Ant Colony System Parallelized

Nathan Klisch

December 13, 2020

1 Introduction

The Ant Colony System (ACS) was developed by Marco Dorigo and Luca Maria Gambardella as an improvement over the Ant System that was also created by Marco Dorigo. The Ant System was inspired by how ants' use pheromones to help the colony find the shortest path to food and other resources.

The ACS is used to solve combinatorial problems, like Traveling Salesman Problem(TSP) or Knapsack Problem. Good, near optimal solutions to large Traveling salesman problems are incredibly useful in a variety of fields from Genome Sequencing to minimizing fuel use of spacecraft. Exact algorithm's to solve a Traveling Salesman Problem are considered NP-hard - meaning currently there is no-known algorithm that can solve them in polynomial time. This is an issue because as the problem size grows the computation time grows exponentially, making large problems currently impossible to solve using an exact algorithm.

The Ant Colony System is a means to learn a near optimal solution more quickly. By simulating ant's moving around the graph, updating the pheromones as they travel from node to node and updating the global pheromone levels based on the best ant's tour, the ant's learn to find better and better solutions. In the paper Dorigo showed that ACS is not only effective but it is relatively quick in comparison to other methods of the time. Dorigo also found that combining ACS with a 3-opt local optimization technique before calculating the global best Ant's tour make's ACS come close to finding close to the global minimum consistently even on larger problems. It also sped up the computation time as 3-opt helped improve the placement of the pheromones thus taking fewer iterations to find a good solution.

The goal of this project was to determine what kind of performance improvement we can achieve by applying OpenMP multi-threading to help solve this problem more quickly. By improve the speed of this algrithm it would be possible to find good solutions to large Traveling Salesman problems.

1.1 Approach

To create a benchmark, gain understanding of the algorithm and have a baseline set of code to build from, I implemented the original Ant Colony System in C++ it's original form with no optimizations or reductions based off the original paper's provided pseudo-code of the algorithm. After I successfully created the base line code, I measured the performance of the algorithm. The computation time grew very quickly exceeding a full minute of computation time for $N > 800$.

I then created a version that used Candidate List. A Candidate List is a two dimension data structure where the rows are associated with a particular node of the graph and the columns in order start with the identity of the closest node to the that row's node. The candidate list is typically bounded by some constant M so that in the algorithms that use it they typically only look at the first M nodes in a particular node's candidate list. This changes the time complexity of algorithms that use it from $O(n^2), O(n^3)$ to $\approx O(n)$. This data structure is used in both the Ant Colony System and in the 3 Opt local optimization algorithm to significantly speed up computation.

After completing the Candidate List implementation, I proceeded to implement the 3Opt algorithm to improve the results of ACS to be closer to optimal. While this slows down computation, the improvement in nearing optimal tours is worth the time increase. I implemented 3Opt using don't look bits and candidate lists which significantly improved it's computational time complexity.

1.2 Challenges

When I started to parallelize the Candidate List algorithm, I started by just parallelizing the Ant's traversal of the graph and generation of tours. I saw little to no performance gain this fist #pragma omp for. It turned out the ant's generation of the tours isn't as a significant amount of the computation as I might have expected. I determined that the global update of the pheromones, selected from the best performing ant's tour, was my bottleneck. After I parallelized this loop I started seeing much better performance, though still not to the degree I had hoped.

In addition, I didn't have time to implement 3Opt in CUDA, but after analysis of the sequential nature of certain parts of the algorithm and where much of the work lies, I am not sure how much doing the optimized form of 3Opt in CUDA would help in this case. The size of the problem would likely have to grow to a degree where the memory actually becomes the constraining factor, before we see the improvements on computation using CUDA, due to the lag of moving memory down to the GPU and back. The nature of the Ant Colony System is the sequential trials of the ant's traversing the graph. Theoretically, more iterations should improve the resulting tours, but this means each trial has to be done in sequence and cannot, by definition, be parallelized, even with clever approaches of data transformations and reductions.

My first implementation of the Candidate List data structure attempted to hold both the node identifies and the distances from to that node from the node's accessing node in the same contiguous memory block, locally stored, to improve memory locality. Unfortunately, when I implemented the 3Opt version of the ACS algorithm it became clear that this structure was not conducive to the accessing patterns that 3Opt required, so I returned to the more traditional two dimensional array with $index : x, y$ being the distance of edge (x, y). I did keep my initial implementation of the Candidate List for the ACS that does not include the 3Opt optimization.

