NTHU CS542200 Parallel Programming
Homework 1: Mandelbrot Set

111062625 蔡哲平

1. Implementation

i. How do you implement each of the requested versions, especially for the hybrid parallelism?

- Pthread Version

  I use the core number as the number of threads. There is a shared variable called *curr_row* which records the next row to be computed and needs to use a mutex lock to synchronize its access. Each thread will run a function that has a loop inside to keep getting the next row once it finishes calculating the previous one. The loop will terminate when *curr_row* is bigger than the input image's height which means all rows have been calculated and the main thread will write the result into the output image.

- Hybrid Version

  First, determine each row should be calculated by which process with the static scheduling technique. Below is a graph demonstrating my method. Different color represents different process.

row4
row3
row2
row1
row0

  Assigning rows to processes in turn can achieve better load balancing than distributing consecutive rows to a process since the nearby pixels usually have the same level of computation overhead. As for the thread's workload, I apply the dynamic scheduling method using OpenMP's *schedule(dynamic, 1)*. Each thread in a process will only calculate a pixel at a time and it'll automatically get another pixel once it finishes. Finally, use *MPI_Gather* to collect each process's computation result and write them into the image. Note that the row writing order in the function *write_png* should also be adjusted since it won't be ordered when we collect them.

ii. How do you partition the task?

- Pthread Version

I divide the input image into rows and consider the row as the basic unit of computation. Each thread needs to complete a row calculation to get the next row.

- Hybrid Version

  I divide the input image into rows and distribute them to processes in a static scheduling way. On the other hand, I divide a row into pixels and use a dynamic scheduling way to distribute them to threads.

iii. Other efforts you made in your program.

- Vectorization

  Since the server only supports SSE instruction, which has 128-bit registers, I use it to calculate 2 pixels(double is 64-bit) at a time. This method can significantly reduce the computation time.

- Channel Computing

  Following the above method, since the computation between 2 pixels can also be imbalanced, a channel will idle when it finishes its calculation but the other doesn't. Hence, once a channel ends computing, it will immediately compute the next pixel, which will minimize the idle time in both channels.

2. Experiment & Analysis

i. Methodology

- System spec

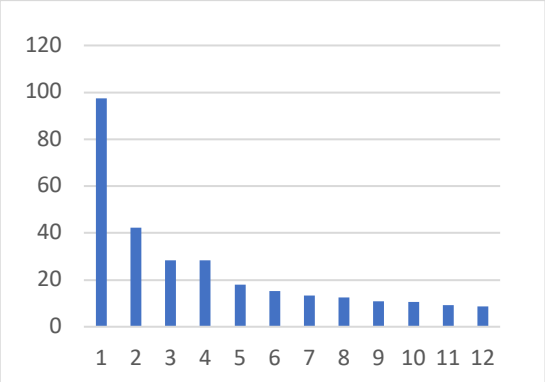
  All tests were run on the clusters provided by this course.

- Performance Metrics

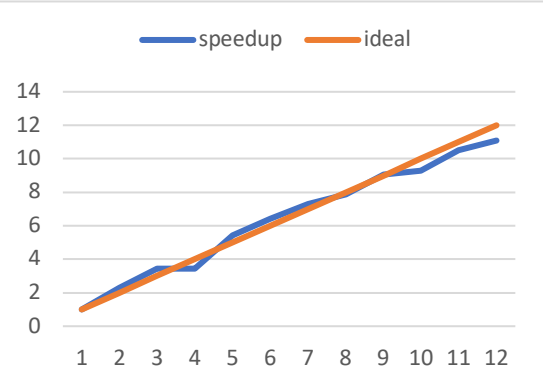
  Use *std::chrono::high_resolution_clock* in the pthread version. Use *MPI_Wtime* and *omp_get_wtime* in the hybrid version.

ii. Plots: Scalability & Load Balancing

- Experiment 1: Under a single-node environment (N=1), observe the results of running different numbers of threads. (Pthread version)



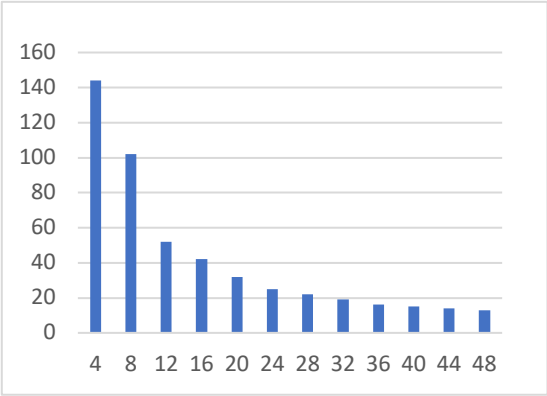
Runtime of different numbers of threads
[X: thread num, Y: runtime(sec)]



