# CP431: Term Project Report

**Due date:** Dec 1, 2019

**Group ID:** 5

**Authors:**

Amardeep Sarang - 160112080

Xiaochao Luo – 171422220

Liam Gordon - 150755500

# Index

## 1. Project Description

"The Julia set of a complex function is named for the French mathematician Gaston Julia, who discovered many of the basic properties of the set that bears his name in the early twentieth century. A precision definition of the Julia set of a polynomial is that it is the boundary of the set of points that escape to infinity, but arbitrarily nearby there are points whose orbits do escape." [1]

In this project, we'll develop a parallel program to generate a Julia set data points, then use the data points to generate the related image. We'll test those points offered from textbook to see how a parallel program scale up with the increasing processors.

## 2. Program Implementation

This program is being divided into two parts. First part is generating the Julia Set by using C, second part is using the generated set to generate corresponding image by Python. Parallel programming is implemented on both parts. The reason to divide the program is to make the program memory efficiency. After the Julia set is generated by the first part, all the data will be saved into several output files depends on the number of processors. Then the second part will read these data points from the output files and write it to the image file.

### 2.1 Generating Julia Set

The escape radius we use in this project is R = 2 and max iteration is 50. Then functions like $Q_c$(z) = $z^2 + c$ and $T_c$(z) = $z^3 + c$ will be used for the orbit of a point $z_0$, where z = x + iy, c = $c_1$ + i$c_2$.

First we transform screen coordinates to complex number that is scaled between **-R** and **R** using range transformation**.** Then iterate the function, stop the function if it reaches the max iteration or it lies outside of $R^2$ . Record the coordinates and the number of iterations to the output file. For more information, please check the code details in section 4.

## 2.1.1 Processors

Communication Protocol used: MPI

```
MPI_Init(&argc,&argv);
MPI_Barrier(MPI_Comm communicator);
MPI_Wtime();
MPI_Comm_rank(MPI_Comm communicator, int* rank);
MPI_Comm_size(MPI_Comm communicator, int* processor);
```

Processor Count: p, in range[1, 126]
Rank of Processor: 0 to p - 1
The Processor of destination: None

### 2.1.2 Workload Concept

Each processor will get evenly distributed rows to work with. For example, an image with 100 x 100 grid with 4 processors, then each processor will get 25 rows to record the iterations of each point on the row.

To determine the row r for processor i, we use :

**Row r % p == my_rank i**

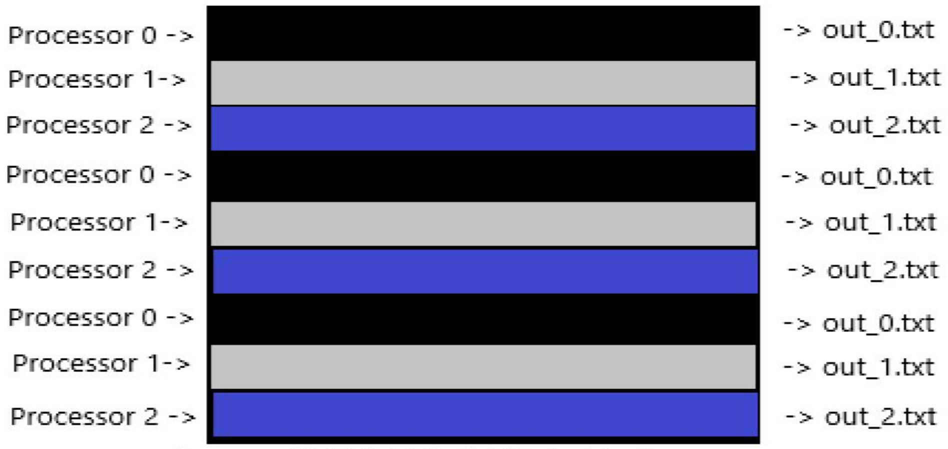


Image with Height 9 distributed to 3 processors

When generating Julia set, communication is not required. Each processor will write the data into an output file name "out_i.txt" in a format of (height, width, iterations).

