Mandelbrot set

October 30, 2021

Contents

1 Introduction

The Mandelbrot set is the set of points c in the complex plane that are quasi-stable (will increase and decrease, but not exceed some limit) when computed by iterating the function

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

$\mathbf{z_{k+1}}$: the (k+1)th iteration of the complex number $z = a + bi$

c: a complex number giving the position of the point in the complex plane.

Iteration boundary: magnitude of z is larger than 2 or calculating iterations exceed the arbitrary limit.

The Mandelbrot set computation can be executed in parallel, since computation for every single pixel is mutually independent. In this project, I used both MPI and Pthread methods to implement the parallel algorithm. Generally, the parallel computing performs much better than the serial computing, and generally, the multi-threads parallel computing performs better than the multi-process parallel computing.

2 Design Method

2.1 Serial computation

Each pixel is executed sequentially. If one pixel reaches the arbitrary limit (*k_value*), this pixel will be colored.

2.2 Parallel computation

- Job partition/allocation

  The job-partition is one of the most significant problem in the parallel computation. Generally, there are three different partition methods mentioned in previous study. 1) Row based partition; 2) master-slave scheduling partition; and 3) alternating row-based partition [1].

  1. Row based partition: suppose there are total 4 processes/threads, the job is equally divided and allocated to each process/thread.

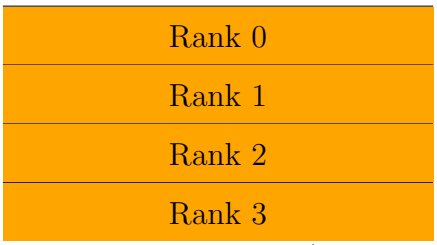
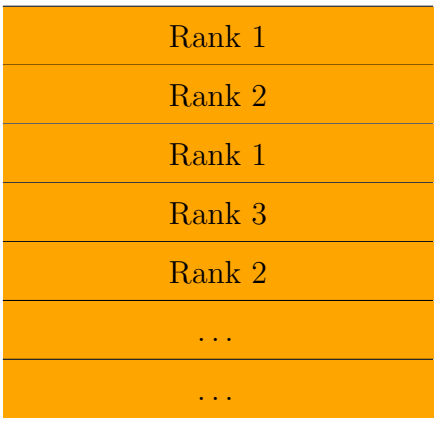


Figure 1.1. Row based partition (multi-process view)

  2. First-come-first-serve partition: once the process/thread is idle, it will take the next computation task until the whole job is completed.



[1] https://arxiv.org/abs/2007.00745

Figure 1.2. First-come-first-serve partition (multi-process view)

3. Alternating row-based partition: the process/thread will be initially allocated with the tasks alternatively.

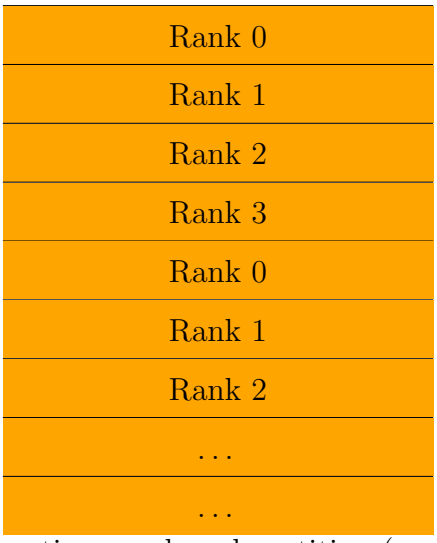


Figure 1.3. Alternating row-based partition (multi-process view)

