THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC4005 PARALLEL PROGRAMMING

Homework 2 Report

Name: Xiang Fei
Student ID: 120090414
Email: xiangfei@link.cuhk.edu.cn
Date: 2022.10

Table of Contents

List of Figures

List of Tables

1 Introduction

In homework 2, we are required to design a program for Mandelbrot set computation and render the points on screen with a type of GUI. The Mandelbrot set is a set of points in the complex plane that is quasi-stable when computed by an iterative function:

$$z_{k+1} = z_k^2 + c \tag{1}$$

$$z^2 = a^2 + 2abi + bi^2 = a^2 - b^2 + 2abi \tag{2}$$

$$z_{real} = z_{real}^2 - z_{image}^2 + c_{real} z_{imag} = 2 z_{real} z_{imag} + c_{imag} \tag{3}$$

Here, z is a complex number in the plane and k represents the iteration number. The parameter c here is calculated by:

$$c = \frac{x - height/2}{height/4} + \frac{y - width/2}{width/4} \times i \tag{4}$$

Scaling c with different calculation, the generated shape will change accordingly.

However, if we apply the sequential method to compute it, the computation time becomes quite large as the image size increases. In theory, if we choose a fixed side length for the image, sequential computation will count pixels one by one. Therefore, this project aims to study different versions of parallel computing to optimize the computing speed of Mandelbrot sets. In this experiment, I designed three parallel computing versions, including MPI, static Pthread and dynamic Pthread, and compared these versions with the Squential version to find out the version with the best computing performance in different scenarios

The basic problem and task is: given a set of numbers, regardless of the size of the picture and the number of cores/threads, the program should perform parallel computing strictly according to the rules of Mandbrot set computing. Run parallel computing programs in a cluster to obtain experimental data and analyze performance in specific cases such as different cores/threads and different input sizes

Apart from the implementation, my project would also investigate the following specific questions:

1. Investigation of the relationship between number of processes/threads and the performance of parallel computing via MPI or Pthread.

2. Investigation between sequential computing and multi-threads computing.

3. Investigation of the relationship between problem size and the performance of parallel computing via MPI or Pthread.

4. Comparation between dynamic scheduling and static scheduling for pthread case.

5. Comparation of different features between MPI and Pthread under different situations.

The rest of the report consists of four main sections: **Design and Methods**, **Execution**, **Result & Analysis**, and **Conclusion**. The **Design and Methods** describes how I designed my three versions of parallel computations for specific motivations and how to implement these codes. The **Execution** shows you the files I included to complete the project and how to run it on the server. It also includes a script file to show you how I get my experiment data on the server. The **Result & Analysis** is a performance comparison between those methods that implement the Mandelbrot computation. I set different variables for these programs to see which would perform better under which circumstances. The **Conclusion** part focuses on the analysis results, and gives a table of the analysis results between the Mandelbrot set calculation programs.

2 Design and Method

2.1 Code Implementation

2.1.1 Pthread Design

For the implementation via Pthread, the program basically uses dynamic scheduling to distribute the workload to each thread and let the root thread do the I/O work. Apart from some I/O and GUI, my design mainly consists of three parts:

1. Pixel Distribution
2. Pthread Initialization
3. Parallel Computing and Dynamic Scheduling.

Pixel Distribution:

This part is mainly to prepare for dynamic scheduling, partition pixels, and different areas are calculated by different threads. For $n \times n$ dimensions, the pixels will be equally divided into n columns. With dynamic scheduling, each thread can approach each block. The division here uses block division. So the partition is like the following:

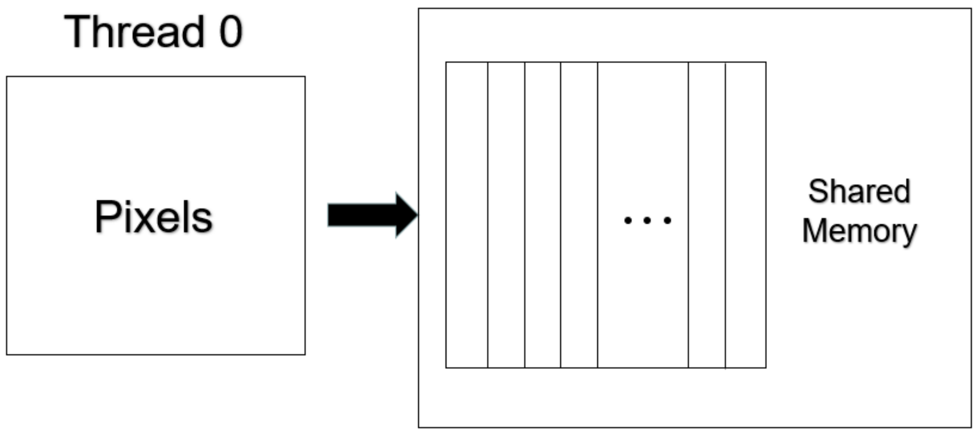


Figure 1: The illustration of pthread partition

Pthread Initialization:

Numbers of Pthreads would be initialized and executing the function worker, which is the function to be executed parallelly. In addition, the thread id of each thread and the number of threads would be passed as argument by the Args structure defined myself.

```
pthread_create(&thds[thd], NULL, worker, &args[thd])
```

In addition to pthread initialization, mutexes are initialized to protect shared data. In this way, the same block of pixels can only be calculated once by one thread.

Parallel Computing and Dynamic Scheduling:

The basic idea of implementing the dynamic scheduling:

- At the beginning, each thread competes for a mutex to access a block of pixels. When one has accessed the pixel block, it releases the lock and other threads will compete for the block next to it until all threads are counting pixels.