## 2.2 Generating Image

To generate a smooth and nice-looking Julia set coloring that is close to the textbook's example, a hue saturation value color model will be used. The saturation is always set to 255. Then the hue is set to the percentage of iterations that were completed before the escape radius is reached. It then multiplied by 255 as the following equation: Hue = 255 * iterations / max_iterations. Finally the value parameter is set to 255 if the point escaped before the maximum number of iterations and 0 otherwise. These values are then converted into rgb color. This idea is inspired from **coding game**[2]**.**

### 2.2.1 Processors

In this part, **mpi4py** is used for faster development purpose. However, it does sacrifice some performance and cost more on memory. In order to make the whole program persistent, the number of processors in this part will be the same as the previous part. So, if 4 processors are used to generate the Julia set, then, 4 processors will be used to generate the image.

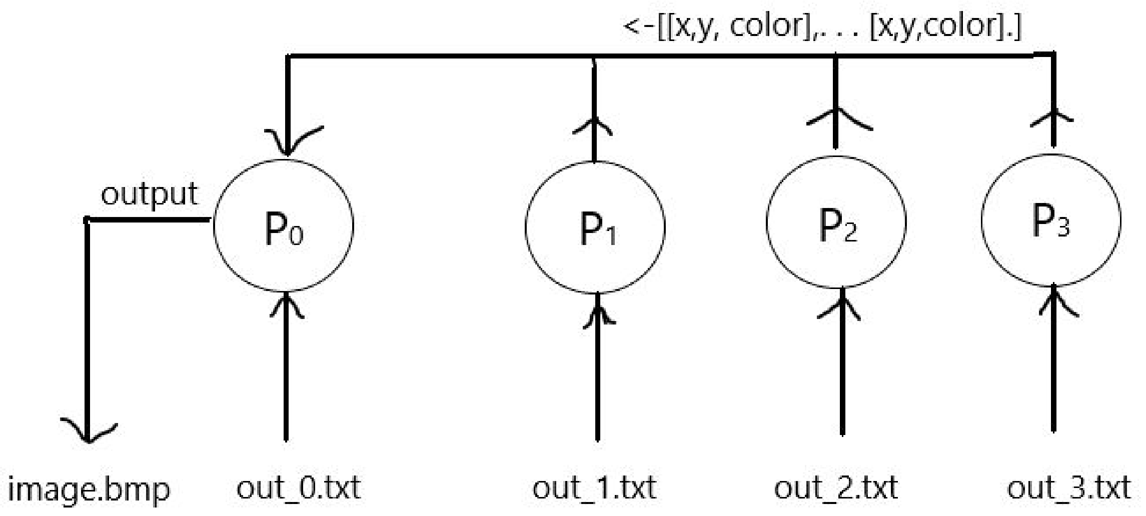
Communication Protocol used: **MPI** from **mpi4py**

```
MPI.COMM_WORLD
MPI.COMM_WORLD.Get_size()
MPI.COMM_WORLD.Get_rank()
MPI.COMM_WORLD.recv(source, tag)
MPI.COMM_WORLD.send(data, dest, tag)
```

Processor Count: p, in range[1, 126]
Rank of Processor: 0 to p - 1
The Processor of destination: 0

### 2.2.2 Workload Concept

The idea is plain and simple. Suppose we have 4 output files that is generated by processors from previous part. Then 4 processors in this part will load the data from output files individually. Processor 0 as the destination processor will receive all the data from other processor and convert them to pixels and save the image file after it is done.

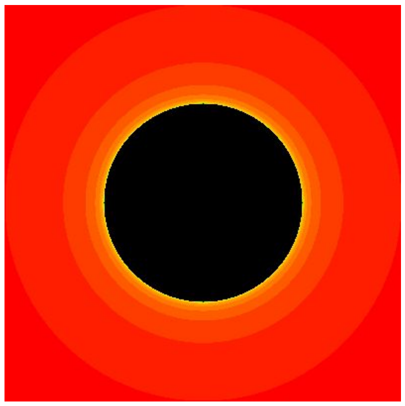
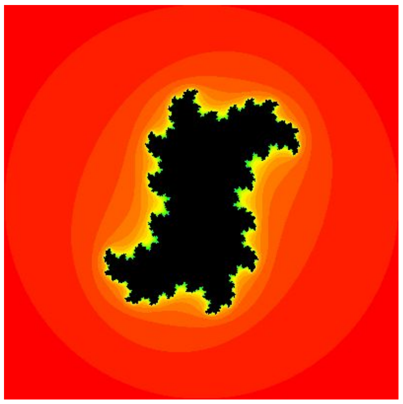