2 Algorithm Descriptions

2.1 Naive Ant Colony System

The basic naive version Ant Colony system involves a representation of a graph, the distances between points and tracking the pheromones placed on any particular edge. My algorithm only works for fully connected symmetrical graphs as I didn't have time to implement the broader algorithm for asymmetrical partially connected graphs. We start by applying the nearest neighbor algorithm to jump start our pheromones. We use the nearest neighbor solution to calculate the starting pheromones on any given edge. Then we send off ten ants to travel around the graph. These ants use a calculation based on the pheromones on an edge and the distance to a node to decide where they will travel to next. When they travel along an edge, they eat some of the pheromone, reducing it's presence. This lends itself to more exploration. The downside is that while we can parallelize each of the ant's traversal of the graph, we must make this local update a critical section, so only one ant is updating an edge at a time. In my experiments though, this critical section had little to no impact on speed up. After each ant has traversed the graph creating a complete tour, we select the ant that found the shortest tour and we update the pheromones with the edges in that tour. This way we tell the ants in future iterations that this path is one of the better paths.

2.2 Candidate List

One of the major optimizations suggested by Marco Dorigo is using a candidate list for when the ant is selecting it's next move. By limiting the number of choice down to a constant number of the closest nodes to any particular node, we can fix the growth of the calculation. The ACS only selects node outside the candidate list if at a node, there exists no unvisited nodes in their candidate list. When this occurs, we go beyond to the longer complete candidate list, where all the nodes are ordered by distance.

2.3 3Opt

3 Opt is a local optimization/search algorithm that takes complete tours generated by some other algorithm and optimizes them. It does this by selecting 3 edges, with 6 unique nodes. It then tries different ways of re-arranging the edges and tests if it would decrease the overall tour. If it does, we make the needed changes with those edges in the tour and then keep looking until no improvement can be found. One common optimization is the "don't look bit" where if when looking at edges from a node no improvement can be found, we mark it off, so we don't try it again. We only then unmark it if the graph changes in some way that is directly connected to this node. This prevents unneeded searches after we find each improvement and start over, as looking at a node agian if nothing has changed will reveal the same result the next time. The other optimization is using a candidate list. The candidate list limits the edges we can select from to be within the candidate list of a particular node. This does provide a chance of missing an optimization, but it has empirically been shown that this works as well as searching across the whole set of N.

