Parallel Computing Research
Investigation Report

Topic: Traveling Salesman problem

Team Parallel Programmers
- Krish Rohra
- Pragash Vijayaragavan
Table of Contents

Overview
   Description of computational problem
Relevant work :
   Analysis of first research paper
   Analysis of second research paper
   Analysis of third research paper
Heuristics
Sequential approach
Parallel approach
Developer's manual
User's manual
Strong scaling performance data
Reasons for non ideal strong scaling performance
Weak scaling performance data
Reasons for non ideal weak scaling performance
Future work
Learning from the project
Contribution of each team member
References
Overview

Description of the computational problem:

One of the most well known problems which is considered NP hard is the Traveling Salesman Problem. These kind of exhaustive search problems are difficult to solve in polynomial time and these get even more problematic and time consuming when the problem size increases. However these problems are important enough to be solved and a solution is found.

The traveling salesman problem can be considered one such problem. The naming comes after finding the problem to be same as what a salesman has to do in order to visit cities to sell his product. The salesman will start from his home city and travel all cities where he has to sell his product only once and return back to his home city by spending minimum amount on travel tickets. Let us discuss this in a more formal manner.

Consider a graph which has N nodes in the graph and E edges, as per the traveling salesman problem we need to find a path from the source node say Ns to all other nodes in the graph through the edges E. Also one consideration in this graph is that the size of E is N!, that is there is an edge between each pair of nodes in the graph, in other words its called a complete graph. The final constraint in this graph is that when you start from a source Ns, you need to finish at a node which is a neighbor of node Ns, this is called a Hamiltonian cycle. Given these there can exist more than one solution for the problem. But we are only interested in the most optimum solution. The most optimum solution is nothing but the hamiltonian cycle in the graph with the minimum cost. To summarize, the problem description is finding the minimum cost hamiltonian path or cycle in a complete graph.

Solving this problem can be very difficult when the number of nodes is big. Say for a problem size of 12 cities, the number of paths which are in the graph is 12! (479001600). This huge problem size can be very time consuming to solve if you use a single machine but the same can be done much faster with parallel computing. In this project we have discussed how we have designed and implemented this problem so that we can solve it much faster using more cpu cores. Since the problem size is too big, we have imposed methods to reduce the problem size by pruning and by using heuristics. Remember we are interested only in the optimum path and not any path which is a solution.
Analysis of Research Papers

For further understanding how this problem is approached and solved, we went through 3 research papers.

**Paper 1 - Ant Colony Algorithm:**

This algorithm was proposed by Dorigo, an Italian scholar. This algorithm imitates the foraging behavior of the ants. The ants release pheromone while searching for food. The ants also have the ability to sense the amount of pheromone in the paths. Eventually all the ants choose the path with the most pheromone concentration which will be the shortest path. Ant colony algorithm can be used to solve the traveling salesman problem.

The steps which will be followed during this algorithm are as follows.

- Go through the search tree.
- If we find a solution which cost more than the solution we found till now, we penalize that path.
- If we find a solution which cost less than the solution we found till now, we complement that path.
- So, after a point a costly edge won’t be considered in the solution path.

Figure 1 shows how the ants travel in the path to find the solution.

![Figure 1. Ants behavior of foraging](image)

Although the Ant Colony Algorithm finds a good solution quickly, there is no mathematical foundation that proves that the algorithm will find the optimum path in shortest time. Also since this approach works on negative and positive feedbacks, it takes a long computation time. Sometimes it is possible that the algorithm can get stuck in a local maxima.
Paper 2 - Co-evolutionary Genetic Algorithm:

The Co-evolutionary Genetic Algorithm was proposed on 2008 at the International Symposium on Electronic Commerce and Security by Zhu Qiang. The common algorithms to solve the Traveling Salesman Problem include neural networks, quantum algorithm, ant colony and genetic algorithms. Genetic Algorithms are good for solving TSP because they quickly direct the search towards promising areas of the search space. Genetic Algorithms are based on the mechanics of natural selection and natural genetics.