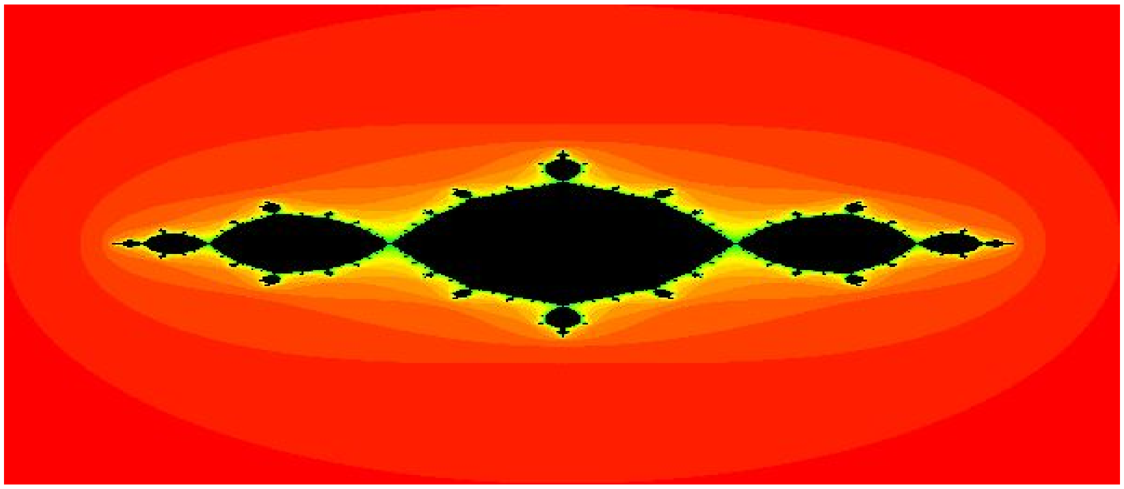
## 3. Example outputs and benchmarks

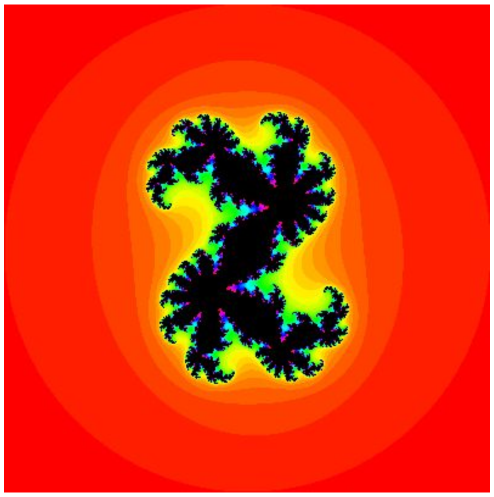
All the examples are using **R** = 2 and **max iteration** = 50

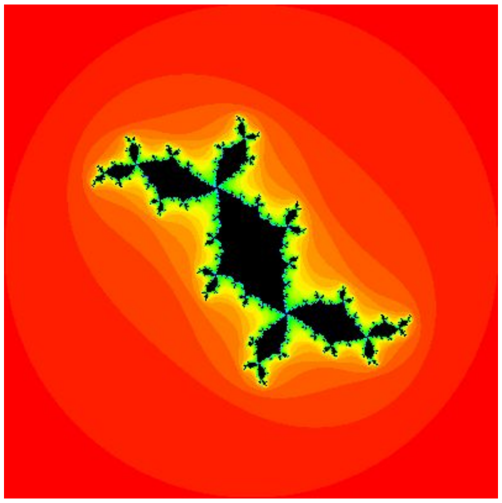
### 3.1 $Q_c$ **(z) =** $z^2 + c$

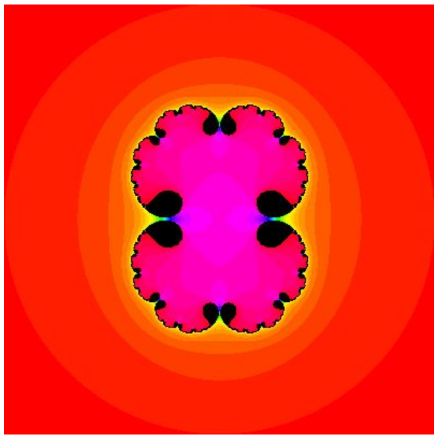
The size of image pasted below were from 500x500 or 400x400 and it is readjusted to fit the table better.

