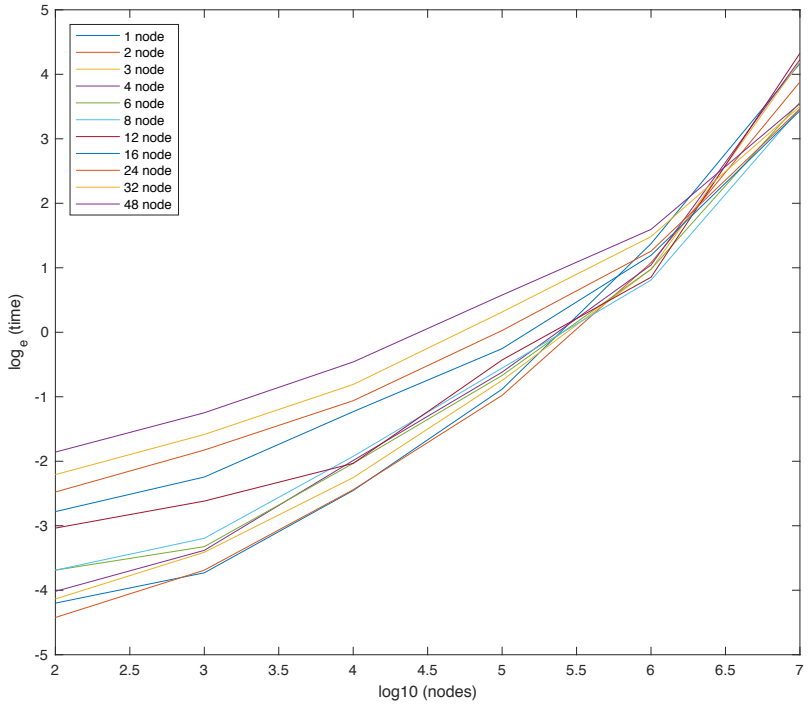
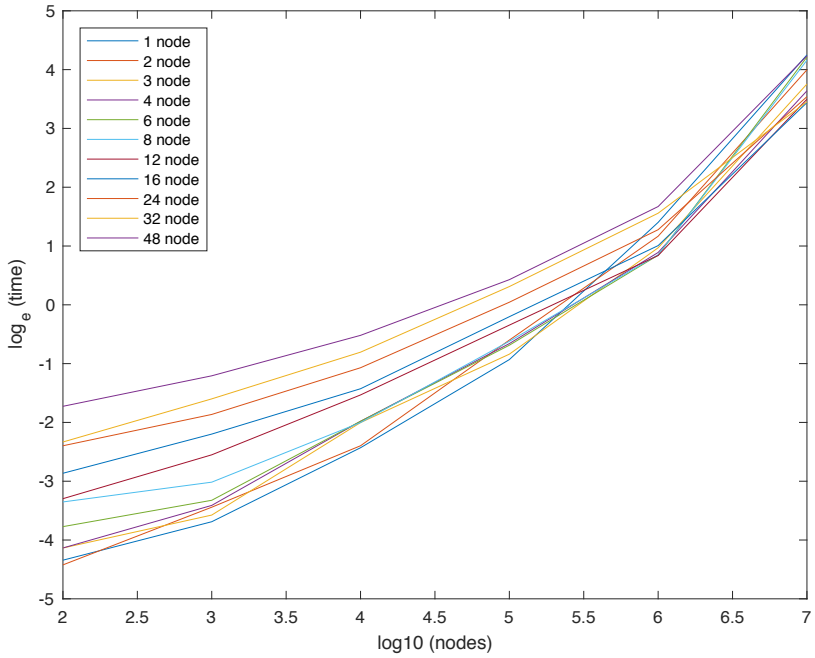
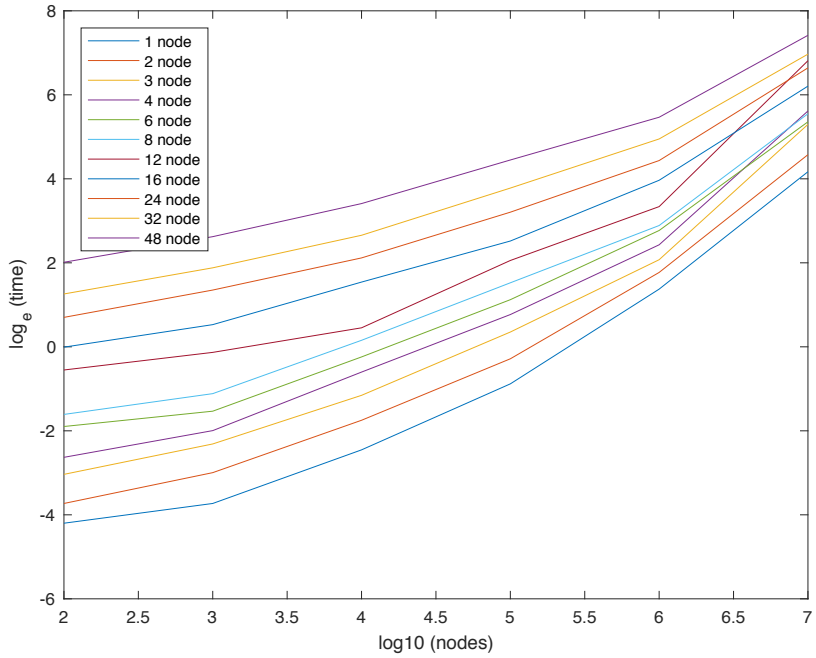
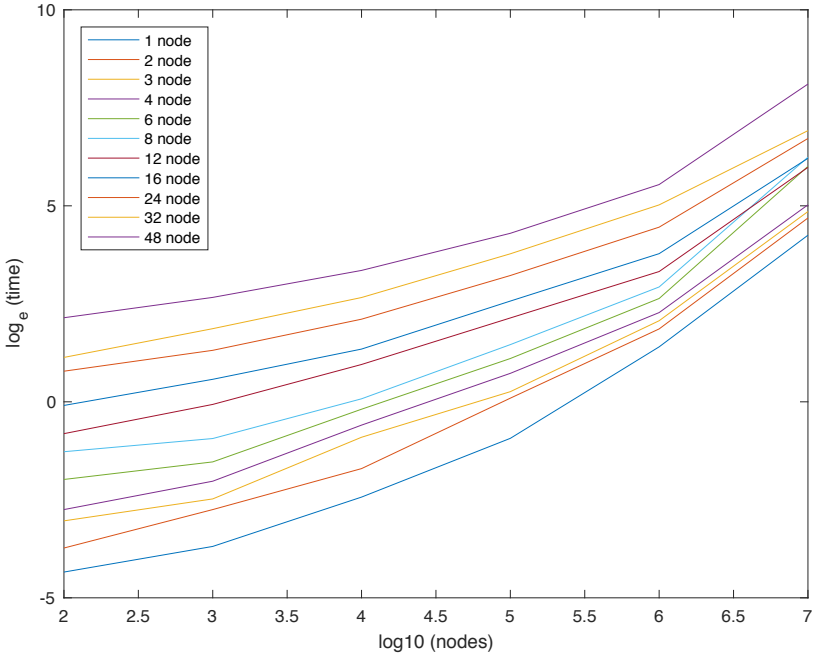
Below are the results that I ran my program on node2x18a. The number of threads I chose are 1, 2, 3, 4, 6, 8, 12, 16, 24, 32, and 48. The testing number of vertex are from 100 to 10,000,000 incremented exponentially by 10. Note that, the run time is plotted by loge and the node size is plotted by log10.





