Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing **2022**

Student: Fabio Marchesi Discussed with: Raffarle Morganti, Gioele De Pianto, Nicholas Carlotti

---

Solution for Project 4

Due date: 23.11.2022, 23:59

---

1. Ring maximum using MPI [10 Points]

Given a ring topology of N nodes, where each node is a process, we want to allow processes to exchange information with their neighbors in order to find the maximum of a specific function. Each node should compute the maximum of the function: $f = 3 \times process\_rank_i$ MOD $2N$.
Each node starts computing the value of f using its rank, and then it passes it to the right neighbor. Each node will keep the maximum between its own value and the one received from the left neighbor, and will pass it again to the neighbor on the right. This process is repeated $N - 1$ times and in the end each node will have the maximum value of f.
So, for each process we compute its neighbors and the value of f using its rank.

```
right = (my_rank + 1) % size;
left = (my_rank - 1 + size) % size;
max = (3 * my_rank) % (2 * size);
```

Then, for $N - 1$ times, each processor:

- Sends the maximum it found to the right neighbor (without waiting for the notification that the message has been received).
- Prepares itself to receive for the left neighbor.
- Wait for the message of the left neighbor.

The code looks like this:

```
for (int i = 0, snd_buf = max; i < size; i++, snd_buf = max) {
        MPI_Issend(&snd_buf, 1, MPI_INT, right, 1, MPI_COMM_WORLD, &request);
        MPI_Recv(&rcv_buf, 1, MPI_INT, left, 1, MPI_COMM_WORLD, &status);
        MPI_Wait(&request, &status);
        if (rcv_buf > max) {
                max = rcv_buf;
        }
}
```

Each process uses sen_buf to store the value to send and rcv_buf to store the value that will receive. In the end, max will contain the maximum value for each process. In order to avoid deadlock, we use MPI_Issend which is the synchronous non-blocking send operation, this mean that each process doesn't wait for the recipient acknowledgment and can receive a message itself. Doing $N - 1$ iterations we are sure that each function value has the possibility of reaching every other node, this means that in the end all the process will contain the maximum value.

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

In this exercise, we have 16 processes arranged in a 4×4 grid. Each process is responsible for a 8×8 matrix, where every value is equal to the rank of the process that is storing the matrix. The starting matrix for each process looks like this.

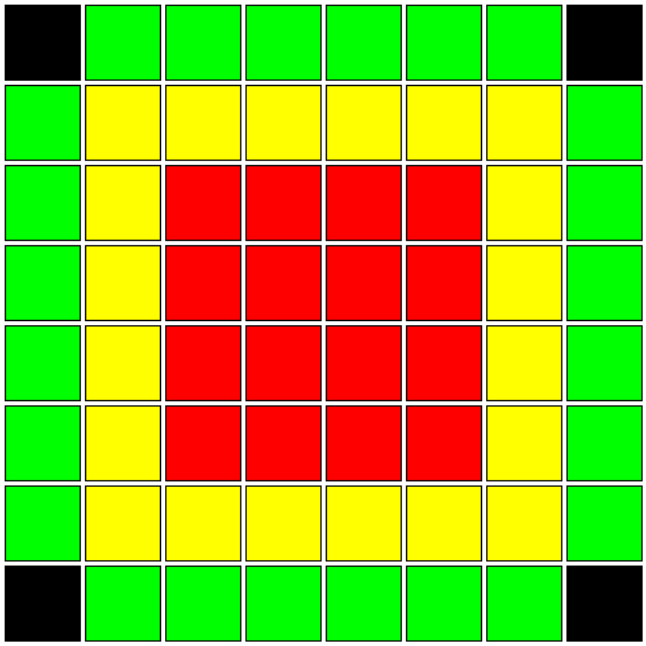


Figure 1: Matrix for each process

We have 16 central cells that won't be used or changed. The yellow cells can be thought as two rows and two columns that will be sent to the 4 neighbors. And the green cell represents the rows and columns that we will receive from the neighbors. So each process will send/receive a row or column to/from its neighbors. In order to put the processes into a grid we use a Cartesian 2-dimensional communicator with periodic boundaries. Firstly, we need to find the neighbors of each cell.

```
int dims[2] = {4, 4};
int periods[2] = {1, 1};
// Create grid
int ierr = MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm_cart);

// Finding top/bottom/left/right neighbors
MPI_Cart_shift(comm_cart, 0, 1, &rank_top, &rank_bottom);
MPI_Cart_shift(comm_cart, 1, 1, &rank_left, &rank_right);
```

And we also need to define two data-type for sending rows and columns with openMPI. **DOMAINSIZE** represents the number of cells for each row/column and **SUBDOMAIN** represents the number of cells to send/receive, so six in this case.

```
MPI_Type_vector(SUBDOMAIN, 1, 1, MPI_DOUBLE, &data_ghost_row);
MPI_Type_vector(SUBDOMAIN, 1, DOMAINSIZE, MPI_DOUBLE, &data_ghost_column);
MPI_Type_commit(&data_ghost_row);
MPI_Type_commit(&data_ghost_column);
```

