2d 65536 points per MPI Rank by running 256x256 with 1 MPI Ranks, 512x512 with 4 MPI Ranks and 1024x1024 with 16 MPI Ranks is calculated.

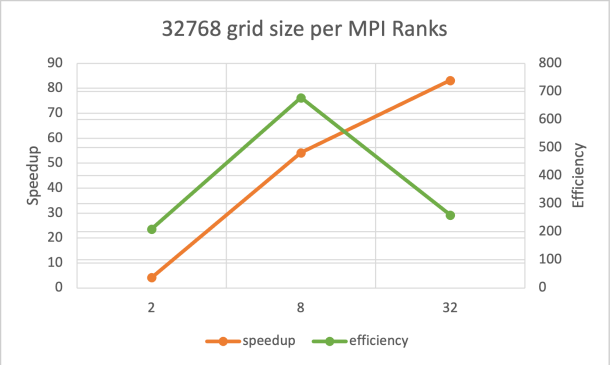
- The parallelization seems effective for smaller grid sizes, but for larger grid sizes, there is a decrease in efficiency with a higher number of MPI ranks, possibly due to increased communication overhead. The choice of the number of MPI ranks and grid size would depend on the specific characteristics of the problem and the computing resources available.



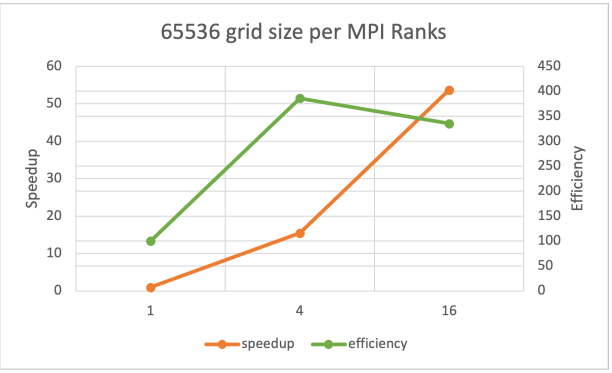
(a) 2048 grid size per MPI Rank



(b) 4096 grid size per MPI Rank



(c) 8192 grid size per MPI Rank



(d) 16384 grid size per MPI Rank

Figure 2: Weak scaling result

2. Python for High-Performance Computing (HPC) [in total 50 points]

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

Summing ranks using all-lowercase method `MPI.Comm.reduce` and uppercase method `MPI.Comm.Reduce` are both implemented in the file `hpc-python/sum_of_ranks.py`. Two methods produce same results.

2.2. Domain decomposition: Create a Cartesian topology [5 Points]

In the file `hpc-python/domain_decomposition.py`,

- Size of grid is determined by `MPI.Comm.Create_cart`.
- The Cartesian topology is created using `MPI.Comm.Create_cart` and the coordinates of process is given by `MPI.CartComm.Get_coords`.
- The ranks of neighboring processes are determined using `MPI.CartComm.Shift`.

2.3. Exchange rank with neighbours [5 Points]

The exchange of ranks with neighbors are implemented in the file `hpc-python/domain_decomposition.py` using `MPI.Comm.sendrecv` to avoid deadlock. The result is verified using `assert` statement which compares the received rank from neighbors with the rank of neighbors computed in the previous task.

2.4. Change linear algebra functions [5 Points]

- The collective operation `MPI.Comm.Allreduce` which gathers local results is used in `hpc_dot` and `hpc_norm` to compute the dot product and the norm, respectively.

- The local result is computed by `numpy.dot`. On 128×128 grid, it speeds up compared to the `for`-loop implementation. There're multiple reasons for this speedup. The Numpy library is fully optimized on scientific computing and uses vectorized operations in `numpy.dot`, while Python doesn't impose automatic vectorization on `for`-loop.
- Numpy, a Python library, was used many times in this file. The dot product is computed locally and can be shared and disseminated to multiple ranks. The vector must be dispersed throughout all ranks because all of the elements must communicate since the entries of the input vector are needed to build the vector that will be outputted.