The traditional Genetic Algorithms co-evolves individuals and are computationally intensive. They may also not provide optimal or acceptable paths within the time limit available and may be stuck at local optima. The Coevolutionary Genetic Algorithms are similar to Genetic Algorithms but they work on 2 different populations or more. Figure 2 shows how the genetic algorithms work. The CGA will have a host GA and a parasite GA from which the new generation of offsprings is created. CGA uses genetic operators which are similar to GA but adapted for the host-parasite model.

![Figure 2. Flowchart of genetic algorithms](image)

Initially we thought that the CGA will provide an near Optimal or optimal solution in comparatively less time than GAs. We planned to utilize the tuple space to do the inter
selection, crossover and mutation genetic operations. Though it seemed a good idea, since the mutation is randomized, we were not able to come up with a stop criteria.
**Paper 3 - Enhanced Hybrid Particle Swarm Optimisation** :

Particle Swarm Optimisation is a population based stochastic optimization technique developed by Dr. Eberhart and Dr. Kennedy in 1995. The Enhanced Hybrid PSO combines some principles of PSO, Cross-over operation of Genetic Algorithms and 2-OPT improvement heuristic. It also uses the MapReduce framework.

2-OPT is a well know algorithm to solve the TSP. The 2-OPT starts with a given solution and then searches in the neighborhood of the current solution. It also performs swapping of two edges to check whether it reduces the cost. The time complexity of 2-OPT is \(O(n^2)\).
First divide the search base into groups. Each group has a local best experience. (Local optima). Message sharing happens between the groups. So, every group will be learning from the best group and the worst group will be picked out.

The Algorithm has a punishment operation where in the particles in the worst groups are replaced by the particles in the best groups.
Heuristics

The Traveling Salesman Problem is a complete graph problem and hence the number of paths to be traversed in the graph increases factorial vise as the number of cities increases. This huge problem space can be reduced by using heuristics. In this section we will discuss the heuristic which we will be using in our program.

The Nearest Neighbour Algorithm:

The Nearest Neighbour Algorithm is a naive approach to calculate a minimum hamiltonian cycle in the graph. This algorithm does not provide the optimum solution, however the solution provided by this algorithm can be used as a heuristic to find the solution. Also this algorithm runs in polynomial time and hence will not take much processing to calculate the heuristic.

The Nearest Neighbour Algorithm selects the next nearest neighbour from the source, and keeps on selecting the next nearest not visited neighbour and hence eventually leading to a hamiltonian cycle. This algorithm is used because our graph is a complete graph. Given below is an example of how the nearest neighbour is calculated and the path cost is found.

![Figure 3. Sample nearest neighbour calculation](image)

Other algorithms can also be used to find a better heuristic like the N Nearest Neighbour Algorithm, (ie) we find the solution from different sources using the Nearest Neighbour algorithm. This algorithm gives a better heuristic than the nearest neighbour approach. However, for our program we are using the Nearest Neighbour algorithm.
Sequential Design And Operation of the Program

The sequential approach involves using a single thread to calculate the solution. This approach is the naïve brute force approach to find the solution. We do a depth first search on the graph to find path which are better than the heuristic which we found using the nearest neighbour algorithm as explained in previous section and then we prune nodes based on that heuristic. Also we update the solution if our DFS search finds a better solution than the heuristic which is already present. This way the number of pruned nodes will keep on increasing and the problem space will become small enough to finish and find the solution in a better time.

The time complexity of this is still $O(n!)$ and hence is very difficult to calculate solutions for higher number of cities. Below diagram shows how we do this approach.

![Sequential design](image)

Figure 4. Sequential design
Parallel Design And Operation of the Program

As the TSP is difficult to solve using a single thread and consumes lot of computational time, we have followed the parallel programming concepts to solve this problem. The computational problem can be divided and the work can be given to different threads/cores.

We will be taking advantage of the cluster parallel programming concepts taught in class to utilize the tardis cluster. This type of programming can use the full computing power of the cluster to provide the results. In order to do this, we need to divide the task among the various nodes in the cluster.

