Student ID: 101062319
Name: 巫承威

Homework 3 - Report

- Design
  - (a) What are the differences between the implementation of six versions?
    - i. MPI-Dynamic

      In my dynamic version, I let the master process(rank=0) to handle the communications between itself and all the other slave processes. At first, it will assign the task to every other slave process. Then the slave process will consequently receive the message sent from the master(the unit of transmission message is one row of data). After finishing computing the results of data, slave process will package the result and send it back to the master. Second, after slave process sending the result back to master, it will continuously wait for the next assigned command(task) from master. On the other hand, from master process's point of view, when it receive the result from slave process, it will decrease the number left jobs(or increase the number of finished jobs) and check if all tasks were finished. If all tasks were finished, it will send the flag to all slave processes to indicate them to terminate themselves. Otherwise, it will continuously assign the task to the slave process who is idling(waiting) for the next task. That's the nuclear concept of my implementation.

    - ii. MPI-Static

      The differences between static version and dynamic version are about three main points. The first one is that the program will compute the number of row of data, then try the best to assure that every slave process will be assigned the same number of task. And the second one is that the master process won't send the message to other slave processes. It only wait for receiving the results from slave processes, and finally draw the graph or not. The last one is similar with the second point that the slave process won't receive the message from any other process, it will do it best to compute the result(the task assigned to it), then send it back to the master. The above three points are the main differences between the mpi-dynamic version and mpi-static version.

iii. OpenMP-Dynamic

In this version of implementation, I only add the syntax of executing OpenMP to the sequential version, and change some structs of the codes to follow the rules of OpenMP. And I don't assign the chunk size of this dynamic version. Since I set the chunk size to be (#row of data / #thread) both in dynamic version and static version in the beginning, I found out the results are almost the same, and it's difficult to recognize the difference between these two version, so finally I remove the chunk size.

iv. OpenMP-Static

The only difference between static version and dynamic version is the parameter of the OpenMP syntax like following codes:

```
#pragma omp for schedule(static)
#pragma omp for schedule(dynamic)
```

That's the only difference.

v. Hybrid-Dynamic

The hybrid version is the most complex one; however, after finishing the MPI and OpenMP version, it will be easier to implement. In this version, I use MPI version to be the template. And I parallelize the for loop in the slave process(the computing segmentation) by OpenMP. So this version will be something like the Kinder Surprise, the outside chocolate part is like the MPI implementation and the inside toy just like the OpenMP implementation, which also represents that the backbone computation part is done by OpenMP.

vi. Hybrid-Static

The hybrid static version is more complex than the dynamic one since it need to decide the number of task assigned to each slave process in the beginning. Another difference is the same with the OpenMP version, the static syntax and dynamic syntax.

(b) How do you partition the task?

I partition the task by the row of data. Since the graph has its own number of pixels(points) of width and height, I partition the row into several blocks, and assign it to the processes or threads. For example, in my MPI-dynamic version, the master process will send one row of data to one slave process at a time. And slave process will also send one row of data to the master process at a time again. Another example for static version is my MPI-static version, it will compute the number of rows of data of each slave process, so the master process will send several rows of data at a time, which like the shape of rectangle(a block of data). And also the slave will return the same size of data. That's the main concept of my six implementations of different structure of parallelizing the codes.

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

I store the graph data by dynamically allocating the space. And in my MPI implementation, I use the function "MPI_Isend" to perform the non-blocking call, and do little variables computation before the "MPI_Wait" call to synchronize(although it seemed to has no difference and even get worse…, I will show the graph below).

(e) Other efforts you've made in your program?

I define some mathematical computation equations to be the type of macro such as computing the cube or square value of specific number. And also define some value which will be use in high frequency such like

"#define DRAW_CONST_MUL 1048576"

"#define DRAW_CONST_MOD 256"

while computing the color information of points.

