HIGH PARALLEL COMPUTING

ECOLE CENTRALE LILLE

MASTER DATA SCIENCE

Lab2: the Mandelbrot set

Authors:
Ayoub Youssoufi
Yassin El Hajj Chehade

Professor:
Pierre Fortin

September 18, 2023

1 Introduction

An image is a two-dimensional array. Each element of this array is named a pixel, short for picture element. Its value is, depending on the image type, a gray level value, a color or a radiance value. The image is organized in memory row by row: we have the first row, then the second, and so on. In particular, we are handling images encoded on one byte and the value for each pixel is between 0 to 255.

Figure 1: The image presentation with pixel

2 Work description

2.1 Simple implementation

We want to create a first parallel algorithm using MPI which distributes the pixels on processors. Each processor is going to process part of the image. The chosen number of processors in this case is dependant on the height h of the image. Therefore, the partition of the image to fraction $\frac{height}{NumberProcessor}$ has to be an integer. In this problem, we decided to choose 5 processors.

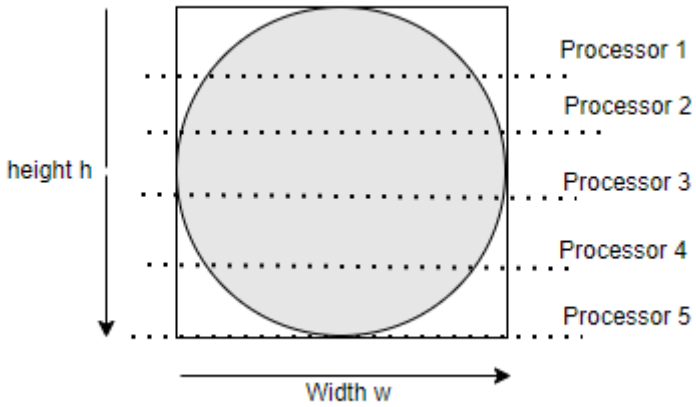

Figure 2: The image presentation with the processors

2.1.1 Detailed algorithm part 1

We will break down the code into 3 parts, the first is the declaration of the message variables where we will store the data. Then, the second part where we will parallelize the process of the algorithm on the image. Finally, we will finish with the part where we will gather the data to create the full image.

In the algorithm 1, we allocate memory for each processor and just one more array for the process 0 to gather all the values.

In algorithm 2 , the idea is to use the variable my_rank which indicates the rank of the process to start in the right place.

Now, in the algorithm 3, MPI_Probe is method used to check which process is sending the result of the computation of the image. If a call to MPI_Probe has been issued by a process, and a send that matches the probe

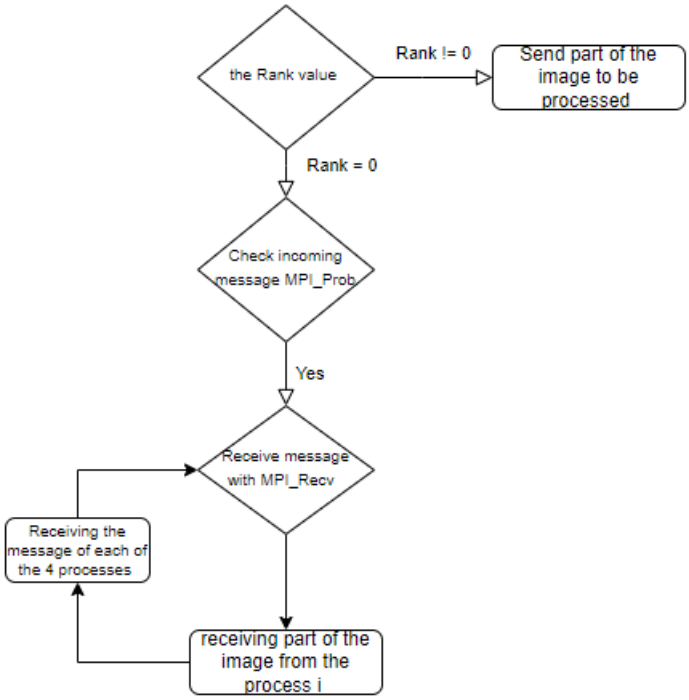

Figure 3: The process of the MPI_Send and MPI_Recv of the image computed

Algorithm 1 Allocate memory for each processor

```
if my_rank == 0 then
    imag ← (unsigned char *)malloc( w*h*sizeof(unsigned char));
end if
ima ← (unsigned char *)malloc( w*h*sizeof(unsigned char)/p );
```

has been initiated by some process, then the call to MPI_Probe will return and will initiate MPI_Recv to receive the message of the specific process. As the result, the processor with rank equal 0 will receive all the results of the other processes. This operation will be executed until the reception of all the computation of the different processors is terminated. You will find above the flowchart diagram of the algorithm implemented.(Figure3)

Another possible solution was to use MPI_Gather to gather the data processed by each process and send them to process 0, to create the image. Algorithm 4 shows this method.

