Traveling Salesman Problem Optimization

Bug (Chungeun) Lee
Computer Science
Virginia Tech
Blacksburg, USA
bug19@vt.edu

***Abstract*—Project repository: https://github.com/bug-vt/tsp_opt**

***Index Terms*—Traveling Salesman Problem, Optimization, Approximation algorithm, Dynamic programming**

I. Introduction

This project focuses on the selected algorithms and techniques introduced in Writing Efficient programs by Jon Bently [4] to solve small-scale TSP (n=30). Parallelization using OpenMP was also explored for additional speed up. The book claim that the use of compiler optimization and parallelization can speed up the computation, but only by constant factors. As a result, each optimization only allows for an acceptable input size to grow by 1 or 2. The more effective approach that the book suggests is pruning the search space using an approximation algorithm and reducing the number of recomputations using caching. The result of the combined technique reduces the runtime complexity of the TSP algorithm from $O(n!)$ to $O(n^2 2^n)$, reducing the factorial growth to exponential growth.

II. Background

The first Traveling Salesman Problem (TSP) arose in 1856 by Hamilton, where the objective was to find "a path such that every vertex is visited a single time, no edge is visited twice, and the ending point is the same as the starting point" [1]. This is known as finding a Hamiltonian cycle. Now, TSP adds one additional constraint: find the lowest cost Hamiltonian cycle for the given weighted graph. In plain English, "What is the shortest route that starts and ends at the home city and lets the salesman visit all n cities exactly once?"

The objective is simple and it is easy to come up with a brute-force solution. The brute force solution is to try all permutations of visiting n cities in order, then select the ordering that gives the optimal result. However, the brute force approach requires computing (n-1)! different ordering of cities, where search space become intractable even for the small graph.

Many algorithms, techniques, and optimizations have been introduced and attempted to solve or approximate the solution for TSP. The exact solution for 24978 cities was solved by David Applegate at el. in 2001, and later for 85900 cities was solved with Concorde TSP solver in 2006 [2]. On the other hand, the use of modern TSP heuristics can find a suboptimal solution for 1904711 cities within 0.1% optimum [3].

III. Approach

A. Setup

All measurements were performed inside the rlogin machine. The specification of the rlogin machine is:

- Architecture: x86-64
- Logical CPU threads: 64
- Model: Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz
- L1d cache: 32 KB
- L2 cache: 1024 KB
- L3 cache: 22528 KB
- RAM: 375 GB
- OS: CentOS Stream 8
- Kernel: Linux 4.18.0-489

The brute force approach and all optimizations are written in C++ and compiled with g++ compiler version 8.5.0. Parallelism was implemented using OpenMP version 4.5.

Optimizations were implemented and tested incrementally. The number of input cities ranged from 8 to 30. The location of each city was set to have a random x-coordinate between 0 to 500 and a random y-coordinate between 0 to 500. Note that some optimizations were not sufficient to compute certain input sizes and above without requiring years of computation time. Therefore, some measurements from optimizations that do not include pruning are an estimation based on the growth pattern.

B. Brute force algorithm

The brute force approach set the baseline performance of the algorithm. All possible permutations of visiting order of cities can be generated using the recursive algorithm. The tour cost is computed at the base case by following the parent pointer. Also, we set aside a variable to keep track of the current minimum tour cost during execution. Once the algorithm terminates (i.e. exhausted all possible tours), the variable holds the minimum tour cost of the graph.

The following subsections describe the serial and parallel optimizations used for speeding up the TSP. Note that optimizations were applied incrementally in order, whereas later optimization implicitly uses all earlier optimizations.

C. Compiler optimization flag

Compiler optimization truncates output assembly or machine code without changing the semantics of the original code. It performs dead code eliminations, constant folding, strength reduction, etc. The O3 compiler flag was used for all measurements. A constant factor of speedup is expected.

D. Approximation algorithm and Caching

The idea here is to safely ignore computing some of the paths by making an approximation that the subpath to be computed cannot be part of the solution (i.e. path with minimum tour cost). To do so, the algorithm should never make an over-approximated guess. Over-approximation can result in ignoring the potential solution with minimum tour cost. Therefore, the approximation should be always less than the true cost of the path. For this reason, the cost of the Minimum Spanning Tree (MST) of the path can be used for the approximation.