2.5. Exchange ghost cells [5 Points]

In `data.py`,

- The exchange of ghost cells is implemented in a similar way as in the C/C++ version. The requests for `MPI.Comm.Irecv` is stored in object field `_reqs`.
- In `exchang_startall`, copying data to the send buffer uses `self._inner`. This method is faster than `self._buff = array_to_copy.copy()` which needs memory allocation operation.
- The results, including output images and the number of iterations, are similar to those of the C/C++ implementation.

2.6. Scaling experiments [5 Points]

2.6.1. Settings

The settings of experiments are identical to those of the C/C++ implementation.

2.6.2. Output

The output image is shown in figure 3.

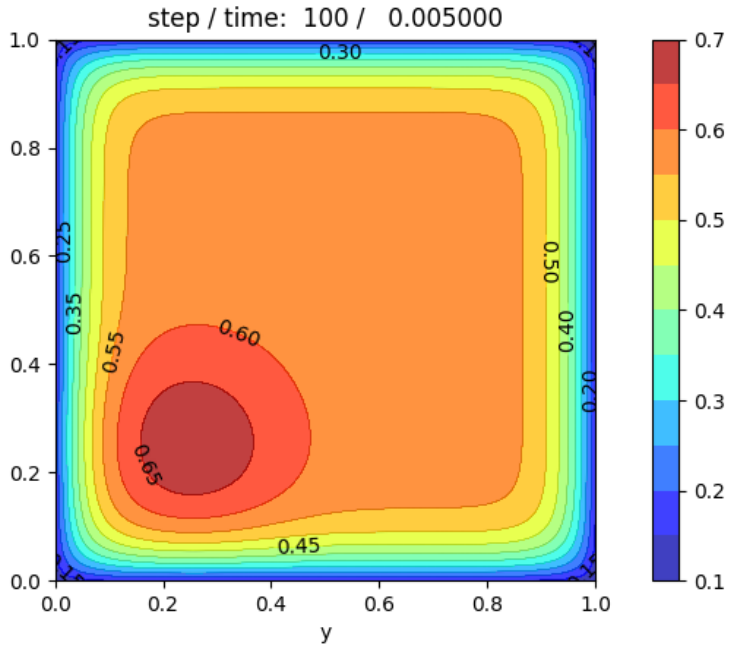


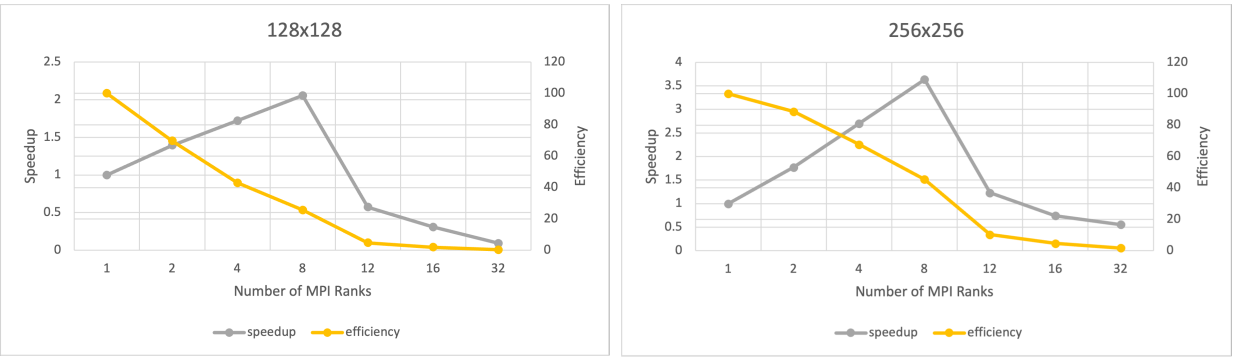
Figure 3: Output of Python version on 256×256 grid, 100 time steps and simulation time 0-0.005s.