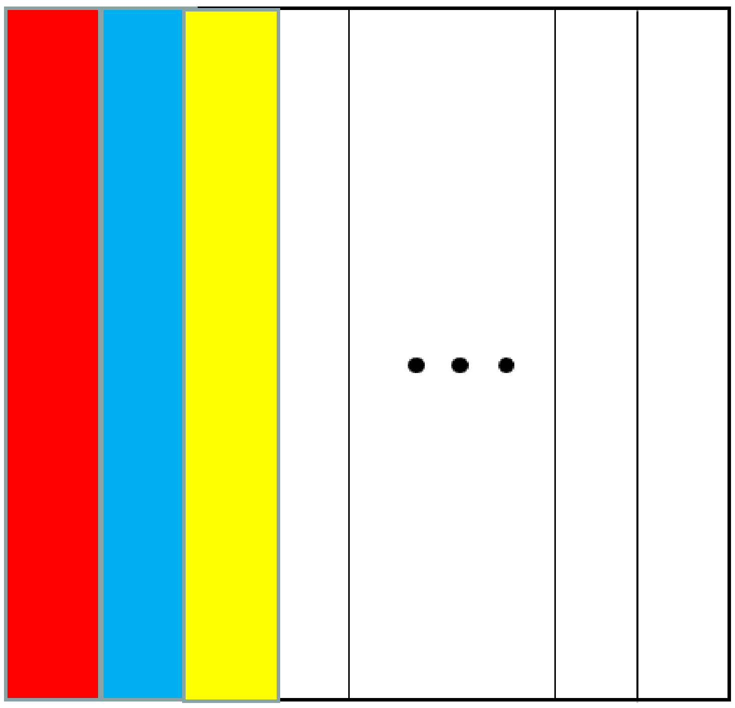


Figure 2: : blocks of pixels are initially assigned to each thread (suppose 3 threads here)

- Then one thread might finish the computation earlier than the others, so that thread will acquire the mutex and allocate the next block of pixels. So once a thread has finished its local work, it will compete for the mutex and get the base and length of the next block of pixels. This strategy will guarantee a balance of total pixels allocated to each thread, reducing bottlenecks and improving overall performance (Analysis of static and dynamic scheduling will be studied later.)

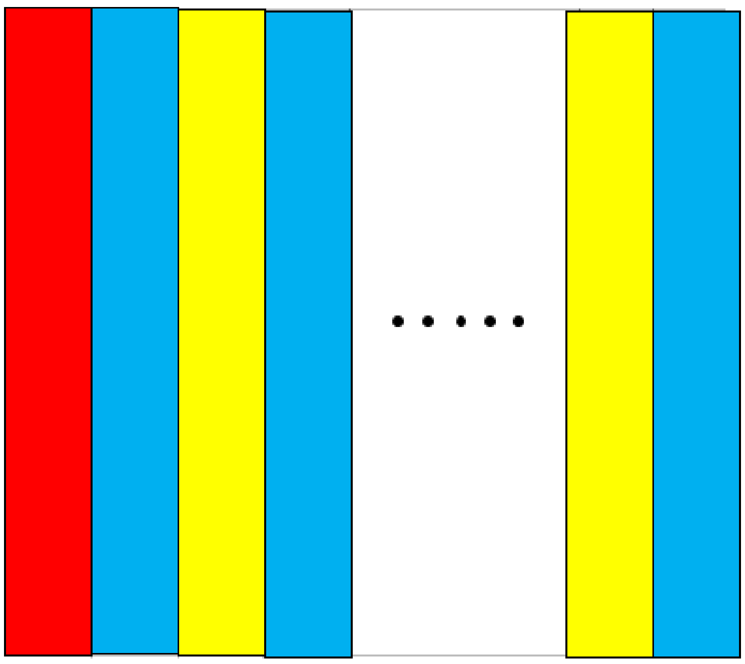


Figure 3: blocks of pixels distributed to each threads during parallel computing

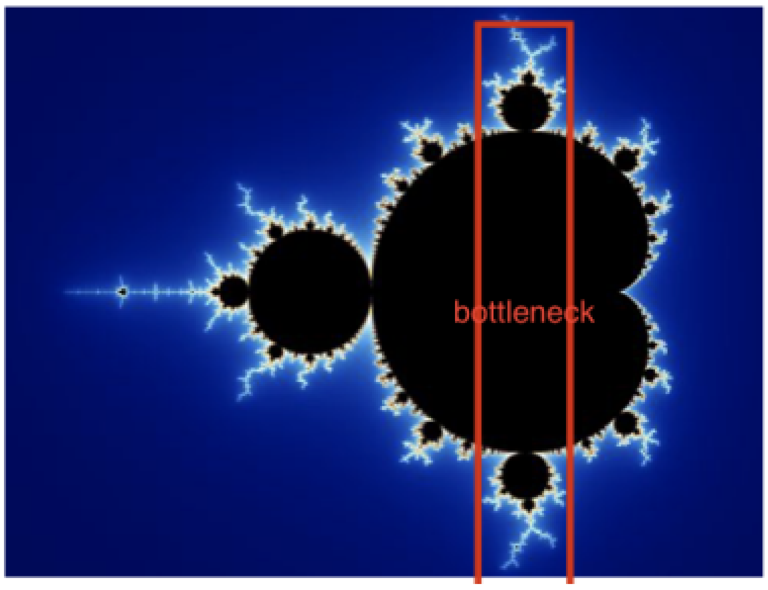


Figure 4: The schematic diagram of bottlenecks

Remarks: As shown, some threads, process fewer tiles than others, due to the fact that their tiles have more work than others.

- I divide task into blocks, which means that if the picture is 800×800 pixels, then it will be divided into 800 clocks. And each thread will calculate one block at a time.
- After all pixel blocks are calculated, the thread will exit and the root thread will be responsible for I/O.

Static scheduling:

- To study the difference in performance between dynamic scheduling and static scheduling, I also designed a program that does exactly the same idea as above except for the scheduling policy. This means that all pixels will be distributed equally to each thread at the beginning.
- However, we need to consider the remainder case. If size is not divisible by the number of threads, then we need to deal with this remainder case. For threads with a thread ID less than the remainder, the block width they process will increase by 1, so that all pixels can be calculated.

