Parallel Programming
Homework 2: Mandelbrot Set

王鈺鎔 106062600

Implementation

(a). *How you implement each of requested versions, especially for the hybrid parallelism and dynamic scheduling algorithm.*

Omp

Add the line, `#pragma omp parallel for schedule(dynamic) num_threads(num_threads)`, before the first for loop to dynamically assign one row to a thread each time.

Mpi_static

Add the condition, `if (j%size == rank)`, before the second for loop to assign "one" row to each thread per time according to the module. When all the processes finish all their work, use `MPI_Reduce()` to get the result. Each element (i, j) of the result is the sum of all the element (i, j) of the image of each process.

Mpi_dynamic

Firstly I use the same techniques as mpi_static. Let the rank 0 be the master and other processes be the slaves. Each process calculates "one row" each time. Whenever any process finishes its work, it will inform the master. The master will assign "the next row" (there is a counter in the master) to the process.

When the master assigns all the rows to processes, it will count that it needs to receive all the slave finishing message one time and also inform the slave that the whole work is complete. When the slave receives the competition message, the slave finishes its calculation work.

When all the processes finish all their work, use `MPI_Reduce()` to get the result. Each element (i, j) of the result is the sum of all the element (i, j) of the image of each process.

Hybrid

The hybrid version is the combination of omp version and mpi_static version. That is, use mpi_static as the base, and add the omp techniques to split the outer for loop to the threads of the process.

Compared with mpi_static version, mpi_dynamic version loses the master process' computing capability to assign the work to slaves. Although when there are multi-threads, we can modify mpi_dynamic, the master process still needs to sacrifice one thread for assigning work. Thus, we choose the combination of mpi_static and omp instead of the combination of mpi_dynamic and omp to gain better performance.

(b). *How do you partition the task?*

Omp

Dynamically assign “one row” to a thread each time.

Mpi_static, Mpi_dynamic and Hybrid

Add the condition, `if (j%size == rank)`, before the second for loop to assign “one row” to each thread per time according to the module. Compared to divide the height into batches, `for (int i=rank*batch; i<(rank+1)*batch; i++)`, assign rows to threads according to the module can divide the whole work more balance. The reason is that dividing the height into batches needs to consider the remaining.

(c). *What technique do you use to reduce execution time and increase scalability?*

When using omp to parallel compute Mandelbrot Set, I choose dynamical omp instead of static omp to balance the execution time among thread and avoid the process blocks in some long-time threads, which can reduce execution time and increase scalability better.

In mpi_static, mpi_dynamic and hybrid, I choose `MPI_Reduce()` to get the result instead of `MPI_Gather()` which needs to handle the remaining problem and may need additional send/receive. `MPI_Reduce()` can gather the result in one send/receive per process, which decreases execution time.

In mpi_static and hybrid, I assign “one row” to each thread per time according to the module instead of dividing the height into batches, which do the better load balance among processes. Thus it can reduce execution time and increase scalability better.

(d). *Other efforts you made in your program*

Before choosing a good chunk size of omp parallel, I did a small experiment to test some different chunk sizes and found that the chunk size 1 is the best.

In mpi_dynamic, I did a small experiment to test some different numbers of rows to be assigned to a slave per time and found that assign “one row” to a slave per time is the best option.

Furthermore, before choosing mpi_static instead of mpi_dynamic as the base of hybrid, I also did an experiment to compare mpi_static and mpi_dynamic. The combination of mpi_dynamic and omp can do the better load balance among slaves, but it sacrifices the master process’s computing capability. Therefore, the combination of mpi_staic and omp has a slightly better result.

Experiment & Analysis

Methodology

(a). System Spec

Using the provided cluster to measure.

