Università della Svizzera italiana | Institute of Computing CI

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

Student: Dennys Mike Huber Discussed with: Alberto Finardi & Leon Ackermann

Solution for Project 4

1. Task: Ring maximum using MPI [10 Points]

There are multiple possible solutions to solve this problem without having a deadlock. I chose to separate my processes into two disjointed groups based on their rank id. More precisely into even and odd ranks, where they alternate between sending and receiving and therefore preventing deadlocks. In this implementation even rank ids send in a first step and then receive in a second step, the odd rank ids do the opposite. For an even number of total processes this works because the highest rank id is odd and therefore for a given process all its neighbors are sending while the particular process is receiving and vice versa. When having an odd number of total processes, this results in the highest rank id being even and wanting to send their content to rank 0. Even though both processes are sending at the same time, based on the given topology we know that process 0 will first send it contents to an odd rank process and is then able to receive the contents from the highest rank process.

```
int sum = rank;
int rec_rank = rank;
int send_rank = rank;
for (int i = 0; i < size - 1; i++) {
  if (rank % 2 == 0) {
    MPI_Send(&send_rank, 1, MPI_INT, (rank + 1) % size, tag, MPI_COMM_WORLD);
    MPI_Recv(&rec_rank, 1, MPI_INT, (rank - 1 + size) % size, tag,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(&rec_rank, 1, MPI_INT, (rank - 1 + size) % size, tag,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&send_rank, 1, MPI_INT, (rank + 1) % size, tag, MPI_COMM_WORLD);
  }
  sum += rec_rank;
  send_rank = rec_rank;
}
```

Listing 1: Ring topology using alternating send and receive

2. Task: Ghost cells exchange between neighboring processes [15 Points]

For this exercise we create a Cartesian communicator with periodic boundaries, in order to find the neighboring ranks using the `MPI_Cart_Shift` function. After finding all four neighbors we do four

MPI_Sendrecv executions, where each processes, for example receives the values from the southern neighbor and sends its values to the northern neighbor, which is then repeated for each direction. Note that this is possible because we have periodic boundaries which means every processes has a neighbor in every direction.

```c
// Create cartesian topology
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_cart);

// Finding neighbors
MPI_Cart_shift(comm_cart, 0, 1, &rank_top, &rank_bottom);
MPI_Cart_shift(comm_cart, 1, 1, &rank_left, &rank_right);

// Send values to northern neighbor and receive from southern neighbor
MPI_Sendrecv(&data[1 * DOMAINSIZE + 1], 1, T_row, rank_top, 0,
       &data[(SUBDOMAIN + 1) * DOMAINSIZE + 1], 1, T_row, rank_bottom,
       0, comm_cart, MPI_STATUS_IGNORE);

/* REPEATED FOR EACH DIRECTION */
```

Listing 2: Example

In order to simply the communications, we introduce two new MPI data types. For the row we define a MPI_Type_contiguous, which is simply for values that are stored in a contiguous location. For the columns we define MPI_Type_vector data types, which allows us to define a stride and therefore columns in a contiguous stored matrix.

```c
MPI_Datatype T_row, T_col;
MPI_Type_contiguous(SUBDOMAIN, MPI_DOUBLE, &T_row);
MPI_Type_vector(SUBDOMAIN, 1, DOMAINSIZE, MPI_DOUBLE, &T_col);
MPI_Type_commit(&T_col);
MPI_Type_commit(&T_row);
```

