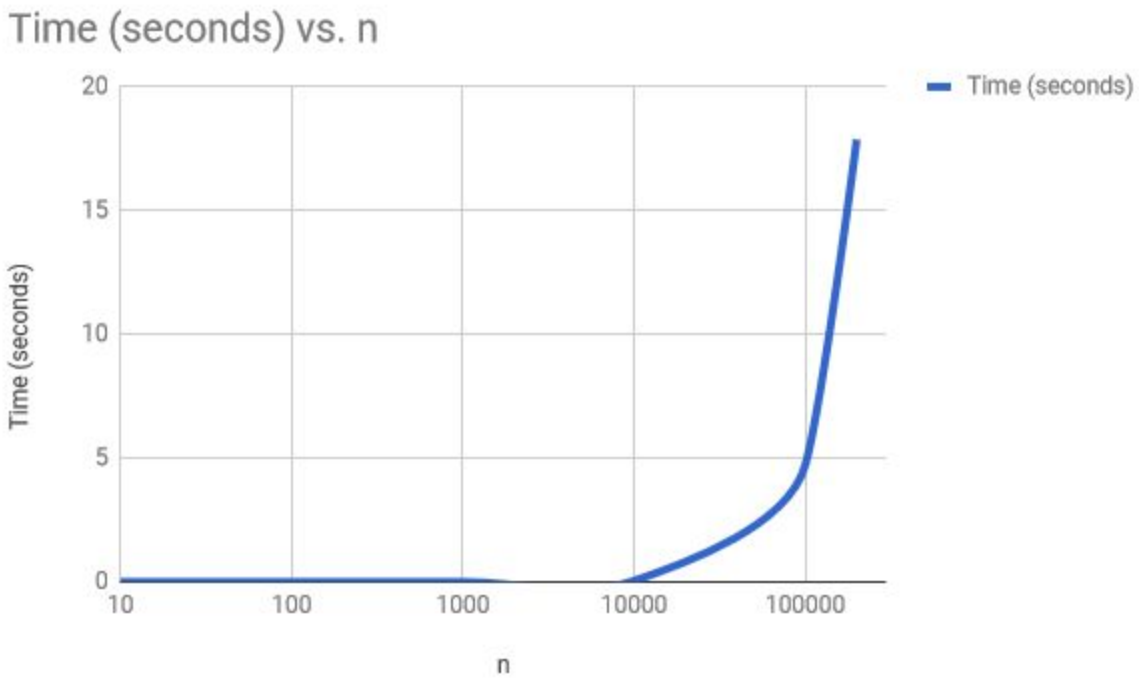
Serial Quicksort:

n	10	100	1,000	10,000	100,000	200,000	300,000
Time (seconds)	0.000001	0.000009	0.000473	0.044574	4.703213	17.863690	40.306.187



I implemented the recursive quicksort with the last element as pivot as it is the simplest to implement. For serial quicksort, we notice that the time in seconds increases on an almost O(n^2) scale.

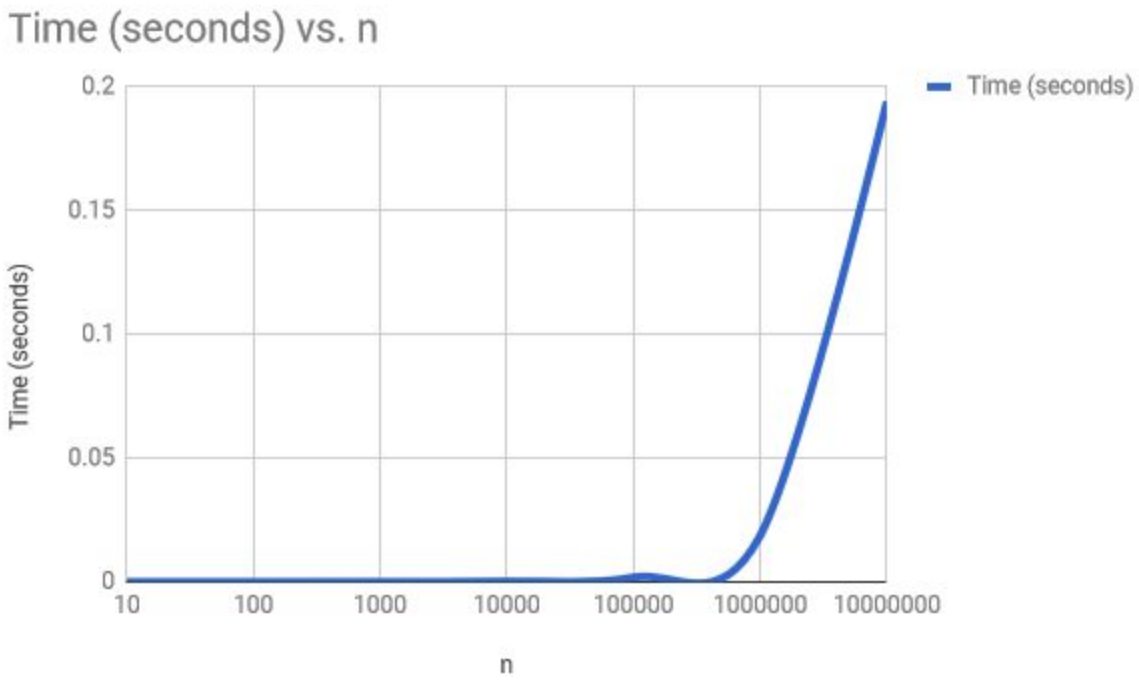
Parallel Quicksort:

N-> P|	10	100	1000	10,000	100,000	200,000
1	0.000018	0.000041	0.000674	0.045997	4.47532	17.913703
4	0.000222	0.001973	0.006075	0.093502	5.438144	21.090194
16	0.000484	0.000700	0.004861	0.109262	5.440616	20.244478
32	0.000877	0.001281	0.007231	0.108558	5.353398	20.069575
64	0.001595	0.002017	0.007867	0.106609	5.430755	20.025339
1000 threads	0.003826	0.002961	0.031286	0.119741	5.270776	17.753798

The parallel quicksort posed some problems in implementation. When trying to find areas in the code to parallelize, I immediately discarded the partition function as it is extremely hard to properly parallelize with openMP. I focused on parallelizing the main quicksort function and I found two ways. The first is using the pragma omp sections call. The sections call basically divides the work onto threads. We surround each recursive call with a section so that it can be performed by a single thread. This makes sense. However, when testing for values larger than 110,000 , I was getting an internal segfault specific to openMP. I did some research as it was very weird what was happening since the bug is not related to human-error. I concluded that too many threads are being called and the problem probably emanates from the thread stack size, which is not enough for the program. I found another way to parallelize the code using tasks (pragma omp task). The main difference between tasks and sections is that a task does not wait for others threads to finish. For some reason, I was able to test for any value with tasks and so I chose it as my main implementation. My old implementation is commented out and so it can still be checked out if needed. As I increase the number of threads, I did not see an improvement in performance, which was weird. I expected performance to drastically increase as all the work is being divided. I also found a research paper online that states visible improvements when using openMP to parallelize quicksort. The worse performance is also logical however. As we increase the number of threads through recursive calls, we get a giant number of threads that need to be tracked. And so, the number of context switches becomes incredibly large thus the overhead.

Serial List Ranking:

n	10	100	1,000	10,000	100,000	1,000,000	10,000,000
Time (seconds)	0.000001	0.000002	0.000017	0.000188	0.001820	0.018683	0.194036



To implement the serial list ranking, I followed exactly the pseudocode in the slides. However, I encountered an endless loop that was hard to diagnose. To fix it, I included an if statement that eliminates the possibility of there being a Q[i] value equal Q[Q[i]] as in that case, the loop does not end. The performance for the algorithm is pretty good and as expected. The time increases based on the size of the array.

Parallel List Ranking:

N-> P|	10	100	1000	10,000	100,000	1,000,000	10,000,000
1	0.000013	0.000013	0.000018	0.000073	0.000633	0.006331	0.066600
4	0.000106	0.000111	0.000106	0.000147	0.000298	0.002737	0.018467
16	0.0.000489	0.000445	0.000483	0.000962	0.001289	0.002493	0.015109
32	0.001031	0.001089	0.000881	0.001072	0.001593	0.003082	0.015299
64	0.0.001759	0.001769	0.001731	0.001786	0.002168	0.003507	0.015349

For the parallel list ranking algorithm, I parallelized the three loops that I had in the serial version and it produced immense performance improvements. However, as we saw in the last lab, performance reaches a maximum then starts decreasing, as the threads don't bring any additional benefit anymore and overhead is increased.

In conclusion to the lab, we notice that parallel code on average increases total performance. However, there are some important considerations to take as to what code to parallelize. Some algorithms are very optimal for parallelizations while others are not.