Paralllel Programming HW3 - Mandelbrot Set

111062526
邱琬晶

1. Implementation
    a. Pthreads version with vectorization
    I used the dynamic load balancing for this version of assignment. In the end, after trying several load balancing method, I chose to distribute the workload based on row. Every threads will ask for a row if they have nothing to work on. When a thread asks for workload, they need to first acquire the lock of the current row, then if there is no work available anymore, every finished threads will exit and finished the program.

    Next, since vectorization is effective in dealing with 2 data simultenously, I calculated 2 pixels at a time using vectorization techniques. In addition, to save time as much as possible, every time if one of the pixels inside the register has finished, then the register will continue loading new pixel and also the remaining pixel from before that has not finished yet. Since in this version, the frequency of changing the register value is high, union data structure is especially useful here, as we can read and change the value more efficient compared to store then load.

    b. Hybrid version with vectorization (MPI + OpenMP)
    For hybrid version, I divide the workload by row and each of the process will get **at least** $\lfloor \frac{n}{size} \rfloor$**,** n is the height in this case and size will be the number of processes then for process that has rank >= the remainder of $\frac{n}{size}$, it will get extra 1 row, compared to the process that has rank < remainder. Since I want to utilize the parallel omp for collapse, I use 2 loop to iterate each pixel of its workload, and since vectorization is more effective on computing 2 data simultaneously, I iterate 2 column at once, which means that if width is odd, then there will one remaining column for each row. Therefore, I add another loop after the loops before have executed, the new loop will iterate every 2 row, because 2 last column data will be calculate at the same time by using vectorization. In the end, if the number of row that the process has to work on is odd, then there will be one remaining pixel need to be calculated. The last remaining pixel will be calculated separately. The illustration is similar to the following figure. Here the row that the process has to work on is 5 (example) and

the width is also 5. The order of execution will be red, then green, and lastly orange. In addition, the number can indicate the order of iteration.

0	0	1	1	10
2	2	3	3	10
4	4	5	5	11
6	6	7	7	11
8	8	9	9	12

In this version, since I wanted to use the “omp for collapse”, so I arrange the calculation, such that if one of the pixels has finished, it has to wait for its partner to finish as well. For the scheduler in omp parallel for, I chose the dynamic scheduler with chunk size of 5, the reason for this can be seen in the experiment section later.

Next, is the row distribution, I distributed the row similar to round-robin method, so a row will be distributed into row % size, for example, there are 3 processes, then row 4 will be distributed into process that has rank 2 (5 % 3 = 2). The illustration is similar to the following: (size = 2, number is the process’s rank).

0	0	0	0
1	1	1	1
2	2	2	2
0	0	0	0

This way, we can avoid the locality properties of Mandelbrot set as much as possible. The work load for each process will be much balanced compared to distribute the row in order, and increase scalability as well, which can be seen from the plot in the next section.

Lastly, since each process will have its own version of local image, we need to combine them into one complete image. I combined them using MPI_Gatherv

from MPI. Each of the process will have its own local image with size (the number of workload * width), since the row is not in order after MPI_Gatherv, we need to recalculate the index of the complete image when writing the image into a png image.

Also, I used union data structure to access the data inside the register, instead of use the store intrinsics and load intrinsics instead of using set, because as I will mention later in the experience section of this report, load is a lot faster compared to using set, and union is faster compared to store the data from register to the memory every time I need to access the data.

2. Experiment and Analysis
    a. Methodology
        i. System Spec:
        Used the system TA provided to us.

        ii. Testcase: strict 30
        Iters: 10000,
        Left, right: -0.5506211524210792 -0.5506132348972915
        Lower, upper: 0.6273469513234796 0.6273427528329604
        Width, height: 7680 4320

        iii. Time:
        Communication time (hw2b): Time to do the MPI communication, in the program, it is the time to do the MPI_Gatherv, to gather all the local image from each processes.

        CPU time (hw2a, hw2b): the time to do the rest of the computation, including variable initialization and others.