Listing 3: Example

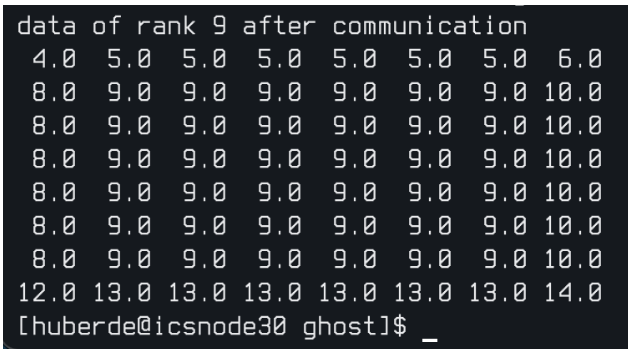
For the exchange of ghost cells of the neighbors in ordinal direction, we use the MPI_Cart_coords function, which returns the coordinates of a process in a communicator of a cartesian topology. By adding $+1$ or -1 to its own x and y coordinates, a process can figure out its ordinal neighbors as seen in Listing 4. Finally we do a send and receive similarly as before, just in ordinal direction. For example sending the value to the northeast and receiving the values from southwest.

```c
int top_right_rank, top_left_rank, bottom_right_rank, bottom_left_rank;
int top_right_coords[2] = {coords[0] - 1, coords[1] + 1};
MPI_Cart_rank(comm_cart, top_right_coords, &top_right_rank);
int top_left_coords[2] = {coords[0] - 1, coords[1] - 1};
MPI_Cart_rank(comm_cart, top_left_coords, &top_left_rank);
int bottom_right_coords[2] = {coords[0] + 1, coords[1] + 1};
MPI_Cart_rank(comm_cart, bottom_right_coords, &bottom_right_rank);
int bottom_left_coords[2] = {coords[0] + 1, coords[1] - 1};
MPI_Cart_rank(comm_cart, bottom_left_coords, &bottom_left_rank);

// Send top right and receive from bottom left
MPI_Sendrecv(&data[1 * DOMAINSIZE + (DOMAINSIZE - 2)], 1, MPI_DOUBLE,
               top_right_rank, 0, &data[(DOMAINSIZE - 1) * DOMAINSIZE + 0], 1,
               MPI_DOUBLE, bottom_left_rank, 0, comm_cart, MPI_STATUS_IGNORE);

/* REPEATED FOR EVERY ORDINAL DIRECTION */
```

Listing 4: Example



```
data of rank 9 after communication
 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
[huberde@icsnode30 ghost]$ _
```

Figure 1: Result for process with Rank 9

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

In 2 you can see the performance of the Mandelbrot set spread out over different number of processes. This shows how much work each thread has to do depending on the number of processes. If we have more than two processes we can clearly see that the workload is unevenly distributed across the processes. This imbalance is caused by the fact that different regions of the complex plane need varying numbers of iterations to determine if a point is part of the mandelbrot set. For higher number of processes we can see that workload is distributed more evenly, yet there are still processes that do very little compared to others.

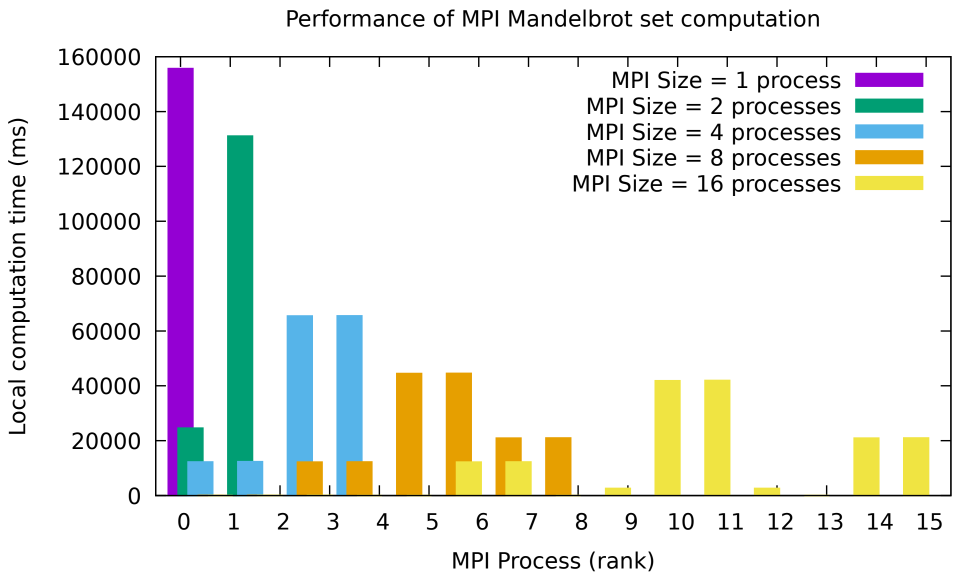


Figure 2: MPI Performance of the Mandelbrot on the Rosa Cluster for 1 to 16 Processes

