Solving Traveling Salesman Problem on High Performance Computing using Message Passing Interface

IZZATDIN A. AZIZ, NAZLEENI HARON, MAZLINA MEHAT, LOW TAN JUNG, AISYAH NABILAH
Computer and Information Sciences Department
Universiti Teknologi PETRONAS
31750 Tronoh, Perak
MALAYSIA
{izzatdin, nazleeni, mazlinamehat,lowtanjung}@petronas.com.my, allysa85@yahoo.com

Abstract: - In this paper, we present a parallel implementation of a solution for the Traveling Salesman Problem (TSP). TSP is the problem of finding the shortest path from point A to point B, given a set of points and passing through each point exactly once. Initially a sequential algorithm is fabricated from scratch and written in C language. The sequential algorithm is then converted into a parallel algorithm by integrating it with the Message Passing Interface (MPI) libraries so that it can be executed on a cluster computer. Our main aim by creating the parallel algorithm is to accelerate the execution time of solving TSP. Experimental results conducted on Beowulf cluster are presented to demonstrate the viability of our work as well as the efficiency of the parallel algorithm.

Key-Words: - Traveling Salesman Problem (TSP), High Performance Computing (HPC), Message Passing Interface (MPI)

1 Introduction
Traveling Salesman Problem (TSP) is a well known problem that involved repetitive process which would be resource exhaustive if it is applied on a huge coordinate set and if it were to be executed using sequential machine.

In TSP a set of N cities is given and the problem of finding the shortest route connecting them all, with no city visited twice and return to the city at which it started. For any two cities c1 and c2 the distance is given by d(c1, c2). It is a symmetric TSP (STSP), if the distances satisfy d(c1,c2) = d(c2,c1). Otherwise the TSP is called asymmetric (ATSP). The sum of all distances of a valid route is called the tour length [4].

Since the task of solving the TSP accurately is not feasible, to get a solution for a TSP problem one could either focus on only small instances, or look for an approximate solution within polynomial time. If one chooses to focus only on small instances, one will loose the possibility to solve many interesting problems. One of the reasons for the interest in the TSP is that it often is a part of another problem that can be solved by using a TSP solver. Solving small problems of this type is not often enough since large instances of the TSP problem is related to many industrial and scientific modeling tasks. Since the focus of this research is only interested in the underlying technology of TSP, then there is no need to focus on small instances. Therefore the study will be on finding the approximate solution for solving TSP.

Ideally TSP should be solved by an algorithm that could perform fast computations on large data sets. In this paper we proposed a plausible approach in solving TSP computation by developing a parallel algorithm using C language and Message Passing Interface (MPI) directives. It has been proven that tasks accomplished through parallel computation results in faster execution as compared to a computational processes that runs sequentially [2]. MPI was chosen due to the fact it is designed for high performance computing on parallel machines or cluster of workstations [1].

Choosing the best parallel programming paradigm is actually an imperative concern when it comes to parallelization of an application or algorithm. There are a few parallel programming paradigms available such as MPI, OpenMP and Parallel Virtual Machine (PVM). We have chosen MPI as the paradigm of choice due to the nature of our problem, the hardware components and the network setup that we have in the laboratory [3]. MPI consists of specifications for message passing libraries that can be used to write parallel programs. This message passing paradigm not only can be employed within a node but also across several nodes in a cluster. This is the advantage of MPI over OpenMP. Unlike OpenMP, MPI is also viable for wide range of problems. Besides that, MPI offers the user’s complete control over data distribution and process synchronization. This feature is vital in order to ensure optimum performance of the parallelization. PVM may be more suitable for heterogeneous network setup and although MPI does not
have the concept of a virtual machine, MPI does provide a higher level of abstraction on top of the computing resources in terms of the message-passing topology.

The resulting implementation is tested on High Performance Computing (HPC) architecture that is made of Beowulf-style computing cluster. The parallel program designed caters for 50 cities or points.

Due to its famous nature, many literatures have existed in providing solutions to solve the TSP problem. However, only few references can be found on parallel implementations of the TSP [4-7]. The main difference of these works with ours is the choice of parallel programming paradigm.

## 2 The Sequential Solution

This section explains our design of sequential solution.

