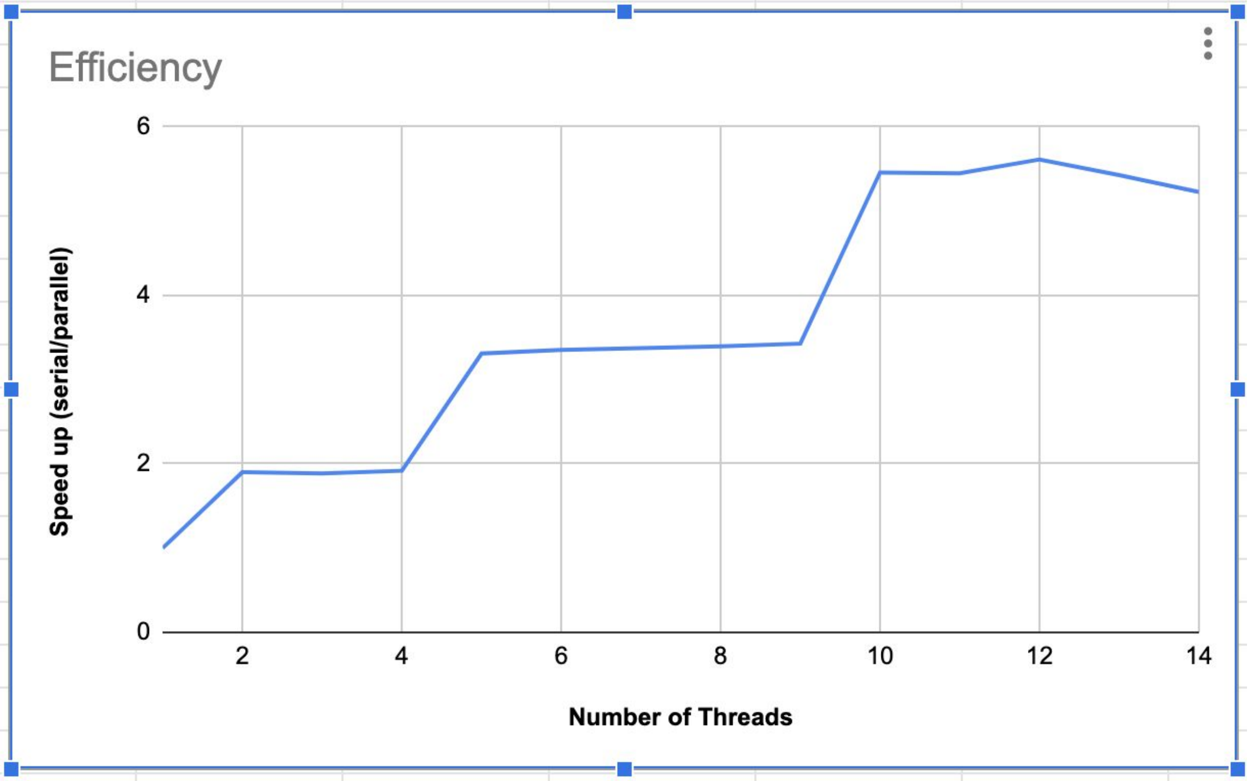
Joel Koehler
CS453
Lab 3 writeup

Efficiency

The efficiency for my algorithm is shown in the efficiency graph below. My peak speed up was 5.61 at 12 threads. This was confusing to me because a video for this module mentioned that the peak would likely be at 8 threads on onyx. I ran this on onyx, which according to the “cpu cores” section of /proc/cpuinfo, has 12 cores. My initial inclination was that making one thread per core would yield the best results and that turned out to be true. However, as I looked into it more my expectation changed. To quote an answer on stackoverflow: “If your threads don't do I/O, synchronization, etc., and there's nothing else running, 1 thread per core will get you the best performance. However that is very likely not the case. Adding more threads usually helps, but after some point, they cause some performance degradation.” Because of this I assumed that the optimal thread count would be just under 12 due to thread overhead, but strangely this was not the case. The speedups are not as linear as they are “stair-stepped”. Thread 4 to thread 5 shows a step up, and thread 9 to thread 10 shows another.