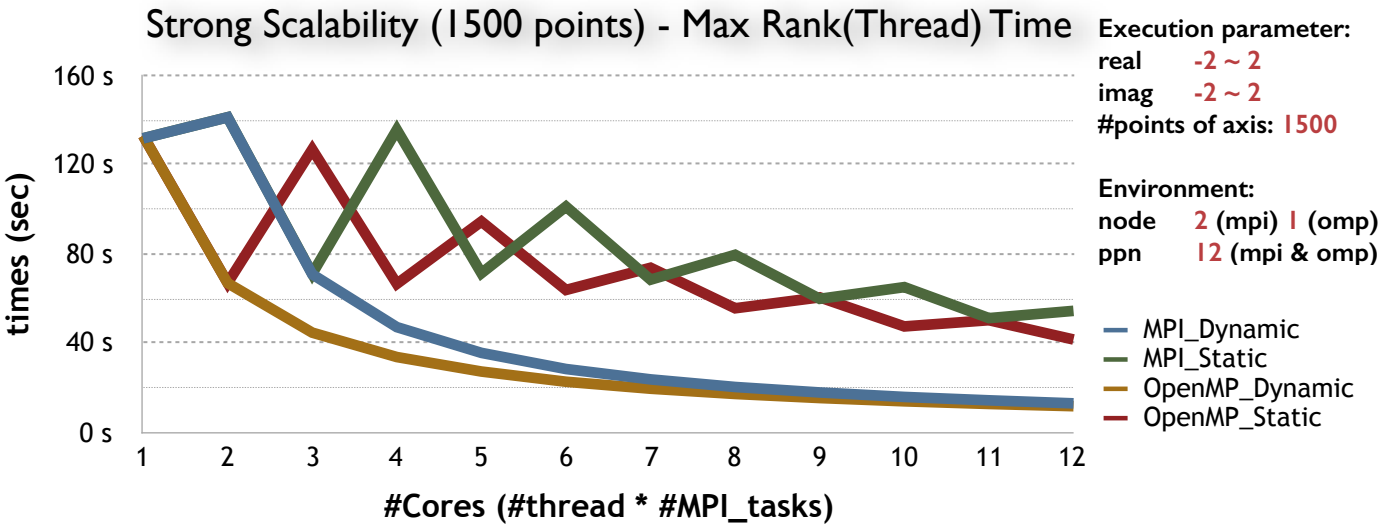
• Performance analysis

(a) Scalability Chart

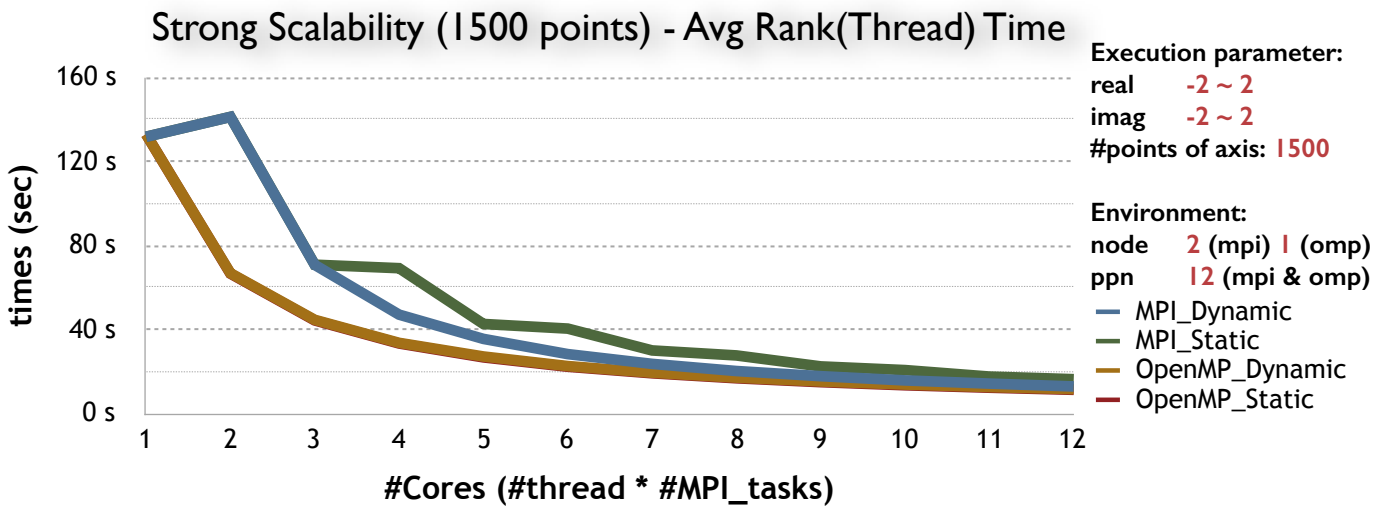
i. Strong Scalability (scalability to number of cores)

(Problem size is fixed, 1500 points for both real-axis & imag-axis)

{MPI, OpenMP} x {Dynamic, Static}



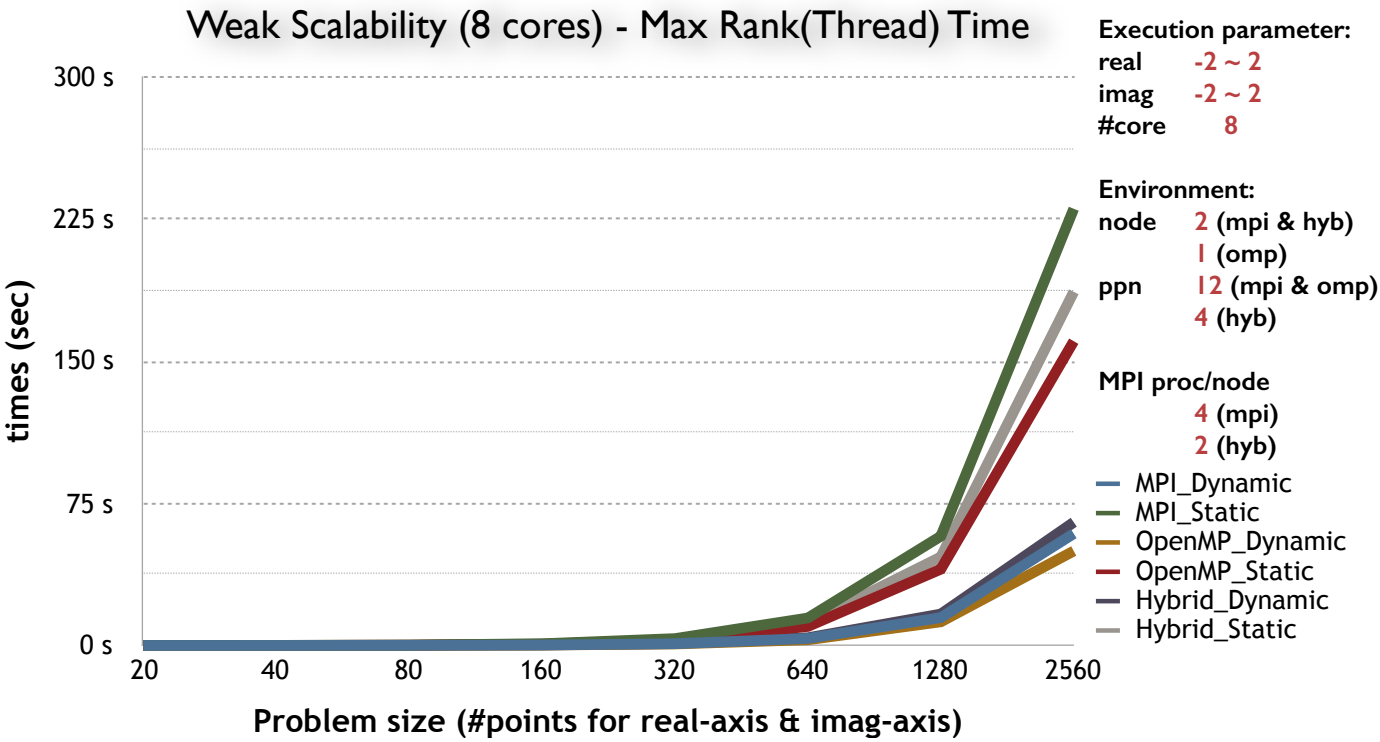
To plot the above graph, I run the experiment that use 2 nodes to create the combination of 12 cores. It's apparently that the dynamic versions get better performance while more cores are working simultaneously. Since all tasks of the program will be assigned to each core more efficiently, it can prevent cores form idling compared to the static version. And I plot the points according to the rank(thread) who takes the longest time.



