PHYS 244 - Parallel Computing for Science and Engineering, Spring 2019 # Term Project Report

Unravelling the Exact Travelling Salesman Problem (TSP) [*]

Rohit Gupta
A53272428
r4gupta@ucsd.edu

Zhimin Liang
A12577466
zhl213@eng.ucsd.edu

ABSTRACT

In this paper, we provide algorithms for finding an exact solution to the famous NP-Hard problem, the Travelling Salesman problem. These algorithms have been developed in three flavors - serial, parallel using OpenMP, parallel using Hybrid of OpenMP along with optimizations. In terms of speedup over the serial algorithm and on the datasets tested, OpenMP gives us a maximum of 8.48x while Hybrid algorithm provided us 19.59x. The hybrid algorithm can be scaled to any #city dataset and we performed a grid search for different node configurations, before reporting our best speedups for each of case.

1. INTRODUCTION

Minimizing operation cost and maximizing profit has been a founding ideology for every operations company and this is when the Travelling Salesman Problem (TSP) comes to mind. It has an application in every aspect - from tourists exploring a city to bots following the shortest paths in warehouses. Hence, it has been under the lens of many scholars starting from the 1930's in Vienna & Harward [1]. Although it was in 1972, when Karp [8] showed that this extended Hamiltonian Cycle Problem was actually *NP-Complete*, which implies the NP-Hardness of TSP.

Therefore, to solve it in a limited amount of time, there have been several algorithms that find an approximate solution, instead of the exact. Some imitate the artificial ant colony as ants are known to find the shortest distance by following pheromone trails [3]. Other algorithms devise crossover and mutation operators following the lead of genes [2]. There have even been neural network approaches to solve the 2D Euclidean TSP [4].

However, we base this work on the learnings of *Parallel Computing* from the class and apply it to the classic TSP (section 2), leaving "optimality" for future work. We chose to implement an exact algorithm which has much higher computational load than approximate algorithms. Our arsenal contains *OpenMP* and *MPI*, while our code base is in *C*. Our experiments and their algorithms are listed in section 3. We use the metric *speedup* over the serial algorithm to rate our results in section 4.
The computational resources are provided by Comet, courtesy of Prof. Michael Norman, who also serves as the *Director* of San Diego Super Computer Center (SDSC).

[*]Our code is also available on our descriptive GitHub Repository.

2. THE TRAVELLING SALESMAN PROBLEM (TSP)

The TSP asks the following question - "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?". This question can be solved by the use of *Hamiltonian Cycles*, although we have to select the minimum cost cycle. As the number of cities increases, the complexity becomes a limiting factor. In general, the number of combinations are given by `(n-1)!` but if we assume symmetricity in paths from one city to another and back, it reduces to $^{(n-1)!}/_{2}$. Further, the nearest neighbor heuristic also does not notably lead us to the shortest path [5].

Table 1: Number of combinations possible for n cities. It also shows the extreme scaling this problem undergoes even with small number of cities

# Cities	Combinations Possible
2	2
3	6
4	24
5	120
6	720
7	5,040
8	40,320
9	362,880
10	3,628,800
11	39,916,800
12	479,001,600
13	6,227,020,800
14	87,178,291,200
15	1,307,674,368,000
16	20,922,789,888,000
17	355,687,428,096,000
18	6,402,373,705,728,000

3. METHODOLOGY

In this work, we have implemented algorithms to solve the *exact TSP*. We increase our computational load in order to gain the best solution, as opposed to some algorithms that settle on approximate solutions. **Moreover, in order to be as general as possible we purposely did not make any assumptions about whether the distances between cities are symmetrical, or whether they follow Euclidean space.** Our aim is to provide an scalable algorithm that maximizes speedup for any number of cities and we employ *OpenMP* and *MPI* for this task.

3.1 Algorithm: Depth First Tree Search (DFS)

Our implementations treat cities as vertices and the paths between them as edges. Thus, as in a tree search, we make use of the *depth first* method instead of *breadth first* as the former finds a complete path quickly. This complete path can then be used as a criteria for rejecting more costly paths and can save a lot of waste computations in parallel algorithms. Table 2 provides definitions of some key terms that are consistently used in further sections.