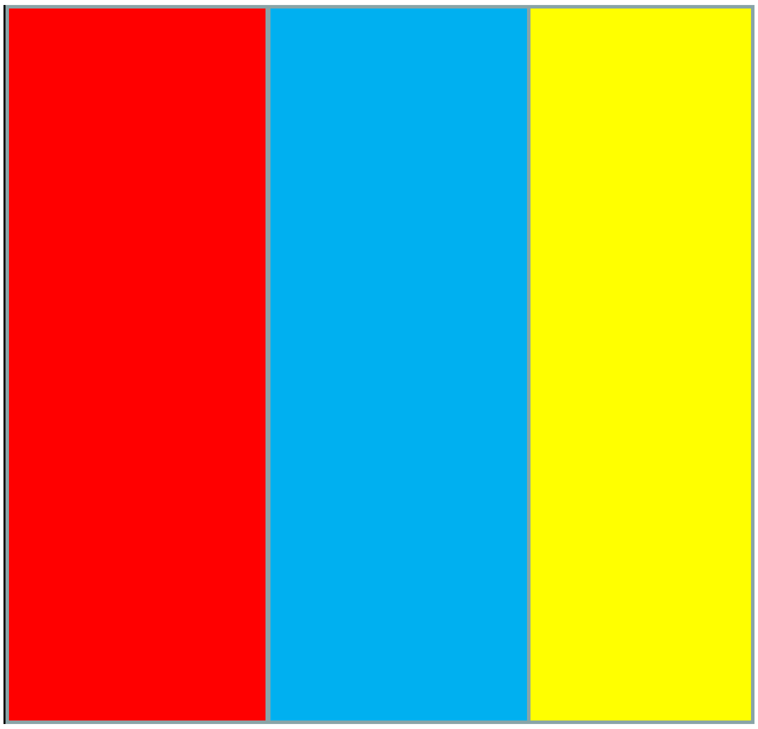


Figure 5: static scheduling with evenly distributed (suppose 3 threads here)

Overall diagram of Pthread design:

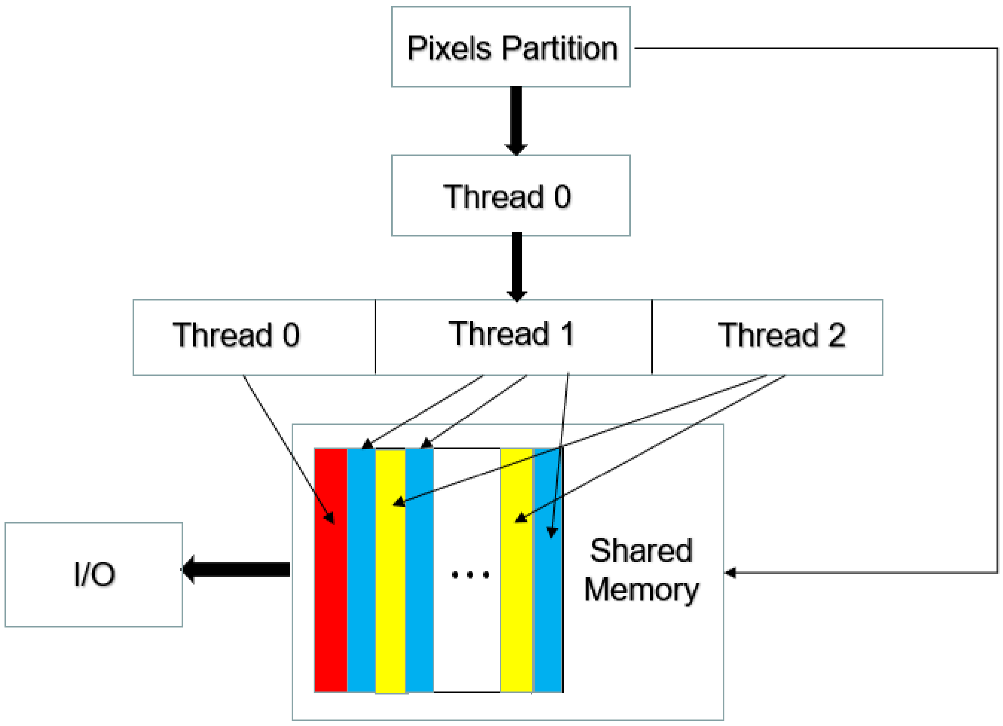


Figure 6: Overall diagram of Pthread design

2.1.2 MPI Design

Design of MPI program are mainly composed of 5 parts:

- Construct the mpi_pointtype structure
- Scatter the point array
- Parallel computing
- Result gathering
- Output the result

Construct the mpi_pointtype structure:

When using MPI, communications should be used as little as possible: instead of sending several small communications, we are better off sending one large one at a time. This is easy to do if the data is contiguous and of the same data type (like in an array). But sometimes, you may need to send discontinuous information, and the types of these data are not uniform (such as some member variables in an object). At this point we need to create our own MPI variable types (or even other data types that you create yourself) by including some simple MPI types.

```
MPI_Aint displacement[3] = {0,4,8};
int blockLength[3] = {1,1,1};
int types[3] = { MPI_INT, MPI_INT, MPI_FLOAT };
MPI_Datatype mpi_pointtype;
```

```
MPI_Type_create_struct(
        3,
        blockLength,
        displacement,
        types,
        &mpi_pointtype
);
MPI_Type_commit(&mpi_pointtype);
```

Scatter the point array:

Next, we need to allocate the part that each process needs to calculate, and mpi_scatter to achieve.

```
int my_length = X_RESN / world_size;
int remainder = X_RESN % world_size;

Point my_point[my_length*Y_RESN];
MPI_Scatter(data,my_length*Y_RESN,mpi_pointtype,my_point
,my_length*Y_RESN,mpi_pointtype,0,MPI_COMM_WORLD);
```

we also need to consider the remainder case here. If size is not divisible by the number of processes, then we need to deal with this remainder case. Here we just use the master process to do the remaining work.

Parallel computing:

This part, all processes will conduct their task and compute the result of each block.

```
for (int i=0; i<my_length*Y_RESN; i++) {
        compute(&my_point[i]);
}
```

Then, deal with the remainder case:

```
if(rank==0){
        for (int i=0;i<remainder;i++){
                compute(&data[my_length*Y_RESN*world_size+i]);
        }
}
```

Result gathering

In this part, the computed result of each processes will gather and become the final complete results.

```
MPI_Gather(my_point,my_length*Y_RESN,mpi_pointtype,
data,my_length*Y_RESN,mpi_pointtype,0,MPI_COMM_WORLD);
```

