Parallel Cluster Programming to Solve
The Traveling Salesman problem

Project Report

Jaydeep Untwal
Rochester Institute of Technology
jmu8722@rit.edu

Sushil Mohite
Rochester Institute of Technology
spm6749@rit.edu

Harsh Sadhvani
Rochester Institute of Technology
hhs5765@rit.edu

ABSTRACT
As technology is continuously changing, faster and faster cluster and multi-core computers are available which can help solve problems that usually take years for an everyday computer to solve. One of such NP-Hard problem we look in this paper is the famous Traveling Salesman Problem. Solving the Traveling Salesman Problem using a multi-cluster parallel program gave a near optimal solution in a very feasible time.

1. COMPUTATIONAL PROBLEM
The Traveling Salesman Problem aims at finding the minimum cost path to visit ‘N’ cities and return to the origin. Looking at it as a completely connected graph of cities as vertex and weighted edges as cost to travel between cities. Finding a Hamiltonian Cycle with the minimum cost in such a graph is a highly expensive computational problem.

2. RELATED WORK
2.1 Research Paper 1
Paper [1] S. Dhakal et. al. discusses various approaches to solve the Traveling Salesman Problem. It then focuses on the Nearest Neighbor and Progressive Improvement Approach. It then discusses about a new hybrid approach that combines the previous to approaches to create a new algorithm to solve the Traveling Salesman Problem.

2.1.1 Nearest Neighbor Algorithm
1. From current town, find nearest unvisited town B
2. Mark town B as visited
3. If no visited town remains, terminate
4. Set town B as current town
5. Go to step 1

2.1.2 Progressive Improvement Algorithm
1. Searches through a list of solutions, till it finds an optimal solution
2. Starts with tour of 3 cities
3. Adds additional cities to it until it reaches the nth city.
4. The shortest path is then constructed backtracking from there

2.1.3 Results of the paper
[1]S. Dhakal et. al. finally experimentally compares the result of nearest neighbor and hybrid approach, to conclude that the hybrid approach gave better results.

2.1.4 Ideas from the paper
We decided to implement the Nearest Neighbor approach mentioned in the paper for our project. We observed that it gave ideal results for our test cases compared to the Heuristic Search approach that was also implemented.

2.2 Research Paper 2
Paper [2] X. Xuo-Yong et. al. discusses different algorithms to solve the Traveling Salesman Problem. It explains the most common Nearest Neighbor Algorithm and its advantages. It also proposes an improved Nearest Neighbor algorithm and then compares the two algorithms.

2.2.1 Corner Node Insertion Algorithm
The corner node insertion algorithm virtually plots the cities using the x and y co-ordinates. It then searches for the minimum cities on both X and Y axes. These are called the corner nodes. It then connects the corner node. For every other disconnected node, it replaces an existing edge AB with minimum (AC + CB). In other words, the path from A to B is not A to C then C to B.

The pros of this algorithm is that it is easy to implement and gives better performance than the Nearest Neighbor algorithm.

The cons of this algorithm is it is sequentially dependent and cannot be processed in parallel, this does not qualify our requirement.

2.2.2 Euclidean Plane Algorithm
The Euclidean Plane Algorithm virtually plots the cities using the x and y co-ordinates. It then divides the whole plot into segments. This algorithm is similar to the previously discussed algorithm. The difference is, it chooses a random point and starts an orientation search. The orientation search chooses a corner point (The upper Left) and tries to connect other points in a nearest (using euclidean distance) clockwise direction fashion. After performing the sub-algorithm in all four segments, it connects the whole tour and prints it.

The pros of this algorithm is, it gives better performance than Nearest Neighbor algorithm.

The cons of this algorithm is, it is very complicated and
again sequentially dependent like the previous algorithm.

2.2.3 Ideas from the paper
As we wanted to implement a cluster parallel program which requires sequential independence, we chose an idea from the Euclidean Plane Algorithm of using euclidean distance for calculating the distance between the cities for our metric.