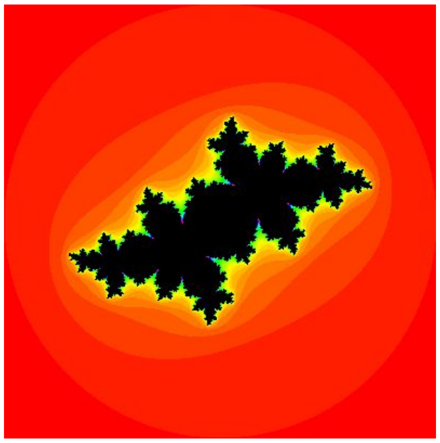
| | |
|---|---|
|  C = 0 |  C = 0.3 - 0.4i |

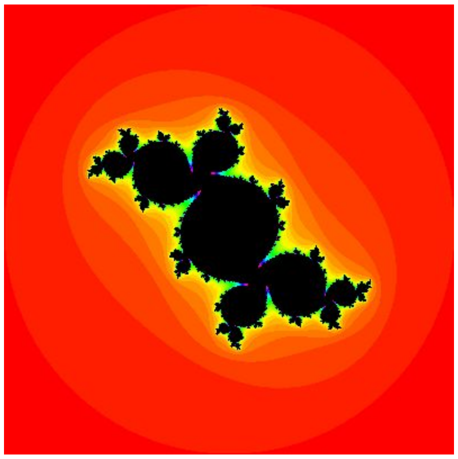
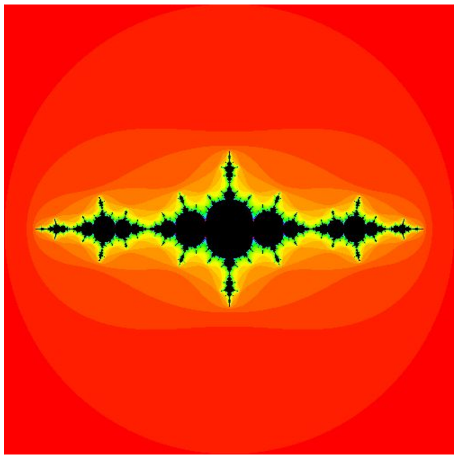

C = -1


C = 0.360284+0.100376i

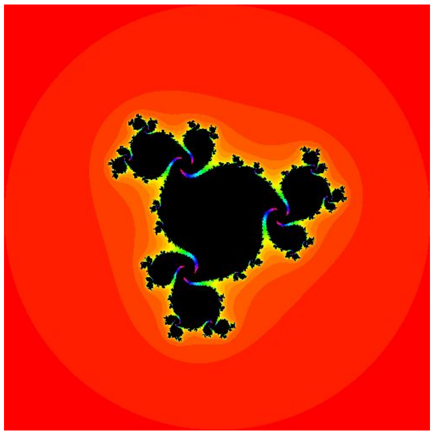
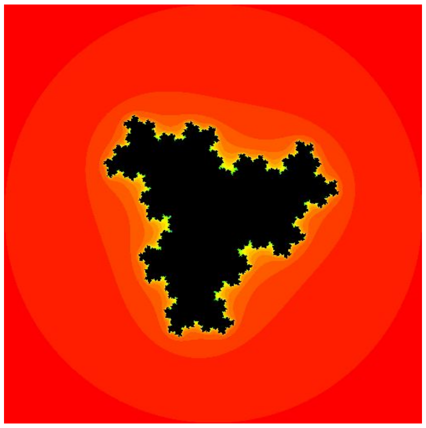
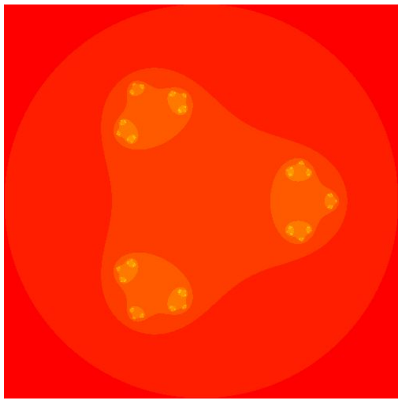
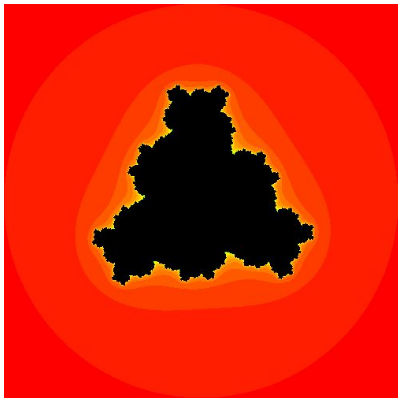