Overall diagram of MPI design

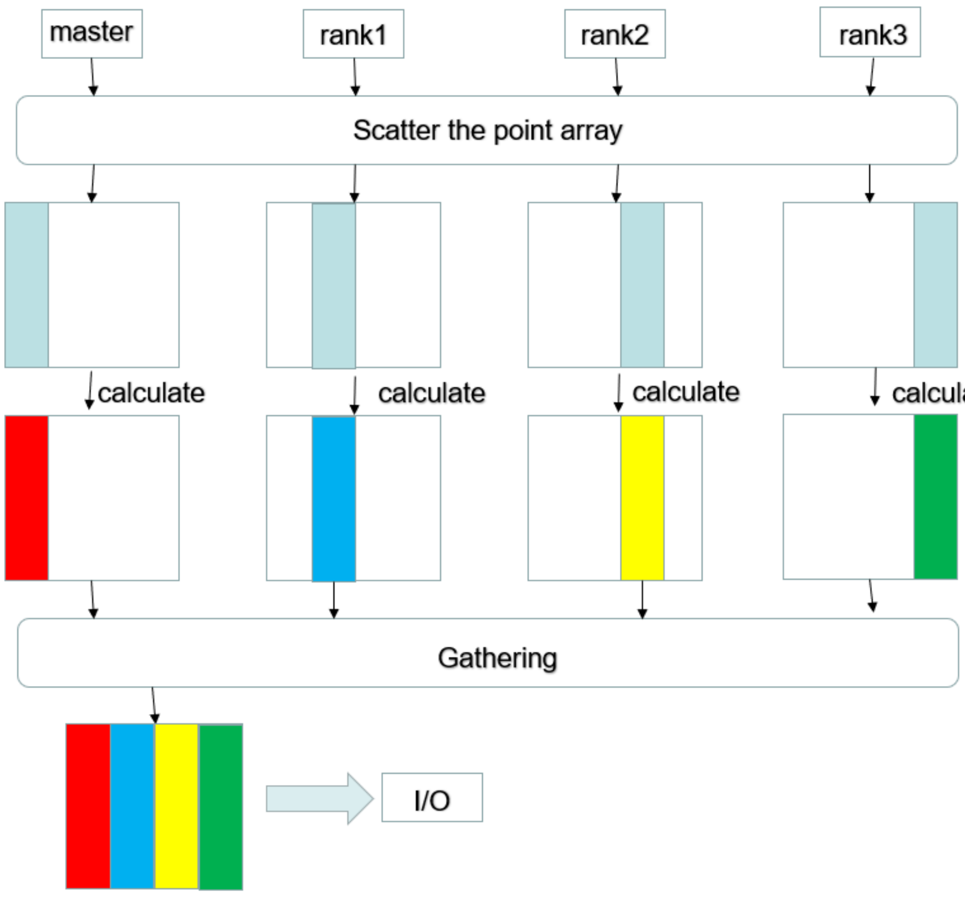


Figure 7: Overall diagram of MPI design

2.2 Experiments Design

Totally five topic will be investigated in this projects, which are:

1 Number of processes/threads

2 Problem size

3 MPI and Pthread

4 Sequential computing and multi-threads computing

5 Dynamic scheduling and static scheduling

- **Number of processes/threads**

  In this experiment, I set the problem size to be constant and in middle-size (800), then for pthread case, change the number of threads from 1 to 20. What's more, the scheduling strategy I adopted here is static scheduling. (Discussion about static and dynamic scheduling will be conducted in experiment 5) Then, I observe the tendency of performance change (using speedup compared with sequential one here) as the number of threads change.

And for MPI case, 1-40 cores are tested on the problem with size equal to 800 using static scheduling.

- **Problem size**

  In this experimental design, the problem size was selected as three modes: small (800, 1600, 3200), medium (5000) and large (10000, 20000). Also, to be more general, I tested different problem sizes on 20 threads. Then, a trend of performance change with increasing problem size will be observed. Here, I use dynamic pthread case.

- **MPI and Pthread**

  In this design, I mainly focus on the comparison of MPI and Pthread on parallel Mandelbrot set computation. To be fair, both MPI and Pthread scheduling policies are static scheduling. What's more, the comparison focuses on two dimensions. The first dimension is the number of cores/threads. 1, 2, 4, 8, 16, 32 Cores/threads will be tested for the same problem and compared. The second dimension is the size of the problem. 200, 400, 800 will be tested.

- **Sequential computing and multi-threads computing**

  Compare sequential and parallel performance of Pthreads in a broader way. Sequential calculations will be performed first. Then 5 thread choices (2, 4, 8, 16, 32) were tested on 6 different problem sizes (200, 400, 800, 1600, 5000, 10000). Then we expect a 5*6 computational speedometer. To compare each core selection with sequential cores, I use the speedup (speed of pthread computation/sequential computation) as a criterion. Then, the key points of observation are: 1. Whether Pthread has better performance than sequential thread in different situations. 2. How much performance can Pthread improve.

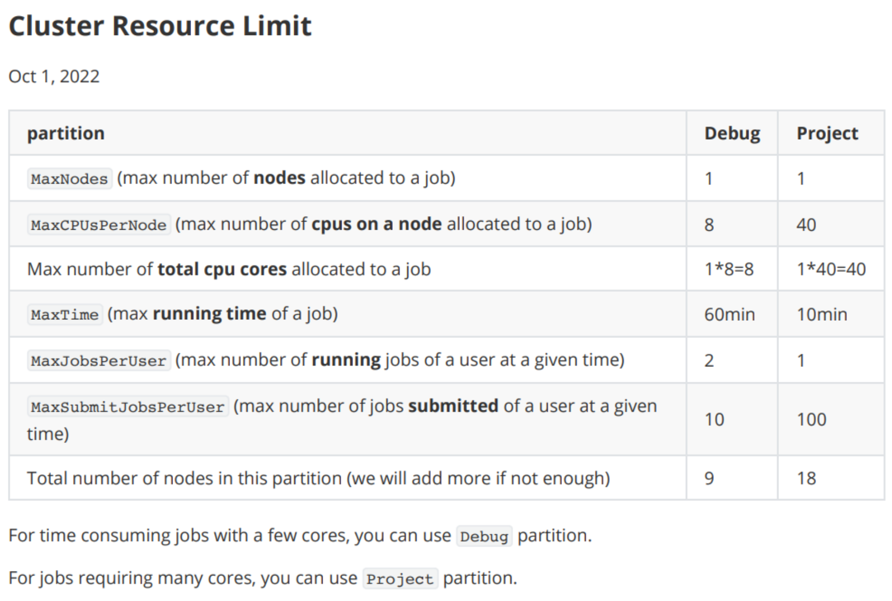
- **Dynamic scheduling and static scheduling**

  In this design, I implemented two scheduling methods, static scheduling and dynamic scheduling. To compare their performance, problems of size equal to (200, 400, 800, 1600, 5000, 10000) were tested on (2, 4, 8, 16, 32) threads with dynamic and static scheduling. Then we expect a 5*6 computational speedometer. To compare these two implementation, I use the speedup (speed of dynamic pthread computation/static pthread computation) as a criterion. Then, the key points of observation are: 1. Whether dynamic one has better performance than static one in different situations. 2. How much performance can dynamic scheduling improve.