NOTE: I ran more than just 8 cores because I noticed that the speedup was more significant (and in fact peaked) in the low-teens. I wanted to show that in my graph, I hope that is okay :)

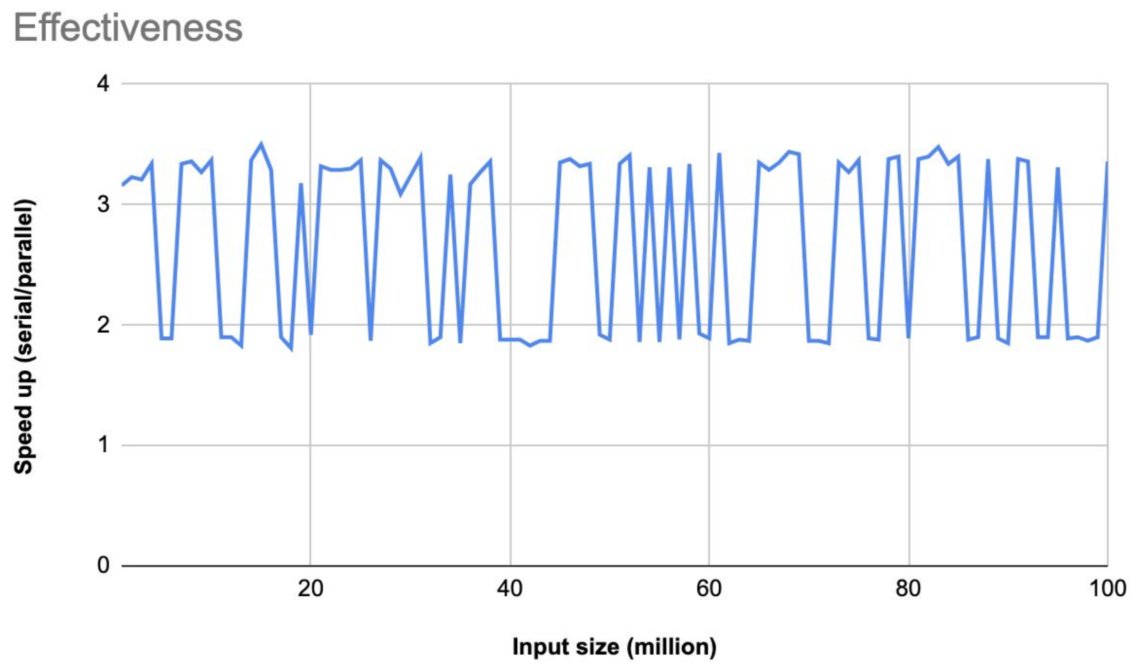


Effectiveness

The effectiveness of my implementation is shown below in the “Effectiveness” graph. I slightly misunderstood the instructions when generating my data -- I only generated the speedup ratio instead of printing out the exact times. It took a *long* time to run 100*2 test cases with the input size in the millions (all evening), so I didn’t have time to regenerate the data. I hope that is okay! I believe that all the same information is displayed save for the ballpark time my program ran in for a given input size, so I have included my results for 1M, 10M, and 100M with the actual times generated just below the graph.

Now onto the analysis of the effectiveness data; it’s very strange. I noticed as I tested my solution that the parallel speeds would get about 70% worse almost every other time I would run identical tests. I didn’t notice this at first using small input sizes, but as I increased the input the ratios became more “consistently inconsistent”. As can be seen in the graph, average speedups are either ~3.3 or ~1.9 with really no in-between. This has me very confused. At first I thought it was a problem with critical sections, but my lists *always* get sorted correctly (messing with the critical section breaks this). Then I thought it was the random list, so I ran my tests again (this second run in the data from the graph) with a seed of 30 each time and the inconsistency is still there.

I didn’t use locks explicitly, but instead used the implicit lock of pthread_join to capture my critical section (the start routine and recursive call). This solution works perfectly in terms of sorting correctness, but perhaps this is why the run time ratio is inconsistent? I’m not sure. Now I understand that this time inconsistency issue is not preferable, but since the parallel solution correctly sorts the list every time and is <u>always</u> faster than the serial implementation, I would say this is still a success. Moreover, when I run this test with 12 threads instead of 5 I get ranges of ~5.40 and ~3.15, so the speedup is “greater than 2 with 4 or more (12 in this case) cores”.



```
Running mytests with a size of 1000000 using 5 thread(s):
Sorting 1000000 elements in *serial* took 0.30 seconds.
Sorting 1000000 elements in *parallel* took 0.10 seconds.

Running mytests with a size of 10000000 using 5 thread(s):
Sorting 10000000 elements in *serial* took 3.48 seconds.
Sorting 10000000 elements in *parallel* took 1.07 seconds.

Running mytests with a size of 100000000 using 5 thread(s):
Sorting 100000000 elements in *serial* took 39.83 seconds.
Sorting 100000000 elements in *parallel* took 11.70 seconds.
```