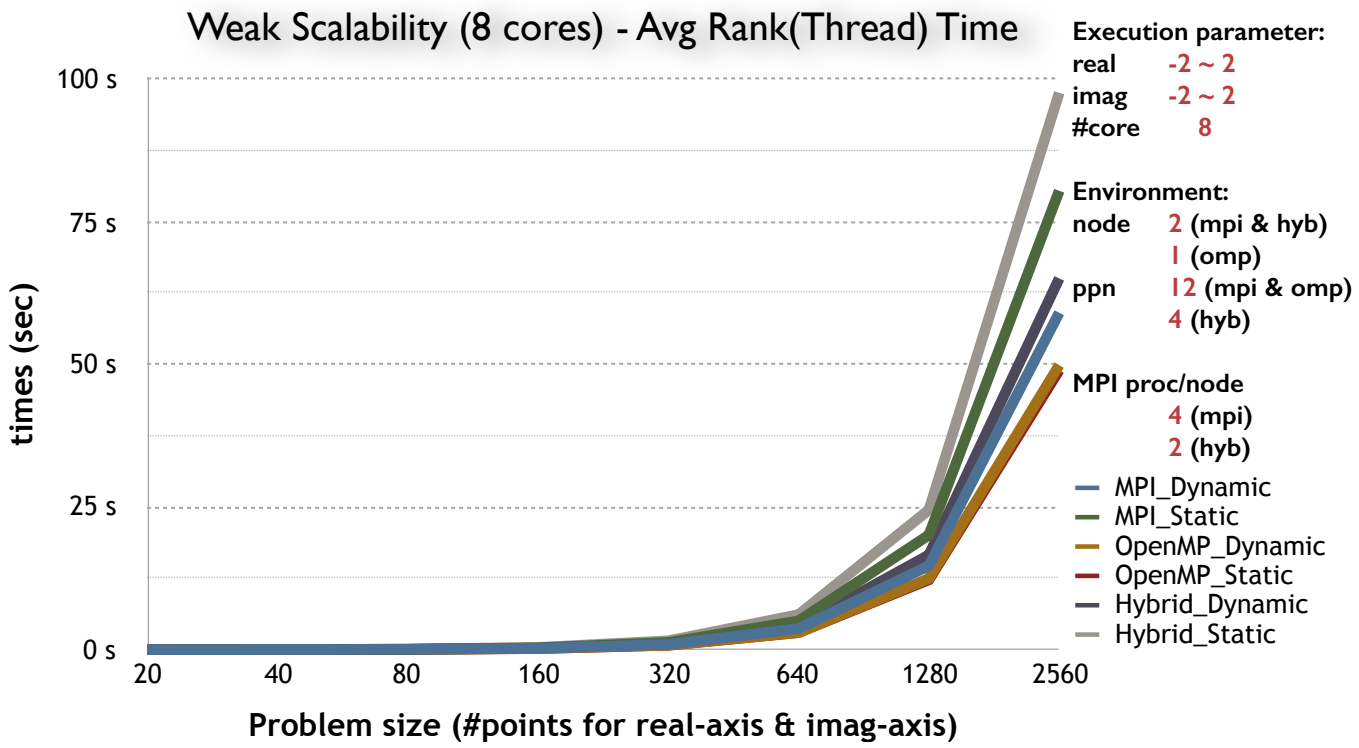
This graph's input parameters are the same with the previous graph. However, different from the previous one, it plot the average time of all rank(thread). To my surprise, if I chose to record the average time of all rank(thread) in static version, it's results are almost the same with the dynamic version. With the increasing of the number of cores, both dynamic and static version take less time to finish the task.

ii. Weak Scalability (scalability to problem size)

(#cores is fixed, 8 cores for problem) {MPI, OpenMP, Hybrid} x {Dynamic, Static}



In this experiment, I try the different problem size to see the execution time. I set the number of point in both real-axis(X-axis) and imagine-axis(Y-axis) from 20 points to 2560 points. According to the results, while the problem size is small enough(#point < 320), every version get almost the same performance. Nevertheless, if the problem size increases up to 640 points and higher, it's easier to see that the dynamic versions get better performance than the static version. Something interesting is that while in the **static** version, the performance: **OpenMP > Hybrid > MPI**, and the **dynamic** version, the performance: **OpenMP > MPI > Hybrid**.