2.3 Research Paper 3

2.3.1 Approach
Authors of paper [4] talk about branch and bound approach to solve the Traveling Salesman Problem (TSP). They discuss potentially how large a TSP problem can become. Solving such a problem using single core could lead to virtually infinite time or failure. Due to this, the authors propose a new parallel approach using branch and bound technique for solving the Traveling Salesman Problem.

The authors initially talk about two solutions for this:

1. Optimize already existing algorithms with respect to running time and efficiency to solve the Traveling Salesman Problem
2. Improve the hardware equipment which will directly reduce the running time of the problem

The issue with the above solutions is that optimizing the existing algorithms would not yield much enhanced performance as the reduction in the running time would be very small compared to the total time. Also, the growth speed of hardware performance cannot catch the growth of exponential time due to the increase scale of Traveling Salesman Problem.

The authors introduce a Master-Slave approach using multicore systems. In this, we initialize the master by searching the children of the root node and calculating the lower and upper bounds of the problem. Then, master assigns each child node to a slave core. Further, the slave cores search the optimal path with the lowest cost.

The above solution addresses the following two issues:

1. Load Balancing: Mapping tasks equally to the idle cores so that each core has the same execution time
2. Distribution Problem: Task allocation in distributed systems and scheduling algorithm for multiple cores

Authors address these problems by making the master perform a static method of dividing tasks which is provided by the programmer. Slaves perform a dynamic method of further dividing the each task into subtasks and sharing the work with other idle slave cores. Next, the slave core with the highest cost path is given the highest priority. Finally the authors conclude by stating that the efficiency can be increased by improving the load balancing techniques and partitioning of task.

2.3.2 Ideas from the paper
We used the task partitioning techniques used in this paper for developing our parallel implementation of the Traveling Salesman Problem. Trying different partitioning techniques helped us compare and select the approach with the best efficiency.

3. IMPLEMENTATION
Our implementation uses the Parallel Java 2 library developed by Professor Alan Kaminsky [3].

3.1 Sequential Design
We implemented two sequential algorithms

1. Nearest Neighbor
Algorithm:
(a) From current city, find the nearest unvisited city
(b) Mark this city as visited and make it the current city
(c) If all cities have not been visited yet, then repeat the first step

Class TSPSeq is a sequential program that uses Nearest Neighbor technique to find the minimum cost path. Each iteration say ‘i’ initializes a new TSP path starting from the ith city. At the end of each iteration, a TSP path is generated using this greedy approach and passed to a reducer task. This reducer task compares the newly found path cost with the previous path cost and stores the best path.

2. Heuristic Shuffle-Tour
Algorithm:
(a) Start with a random tour and calculate the path cost
(b) For each iteration in T, shuffle the order of cities in the initial path and calculate its cost
(c) At the end of each iteration in T, compare the new and initial cost to store the best path

Class TSPRandomSeq is a sequential program uses Heuristic Shuffle-Tour technique to find the minimum cost path. It initially generates a path with random sequence of cities. For each iteration in T, it clones the initial path and tries to shuffle the sequence of cities using a seed defined by the user. At the end of each iteration, this newly found path is then reduced with the initial path to find the best path.

3.2 Parallel Design

1. Algorithm (Nearest Neighbor):
(a) Partition N iteration amongst K workers
(b) In each worker, for each starting city ‘X’ in the partition
(c) Mark city ‘X’ and find the nearest city ‘Y’ to city ‘X’
(d) Repeat previous step with city ‘Y’ until all cities are visited
(e) Reduce best tour amongst threads
(f) Put Tuple of the best tour for the worker
(g) In reducer, reduce best path from K workers to display best result
Class TSPRandomClu is a cluster parallel program that searches a minimum cost path for given list of cities using Nearest Neighbor Algorithm. Nearest Neighbor (NN) Algorithm starts with each of N cities and recursively travel to next minimum city. The minimum of all is selected. It divides (N - 1) cities into partition and each partition is given to a cluster node, where it searches for a optimal path from every starting city in its partition in parallel on multiple threads. If found, it reduces it to the best path. Worker then puts a tuple in tuple space for the reducer task. A separate Reduce Task collects all tuples after the all workers finish and prints the best path.