The two types are defines as follows. Each process store the martix in the form of an array, so the first 8 cells will represent the first row, from 8 to 15 the second row and so on. When we define a vector type we need to specify the number of values that it will contain (six in this case), the number of elements for each position (int this case just one), the stride and the type. The stride needs to be equal to one when we want to send/receive a row because all the elements will be in continuous cells on the grid. Instead, when we want to send a column we need to skip from a row to the next one, so the stride will be equal to **DOMAINSIZE**. We can use the new defined types to send these cells.

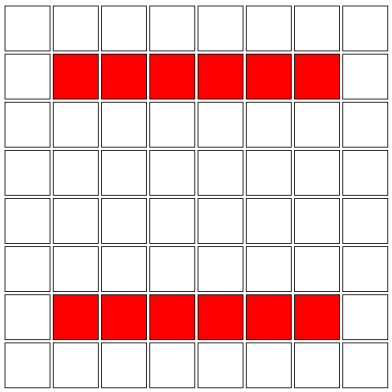


Figure 2: Rows data-type

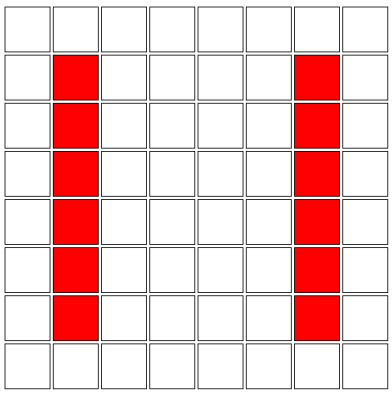


Figure 3: Columns data-type

After that, we just need to prepare the buffers that will store the messages and set that we are ready to receive (**MPI_Irecv** isn't blocking).

```
double rcv_up[SUBDOMAIN];
double rcv_left[SUBDOMAIN];
double rcv_bottom[SUBDOMAIN];
double rcv_right[SUBDOMAIN];
MPI_Irecv(&rcv_up, SUBDOMAIN, MPI_DOUBLE, rank_top, 0, comm_cart, &r1);
MPI_Irecv(&rcv_left, SUBDOMAIN, MPI_DOUBLE, rank_left, 1, comm_cart, &r2);
MPI_Irecv(&rcv_bottom, SUBDOMAIN, MPI_DOUBLE, rank_bottom, 2, comm_cart, &r3);
MPI_Irecv(&rcv_right, SUBDOMAIN, MPI_DOUBLE, rank_right, 3, comm_cart, &r4);
```

Send our data to the neighbors. (See how we use **data_ghost_row** to send the rows to the processes above and under the current one, and we use **data_ghost_column** to send the columns to the processes to the left and the right of the current process).

```
MPI_Send(&data[DOMAINSIZE * SUBDOMAIN + 1], 1, data_ghost_row, rank_bottom, 0, comm_cart);
MPI_Send(&data[DOMAINSIZE + SUBDOMAIN], 1, data_ghost_column, rank_right, 1, comm_cart);
MPI_Send(&data[DOMAINSIZE + 1], 1, data_ghost_row, rank_top, 2, comm_cart);
MPI_Send(&data[DOMAINSIZE + 1], 1, data_ghost_column, rank_left, 3, comm_cart);
```

Now we just need to wait for the messages that out neighbors sent us and then update the matrix like explained before.

```
MPI_Wait(&r1, MPI_STATUS_IGNORE);
MPI_Wait(&r2, MPI_STATUS_IGNORE);
MPI_Wait(&r3, MPI_STATUS_IGNORE);
MPI_Wait(&r4, MPI_STATUS_IGNORE);

for (int i = 0; i < SUBDOMAIN; i++) {
        data[i + 1] = rcv_up[i];
        data[(1 +  i) * DOMAINSIZE] = rcv_left[i];
        data[(DOMAINSIZE - 1) * DOMAINSIZE + i + 1] = rcv_bottom[i];
        data[(1 + i) * DOMAINSIZE + SUBDOMAIN + 1] = rcv_right[i];
}
```

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

In order to parallelize the Mandelbrot set, we need to divide the whole grid in small partitions and each partition will be computed by one process. The first thing that we need to do is creating the grid and assigning to each process the correct coordinates. When creating the grid, we let MPI choose in which way the matrix will be split.

```
Partition createPartition(int mpi_rank, int mpi_size) {
    Partition p;
    int dims[2] = {0, 0};   // No limitations on the grid shape
    MPI_Dims_create(mpi_size, 2, dims);
    p.nx = dims[1];
    p.ny = dims[0];
    MPI_Comm comm_cart;
    int periods[2] = {0, 0};
    int ierr = MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm_cart);
    p.comm = comm_cart;
    int my_coords[2];
    // Finding the coordinates for this process
    int err = MPI_Cart_coords(p.comm, mpi_rank, 2, my_coords);
    p.x = my_coords[1];
    p.y = my_coords[0];
    return p;
}
```