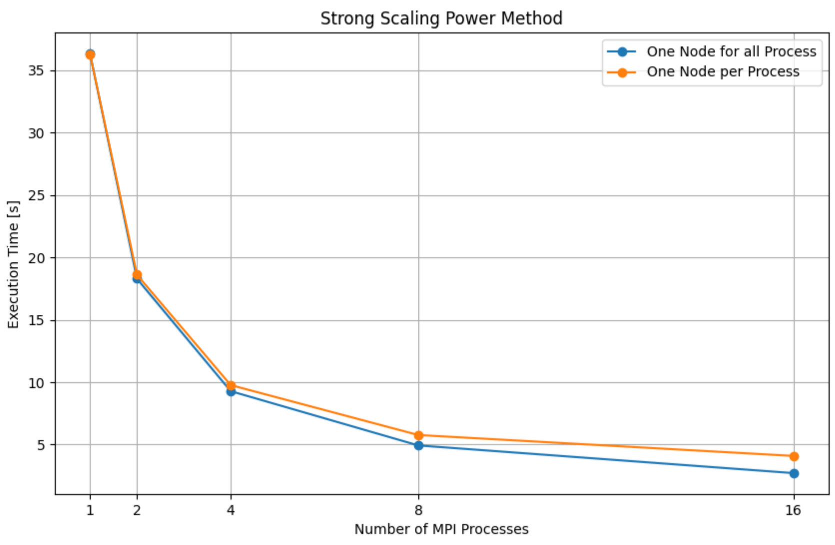
In order to improve the performance a different partitioning scheme with different sized grids could be used. I would make the number of grids higher in computational demanding areas and as a result decrease the size of this grids in said area. Furthermore the grids would be bigger for less computational demanding areas. This would result in an improved workload balanced across all the processes.

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

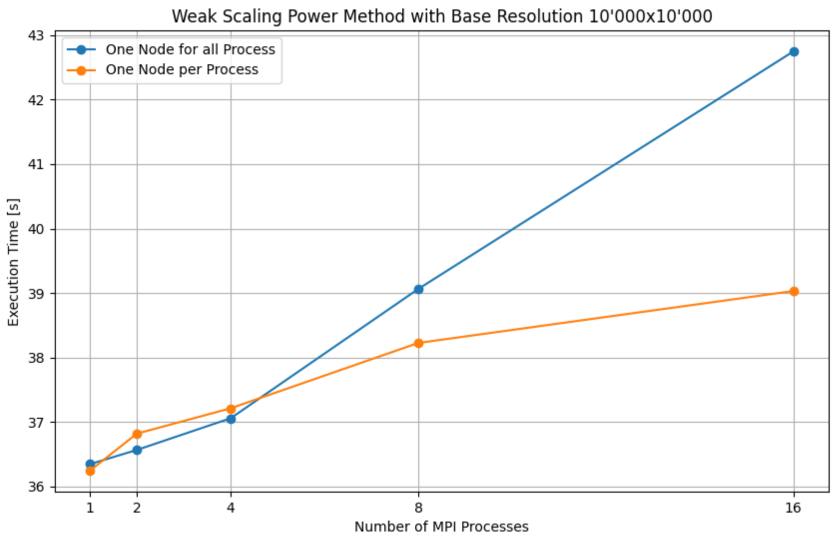
For both, strong and weak scaling Test case 3 was used where $A(i, j) = i$ if $i == j$ for $1 <= i,\ j <= n$ and the eigenvalue should converge to $\theta = n$, the dimension of the quadratic matrix.

Furthermore the iteration were capped to 300 iterations and no tolerance limit was set, therefore in the case of early converges, the calculation would continue. For the strong scaling we used the fixed matrix size of $n = 10000$. This size was also used as the base matrix size for the weak scaling version, which was then multiplied by $\sqrt{p}$, where p is the number of MPI processes.

When comparing the execution time between running the MPI processes on a single node or distributed across multiple nodes, in the strong scaling case, seen in Subfigure 3a, we can see that running the power method on a single or multiple are basically identical. There's a strong increase in performance for a low number of processes, which then later becomes less when the overhead increases. It's slightly diverging for higher number of processes we can be caused by other users running their program on the server or the communication overhead that needs to travel a larger distance.