2. Algorithm (Heuristic-Shuffle-Tour):
   (a) Partition T iteration amongst K workers
   (b) In each worker, start with an initial tour
   (c) Shuffle the tour ‘X’ times, where X is the partition of T and reduce to get minimum cost path
   (d) Reduce best tour out of X iterations
   (e) Put tuple of best tour and cost for reducer
   (f) Reduce best path from K workers to display best result

Class TSPRandomClu is a cluster parallel program that searches a minimum cost path for given list of cities using Heuristic Search. Using Hybrid Parallel program, a master-worker pattern is used. It divides T iterations (specified by user) to partitions and each partition is given to a worker task, where it shuffles the tour in parallel on multiple threads in hope of searching a minimum cost path. If found, it reduces it to the best path. Worker then puts a tuple in tuple space for the reducer task. A separate Reduce Task collects all tuples after the all workers finish and prints the best path.

3.3 Developer Manual

These programs have been tested on RIT CS department machine Tardis that allows to run Cloud programs. To run this on the Tardis multi cluster machine all the class files needs to be put into a jar package. Steps to compile and create JAR package:

1. Set classpath and PJ2 on the Tardis machine.
   For Bash execute the following commands:
   - export CLASSPATH=.:../var/tmp/parajava/pj2/pj2.jar
   - export PATH=../usr/local/dcs/versions/jdk1.7.0_11_x64/bin:$PATH
   For CSH execute the following commands:
   - setenv CLASSPATH .:/var/tmp/parajava/pj2/pj2.jar
   - setenv PATH /usr/local/dcs/versions/jdk1.7.0_11_x64/bin:$PATH
2. Compile the java code:
   javac *.java
3. Create jar file: jar cf FileName.jar *.class

Once the JAR package is created you can use this execute the Java code.

3.4 User Manual

Use the JAR package created to execute the code.

- To execute Nearest Neighbor TSP Sequential Program:
  java pj2 jar=<jarFile> TSPSeq <N> <seed>
  Where <K> is a number of workers, <N> is a number of type int giving the number of cities >= 1 and <seed> is a number of type long giving the random seed.

- To execute Nearest Neighbor TSP Parallel Program:
  java pj2 jar=<jarFile> workers=<K> TSPClu <N> <seed>
  Where <K> is a number of workers, <N> is a number of type int giving the number of cities >= 1 and <seed> is a number of type long giving the random seed.

- To execute Random Heuristic Search TSP Sequential Program:
  java pj2 jar=<jarFile> TSPRandomSeq <N> <T> <seed>
  Where <K> is a number of workers, <N> is a number of type int giving the number of cities >= 1, <seed> is a number of type long giving the random seed and <T> is a number of type long >= 0.

- To execute Random Heuristic Search TSP Parallel Program:
  java pj2 jar=<jarFile> workers=<K> TSPRandomClu <N> <T> <seed>
  Where <K> is a number of workers, <N> is a number of type int giving the number of cities >= 1, <seed> is a number of type long giving the random seed and <T> is a number of type long >= 0.

4. PERFORMANCE

4.1 Strong Scaling

Table 1 shows the strong scaling performance for Number of cities (N) of 100 to 1250. Here the T value is taken as N*N*10. So T value varies from 0 to 15625000 for N of 1250.

The performance for N of 100 is not ideal as the T value is low to see any performance increase in the parallel version. Thus we can see in the graph below that the speed up is only of about 2.9599 for 8 workers. While other test cases see a speed up of more than 7 and in some cases more than 7.9.

The Running time decreases as the number of workers is increased as it can be seen from the graph.

We can also see in the Efficiency vs K graph that the efficiency falls as the workers increase for N of 100, but after N of 500 the efficiency is constant.