- Consideration of the three partition methods

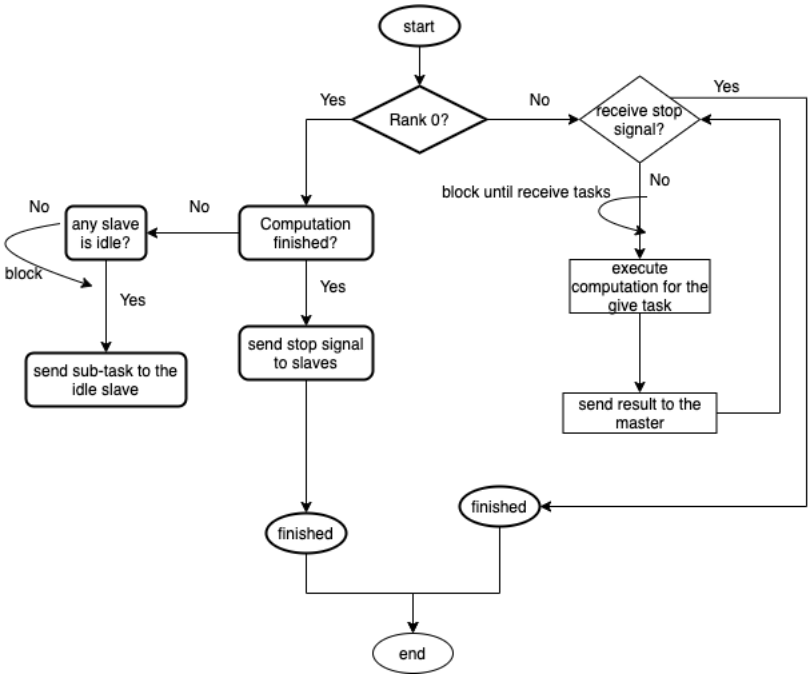
  The first strategy is easy to be implemented, however, the infinite loop will be executed by the Rank 1 and Rank 2 (Figure 1.1). That will lead to imbalanced workload. Therefore, the third strategy is an alternative solution, but the imbalanced workload also exists. For the second strategy, it can maximally utilize the resources, but the communication time between processes is costly.

  In this project, the partition method used is — "First-come-first-serve partition", to dynamically allocate the job and maximally utilize the resources.

- Multi-process implementation using MPI

– Job partition (Figure 1.2): there is one master process, and the others are "slaves". Once the slave is idle, it will tell the master so that the master can allocate job to it.

– Workflow



– Implementation

Message-passing among distributed memory

master: Boardcast all basic information to slaves, e.g. size, scale, center_x, center_y, k_value (*MPI_Bcast*).

Initially allocate tasks (start index and workload) to slaves(*MPI_Send*).

Block receive (*MPI_Recv*) the result from slaves, which also indicates that slave is idle (*status.MPI_SOURCE*).

Update the canvas.

Send stop signal (start index = -1) to slaves (*MPI_Send*) if there

is no remaining task.

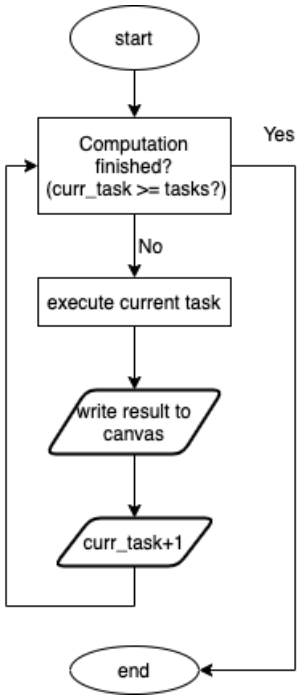
slaves: Receive the start index and workload from master.

Stop when receive stop signal (start index = -1).

Execute computation. (*mpi_calculate*)

Send the result back to the master.

- Multi-thread implementation using Pthread
  - Job partition: the threads keep executing computation until the entire job is finished.
  - Workflow



  - Implementation

1. **Shared resources**: tasks=m (the entire job is divided into m parts), curr_task=0 (current task start from 0)
2. **Mutex Lock Strategy**: (DATA RACE) curr_task is written & read by each thread. When multiple threads try to read the curr_task at the same time, then, those threads will do the same task based on the value of the same curr_task, which is not allowed. Besides, update the value of curr_task should also be atonic. Therefore, use a mutex to lock a critical section.

   ```
   pthread_mutex_lock(&thread_task_mutex);
   ....const int current_task = curr_task;
   ....curr_task++;
   pthread_mutex_unlock(&thread_task_mutex);
   ```

3. **Threads creation**: since the one node is with 32 cores, the total number of threads is set to 32.