Table 2: Definitions of key terms used

Term	Description
Tour/Path	Path being built starting from the origin city and ending with it as well. It may be implemented using a stack or linked list. Each node in the latter also carries information of cost until that node
City	Each city is represented as a unique number that ranges from 0 till (#cities - 1)
Paths	It represents a collection of several tours, represented as a linked list. It may contain complete or incomplete tours
"best"	The adjective is used for a tour to denote that it is the lowest cost completed tour at that point

3.2 Serial Traversal

3.2.1 Recursive DFS

This is our most memory efficient (least memory footprint) serial algorithm which exploits recursion to perform DFS. It uses a single *stack* to gradually build the best tour. Since stack enjoys the benefit of *spatial locality* in terms of memory storage, it is faster than the *linked list*. Although a drawback is that it does not support dynamic expansion.

It maps to `depthfirst_serial_recursion_dfs.c` in our code and the pseudo-code is present in Algorithm 1.

3.2.2 Iterative DFS

In this algorithm, we make use of linked lists to implement tours and *Paths*. It is closer to the above recursive version, but more *specialized for parallel tasking*. The pseudo code is present in Algorithm 2, where we push partial tours into `PathsLL`. Although this is a serial version, it is worth noting that these partial tours can then be worked upon simultaneously, which we exploit fully in our further sections.

This algorithm can be found in `depthfirst_serial.c` in our code.

Algorithm 1: Recursive DFS Algorithm

```
minCost = 0;
DFS(origin, this city, minCost, visited[]) for each city
 do
    if already visited then
        if is tour complete and the best then
            Update minCost
        else
            //ignore this city;
            continue
        end
    else
        //visit the city;
        DFS(origin, this city, minCost, visited[])
    end
end
```

Algorithm 2: Iterative DFS Algorithm

```
minCost = 0;
//PathsLL = collection of tours, or, 'Paths' from Table 2;
PathsLL = empty add origin city to tour add tour to
 PathsLL
while PathsLL is not empty do
    pop a tour;
    if tour complete then
        update minCost if this cost is lesser;
    else
        for each city do
            //check feasibility based on cost and visited;
            if city feasible then
                //visit the city;
                add city to tour;
                push tour to PathsLL;
            end
        end
    end
end
```

3.3 Traversal using OpenMP

This algorithm builds on previous *Iterative DFS* algorithm and utilizes *OpenMP* to make use of the available threads inside a single node of *Comet*. We also aim to use the number of threads available wisely and provide the best speedup. We tried two major approaches which involved experimenting with *tasking* from *OpenMp 3.0*.

With Tasks.

From Algorithm 2, we enclosed each iteration of the while loop as a task. Each thread could therefore work on it's own path independently from the others and add back feasible

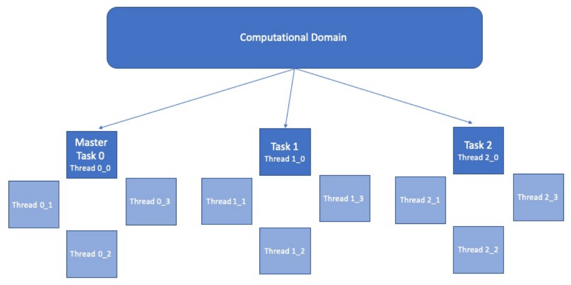

Figure 1: Depicts the layout of our hybrid program. This figure shows an example with 3 MPI tasks and 4 OpenMP threads each.

paths. Illustrated in `depthfirst_tasking.c`. This way no threads would be idle.

Without Tasks.
Again, from Algorithm 2, we observe some pieces of code that can work independently. While working on each city (& checking feasibility), same actions are being repeated independently. Another observation is that each tour from the *PathsLL* can be worked upon independently as well.

The final code of parallel traversal exists as `depthfirst_parallel.c` in our code, but it is based on the identified successful approach and further optimizations as mentioned in Section 4.3.

3.4 Parallel Traversal using Hybrid of MPI and OpenMP

Since OpenMP allows us to parallelize code only over a shared-memory system, we required another interface - MPI which provides us the capability to combine several nodes (with separate memories) to work on a single code together in synchronization. Additionally it allows us to use more than 24 threads one node limits us to.

There are two levels of communication involved (Figure 1). First level is between MPI *tasks*, while the second level is among the OpenMP *threads*. Task 0 is classified as the *master*.