(b). Performance Metrics

1. Use `clock_gettime(CLOCK_MONOTONIC)` to measure omp's total execution time.
2. Use `MPI_Wtime()` to measure mpi_static, mpi_dynamic and hybrid's total time.
3. Use `omp_get_wtime()` to get each thread's CPU time, sample code like:

```
double thread_time[num_threads];
for loop {
double wtime = omp_get_wtime();
// some calculation …
thread_time[omp_get_thread_num()] += (omp_get_wtime() - wtime);
}
```

Plots: Scalability & Load Balancing

Test data parameter:

Lower: -1.5
Upper: -1.5
Left: 0
Right: 0
Width: 2000
Height: 2000

In all the experiment results, I choose the longest core time to record. I do the 4 experiments to test the scalability of the 4 versions in 4 different thread and node conditions:

1. In fig. 1, I test the speedup of mpi_static, mpi_dynamic and hybrid in single thread and a single node condition.
2. In fig. 2, I test the speedup of the 4 versions in multi-cores and a single node condition.
3. In fig. 3, I test the speedup of mpi_static, mpi_dynamic and hybrid in single thread and multi-nodes condition.
4. In fig. 4, I test the speedup of mpi_static, mpi_dynamic and hybrid in multi-cores and multi-nodes condition.

Moreover, I also do the experiment to record the load balance of the 4 versions in 12 cores. When measuring the load balance, I don't consider the I/O time of them. The reason is that in all my implementations, the image is written by a single node, so the I/O time will additionally add to a specific node without doubt. Thus, I remove the I/O time. The result is shown in fig. 5.

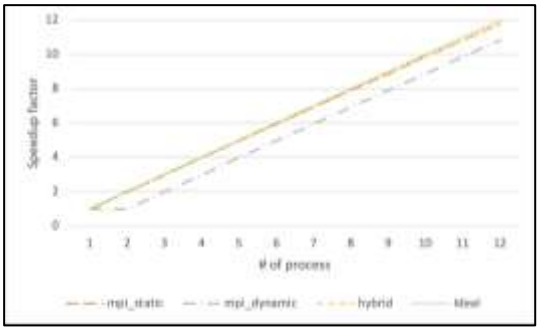

Figure 1. Speedup in single thread, single node.

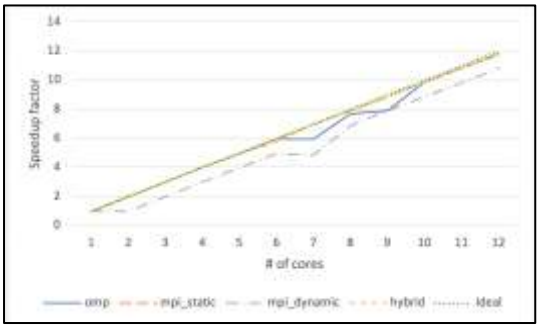

Figure 2. Speedup in multi-cores, single node.

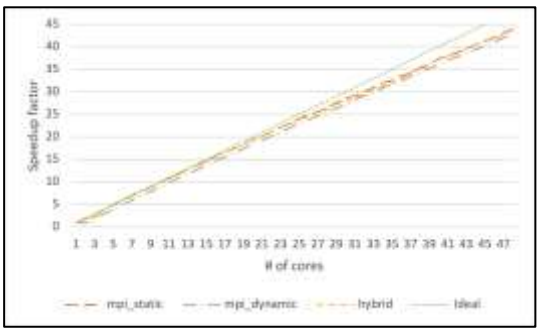

Figure 3. Speedup in single thread, multi-nodes.

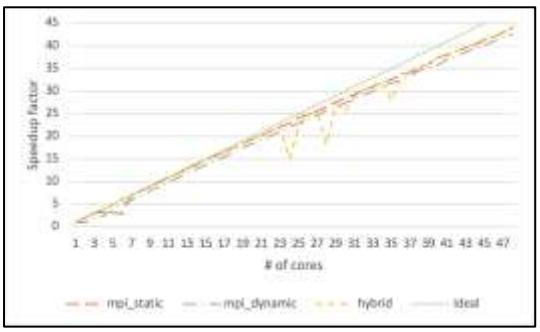

Figure 4. Speedup in multi-cores, multi-nodes.

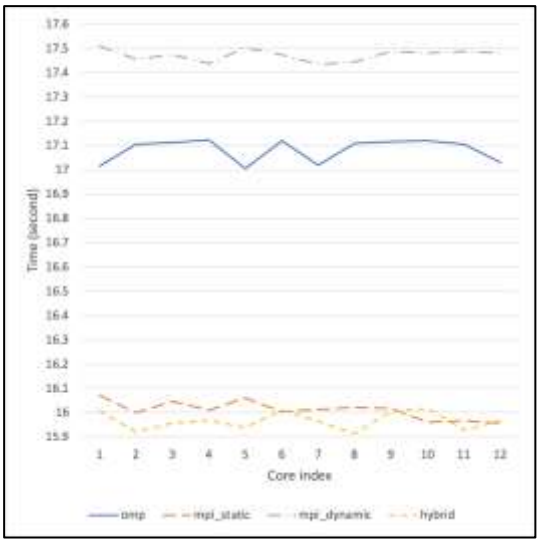

Figure 5. Load balance among each core.