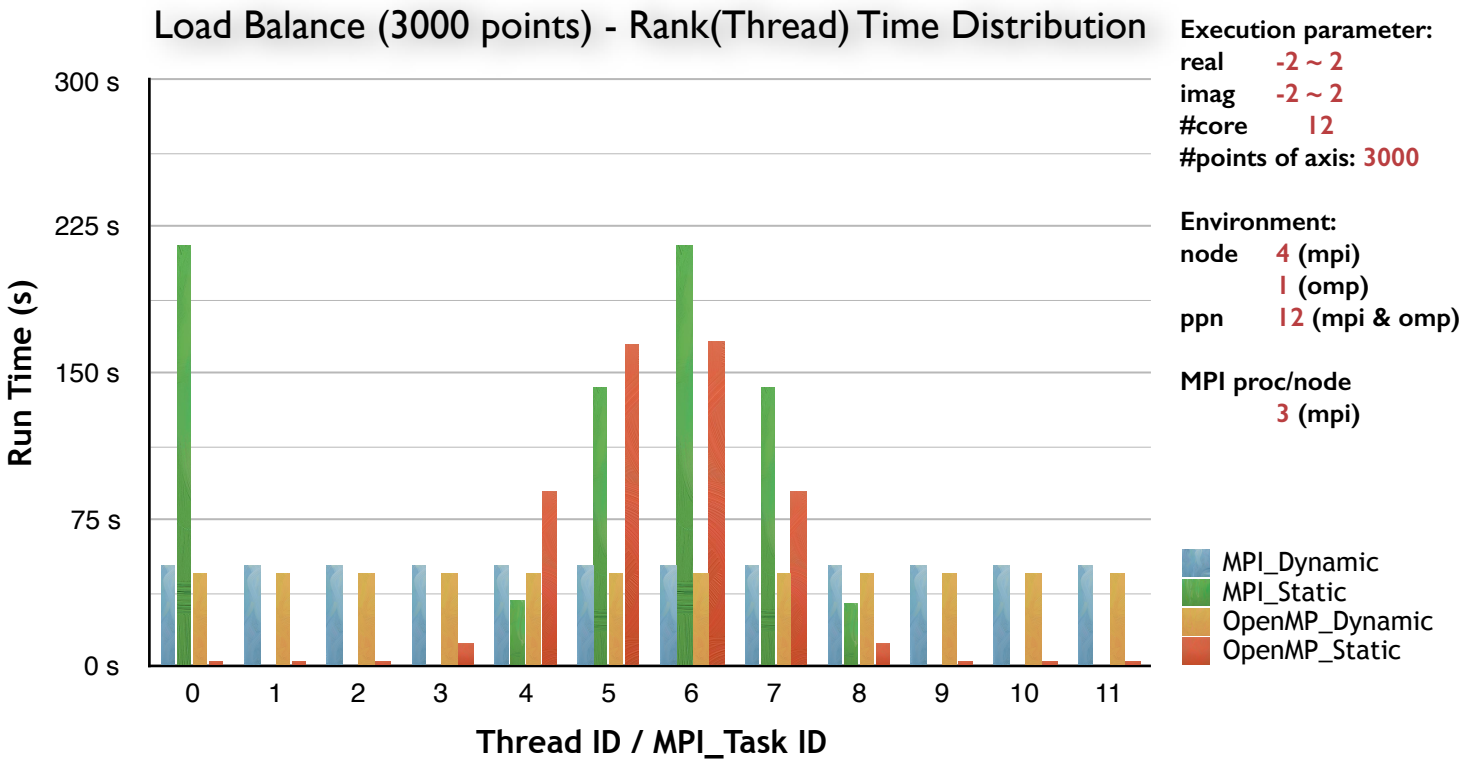


And this experiment record the average execution time of every rank(process, thread). Also something interesting are that the OpenMP version also both get better performance, MPI-static version is better than the Hybrid-static(although Hybrid use the OpenMP) but Hybrid-dynamic is better than MPI-dynamic.