2.2 Improvement using Master/worker

In this case, the partition of the loop has to be done as follow master-slave procedure:

- Divide the range of the array into h/nlines parts, approximately of the same size, where h is the height of the image and nlines is the number of lines to process for each task attached to a processor. 4 processors are chosen in this case

Algorithm 2 Image process

```
y ← ymin + my_rank*yinc*(h-1)/p;
for int i ← 0; i < h/p; i ← i + 1 do
    x ← xmin;
    for int j ← 0; j < w; j ← j + 1 do
        ima[j+i*w] ← xy2color(x,y,prof);
        x ← x + xinc;
    end for
    y ← y + yinc;
end for
```

Algorithm 3 Creating the image

```
if my_rank ! = 0 then
  MPI_Send(ima,w*h/p , MPI_CHAR, 0, tag, MPI_COMM_WORLD);
end if
if my_rank == 0 then
  imag ← ima;
  for int i ← 1; i < p; i ← i + 1 do
    MPI_Probe(MPI_ANY_SOURCE,tag,MPI_COMM_WORLD,&status);
    int rank ← status.MPI_SOURCE;
    MPI_Recv(imag+(w*h*rank/p),w*h/p,MPI_CHAR,rank,tag,MPI_COMM_WORLD,&status);
  end for
end if
```

Algorithm 4 Alternative to gather data

```
if my_rank ! = 0 then
  MPI_Gather(ima,w*h/p , MPI_CHAR, NULL, 100, MPI_CHAR,0, MPI_COMM_WORLD);
end if
if my_rank == 0 then
  MPI_Gather(ima,w*h/p, MPI_CHAR,imag,*w*h/p,MPI_CHAR,0,MPI_COMM_WORLD);
end if
```

- Compute the k*lines part by using processor i, and send the result to process master
- While the end of the image has not been reached yet , the master will keep sending new task to do for each process.
- At the end, the master process will create the final image.

The distribution of the tasks between master/slave processors is described in (Figure 4)

2.2.1 Detailed algorithm part 2

Now let us detail the algorithm used:

The code is divided into 3 parts. The first part is related to the declaration of the message variables where the data is stored. In the second part, we explain the idea behind the worker processor. And finally, we will explain how the master processor works.

Algorithm 5 Allocate memory for each processor and message

```
if my_rank == master_rank then
  imag ← (unsigned char *)malloc( w*h*sizeof(unsigned char));
end if
ima ← (unsigned char *)malloc( w*nlines*sizeof(unsigned char));
rec_send ← (double *)malloc(2*sizeof(double));
```

The algorithm 5 rec_send contains 2 elements. it has either 0 or 1. 0 for no longer data to process and 1 to indicate that a specific processor still have to do some computations. The second element in rec_send will be the starting part of the image that need to be processed. Notice that the floating point double is used since the variable y and yinc are of type double.

The algorithm 6 shows the worker procedure where we start by receiving the message from the master. If the first element is 0, the code stop running, otherwise, we process the part of the image that start in rec_send[1] and then we send to the master the processed array. This loop runs until receiving the specific message to stop.

In algorithm 7 the master start by sending the first job to each processor, and increasing each time the actual line to compute by nlines. This condition is added in order to avoid sending task to the master since the master only organise the whole process and do not contribute in the computation.

In algorithm 8 the master starts by receiving the completed tasks from each processor that finished its job. Depending on the current line to process if the line is arrived to the end (i.e. it is equal to the height of the image to be processed in that task), the master sends to this worker a message to indicate that its work is done. The master keep waiting on receiving the results from other workers until all of them have finished the task. In the other case, if the actual line to process is lower than the height, the master sends for this worker

Algorithm 6 Slave processors

```
int master_msg[2];
while 1 do
  MPI_Recv(rec_send, 2, MPI_DOUBLE, master_rank, tag, MPI_COMM_WORLD, &status);
  if rec_send[0] == 1 then
    y ← ymin + rec_send[1]*yinc*(h-1)/h;
    for int i ← 0; i < nlines; i ← i + 1 do
      x ← xmin;
      for int j ← 0; j < w; j ← j + 1 do
        ima[j+i*w] ← xy2color(x,y,prof);
        x ← x + xinc;
      end for
      y ← y + yinc;
    end for
    MPI_Send(ima, w*nlines, MPI_CHAR, master_rank, tag, MPI_COMM_WORLD);
  end if
  if rec_send[0] == 0 then
    break;
  end if
end while
```