C = -0.1 + 0.8i


C = 0.255


C = -0.48176 - 0.53165i

| | |
|---|---|
|  C = -0.11 + 0.67i |  C = -1.25 |

## 3.2 $T_c$ (z) = $z^3 + c$

| | |
|---|---|
|  C = -0.48176 - 0.53165i |  C = -0.39054 - 0.58679i |
|  C = -1.25 |  C = -0.11 + 0.67i |

## 3.3 Benchmark

1. In this section, R = 2 and max iteration = 50.
2. only $Q_c$ **(z) =** $z^2 + c$ is used to test parallelization efficiency
   Where c = -1
3. 10k x 10k cases for both generating Julia set and image
4. 100k x 100k cases for only generating Julia set
5. Processor numbers from 1 to 8.
6. Time unit is seconds(s)

### 3.3.1 10k x 10k case

| **Processors** | **Generate J.Set** | **Generate Image** | **Total Time** | **Serial Time** | **AMD Speedup** |
|---|---|---|---|---|---|
| 1 | 21.75345 | 524.0412 | 545.7946 | Not Applicable | Not Applicable |
| 2 | 10.48993 | 482.4233 | 492.9132 | 224.3065 | 1.374511 |
| 3 | 6.97659 | 486.5431 | 493.5196 | 156.542 | 1.835547 |
| 4 | 5.261807 | 471.3893 | 476.6511 | 114.6981 | 2.323016 |
| 5 | 4.335344 | 465.9448 | 470.2802 | 92.35041 | 2.800348 |
| 6 | 3.53299 | 463.8277 | 467.3607 | 76.01093 | 3.530579 |
| 7 | 3.051755 | 472.4132 | 475.465 | 66.57721 | 3.804032 |
| **8** | **2.719109** | **466.5513** | **469.2704** | **57.93739** | **4.291296** |

Apparently 8 processors have the best result. However, notice that 8 processors would require at least 60 gigabytes of memory while 1 processor would only require 15 gigabytes. If look closer, the scale up of the program by increasing the processor is not as good as expected. As the following graphs would show more details.

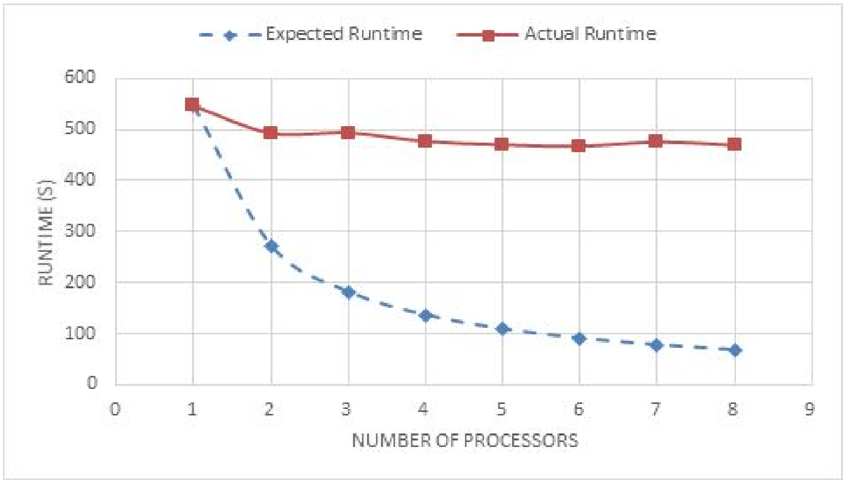


Figure 1: Expected vs. Actual runtime (including time to generate image) on 1 to 8 processors for a 10k x 10k resolution Julia set.