(a) Strong Scaling



(b) Weak Scaling

Figure 3: Scaling Plots of the Power Methods

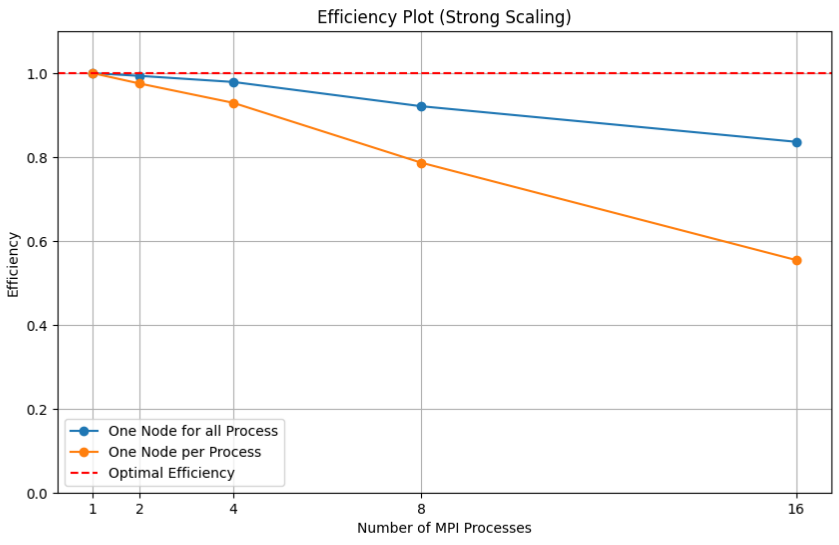
In the weak scaling case in Subfigure 3b where we increase the matrix size proportionally with the number of processes. Here the optimal case would be that the execution time stays constant, but we can clearly see that single node version, scales significantly worse, compared to running the power method on multiple nodes. This caused that more and more resources are contested on a single node, compared to running the program on multiple nodes. This impact can be especially felt with a higher number of processes, where by definition the matrix size is significantly bigger. Hence the execution time on a single node increase a lot stronger on a single node.

Looking at the efficiency plots (4) they paint a similar picture than the previous plots. For the strong scaling case, the efficiency is given by

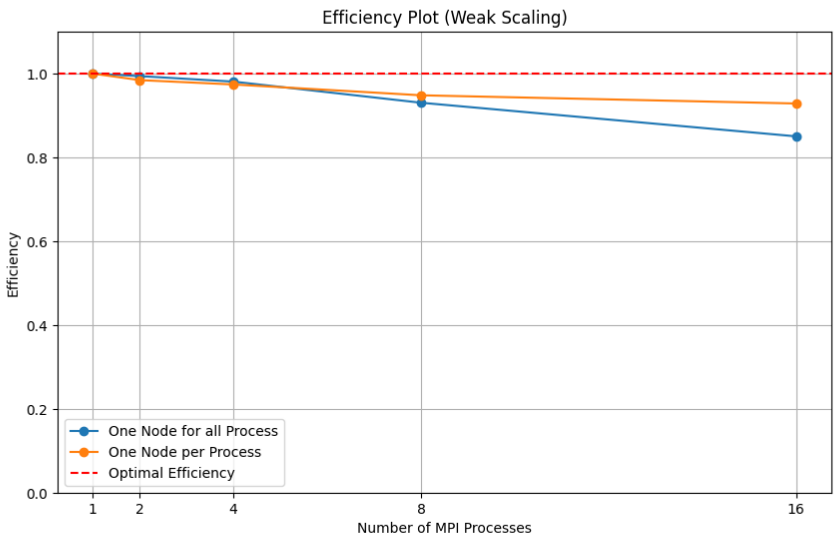
$$E_S = \frac{T_1}{T_p \cdot p} \tag{1}$$

For the weak scaling the efficiency formula slight differs and is given as follows

$$E_W = \frac{T_1}{T_p} \tag{2}$$



(a) Strong scaling



(b) Weak Scaling

Figure 4: Efficiency Plots for the Weak and Strong Scaling case

In the strong scaling case in Subfigure 4a, for smaller number of processes the efficiency is relatively high and then decreases for higher number of processes due to the overhead by using more processes. The efficiency drops strong for the power method run on multiple nodes compared to the one run on a single node, this again highlights the increased overhead created when communication between multiple nodes.

For the weak scaling case in Subfigure 4b, due to the proportionally increased workload per number of processes the efficiency is clearly better than in Strong scaling case. Due to increasing size of the matrix the overhead to workload ratio is lower, resulting in higher efficiency. Comparing the single to the multiple node, we can see that the power method run on multiple nodes is slightly more efficient, especially for a higher number of processes.