Università della Svizzera italiana | Institute of Computational Science ICS

High-Performance Computing Lab **Fall 2020**

Student: Gabriele Berra Discussed with: -

Solution for Project 6

Due date: 27.11.2020 23:59 (midnight)

1. Ring maximum using MPI [10 Points]

In this first exercise I implemented the function *max_ring.c* parallelized with MPI. Similarly to what we did in class I firstly define the left and right neighbour and the value of the function

$$\text{max} = 3 * \text{process\_rank}_i \mod (2 * \text{communicator\_size}) \tag{1}$$

for each processor. Since we have to run the function with 4 processors we know that the global maximum of Eq. 1 is reached in the processor 2 with max = 6. In the main body of the function, on one hand, each processor, with the MPI routine MPI_Send, exchanges its own maximum with the next processor. On the other hand, with the MPI routine MPI_Recv, the processor that receive the data check if its own maximum is bigger or lower than the maximum that it has received and keeps the bigger one. Repeating this routine in a circular manner 4 times gives the following results:

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

Of course, every time we run the script the order of the processes will change. In what follows I reported the main body of the function *max_ring.c*.

```
for (int i = 0, snd_buf = max, rcv_buf; i < size; i++, snd_buf =
    max)
{
    MPI_Request request;
    ierr = MPI_Issend(&snd_buf, 1, MPI_INT, right, 0,
        MPI_COMM_WORLD, &request);

    MPI_Status status;
    ierr = MPI_Recv(&rcv_buf, 1, MPI_INT, left, 0, MPI_COMM_WORLD,
        &status);

    if (ierr != MPI_SUCCESS){
        printf("ProcID %i did not successfully receive a value! \n
            ", my_rank);
    }

    if(max < rcv_buf){
        max = rcv_buf;
    }

}
```

2. Ghost cells exchange between neighboring processes [20 Points]

For this second exercise we have to parallelize wit MPI the exchange of ”ghost cells” between adjacent processors in a defined grid. The input data on which we have to work is a $(6+2)\times(6+2)$ matrix associated with each processors in which each entries has value equal to the rank of the processor. The first step is to create a Cartesian communicator of dimension 4×4 (which is a sort of matrix for the processors) with periodic boundaries, which means that the processor 0 - which is in position $(0, 0)$ - can communicate with processor 3 and processor 12 on the left and top respectively. In order to find, for each processors, the top, bottom, left and right neighbour we have to use the MPI routine MPI_Cart_shift which, given a shift direction and the amount of ”cells” to be shifted, returns the source and destination rank. In our case, to find the left and right neighbour we use:

```
MPI_Cart_shift(comm_cart,1,1,&rank_left,&rank_right);
```

which shifts on the x axis (second input) of 1 cell (third input). Similarly, to find the top and bottom neighbour we use:

```
MPI_Cart_shift(comm_cart,0,1,&rank_top,&rank_bottom);
```

which, in this case, shifts on the y axis - vertically - of 1 cell.
Now that we know the neighbourhood of each processor we have to exchange the correct data between them. We know that, since C stores data in row-major order, we have no problem in exchanging rows of a matrix because the data are contiguous in memory. The problem occurs when we want to exchange data that are not contiguous in memory. Fortunately MPI gives us the routine MPI_Type_vector which can create a vector-type object. In order to use it in the correct manner we first have to understand how it works. As input it accepts:

- *count*: number of blocks that we want to send;
- *blocksize*: dimension of the block (in our case, since we want to sent only double, is equal to 1);

- *stride*: distance in memory between elements that we want to send;
- *type*: type of the item that we want to send.

The last element in MPI_Type_vector is the output. Now we have created only a structure without any data inside it. When we use it in MPI_Send, what it does is to take *count* elements with distance from each other equal to *stride*.
The last step is the exchange of the correct data between neighbours. To do so I used the MPI routine MPI_Send and MPI_Recv in which, for each processor, I specify the correct position of the data that has to be sent and the position in which the receiver has to put the data received. As explained before, I used the vector-type object only to exchange data between left and right neighbours. The code I implemented is the following:

```
// to the top
MPI_Send(&data[DOMAINSIZE+1],SUBDOMAIN, MPI_DOUBLE, rank_top, 0,
   MPI_COMM_WORLD);
MPI_Irecv(&data[DOMAINSIZE*(DOMAINSIZE-1)+1],SUBDOMAIN, MPI_DOUBLE,
     rank_bottom, 0, MPI_COMM_WORLD, &request);
MPI_Wait(&request, &status);

// to the bottom
MPI_Send(&data[DOMAINSIZE*(DOMAINSIZE-2)+1], SUBDOMAIN, MPI_DOUBLE,
     rank_bottom,0,MPI_COMM_WORLD);
MPI_Irecv(&data[1], SUBDOMAIN, MPI_DOUBLE, rank_top, 0,
   MPI_COMM_WORLD, &request);
MPI_Wait(&request, &status);

// to the left
MPI_Send(&data[DOMAINSIZE+1], 1, data_ghost, rank_left, 0,
   MPI_COMM_WORLD);
MPI_Irecv(&data[DOMAINSIZE*2-1], 1, data_ghost, rank_right, 0,
   MPI_COMM_WORLD, &request);
MPI_Wait(&request, &status);

// to the right
MPI_Send(&data[DOMAINSIZE*2-2], 1, data_ghost, rank_right, 0,
   MPI_COMM_WORLD);
MPI_Irecv(&data[DOMAINSIZE], 1, data_ghost, rank_left, 0,
   MPI_COMM_WORLD, &request);
MPI_Wait(&request, &status);
```