2.3 Sequential computing and multi-threads computin

3 Execution

My code executes on the remote cluster. The cluster resource limits is showed in the following graph.



Cluster Resource Limit

Oct 1, 2022

partition	Debug	Project
`MaxNodes` (max number of **nodes** allocated to a job)	1	1
`MaxCPUsPerNode` (max number of **cpus on a node** allocated to a job)	8	40
Max number of **total cpu cores** allocated to a job	1*8=8	1*40=40
`MaxTime` (max **running time** of a job)	60min	10min
`MaxJobsPerUser` (max number of **running** jobs of a user at a given time)	2	1
`MaxSubmitJobsPerUser` (max number of jobs **submitted** of a user at a given time)	10	100
Total number of nodes in this partition (we will add more if not enough)	9	18

For time consuming jobs with a few cores, you can use `Debug` partition.

For jobs requiring many cores, you can use `Project` partition.

Figure 8: The cluster resource limit

3.1 Static pthread

- **compile the code (without GUI)**

```
g++ static_pthread.cpp -o spthread -lpthread -O2 -std=c++11
```

- **compile the code (with GUI)**

```
g++ spthread.cpp -o spthreadg -I/usr/include -L/usr/local/lib -L/usr/lib -lglut
-lGLU -lGL -lm -lpthread -DGUI -O2 -std=c++11
```

- **run the code**

  sbatch script

```
#!/bin/bash
#SBATCH --job-name=your_job_name # Job name
#SBATCH --nodes=1                    # Run all processes on a single node
#SBATCH --ntasks=1                   # number of processes = 1
#SBATCH --cpus-per-task=20 # Number of CPU cores allocated to each process
#SBATCH --partition=Project           # Partition name: Project or Debug

./spthread 1000 1000 100 4
./spthread 1000 1000 100 20
./spthread 1000 1000 100 40
./spthread 1000 1000 100 80
./spthread 1000 1000 100 120
./spthread 1000 1000 100 200
```

```
...
```

To submit your job, use

```
sbatch xxx.sh
```

Interactive: salloc

```
salloc -n1 -c20 -p Project
srun ./spthread 1000 1000 100 20 # 20 is the number of threads.
```

3.2 dynamic pthread

- **compile the code (without GUI)**

```
g++ pthread.cpp -o pthread -lpthread -O2 -std=c++11
```

- **compile the code (with GUI)**

```
g++ pthread.cpp -o pthreadg -I/usr/include -L/usr/local/lib -L/usr/lib -lglut
-lGLU -lGL -lm -lpthread -DGUI -O2 -std=c++11
```

- **run the code**

  sbatch script

```
#!/bin/bash
#SBATCH --job-name=your_job_name # Job name
#SBATCH --nodes=1                    # Run all processes on a single node
#SBATCH --ntasks=1                   # number of processes = 1
#SBATCH --cpus-per-task=20 # Number of CPU cores allocated to each process
#SBATCH --partition=Project           # Partition name: Project or Debug

./pthread 1000 1000 100 4
./pthread 1000 1000 100 20
./pthread 1000 1000 100 40
./pthread 1000 1000 100 80
./pthread 1000 1000 100 120
./pthread 1000 1000 100 200
...
```

To submit your job, use

```
sbatch xxx.sh
```

Interactive: salloc

```
salloc  -n1 -c20 -p Project
srun ./pthread 1000 1000 100 20 # 20 is the number of threads.
```

3.3 MPI

- **compile the code (without GUI)**

```
mpic++ mpi.cpp -o mpi -std=c++11
```

- **compile the code (with GUI)**

```
mpic++ mpi.cpp -o mpig -I/usr/include -L/usr/local/lib -L/usr/lib -lglut
-lGLU -lGL -lm -DGUI -std=c++11
```

- **run the code**

  sbatch script

```
#!/bin/bash
#SBATCH --job-name=your_job_name
#SBATCH --nodes=1
#SBATCH --ntasks=20
#SBATCH --cpus-per-task=1
#SBATCH --partition=Project

mpirun -np 4 ./mpi 1000 1000 100
mpirun -np 20 ./mpi 1000 1000 100
mpirun -np 40 ./mpi 1000 1000 100
```

  To submit your job, use

```
sbatch xxx.sh
```

  Interactive: salloc

```
salloc  -n20 -c1 # -c1 can be omitted.
mpirun -np 20 ./mpi 1000 1000 100
```

