Università della Svizzera italiana

Institute of Computing
CI

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

Student: Kaan Sen Discussed with: Lorenzo Varese

Solution for Project 4

1. Ring maximum using MPI [10 Points]

The ranks of the neighbors are calculated the following way, where my rank is the rank of the current process, and size is the total number of processes in the communication.

```
right = (my_rank+1)%size;/* get rank of neighbor to your right */
left  = (my_rank-1)%size;/* get rank of neighbor to your left */
```

And the Ring maximum is as follows:

```
    max = (my_rank*3) % (size*2);
    for (i = 0; i < size; i++)
{
        snd_buf = max;
        MPI_Issend(&snd_buf, 1, MPI_INT, right, tag, MPI_COMM_WORLD, &request);
        MPI_Recv(&rcv_buf, 1, MPI_INT, left, tag, MPI_COMM_WORLD, &status);
        MPI_Wait(&request, &status);
        max = (max < rcv_buf) ? rcv_buf : max;
}
```

To determine the maximum value within a ring of processes, each process first computes its own value and stores it in a variable called "max." Subsequently, this "max" value is transmitted to the neighboring process on the right, which receives a value from the process on the left. The received value is then compared to the local "max" value, and the larger of the two values becomes the new "max" for that process. This process is repeated a total of "size" times, ensuring that all processes eventually converge to the same global maximum value. When executed with four processes, the result is a unified maximum value for the entire system.

the output of the code can be seen below :

```
Process 0:      Max = 6
Process 1:      Max = 6
Process 2:      Max = 6
Process 3:      Max = 6
```

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

For this part, only 16 processes are used. So, the dimensions of the processor grid and periodic boundaries in both dimensions are set as follows:

```
    dims[0] = 4;
    dims[1] = 4;
    periods[0] = 1;
    periods[1] = 1;
```

The cartesian communicator and shifting can be seen below :

```
     MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm_cart);
     MPI_Cart_shift(comm_cart, 1, 1, &rank_left, &rank_right);
    MPI_Cart_shift(comm_cart, 0, 1, &rank_top, &rank_bottom);
```

Then created col and row data ghosts to not exceed the boundaries.

The cells are exchanged by using the blocking send call MPI Send(), the non-blocking receive MPI Irecv(), and the MPI Wait() function.

The result of the boundary exchange on rank 9 as expected:

```
9.0  5.0  5.0  5.0  5.0  5.0  5.0  9.0
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
9.0  13.0  13.0  13.0  13.0  13.0  13.0  9.0
```

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

In the file consts.h, the function createPartition completed as :

```
//  determine size of the grid of MPI processes (p.nx, p.ny), see MPI_Dims_create()
    int dims[]={0, 0};
    MPI_Dims_create(mpi_size, 2, dims);
    p.nx = dims[0];
    p.ny = dims[1];

//  Create cartesian communicator (p.comm)
    MPI_Comm comm_cart;
    int periods[] = {1, 1};
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm_cart);
    p.comm = comm_cart;

//  Determine the coordinates in the Cartesian grid (p.x, p.y), see MPI_Cart_coords()
    int my_coords[]={0, 0};
    MPI_Cart_coords(p.comm, mpi_rank, 2, my_coords);
    p.x=my_coords[0];
    p.y=my_coords[1];
```

the resulting image can be found below:

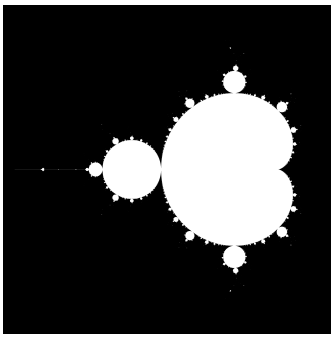


Figure 1: Mandelbrot parallel computation

The performance graph can be seen below :

