CS542200 Parallel Programming

Homework 2: Mandelbrot Set

109065517 林佳縈

1. Implementation

In this section, I'd like to briefly introduce how I implement the Mandelbrot set algorithm by elaborating the algorithm's working flow, accompanied with some screenshots which might help for clearer understanding.

- **For pthread version**

The implementation is based on the sequential code. Places they're different are:

(1) pthread_create to create threads

(2) mandelbrot_set for repeating times calculation.

(3) pthread_join to wait every thread to finish its job.

The main differences I'd like to emphasize are:

a. How I partition the task is as followed:

Since the loop for calculating repeating times of each point is written as in sequential code:

```
for(int j=0; j<height; j++){
    double y0 = j * ((upper - lower) / height) + lower;
    for(int i=0;i<width;i++)
```

To parallelize it, I partitioned the task by height with the amount of CPUs I have. Simply dividing height by ncpus(number of CPUs), I then can get the amount of work each CPU has to take responsibility of.

However, for those height mod ncpus not equals to zero, it is also a scenario that has to be considered. To also take care of this case, I did the following:

as could be understood, if there is no remain for height%ncpus, the part each thread has to work can be denoted as: tid(thread_id)*partition

to take care of the remain, first I calculate of which tid and partition_mod is smaller, then I add the smaller one to the tid*partition, and take it as the for loop start. Thus, about the end of the for loop, same case, if there is no remain, it can simply be assigned for end = start + partition, but take the remain for consideration, again, if the partition_mod is bigger than tid, I'll add 1 to end. In brief, it can be written as following:

```
int min = tid;
if(partition_mod<tid) min = partition_mod;
int start = tid*partition + min;
int end = start + partition - 1;
if(partition_mod>tid) end += 1;
```

b. As for what technique I use to reduce execution time and increase scalability, since load balancing in Mandelbrot set is a critical problem, instead of assigning tasks to each thread like this:

```
for (int j=start; j<=end; j++)
```

I assigned the task for each thread like this:

```
for (int j=tid; j<height; j+=ncpus)
```

It reduced nearly half of the execution time.

- **For hybrid version:**

Still, I tried to parallelize the sequential code with both OpenMP and MPI to complete the hybrid version. Basically the implementation is all in the calculation of repeating times. The following would briefly introduce what I've implemented, which is different from the sequential version:

(1) MPI initialization:

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
MPI Comm rank(MPI COMM WORLD, &rank num);
```

(2) Get the number of CPUs:

```
cpu_set_t cpu_set;
sched_getaffinity(0, sizeof(cpu_set), &cpu_set);
num_thread = CPU_COUNT(&cpu_set);
```

(3) Modify the for loop to a nested for loop

This part is probably the biggest change to the sequential code compared to other modification. Since I'd like to use an OpenMP instruction which is **`collapse(2)`,** which means that it can fold the original two-level nested for loop to only one big for loop but under the premise that it is a nested for-loop where there exist no other instruction between the two "for".

(4) OpenMP to parallelize the for loop

So after the nested for loop is modified, I had OpenMP help me with the parallelism.

