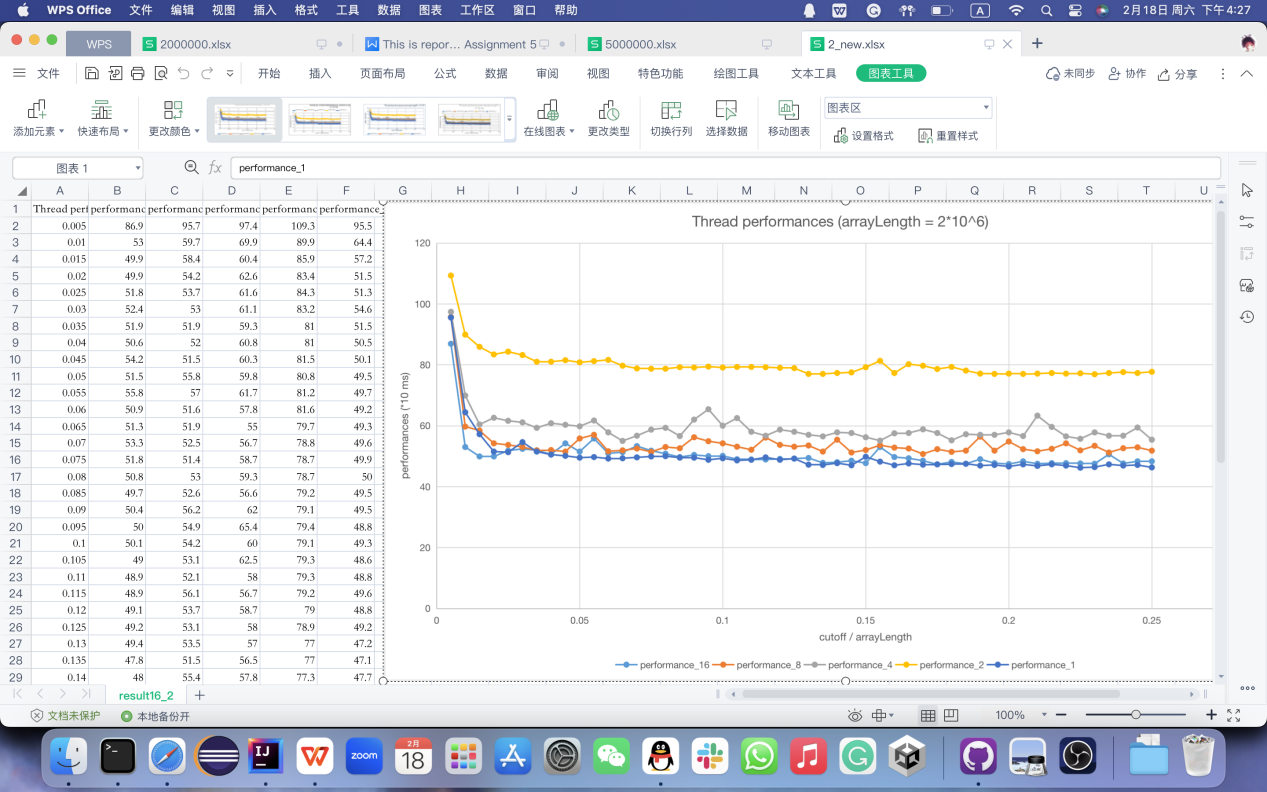
This is the report for INFO6205 Assignment 5.