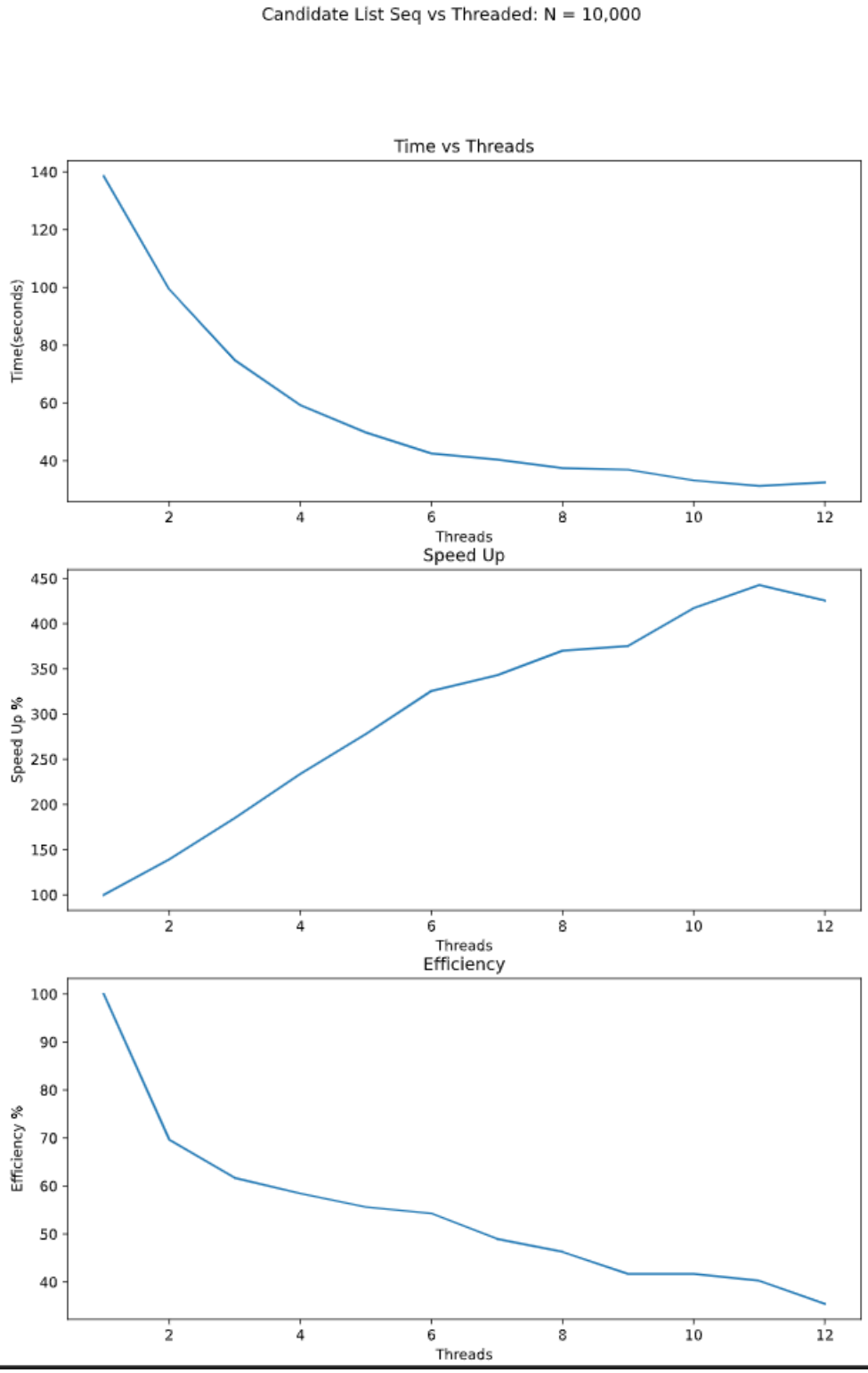
3 Experimental Setup

The experiments will be run on cod and brill which has a Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz. This processor has 6 cores and 2 threads per core for 12 threads. The L1, L2 and L3 cache size is 192 KiB, 1.5 MiB, 15 MiB respectively. I will be running the OpenMP vs MPI implementations on 1-12 threads/processes. There are a few different experiments I wanted to explore. I have collected data on Naive up to $N = 800$, Candidate List up to $N = 10,000$ and 3-Opt + Candidate List up to $N = 10,000$. Candidate List and 3-Opt + CL have been run with threads 1-12. The naive implementation has not been parallelize but I wanted to see how much of an improvement I would see between implementations. I will be exploring both the speed up and efficiency of the different algorithms as well as analyze the big O growth as the problem size increase of each algorithm.

4 Experimental Results

4.1 Sequential vs Threaded

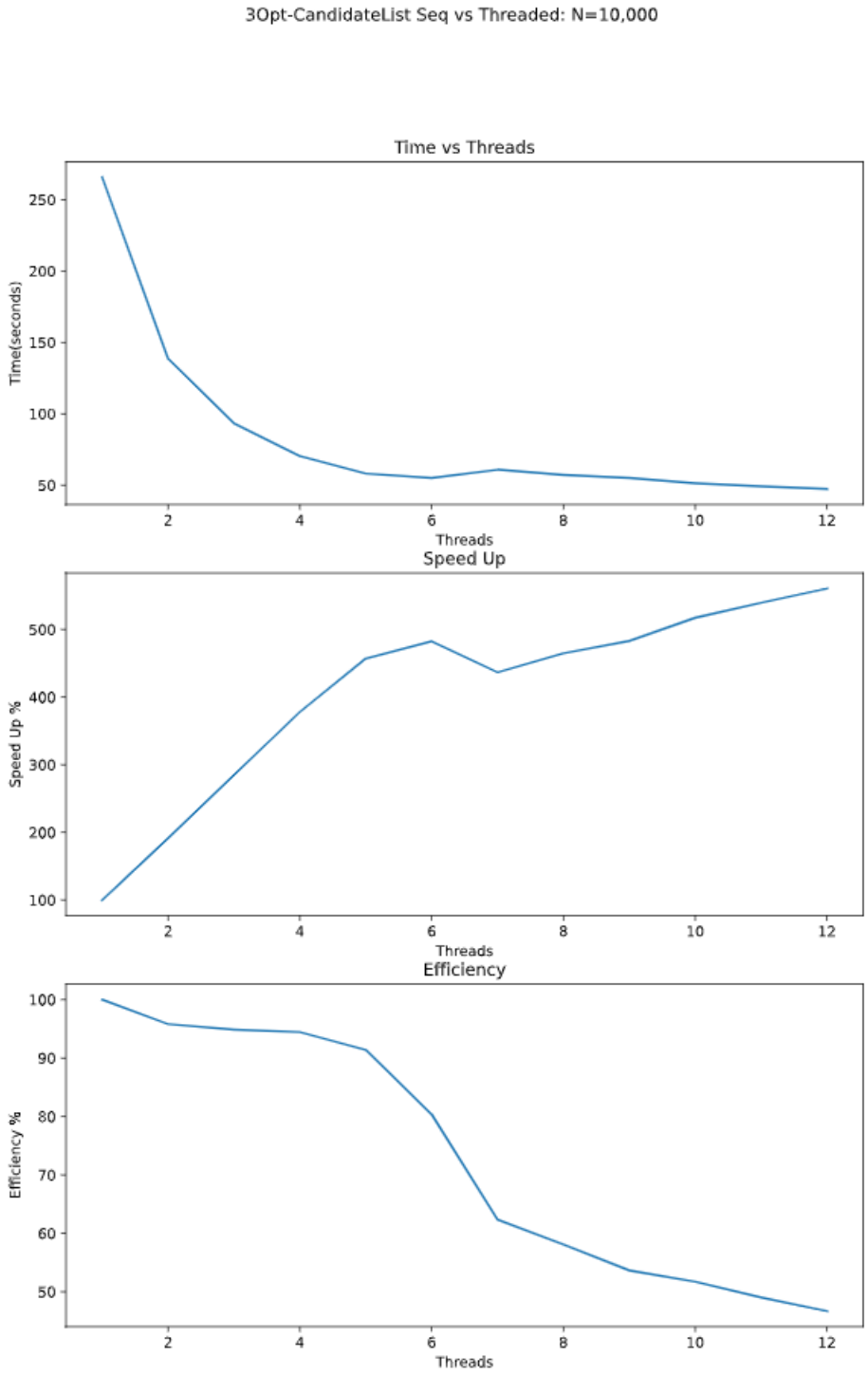
Candidate List, N=10,000			
Threads	Time (Seconds)	Speed Up	Efficency
1	138.505476	100	100
2	99.398439	139.343714	69.671857
3	74.838361	185.072834	61.690945
4	59.252441	233.754885	58.438721
5	49.803296	278.105038	55.621008
6	42.531796	325.651607	54.275268
7	40.382389	342.984852	48.997836
8	37.419772	370.139821	46.267478
9	36.90084	375.345046	41.705005
10	33.197163	417.220825	41.722083
11	31.287441	442.687139	40.244285
12	32.552123	425.488299	35.457358