3 Execution

- GUI version (under ./csc4005_imgui)

  The executable target files are `csc4005_imgui, thread, sequential`

  ```
  $ mkdir build && cd build
  $ cmake ..
  $ make
  $ mpirun -n $process_number ./csc4005_imgui  # MPI version
  $ ./thread  # pthread verision
  $ ./sequential  # sequential version
  ```

- Benchmark version (under ./csc4005_as2_bench) The executable target files are `bench_mpi bench_thread`, who should take input as arguments to execute:

```
$ mpirun -n $process_number bench_mpi $tasks $k_value $problem_size
$ ./bench_thread $tasks $k_value $problem_size $thread_number
```

You can also run the follow shell scripts:

```
$ ./build.sh
$ ./sbatch_mpi.sh  # MPI
$ ./sbatch_thread.sh  #Pthread
```

4 Result

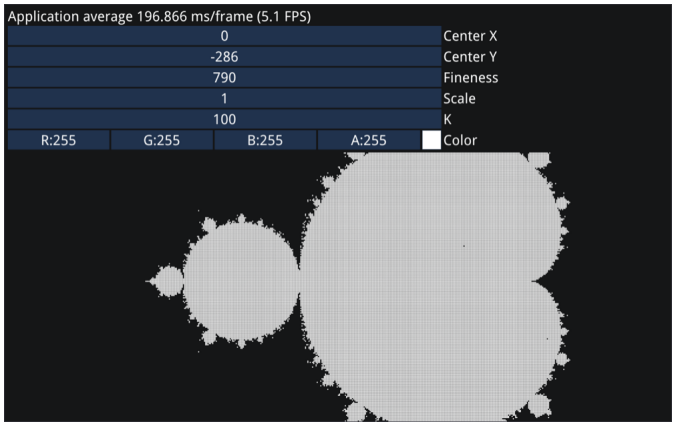


Figure 1.1 800x800 Mandelbrot set graphic

4.1 Variable declaration

problem_size: canvas width or height.

k_value: the maximum iteration per pixel.
process_number: used for mpi, defined as the number of processes/cores.
thread_number: used for pthread, defined as the number of threads.
tasks: the number of partitions that the canvas is divided into.
speed_up: the ratio of the compute time for the sequential algorithm to the time for the parallel algorithm.
Esp (speed up efficiency): $\frac{speed\_up}{\#cores} \times 100\%$

In my project, I choose the k_value to be 800, since the more computing iterations gives a more accurate result.

4.2 MPI

1. Variable: problem_size (400˜ 1000) & process_number (2˜ 128) (Fixed: tasks=400, k_value = 800)

   (Figure 2.1)The speed keeps increasing as the problem_size increasing when the process_number is large (> 60). However, when the process_number < 30, the speed tends to decrease as the problem_size enlarging. Since the problem size is increasing, a subtask's workload is going to be heavier. If only use 20 cores, each core takes longer time to complete one subtask. If we use 80 cores, although the computation time for the subtask also increased, the cores are largely be utilized as the problem size enlarging, thus, increasing the speed.

