Università della Svizzera italiana

Institute of Computing
CI

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

Student: Alessio Barletta Discussed with: Pablo Landrove, Daniele Buser

Solution for Project 4

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.

1. Ring sum using MPI [10 Points]

In this task, we are asked to implement a ring sum operation using MPI. The most important implementation aspect is avoiding risk of deadlock. The scenario that we want to avoid is the following:

- Every process enters `MPI_Send` (which is blocking) at the same time, they block waiting for their respective destination processes to receive the data.
- Since every destination process is also currently stuck in a send operation, no one can initiate a `MPI_Recv` operation, leading to a deadlock

To avid this deadlock, I used `MPI_Sendrecv`, which combines sending and receiving as a single atomic operation. This way, each process can send its data while simultaneously receiving data from its neighbor, ensuring that no process is left waiting indefinitely.

What my algorithm does is sending its current value to the next process in the ring and receiving the value from the previous process, then summing the received value to its local sum. Repeating this for the number of processes leads to every process having the sum of all ranks.

2. Cartesian domain decomposition and ghost cell exchange [15 Points]

2.1. Cartesian two-dimensional MPI communicator

To create a 4×4 periodic Cartesian communicator, I used the `MPI_Cart_create` function. The dimensions are set to 4 in both directions, to allow periodicity, I set the periods array to `{1, 1}`, meaning that both dimensions wrap around.

2.2. Derived data type

This data type is used to send and receive columns of the 2D domain without manual packing. Is needed because, unlike the rows, the columns are not contiguous in memory, so we create a derived datatype using `MPI_Type_vector`. This function allows us to define a new MPI datatype that represents a column, so we don't need to do a send/receive for each element of the column or manually pack/unpack the data into a contiguous buffer.

2.3. Exchange ghost cell with the neighboring cells

To exchange ghost cells with the four cardinal neighbors (N, S, E, W), I used `MPI_Cart_shift` to get the shifted source and destination ranks, given a shift direction and amount. Then, I used `MPI_Sendrecv` to simultaneously send and receive the ghost cell data, as did for the previous task to avoid deadlocks. Specifically, I send to the neighbor in one direction and receive from the neighbor in the opposite direction. For example, to exchange data with the north and south neighbors:

```
1 MPI_Sendrecv(&data[1*DOMAINSIZE +1], SUBDOMAIN, MPI_DOUBLE, rank_top, 0,
2              &data[(DOMAINSIZE-1)*DOMAINSIZE +1], SUBDOMAIN, MPI_DOUBLE,
                  rank_bottom, 0,
3              comm_cart, &status);
```

2.4. Diagonal Neighbor Exchange Bouns [10 Points]

Here we first have to compute their coordinates in the Cartesian grid, then convert those coordinates to rank IDs:

```
1 int my_coords[2], ne_coords[2];
2 MPI_Cart_get(comm_cart, 2, dims, periods, my_coords);
3 ...
4 // to my top and right
5 ne_coords[0] = (my_coords[0] - 1 + dims[0]) % dims[0];
6 ne_coords[1] = (my_coords[1] + 1) % dims[1];
7 MPI_Cart_rank(comm_cart, ne_coords, &rank_ne);
```

- `MPI_Cart_get` Retrieves Cartesian topology information associated with a communicator and outputs the coordinates of the current process.
- `MPI_Cart_rank` Converts the Cartesian coordinates of a process into its corresponding rank in the communicator.

As before, we use `MPI_Sendrecv` to exchange the ghost cells with the diagonal neighbors, I send the SW corner to the NE neighbor and receive the NE corner from the SW neighbor, and similarly for the other diagonal directions.

2.5. Output

The Output matrix for rank 9 after ghost cell exchange (including values in ordinal directions) is shown in the table below:

4.0	5.0	5.0	5.0	5.0	5.0	5.0	6.0
8.0	9.0	9.0	9.0	9.0	9.0	9.0	10.0
8.0	9.0	9.0	9.0	9.0	9.0	9.0	10.0
8.0	9.0	9.0	9.0	9.0	9.0	9.0	10.0
8.0	9.0	9.0	9.0	9.0	9.0	9.0	10.0
8.0	9.0	9.0	9.0	9.0	9.0	9.0	10.0
8.0	9.0	9.0	9.0	9.0	9.0	9.0	10.0
12.0	13.0	13.0	13.0	13.0	13.0	13.0	14.0

Table 1: Output matrix for rank 9 after ghost cell exchange (including ordinal directions)

3. Task: Parallelizing the Mandelbrot set using MPI [20 Points]

3.1. Create the partitioning of the image

To create the partitioning of the image, I have completed the `createPartition` method using `MPI_Dims_create` to partition the number of processes in balanced division across the Cartesian grid.