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

With the aim of creating the mandelbrot set using MPI I firstly implemented the functions *createPartition()*, *updatePartition()* and *createDomain()* in the file *consts.h* as follows:

```
Partition createPartition(int mpi_rank, int mpi_size)
{
    Partition p;

    // TODO: determine size of the grid of MPI processes (p.nx, p.ny),
       see MPI_Dims_create()
    int gridSize[2] = {0,0}; // automatically select how many process
        in x and y
    MPI_Dims_create(mpi_size, 2, gridSize); // distribution of process
        in x and y
    p.nx = gridSize[1];
    p.ny = gridSize[0];
    // p.ny = 1;
    // p.nx = 1;


    // TODO: Create cartesian communicator (p.comm), we do not allow
        the reordering of ranks here, see MPI_Cart_create()
    int periods[2] = {1,1};
    MPI_Cart_create(MPI_COMM_WORLD, 2, gridSize, periods, 0, &p.comm);
    // MPI_Comm comm_cart = MPI_COMM_WORLD;
    // p.comm = comm_cart;


    // TODO: Determine the coordinates in the Cartesian grid (p.x, p.y)
        , see MPI_Cart_coords()
    int coords[2];
    MPI_Cart_coords(p.comm, mpi_rank, 2, coords);
    p.y = coords[0];
    p.x = coords[1];

    //p.y = 0;
    //p.x = 0;

    return p;
}

/**
Updates Partition structure to represent the process mpi_rank.
Copy the grid information (p.nx, p.ny and p.comm) and update
the coordinates to represent position in the grid of the given
process (mpi_rank)
*/
Partition updatePartition(Partition p_old, int mpi_rank)
{
    Partition p;

    // copy grid dimension and the communicator
    p.ny = p_old.ny;
    p.nx = p_old.nx;
    p.comm = p_old.comm;

    // TODO: update the coordinates in the cartesian grid (p.x, p.y)
        for given mpi_rank, see MPI_Cart_coords()
    int coords[2];
    MPI_Cart_coords(p.comm, mpi_rank, 2, coords);
    p.y = coords[0];
    p.x = coords[1];
    //p.y = 0;
    //p.x = 0;

    return p;
}

/**
Structure Domain represents the information about the local domain of
    the current MPI process.
It holds information such as the size of the local domain (number of
    pixels in each dimension - d.nx, d.ny)
and its global indices (index of the first and the last pixel in the
    full image of the Mandelbrot set
that will be computed by the current process d.startx, d.endx and d.
    starty, d.endy).
*/
Domain createDomain(Partition p)
{
    Domain d;

    // TODO: compute size of the local domain
    d.nx = IMAGE_WIDTH/(p.nx);
    d.ny = IMAGE_HEIGHT/(p.ny);

    //d.nx = IMAGE_WIDTH;
    //d.ny = IMAGE_HEIGHT;

    // TODO: compute index of the first pixel in the local domain
    d.startx = (p.x * (IMAGE_WIDTH)/p.nx);
    d.starty = (p.y * (IMAGE_HEIGHT)/p.ny);

    //d.startx = 0;
    //d.starty = 0;

    // TODO: compute index of the last pixel in the local domain
    d.endx = (p.x + 1)*(IMAGE_WIDTH/p.nx)-1;
    d.endy = (p.y + 1)*(IMAGE_HEIGHT/p.ny)-1;

    //d.endx = IMAGE_WIDTH - 1;
    //d.endy = IMAGE_HEIGHT - 1;

    return d;
}
```

As we can see in figure 1 the time for the computation of the mandelbrot set using two processors is halved compared to the time using only one processors. In contrast, the differences between the time using 4 processors and the time using 8 processors is almost the same. We can notice that the computation time of, as example, the processor 5, in the run using 16 processes, is exactly equal to the computational time of processor 9. This comparison can be extended to all the processes. The reason for this behaviour lies in the symmetry, with respect to the x axis, of the mandlebrot set. Thus, in a 4×4 grid reported in the left side of Fig.2 the pairs of processors with the same computational time are: (0 - 12), (1 - 13), (2 - 14), (3 - 15), (4 - 8),(5 - 9),(6 - 10) and, (7 - 11). We know that different areas of the mandelbrot set require different amount of time to compute.