Once we have determined the coordinates for each process, we need to compute which pixel it needs to compute. In order to do that, we can use its coordinates and the width and height of the whole image.

```
Domain createDomain(Partition p) {
    Domain d;
```

```
    d.nx = IMAGE_WIDTH / p.nx;
    d.ny = IMAGE_HEIGHT / p.ny;
    // Compute index of the first pixel in the local domain
    d.startx = IMAGE_WIDTH / p.nx * p.x;
    d.starty = IMAGE_HEIGHT / p.ny * p.y;
    // Compute index of the last pixel in the local domain
    d.endx = d.startx + d.nx;
    d.endy = d.starty + d.ny;
    return d;
}
```

Now each process has the information about the part of the image that it needs to process, so it can just compute the color of each pixel. So, after the computation of the current process (excluding the master) is finished, it needs to send the values computed to the master. After this operation, each process doesn't have to do anything else, so it just uses a synchronous send.

```
    MPI_Send(c, d.nx * d.ny, MPI_INT, 0, mpi_rank, p.comm);
```

Instead the master process needs to copy the part that it computed on the image and start waiting for the other processes. An easy way to do it consists of waiting for the result of the process 1, then 2, $\dots N-1$. It's true that some process i could finish before process $j < i$, so the master could start to copy the result of i before j but the process of copying is fast compared to the computation of each partition of the grid, so anyway the total time required is similar to the time required for the partition that requires the most time. So process 0 could do something like this:

```
if (mpi_rank == 0){
    Write own data
    // Receive and write the data from other processes
    for (int proc = 1; proc < mpi_size; proc++) {
            Partition p1 = updatePartition(p, proc);
            Domain d1 = createDomain(p1);
            MPI_Recv(c, d1.nx * d1.ny, MPI_INT, proc, proc, p1.comm, MPI_STATUS_IGNORE);
            // write the partition of the process proc
    }
    Write image
}
```

After that, we compared the total time required to plot the Mandelbrot set of size 4096x4096 with $p = 1, 2, 4, 8, 16$ processes. And this is what I got at first.