I then implemented the `updatePartition` function, so that each process can update the partition to represent its mpi_rank.

3.2. Determine the dimensions and the start/end of the local domain

We can determine the dimension of the local domain by dividing the desired width of the image by the number of processes. Then we need to compute the start coordinates for the first pixel to compute in the local domain, this can be done by multiplying the local domain size by its start index. Lastly to determine the index of the last pixel in the local domain we sum to the start index the size of the local domain.

3.3. Send local domain to master process

The master process ($mpi_{rank} == 0$) is responsible for receiving the computed local domains from all processes and plotting each partition in the final image. Each process sends its local data to the master process using `MPI_Send`. The master process uses `MPI_Recv` to receive the data and then plots the corresponding portion of the final image.

3.4. Observed performance

The graph below shows the execution time for different image sizes and number of processes.

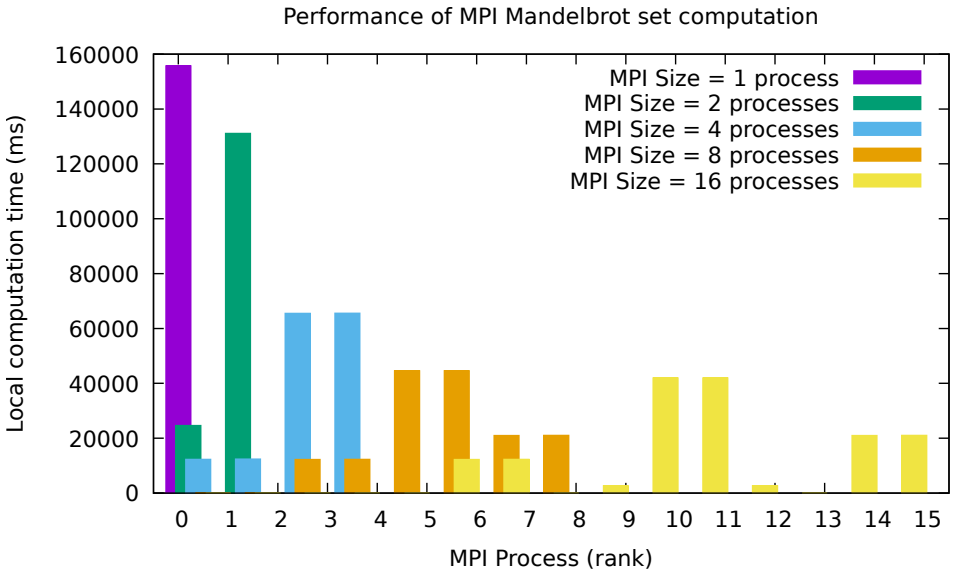

Figure 1: Mandelbrot Set Performance with different process counts

As we can see from the plot, some processes take much longer to compute compared to others. This is due to the fact that the Mandelbrot set has regions that are more computationally intensive, notably around the boundary of the set. Processes assigned to these regions will take longer to compute, leading to load imbalance.

Looking at the documentation, we find that the way the function `MPI_Dims_create` creates the partitioning is by trying to set the dimensions of the grid to be as close to each other as possible, essentially trying to create a distribution of processes that is as balanced as possible across all directions. Following this logic, the current implementation creates a grid of 4x4 processes. The following image shows the partitioning created for 16 processes:

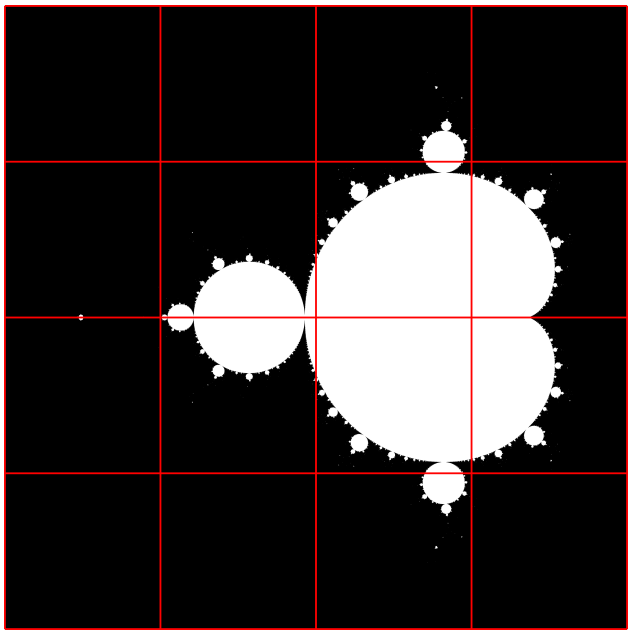

Figure 2: Partitioning created for 16 processes

Looking at the image above, we can see that some processes are assigned regions that are more computationally intensive than others, leading to load imbalance. For example, we might expect processes that are assigned to the border regions of the Mandelbrot set to take longer to compute compared to the others. This is consistent with what we observe in the performance graph.

One way to tackle this issue could be to divide in image in a way that each process gets a mix of both computationally intensive and less intensive regions, for example by using a block-cyclic distribution [1].

- Cyclic distribution can help in balancing the load among processes, as each process would get a fair share of both easy and hard-to-compute pixels, but it would also increase the communication overhead, as processes would need to communicate more frequently to gather their assigned pixels.
- Block distribution corresponds to the current implementation, where each process gets a contiguous block of pixels. This approach minimizes communication overhead, as each process can compute its assigned block independently, however as we have seen, it can lead to load imbalance, due to some blocks may be more computationally intensive than others.

Block-cyclic distribution is a hybrid approach that combines the benefits of both block and cyclic distributions. Here, the image is divided into smaller blocks, and these blocks are distributed cyclically among the processes.

4. Task: Parallel matrix-vector multiplication and the power method [40 Points]

In this section we are asked to modify the current implementation of the power method to use MPI. Regarding the implementation, the most important changes were the partitioning of the work, the use of a local array for each process and the subsequent gathering of the results. In particular, for this last part the use of `MPI_Allgatherv` was necessary to obtain the correct result, since if the original domain size is not perfectly divisible by the number of processes, each process may have a different number of elements to compute and subsequently gather. To do this I also had to create the arrays `displ[size]` and `recvcounts[size]`, the first one to indicate the displacement of each process in the final gathered array, the second one to indicate how many elements each process will send.

4.1. Strong Scaling

The plots for the strong scaling speedup, efficiency and runtime are shown below:

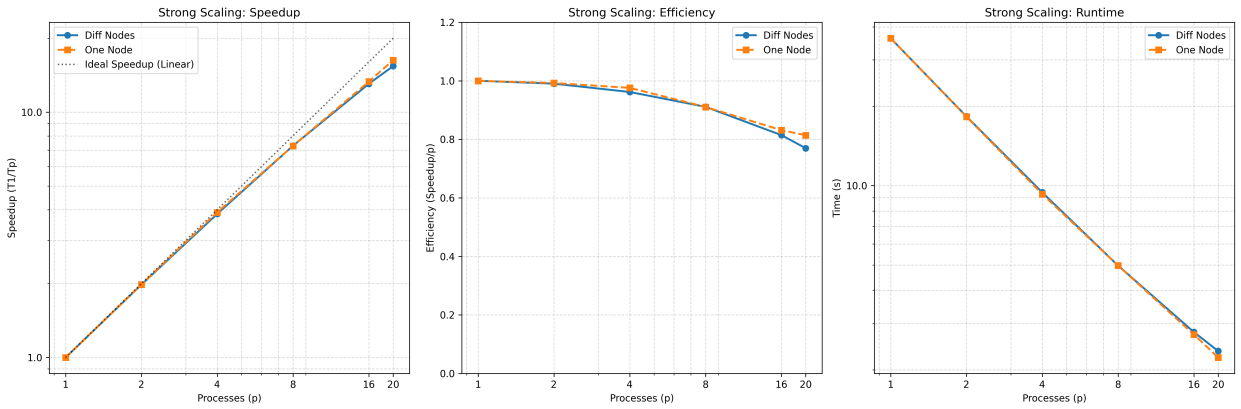

Figure 3: Strong Scaling for Power Method with different matrix sizes

As usual, the speedup is defined as

$$S_p = \frac{T_1}{T_p} \tag{1}$$

While the efficiency is just the speedup divided by the number of processes:

$$E_p = \frac{S_p}{p} = \frac{T_1}{p \cdot T_p} \tag{2}$$

This is because we are interested in seeing how well the algorithm scales with increasing number of processes, an efficiency close to 1 indicates that the algorithm scales linearly with the number of processes, example doubling the number of processes halves the execution time.