Figure 5. Branch and bound approach
We use a branch and bound approach to solve the TSP as shown in Figure 5. The branch and bound approach does a breadth first search till a threshold to get the subproblems and then does a DFS on these subproblems individually. We also use the same heuristic which we used in the sequential approach to make a fair comparison of the results. Figure 6, shows how this design is done.

![Diagram of parallel cluster program design](image)

**Figure 6. Parallel cluster program design**

Our cluster program first does a BFS on the graph to fill the tuple space which is used as a work queue and then every thread takes a task from the tuple space to find the solution to that subproblem. We also have our heuristic calculated using the nearest neighbour algorithm in the tuple space. Each thread after it has calculated a solution will take the solution object (as shown in Figure 7) and update the solution object if a better result is found. Paths which have higher value than the solution value are pruned. This way the solution object will always have a minimum value which has to compared and eventually the solution object will have the optimum solution.
Figure 7. Cluster parallel execution

Finally the main program gets the results from the tuple space after all tasks are finished and displays the result as shown in Figure 8.
Figure 8. Result of cluster
Developer’s Manual

The Data File:
  a. A sample data file is given below:

<table>
<thead>
<tr>
<th></th>
<th>20833.3333</th>
<th>17100.0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>20900.0000</td>
<td>17066.6667</td>
</tr>
<tr>
<td>3</td>
<td>21300.0000</td>
<td>13016.6667</td>
</tr>
<tr>
<td>4</td>
<td>21600.0000</td>
<td>14150.0000</td>
</tr>
<tr>
<td>5</td>
<td>21600.0000</td>
<td>14966.6667</td>
</tr>
</tbody>
</table>

The first column consists of the city id. The second and third column consists of the x-axis and y-axis of the city location respectively.

Sequential Program:
In order to compile the sequential program, the following files are required:
  - TSPSeq.java
  - Vertex.java
  - <sample input txt file>

Steps to compile the sequential program:
  1. Include PJ2 library in the machine’s classpath with the command,
     ```
     export CLASSPATH=.:~/var/tmp/parajava/pj2/pj2.jar
     ```
  2. Now, compile the program with the command,
     ```
     javac *.java
     ```
  3. After you have compiled the program, you need to package all your .class and .txt files into a jar. The .txt files are also included into the jar because while running the program, the jar will be moved to different nodes and the program tries to find the .txt file in the path the class files are present. Use the following command,
     ```
     jar cf TSPSeq.jar *.class *.txt
     ```
Parallel Program:
In order to compile the parallel program, the following files are required:
   TSPParallel.java
   Vertex.java
   <sample input txt file>

Steps to compile the parallel program:
1. Include PJ2 library in the machine’s classpath with the command,
   
   export CLASSPATH=.:./var/tmp/parajava/pj2/pj2.jar

2. Now, compile the program with the command,
   
   javac *.java

3. After you have compiled the program, you need to package all your .class and .txt files into a jar. The .txt files are also included into the jar because while running the program, the jar will be moved to different nodes and the program tries to find the .txt file in the path the class files are present. Use the following command,

   jar cf TSPParallel.jar *.class *.txt
User’s Manual

Sequential Program:
In order to run the sequential program, the following files are required:

```
java pj2 debug=makespan jar=TSPSeq.jar TSPSeq <textFileName>
```

- `<textFileName>`: The file that has the cities data

Parallel Program:
In order to run the parallel program, the following files are required:

```
java pj2 jar=TSPParallel.jar TSPParallel <textFileName> <threshold>
```

- `<textFileName>`: The file that has the cities data
- `<threshold>`: The threshold depth till which you want to carry out BFS and fill the work queue

Finally,

The program outputs 2 lines. The first line is the number of nodes visited by the program and the second line is the TSP solution found by the program.
Strong Scaling Performance Data

One thing to note was due to the way we build the parallel part, we couldn’t run it on a specified number of cores as the tasks were put into the tuple space and all the threads available would pick up the tasks. So we tested our program using the threshold parameter. For threshold 1, the number of tasks put in the tuple space were \( n(\text{cities}) - 1 \), so those many number of threads are used for the program. Also, for threshold 2, since more than 40 tasks were put into the tuple space, it was made sure before running the program that all the 40 cores were available on TARDIS cluster and it was seen that the program utilized all the 40 cores.