The above table and graph show the results of the ACS Candidate List only

implementation. We see that the speedup seems to be about +50% speed up per thread, which is about half of optimal speedup. The efficiency mirrors this. This suggested to me that the overhead is high for the amount of improvement parallelizing each ant provides. Adding more work to each ant computation may improve the speedup percentage - which we may see in the 3-opt version.

3Opt + Candidate List, N=10,000			
Threads	Time (Seconds)	Speed Up	Efficency
1	265.561062	100	100
2	138.556337	191.662877	95.831438
3	93.315159	284.585126	94.861709
4	70.285652	377.831115	94.457779
5	58.110312	456.994731	91.398946
6	55.058119	482.328617	80.388103
7	60.82566	436.593801	62.370543
8	57.14762	464.693124	58.086641
9	54.998011	482.855757	53.65064
10	51.330278	517.357541	51.735754
11	49.223631	539.499136	49.045376
12	47.384971	560.433101	46.702758



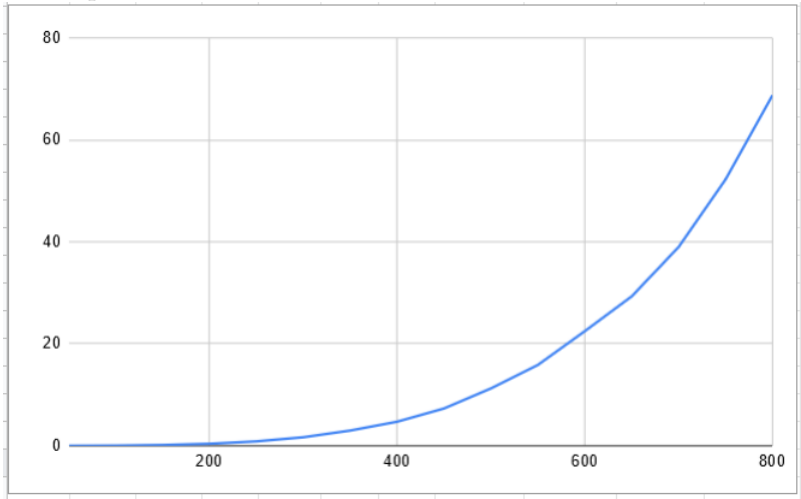
As we can see, the prediction that more work for each ant to accomplish through applying the 3-opt procedure significantly improves our speed up slope for the first 6 physical cores to nearly 100% per added thread. Our speed up

drops off as soon as we run out of physical cores to run on. This suggests that we don't get near the expected performance from multi-threading with this algorithm. This means there isn't much waiting on each other, leading supporting evidence to the fact that the one critical section doesn't impact performance much at all.

Overall, these results are pretty solid, with our speed up looking increase linearly with our physical cores.