Algorithm 7 Master processor 1

```
double actual_line ← 0;
double master_send[2];
master_send[0] ← 1;
int store_lines[p];
for int i ← 0; i < p; i ← i + 1 do
  master_send[1] ← actual_line;
  if i == master_rank then
    i ← i + 1
  end if
  MPI_Send(master_send, 2, MPI_DOUBLE, i, tag, MPI_COMM_WORLD);
  store_lines[i] ← actual line;
  actual_line ← actual_line + nlines;
end for
```

Algorithm 8 Master processor 2

```
int last_send ← 0;
while 1 do
  MPI_Probe(MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status);
  int rank = status.MPI_SOURCE;
  MPI_Recv(imag+store_lines[rank]*w,w*nlines, MPI_CHAR, rank, tag, MPI_COMM_WORLD, &status);
  if actual_line >= h then
    master_send[0] ← 0;
    MPI_Send(master_send, 2, MPI_DOUBLE, rank, tag, MPI_COMM_WORLD);
    last_send ← last_send + 1;
    if last_send == p-1 then
      break;
    end if
  end if
  if actual_line < h then
    master_send[1] ← actual_line;
    store_lines[rank] ← actual_line;
    MPI_Send(master_send, 2, MPI_DOUBLE, rank, tag, MPI_COMM_WORLD);
    actual_line ← actual_line + nlines;
  end if
end while
```

the new task to do and increment the actual line. To store the number of each lines processed by the worker we used the array store_lines that contain for each element the actual line. At the end, by using probe MPI function we can access to the rank of the worker. Hence, storing the part of the image in the right place of the final output. Find bellow the final output of the computation.

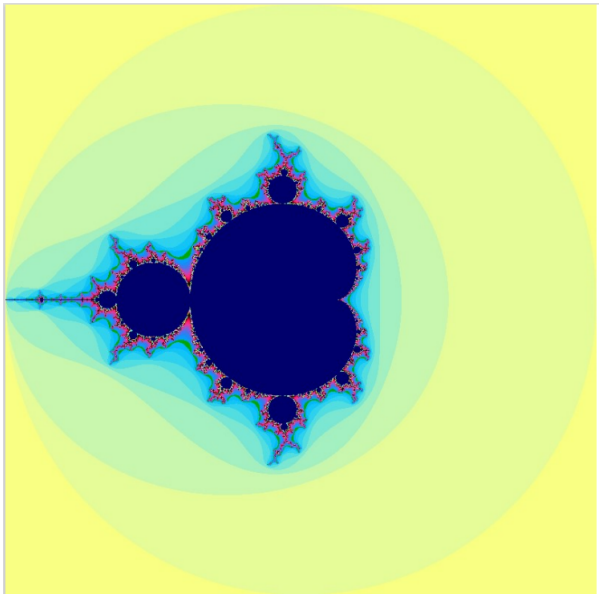

Figure 4: The result of the image of the computation

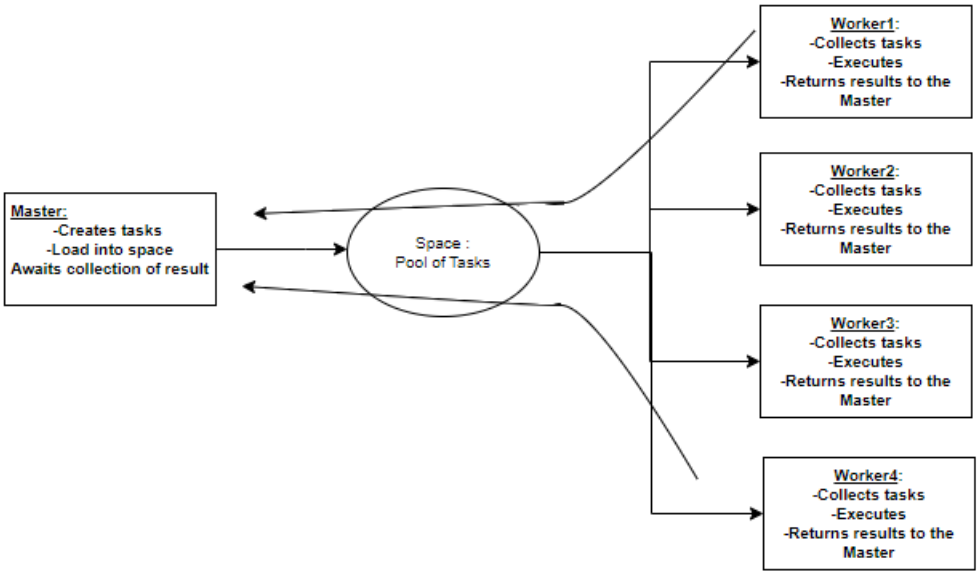

Figure 5: The image presentation with the processors Master/Worker

3 Performance:

We start by executing the code using only one processor and we got a 5 seconds time execution, with $height = 800$

Table 1 shows the time required for the execution of every processor in Algorithm 1. We can conclude that each processor has different time execution depending on the part executed, some areas in the image needs more computation than others. Also, we can see that the sum of time of the whole process is roughly equal to the time of execution of one processor.