This algorithm was built on top of `depthfirst_parallel.c` from previous section and it is located as `depthfirst_hybrid.c` in our code.

In this algorithm, each node along with its group of threads, works on a set of tours and provides update to other nodes. In the initial stages, each MPI task performed independently without any communication. Thus, each being unaware about others' minima, took more iterations to complete as it evaluated more Hamiltonian Cycles.

However, we added *asynchronous messaging* among the MPI tasks. Specifically, each task has a *master thread* which handles communication between tasks. When the lowest distance path of the master thread of a task is less than `updRat` times the previously shared distance between tasks it sends that value to the master threads of all tasks. Each master thread then compares their local minimum to this propagated minimum and uses the lesser of them, also propagating it to the other OpenMP threads in their task. This helps them in converging to the minimum cost and reduce wasteful computations.

Each master thread periodically uses *Iprobe* to check for incoming messages depending on the parameter `recMess` iterations.

Each master thread tells the others when it finishes its *Paths*. It also keeps track of which other task threads have finished. Once done, it will continue receiving messages into a buffer without acting on them until all task threads are finished. These actions are to prevent *deadlock*.

4. RESULTS

4.1 Experimental Settings

Computational Resources.
We have used *Comet* nodes for our experiments and their specifications are mentioned in Table 3. Although there are *Large Memory Nodes* with 64 cores as well, we only use the *Compute Nodes* throughout our experiments.

Table 3: Specifications of Standard Compute Nodes on Comet [6]

System Component	Configuration
Processor Type	Intel Xeon E5-2680v3
Sockets	2
Cores/socket	12
Clock speed	2.5 GHz
Flop speed	1.866 PFlop/s
Memory capacity	128 GB DDR4 DRAM
Flash memory	320 GB SSD
Memory bandwidth	120 GB/s
STREAM Triad bandwidth	104 GB/s

Modules used.
Our work greatly benefitted from the modules in table 4. We used Gprof to profile our *serial code* and hence optimizing the critical sections. Valgrind helped us in profiling our memory accesses, thereby, cutting down our memory footprint. DDT enabled us in flushing some parallel execution race conditions.

Datasets.
We obtained the datasets consisting of 5, 15, 17 & 26 cities from John Burkardt's TSP collection [7]. To fill the gaps, we *augmented* the 26 city dataset and created the remaining datasets from 6 till 18 cities. The datasets are located inside the `datasets` folder with `d` in the file name indicating *dataset* and `s` referring to *dataset solution*.

Table 4: Modules used for our work; all of them were present on Comet

Module	Version
GCC	4.9.2
GNU gdb	7.9
GNU Gprof	2.25
ICC	14.0.2 20140120
Valgrind	3.11.0
ARM DDT	19.0.2
Python	3.5.0

Execution Infrastructure.

We used shell and python scripts to run our experiments in batches. The `run_parallel.sh` runs several variations of `OMP_NUM_THREADS` over the selected dataset and reports speedup for each of them. This script is supposed to be executed on an entire compute node.

The `regress_hybrid.py` is more advanced and runs our hybrid code on selected dataset. Since the variations here are in terms of #nodes and their configurations, it launches separate *slurm* jobs and then aggregates their result, finally reporting the speedup over the serial code.

4.2 Serial Codes

Table 5 shows the time taken by different algorithms while using a single thread. It can be seen that although *Recursive DFS* is the fastest among them for shorter datasets, as we go reach 12-city dataset, it scales disproportionately. This is because the recursive DFS checks for feasibility only when the tour gets complete. However, the *Iterative DFS* checks for feasibility before every addition to tour. So, it needs to evaluate only a handful of complete paths, rather than all of them.

Another point to note is that *Parallel OMP Version* takes slightly more time than the *Serial DFS* due to the overhead added by OpenMP directives, despite using a single thread.

Table 5: Serial Execution of algorithms on different datasets. Empty boxes denote they could be calculated due to time limit restrictions.