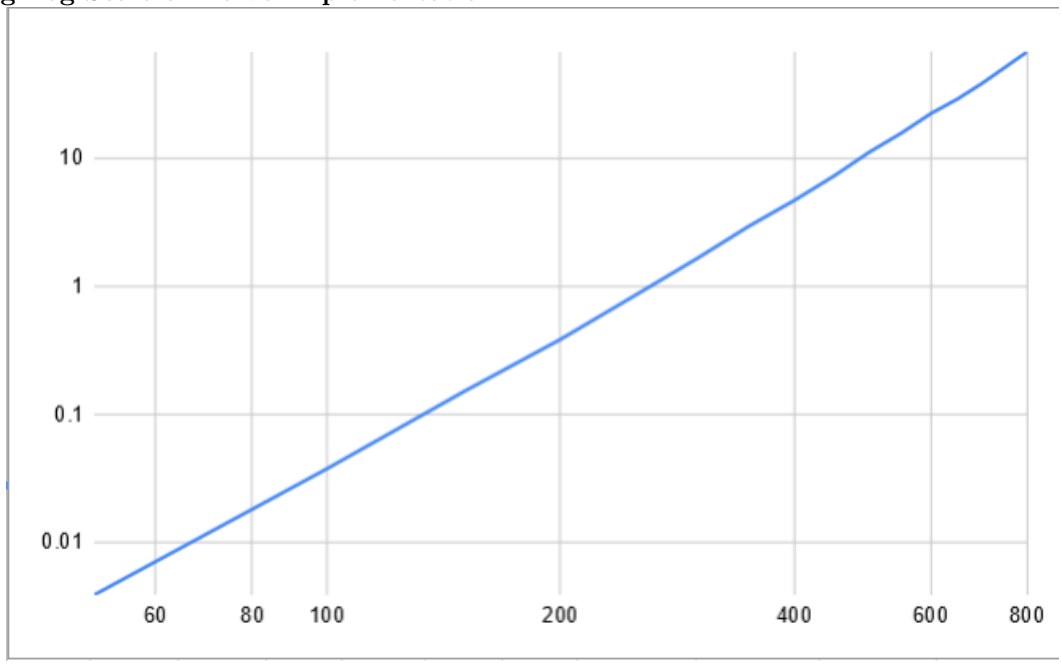
4.2 Problem Size Analysis

Naive Implementation



N	Time(Seconds)
50	0.00393
100	0.03819
150	0.152317
200	0.390106
250	0.865286
300	1.669368
350	2.984692
400	4.720573
450	7.319432
500	11.255898
550	15.861652
600	22.484951
650	29.314687
700	38.995469
750	52.24763
800	68.751384