(b) Load balance chart

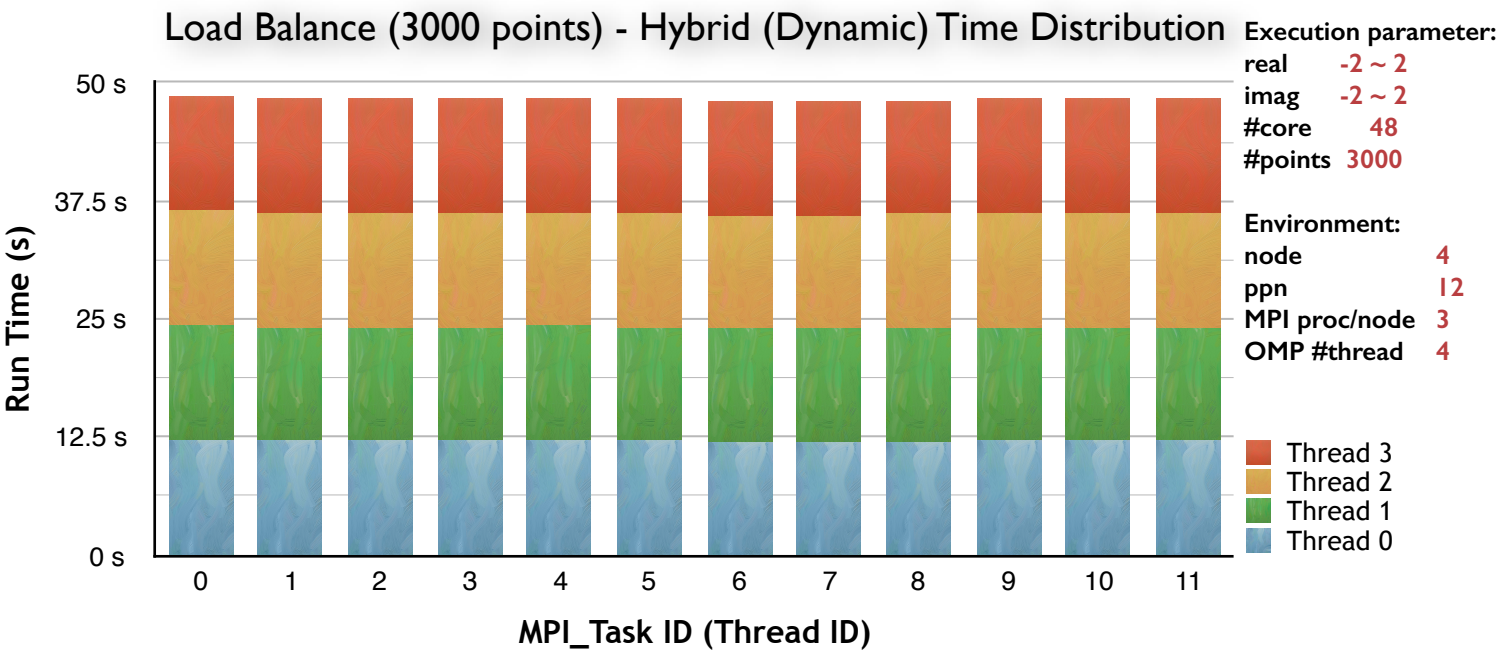
{MPI, OpenMP, Hybrid} x {Dynamic, Static}

i. Load balance of MPI & OpenMP



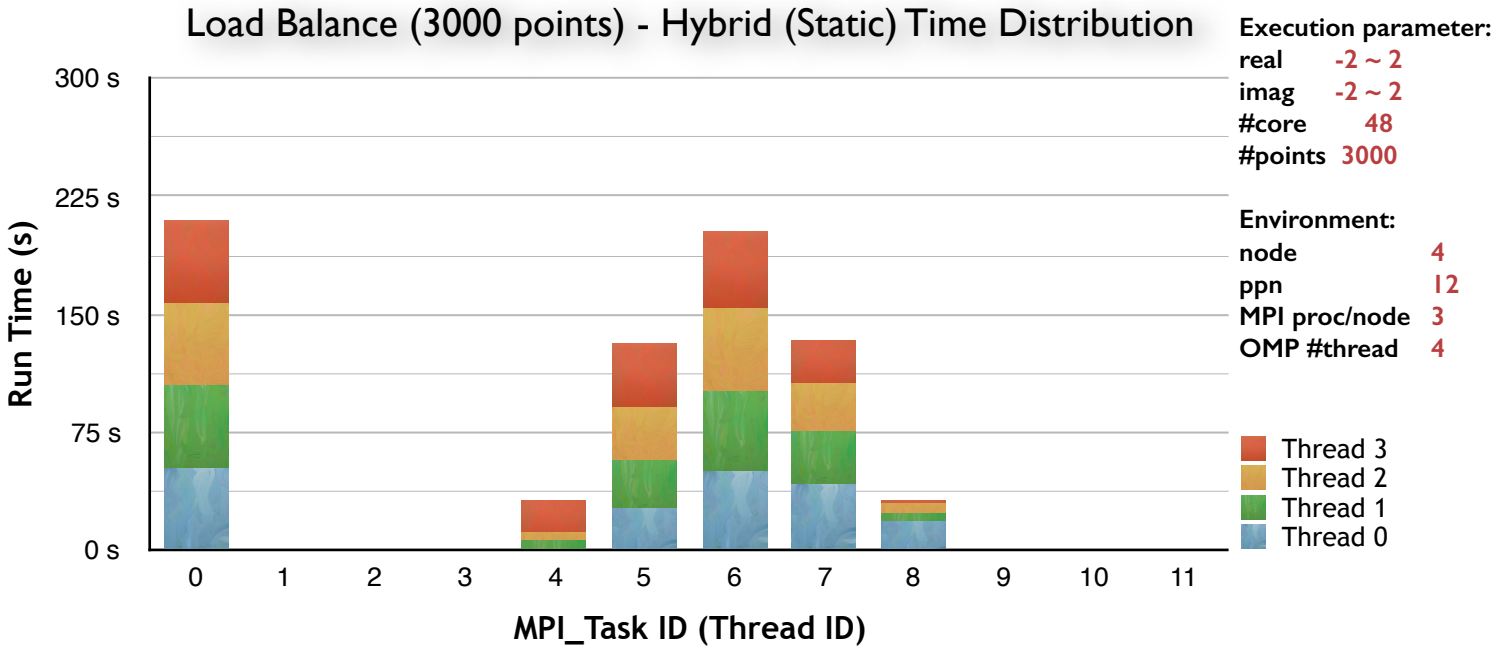
The above graph shows that both MPI and OpenMP's dynamic versions are much more balancing than the static versions. What's more, both MPI and OpenMP's static versions rank(thread) time are symmetric to the middle rank(thread), except the master process of MPI-static version since it will continuously wait for the other processes' computing results.