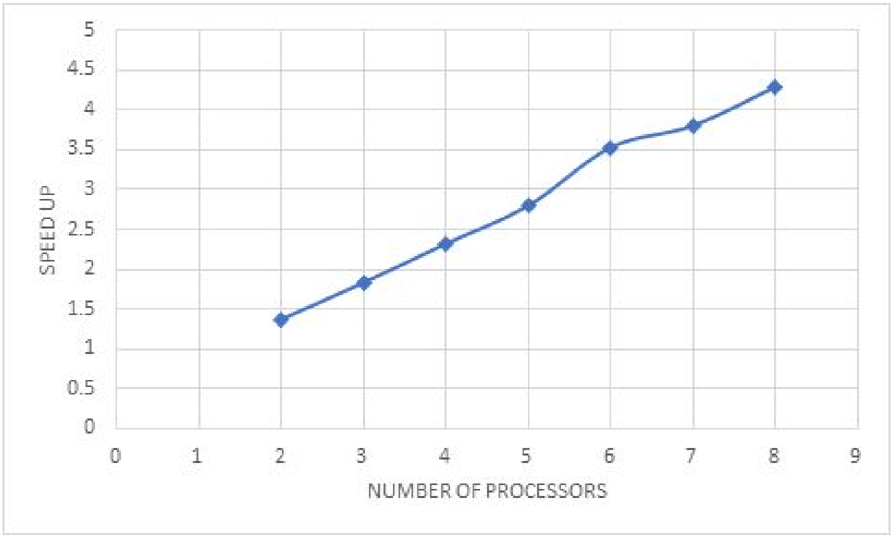
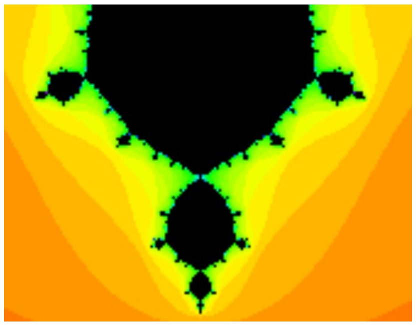


Figure 2: speed up value with n processors for a 10k x 10k resolution Julia set.

AMD's speedup ignores the overhead cost, which is why it looks good. For a 10k x 10k image, a total of 1.1 gigabytes of dataset is required to generate the image. The overhead would be massive if more processors are used and the memory would be another challenge.

A peek of a corner of **10k** x **10k** image after zoom in 100 times for C = -1.



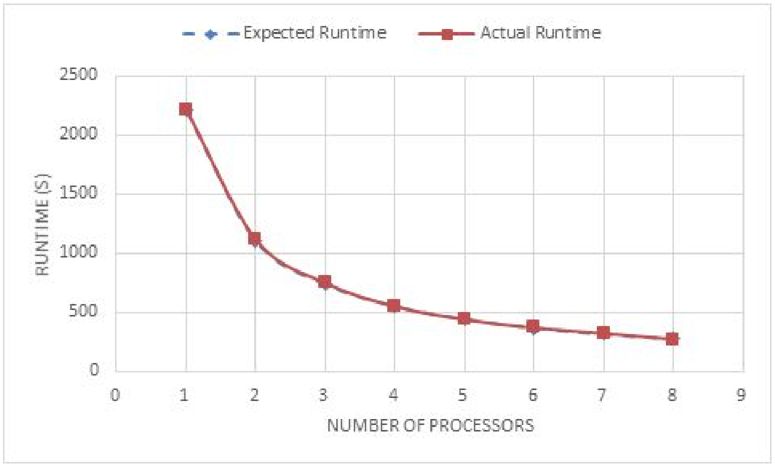
A 10k x 10k image would have size around 286 megabytes, and it is still clear after zoom in 100 times.