<table>
<thead>
<tr>
<th>cities</th>
<th>Sequential</th>
<th>Parallel</th>
<th>Threshold 1</th>
<th>Parallel</th>
<th>Threshold 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time Taken</td>
<td>Edges Visited</td>
<td>Cores</td>
<td>Time Taken</td>
<td>Edges Visited</td>
</tr>
<tr>
<td>12</td>
<td>148</td>
<td>1.1*10^5</td>
<td>11</td>
<td>880</td>
<td>4.6*10^4</td>
</tr>
<tr>
<td>13</td>
<td>246</td>
<td>5.3*10^5</td>
<td>12</td>
<td>885</td>
<td>1.1*10^5</td>
</tr>
<tr>
<td>14</td>
<td>714</td>
<td>2.4*10^6</td>
<td>13</td>
<td>1134</td>
<td>6.8*10^5</td>
</tr>
<tr>
<td>15</td>
<td>4477</td>
<td>1.6*10^7</td>
<td>14</td>
<td>3975</td>
<td>5.3*10^6</td>
</tr>
<tr>
<td>16</td>
<td>37255</td>
<td>1.3*10^8</td>
<td>15</td>
<td>19373</td>
<td>3.3*10^7</td>
</tr>
<tr>
<td>17</td>
<td>99207</td>
<td>3.3*10^8</td>
<td>16</td>
<td>65143</td>
<td>7.6*10^7</td>
</tr>
<tr>
<td>18</td>
<td>228261</td>
<td>7.1*10^8</td>
<td>17</td>
<td>155465</td>
<td>2.0*10^8</td>
</tr>
<tr>
<td>19</td>
<td>768385</td>
<td>1.8*10^9</td>
<td>18</td>
<td>460673</td>
<td>6.5*10^8</td>
</tr>
<tr>
<td>20</td>
<td>1368445</td>
<td>3.6*10^9</td>
<td>19</td>
<td>939833</td>
<td>1.0*10^9</td>
</tr>
</tbody>
</table>

Figure 9. Strong Scaling Data
Plots:

Running time vs # of cores

Speedup vs # of cores
Efficiency vs # of cores
Reasons for non ideal strong scaling performance

- Shared Minimum hamiltonian cycle solution object
- Number of edges visited

Each worker tries to take tuple and update the result. This operation is thread safe and hence when 2 or more threads try to take tuple they wait for the tuple to be available in the tuple space. Thus this becomes a major bottleneck especially when more than one thread finds a solution at the same time.

The number of edges visited by each thread is different as each node works on different areas of the problem space. The nodes which have promising areas of the search space do not prune much and hence will have longer paths to calculate than other threads and hence we don’t get ideal scaling.

To solve this problem, we tried to put different answers into the tuple space by each thread without accessing a common object to avoid the bottleneck, but in this case all the threads had a bigger problem space to work with because they were not pruning much by comparing their results with the common object and hence had bad performance compared to our initial method.
## Weak Scaling Performance Data