ii. Load balance of Hybrid-Dynamic (MPI + OpenMP)



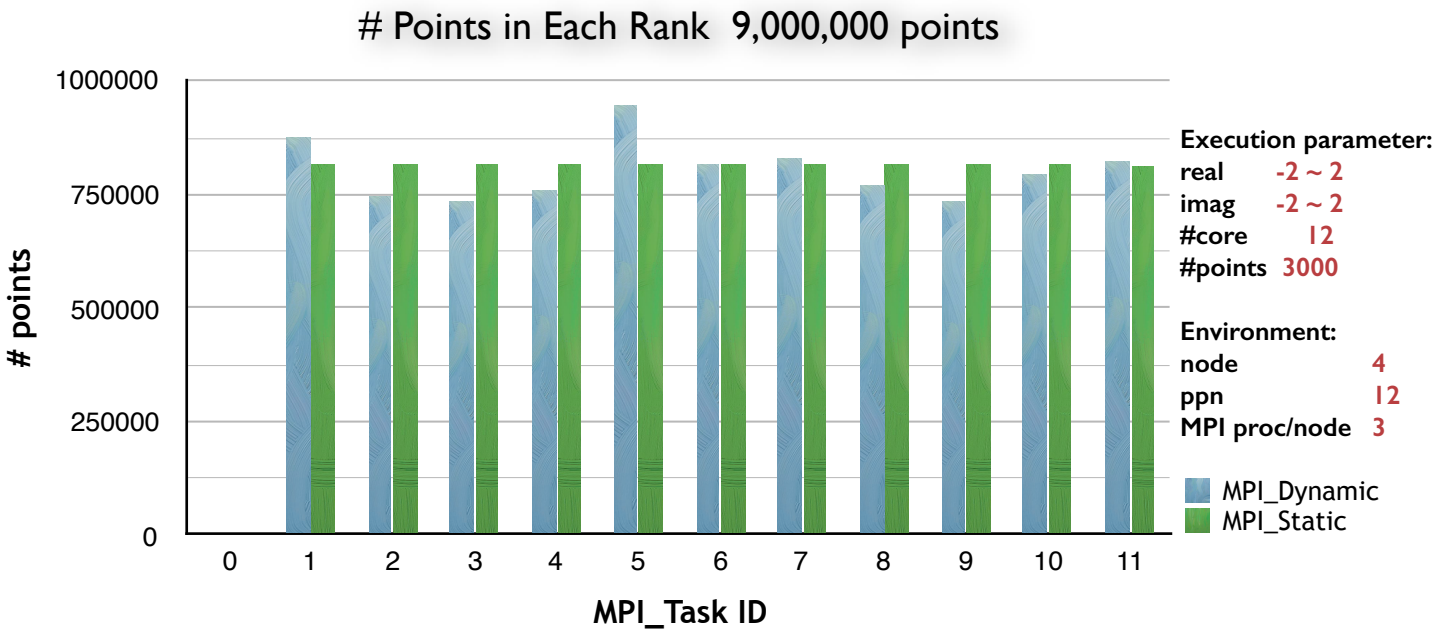
This graph shows the time distribution of Hybrid-dynamic version of all ranks from 0 ~ 11 and their own threads. It's easy to see that all ranks and all group of threads have consumed almost the same time. That's why the dynamic version is more balancing than static version. The static version will be showed below.

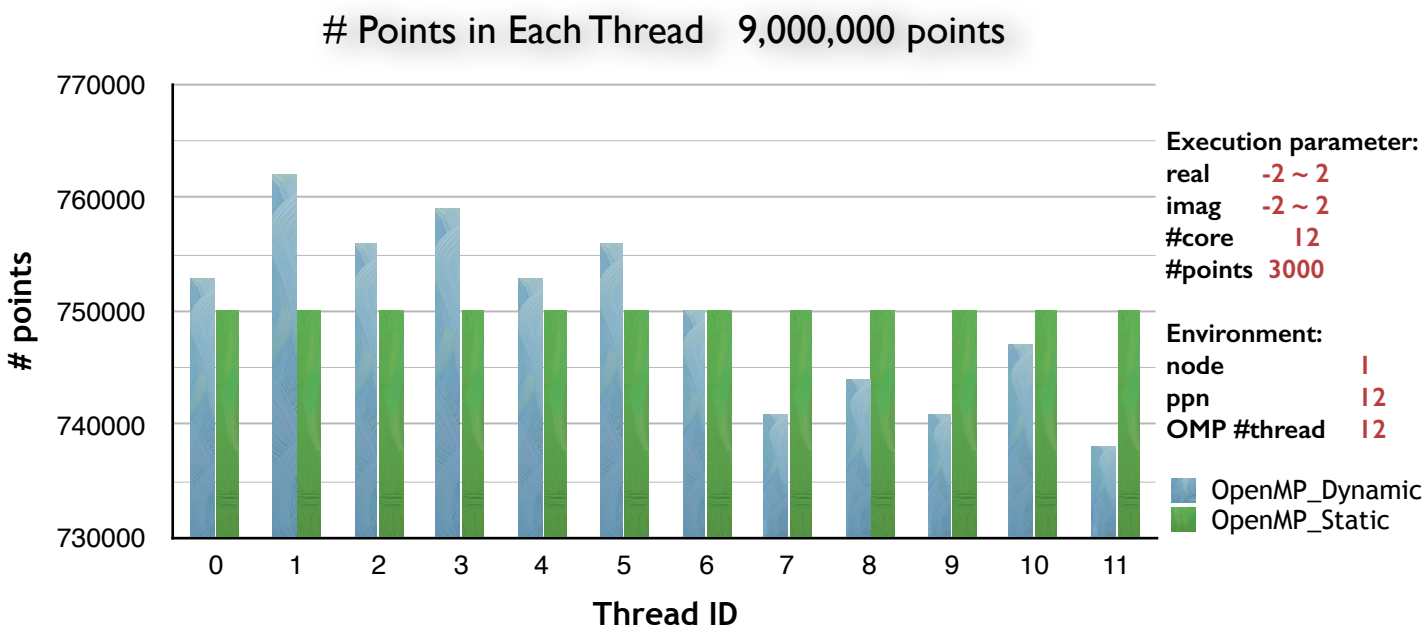
iii. Load balance of Hybrid-Static (MPI + OpenMP)