        The time is measured using clock_gettime(CLOCK_MONOTONIC, &starttime);

b. Plots:

i. Scalability

Figure 1a:

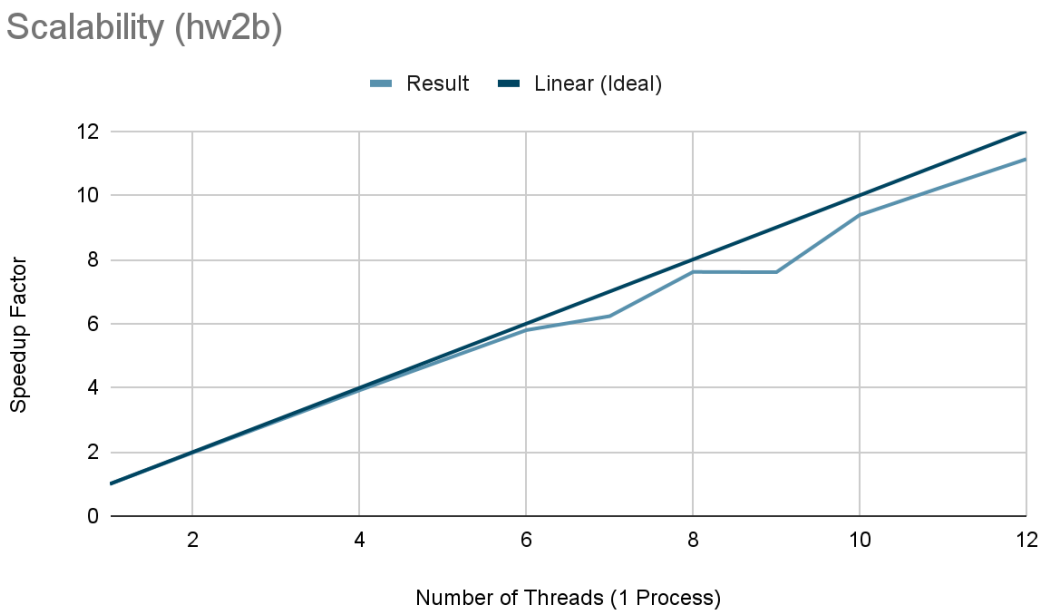


Figure 1b:

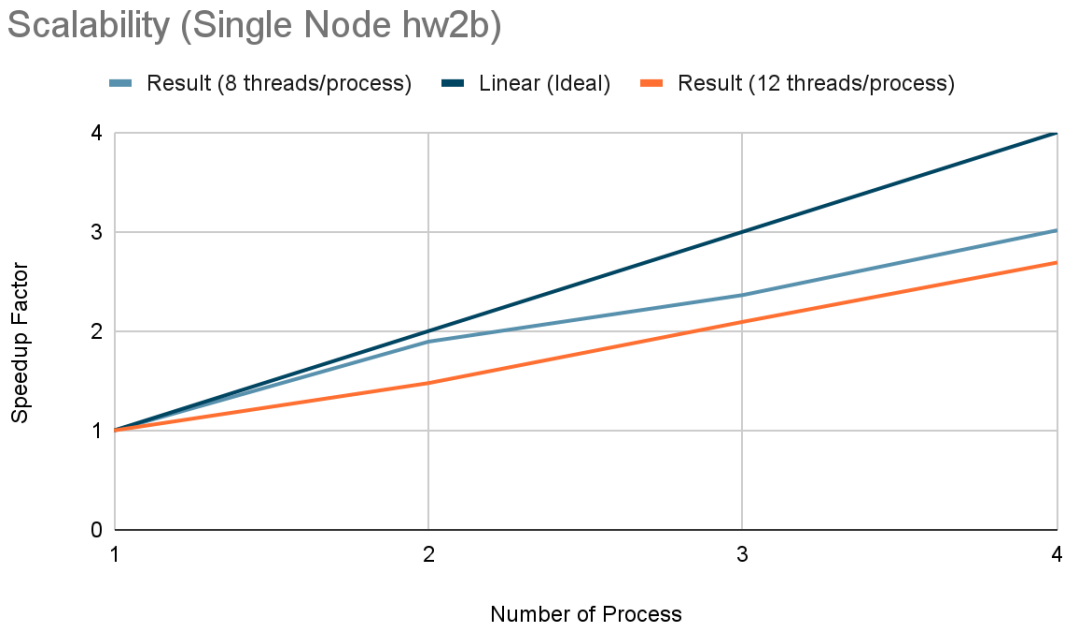
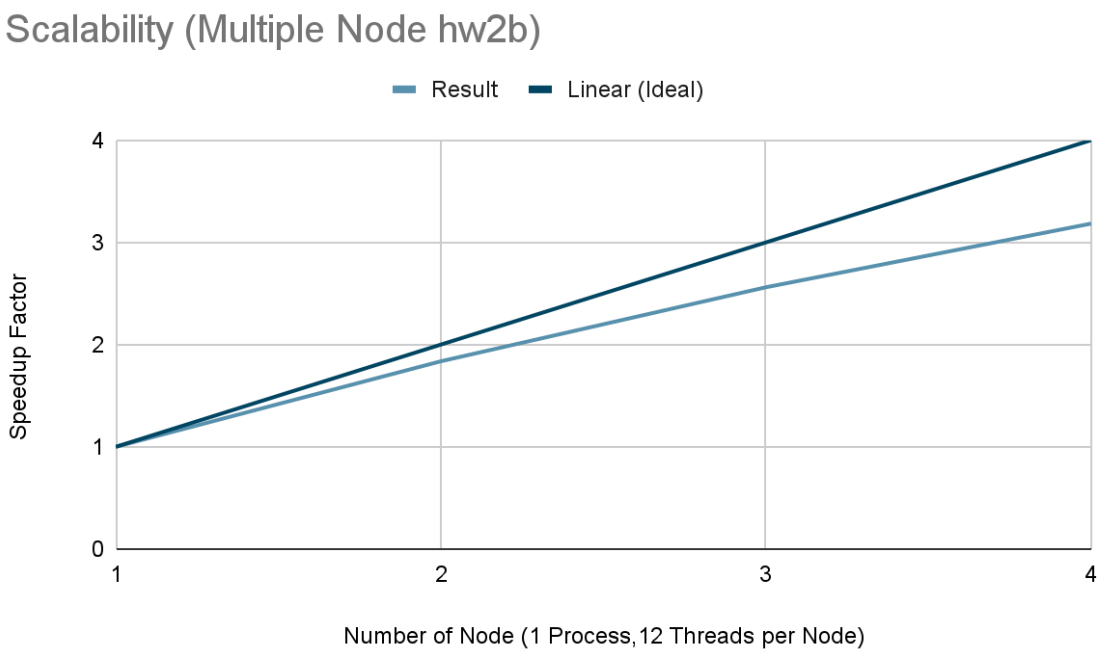