The next two figures are the plots that the accumulative runtime with node numbers. Here, I define the accumulative time = runtime * numOfNode. And I plotted the same results that corresponds to the two plots above.



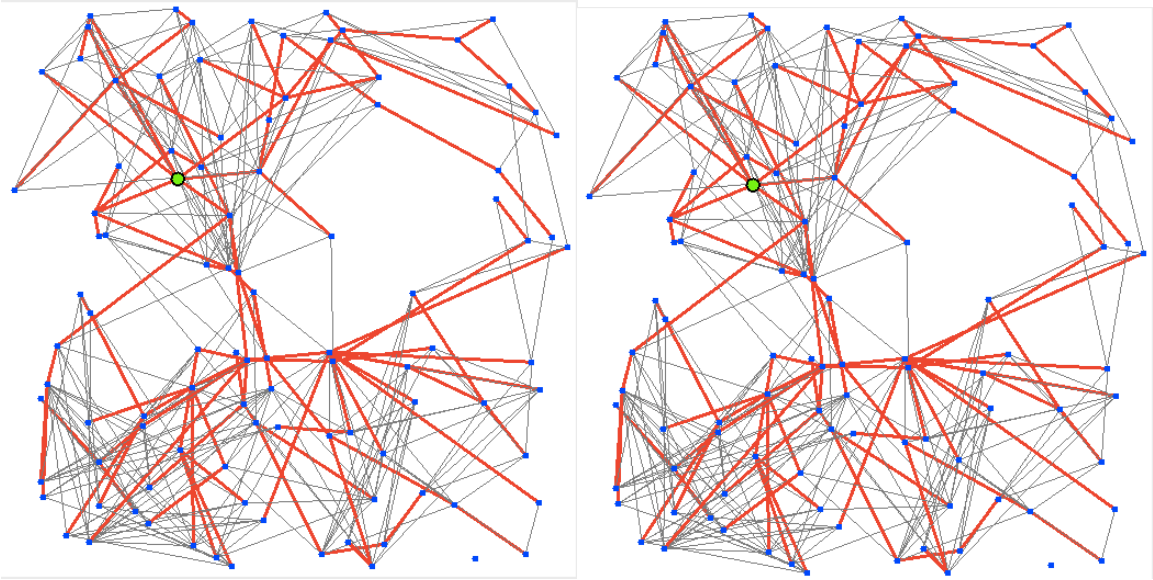


Conclusion:

1. Although, multi-thread is able to speed up by using right amount of threads (in my case, the number of threads between 6-16 are the best options), the speed up is not proportional to the number of thread. The best case, I can speed up my program by 2, comparing single-thread's result and using delta-stepping algorithm.
2. The main bottlenecks can come from these several places. First, the most of time might spend on waiting other threads finish their jobs. Since, in this HW, we used a lots of barriers, this will cause drag the total runtime. Second, the distribution of vertexes handled by each thread. The worst case scenario, we split all the vertexes into the threads that are being solved sequentially, then, not only we don't have any parallel part and we have to pay the penalties for communications between different threads. Third, since I used a single array of buckets, when, different threads try to access to it, this can be a bottleneck with single barrier that all the threads need to wait.
3. The best result I can get from multi-threads are 6-8 threads, and the speed up is about 2 times faster than single thread runtime.

Some Debugging plots (although, it is not the best way to debug)

1. 100 nodes and with 1 or 100 threads



2. 1000 nodes and with 1 or 100 threads