2.6.3. Performance Analysis: Strong Scaling

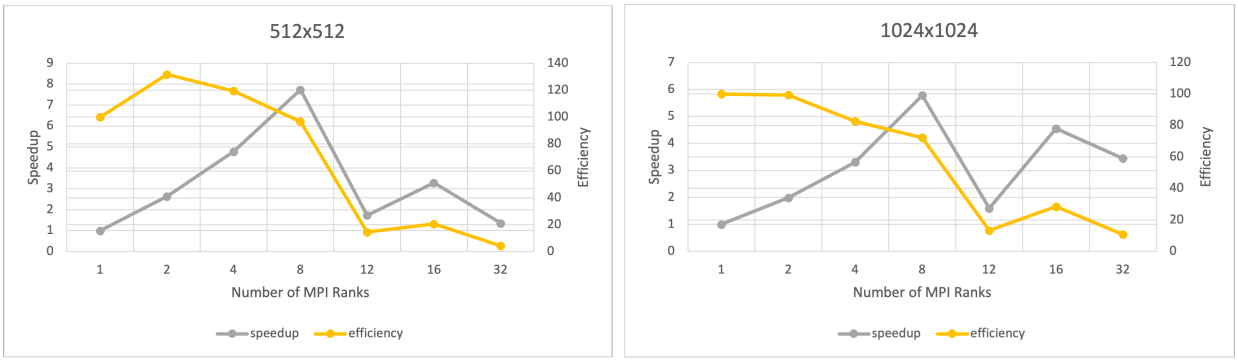
The strong scaling results (100 time steps and simulation time 0-0.005s) are shown in figures of 1 This section focuses on the difference between the C/C++ version and the Python version.

Comparison The Python version is significantly slower. It can be explained as follows.

- Python is an interpreted language and not compiled. The C/C++ compiler can do many tricks when compiling, but it's restricted in Python. To make things worse, Python has dynamic types, which means it does expensive type inference at runtime and any optimization possible on static code functionality is harder to impose[**?**].
- When doing operations like `A * B + C` with `numpy` arrays, it first computes `A * B` and stores the data in a temporary array, then add this temporary array to `C`. In C/C++ version this computation can be done elementwise, which does not need to allocate memory for the intermediate result and enjoys better temporal locality since only registers are used. The function `diffusion` has many this kind of operations and suffers a lot from it.