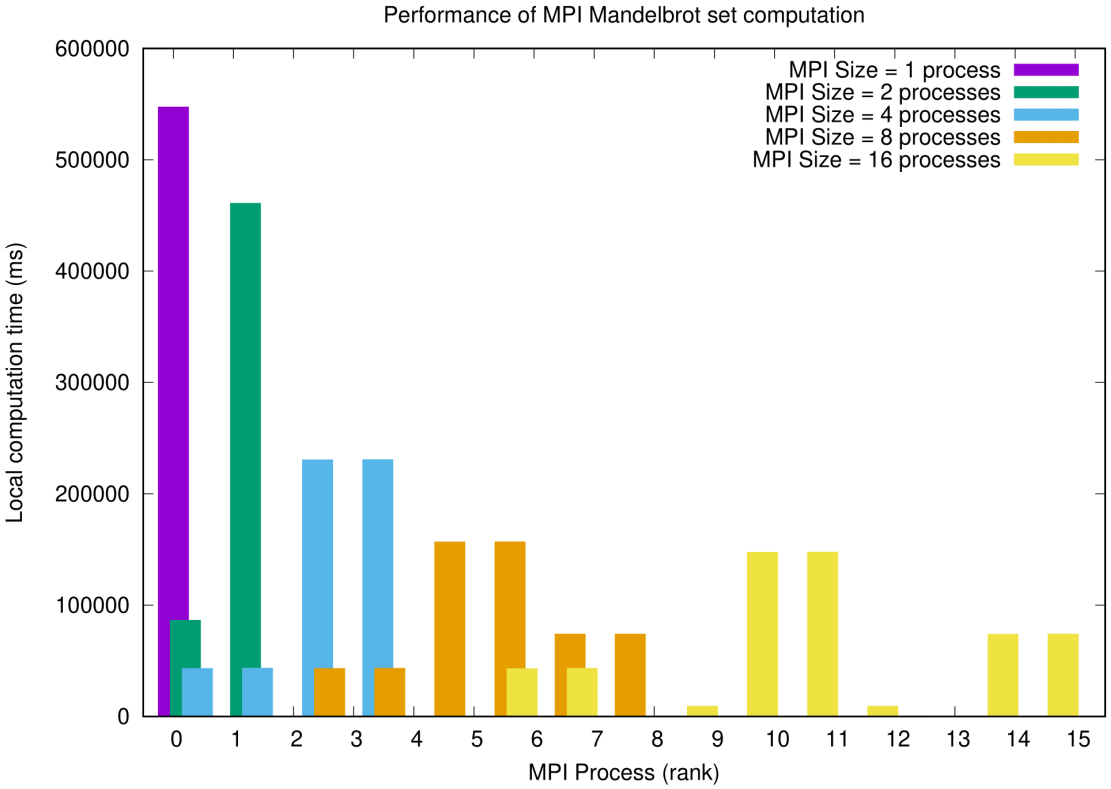
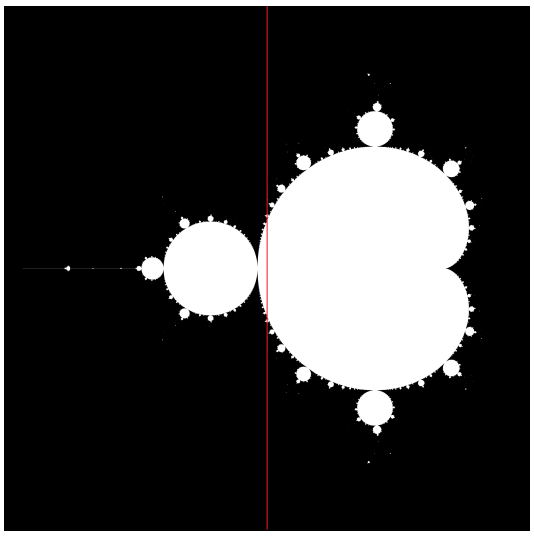
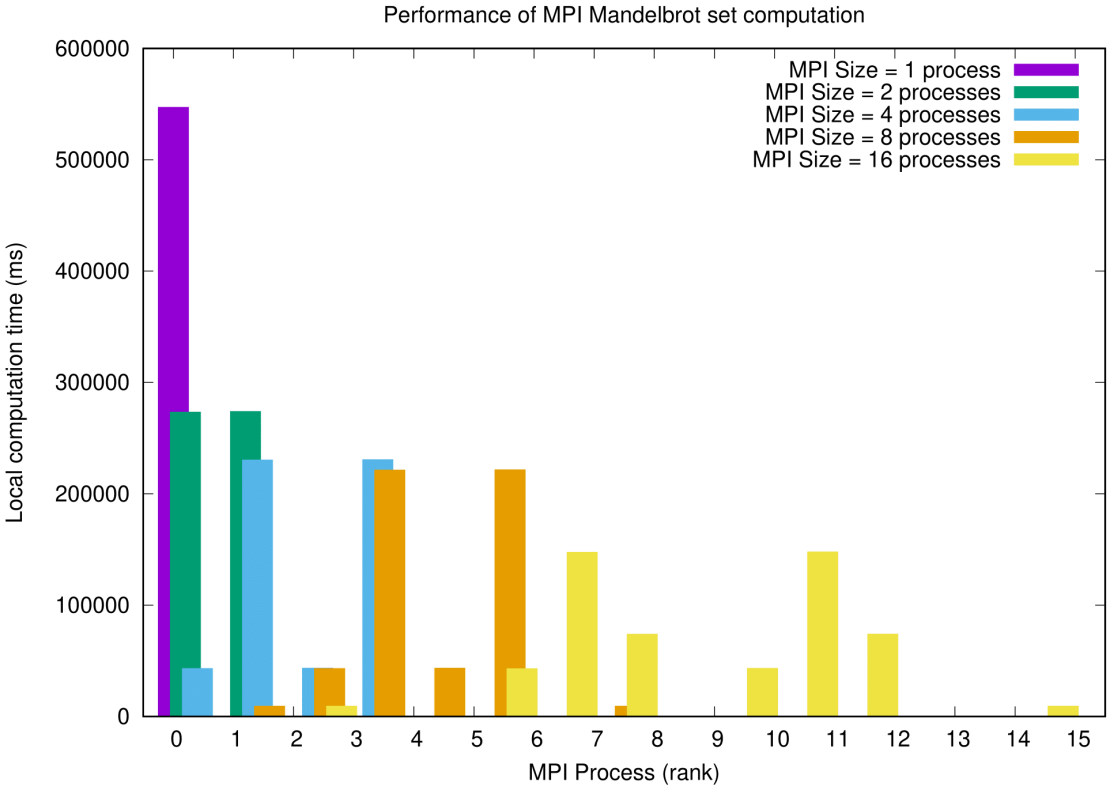


Figure 4: First results obtained

What we really look for in each test it the highest column, because that column would be the last one to finish. And at first this graph look good because if we look at the highest column for each test we notice that it becomes shorter increasing the number of processes. But looking at the chart with $p = 2$ I noticed how the two columns look very different from each other. That's because the brighter each pixel is and the more time it requires to be computed, and now the image is split vertically. And the right part contains many more white pixels than the left part.