4 Result & Analysis

Display an image with size of 8*8

- static pthread

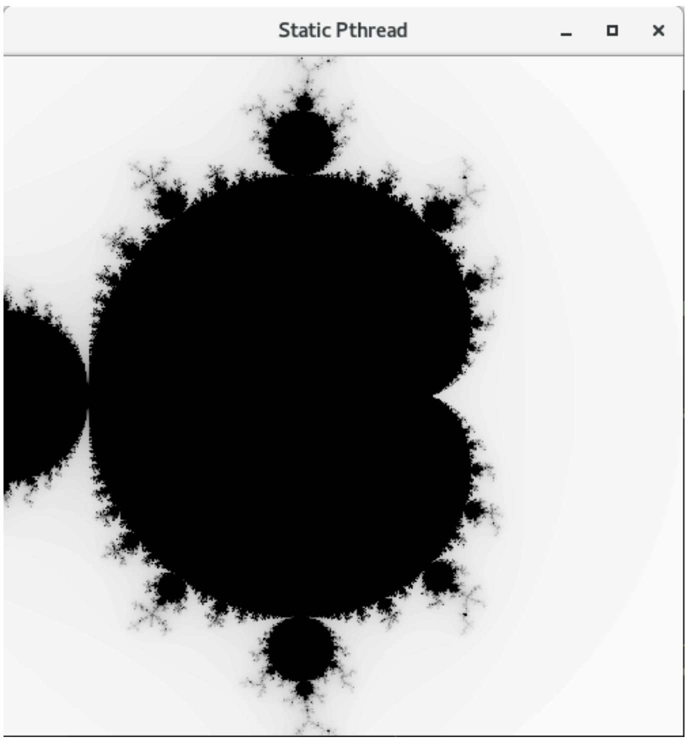


Figure 9: The display of static pthread with size of 800*800

- dynamic pthread

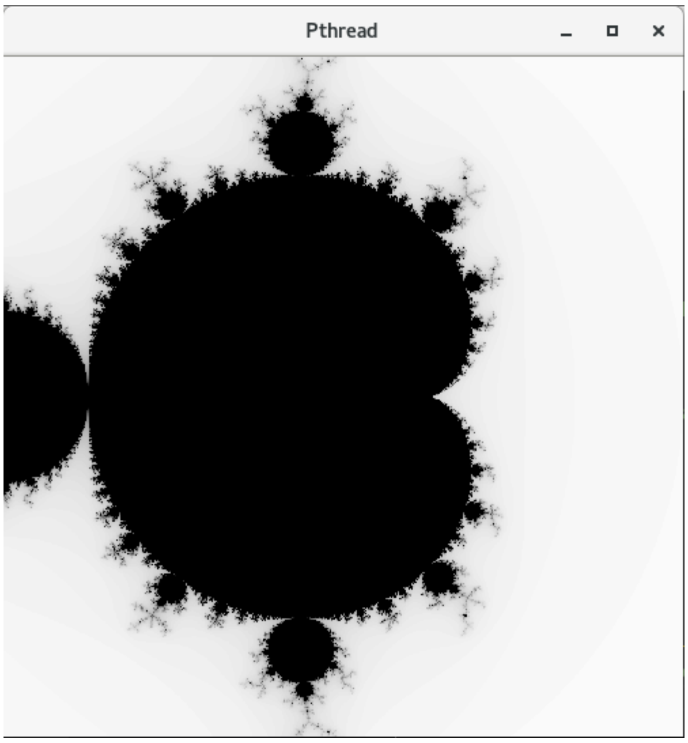


Figure 10: The display of dynamic pthread with size of 800*800

- mpi

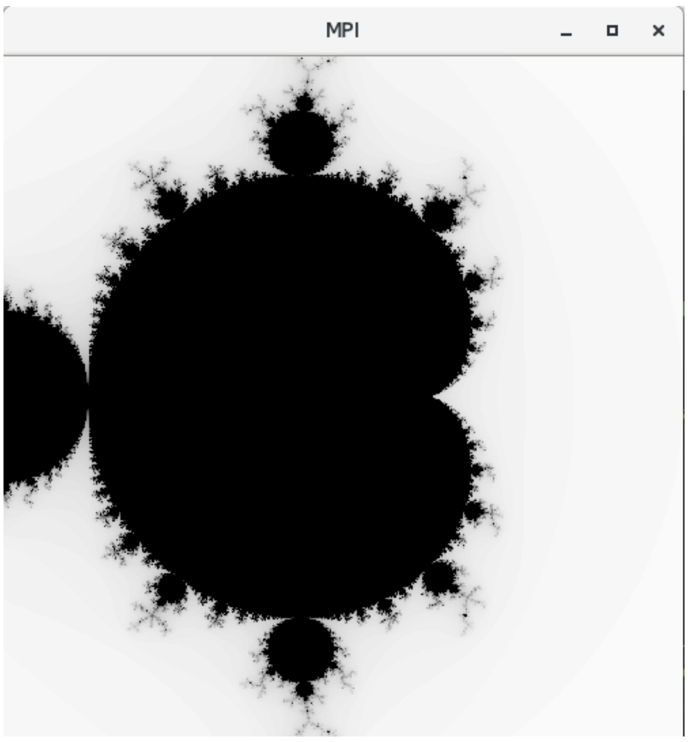


Figure 11: The display of mpi with size of 800*800

4.1 number of threads and performance change

- Based on the experiment design 1, we get the result about the relationship between performance change and number of threads. The following is the result.

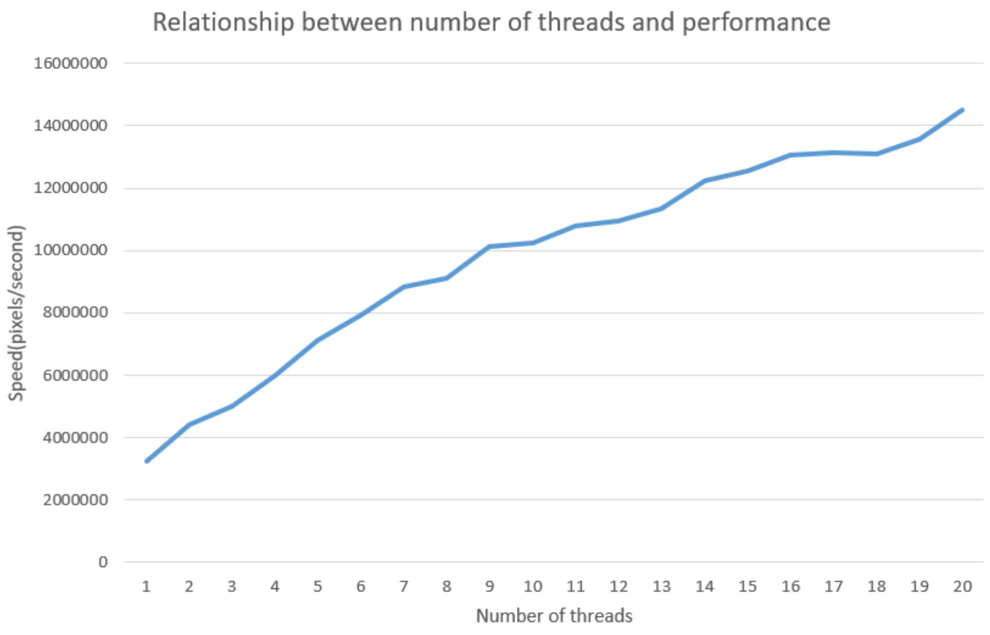


Figure 12: The relationship between number of threads and computing performance

- we can find that the performance(using speed(pixels/s)) of parallel computing is generally increasing in nearly linear way as the number of threads increases. So we can conclude that in general, applying more threads to Mandelbrot set computing will improve parallel computing performance, and the increasing trend is stable.
- The general increasing tendency can be explained by the Aldam's law, which is:

$$S(n) = \frac{n}{1 + (n-1)f} \tag{5}$$

  Note that the Mandelbrot set computation is ideally an embarrassingly parallel computation, which means that there are no data dependencies in this problem (all pixels do not affect each other). Therefore, the serial part (f in Aldan's law) is very low (ideally 0) in the calculation. That's why the increasing trend can be maintained as the number of threads increases.