Figure 1c:



ii. Load Balancing

Figure 2a:

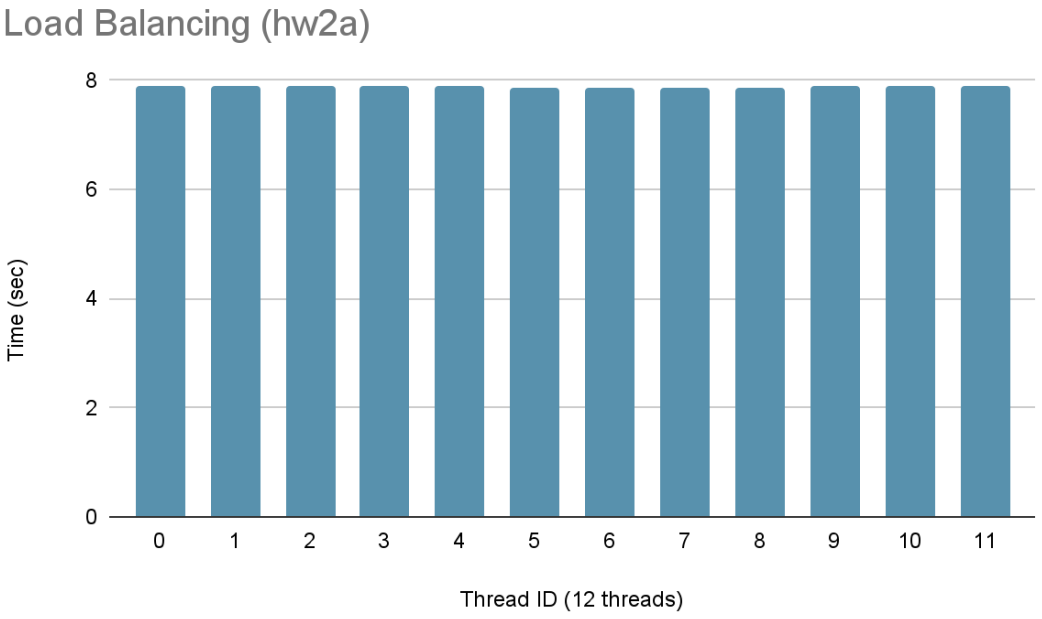


Figure 2b:

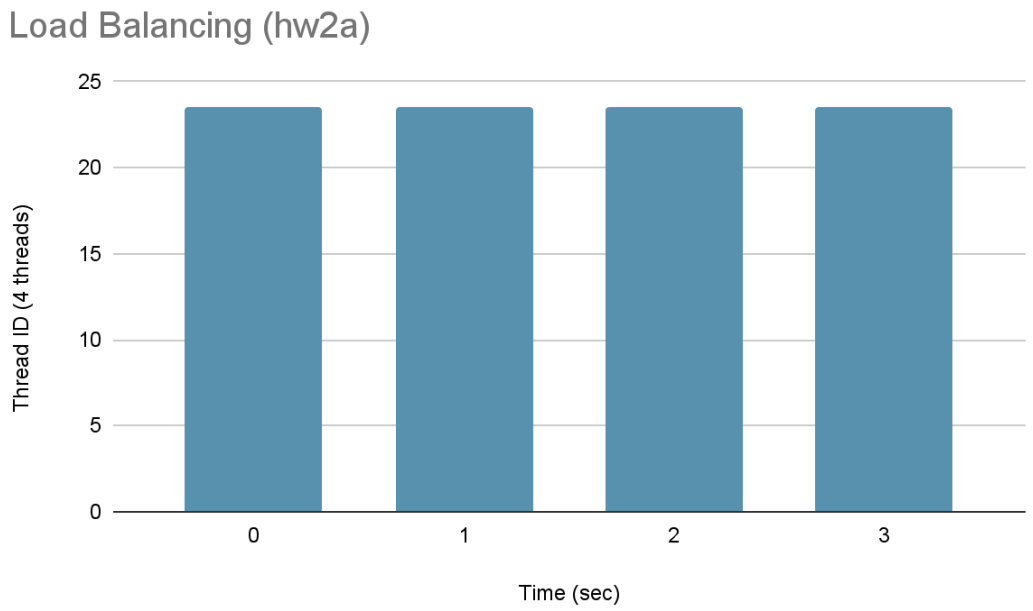
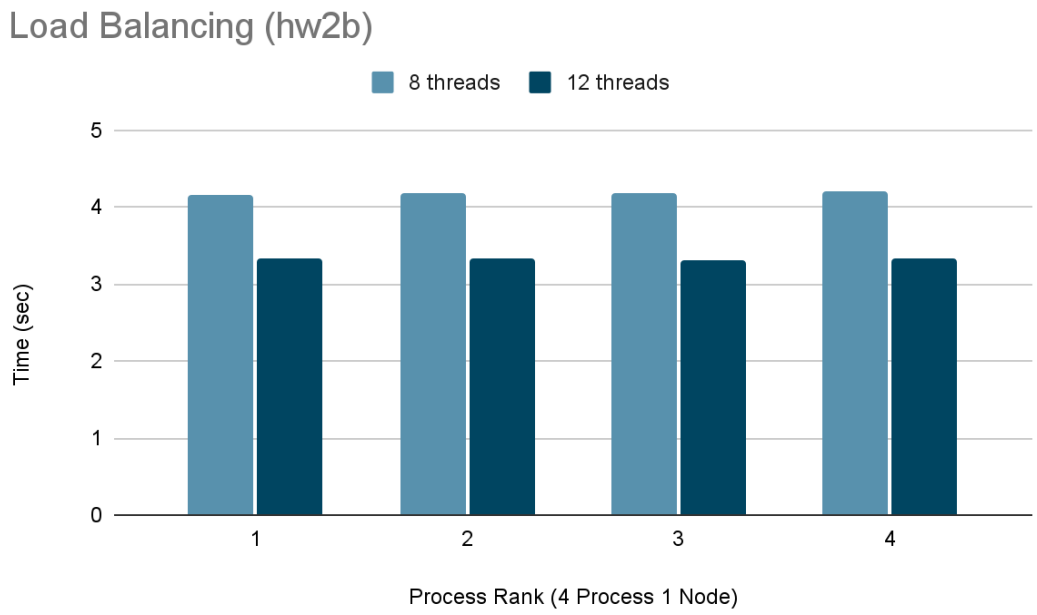


Figure 2c:



c. Others

Figure 3a:

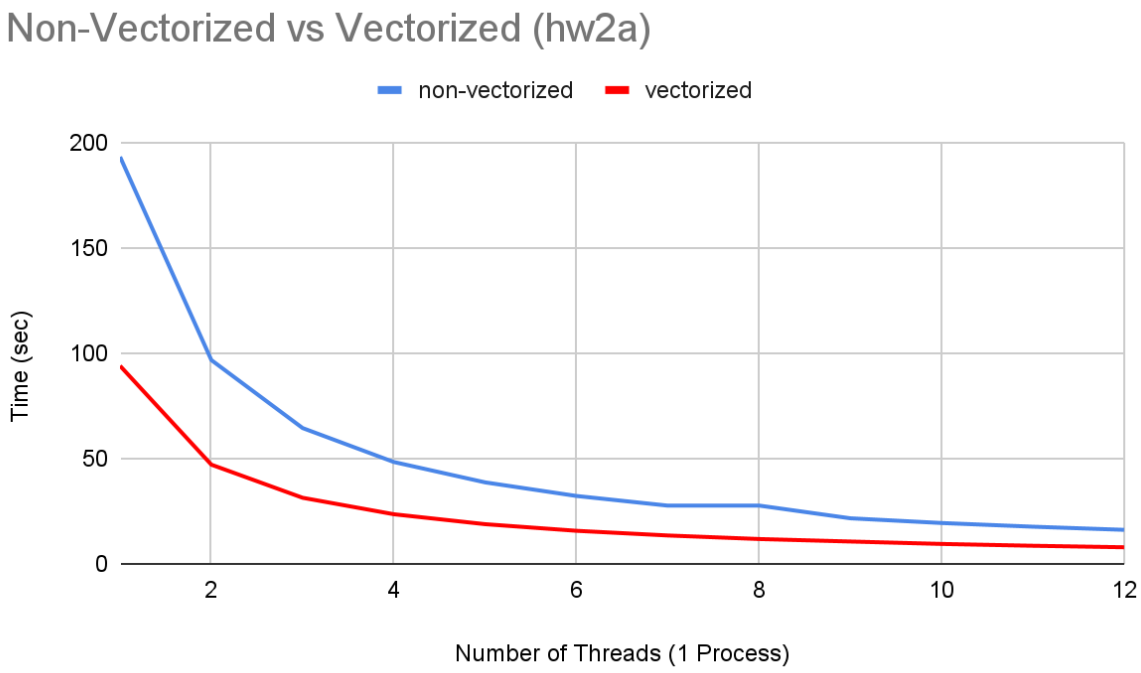


Figure 3b:

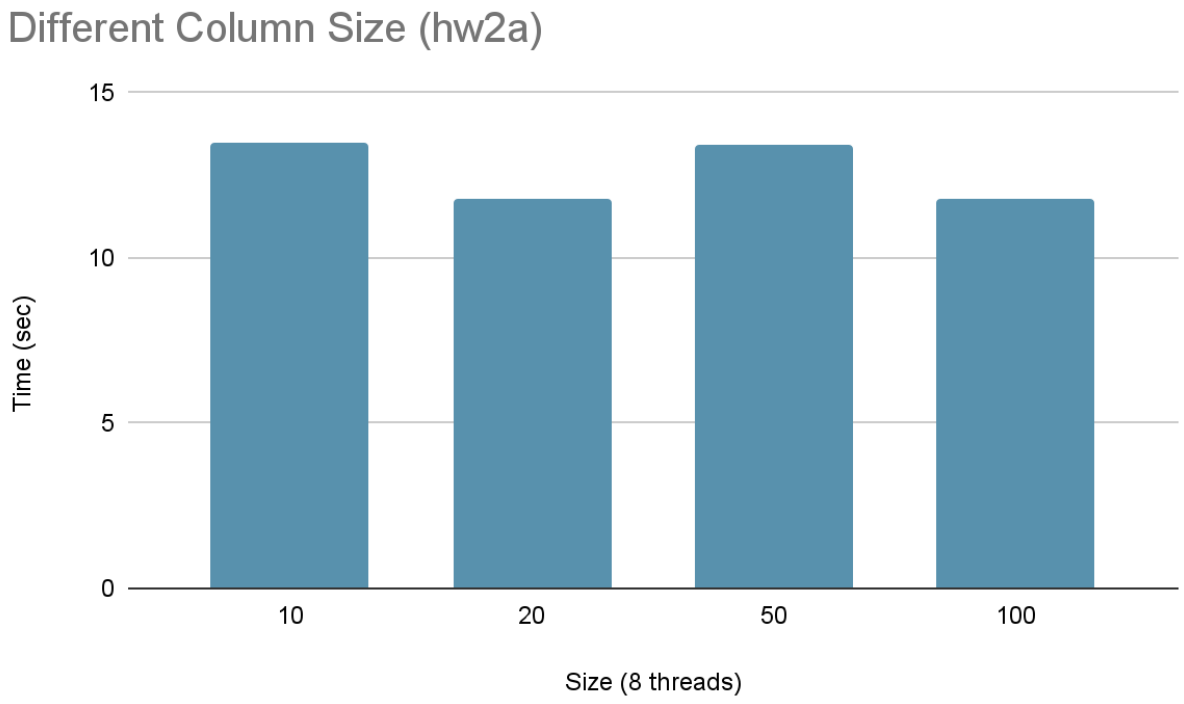


Figure 3c:

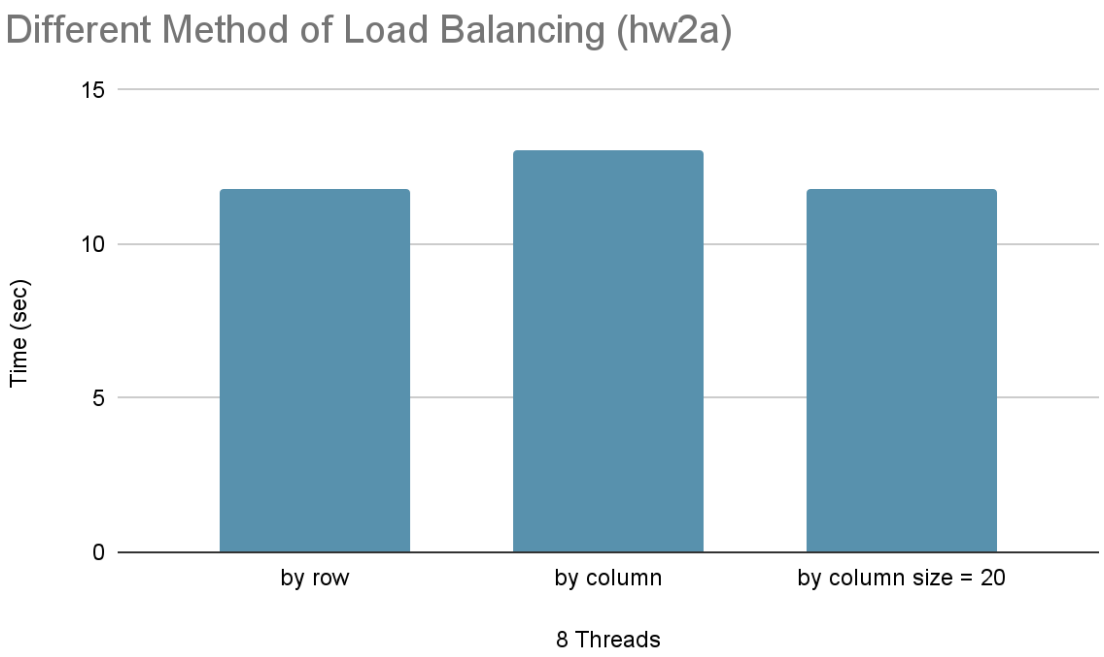


Figure 4a:

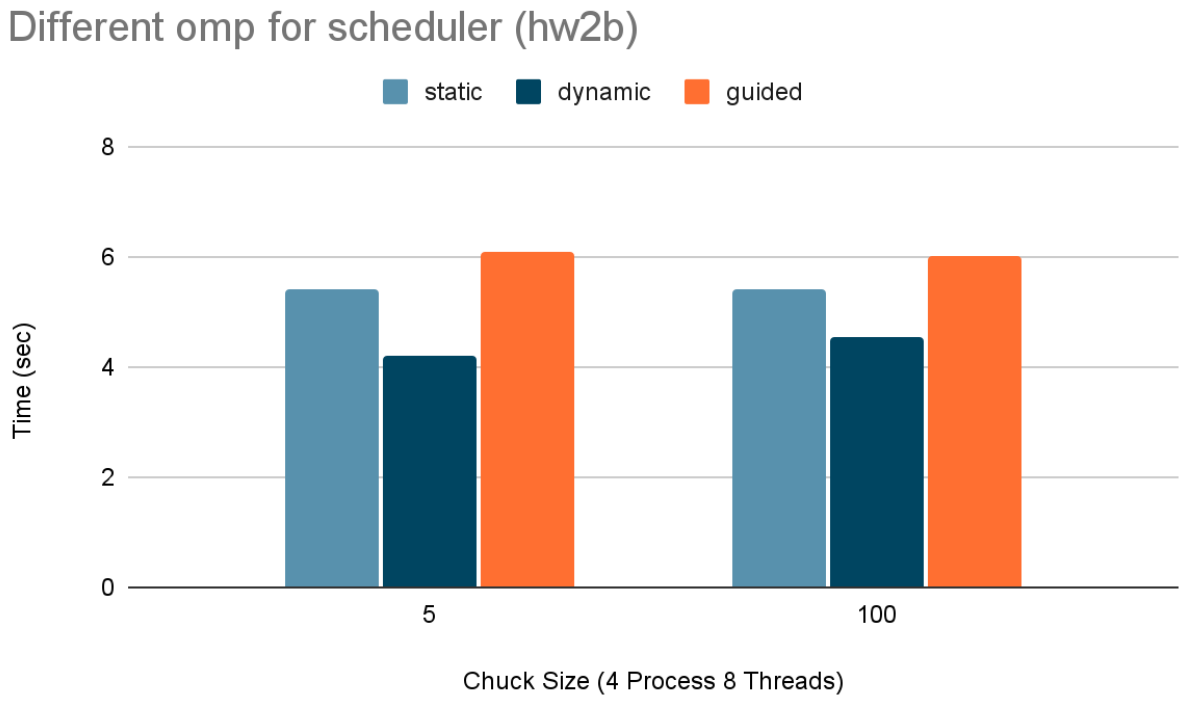
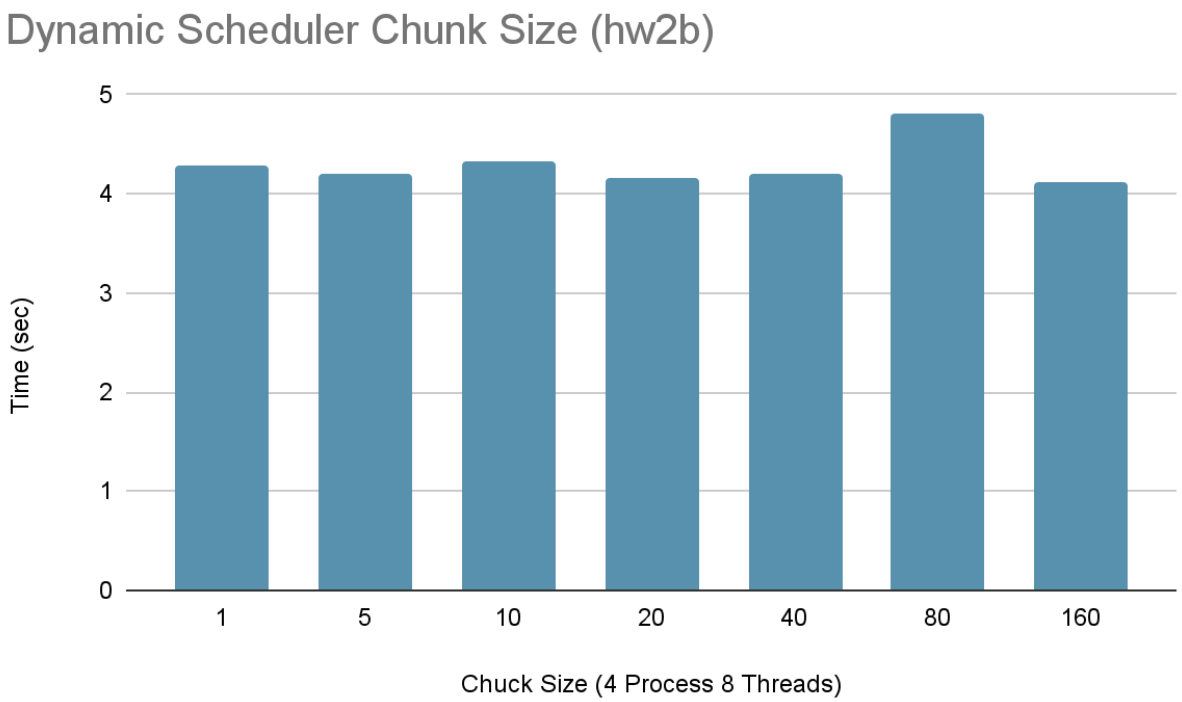


Figure 4b:



d. Discussion

As it is shown on the scalability plots, the scalability for both hw2a and hw2b with vectorization almost reached the ideal values. Speedup factor is supposed to be compared with the sequential version of the program, but here I compared each values to the either the version of only using 1 process and 1 threads (figure 1a and 1b), or to the version of only using 1 process and 1 node (figure 1c). If we compared them to the sequential version of not using any vectorization, the result will be over the ideal scalability plot.

From the load balancing plots, we can see that the work load for each process or each thread is balanced as the finishing time for all of them is quite the same. The work load is still balanced even with different number of threads (figure 2c). Because the work load is balanced among the threads, therefore when we increase the number of threads or processes, the workload for each thread or process will also be divided proportionally to the number of threads or processes, this is why the scalability plots are almost equal to the linear or ideal scalability plots. As for why it is not equal to the linear scalability is because although the work load is quite balanced, but there are still parts of the computation that cannot be parallelize, for example, when waiting for work load

and there are also some pixels that took longer time to compute (so the time is not same for each threads or processes).

Hw2a Analysis:
Next, the vectorization technique is very effective in increasing the performance, as we can see from the figure 3a above. The vectorization version of pthreads is much more effective compared to the non-vectorization version, as they perform the computation half the time compared to non-vectorization version. I do not do the non-vectorization version of hybrid optimization as it is quite obvious from the pthreads optimization that vectorization is definitely helpful in reducing the computation time.