Instead, if I split the image horizontally, the performance chart with two process results more balanced.



Now we see that, when we have 2 processes, the time required for each of them is the same and that's because the image has a horizontal axis of symmetry. But the problem persists with 8 and 16 processes because there will be some partitions that are almost completely black and other that are mostly white. If we split the grid like we are doing now, so where each process computes a contiguous region of the image there is no way of having all the columns of the same height (this would be the perfect case). If we wanted to do that, we could exploit the fact that each white pixel has a higher probability of being surrounded by white pixels than a black pixel, so, we could split the whole image in sub-matrices of size p and let each process do a pixel of each one of those sub-matrices. But also with the current way of splitting the work, we can see that having more processes is beneficial in terms of total time required to compute the matrix. But the time improvement isn't proportional to the number of added processors (remember that what really matters is the height of the highest column in the chart), indeed if we look at the case with 16 processors the slowest process takes around 140 seconds while with only one process the whole computation takes around 540 seconds. So the best thing to do here would be, like suggested before, to change the way pixels are split between processes in a way that each process has roughly the same number of operations to do.

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

We want to compute the power method with the data (matrix A) being split between different processes efficiently. Let's suppose that A is a matrix of size NxN and that we have p processes working on it, where N is a multiple of p. This means that each process will store just $\frac{N}{p}$ rows of A. For this reason, we let the processes generate their own rows. The indices of the rows of each process are established using the rank of the process itself, so the first process (the master) will

store rows $0, \ldots, \frac{N}{p} - 1$ the second process will store rows $\frac{N}{p} \ldots \frac{2N}{p} - 1$ and so on. We also need a vector $\vec{x}$ of size N this vector is managed by the master, indeed it's stored in process 0. The power method repeates the following operations multiple times:

- Normalize $\vec{x}$.
- $\vec{x} = A \cdot \vec{x}$

And in the end, the norm of $\vec{x}$ will correspond to the greatest eigenvalue in absolute value. The first step is done by the master process that stores the current version of $\vec{x}$, after that the master will send the vector $\vec{x}$ to the other processes, so they can compute their part of the product. Each process has now $\vec{x}$ and its set of rows, let's say that the current process has the sub-matrix $\bar{A}$ that contains the rows from i to j of A. The process will compute, $\vec{\bar{x}} = \bar{A} \cdot \vec{x}$ obtaining a vector with $j - i + 1$ elements. Now each process except the master needs to send its $\vec{\bar{x}}$ to the master so it can merge them all together and go to the next iteration. Looking at the code we start by generating A and $\vec{x}$:

```
int startrow = N / size * my_rank;
int endrow = N / size * (my_rank + 1) - 1;
int sz = endrow - startrow + 1;
double* A = hpc_generateMatrix(N, sz);
double* x = generateMatrix(N, 1);
```

And then for each iteration we let the master process compute the norm and normalize $\vec{x}$

```
if (my_rank == 0) {
        double current_norm = norm(n, x);
        for (int j = 0; j < n; j++) {
                x[j] /= current_norm;
        }
}
```

The master sends $\vec{x}$ to all the other processes using a broadcast message (**MPI_Bcast**)

```
// Sharing an array of N doubles
MPI_Bcast(x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Each process (even the master) computes its own part of the matrix-vector product.

```
double* prod = matVec(A, x, N, sz);
```

Here each process has prod which is a vector of size $\frac{N}{p}$, so it sends it to the master, so it can merge them. In order to do that, we use **MPI_Gather**.

```
MPI_Gather(prod, sz, MPI_DOUBLE, x, sz, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

So now the master has the new complete vector $\vec{x}$ and can use it in the next iteration. After k iterations, we have that $|\vec{x}|$ represents the greatest (in absolute value) eigenvalue of A. And if A has an eigenvalue that is strictly greater in magnitude than its other eigenvalues and the starting vector $\vec{x}$ has a nonzero component in the direction of an eigenvector associated with the dominant eigenvalue, $\vec{x}$ then converges to an eigenvector associated with the dominant eigenvalue.

4.1. Strong Scale Analysis

In order to understand how well our code works with an increasing number of precesses, we preform a strong scale analysis. This mean that we keep the same input, in this case we work with a matrix of side $2^7 = 16384$ and perform 1000 iterations. but we increase the number of processes. The goal of this analysis is to check whether the improvements in performance are proportional to the increase of resources. We performed the test using $p = 1, 4, 8, 12, 16, 32, 64$.