### 2.1 The sequential algorithm

The algorithm of the sequential program is as follows:

**Start**

- Open file and get the input for the coordinates of a node
- Initialize the source and destination nodes
- Initialize dynamic 2D array
- Compute for all possible path using permutation algorithm and stores in the dynamic 2D array
- Compute for the distance for all possible paths
- Compare the distance to find the shortest path
- Display shortest distance and shortest path

**End**

The sequential program begins by getting input from a text file (.txt) that holds the coordinates of all the nodes. Default value is used for the destination and source node. A dynamic 2D array is created and all computed possible paths are stored in it. The distances for all possible paths are calculated and the shortest distance is determined. The program then displays the shortest distance and the shortest path. The total number of all possible paths can be calculated by using simple factorial method. The number of nodes must first be defined. Later the number of possible path shall be defined using the formula below:

\[ \text{Number of possible paths} = n! \]

Where, \( n = \text{number of cities} \)

Table 1 shows the numbers of possible paths derived from this formula.

<table>
<thead>
<tr>
<th>Number of cities</th>
<th>( n! )</th>
<th>Number of possible path(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3!</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4!</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>5!</td>
<td>120</td>
</tr>
<tr>
<td>6</td>
<td>6!</td>
<td>720</td>
</tr>
<tr>
<td>10</td>
<td>10!</td>
<td>3628800</td>
</tr>
<tr>
<td>50</td>
<td>50!</td>
<td>( 3.04 \times 10^{64} )</td>
</tr>
</tbody>
</table>

### 2.2 Array of possible paths

The program uses dynamic 2D arrays. That is, using `calloc` function to create a table that contains the nodes that represent all possible paths from one source node to a destination node. The number of rows and columns is equal to the number of possibilities calculated and the number of processing nodes defined respectively.

The program will fill the first column of every row and the destination node will fill the last column of every row. The in between cells of the array will be filled up with all other possible nodes generated from the permutation function. For example, let’s take 3 cities; the number of possibilities will be 3! equivalent to 6. Assuming the source node is 0 and destination node is 2.
pointer to pointer variable **poss_array in master will point to an array of pointers that subsequently point to a number of rows; this makes up a table of dynamic 2D array. The number of rows and columns of the array are both defined by the number of possibilities and the number of cities respectively. After the table of dynamic 2D array is created, master will then compute and fills in all possible paths in the array. This process uses the permutation function to compute all possible paths. This idea is illustrated in Fig. 1.

**poss_array

poss_array [0] poss_array [0][1] poss_array [0][2] poss_array [0][3] poss_array [0][n]


poss_array [n]

Fig. 1: Master creates dynamic 2D array

The parallel segment begins when Master broadcasts the dynamic 2D array to all nodes by using MPI_Bcast. The Master will then equally divide the rows by the number of slaves available in the grid cluster. Each slave is given an equal number of rows to compute and find the distance for the shortest path. Each slave will be receiving \( n \) numbers of rows to be computed and this is where the parallel processing takes place. The slaves will process each row given concurrently, where each slave will find the shortest distance and shortest path for the all rows received. After the slaves have processed all the rows, it will return the results of the shortest distance and shortest path computed to the Master. The master will then compare all the results from the slaves to determine the shortest distance and shortest path.

Let’s take the previous example where there are 3 cities and 6 possibilities. Therefore the dynamic 2D array should have 6 rows and 3 columns. Assuming that there are 3 slaves available to execute the task, therefore when Master divides the number of rows with the number of slaves, each slave will compute 2 rows. The overall process is depicted in Fig. 2.

<table>
<thead>
<tr>
<th>Number of rows: 6</th>
<th>Number of column: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>row[0] 1 2 3</td>
<td></td>
</tr>
<tr>
<td>row[1] 1 3 2</td>
<td></td>
</tr>
<tr>
<td>row[2] 2 1 3</td>
<td></td>
</tr>
<tr>
<td>row[3] 2 3 1</td>
<td></td>
</tr>
<tr>
<td>row[4] 3 1 2</td>
<td></td>
</tr>
<tr>
<td>row[5] 3 2 1</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2: Example of assigning 6 rows to 3 slaves

Slave 1 will be processing row [0] up to row [1], slave 2 will be processing row [2] up to row [3] and lastly slave 3 will be processing row [4] up to the last row, row [5]. After each slave returns the shortest distance and shortest path to Master, Master will then compare all the results and determine the shortest distance and shortest path. The program will then display the shortest distance and shortest path calculated.

Below is the outline of the parallel algorithm of TSP.