```
#pragma omp parallel for schedule(static) private(c) collapse(2)
```

2. Experiment & Analysis

- **Methodology**
  - **System Spec**

    During the entire work, I implemented and tested on Apollo.
  - **Performance Metrics**

    As for how I measure the computing time so on and so forth, I chose the **srun** to calculate the time. The instruction is as follow:

    For pthread version: `srun –nx –cy time ./hw2a`

    For hybrid version: `srun –Nm –nx -cy time ./hw2b`

    with m, x, y denoted as number of Node, process, and thread respectively.

- **Plots: Scalability & Load Balancing**

To conduct an experiment on the scalability and load balancing test, I chose

the testcase fast01.txt as standard and launch my experiment.

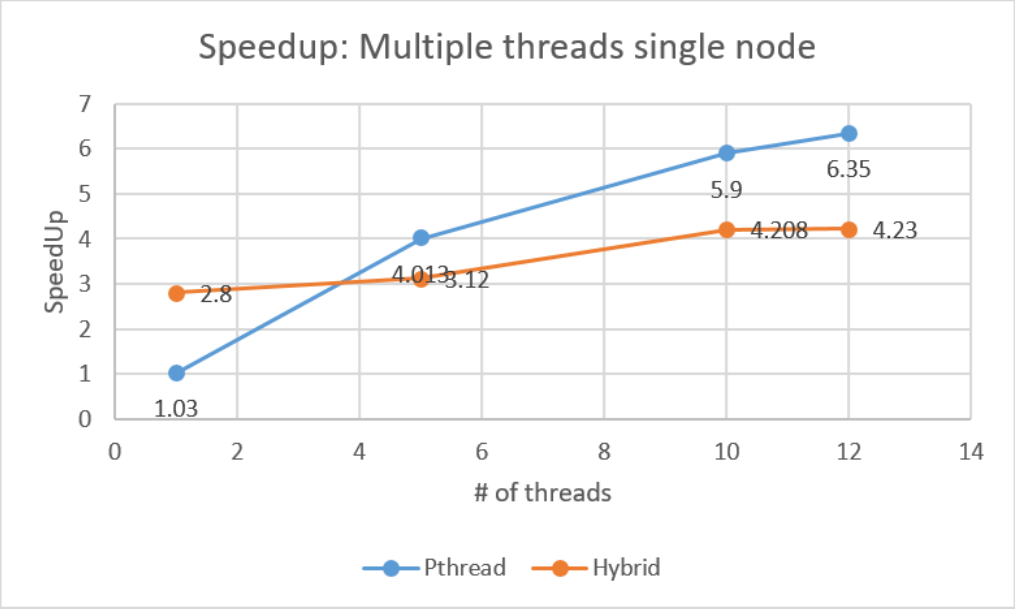
- **Scalability**

  In scalability section, I'd like to demonstrate it with the following experiments.

  - **SpeedUp: Multiple threads single node:**

    Arguments: -N1 –n1 –c1/-c5/-c10/-c12

    Testcase: fast01.txt



In this experiment, I'd like to see how number of threads impacts on the result.

As for the chart, x-axis is denoted as number of threads while y-axis stands for the degree of SpeedUp. Note that the way I calculate the degree of SpeedUp is:

SpeedUp = (Ep or Eh) / Es

- *Ep: Execution time of pthread version*
- *Eh: Execution time of hybrid version*
- *Es: Execution time of sequential version*

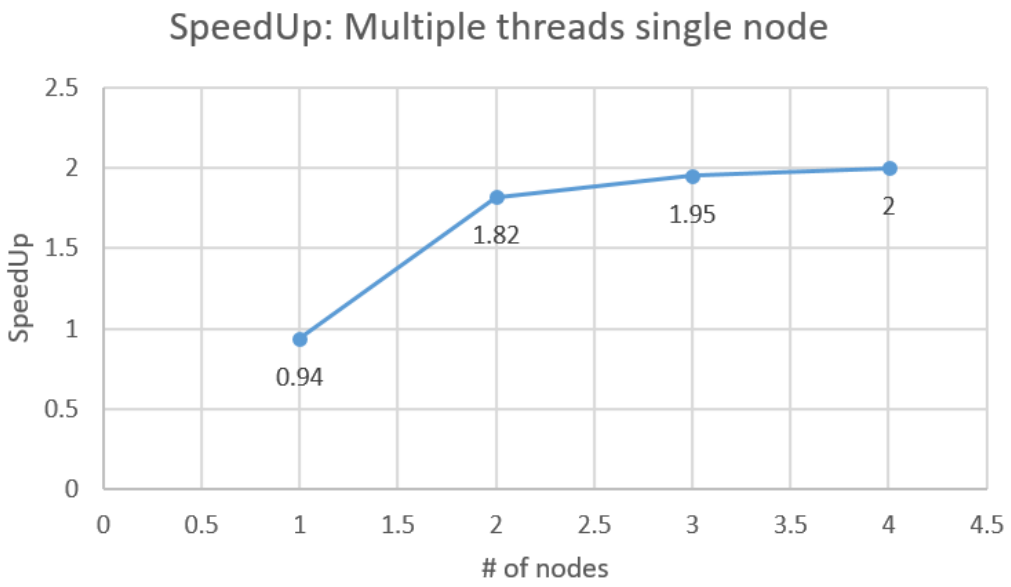
While blue line indicates the result of Pthread version, Hybrid is denoted as the orange one. Note that the testing environment, the amount of node is set to 1.

We can tell from the graph that as the threads number rise, the Pthread version has great progress on speed, and hybrid follows also similar trend.

- **SpeedUp: Multiple nodes single thread**

  Arguments: -N1~3 –n1~3 –c1

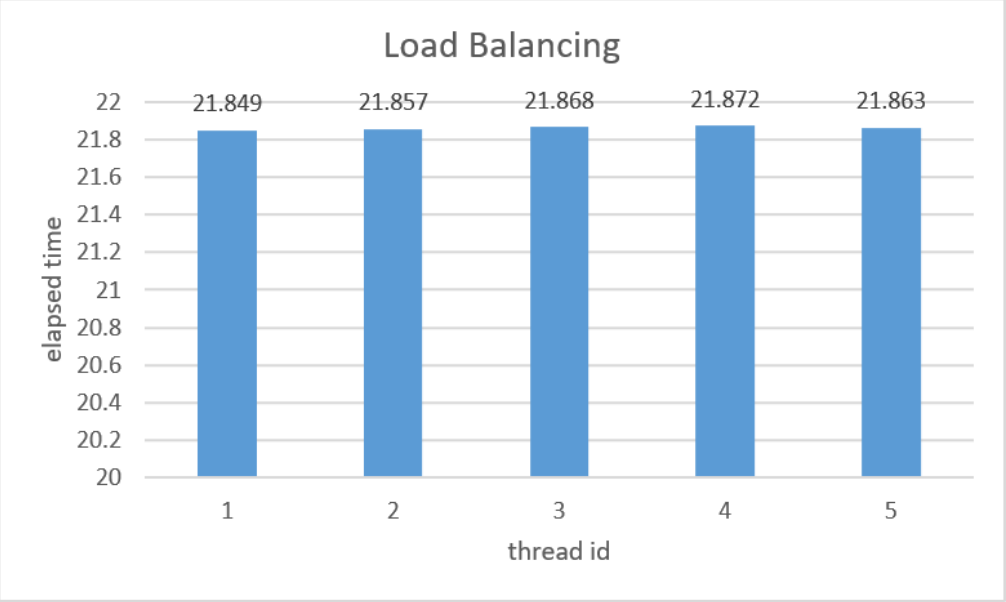
  Testcase: slow5.txt



Different from the above mentioned experiment, during this experiment, I'd like to see the influence on different node. Note that since for Pthread version, the node is always 1, so for this case, I conducted the experiment with only the Hybrid version.