Proof: We know that TSP finds a tour cycle that visits all vertices. After removing an edge from the TSP cycle, the solution becomes a spanning tree since all vertices are connected without any cycle. From this, we can see that there exists at least one spanning tree that cost less than the TSP tour cycle. Then, the lowest cost spanning tree (i.e. Minimum Spanning Tree) must cost less than the TSP tour cycle. Therefore, we conclude that it is safe to approximate the cost of a path using MST.

Prim's algorithm was used to compute the MST. The main idea behind Prim's algorithm is to find the next unvisited vertex with minimum cost and then visit it. A priority queue is generally needed for managing what vertex should visit next. The time complexity of Prim's algorithm with priority queue is known as $O((n+m)\log n)$ (= $O(n^2 \log n)$ for the complete graph). However, in the case of a complete graph, the next unvisited vertex with minimum cost must be one of the neighbors of the most recently visited vertex. Therefore, a simple scan of neighbors is enough to determine what vertex should be visited next, which can be stored in a variable instead of a queue. This simple modification allows MST to be computed in $O(n^2)$ time.

Another catch to computing MST is that many paths contain common MST. If two paths contain the same vertices, then the same MST can be constructed. So, to avoid recomputing MST for each path, the idea here is to cache the MST after the first encounter. The hash table was used for this purpose where a set of vertices as a key and the corresponding MST cost as a value.

Efficiency: For a set of n vertices, there are 2^n possible subsets. Therefore, there are at most 2^n unique MSTs that need to be computed. Assume that cached MST cost can be obtained in $O(1)$. Combining the approximation algorithm with caching, the total time complexity of the TSP now reduces to $O(n^2 2^n)$ in the average case.

E. Parallelization and Scheduling Policy

The idea here is to distribute the computation of the sub-path to multiple threads. For this reason, exploratory decomposition was used. Starting at the home city, there are $n-1$ sub-path that need to be explored. This naturally leads to partitioning the search space of TSP into $n-1$ parts that can be concurrently executed.

To maximize parallelism, a few adjustments were made to the serial implementation. First, instead of sharing computed MST among all threads, each thread gets assigned its own hash table for storing MST. Unlike serial implementation, now some MSTs get recomputed by multiple threads. As a result, total computation work is now become $O((n-1)\cdot(n-1)^2 2^{n-1})$ instead of $O(n^2 2^n)$. Second, the global minimum tour cost is shared among all threads. The global minimum tour cost is accessed in two scenarios. One scenario is when checking whether the sub-path can be pruned after comparing approximated cost with the minimum tour cost. Since minimum tour cost only decreases, accessing out-of-date value is still safe from causing over-approximation hence do not harm the correctness even without locking. Although it result in a more conservative pruning, avoiding locking contention was observed to be more effective. The other scenario is when the minimum tour cost must be updated. Unlucky scheduling order of concurrent threads can accidentally overwrite the potential minimum tour cost, so read and write must happen atomically. The lock must be used in this case, which can be problematic as the number of thread increase. Finally, a dynamic scheduling policy was used. Due to pruning and caching, the workload of each partition is highly unbalanced, making it a good candidate for dynamic scheduling.

Parallelism: Let parallelism $P = \frac{W}{S}$ where W represents the sum of total computation and S represents the longest/critical path length after parallelization. First, we know that search space is partitioned into $n-1$ parts where each part takes $O((n-1)^2 2^{n-1})$ to compute the optimal sub-path. From this, we find that total work is $O((n-1)\cdot(n-1)^2 2^{n-1})$. Next, let T be the number of threads. Then each thread gets assigned with $\frac{n-1}{T}$ partition of search space. There is one constraint that each thread cannot get assigned a < 1 partition. So, the ciritical path length is $O(\frac{n-1}{T}\cdot(n-1)^2 2^{n-1})$ for $T \leq n-1$. Therefore, $P = O((n-1)\cdot(n-1)^2 2^{n-1})/O(\frac{n-1}{T}\cdot(n-1)^2 2^{n-1}) = O(T)$ for $T \leq n-1$. However, this is a loose upper bound since it does not account for the lock contention to update the global minimum tour cost.