Cities	Serial DFS (s)	Recursive DFS (s)	Parallel OMP Version (s)
5	0.000113	**0.000018**	0.000053
6	0.000214	**0.000047**	0.000262
7	0.001104	**0.000208**	0.001401
8	0.005972	**0.001449**	0.008542
9	0.031513	**0.012353**	0.045549
10	0.156429	**0.120017**	0.215348
11	2.310246	**1.313489**	2.393531
12	**2.23331**	15.348716	3.803033
13	**10.233092**	199.032149	16.892092
14	**107.936081**		131.425222
15	153.800294		**145.449833**
16	**6471.245587**		8503.010179
17	59900.22373		**51638.32985**

4.3 Parallel OMP Version

4.3.1 Optimizations

Our final code from section 3.3 is inspired by the following optimizations.

Tasking.

In `depthfirst_tasking.c`, we tried to enclose the entire algorithm as an untied task. One task would involve creating other tasks. The other threads would execute these other tasks. We made the task-generating task as untied to allow other threads to continue generating tasks if the original thread was occupied.

Ignoring parallel loop construct while adding cities to a tour.

From Algorithm 2, we notice that multiple threads can be allowed to iterate for each city using the `parallel for` construct. However, our results indicate that the OpenMP overhead eclipses the expected gain, and we end up with lesser speedup. We surmise that the loop statements are shorter for any multi-processing benefits to be visible.

Explicitly flushing update to shared variable.

Again, from Algorithm 2, while after confirming that the local minimum is better than the global minimum, updates to global minimum are explicitly flushed to ensure a quicker propagation to other threads. This increases the speedup by atleast 5% for all used datasets, some of them are shown in table 6.

Selecting non-tasking approach.

As mentioned above in section 3.3, we tried two approaches. However, after optimizations the tasking approach underperformed as compared to the non-tasking approach.

Table 6: Effect on execution times of explicitly flushing update for some datasets

Cities	Time (s) With Explicit Flush	Time (s) Without Explicit Flush
5	0.000315	0.000336
6	0.000726	0.000793
7	0.001182	0.001376
8	0.002043	0.002279
9	0.009247	0.009808
10	0.045634	0.048195

4.3.2 Speedups Obtained

Table 7 lists the time taken by our parallel version using OpenMP, along with their speedup. We observe that the speedups grow linearly with increasing city count (Figure 2). Although, we were unable to run 25+ city dataset (Comet time limit restrictions), but we conjecture that the speedups would reduce since the maximum possible threads

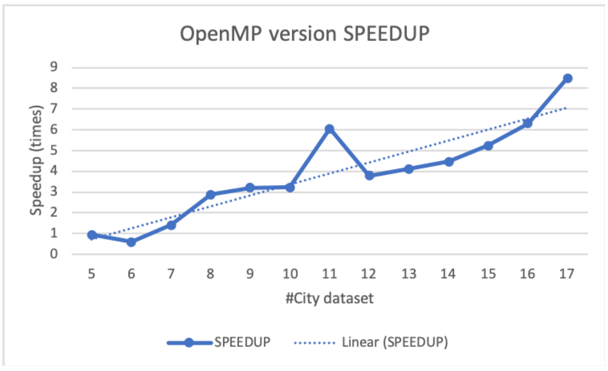

Figure 2: Speedups obtained through Parallel version using OpenMP. It shows a linearly growing trend.

supported by a node on Comet being 24. Beyond that, each thread would have to do double the work it currently does.

Table 7: Execution times taken by Parallel versions using OpenMP and Hybrid. Their speedups have been calculated using the best Serial Execution time from section 4.2. Empty boxes denote we could not calculate them in the given 48hrs time limits. + sign means that they're estimated using the serial time as 48hrs. So, they're the minimum guaranteed speedup. Also notice that with Hybrid program, we are able to run larger datasets as it provides more speedup.

Cities	OpenMP (s)	Speedup (times)	Hybrid (s)	Speedup (times)
5	0.000119	0.94957	0.000181	0.62430
6	0.000358	0.59776	0.000388	0.55154
7	0.000787	1.40279	0.000731	1.51025
8	0.002078	2.8739	0.001504	3.9707
9	0.009833	3.2048	0.005118	6.1572
10	0.048573	3.2204	0.022573	6.9299
11	0.382045	6.0470	0.168446	13.7150
12	0.588302	3.7961	0.252583	8.8418
13	2.490218	4.1093	1.07256	9.5408
14	24.171861	4.4653	10.224713	10.5563
15	29.31297	5.2468	22.435636	6.8551
16	1028.0012	6.2949	679.561243	9.5226
17	7056.0057	8.4892	3057.0549	19.5940
18	27464.6307	6.2917+	17295.8113	9.9908+
19			55955.1341	3.0881+

4.4 Parallel Hybrid Version

4.4.1 Optimizations

Non blocking send.
When sending a minCost to all other task we use a non-blocking send and then waits for all the sends to finish to be faster than using blocking sends.