Following by the number shown on the graph, we can see that not the more nodes the better the performance is.

- **Load Balancing**
  - **SpeedUp: Load Balancing**

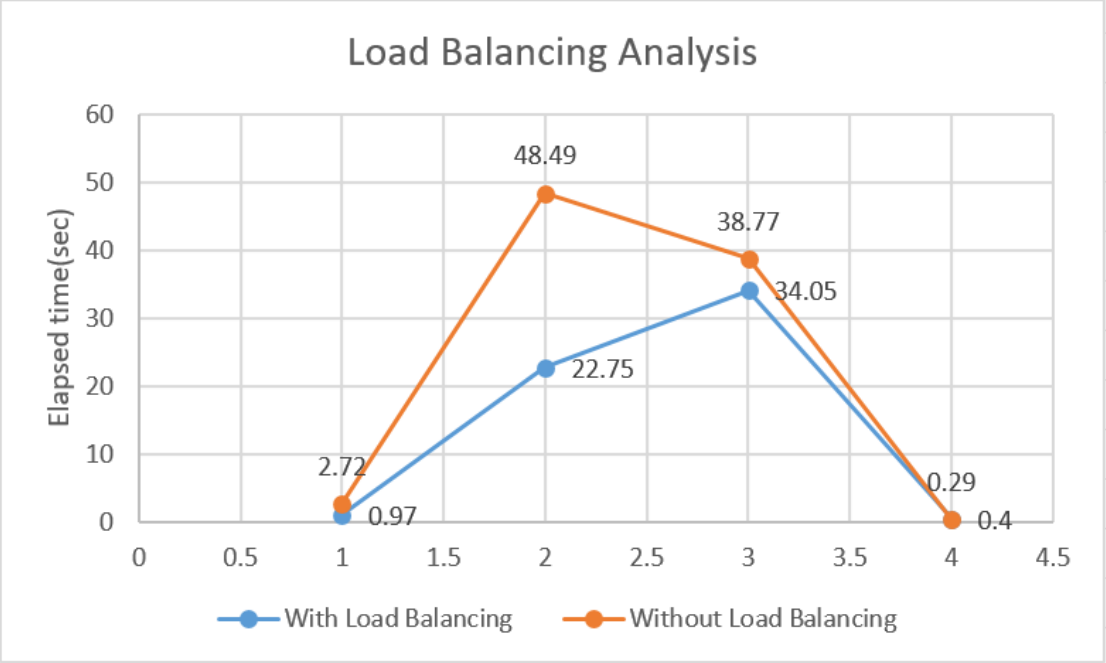
To analyze load balancing, I chose the MPI function: MPI_Wtime() to calculate the time each thread use to work on its own job.
This is the result I tested on Pthread:
Arguments: -N1 –n1 –c5
Testcase: slow05.txt



According to the chart, we can tell that for each thread, it spent nearly the same time for their own job. The x-axis denoted as thread id and y-axis is the amount of time each thread spent respectively.

To show load balancing's effort on reducing the execution time, I also compare the version of paralleled code but haven't done load balancing with the one has been done load balancing.
So let's see the difference in between:
Arguments: -n1 –c5
Testcases: fast14.txt(denoted as 1 in x-axis in the graph), slow05.txt (denoted as 2), slow15.txt (denoted as 3), strict09.txt (denoted as

4).



I chose the following 4 different testcase to analyze load balancing, they are fast14.txt(denoted as 1 in x-axis in the graph), slow05.txt (denoted as 2), slow15.txt (denoted as 3), strict09.txt (denoted as 4). According to the graph, we can clearly tell that with load balancing, the execution time is always less than those without.

3. Experience & Conclusion

When I first implemented the Mandelbrot set Pthread version, I haven't actually noticed how important load balancing is. So I simply modified the code as how I modified mine in Lab3; and undoubtedly, it worked fine but really slowly. Thus, I found that it might be the problem of load unbalanced, thus, just as mentioned above, I slightly adjusted my for-loop, magically, it speeded up with execution time only 1/2 of original one. Later on, for hybrid version, I've also tried the master-slave version, but unfortunately, I didn't actually carry it out. But still, knowing how to do some load balancing tricks did help!