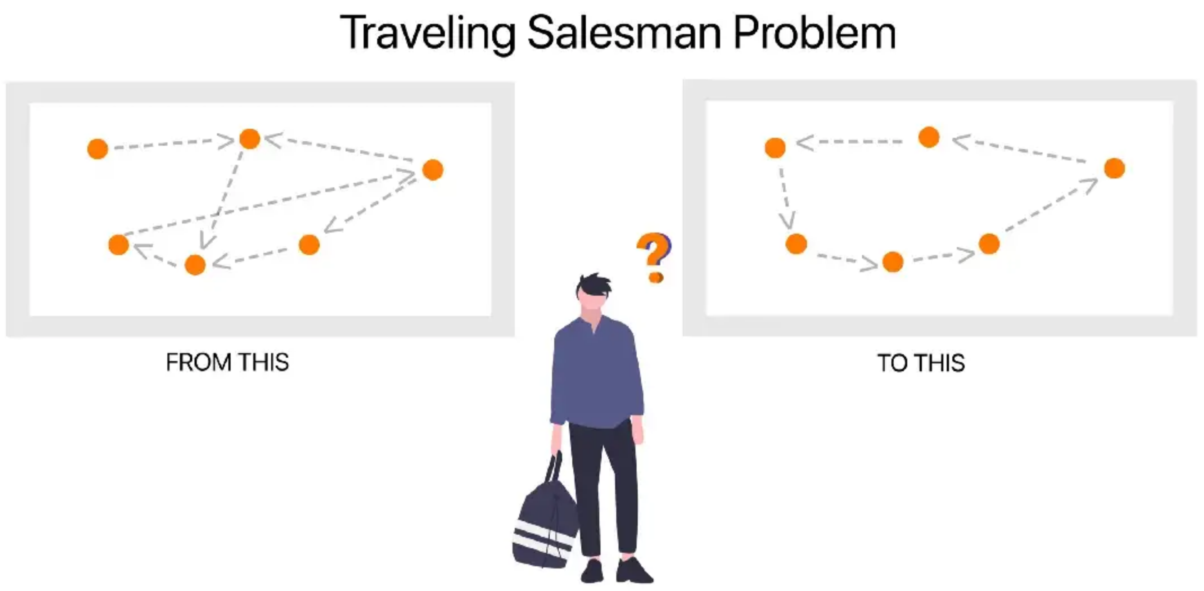
EE451 Final Project Report

David The and Scott Susanto

Introduction

From an undergraduate perspective, a problem without an efficient solution would be one that lies in the NP-Complete space, an ideal target for parallel implementations to accelerate. One such instance would be the Travelling Salesman Problem (TSP), which has an algorithmic time complexity of O(N!) and space complexity of O(N^2). Formally, the TSP is defined as follows: given a weighted, directed, non-negative graph, is it possible to find the path of least cost that visits all the vertices in the graph and returns to the origin? Effectively, we're trying to find the value and route of the shortest Hamiltonian Cycle.

Importantly, the TSP is a pervasive problem that appears very frequently in route planning, logistics, microchip manufacturing, DNA sequencing, and so many more. Its O(N!) serial time complexity means that modern computers are unable to solve inputs of more than 14 nodes in a realistic amount of time. Because of its time complexity, algorithms experts have designed numerous algorithms to attempt to solve the problem.



Approximate vs Exact Algorithms

In general, there are 2 broad categories of solutions: exact and approximate solutions. Today, approximation algorithms employ intelligent heuristics that return an optimal solution with about 95% accuracy in a realistic amount of time. Some examples include the Branch and Bound, Nearest Neighbor, k-Opt, and V-Opt heuristics, Christofides' Algorithm or Genetic Algorithms.

However, what we're interested in is exact algorithms which are still either slow or inefficient. The most common, efficient exact algorithm is one called the Held-Karp algorithm which uses recursive backtracking with Dynamic Programming, and is able to take the O(N!) time complexity to $O(2^N N^2)$, which is really good, except that it also takes $O(2^N)$ memory, making it equally infeasible to process *large* input graphs, which we will talk more about in the next section.

Memory Requirements and our Mistake

For a 17x17 input graph, the Held-Karp algorithm would require about 1MB of memory. This is a perfectly reasonable memory requirement especially given modern computers, which typically have RAMs of at least 8 GB, which means the Held-Karp algorithm is an ideal candidate for solving the TSP for a 17x17 problem size. However, during our initial research, we mistakenly thought the exponential space complexity of the Held-Karp algorithm was too much for *large* problem sizes, with *large* being poorly defined, leading us to further believe that a parallelized brute force algorithm would be better for our experiment. In hindsight, this would not be the case, and we acknowledge our mistake before moving forward with our experiment.

Number of Calculations and Time Complexity

In any case, we still believed that we would achieve massive speedup from parallelizing the serial brute for approach, a process which we hope would still champion the parallelization concepts taught in class.