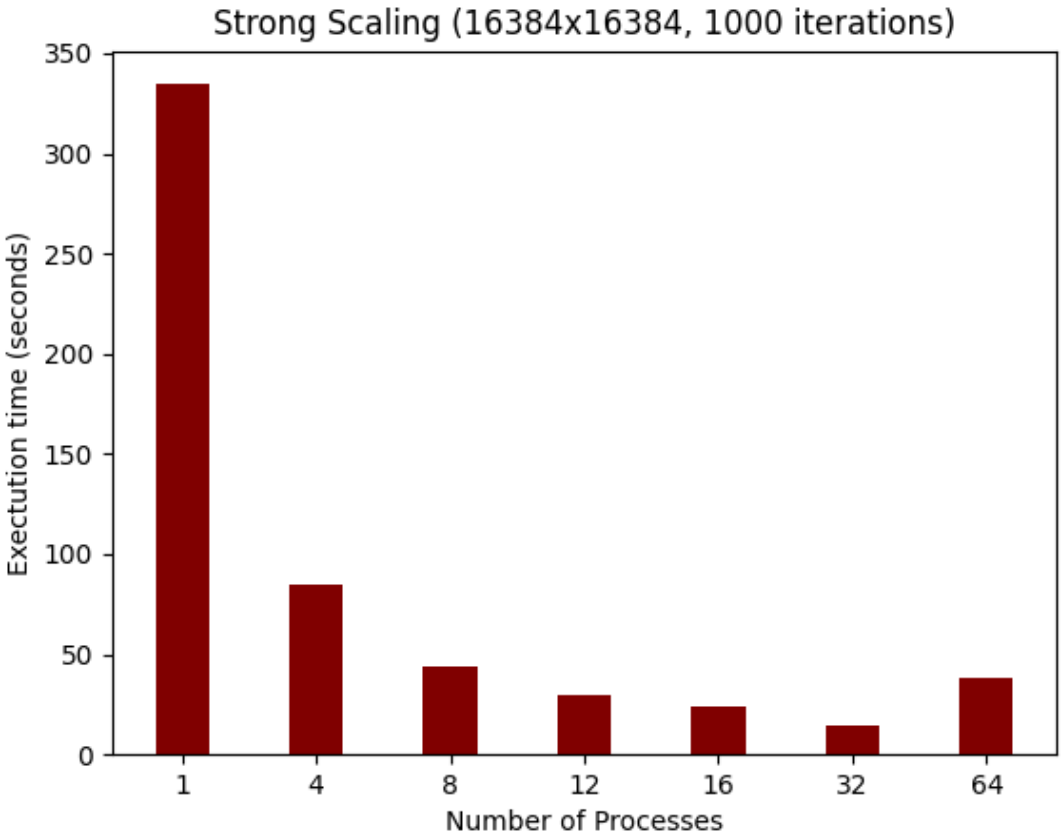


Figure 5: Strong Scale Analysis

The result are pretty good, in most cases increasing the number of processes will give a big performance improvement. I say in most cases because we can see that when $p = 64$ we have a slower execution than with $p = 12, 16, 32$. The reason behind that is that when we work with **Open MPI** we have that the processes that we have spawned are memory independent and, in order to share information, they need to communicate (like when we used **MPI_Bcast** and **MPI_Gather**), All the processors on the cluster have 20 cores this means that if we have 64 processes they are at least on 4 different processors, and this increase the communication time. Another way to check whether our application has a good strong scaling is to check parallel efficiency. We compute the parallel efficiency as $\frac{speed\_up}{used\_cores} \times 100$ where $speed\_up = \frac{sequential\_time}{parallel\_time}$. This metric is particularly useful to understand which percentage of the parallel computational power are we actually using.