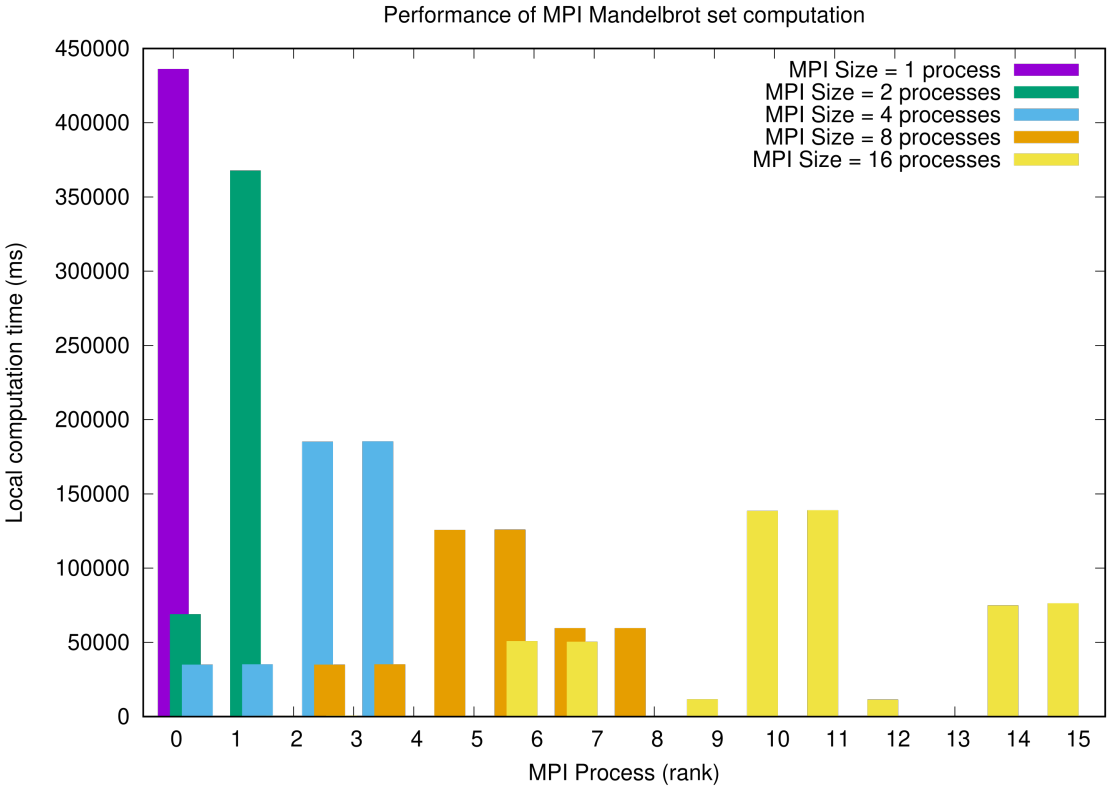


Figure 2: Computation time of each process for multiple run with different number of processes

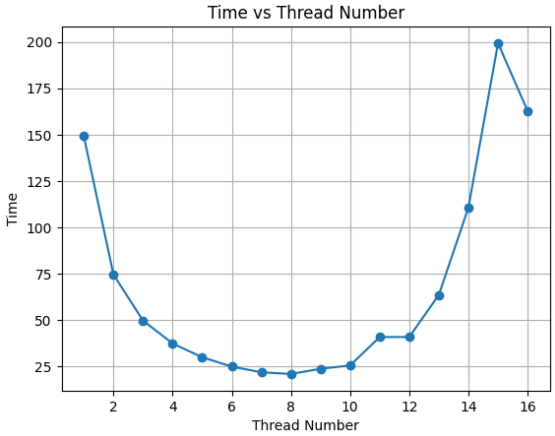
Even though performance is limited by the communication network between the nodes, MPI is significantly more efficient since it balances the load between multiple processes. As expected, the worst performance is for n = 1 and n = 2, since the number of processes is insufficient. For n = 4, the performance increases, and the total execution time is reduced due to the workload distribution between nodes. For n = 8, 16, the performance is the best since the load is balanced between a larger number of processes. But as we can see workload is not evenly distributed between each process it is because some processes stay in only the black zone which makes them easy to compute.

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

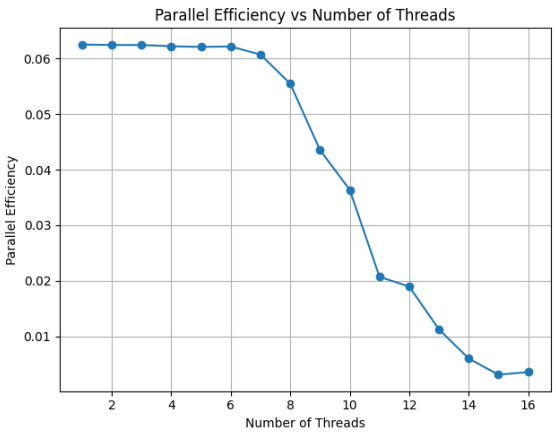
For this part, the power method given in the Matlab code is adapted to the project requirements. The program is run with various combinations of the number of processes and matrix sizes in order to do strong and weak scaling analysis. Necessary modifications can be obtained from the code.