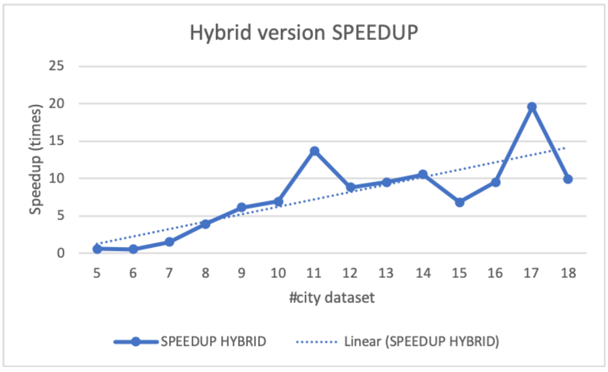

Figure 3: Speedups obtained through Hybrid version using OpenMP & MPI. It also shows a linearly growing trend but more aggressive than that of OpenMP.

UpdRat and recMess.
The values of `updRat` and `recMess` affected execution time and had to be tuned with grid-searching. Typical values altered execution time for 11 cities by up to 30 ms. We found `updRat` as 0.9 and `recMess` as 500 to be performing better.

Explicitly flushing update to shared variable.
The explicit flushing we did in the 4.3.1 no longer offered performance advantages, so we removed it.

4.4.2 Speedups obtained

The best case speedups achieved using hybrid versions among many configurations for each dataset is reported in Table 7. As can be observed from figure 3, the linear increasing trend in this hybrid version are more aggressive than that of OpenMP. Although, this model can be scaled to any city-dataset, we ran into `max time limit` errors for 20-city dataset.

5. CONCLUSION

Overall, we think that our algorithms provide promising results, especially *hybrid*. The maximum speedup achieved was 16.4864x for 17-city dataset where we used 16 MPI *tasks* with a single OMP *thread* each. The speedup seems to be increasing linearly with no symptoms of slowdown (on our tested datasets). Thus, we're confident that it would still grow until we run out of unique MPI tasks.

Future Work.
We think that to extract more parallelism in the hybrid algorithm, we would need to profile the communication patterns of all the tasks. This could further bolster the convergence rate among all threads (under all tasks) by avoiding unnecessary Hamiltonian cycles. Part of this involves finding the best combination of `updRat` and `recMess`. Alternatively, instead of probing every recMess iterations one could check via some function depending on the number of iterations (such as more often early on and less often later).

Additionally, currently if the master thread in a task finishes its set of *Paths* early, then it stops communicating with other tasks. Fixing this could help.

An alternative option is switching back to tasking and optimizing it. Likely due to the numerous critical sections which bottle necked especially as more threads joined. Further, there was no priority given for threads to finish almost completed tasks instead of newly started ones. This slowed the `minCost` convergence.

6. REFERENCES

[1] Cook, William J. In Pursuit of the Traveling Salesman: Mathematics at the Limit of Computation. Princeton, NJ: Princeton UP, 2012. link

[2] Larranaga, P., Kuijpers, C.M.H., Murga, R.H., Inza, I. and Dizdarevic, S., 1999. Genetic algorithms for the travelling salesman problem: A review of representations and operators. Artificial Intelligence Review, 13(2), pp.129-170.

[3] Dorigo, M. and Gambardella, L.M., 1997. Ant colonies for the travelling salesman problem. biosystems, 43(2), pp.73-81.

[4] Joshi, C.K., Laurent, T. and Bresson, X., 2019. An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem. arXiv preprint arXiv:1906.01227.

[5] (Originally in German) Karl Menger, Ergebnisse eines Mathematischen Kolloquiums (link

[6] Technical Summary of Comet is present here.

[7] TSP Datasets managed by John Burkardt of Florida State University (FSU) (link)

[8] Held, Michael, and Richard M. Karp. "The traveling-salesman problem and minimum spanning trees." Operations Research 18, no. 6 (1970): 1138-1162.