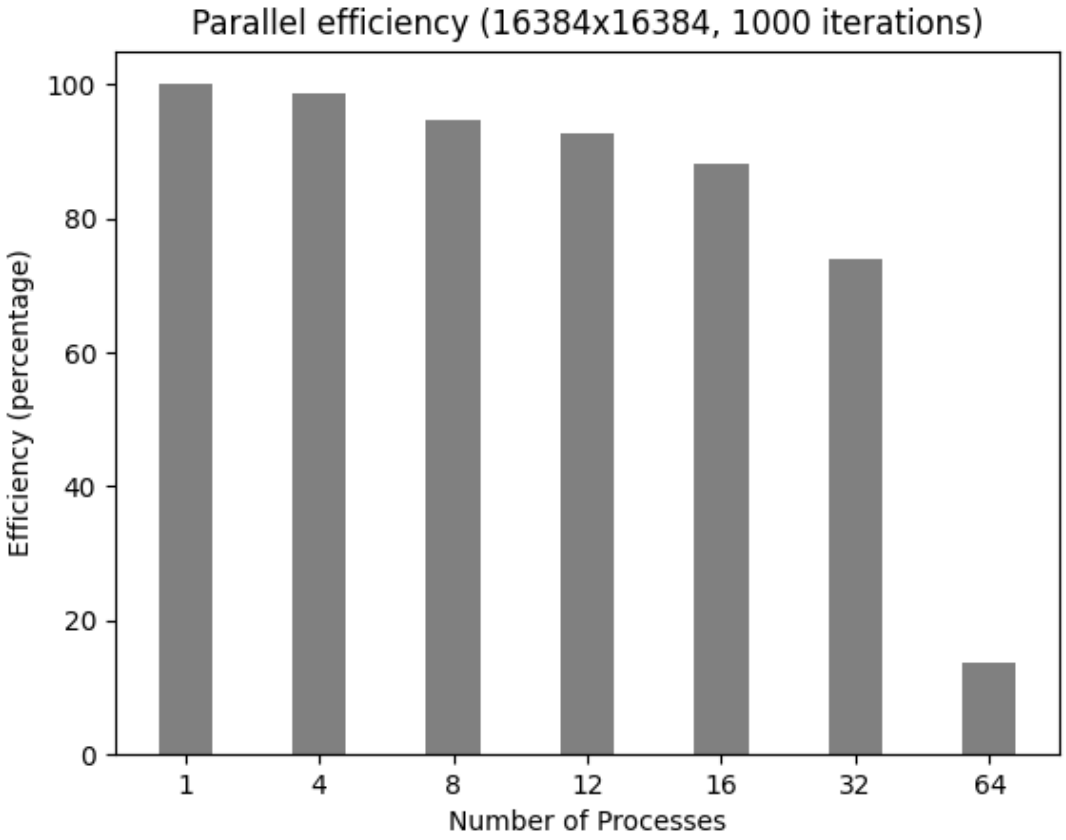


Figure 6: Strong Scale Efficiency

We can see that the parallel efficiency for $p = 4, 8, 12, 16$, which are the cases where all the processes fit in one processor, is pretty high (above 88%). With $p = 32$ the efficiency decreased a bit, reaching 73%, with $p = 64$ we have an awful efficiency (13%).

4.2. Weak Scaling

In order to measure how our code deals with weak scaling, we need to run simulations with different number of processes and different matrix sizes in a way that the workload for each core remains the same. For this test we decreased the number of iteration from 1000 to 100. In order to do that, we used the following values (the number of cells for each process isn't the exact same for each case, but is really similar).

Number of Processes	Side of the matrix	Cells for each process
1	16.384	268.435.456
4	32.768	268.435.456
8	46.340	268,424,450
12	56.756	268.436.961
16	63.536	268.435.456
32	92.691	268.488.171
64	131.072	268.435.456

And these are the result that we have got.