IV. RESULTS

A. Serial optimizations

I As predicted, compiler optimization showcased a constant factor of improvement over the plain brute force algorithm. However, it can be observed that constant factors of improvement have minimal impact on scaling the TSP. On the other hand, the approximation algorithm with caching has reduced the overall growth rate, making TSP more scalable than before.

B. Parallel optimizations

II The result shows linear/sub-linear speedup for a small number of threads $T \leq 4$. However, unlike the theoretical prediction, the result shows poor parallelism after 4 threads. The main reason is due to the heavy lock contention to update the global minimum tour cost as it increases in the number of threads.

TABLE I
SERIAL OPTIMIZATIONS

	Optimizations		
Size	**Brute force (sec)**	**O3 flag (sec)**	**Approx and Cached (sec)**
8	0.004	0.000	-
9	0.030	0.003	-
10	0.254	0.030	0.003
11	2.503	0.286	0.008
12	27.574	3.191	0.023
13	351	39.510	0.019
14	82 min	553	0.019
15	20 hours	2.3 hours	0.093
16	14 days	36 hours	0.049
17	232 days	26 days	0.282
18	11 years	470 days	0.189
19	217 years	24 years	0.123
20	4348 years	489 years	0.319
21			0.328
22			0.614
23			2.111
24			3.511
25			5.957
26			101.089
27			43.312
28			56.104
29			49.649
30			242.829

TABLE II
PARALLEL OPTIMIZATIONS

	Number of threads					
Size	**1**	**2**	**4**	**8**	**16**	**32**
15	0.101	0.054	0.050	0.051	0.065	0.065
16	0.055	0.033	0.030	0.031	0.037	0.039
17	0.295	0.175	0.131	0.129	0.117	0.122
18	0.196	0.109	0.091	0.066	0.053	0.046
19	0.129	0.091	0.084	0.070	0.077	0.077
20	0.335	0.226	0.161	0.170	0.163	0.160
21	0.323	0.171	0.113	0.092	0.111	0.113
22	0.616	0.423	0.288	0.278	0.268	0.299
23	2.048	1.408	1.166	0.895	0.810	0.854
24	3.466	1.846	1.131	0.723	0.861	0.816
25	6.166	3.691	1.860	1.732	1.914	1.852
26	99.790	56.264	30.239	24.574	22.653	23.994
27	42.289	22.271	20.988	20.907	20.530	20.580
28	54.552	35.794	32.676	28.118	26.221	25.860
29	49.031	25.161	22.197	19.127	19.286	18.819
30	235.528	151.188	98.473	81.753	73.378	67.008

V. CONCLUSION

This project has explored various algorithms and techniques to tackle the famous TSP problem. For serial optimization, compiler optimization, approximation algorithm, and caching were used. Compiler optimization made the program run faster by a constant factor whereas the approximation algorithm combined with caching have reduce the growth rate of TSP. With the growth rate change, the goal input size of 30 was now solvable in tractable time. For parallel optimization, the main technique was to minimize dependency and shared resources among threads without losing correctness. However, not all locks could have been removed, which caused lock contention among threads as the number of threads increased. Although the approach used in this project is far from state of art for solving TSP, techniques and optimization learned from the project were valuable and applicable to many real-world problems outside of solving TSP.

REFERENCES

[1] D. Biron, The Traveling Salesman Problem: Deceptively Easy to State; Notoriously Hard to Solve, Lynchburg, VA, Liberty University, 2006.

[2] B. Xavier, A Massively Parallel Exact TSP Solver for small problem sizes, San Marcos, TX, Texas State University, 2022

[3] R. Mariescu-Istodor and P. Fränti, “Solving the large-scale TSP problem in 1H: Santa Claus Challenge 2020,” Frontiers in Robotics and AI, vol. 8, 2021. doi:10.3389/frobt.2021.689908

[4] J.L. Bentley, Writing Efficient Programs, 1st ed. Prentice-Hall Software Series, 1982.