As the graph showing, Hybrid-static version's time distribution is similar with MPI-static and OpenMP-static versions. The master process consumes the most time, and the middle id slave process also consumes much more than other slave processes. The distribution shape is also symmetric.

iv. #Points in Process(Thread)

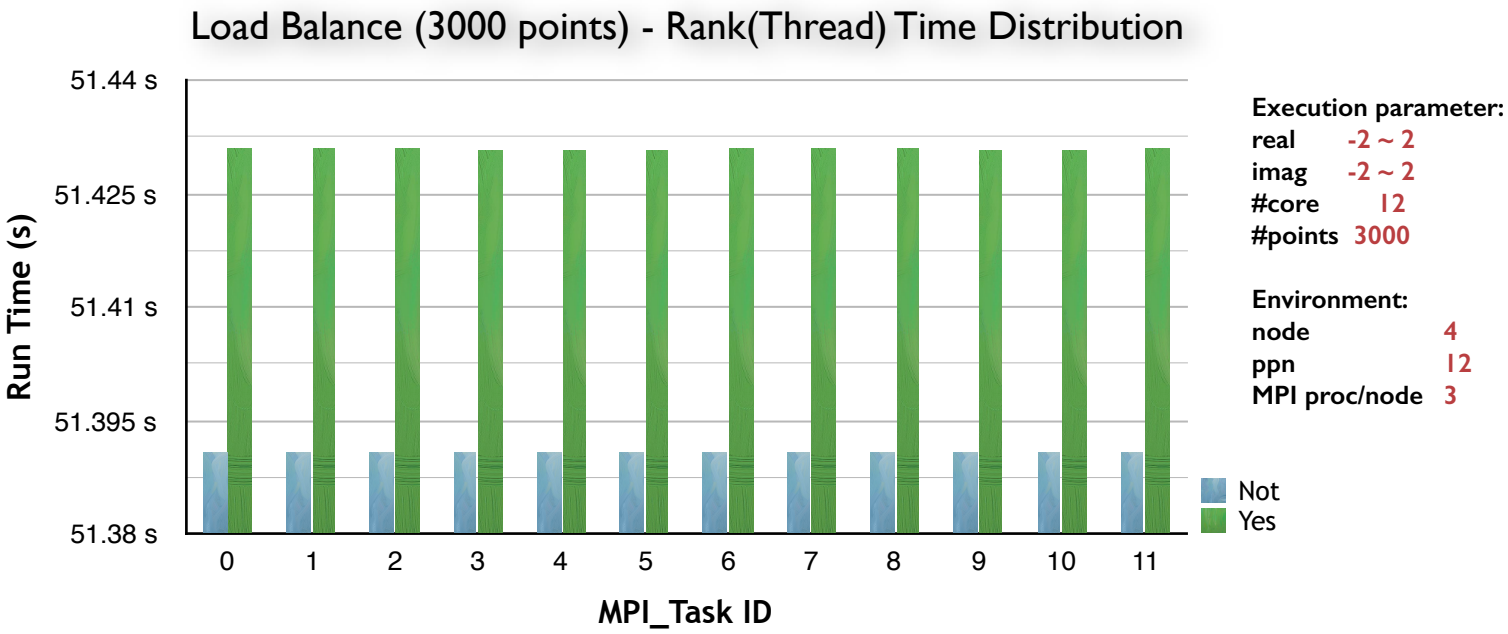




These two graphs show that the distribution of the computed points of the Mandelbrot Set graph. As they showing, static versions assign the task(#point) to every process(thread) more equally, but the dynamic versions don't. Since dynamic versions' implementations and main concepts are that assigning the task to the process(thread) which is idling and waiting for the next task. Thus, if some processes(threads) are running fast, they will gain more tasks. And that's why some specific processes(threads) will consume more time to execute the codes.