NUid: 002773368

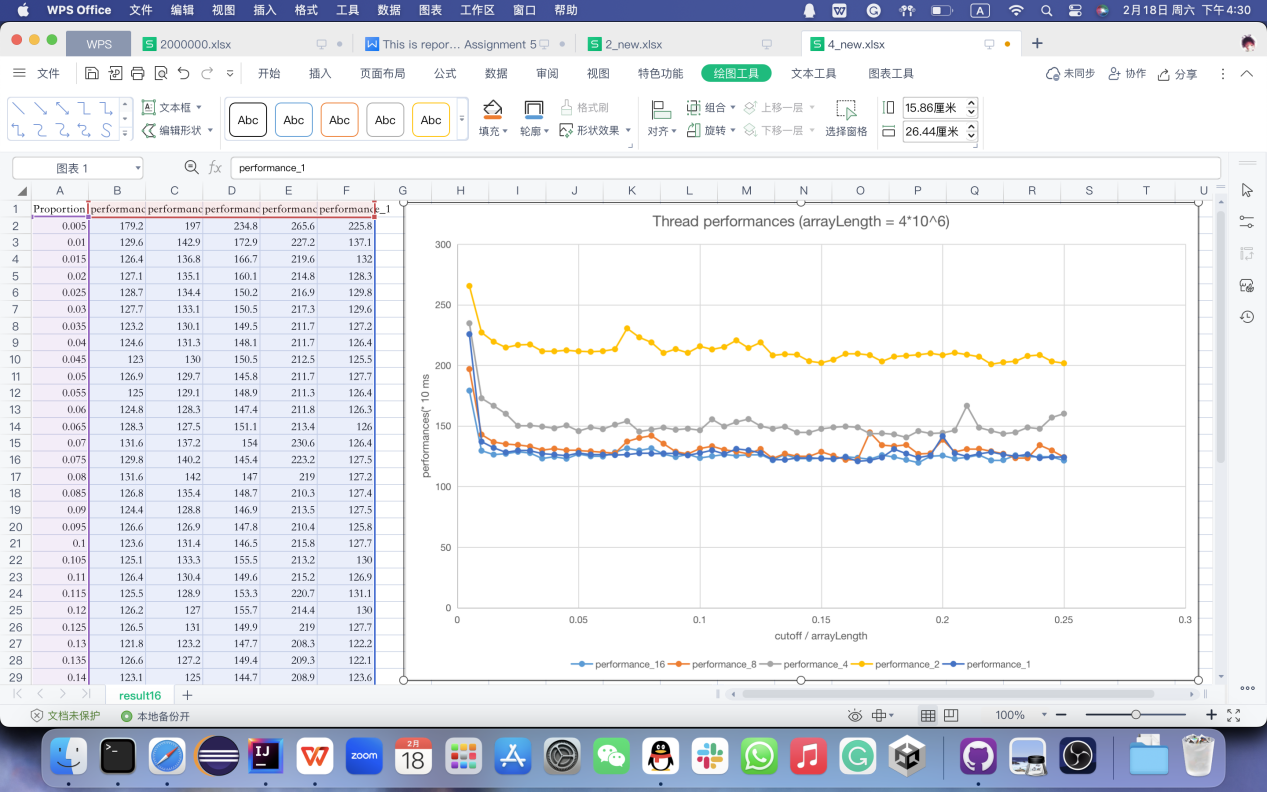
Name: Cheng Yuxuan

In this report, we will introduce how we made changes to the original file. Then we will discuss what effect proportion of (cutoff / arrayLength)’s change has on performance of sorting. Then we will discuss the effect of different number of threads.

We have two groups of results as followed:



(1)



(2)

The long runtime at first tend to be the “warmup” time for compiler - it takes some time to have processor deal with large amount of data, so the run time can be considerable long.

1. Effect of number of threads.

This is done by setting the running configuration. The running of Main has parameter args and is passed to set the number of threads using System.*setProperty*("java.util.concurrent.ForkJoinPool.common.parallelism", y);

(at first we try to define a new ForkJoinPool method but found it no difference happened - By reading the test printout

System.*out*.println("# threads: "ForkJoinPool.*commonPool*().getRunningThreadCount());

we found the actual running method limitation remained the default value: 7.)

After modifying this we made experiments on 2 different arrayLength and 5 different thread parallel limitations (that is to say, value in [1,2,4,8,16]), we made 10 experiments to find the most effective number of threads.