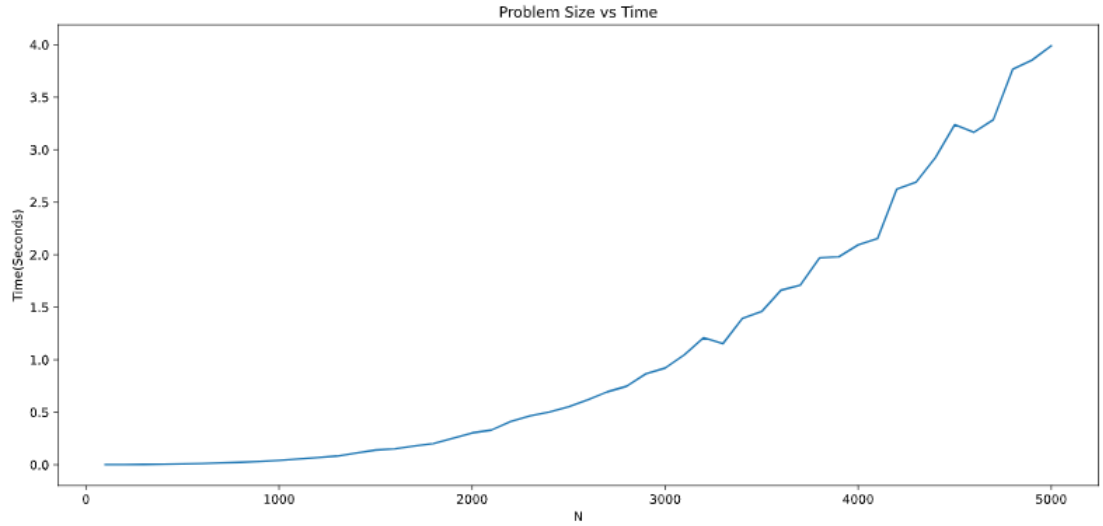
Log Log Scale of Naive Implementation

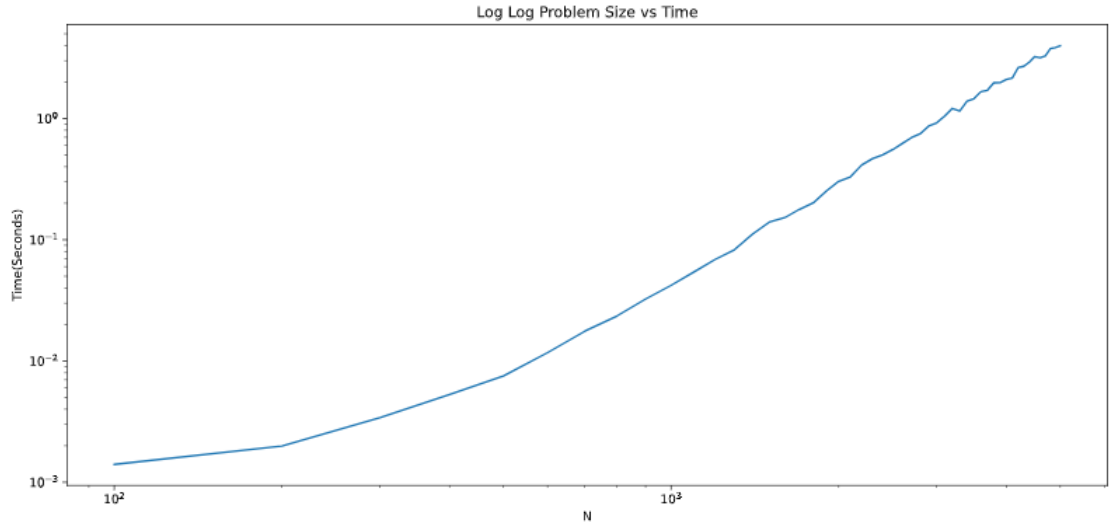


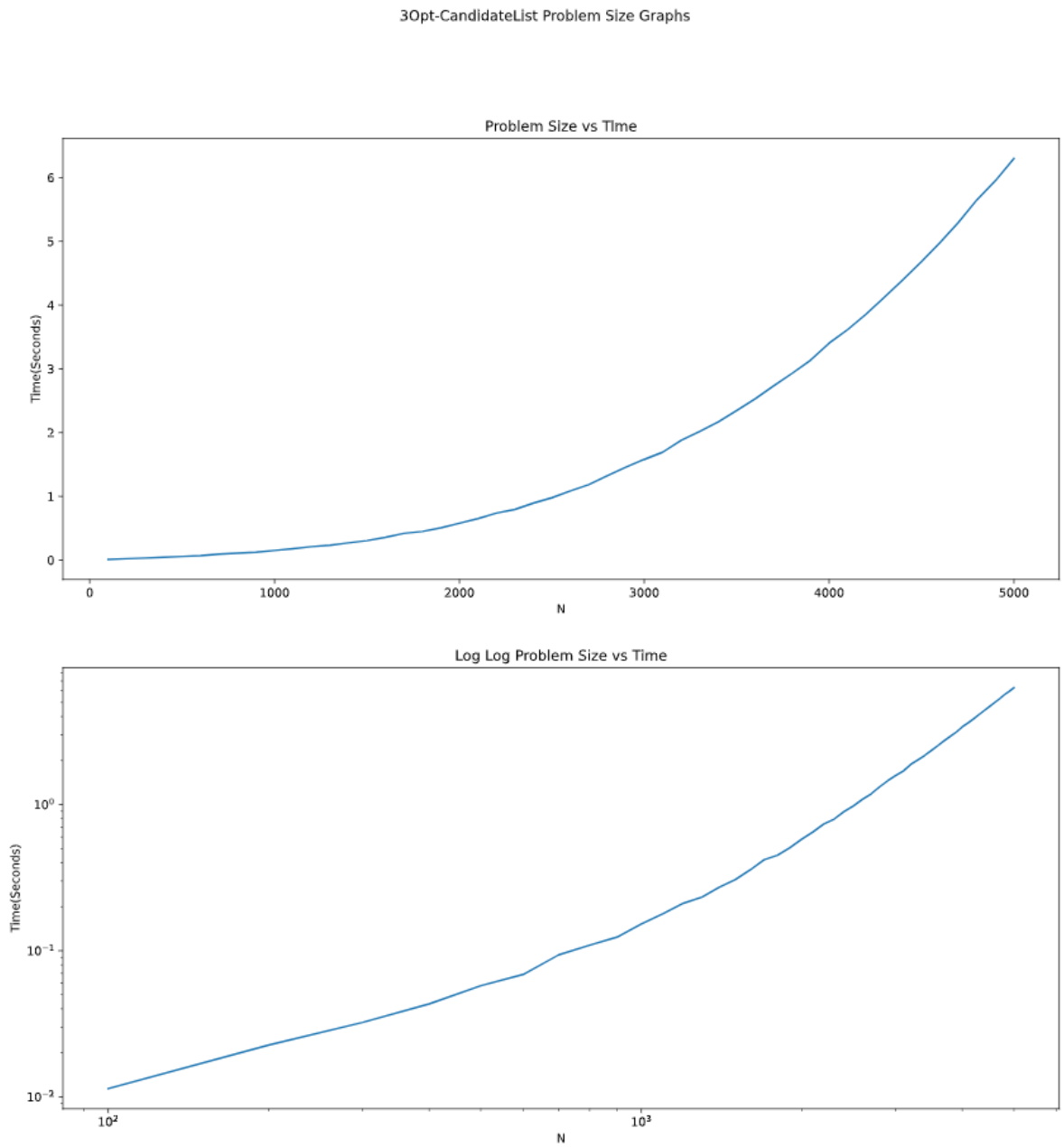
This is the problem size graph for the Naive implementation. We can see