(a) 128 × 128 grid (b) 256 × 256 grid

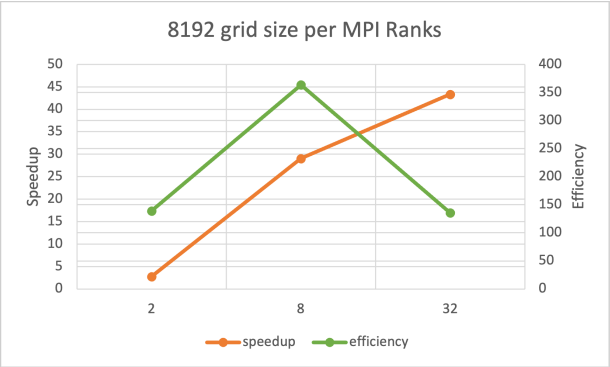


(c) 512 × 512 grid (d) 1024 × 1024 grid

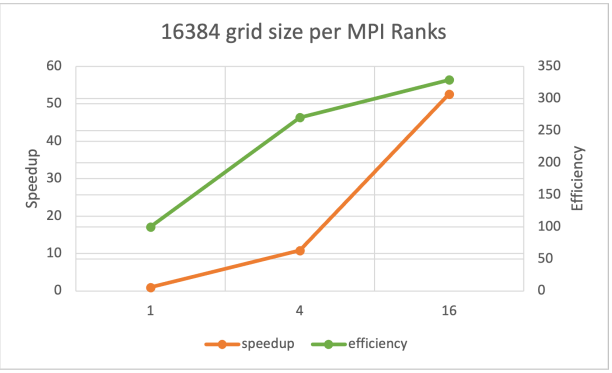
Figure 4: Strong scaling result

2.6.4. Performance Analysis: Weak Scaling

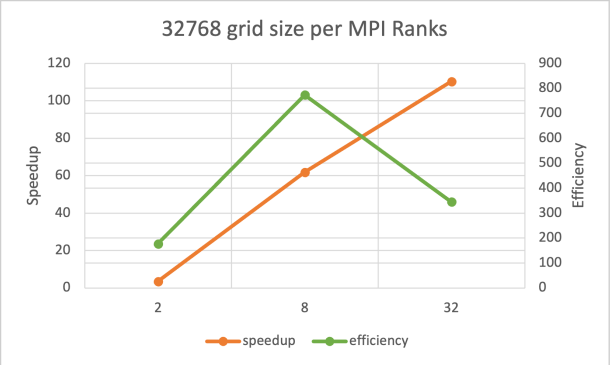
The weak scaling results (100-time steps and simulation time 0-0.005s) are shown in the figures of 2 which are also similar with those of the C/C++ version.



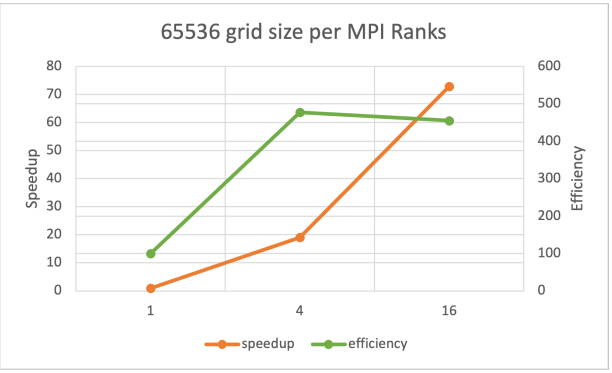
(a) 2048 grid size per MPI Rank



(b) 4096 grid size per MPI Rank



(c) 8192 grid size per MPI Rank



(d) 16384 grid size per MPI Rank

Figure 5: Weak scaling result

2.7. A self-scheduling example: Parallel Mandelbrot [20 Points]

2.7.1. Implementation

Suppose the number of workers is p and their ranks are $\{1, \cdots, p\}$.

Manager

1. Send `tasks[i]` to worker i with tag `TAG_TASK`, $i = 1, \cdots, p$.
2. Receive a finished task from worker and get the rank k of worker using `MPI.Status.Get_rank`.
3. If there's any unfinished task, send it to worker k with tag `TAG_TASK`; otherwise send `None` to worker k with tag `TAG_DONE`.
4. Repeat step 2 until all finished tasks are received.

Worker

1. Receive task from manager and get the tag of the task using `MPI.Status.Get_tag`.
2. If the tag is `TAG_DONE`, break; otherwise finish the task and send it back to manager with tag `TAG_TASK_DONE`.
3. Repeat step 1.

Appending new tasks to `list` object may lead to memory allocation and data movement, which should be expensive. Two methods, updating the original list `tasks` with finished tasks in place or creating new list `tasks_done` for finished tasks, are tested. However, they show no difference in performance. It implies that the cost of `list` appends is negligible compared to the computation time of doing tasks. Using blocking `MPI.Comm.send` and nonblocking `MPI.Comm.isend` also show no difference due to similar reason.

2.7.2. Output

The output image is shown in figure 6.

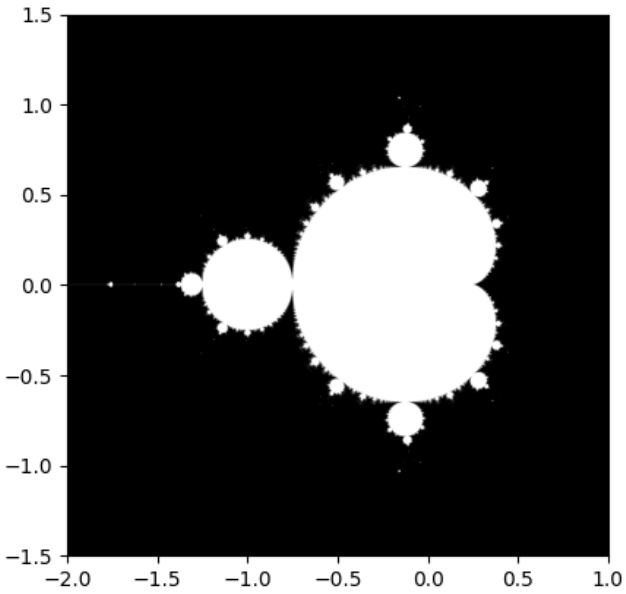


Figure 6: Output of parallel Mandelbrot computation on 4001×4001 grid decomposed into 50 tasks.