From the plots, we can see that for smaller matrix sizes (10000x10000), the efficiency remains high but decreases slightly as the number of processes increases. This decline occurs because, as the workload per processor shrinks, the overhead of communication and the serial parts of the code (which remain constant regardless of the number of processes) become more significant relative to the computation time, preventing perfectly linear scaling.

Moreover, we can observe a very slight difference between one node and multiple nodes, this might be caused by the slightly longer communication time when using multiple nodes, as data has to travel over the network rather than just within the same node's memory.

4.2. Weak scaling

The plots for the weak scaling speedup, efficiency and runtime are shown below:

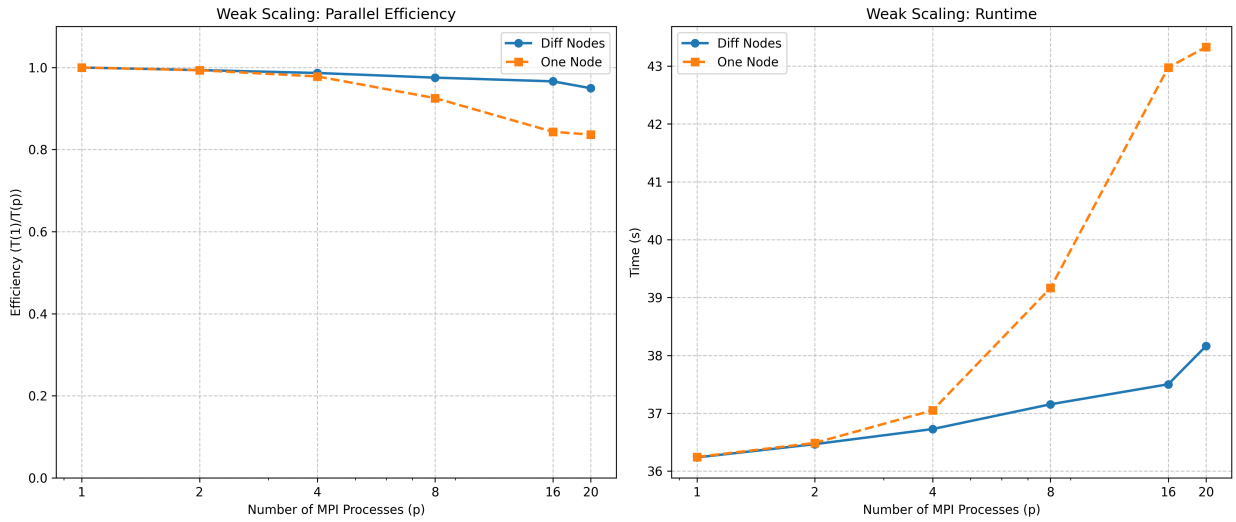

Figure 4: Weak Scaling for Power Method with different matrix sizes

In weak scaling, we keep the workload per processor constant as we increase the number of processors. Efficency is defined as:

$$E_p = \frac{T_1}{T_p} \tag{3}$$

Here, an efficiency close to 1 indicates that the algorithm can handle larger problems effectively as more processors are added, in other words if we double the number of processors and increase the problem size accordingly (proportional to $\sqrt{p}$), the execution time remains roughly the same.

We can see that as the number of processes increases, especially after 8, the efficiency starts to drop more significantly. This is likely due to the fact that on a single node on Rosa we have 2 processors each with 10 physical cores (20 total), so when we get close to this number, the memory bus and the L3 cache (that is shared among all cores of a single processor) start to become saturated leading to a decrease in performance. This does not happen when using multiple nodes, as each node has its own memory bus and cache, allowing for better scalability, suggesting that the overhead of slightly higher communication time when using multiple nodes is offset by the improved memory bandwidth and cache availability.

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

References

[1] P. Arbenz, *Parallel Numerical Computing*, Lecture 6, ETH Zürich. Available at: `https://people.inf.ethz.ch/arbenz/PARCO/6.pdf`.