Parallel Approach to the TRAVELLING SALESMAN Problem

First Author
Affiliation / Address line 1
Affiliation / Address line 2
Affiliation / Address line 3
email@domain
& *SecondAuthor*
Affiliation / Address line 1
Affiliation / Address line 2
Affiliation / Address line 3
email@domain

1 Abstarct

The Travelling Salesman Problem (TSP) represents a fundamental challenge in classical optimization within computer science and operations research. This study addresses the inherent complexity of TSP by proposing a parallelized solution to efficiently compute near-optimal routes for large datasets. TSP involves finding the shortest route visiting each city exactly once before returning to the origin city, a task exacerbated by its NP-hard nature. Traditional algorithms face computational intensiveness when applied to sizable datasets. To mitigate this challenge, we propose parallelized approaches to famous algorithms, leveraging modern computing architectures to distribute computation across multiple threads. By doing so, we aim to improve computational efficiency while maintaining solution accuracy. Through experimentation and analysis, we demonstrate the efficacy of our parallelized solution in addressing TSP, showcasing its potential for enhancing optimization techniques across various domains.

Keywords: Travelling Salesman Problem; optimization; parallel computing; computational efficiency

2 Problem Statement

The Travelling Salesman Problem (TSP) is an important classical optimization problem that exists in computer science and operations research. Given a list of N cities and the pairwise-distances between the cities, the problem aims to compute the shortest possible route that visits each city exactly once, before returning to the origin city. While there are different approaches and solutions to the TSP, it is an NP-hard problem. As such, any algorithm that finds a close-to-optimal solution to the TSP problem for a large dataset will be computationally intensive. A parallelized solution to the TSP would allow computation to be distributed across processes, enabling it to be solved more efficiently on modern powerful computing architectures.

3 Literature Review

In the realm of computational optimization, the Travelling Salesman Problem (TSP) has garnered substantial attention and scrutiny, standing as a cornerstone challenge in computer science and operations research.

Over the years, a plethora of algorithms and methodologies have emerged, each striving to crack the puzzle of finding the shortest route that traverses every city exactly once before returning to the starting point. Among these, the brute-force method stands out as one of the earliest and most straightforward, albeit computationally prohibitive due to its exponential time complexity [2].

To circumvent the computational hurdles inherent in TSP, one of the prominent approach was considered by Richard Bellman by using Dynamic Programming Approach [1]. This approach was one of the first approach to solve the Travelling Salesman Problem with computation bounds, and significantly reduced the time complexity to $O(n^2 2^n)$. Going forward, researchers have turned to heuristic and metaheuristic strategies. These include the nearest neighbor algorithm, which adopts a greedy approach by selecting the closest unvisited city at each step, and the genetic algorithm, drawing inspiration from natural selection principles to evolve a population of potential solutions across generations [4,5]. Additionally, the rise of ant colony optimization (ACO) algorithms has been notable, leveraging the foraging behavior of ants to efficiently explore solution space and unearth near-optimal routes [3].

Despite the strides made in algorithmic innovation, TSP remains entrenched as an NP-hard problem, spurring ongoing exploration into avenues for bolstering solution quality and computational efficiency. Recent endeavors have delved into parallel and distributed computing paradigms, capitalizing on modern hardware architectures' computational prowess. Noteworthy among these are parallelized algorithms like parallel genetic algorithms and parallel ant colony optimization, showcasing promising strides in expediting the quest for high-quality solutions across large-scale TSP instances [5,6].

While the literature reflects significant headway in tackling the computational complexities of TSP, there exist fertile grounds for further exploration and ingenuity. Future research trajectories might encompass the fusion of machine learning techniques with optimization algorithms, the exploration of hybrid methodologies amalgamating diverse strengths, and the tailoring of algorithms to suit specific problem domains or constraints.

In sum, the literature surrounding TSP epitomizes a mosaic of algorithmic ingenuity and theoretical revelations, underscoring an enduring pursuit to unravel the intricacies of this quintessential optimization conundrum.

4 Implementation

Our first approach was with Bellman-Held-Karp Algorithm, which breaks TSP down as a multistage decision problem using dynamic programming. The time complexity of this approach is $O(n^2 2^n)$ due to the nested loops over cities and subsets of cities. We tried to parallelize this algorithm as below:

Algorithm 1 Parallel Bellman-Held-Karp Algorithm for TSP

```
procedure ParallelBellmanHeldKarp($cost\_matrix$)
    $n \leftarrow$ number of nodes
    $dp \leftarrow$ 2D array of size $n \times (2^n - 1)$ initialized to $\infty$
    $dp[START][0] \leftarrow 0$
    parallel for with $NUM\_THREADS$
    for $mask = 0$ to $(2^n - 2)$ do
        for $u = 0$ to $n - 1$ do
            if $mask\&(1 << u)$ then
                for $v = 0$ to $n - 1$ do
                    if $v \neq u$ and $(mask\&(1 << v)) = 0$ then
                        $new\_mask \leftarrow mask \mid (1 << v)$
                        $dp[v][new\_mask] \leftarrow \min(dp[v][new\_mask], dp[u][mask] + cost\_matrix[u][v])$
                    end if
                end for
            end if
        end for
    end for
    $best\_cost \leftarrow \infty$
    for $u = 0$ to $n - 1$ do
        $final\_mask \leftarrow (2^n - 2)$
        $best\_cost \leftarrow \min(best\_cost, dp[u][final\_mask] + cost\_matrix[u][START])$
    end for
    print "Minimum cost to complete TSP:", $best\_cost$
end procedure
```

We implemented the algorithm using Python's Numba package, specifically utilizing the OpenMP threading layer to parallelize the computation. This parallel implementation significantly increased performance, achieving a notable speedup compared to the serial code. Moreover, it successfully handled cases involving up to 20 cities. However, a major challenge encountered was related to memoization: the storage and management of data when each thread updates its calculations based on previously determined parent values. This aspect of the algorithm necessitated a degree of serialization, which limited the potential for further speed improvements. Despite this, the enhancements in computational efficiency were substantial, demonstrating the effectiveness of parallel processing in optimizing complex algorithms like Held-Karp for the Traveling Salesman Problem.

Following up, we explored a brute force approach towards TSP, which has a time complexity of $O(n \times n!)$ in the serial implementation. The algorithm as follows:

Algorithm 2 Parallel Traveling Salesman Problem Solver using Brute Force & OpenMP

```
procedure FINDBESTPATH($cost\_matrix$)
    Initialize $permutation\_base$ with all cities except $START$
    $best\_cost \leftarrow \infty$
    $best\_path \leftarrow$ empty
    for each city $i$ from 0 to $N - 1$ excluding $START$ do
        parallel for with $NUM\_THREADS$
        $perm \leftarrow$ copy of $permutation\_base$ with $i$th city first
        do
            $local\_cost \leftarrow$ compute cost of current $perm$ path
            if $local\_cost < best\_cost$ then
                critical section
                if $local\_cost < best\_cost$ then
                    $best\_cost \leftarrow local\_cost$
                    $best\_path \leftarrow$ current $perm$
                end if
            end if
        while $next\_perm(begin + 1..end)$
    end for
    return $cost, path$
end procedure
```

To further enhance our results for the Traveling Salesman Problem, we explored heuristic-based algorithms, notably the Two-OPT algorithm and the Genetic Algorithm. The Two-OPT algorithm proved to be faster for scenarios involving a smaller number of cities. However, its results were unpredictable. This is due to the Two-OPT algorithm minimizing travel costs by approximating the route rather than comprehensively exploring all potential routes, which lead to sub-optimal solutions.

Our most effective results were achieved using the Genetic Algorithm, another heuristic-based approach which also works on approximation basis. For its implementation, we employed CUDA computing combined with Numba's Just-In-Time compiler (NJIT). This approach significantly enhanced the algorithm's efficiency and scalability. The detailed implementation of the algorithm is as follows:

Algorithm 3 CUDA-based Genetic Algorithm for TSP

```
procedure EVALUATEFITNESS($tours$, $distance\_matrix$, $fitness\_scores$)
    $idx \leftarrow$ CUDA thread index
    if $idx <$ length of $tours$ then
        $total\_distance \leftarrow 0$
        for $i = 1$ to length of $tours[idx]$ do
            $total\_distance \leftarrow total\_distance + distance\_matrix[tours[idx][i-1], tours[idx][i]]$
        end for
        $total\_distance \leftarrow total\_distance + distance\_matrix[tours[idx][-1], tours[idx][0]]$
        $fitness\_scores[idx] \leftarrow \frac{1}{total\_distance}$
    end if
end procedure
procedure RUNGENETICALGORITHM($distance\_matrix$, $num\_generations$, $population\_size$, $threads\_per\_block$)
    $num\_cities \leftarrow$ dimension of $distance\_matrix$
    $population \leftarrow$ array of random permutations of $\{0, \ldots, num\_cities - 1\}$ for each individual
    $fitness\_scores \leftarrow$ array of zeros of size $population\_size$
    Copy $population$ and $distance\_matrix$ to CUDA shared device memory
    Calculate number of blocks needed
    for each generation do
        Call EVALUATEFITNESS on CUDA with blocks and threads configuration
    end for
    Transfer $fitness\_scores$ back to host
    $best\_tour\_idx \leftarrow$ index of maximum value in $fitness\_scores$
    $best\_distance \leftarrow \frac{1}{fitness\_scores[best\_tour\_idx]}$
    $best\_tour \leftarrow population[best\_tour\_idx]$
    return $\lceil best\_distance \rceil$, $best\_tour$
end procedure
```

This implementation has significantly enhanced both performance and robustness. With the genetic algorithm, we consistently achieved a speedup proportional to the increasing number of cities, demonstrating scalability. Remarkably, this approach successfully handled computations for up to 5000 cities, illustrating its capability to manage large datasets effectively.