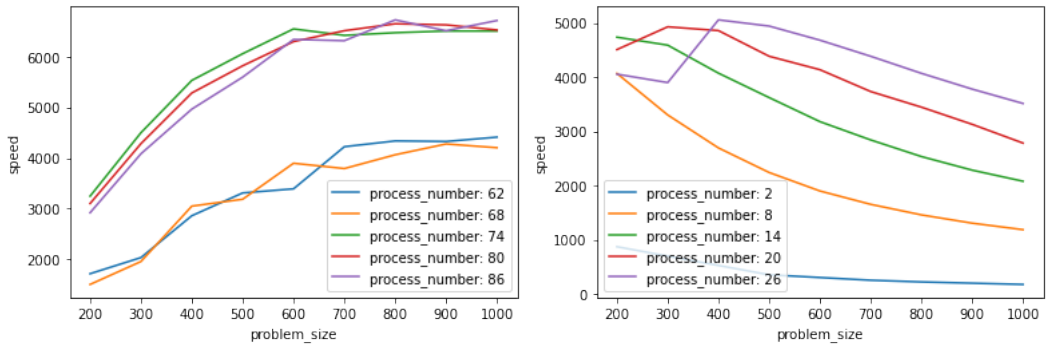


Figure 2.1

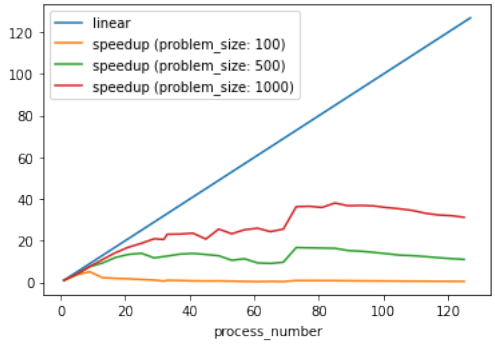


Figure 2.2

From the Figure 2.2, we can conclude that under the same problem size, as the process number enlarging, the speedup is not always increasing. The speedup firstly increases, since more cores enable more parallel computation. However, as the process number keeps increasing, the speedup begins to fall. One possible reason for that is the bottleneck of the master process. If more slaves execute the computation, the master needs to block send and block receive for each slave. In a real situation, each processor does not start and finish in the same time, and the communication overhead and synchronization can harm the

parallel execution when the number of processors increases [2].

2. Variable: tasks (2 ~ 400) & process_number (Fixed: k_value=800, problem_size=800)

   When the process_number is < 70, we can observe a bottleneck that the speed isn't increasing as the tasks enlarging (Figure 2.4). However, if the process_number raises, the bottleneck can be broke (Figure 2.5 / Figure 2.6). Thus, it gives some ideas that, 1) given a settled number of cores, we can divide the job into suitable tasks to optimize the performance; 2) we can speed up the program by increasing the number of both task and cores.

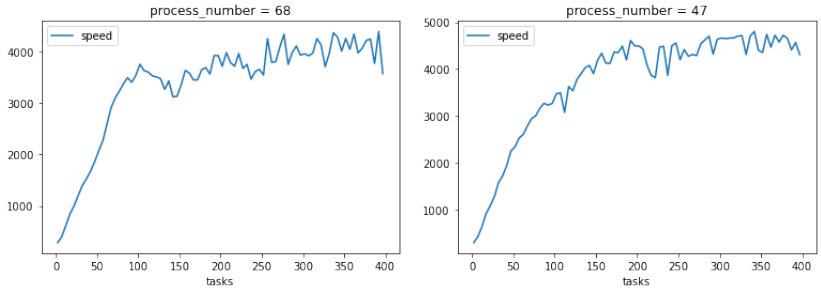
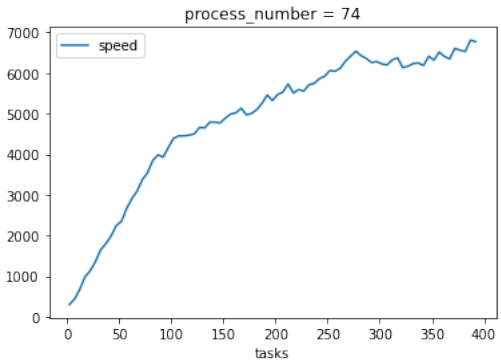


Figure 2.4



[2]https://annals-csis.org/Volume_8/pliks/498.pdf

Figure 2.5

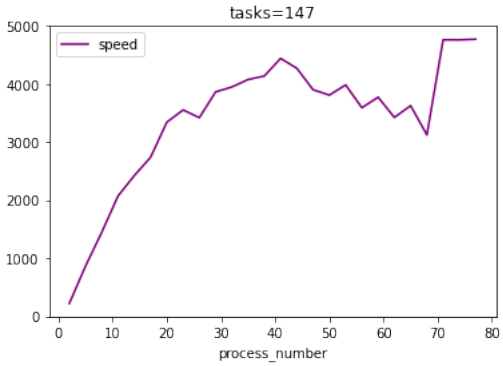


Figure 2.6

4.3 Pthread

1. Variable: tasks (100˜ 1000) (Fixed: problem_size=800, k_value=800)

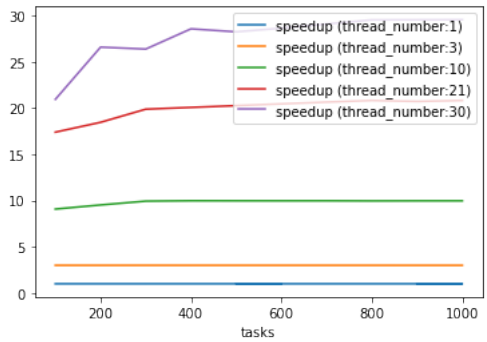


Figure 3.1

Generally, the Figure 3.1 shows the *speedup* is indifference to the number of tasks, which may because the multi-thread parallelism eliminates the communication time, the execution time only cares about the *problem size* under the same number of threads.