4.1. Strong Scaling

For the strong scaling, the value n = 10, 000. The program is run on a different number of processes p = 1...16 execution time and efficiency plot can be found in the figures below:



(a) Execution time for matrix multiplication



(b) Parallel efficiency

Figure 3: Comparison of Execution Time and Parallel Efficiency

As expected, the solution scales well for small p. For p = 1...16, this can be easily observed by looking at the differences in execution time and the parallel efficiency. For larger p, the solution does not scale as well. this is because the local machine has only 8 performance cores, and it can be seen that after 8 execution time increases a high amount. Also when we check the parallel efficiency plot we can observe that after 8 process efficiency directly drops the reasoning of it also the same

4.2. Weak Scaling

To assess weak scaling performance, a meticulous examination was conducted by employing a single thread for the matrix multiplication operation on a 1000x1000 matrix. The evaluation systematically varied the number of threads while maintaining the matrix size constant, yielding an insightful analysis of the algorithm's scalability. The thread numbers and corresponding matrix sizes employed in this investigation are documented as follows:

Threads	Matrix size
1	1000x1000
2	2000x2000
3	3000x3000
4	4000x4000
5	5000x5000
6	6000x6000
7	7000x7000
8	8000x8000

The figure 4 illustrates a clear trend: as the number of threads and matrix size increase, so does the execution time. This relationship is as expected, as the benefits of parallelization diminish for smaller values of n n due to communication overhead in the network. Additionally, when the matrix size grows with a small number of threads (p), the application experiences a slowdown.

This pattern is a result of the complex interplay between computational and communication factors in parallel computing. The delicate balance between task distribution and communication overhead becomes more pronounced with smaller matrix sizes and thread configurations, influencing the overall efficiency of the algorithm.

These observations emphasize the importance of carefully choosing computational settings in parallel computing, taking into account the relationships between matrix size, thread count, and communication overhead. Such insights provide valuable guidance for optimizing the algorithm's performance in parallel processing scenarios.

The results of the weak scaling analysis show that the parallel matrix multiplication algorithm is efficient for large problem sizes, but that the parallel efficiency decreases as the matrix size increases. This is due to the increasing communication overhead in the network.

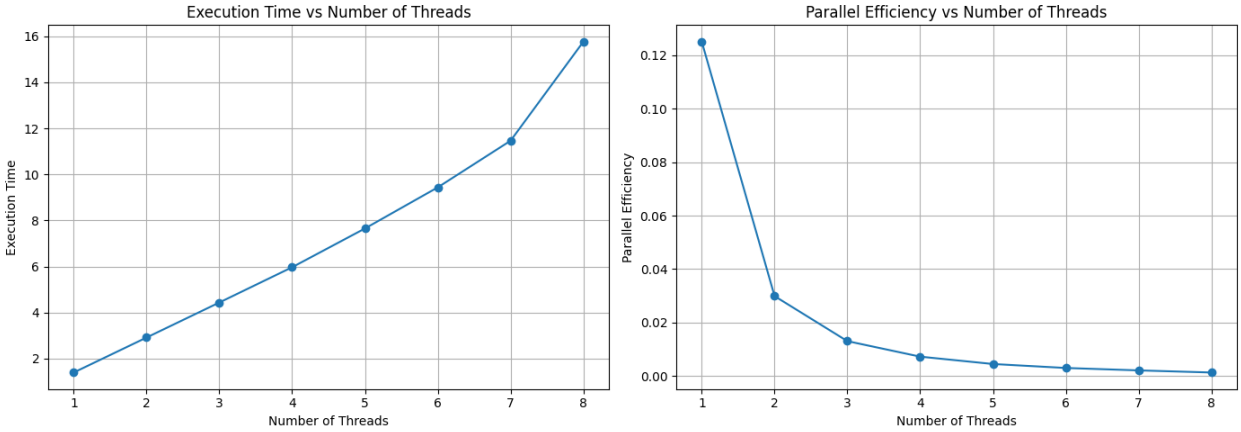


Figure 4: Weak Scaling execution and parallel efficiency