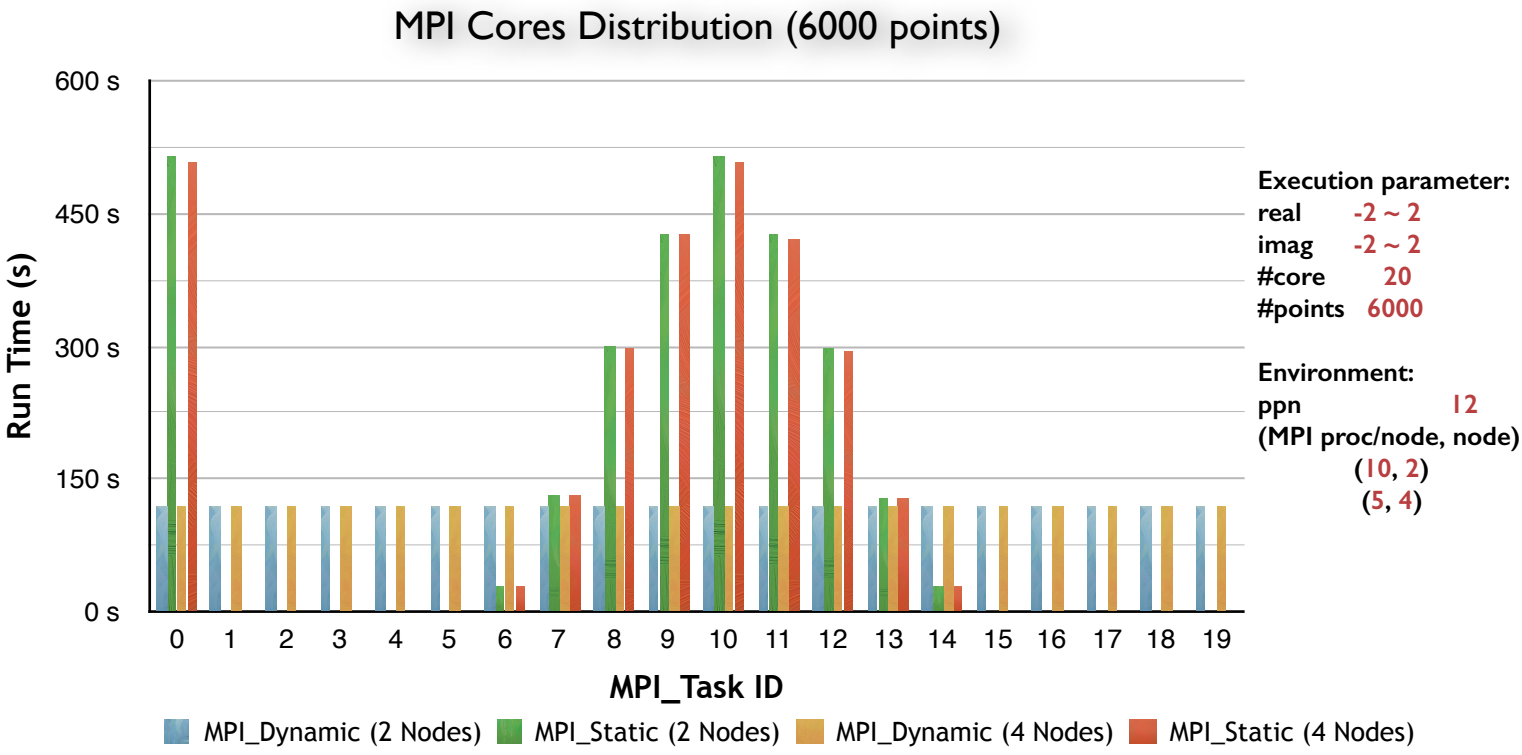
(c) Other chart

i. MPI-Dynamic ("computation between MPI_send & MPI_wait" and "NOT")



This experiment is testing the difference between if there exists the computation between the MPI function call "MPI_Isend" and "MPI_Wait". The computation part must exist, I just change the location where the program executes it. The "Yes" blue bar represents "yes", there exists the variable computation part. On the other hand, the "Not" green bar represents "not", there is no computation part between them. To my surprise, if there is no computation part between them, it will get a little bit better performance than "yes" bar.

ii. MPI-Cores Distribution (20 cores for 2 nodes & 4 nodes)



This is another experiment which test the process distribution between different nodes. I set the points in each axis to be six thousands and use twenty cores. By the above bar graph, as the number of nodes increasing, its corresponding elapsed time will decrease. In other words, although they are both have twenty cores, if these twenty cores were in more nodes, its efficiency will increase. So the "MPI_Static (2 Nodes)" takes more time to execute the program than "MPI_Static (4 Nodes)", and so does the dynamic version even there is only very small difference.

• Experience

(a) What have you learned from this assignment?

At first, I learn how to use the different MPI function call compared to the HW1 such like "MPI_Isend" & "MPI_wait". Second, I learn how to parallelize the sequential code by using OpenMP like "#pragma omp parallel" and "#pragma for schedule(dynamic)". And also understand the difference between the dynamic and static implementations in OpenMP. Finally, while I am trying to implement the Hybrid version, I find out that it is just use MPI version to be the backbone(template), and parallelize the for loop in it by using OpenMP syntax. This combination of MPI and OpenMP interests me a lot.

(b) What difficulty did you encounter when implementing this assignment?

I think the most difficult part is how to implement the first code. Since I need to parallelize the sequential version to MPI version, I need to consider the relationship between master process and other slave processes such like when should master send the task to slave processes and when should the slave send back the computation results. After finishing the first MPI version, it easier to implement the OpenMP version since it only need to add some OpenMP syntaxes before the for loops. I think that's the most difficult part.