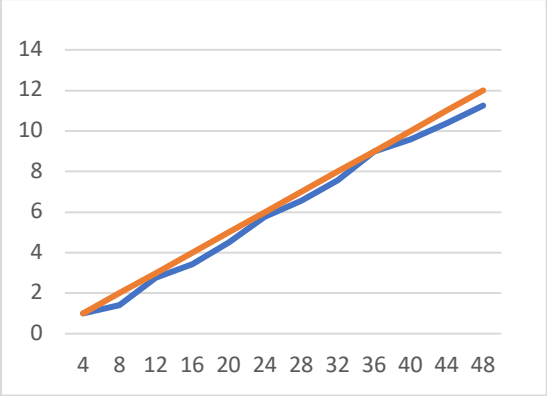
Speedup of different numbers of threads
[X: thread num, Y: speedup]

Observation 1: According to the above figure, we can see that as the number of threads increases, runtime decreases. Furthermore, the speedup is very close to the ideal value, which means my pthread version program scales well. However, there's a slight drop when using many threads, I believe that it's because my program doesn't parallel the writing image part.

- Experiment 2: Under a multiple-node environment (n=4), observe the results of running different numbers of threads. (Hybrid version)



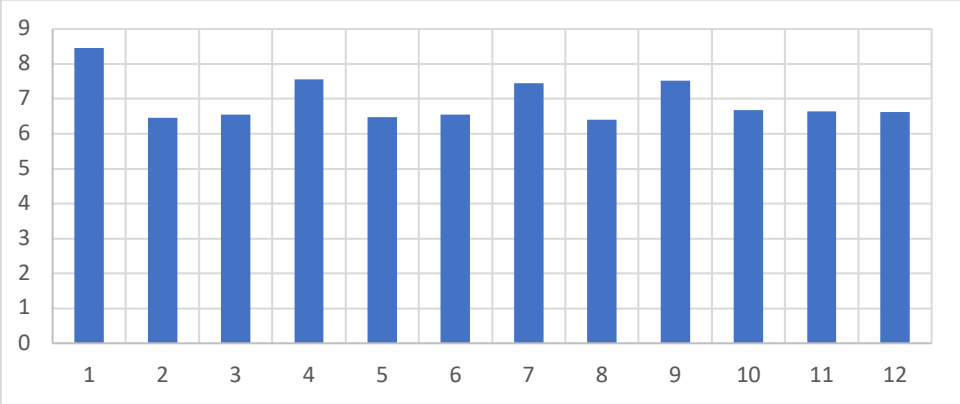
Runtime of different numbers of threads
[X: thread num, Y: runtime(sec)]



Speedup of different numbers of threads
[X: thread num, Y: speedup]

Observation 2: According to the above figure, we can see that as the number of threads increases, runtime decreases. Moreover, the speedup is very close to the ideal value, which means my hybrid version program scales well.

- Experiment 3: Under a fixed thread number environment (c=12), observe the runtime of each thread. (Pthread version)

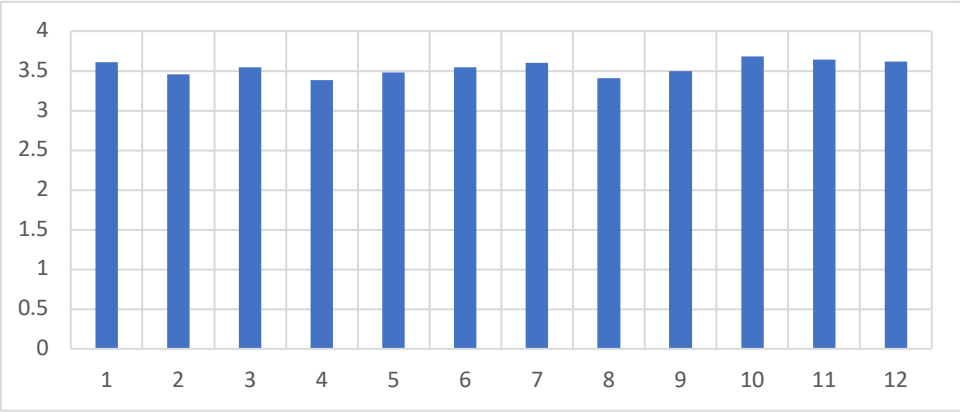


Runtime of each thread
[X: thread id, Y: runtime(sec)]