4.2 1b. number of processes and performance change

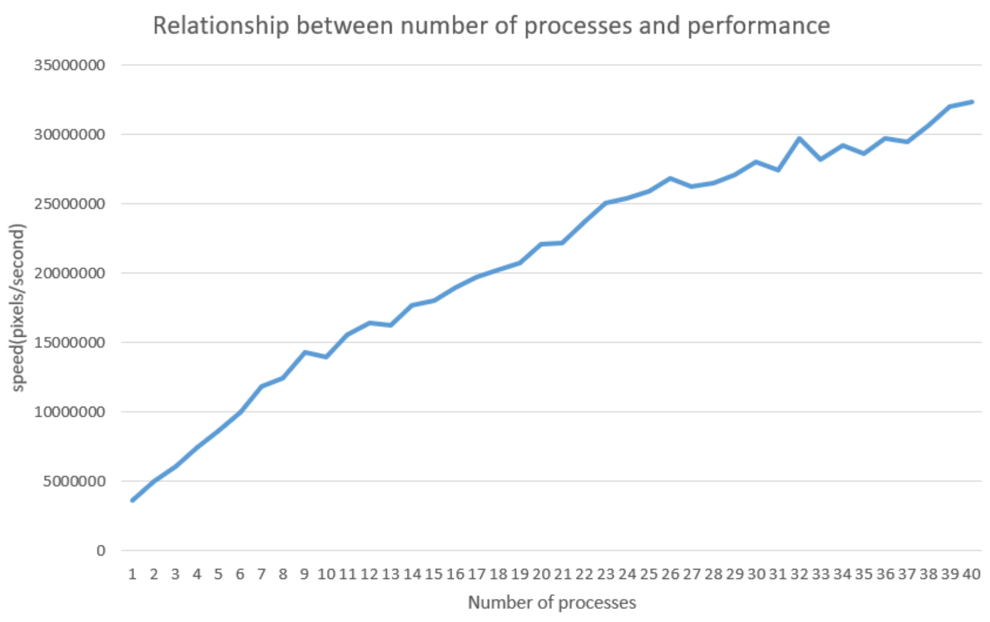


Figure 13: The relationship between number of processes and computing performance

- This experiment uses MPI. The graph shows that as the number of cores increases, the performance will remain close to linear growth. Growth slows slightly as the number of processes increases, but the overall trend does not stop. However, we can also see that sometimes, the number of processes increases and the speed decreases, such as 10 compared with 11, 32 compared with 35.
- For 10 and 11 case, here 10 can be divided by 800, but 11 can not, so for 11, it is required to ask the master process to do more work, therefore, the speed of 11 is slower than that of 10.

- For 32 and 35 case, there is cost from the communication between processes. So, in fact, it is a trade-off. Here, the negative effect of the communication between processes is greater than the positive effect of adding one more process to work. Therefore, the speed of 35 is slower than that of 32.

4.3 problem size and performance change

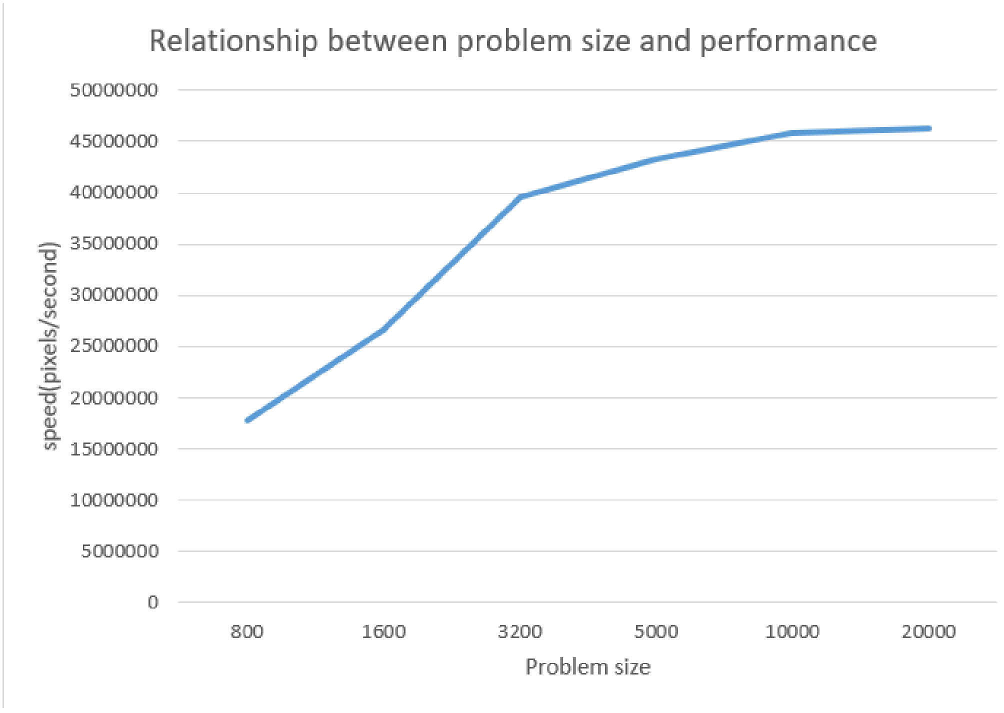


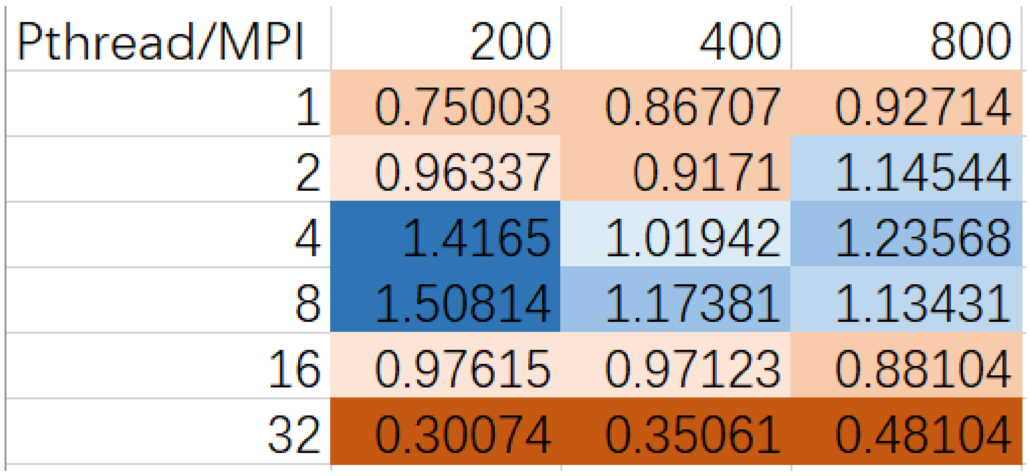
Figure 14: The relationship between problem size and computing performance

- We can see that as the problem size increases, so does the computation speed. But as the problem size gets bigger and bigger, the speedup gets smaller and smaller, approaching a boundary
- increasing performance can be explained by the Gustafson's law, which claims that as the problem size goes up, parallel computing will have a higher performance because of the formulation:

$$speedupfactor = S(n) = n + (1 - n)s \tag{6}$$

- As the problem size increases, there will be fewer and fewer s in the above function. So the speedup is closer to the number of cores.