**Begin algorithm**

**Master part**

Calculate the number of possible path to determine the number of rows

Generates dynamic 2D array, where all elements are the possible path generated from permutation algorithm

Broadcasts the dynamic 2D array to all slaves

Divides the number of rows with the number of slaves

Send the \( n \) rows to each slave

Proceeds with sequential part

**Slaves part**

Receive the dynamic 2D array from Master

Receive \( n \) rows to be computed

Calculate shortest distance and shortest path

Send results to Master

**End algorithm**
The overall process can be summarized in figure 3.

## 4 Results and Discussion

### 4.1 Experimental setup

The experiment was conducted on a Beowulf cluster consists of 20 SGI machines. Each of the machines consists of off-the-shelf Intel i386 based dual P3-733MHz processors with 512MB memory Silicon Graphics 330 Visual Workstations. These machines are connected to a Fast Ethernet 100Mbps switch. The head node performs as master node with multiple network interfaces. Although these machines are considered to be superseded in terms of hardware and performance as compared to the latest version of high performance computers, what’s important in this research is the parallelization of the algorithm and how jobs are disseminated among processors.

### 4.2 Results

Table 2 depicts the performance of the parallel implementation when 1, 2, 3, 5, 10 and 32 nodes are used to calculate the distances between 12 cities.

<table>
<thead>
<tr>
<th>Number of processors/nodes</th>
<th>Execution Time (seconds)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17+</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>11+</td>
<td>1.54</td>
</tr>
<tr>
<td>3</td>
<td>9+</td>
<td>1.89</td>
</tr>
<tr>
<td>5</td>
<td>7+</td>
<td>2.43</td>
</tr>
<tr>
<td>10</td>
<td>6+</td>
<td>2.83</td>
</tr>
<tr>
<td>32</td>
<td>5+</td>
<td>3.4</td>
</tr>
</tbody>
</table>

Based on Table 2, it can be inferred that increased number of processors results in faster execution time. However, there is latency issue if Table 2 is analyzed carefully. Observe that the difference between the first two processes is around 6 seconds. Whereas the difference between the last two processes is only around 1 second. That is, although the number of processors involved is increasing, but the difference between the execution time is decreasing. This is attributable to the communication latency between the master and slaves in performing the computation. It is also observed that the optimal performance for this test case is when using five processors. This is due to the fact that the significant difference in time is between processor one and five.

### 4.3 Experimental platform limitation

The Master node and all the slave nodes in the cluster have its own memory limited to 512MB. During the execution of the parallel program, the possibility table is generated by the Master node. It will then send the pointer of that table to the other slave nodes to compute the shortest path.

Table 3 shows the number of possibilities or the number of rows in the 2D dynamic array.

<table>
<thead>
<tr>
<th>Number of cites/points</th>
<th>Number of possibilities/rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>11! = 39,916,800</td>
</tr>
<tr>
<td>12</td>
<td>12! = 479,001,600</td>
</tr>
</tbody>
</table>

After the number of rows and columns are determined, a dynamic 2D array is created in Master’s main memory and bear in mind the capacity of the main memory is only 512MB. Each element of the array uses 4 bytes to store an integer value. The size of the array for any number of points can be calculated by multiplying the number of rows ($n$) and columns ($m$) to find the number of elements in the array. The result is then multiplied by the size of an integer which is 4 bytes. Therefore, the size of the array for both 11 cities and 12 cities are as below:

- (11 cols x 39916800 rows) x 4 bytes = 159667200 bytes $\approx$ 159 MB
- (12 cols x 479001600 rows) x 4 bytes = 1916006400 bytes $\approx$ 1.9 GB

From the above calculation, it shows that the size of 2D array for 11 cities needs approximately 159 MB of
space per execution time. This means it can be easily created by the master with the 512 MB main memory. Whereas the size of the 2D array for 12 cities needs around 1.9 GB which exceeds the capacity of the Master’s main memory of 512MB. This limitation is observed during the testing stage of the study. The program was developed to cater for 50 points, however Master node do not have enough memory to fit the 2D dynamic array.

Since there is a limitation in the Master’s memory space in our experimental setup, therefore the parallel program can only execute up to 11 cities. In a nutshell, shared memory cluster architecture would be able to portray properly the true remuneration that can be gained from parallelism of the algorithm.

5 Conclusion
In this paper we have presented a parallel implementation of solving Traveling Salesman Problem (TSP). The nature of TSP and the functionalities offered by MPI have made it possible to convert the TSP sequential algorithm to parallel algorithm. The resulting implementation has also demonstrated that it is viable approach and has led to increased execution time of the algorithm. It also has shown some limitations as we increased the number of processors and this will be further investigated in the future work.

References:
[6] Ling Chen, Hai-Ying Sun, & Shu Wang, Parallel implementation of ant colony optimization on MPP.