### 3.3.2 100k x 100k case

In this section, only the time of generating Julia set will be recorded.

| **Processors** | **Generate J.Set** |
|---|---|
| 1 | 2217.68 |
| 2 | 1117.786 |
| 3 | 748.1044 |
| 4 | 550.8125 |
| 5 | 438.3848 |
| 6 | 370.0983 |
| 7 | 323.4477 |
| **8** | **275.5305** |

The program does scale up with more processors.



## 3.4 Conclusion

The result from 10k x 10k seems reasonable. Although the actual running time is not converging to expected running time, 8 processors case is still 100 seconds faster than 1 processor. It is not running as “expected”, but it does scale up. One conclusion we can draw here is in parallel programming, the size of the data to send and receive would play a very important role.

For 100k x 100k image, due to the size of the data, it seems impossible to the image with the parallelization code we created unless we could be granted for more memory, at least more than 128 gigabytes. Otherwise the program would soon run out of memory. However, serial code with only 1 processor seems capable of that. The only offset is it takes a very long time to finish the job. Theoretically it would need at least 15 hours. How to balance between resources and processors is definitely one of the challenges for this project. Another would be development time vs. running time. Using Python’s Pillow library could generate the image very easily, but it takes more time and resources as it is not a compiled language.

Overall, the most challenging part is to submit the right batch job at the right time. It is really devastating when a large memory consuming job is getting CPU time, then running out of memory after 10 hrs. It would really affect our plan on everything. Sometimes, during the peak time, a small batch job would require a lot of waiting time, which delayed the next stage programming.

# 4. Code details

Only the necessary code will be displayed at this section, the trivial code will be neglected as usual. To see the full code and other generated results, please go to **xluo/project** folder on graham. 10k x 10k and 100k x 100k results are taking way too much rooms, to see the result please go to scratch/project folder, but it will eventually be purged in 60 days.

## 4.1 Code for generating Julia set in C

```
//Range transformation helper function to scale x and y in range of R and -R.
float scale_value(float old_value, float old_max, float old_min, float new_max, float new_min) {
        return ((old_value - old_min) / (old_max - old_min)) * (new_max - new_min) + new_min;
        }

//basic settings, n could be 2 or 3
int my_rank, p, dest = 0, int source, height = 500, width = 500, n = 2;
double elapsed_time = 0;

float R = 2; //escape radius
float cx = 0.3, cy = -0.4; //initial c value

//create an output file for each proc
char f_name[12];
snprintf(f_name, 12, "out_%d.txt", my_rank);
FILE *f = fopen(f_name, "w");

//this strategy just splits rows into chunks and gives a chunk into each processor
//each chunk has continuous rows

for (int i = 0; i < height; i++) {
    if(i % p == my_rank){
        for (int j = 0; j < width; j++) {
            int ycord = height - 1 - i;
            //put 0,0 at center
            int y = ycord - height / 2;
            int x = j - width / 2;

             //scale coordinates to be between -R and R
            float zx = scale_value(x, width / 2, -1 * width / 2, R, -1 * R);
            float zy = scale_value(y, height / 2, -1 * height / 2, R, -1 * R);
```

```
                int iteration = 0;
                int max_iteration = 50;

               //To calculate Qc or Tc value within max iteration
                while ((zx * zx + zy * zy < R * R) && (iteration < max_iteration)) {
                if (n == 2) {  //iteration value for Qc, from wikipedia [3]
                        float xtemp = zx * zx - zy * zy + cx;
                        float ytemp = 2*zx*zy + cy;
                        zy = ytemp;
                        zx = xtemp;
               }else if (n == 3){  //iteration value for Tc
                        float xtemp = pow(zx * zx + zy * zy,n/2)*cos(n*atan2(zy, zx));
                        zy = pow(zx * zx + zy * zy,n/2)*sin(n*atan2(zy, zx)) + cy;
                        zx = xtemp + cx;
               }
               iteration = iteration + 1;
           }
           //write down the data to the output file
           fprintf(f, "%d %d %d\n", i, j, iteration);
      }
 }

//calculate the total time (here time is parallel only)
if (i == height - 1 && my_rank == i%p) {
           elapsed_time += MPI_Wtime();
            printf("total elapsed time is %f\n", elapsed_time);
            fclose(f);
}
```