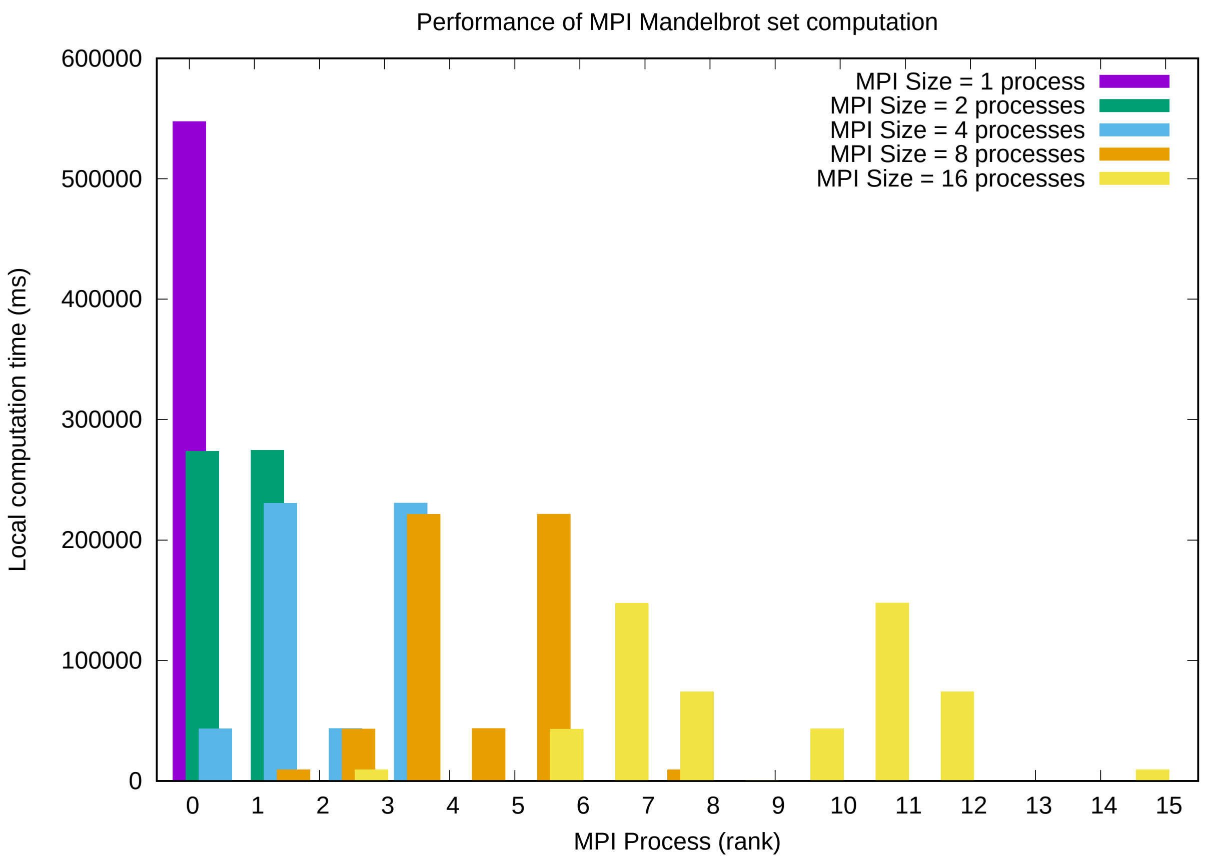
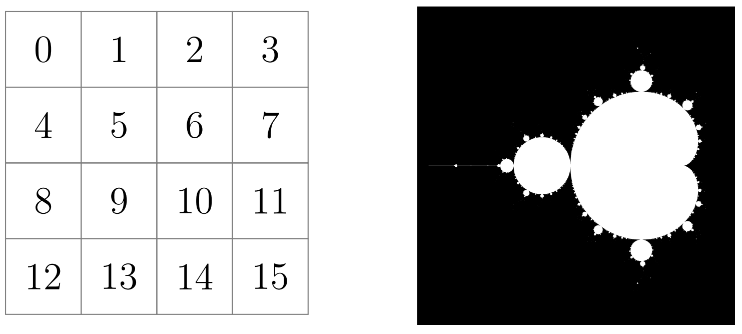


Figure 1: Performance with different amount of processors

In my opinion, the most interesting aspect - which is also the explanation of why this method of partitioning is not the most efficient - is the difference in terms of computational time between different processes. One of the most efficient way to parallelize the mandelbrot set is to distribute the work among the processes in an equal way in terms of computational time.



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 2: Mandelbrot set

4. Option A: Parallel matrix-vector multiplication and the power method [50 Points]