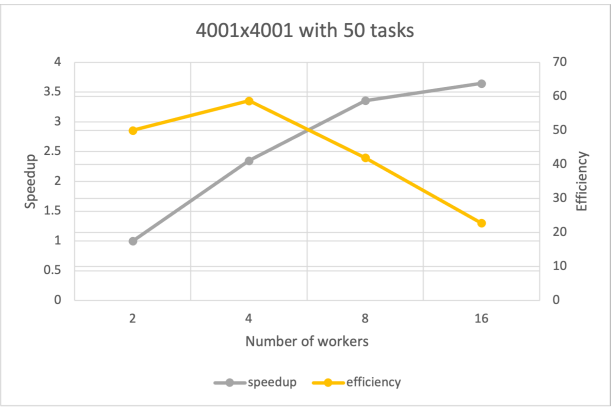
2.7.3. Strong scaling

Settings The experiments run on Apple M2 Pro with 12 cores (8 of them are for high performance, 4 of them are for energy-efficient). Jobs were submitted as requested.

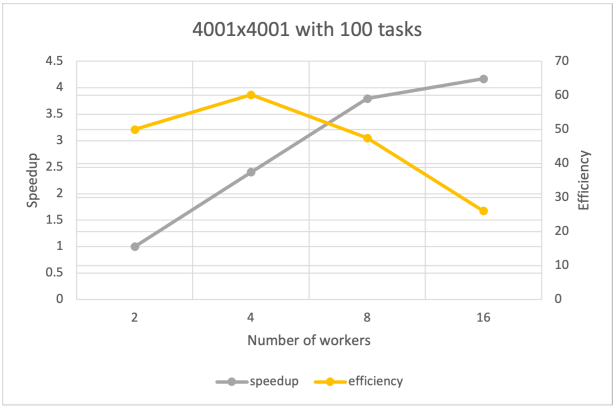
Results The strong scaling results of grid 4001×4001 are shown in figure 7.

Analysis

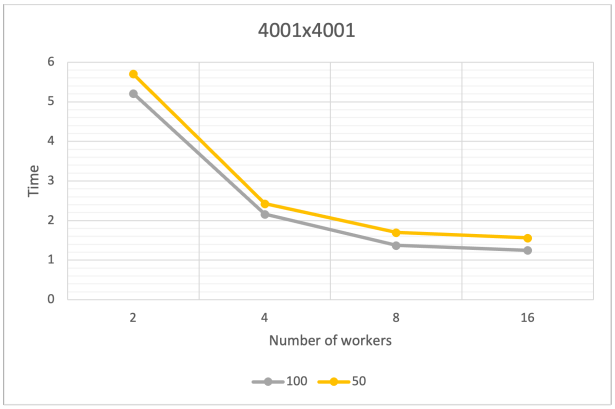
- The diminishing efficiency as the number of processes increases suggests that there might be overhead in managing the parallel processes, and the workload may not be large enough to fully utilize all processes efficiently. This is a common phenomenon in parallel computing known as Amdahl's Law.
- The graphics clearly show that the more personnel available, the faster the project would be completed. The difference in processing time between 50 and 100 jobs is not significant, indicating that workers' duties are being shared well enough to complete numerous tasks in a short period of time. The scenario with 100 tasks scales better than the other.
- In summary, the results show that parallelization provides speedup, but there's a diminishing return as the number of processes increases. This information can be valuable for optimizing the parallelization strategy for this specific workload.



(a) 50 Tasks



(b) 100 Tasks



(c) Time Comparison of 50 and 100 Tasks

Figure 7: Strong scaling result of parallel Mandelbrot computation.

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

Each project will have 100 points (out of which 15 points will be given to the general quality of the written 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;
  - your write-up with your name project_number_lastname_firstname.pdf.
- Submit your .tgz through Icorsi.