that it is a nice smooth curve and when we place it on a log log scale we get a straight line. This means that it is a polynomial time complexity. Through analysis of the algorithm, the time complexity is $O(N^2)$.

Candidate List Problem Size Graphs





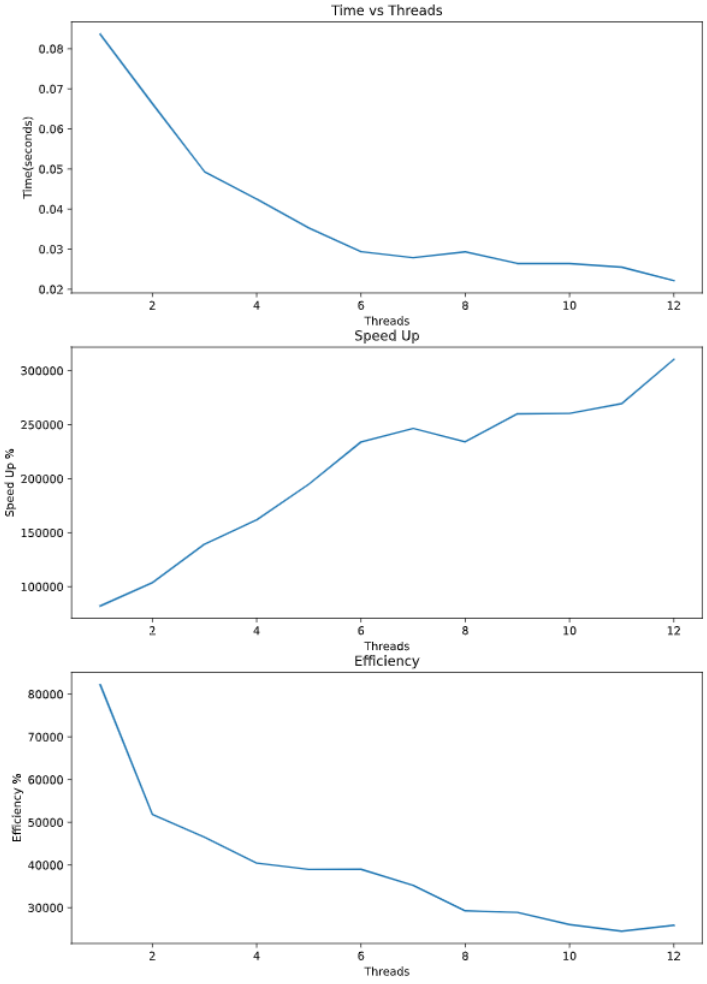


These graph is interesting to me. It looks similar, if jagged, to the naive, but with a shallower slope. When we place it on a log log scale we come close to a straight line but not quite. I am not sure what the time complexity is for this. It seems like it may still be polynomial, but I wonder if it is an Order between 1 and 2.