With this we can see we have seen ideal strong scaling for higher T and N values and after N of 500 and T of 2500000 we see the most ideal strong scaling performance.
<table>
<thead>
<tr>
<th>N</th>
<th>T</th>
<th>K</th>
<th>Time (ms)</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.9960</td>
<td>0.4650</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
<tr>
<td>2</td>
<td>1.6987</td>
<td>0.8493</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
<tr>
<td>3</td>
<td>2.0159</td>
<td>0.6720</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
<tr>
<td>4</td>
<td>2.0114</td>
<td>0.5028</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
<tr>
<td>5</td>
<td>2.6657</td>
<td>0.5331</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
<tr>
<td>6</td>
<td>2.7918</td>
<td>0.4653</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
<tr>
<td>7</td>
<td>2.5727</td>
<td>0.3675</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
<tr>
<td>8</td>
<td>2.9599</td>
<td>0.3700</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
</tbody>
</table>

| 250 | 625000 | 1 | 1.0000 | 1.0000 | 0.9960 |
| 2   | 0.9983 | 0.9942 | 1.0000 | 1.0000 | 0.9960 |
| 3   | 0.9605 | 0.9605 | 1.0000 | 1.0000 | 0.9960 |
| 4   | 0.9380 | 0.9380 | 1.0000 | 1.0000 | 0.9960 |
| 5   | 0.9180 | 0.9180 | 1.0000 | 1.0000 | 0.9960 |
| 6   | 0.8973 | 0.8973 | 1.0000 | 1.0000 | 0.9960 |
| 7   | 0.8846 | 0.8846 | 1.0000 | 1.0000 | 0.9960 |
| 8   | 0.8597 | 0.8597 | 1.0000 | 1.0000 | 0.9960 |

| 300 | 900000 | 1 | 1.0000 | 1.0000 | 0.9960 |
| 2   | 0.9942 | 0.9942 | 1.0000 | 1.0000 | 0.9960 |
| 3   | 0.9605 | 0.9605 | 1.0000 | 1.0000 | 0.9960 |
| 4   | 0.9380 | 0.9380 | 1.0000 | 1.0000 | 0.9960 |
| 5   | 0.9180 | 0.9180 | 1.0000 | 1.0000 | 0.9960 |
| 6   | 0.8973 | 0.8973 | 1.0000 | 1.0000 | 0.9960 |
| 7   | 0.8846 | 0.8846 | 1.0000 | 1.0000 | 0.9960 |
| 8   | 0.8597 | 0.8597 | 1.0000 | 1.0000 | 0.9960 |

| 500 | 2500000 | 1 | 1.0000 | 1.0000 | 0.9960 |
| 2   | 0.9942 | 0.9942 | 1.0000 | 1.0000 | 0.9960 |
| 3   | 0.9605 | 0.9605 | 1.0000 | 1.0000 | 0.9960 |
| 4   | 0.9380 | 0.9380 | 1.0000 | 1.0000 | 0.9960 |
| 5   | 0.9180 | 0.9180 | 1.0000 | 1.0000 | 0.9960 |
| 6   | 0.8973 | 0.8973 | 1.0000 | 1.0000 | 0.9960 |
| 7   | 0.8846 | 0.8846 | 1.0000 | 1.0000 | 0.9960 |
| 8   | 0.8597 | 0.8597 | 1.0000 | 1.0000 | 0.9960 |

| 1000 | 1000000 | 1 | 1.0000 | 1.0000 | 0.9960 |
| 2   | 0.9942 | 0.9942 | 1.0000 | 1.0000 | 0.9960 |
| 3   | 0.9605 | 0.9605 | 1.0000 | 1.0000 | 0.9960 |
| 4   | 0.9380 | 0.9380 | 1.0000 | 1.0000 | 0.9960 |
| 5   | 0.9180 | 0.9180 | 1.0000 | 1.0000 | 0.9960 |
| 6   | 0.8973 | 0.8973 | 1.0000 | 1.0000 | 0.9960 |
| 7   | 0.8846 | 0.8846 | 1.0000 | 1.0000 | 0.9960 |
| 8   | 0.8597 | 0.8597 | 1.0000 | 1.0000 | 0.9960 |