5 Results

5.1 Brute Force Approach

We evaluate the serial version of the brute-force algorithm for different values of N, where N represents the number of cities. We record the execution time (in seconds) for different values of N.

N	Time (in seconds)
5	0.000144005
10	0.00344992
12	0.424023
13	5.58157
14	88.06
15	1700.54

Table 1: Brute-force (serial) Evaluation

We stop recording values beyond N=15 because further evaluations would likely take several hours or days to complete.
We make similar recordings for the OpenMP-based parallelized version of the brute-force algorithm. In this case however, we also vary the number of threads across evaluations to understand how the thread count contributes to performance.

N / THREADS	10	13	14	15	16
2	0.00657606	2.95916	49.0082	857.715	-
8	0.00715017	1.51671	23.1652	435.63	6151.47
16	0.0324709	1.50712	23.2603	435.69	6157.99

Table 2: Brute-force (OMP-parallelized) Evaluation

For small values of N (<=10), the serial version is clearly superior. However, for greater values of N, we see that the parallel version starts to perform much better. While we experiment with the thread count, trying counts of 2, 8 and 16, we notice that performance is best for thread counts that come close to the value of N.

We stop at N=16, because once again, trying to evaluate our program for values beyond this would most likely take several hours or days. We don't evaluate the program for N=2 and THREADS=2 for the same reason.

5.2 Bellman-Held-Karp Approach

We evaluate the serial version of the Bellman-Held-Karp algorithm for different values of N, where N represents the number of cities. We record the execution time (in seconds) for different values of N.

N	Time (in seconds)
5	0.000104189
10	0.004055023
15	0.313786268
20	23.42583203

Table 3: Bellman-Held-Karp (serial) Evaluation

We stop recording values beyond N=20 because further evaluations would likely take several hours or days to complete. Also it was beyond our computational resources.
We make similar recordings for the OpenMP and NJIT-based parallelized version of the Bellman-Held-Karp algorithm. In this case however, we also vary the number of threads across evaluations to understand how the thread count contributes to performance.

N THREADS	5	10	15	20
2	4.0360415	4.1224110	4.223972	15.87155652
8	4.0473103	4.0938608	4.274119	19.2515347
16	4.0449099	4.0318691	4.542940	27.3022568

Table 4: Bellman-Held-Karp (OMP-parallelized) Evaluation

For small values of N (<=10), the serial version is clearly superior. However, for greater values of N, we see that the parallel version starts to perform much better. While we experiment with the thread count, trying counts of 2, 8 and 16, we notice that performance is best for thread counts that is lesser. This is due to the serial nature of the code because of memoization, which prohibites to achieve a better parallel performance particularly in the looping parts of the algorithm, ultimately causing thread overheads in the scenario of larger number of threads.

We stop at N=16, because once again, trying to evaluate our program for values beyond this would most likely take several hours or days. We don't evaluate the program for N=2 and THREADS=2 for the same reason.

5.3 Genetic Algorithm Approach

We evaluate the serial version of the Genetic algorithm for different values of N, where N represents the number of cities. We record the execution time (in seconds) for different values of N. For this algorithm, we went for higher number of cities for evaluation.

N	Time (in seconds)
50	0.0532386
100	0.1050679
500	0.5934829
2000	3.777979
10000	44.387267
20000	146.292566

Table 5: Genetic Algorithm (serial) Evaluation

We stop recording values beyond N=20000 because it was beyond our computational resources.
We make similar recordings for the CUDA-based parallelized version of the Genetic algorithm. In this case however, we also vary the number of threads across evaluations to understand how the thread count contributes to performance. In this scenario, we went for higher values of threads because of relaxation in the computational resource (GPU).

N / THREADS	50	100	500	2000	10000	20000
16	1.5159335	1.5256674	1.616775	2.682469	30.76353025	116.6050260
32	1.6425964	0.2309277	0.304166	2.785823	32.11820102	115.5411098
64	0.2308123	1.5312524	1.603786	1.410170	30.12914705	122.9064562
128	1.529940	0.2396519	0.308249	2.725920	30.11923265	129.0336666

Table 6: Genetic Algoritm (CUDA) Evaluation

For small values of N (<=500), the serial version is clearly superior. However, for greater values of N, we see that the parallel version starts to perform much better. While we experiment with the thread count per block, trying counts of 16, 32, 64 and 128, we notice that there is not a significant difference in the speedup.