4.4 MPI and Pthread

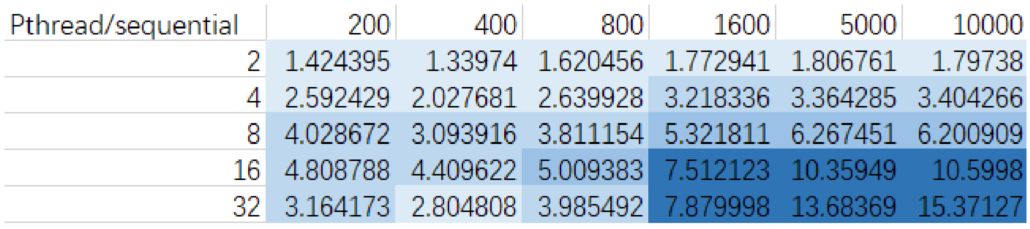


Pthread/MPI	200	400	800
1	0.75003	0.86707	0.92714
2	0.96337	0.9171	1.14544
4	1.4165	1.01942	1.23568
8	1.50814	1.17381	1.13431
16	0.97615	0.97123	0.88104
32	0.30074	0.35061	0.48104

Figure 15: Overall comparation between Pthread and MPI

- The data in the above table shows (speed of pthread / speed of mpi). Blue blocks mean Pthread is better than MPI, brown blocks mean MPI means MPI is better than Pthread. At the same time, the darker the color, the greater the advantage gained.
- When the number of threads or cores is small, than mpi performs better. This is because the scale of the problem is not very big, so in fact, when the number of threads and cores are small, then their speed do not have remarkable gap, but when the number of threads are in large scale, their access to memory may cross the link, and thus the access time would be much greater.

4.5 Pthread and sequential



Pthread/sequential	200	400	800	1600	5000	10000
2	1.424395	1.33974	1.620456	1.772941	1.806761	1.79738
4	2.592429	2.027681	2.639928	3.218336	3.364285	3.404266
8	4.028672	3.093916	3.811154	5.321811	6.267451	6.200909
16	4.808788	4.409622	5.009383	7.512123	10.35949	10.5998
32	3.164173	2.804808	3.985492	7.879998	13.68369	15.37127

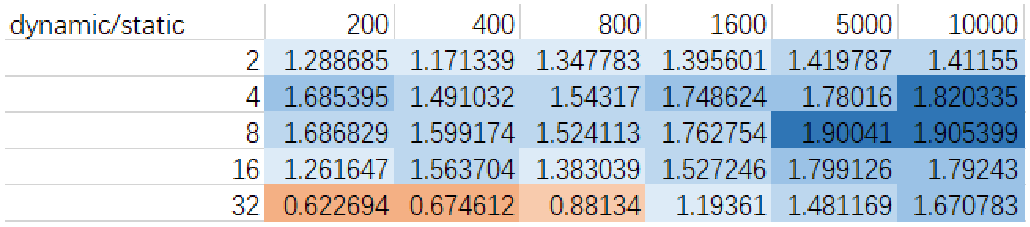
Figure 16: Comparation between Pthread and Sequential computing

- Blue blocks mean Pthread is better than sequential. At the same time, the darker the color, the greater the advantage gained. We can see that pthread performs better then sequential implementation. The advantage is more obvious when the problem size is large and the number of threads is large. And when the problem size is small, too large threads may reduce the growth rate.
- When the problem size is large, it means that the parallel part of the whole problem plays an increasing role in the whole calculation process. Therefore, distributing large-

scale work to each thread will spread the workload and reduce the total computation time.

- However, when the problem size is small, the sequential part and the time each thread accesses the shared memory plays a more important role. The improvements brought by parallel computing are offset by less latency caused by sequential parts and shared memory accesses. Therefore, the growth rate it brings will be smaller.

4.6 Dynamic scheduling and static scheduling



dynamic/static	200	400	800	1600	5000	10000
2	1.288685	1.171339	1.347783	1.395601	1.419787	1.41155
4	1.685395	1.491032	1.54317	1.748624	1.78016	1.820335
8	1.686829	1.599174	1.524113	1.762754	1.90041	1.905399
16	1.261647	1.563704	1.383039	1.527246	1.799126	1.79243
32	0.622694	0.674612	0.88134	1.19361	1.481169	1.670783

Figure 17: Comparation between dynamic and static scheduling

- Blue blocks mean dynamic is better than static one, brown blocks means sequential case is better than dynamic one. At the same time, the darker the color, the greater the advantage gained. We can see that dynamic scheduling performs better then static scheduling in most cases. But when the problem size is very small and the number of threads is large, the speed of static case is larger.

- When the problem size is small, the pixel difference to be processed by each thread is not so large, so some additional operations in dynamic case processing dynamic scheduling may have a greater impact than the pixel difference between threads, so the final completion time may be slower.

5 Conclusion

Here I will summarize what I have learned when writing assignment 2.

- Learn more about parallel computing. Understand the difference between threads and processes. Different processes share different memory, and different threads can share the same global variables.

- Issues to be aware of when writing parallel programs (process synchronization, process communication, thread creation and joining, cost).

- MPI & Pthread API usage.

- I learned how to design experiments to compare different implementation in several dimensions.

- I can manipulate processes and threads more flexibly. For example, using threads to implement dynamic scheduling, and deal with the remainder case for multiple processes.

Bibliography

Wikipedia (2022). *Mandelbrot set.* URL: https://en.wikipedia.org/wiki/Mandelbrot_set (visited on 21st Oct. 2022).