Number of processor	Processor	Time(in seconds)
2	0	2.7
	1	2.7
4	0	0.2
	1	2.8
	2	2.8
	3	0.2
8	0	0.2
	1	0.2
	2	0.7
	3	2.2
	4	2.2
	5	0.7
	6	0.2
	7	0.2

Table 1: Time required in algorithm 1

Number of processor	Processor	Time(in seconds)
	0	5.5
2	1	5.5
	0	2
4	1	2
	2	2
	3	2
	0	1
8	1	1
	2	1
	3	1
	4	1
	5	1
	6	1
	7	1

Table 2: Time required in algorithm 2 for both nlines=4 and 20

Second algorithm nlines = 4 [same for nlines = 20]

Table 2 shows the time required for the execution of every processor in Algorithm 2, we can conclude that each processor has quite the same time execution due to the use of a small part nlines equal to 4. As soon as a process finish its job, it will ask for a new one. For this reason, the time is almost the same. And as we can see when we have only 2 processors the time is as quite the same as using only one processor. This is due to having only one worker processer that will do all the job while the other will be the master that only feed the worker with the specific task to do. Furthermore, we can remark that the sum of time in each case is greater than the time of execution of one processor. This due to the number of operation that was increased in the master-slave algorithm. In addition, when nlines is very big for example height/8, the value obtained with the different number of process is quite equal the time in the algorithm 1 Table 3.

Recall that the formula of Speedup is : $S(n,p) = \frac{T_1(n)}{T_p(n)}$. where $T_1(n)$ is the time required for the execution of the best time to solve the image of size *n* and $T_p(n)$ is the time required for solving the problem using p processors and *n* is the length of the image (height x weight) using only one processor.

Also, the Parallel efficiency is $E(n,p) = \frac{S(n,p)}{p}$, where $S(n,p)$ is the speedup.

Regarding the result obtained in Table 4 we conclude that "algorithm 2" is better than the the "algorithm 1" in term of the time of execution except in the case where we use only 2 processors. This is because in the "algorithm 2" always one algorithm is used to send the task to the worker and gather the data, so it will not do any computation. But, if we want to let the master work, then the other processor will wait until the master finish its part of computation. In this case, the time of execution will change but, at the same time we will have another processor working. As we didn't implement this method we can't conclude which one is the best.

Number of processor	Processor	Time(in seconds)
2	0	5.1
	1	5.1
4	0	2.4
	1	2.4
	2	2.4
	3	1.1
8	0	2.4
	1	0.25
	2	0.25
	3	0.6
	4	2.4
	5	2.4
	6	0.7
	7	0.24

Table 3: Time required in algorithm 2, where nlines=100,height =800

Algorithm	Number of processor	Parallel efficiency
Algorithm 1	2	100%
	4	50%
	8	25%
Algorithm 2	2	50%
	4	62%
	8	62%

Table 4: Efficiency of each algorithm

4 Testing the algorithm

To run the code in Grid5000:

4.1 Algorithm1

See the following commands :

```
$ mpicc -o mandel mandel4.c -lm
$oarsub -l cluster=1/core=4 'mpirun --mca pml ^ucx -machinefile
$OAR_NODEFILE ./mandel [w] [h] [xmin] [ymin] [xmax] [ymax] [prof] '
```

Here we test the first algorithm with 4 processors. The variable between 2 brackets are optional and if is not added, we will use theirs default values: xmin = -2; ymin= -2; xmax= 2; ymax = 2; w=h= 800; prof=10000;

4.2 Algorithm2

See the following commands :

```
$ mpicc -o mandel mandel4_best.c -lm
$ oarsub -l cluster=1/core=4 'mpirun --mca pml ^ucx -machinefile
$OAR_NODEFILE ./mandel [nlines] [master_rank] [w] [h] [xmin] [ymin]
[xmax] [ymax] [prof] '
```

Here we test the second algorithm with 4 processors. The variable between 2 brackets are optional and if they are not added, we will use theirs default values: nlines = 4; master_rank = 0; xmin = -2; ymin= -2; xmax= 2; ymax = 2; w=h= 800; prof=10000;

For the exection of the code (h/nlines) should be an integer.