With that said, allow us to introduce the problem from a quantitative perspective for time complexity by bringing to light the calculations required for input graphs of different sizes. The following demonstrates the lower bound for the number of computations needed for input graphs of sizes 10 to 17:

10! = 3,628,800

11! = 39,916,800
12! = 479,001,600
13! = 6,227,020,800
14! = 87,178,291,200
15! = 130,767,440,000
16! = 2,092,279,000,000

For the serial brute force approach, the times taken to process input graphs of sizes 10 to 17 are as follows:

10x10: 0.128s
11x11: 1.45s
12x12: 20.6s
13x13: 589.9s
14x14: too long to measure
15x15: too long to measure
16x16: too long to measure

Given the number of calculations and serial processing times, we reasoned that if we could use a few thousand threads, we could reduce the time complexity significantly, and potentially process 14x14, 15x15, and 16x16 sized input graphs inside 600 seconds (10 minutes), our loose definition for a *realistic* amount of time.

Serial Brute Force and Parallelization Strategy

Before beginning the experiment, it would be prudent to understand why the serial brute force approach has such a nasty O(N!) time complexity, and what we're doing to parallelize the serial approach. Essentially, we are generating every possible permutation for a list of cities that we could traverse and calculating the costs of travelling that permutation, and finding the minimum cost across all permutations.

Hence, an intuitive yet non-trivial approach would be to spawn a new thread at each decision branch, with each child thread responsible for aggregating the minimum cost across the remaining paths having taken that particular route in the algorithm. At the end of the execution, the main thread would reduce a list of costs and return the lowest cost of all the possible routes, solving the TSP in a parallel fashion.

Hypothesis

This brings us to our hypothesis. While the serial brute force approach could only solve a 14x14 input graph in a *realistic* amount of time, we believe that a parallel brute force approach could solve *larger* problem sizes in a *realistic* amount of time. More formally, our hypothesis is given as follows:

"Parallelized Brute Force allows us to process input graphs of sizes of 14, 15, and 16, while maintaining the O(N^2) time complexity in a realistic amount of time (10 mins)."

Experimental Setup

Data Sets

We are using data sets curated by Florida State University Department of Computing's Cities Dataset Collection. Artificial distances are algorithmically generated with real life constraints, and real distances were gathered from digital maps.

- CO04: 4x4 matrix representing distances between 4 artificial cities
- SH07: 7x7 matrix representing distances between 7 artificial cities
- LAU14: 14x14 matrix cut from the LAU15 data set
- LAU15: 15x15 matrix representing distances between 15 artificial cities
- LAU16: 16x16 matrix with random data added to the LAU15 data set

Because the Cities Dataset Collection has graphs with input sizes of varying granularity, we used the LAU15 dataset to construct input graphs for the sizes we desire. We acknowledge that this might not represent realistic values anymore, but we hope the similarity to the original LAU15 would render these differences negligible.

OpenMP

We chose to use OpenMP simply because it was more intuitive to use than other parallelization strategies. However, we don't believe that there would be a difference if we used PThreads or MPI.

CARC

For the experiment we are using 16 CPUs on the CARC platform, with 16 threads per CPU. In an ideal world, we could try to modify our program to make use of dozens of

CPUs with thousands of threads each because we think the parallelized brute force is a strongly scalable solution. However, in our case, as in the real world, we have limited resources, and were only able to allocate a maximum of 16 CPUs on CARC with 16 threads each.

Usage

- Go to the CARC Terminal and run the following commands:
- salloc --time=2:00:00 --cpus-per-task=16
- export OMP_NUM_THREADS=32
- make
- ./tsp_gnu_openmp X (where X is the input size of the graph between 10 and 16)

Experimental Results

The following table shows the results for our serial vs parallel brute force approach for different problem sizes:

Size of Input Graph	Serial Brute Force (seconds	Parallel Brute Force (seconds)
10x10	0.075	0.05
11x11	0.98	0.14
12x12	12	1.25
13x13	43	13.39
14x14	TLE	197.0
15x15	TLE	TLE
16x16	TLE	TLE

execution time in seconds (s)