| 1250 | 1562500 | 1 | 1.0000 | 1.0000 | 0.9960 |
| 2   | 0.9942 | 0.9942 | 1.0000 | 1.0000 | 0.9960 |
| 3   | 0.9605 | 0.9605 | 1.0000 | 1.0000 | 0.9960 |
| 4   | 0.9380 | 0.9380 | 1.0000 | 1.0000 | 0.9960 |
| 5   | 0.9180 | 0.9180 | 1.0000 | 1.0000 | 0.9960 |
| 6   | 0.8973 | 0.8973 | 1.0000 | 1.0000 | 0.9960 |
| 7   | 0.8846 | 0.8846 | 1.0000 | 1.0000 | 0.9960 |
| 8   | 0.8597 | 0.8597 | 1.0000 | 1.0000 | 0.9960 |

Table 2: Weak Scaling Performance

<table>
<thead>
<tr>
<th>N</th>
<th>T</th>
<th>K</th>
<th>Time (ms)</th>
<th>Sizeup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.9960</td>
<td>0.4650</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
<tr>
<td>2</td>
<td>0.9983</td>
<td>0.9942</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
<tr>
<td>3</td>
<td>0.9605</td>
<td>0.9605</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
<tr>
<td>4</td>
<td>0.9380</td>
<td>0.9380</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
<tr>
<td>5</td>
<td>0.9180</td>
<td>0.9180</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
<tr>
<td>6</td>
<td>0.8973</td>
<td>0.8973</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
<tr>
<td>7</td>
<td>0.8846</td>
<td>0.8846</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
<tr>
<td>8</td>
<td>0.8597</td>
<td>0.8597</td>
<td>1.0000</td>
<td>1.0000</td>
<td>0.9960</td>
</tr>
</tbody>
</table>
4.2 Weak Scaling

Table 2 below shows the weak scaling performance for Number of cities (N) of 100 to 1250. Here the T value is taken as $N^2N^2$. So T value varies from 100000 for N of 100 to 15625000 for N of 1250. As the number of workers are increased, T value is a multiple of the number of workers.

For N of 100 having with T value of 100000 for one worker, we see any increase t to T of 800000 for eight workers. Similarly, the T value is increased for each worker in the other test cases.

For weak scaling we see ideal performance for almost all the test cases. Even for N of 100 having T of 100000 we that the running time remains almost constant for all the workers as the T value is increased.

As far the efficiency the least we see is 0.9198 for N of 100 with T 600000 on 6 workers. But this is also a very much ideal situation for this test case. As for the rest of the test cases we can see that we even got efficiency more than 1 in some cases.

Thus we can say that we have seen ideal weak scaling performance for our parallel program.

5. FUTURE WORK

There are many other algorithms which can be used to solve the Traveling Salesman Problem. But our effort was to find the best possible solution in the shortest time. We can use other algorithms with the Heuristic-Shuffle-Tour and come up with better solutions. Changing the heuristic metric can also improve the solution, e.g., swapping two intersecting edges. Further, we can also use Google Maps to get real distance data between different cities.
6. LEARNING’S

- Algorithm: We learned that Nearest Neighbor algorithm being a greedy approach performs better than Heuristic-Shuffle-Tour algorithm in most cases
- Efficiency: Strong Scaling does not give good efficiency for less iterations. We noticed a close to consistent efficiency in strong scaling after increasing the number of iterations to more than 500
- Heuristic: Running heuristic search for very large number of iterations takes much time to execute but gives a very near optimal result.

7. TEAM CONTRIBUTION

Deciding the project topic and finding related research papers for it was a mutual task performed by each member equally. Sushil designed the sequential approach and implemented most of it. Jaydeep designed and developed the parallel implementation of the sequential program. Harsh came up with various ways of designing the input data. He also performed many test cases of variable problem sizes, recorded results and made strong scaling and weak scaling graphs.

8. REFERENCES