The results show that the thread parallel limitation numbers remain the same rank on length of either 2\*10^6 or 4\*10^6:

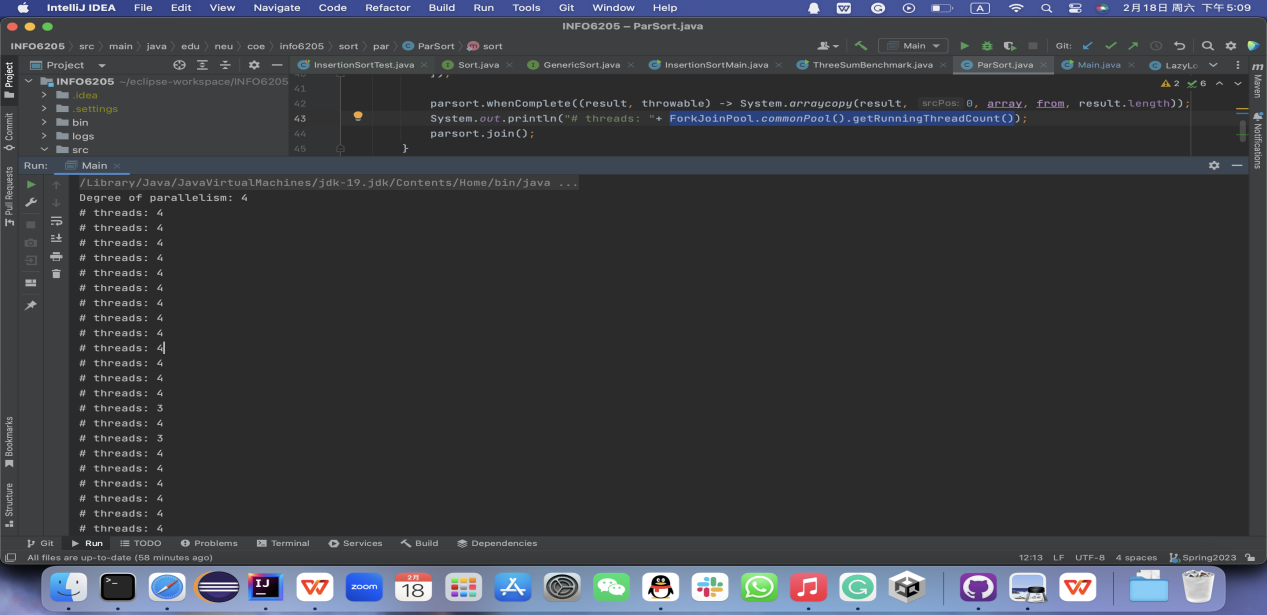
The 2-thread run time is the longest. The runtime is 70% more than any other thread numbers.

The 4-thread run time is the second longest, nearly 20 % more than the remaining 3 thread’s runtime.

The 8-thread, 16-thread, and 1-thread have the best performance, their performances are similar.

This is not intuitive because typically we think larger thread numbers results in better performances. The reason could be the threads are not perfectly used when the number grows - by calling ForkJoinPool.*commonPool*().getRunningThreadCount(), we have most threads do not necessaryly work while running the program. In this way, the multi-thread task cannot make full use of all threads, so the performance was bad.

For example, when the thread is 4, not all threads work together during running process, as followed:



The situation is even worse in the 2-thread case.

The 1-thread case requires extra explanation. During running period, the working thread remains 0 always. We guess it is an automatic allocation of JVM that assigns work to threads out of the common pool.

1. During the running experiments, 2 different arrayLength and 5 different thread parallel limitation are tested, while the value cutoff / arrayLength remains from 0.005 to 0.25.

To have a different arrayLength we have to change Main.java structure. We define new constants named “” and “” to replace corresponding values so that the proportion of cutoff / arrayLength remains the same, and it is easier to make changes.

The result is, despite the experimental error, performances of all numbers of threads remain high when the cutoff value is quite small. when the value grows to around 0.035 the runtime drops to a comparably stable stage; this runtime remains its value when cutoff/aarrayLength grows to 0.25.