Analysis

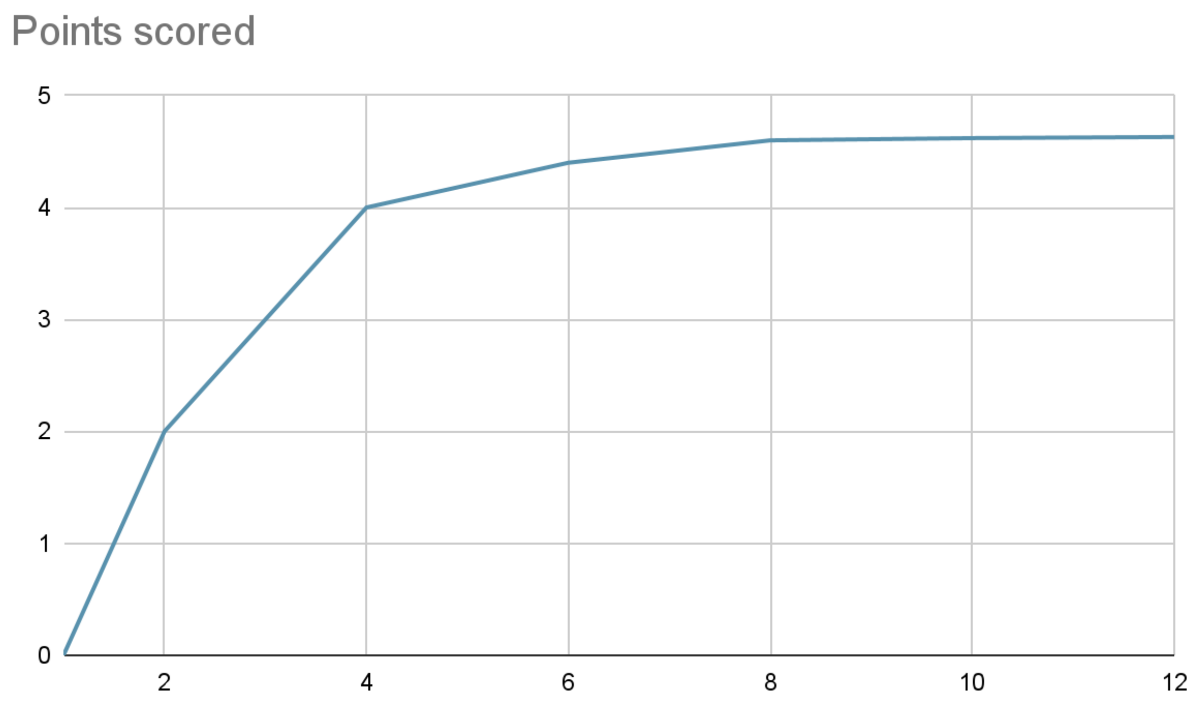
We were able to successfully process a 14x14 input graph in 197s, a little above 3 minutes, well below our own definition of *realistic amount of time*. However, even with parallelization and additional CPUs from CARC, we were still unable to process graphs of size 15x15 or larger as it took way too long.

As a result, **we found some support for our initial hypothesis**, achieving massive speedup in the 14x14 input graph. It is highly likely that we would see a much larger speedup in larger problem sizes, but given how large they are, it is unlikely that those speedups would be useful in any practical way. In that sense, we did not find support in our hypothesis.

Bottleneck

The way we implemented our parallelization is that instead of finding each permutation of path one by one in serial, we would find multiple permutations of path simultaneously in accordance to the number of threads we have. Through our experiments, we have found a significant speedup of our program when done in parallel. However, there comes to a point in our program where we are faced with several bottlenecks.

The first bottleneck which we found is that no matter how many threads we use, the minimum amount of time for any size data would be the time it takes for a single thread to execute. This is because although we can split the different permutations of paths into independent paths, we cannot split the task of finding each node in any singular path any further. This is because each node in the path relies on the previous nodes to find the minimum route. So if we assume all threads running at the same pace, then the minimum time is the time execution is the time it takes to find the longest path in the hamiltonian paths in serial. Since we have shown above that the larger the data set is, it will grow each permutations of paths exponentially, so will the runtime of each thread increase exponentially as data is increased linearly regardless of the amount of threads used. So eventually it will come to a point where the speed of parallelization becomes proportional to the speed of the serial portion (1 / serial portion). We can see that this occurrence is just like the Amdahl's law we saw in class.



The second bottleneck is resources. Since we used CARC as our platform, we only managed to set the max amount cpu as 16. As a result there is a limit to the amount of thread we can set in OpenMP before it becomes ineffective.

Room for Improvement

There is definitely room for improvement for the both of us. For one, we should have performed our initial calculations, running the expected number of calculations and expected space complexity of the brute force approach and Held-Karp algorithm, respectively, before solidifying our choice of parallelizing the brute force approach for this final project. That would have given us the crucial insight that our aim to solve the TSP in a realistic amount of time for *some problem sizes*, as mentioned during the presentation, would be in vain.

For us, perhaps the largest challenge was our lack of specialization in any computer science related field, putting us at a disadvantage against graduate students which were already relatively familiar with their respective fields. It would have been more prudent for each of us to have paired with a graduate student, simultaneously being able to learn more from their accumulated wisdom.

Conclusion

There’s no time for regrets however, and even without a hugely successful experiment, we both learned a lot from this final project. Initial research, research methods, talking to the right people, writing code in C, and parallelizing a serial approach were all incredibly useful skills we would take to our future careers both in the commercial sector and in academia.

We would also take this chance to thank Dr. Victor Prasanna and Jason for an extremely rewarding course which was simultaneously difficult yet painfully enjoyable in a fruitful way.