2. Variable problem_size (200˜ 1000) (Fixed: k_value=800, tasks=400)

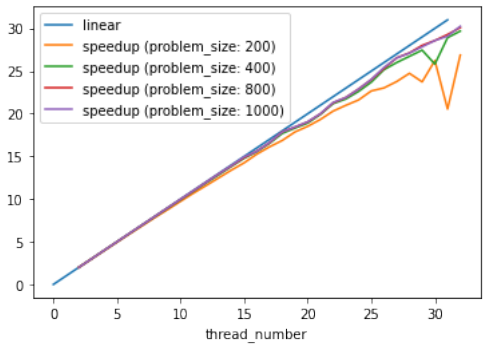


Figure 3.2

From Figure 3.2, we can observe that the multi-thread programs' speedup is close to the theoretical speedup. In other words, $Esp \approx 1$. According to Amdahl's law

$$speedup_n = \frac{T_1}{T_n} = \frac{1}{\frac{F_{parallel}}{n} + F_{sequential}}$$

which gives the max possible speedup:

$$speedup_{max} = \frac{1}{F_{sequential}}$$

Therefore, we can explain the result in Figure 3.2 by Amdahl's law: the sequential fraction in the program cannot be executed in parallel, and the $F_{sequential}$ is relatively small. In my program, the main process just create can join the threads, other execution is carried by the threads, thus, the $F_{parallel}$ is close to 1. Moreover, when the problem size becomes relatively large, more threads give a better speedup, since the more threads can execute the program at the same time.

4.4 Comparing MPI and Pthread

- MPI method: process_number = 32, k_value=800, tasks=400

- Pthread method: threads = 32, k_value=800, tasks=400

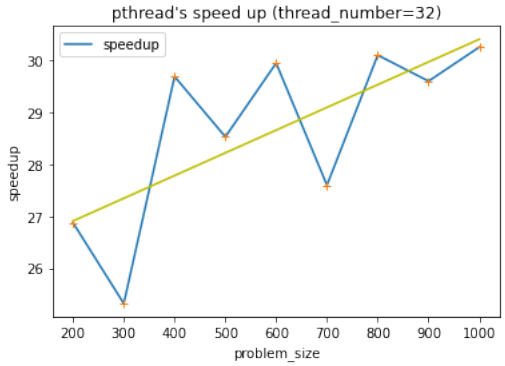


Figure 4.1

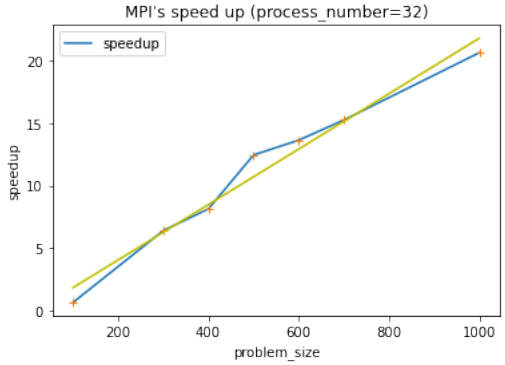


Figure 4.2

Generally, the multi-thread computation gives a much larger speedup compared with the multi-process computation. The multi-thread method largely allocate tasks to all threads, while the multi-process method needs the master process to update the result. Moreover, the multi-thread method uses shared memory so that each thread can access the shared resources, while the multi-process method need communication time between the master and salves. Moreover, we can observe that both the multi-process method and the multi-process method can in-

crease the execution time as the problem size increasing.

5 Conclusion

In this project, I have learned how to implement parallel computation by multi-process (MPI) and multi-thread (pthread), which makes me more clear about the differences of data-management between shared-memory and distributed-memory. Moreover, I compared and analyzed the performance between the two parallel computations with the sequential computations, thus, I know more about the computing estimation. However, there is limitation in my project. The max threads number is 32, which means the theoretical threads number is limited as 32. Thus, I just compare the multi-process and multi-thread under the same scale of parallelism. If we need a speedup above 32, we may need to implement the multi-process. In conclusion, it is hard to say which method is better over another. It depends on the purpose.