Observation 3: According to the above figure, we can see that my pthread version program doesn't achieve good load balancing since the runtime can vary by 2 seconds. I believe it's because each row's computation overhead isn't

the same, hence, even if I use the dynamic scheduling technique it still causes imbalanced computation among threads. To solve this, I think changing the computation size to less than a row or using static scheduling may help.

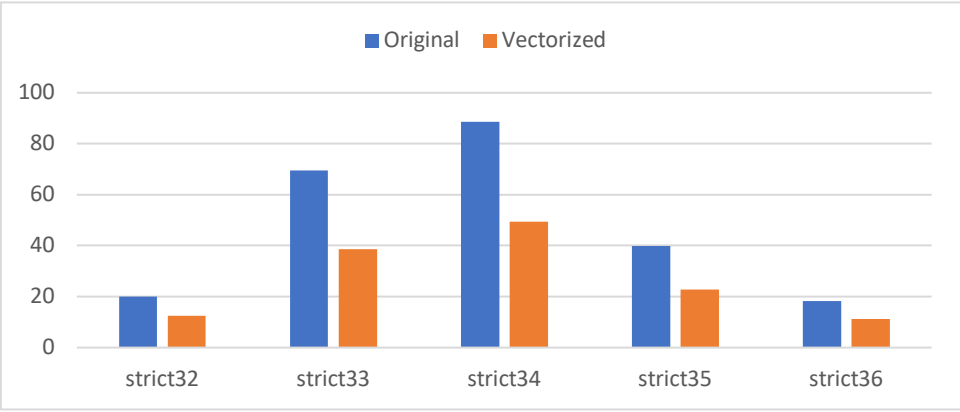
- Experiment 4: Under a fixed thread number and fixed node environment (c=12, n=4), observe the runtime of each thread. (Hybrid version).



Runtime of each thread
[X: thread id, Y: runtime(sec)]

Observation 4: According to the above figure, we can see that my hybrid version program achieves good load balancing since the runtime varies within 0.5 seconds. I believe it's because I distribute rows in turn instead of distributing consecutive rows to processes. This way, processes can have nearly the same computation overhead.

- Experiment 5: Under a fixed thread number environment (c=12), observe the runtime of running the original and the vectorized version on different test cases. (Pthread version)



Runtime of different test cases under original and vectorized version
[X: test case, Y: runtime(sec)]

Observation 5: According to the above figure, we can see that the runtime of vectorized version is nearly half of the runtime of the original version. It's

because it computes 2 pixels simultaneously in the vectorized version, which causes a 2 times speedup of runtime.

iii. Discussion

- Compare and discuss the scalability of your implementations.

  According to the above experiments, we can see that both pthread and hybrid version scale well. It's because the computation of the Mandelbrot set can be parallelized among threads and there's only little communication among threads, which is very different from the last assignment which has heavy communication overhead.

- Compare and discuss the load balance of your implementations.

  According to Expr.3, my pthread version doesn't achieve good load balancing. In my opinion, it's because I use dynamic scheduling instead of static scheduling. Therefore, threads that get easy rows will finish very fast and wait for those with difficult rows. According to Expr.4, my hybrid version achieves good load balancing. I believe it's because I distribute rows in turn to processes, hence, each process will have nearly the same computation overhead.

3. Experiences / Conclusion

In this assignment, it didn't take too long for me to transform the sequential code into parallel code. However, transforming it into vectorized version is so frustrating to me. Coding with SSE is like writing assembly code, you need to change your logic to register level, which is not straightforward. I spent so much time debugging my vectorized version and spent even more time optimizing it, however, it didn't speed up as much as I expected, which is quite frustrating. Moreover, I found that the order of the conditions written in an if bracket also has a lot to do with the execution time, especially those in the while loop. In a nutshell, I spent more time on this assignment compared to the last one, however, my program didn't reach a good position on the leaderboard. Although it's very frustrating, I'm still happy that I learned the technique of using SSE instructions, which is new to me. Hope that I can use it in my further assignments.