## 4.2 Code to Generate Julia Set Image in Python

```
#basic settings
comm = MPI.COMM_WORLD
size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()



def hsv2rgb(h,s,v):
          #helper function
          #use this one to get results similar to text book
          return tuple(round(i * 255) for i in colorsys.hsv_to_rgb(h,s,v))

def colorer_hsv(iterations,max_interations):
          #set hue, saturation and value
```

```
        s=255
        h=int(255*iterations/max_interations)
        v = 255 if iterations<max_interations else 0
        #convert to rgb
        s=s/255
        h=h/255
        v=v/255
        return hsv2rgb(h,s,v)



def main():
        #get arguments
        in_file_path = sys.argv[1]
        max_interations = 50
        if len(sys.argv)>5:
            max_interations = int(sys.argv[5])
            if rank == 0: #rank 0 doing the initial work to create an image object
                out_file_path = sys.argv[2]
                width = int(sys.argv[3])
                height = int(sys.argv[4])
                img = Image.new( 'RGB', (width,height), "black") # Create a new black image
                pixels = img.load() # Create the pixel map
                if os.path.exists(in_file_path + str(rank)+".txt"):  #read from the data file
                    file_path = in_file_path + str(rank)+".txt"
                    file_in = open(file_path, "r")
                    for line in file_in:
                        line_arguments = line.strip().split()
                        #get coordinates and number of iterations before escape
                        i=int(line_arguments[0])
                        j=int(line_arguments[1])
                        iterations=int(line_arguments[2])
                        # color each pixel based of iterations before escape
                        pixels[j,i]=colorer_hsv(iterations,max_interations)

                        for s in range(1, size):
                            elapsed_time_series = time.time()  #that's where serial time begin
                            data = comm.recv(source = s, tag=1) #receive the data from other processor
                                for d in data:
                                    j = d[0]
                                    i = d[1]
                                    pixels[j,i] = colorer_hsv(d[2],max_interations)

                #code for calculating running time and speedup
                elapsed_time_series = time.time() - elapsed_time_series
                print("serial time: %s seconds" %elapsed_time_series)
                img.save(out_file_path+".bmp")
                elapsed_time_total = time.time() - elapsed_time
                print("elapsed time total: %s seconds" %elapsed_time_total)
                elapsed_time_parallel = elapsed_time_total - elapsed_time_series
                print("elapsed time parallel: %s seconds" %elapsed_time_parallel)
                f = elapsed_time_series/(elapsed_time_series+elapsed_time_parallel)
                speedup = 1/(f+(1-f)/size)
                print("AMD law speedup:",speedup)
        else:
            if os.path.exists(in_file_path + str(rank)+".txt"): #check if the data file exists
                result = []
                file_path = in_file_path + str(rank)+".txt"
                file_in = open(file_path, "r")
                    for line in file_in:
                        temp = []
                        line_arguments = line.strip().split()
                        i=int(line_arguments[0])
                        j=int(line_arguments[1])
                        iterations=int(line_arguments[2])
                        temp.append(j)
                        temp.append(i)
                        temp.append(iterations)
                        result.append(temp)    #read the data and append it to the result array
                comm.send(result, dest=0,tag=1) #send the data to processor 0
```

## 4.3 Code for Batch Files

Code for 10k x 10k Dataset on 8 processors

```
#!/bin/bash
#SBATCH --ntasks=8
#SBATCH --time=0-00:1
#SBATCH --mem-per-cpu=400M
#SBATCH --output=10k8core.out
srun ./test 10000 10000
```

Code for 10k x 10k image on 8 processors

```
#!/bin/bash
#SBATCH --time=0-00:10
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --mem=63G
#SBATCH --output=10k8corei.out

module load python/3.6
```

```
source ~/ENV/bin/activate
mpirun python mpipy.py out_ 10k 10000 10000
```

## 5. Reference

[1] Robert L. Devaney (1990). Chaos, Fractals, and Dynamics Computer Experiments In Mathematics. Addison-Wesley Publishing Company, Inc.

[2] Maxime Cheramy. Adding colors to the Mandelbrot Set. https://www.codingame.com/playgrounds/2358/how-to-plot-the-mandelbrot-set/adding-some-colors

[3] Julia set.Wikipedia. https://en.wikipedia.org/wiki/Julia_set