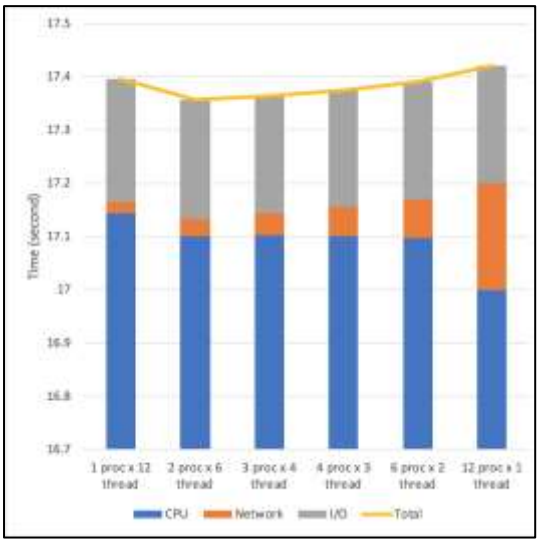

Figure 6. Time profile distribution between processes and threads in single node.

Discussion

(a). Compare and discuss the scalability of your implementations.

I do the 4 experiments to test the scalability of the 4 versions in 4 different thread and node conditions, and I obvious the above the appearance of things:

- **4 versions are all close to the ideal.**
  Surprisingly. The speedup of the 4 schemes is all almost linear and close to the ideal.
- **Omp and hybrid fluctuate in multi-threads sometimes**
  We found that speedup of omp and hybrid fluctuate in multi-threads sometimes, which shows that threads are not so stable as the processes, but overall, their speedup factors increase close to the linear.
- **Mpi_dynamic has the worst performance**
  The speedup factor of mpi_dynamic stably increases in these 4 experiments, but it has the wort performance. The reason is that mpi_dynamic use master-slave structure, and we sacrifice the master' computing capability to assign jobs to slaves. Thus, there is an obvious "one-core" gap between mpi_dynamic and other 3 versions.
- **Mpi_static have the best and most stable performance.**
  The speedup of mpi_static and hybrid are almost overlap since hybrid uses mpi_static as the base. Furthermore, mpi_static has the best and most stable performance since mpi_static does not use threads, which are also the limitation of it.

(b). Compare and discuss the load balance of your implementations.

When measuring the load balance of the 4 versions, I only consider CPU and network time (without I/O time). In all the versions, the load of each core is balanced. Another 2 interesting things are that (1) the lines of mpi_static and mpi_dynamic are similar, and (2) some segments of the line of the hybrid version are similar to omp and other parts are similar to mpi_dynamic.

Others

I test the time distribution between the different number of processes and threads in overall 12 cores and single node. Moreover, I also record the time profile of each case. The recorded data are all from core 0. We can find that the I/O time are all similar since core 0 is assigned to write the resulting image on its own. Another interesting is that using more threads will cause more CPU time, but less network time since the network time is recorded when "processes" communicate with each other. On the contrary, using more processes will cause less CPU time, but more

network time. All in all, when using 2 processes and each process contains 6 threads, we gain the best performance.

Experiences / Conclusion

(a). Conclusion

Omp and hybrid fluctuate in multi-threads sometimes since threads are not so stable. Mpi_dynamic has the worst performance because of the master-slave structure. Mpi_static have the best and most stable performance. Overall, the 4 versions are all close to the ideal, and the load of each core is balanced. In the last distribution experiment, we observe that more processes cause more network time and more threads lead to more network time.

(b). Experiences

In this assignment, I have learned how to write a parallel program on threads and use a master-slave structure. The main difficulty is setting the parameters to run the experiments, especially when executing the hybrid version. For example, in the experiment, the speedup in multi-cores and a single node, I wrote a script containing “48” executing commands with different parameters.