4.3 Naive vs Threaded Candidate List

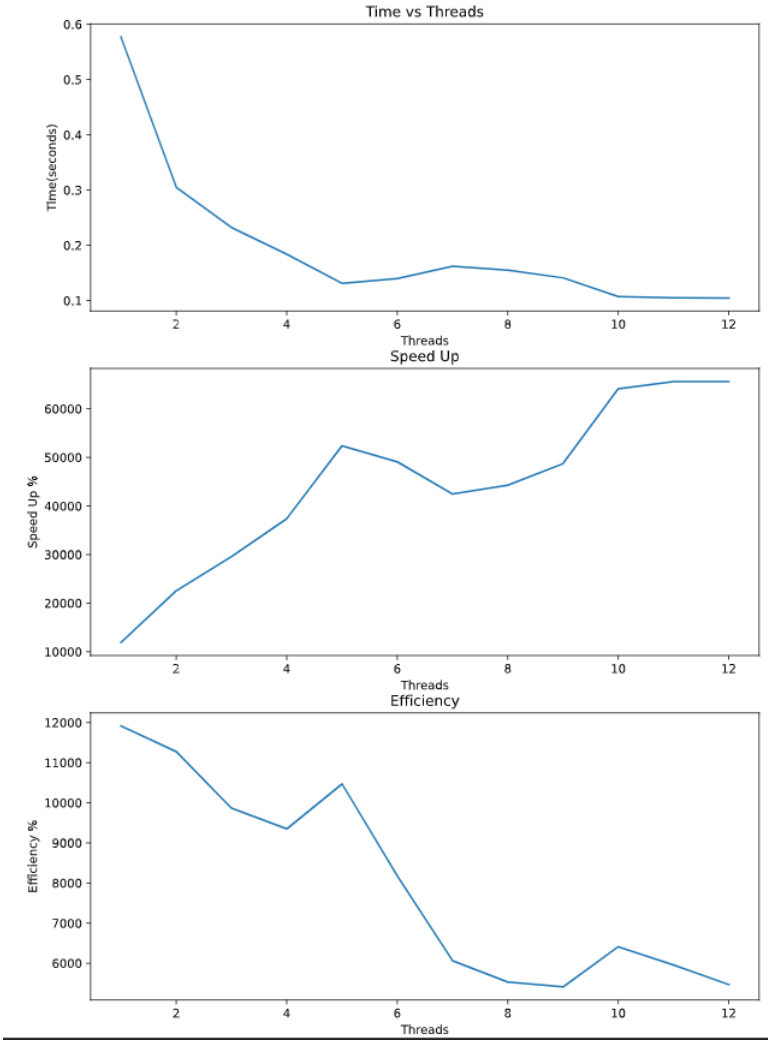
Candidate List vs Naive, N=800			
Threads	Time (Seconds)	Speed Up	Efficency
1	0.08363	82208.99677	82208.99677
2	0.066302	103694.8051	51847.40253
3	0.049314	139415.5493	46471.84978
4	0.042485	161825.0771	40456.26927
5	0.03528	194873.5374	38974.70748
6	0.029377	234031.3306	39005.22177
7	0.02788	246600.4519	35228.63599
8	0.029356	234198.7464	29274.8433
9	0.026432	260106.6283	28900.73648
10	0.026393	260490.9787	26049.09787
11	0.025503	269585.0787	24507.73443
12	0.022148	310413.3524	25867.77937

Candidate List vs Original(Naive): N=800



3Opt + Candidate List vs Naive, N=800			
Threads	Time (Seconds)	Speed Up	Efficency
1	0.577035	11914.60201	11914.60201
2	0.304921	22547.25284	11273.62642
3	0.232245	29602.91301	9867.637671
4	0.183852	37394.96116	9348.740291
5	0.131315	52356.07813	10471.21563
6	0.140037	49095.15628	8182.526047
7	0.16192	42460.09387	6065.727696
8	0.15529	44272.89845	5534.112306
9	0.141192	48693.65567	5410.406185
10	0.107253	64102.06148	6410.206148
11	0.104829	65584.31732	5962.210665
12	0.104769	65622.08548	5468.507124

3Opt-Candidate Lists vs Original(Naive): N=800



These graphs show the Candidate List and 3-Opt + Candidate List algo-

rithms vs my original naive implementation. This was to see how much of an improvement using the candidate list, don't look bit and other optimizations were on our computation. With a problem size of N=800, we see a whopping 310,000% increase in speed for the candidate list version and 65,622% increase for the 3 Opt version when using 12 threads. This shows just how much creating different approaches and reducing the order of your algorithm, if at all possible, will lead to way better performance then just throwing more cores at the problem. Due to the nature of the fact that as N grows these two algorithms speed up difference will likely continue to diverge, making the naive implementation keep taking more and more time.

5 Future Exploration

As I was writing up this report and working on the poster, I had the idea that I could us MPI and actually parallelize the iterative nature of the Ant Colony System. It would mean a change in the underlying algorithm, but the spirit of the algorithm would remain. The approach would be too have different "colonies" of ants distributed to different computers. Each colony would use the algorithm as I have written it for K number of iterations. After k iterations, we would sync and average the pheromone maps across all the colonies. This would allow us to run multiple iterations of the without communicating, while still gaining the benefit of shared trials and exploration of the action space. In the future I may try and implement this using MPI and see what the results look like.

6 Conclusion

This expirence we fun to work with. Not only did I get to apply my skills gained this semester but I also got to learn more about ACS and solving traveling salesman problems with algorithms like 3-opt. I have learned that there are many useful reductions of the traveling salesman problem that greatly improve the computation time. It also showed me that implementation, or thinking about how your algorithm works and being open to similar approaches, that while may not give you exactly the same answer, get you close enough, can greatly improve your computation time - way more then just throwing more processing power at the problem. There may be a lesson in life somewhere there.