<table>
<thead>
<tr>
<th>Cities</th>
<th>1 core Sequential</th>
<th>40 cores Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Edges Visited</td>
</tr>
<tr>
<td>12</td>
<td>144</td>
<td>$1.1 \times 10^5$</td>
</tr>
<tr>
<td>13</td>
<td>246</td>
<td>$5.3 \times 10^5$</td>
</tr>
<tr>
<td>14</td>
<td>714</td>
<td>$2.4 \times 10^6$</td>
</tr>
<tr>
<td>15</td>
<td>4477</td>
<td>$1.6 \times 10^7$</td>
</tr>
<tr>
<td>16</td>
<td>37255</td>
<td>$1.3 \times 10^8$</td>
</tr>
<tr>
<td>17</td>
<td>99207</td>
<td>$3.3 \times 10^8$</td>
</tr>
<tr>
<td>18</td>
<td>228261</td>
<td>$7.1 \times 10^8$</td>
</tr>
<tr>
<td>19</td>
<td>768385</td>
<td>$1.8 \times 10^9$</td>
</tr>
<tr>
<td>20</td>
<td>1368445</td>
<td>$3.6 \times 10^9$</td>
</tr>
</tbody>
</table>
For the weak scaling, the number of edges visited was taken into consideration, if a program which was run on 40 cores had the nodes visited approximately 40 times that of the nodes visited by the program on 1 core then those two were analyzed for weak scaling.

Plots:

Running time vs # of cores

Sizeup vs # of core
Reasons for non ideal weak scaling performance

- Increased use of the Shared Minimum hamiltonian cycle solution object

With the increase in the number of edges visited, the effect of the bottleneck, that is, the shared minimum hamiltonian cycle solution object increases. The larger effect of the bottleneck increases the overall running time.
Future Work

Currently we are using the parallel java 2 library’s cluster parallel programming paradigm to solve the problem. The cluster program is written is such a way that it utilizes all the nodes in the cluster and divides the work among the all available nodes in the cluster. We use the tuple space to update the solution whenever a thread finds one. As our future work we can do the following approaches.

- Since we are accessing the same object in the tuple, we have a bottleneck and somehow avoiding this bottleneck can increase scaling.
- We can use GPU parallel program and utilize the huge processing power of gpus to find the solution for the problem. This will involve dividing the work among the GPUs. We will have to create as many subproblems as the GPUs cores and then do the computations.

The above two methods are for the current approach we use to solve the problem. We can also think of other approaches rather than the brute force heuristic approach.

- Genetic Algorithms: There are different ways to solve the problem based on the principles of selection, crossover and mutation as discussed previously in this paper.
- The second way would be to make use of GPU’s large number of cores while running multiple independent stochastic local search where in we could randomly select two edges and remove one edge and add the missing one between the three.
Learnings from the project

We were involved in learning about the classic TSP and the different approaches followed to solve the problem. Below is the list of learnings from the project.

- The TSP is a NP Hard problem and hence other algorithms like the Nearest Neighbour Algorithm does not provide the correct solution to the problem.
- The TSP is an exhaustive search problem and exhaustive searches are very difficult to solve using limited computational power, but with the parallel computing approach, the solution can be found much faster but still the time increases exponentially with the increase in the problem size.
- Using heuristics considerably reduces the problem size.
- The TSP problem has different areas of the problem space and some of those have more probability of giving a correct solution than others and hence pruning the nodes using a heuristic will divide the problem space unevenly among the available threads.
- Solving the problem using the branch and bound approach is good but selecting the threshold determines how fast the problem will be solved. If the threshold is big there will be lot of subproblems and this will take longer time as the interaction between the threads and the tuple space to get the task will increase.
- No interaction between the threads will result in lesser pruning than possible as other threads could have already found a better solution.

We also learned that other algorithms like the Genetic Algorithm can be used to solve the problem. Heuristic Search is way better to solve such problems as in a real world scenario, we would be happy with an optimal solution in an acceptable time rather the best solution in a really impractical time.
Contribution of each team member

It was team effort on every step of the project, we met up and discussed on what has to be done and how we will be doing it. There were lot of discussion on each stage of the project. For the presentations we created the slides together and gave trail runs on how we will be presenting it. For the coding part, initially we decided on using the cluster approach and designed the program.

The sequential part was done by Krish and the parallel part was done by both Krish and Pragash. For taking readings and execution we worked together, both strong scaling and weak scaling data and graphs were done together with lot of discussions. There was also lot of discussions with Prof. Alan Kaminsky regarding our approach and data collection.
References