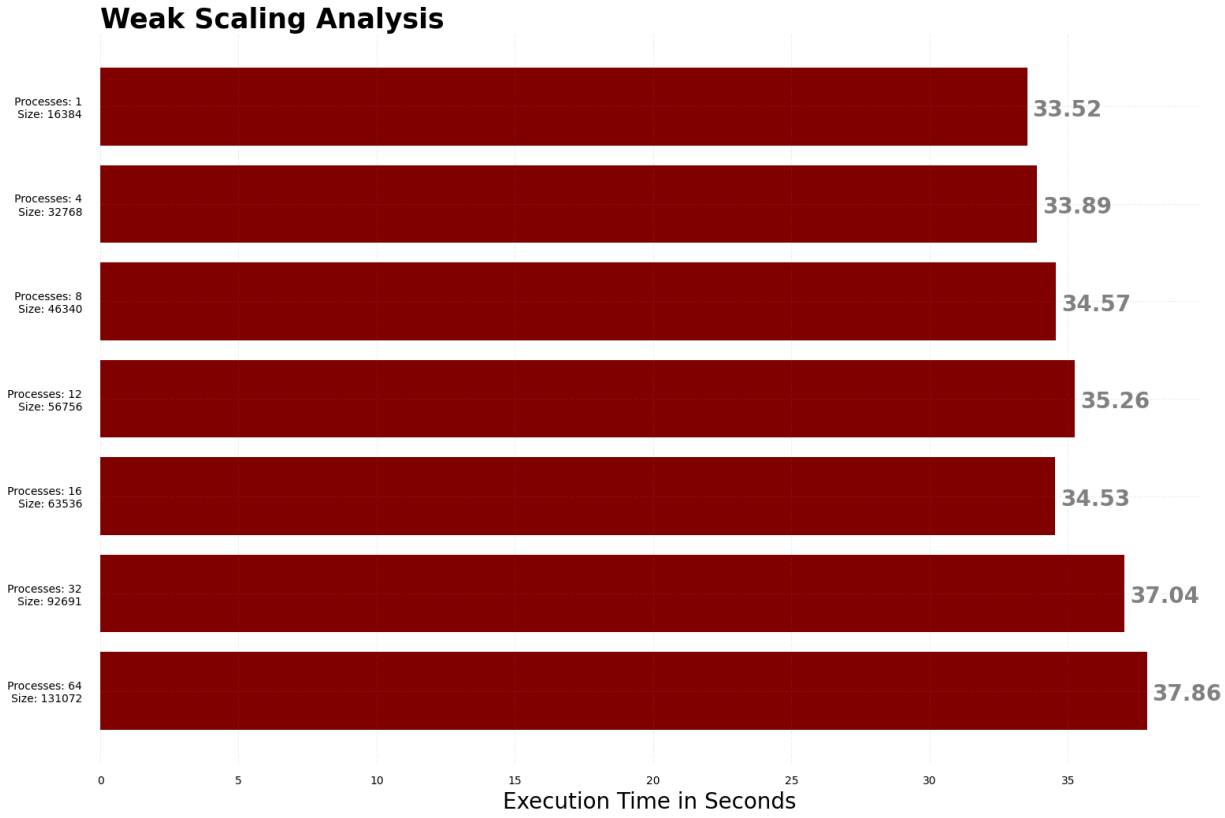


Figure 7: Weak Scaling Analysis

We can see how to program weak scales almost perfectly. Indeed, the communication overhead doesn't seem to play a crucial role because the executions time remain really similar also with many processes. This means that if we want to compute the power method on a massive matrix, we can divide the computation between many processes knowing that there won't be a big overhead on doing that. To prove how well it weak scales, we also computed the efficiency as $\frac{T_1}{T_p}\%$ where T_1 is the execution time with one processor while T_p is the execution time with p processors.

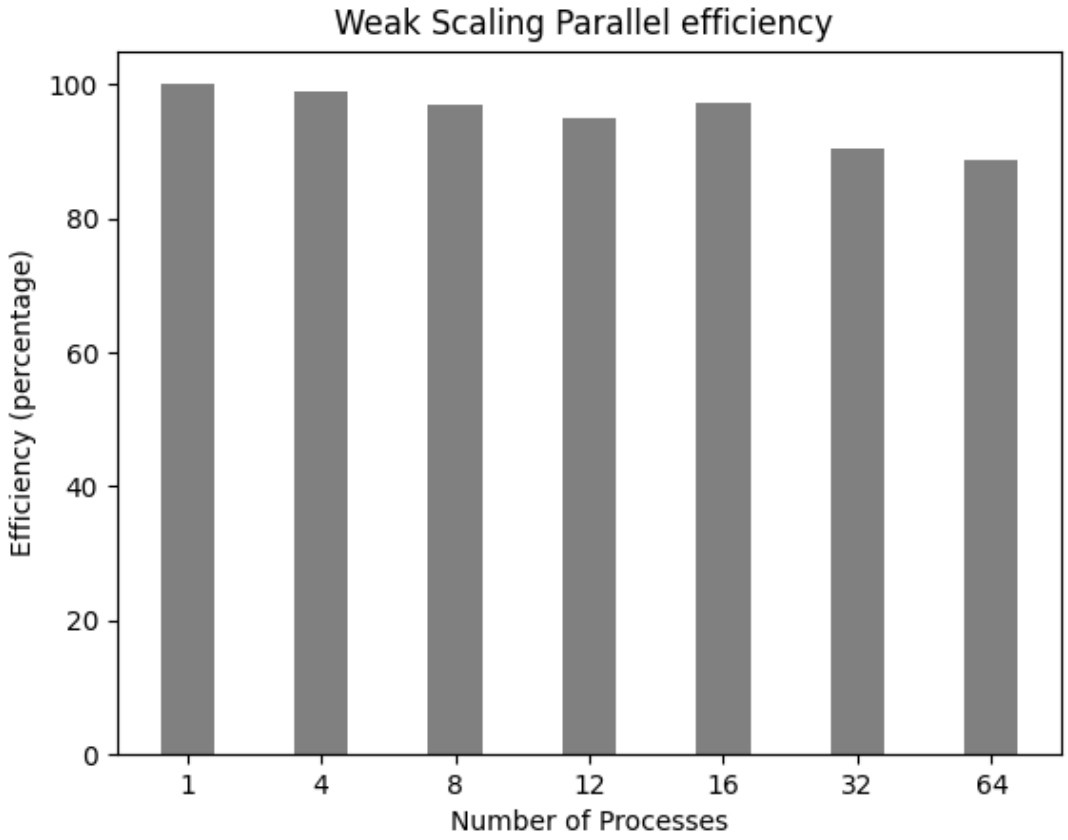


Figure 8: Weak Scale Efficiency

The percentages are really high, this means that also if we increase the matrix size and the number of processors we don't waste too much computational power. If the strong scale analysis helps to understand how well we can split long executions between many processes, the weak scale analysis helps to understand how well we can divide the memory usage between different processes. That is, if we wanted to apply the power method on a huge matrix (like we did in the test with $p = 64$) we can do it with a small overhead.

5. 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 MPI solutions;
  - your write-up with your name project_number_lastname_firstname.pdf,

- Submit your .tgz through Icorsi.