Besides, conducting experiment on scalability and load balancing, I also try some other experiments for the parameter in the program. In the pthreads version, I tried to divide the workload by row, height or row but with limited column sizes. The result is shown on the figure 3b and figure 3c, at last, I chose to divide the workload by row, although the result divided by column size of 20 is a little bit better than other column sizes on the selected test case, but the result in the scoreboard is not as good as divided by row. I think this is caused by the column size is too small, so that the threads or processes have to ask for new workload over and over again, but I have also tried to add the column size, as when we divided by row, the column size is equal to width, so I thought that maybe it will get better if I increase the column size, but from figure 3b, this is not the case. Therefore, I think it is because that I have done a couple of additional computation both when acquiring the lock for workload and after gaining the lock, compared to when I divided the workload based on row.

In addition, I also tried to divide the workload by multiple size of row, so each thread will get more than one row at a time, the result is also not as expected.

Hw2b Analysis:

Based on the result from figure 4a and 4b, I chose to use dynamic scheduler for the parallel sections in computing the longest part. The longest part of my program is to compute the pixels shown using the color red below. (Note: the green and blue part are done separately using dynamic scheduler)

0	0	0
0	0	0
0	0	0

Next, in hw2b, I also tried to divide the workload similar to the figure below. (2 processes, 0 means rank 0 process, 1 means rank 1 process), each box represent 1 pixels, same color means that they are the workload of the same process).

0	0	0	0
0	0	0	0
1	1	1	1
1	1	1	1

After trying to divide the workload similar to the figure above, I realised that since the mandelbrot set have some locality properties in them, and since I am not using dynamic load balancing, the method above is not a good idea, as the some of the process may got a lighter workload, compared to the others, and indeed the result from the method above is slower even compared to the pthread version in the scoreboard.

From the experience above, I changed the load balancing method to still dividing the workload based on row, but not sequentially, it is similar to round robin as shown below.

0	0	0	0
1	1	1	1
0	0	0	0
1	1	1	1

Using the method above, we can avoid the locality properties much better compared to the previous method. At first, I thought that although the method above should be better compared to the previous one, but since I am not using dynamic load balancing method, I expect that the workload might not evenly balanced, but from the figure 2b and 2c, it seems that it is quite evenly balanced.

For omp for scheduler, in the end, I chose to use dynamic scheduler with chunk size of 5, because although in figure 4b, it only a little bit better compared to the other chunk size, but the result in the scoreboard is pretty good, compared to the others.

For hw2b, I have not tried of using MPI to manage a work pool, I think that it is a good idea to have one of the master process's thread to manage the work pool and the rest of the threads doing the computation, but maybe the communication time will be bottleneck.

3. Experience and Conclusion

   From this homework, I learned a lot of new knowledge especially in how to vectorize a program, and I actually changed the program multiple time using different SSE2 intrinsics. The same program using different intrinsic or different calculation method, can result in very different runtime performance, for example, to compare the length to constant double value 4, since this is done many times in the program, I thought that it is not a good idea to store back the length value for each pixel back to the memory every time I need to compare them, so I used the comparison intrinsics from SSE2 (__mm_comigt_sd), as I thought that it is better to do the comparison inside the register itself. However, it turns out that I was wrong, there are many things to be done just to compare a length value, for example, to compare the value of the length register

[127:64], I cannot find intrinsic that can directly take the value from that range, so I used the shuffle intrinsic to do that and it is not so effective if I have to do that for every iteration, and I also tried to store the value back to memory, then compared them. At last, I found out that we can use union to access the data inside the register, this way, I do not need to store back the data to memory every time I need to use them. As expected, it became faster after I used union for accessing the data, also I have read before that if the memory is aligned, then it will also be faster for vectorization.

Another thing is I am not sure if it is a coincidence, but I think it is faster to use load intrinsics to load data from memory to register, compared of using set intrinsics.

In addition, I also learned to consider the number of iteration of every loop in the program, because even 1 extra iteration for every pixels will cost much in the end, so I tried to change some of the calculation order inside the loop and have early break, the result is much better compared to the shorter code but have extra computation.

For the hybrid version, at first, I tried to modify the code of pthread directly and change pthread lock to omp lock, but the result is slower than the pythread version, and since I can directly parallize for loop, I changed to using for loop for hw2b and collapse 2 loop directly using omp, the result is much better compared to before. In conclusion, I think it is better in my program to let the omp decide the workload for each thread than I manually parallel each sections of the code.

For the difficulties that I have encountered in this assignment, they mostly came from vectorized the program and also it is not easy to debug the code. I think there are so few examples related to the vectorization intrinsics.