However, it can be seen that for N (<=20000) the time taken is lesser as the value of N becomes closer to the number of threads used.
We stop at N=20000, because once again, trying to evaluate our program for values beyond this was not possible with our currently available resources.

6 Performance Evaluation

One of the main metrics that we can use to get a quick understanding of the improvement of a parallelized algorithm over a serial one is **Speedup (S)**.

We resort to **Efficiency (E)** as another metric to understand the performance of our algorithm.

6.1 Brute-Force Approach

We first compare the speedup of the OMP-based parallelized brute-force approach, over the naive serial brute-force counterpart.

N THREADS	10	13	14	15
2	0.525	1.886	1.797	1.983
8	0.482	3.68	3.801	3.904
16	0.106	3.703	3.786	3.903

Table 7: Brute-force (OMP) Speedup Comparison

For trivial values of N (<=10), the serial version of the algorithm work very well, even outperforming its parallelized counterpart. For larger values of N, we start to see the benefits of parallelization, even with just 2 threads. The benefits of parallelization for a particular value of N seem to be most pronounced when the number of threads is close to the value of N. Across different values of N (13, 14, 15), the highest speedup seems to be in the range of 3.5 - 4.

We then evaluate the efficiency of our parallel algorithm.

N THREADS	10	13	14	15
2	0.263	0.943	0.899	0.992
8	0.060	0.46	0.475	0.488
16	0.007	0.231	0.237	0.244

Table 8: Brute-force (OMP) Efficiency Comparison

The results for N=10 give a bad impression, but this is because for N<=10, the serial brute-force algorithm outperforms the parallel version. For N=13, 14 and 15, efficiency is fairly high when the number of threads is 2. As the number of threads is increased for the same N, the efficiency falls. This is understandable, as the speedup doesn't increase proportionally with increase in the number of threads. We notice that by increasing the problem size (N), and the number of threads at the same rate, we can achieve almost the same efficiency across N=13, 14 and 15. This suggests that this parallelized algorithm is weakly scalable.

6.2 Bellman-Held-Karp Approach

N THREADS	5	10	15	20
2	0.000026	0.00098	0.0743	1.476
8	0.000026	0.00099	0.0734	1.217
16	0.000026	0.001	0.0691	0.858

Table 9: Bellman-Held-Karp Speedup Comparison

In our evaluations, we see speedups only for N=20, when the number of threads is small. It is logical to assume that for even greater values of N, the speedups will become significantly more noticeable.

THREADS \ N	5	10	15	20
2	0.000013	0.00049	0.03715	0.738
8	0.00000325	0.00012375	0.009175	0.152
16	0.000001625	0.0000625	0.00432	0.0536

Table 10: Bellman-Held-Karp Efficiency Comparison

We notice a decent value of efficiency only when N=20 and number of threads is 2. If we evaluate our program for larger values of N, we are likely to see better values of efficiency. However, even for a particular large value of N, efficiency is likely to drop as the number of threads increases.

6.3 Genetic Approach

THREADS \ N	50	100	500	2000	10000	20000
16	0.0351	0.069	0.367	1.408	1.442	1.255
32	0.0324	0.455	1.951	1.356	1.382	1.266
64	0.231	0.068	0.37	2.679	1.473	1.19
128	0.0348	0.438	1.925	1.386	1.474	1.134

Table 11: Genetic Algorithm Speedup Comparison

In our evaluations, we see speedups for N>=500. We also see a variation of speedup with increase of thread numbers in the case of N=20000. It is logical to assume that for even greater values of N, the speedups will become significantly more noticeable.

THREADS \ N	50	100	500	2000	10000	20000
16	0.0022	0.0043	0.023	0.088	0.0901	0.0784
32	0.001	0.0142	0.061	0.0424	0.0432	0.0396
64	0.0036	0.001	0.0058	0.041	0.0230	0.01859
128	0.00027	0.0034	0.015	0.011	0.01151	0.00886

Table 12: Genetic Algorithm Efficiency Comparison

We notice a decent value of efficiency only when N=20 and number of threads is 2. If we evaluate our program for larger values of N, we are likely to see better values of efficiency. However, even for a particular large value of N, efficiency is likely to drop as the number of threads increases.

References

[1] Richard Bellman. Dynamic programming treatment of travelling salesman problem. *ACM*, 1962.

[2] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. *Graduate School of Industrial Administration, Carnegie Mellon University*, 1976.

[3] M. Dorigo and T. Stützle. Ant colony optimization. *MIT Press*, 2004.

[4] A. E. Eiben and J. E. Smith. Introduction to evolutionary computing. *MIT Press*, 2007.

[5] D. Goldberg. Genetic algorithms in search, optimization, and machine learning. *Addison-Wesley*, 1989.

[6] J. E. Little. Algorithm for the traveling salesman problem. pages 972–989, 1963.