Solving Traveling Salesman Problem on Cluster Compute Nodes

IZZATDIN A. AZIZ, NAZLEENI HARON, MAZLINA MEHAT,
LOW TAN JUNG, AISYAH NABILAH MUSTAPA, EMELIA AKASHAH PATAH AKHIR
Computer and Information Sciences Department
Universiti Teknologi PETRONAS
31750 Tronoh, Perak
MALAYSIA
{izzatdin, nazleeni, mazlinamehat, lowtanjung, emeliaakashah}@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. It is a typical NP-complete problem that has received great attention in research and teaching.

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.

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 [1]. MPI was chosen due to the fact it is designed for high performance computing on parallel machines or cluster of workstations [2]. The message-passing model consists of a number of processors, any pair of which can communicate with each other by exchanging messages via communication link(s).

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. Some other features of OpenMP that are not in our favor include: OpenMP only runs efficiently in shared-memory multiprocessor platforms as proposed in [4], it lacks the reliable error handling capabilities, scalability in OpenMP is limited by node memory architecture, and synchronization between a subset of threads is not allowed. Unlike OpenMP, MPI is found to be more viable for wide range of problems and it 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 Methodology

Traditional approach of system development methodology that needs to get the development model mostly correct in the early stage is impossible as this involves more than just one area of studies such as prime number generation algorithm, primality tests, parallel processing and MPI. Various issues need to be considered that may be unforeseen at the beginning stage of development. Thus different conditions and techniques would involve during development phase.

Evolutionary development is an iterative and incremental approach for system development. The system will be delivered incrementally over time. Evolutionary development is new to many existing professional developer and many traditional programmers as well. Fig. 1 illustrates the phases involved in evolutionary development approach [8].

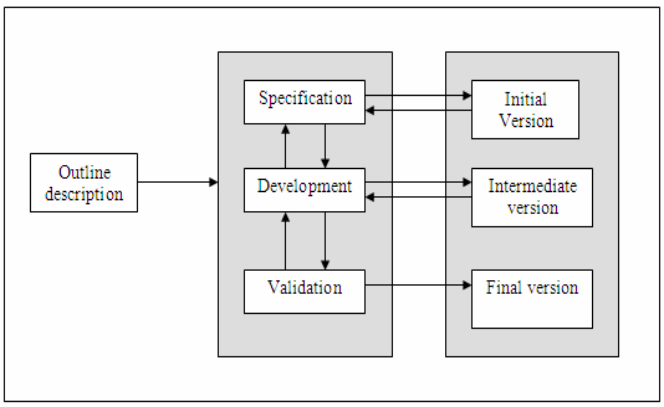

Fig. 1: Phases involved in Evolutionary Development Approach

2.1 Specification Phase

A sequential program of prime number generation in C using MPI libraries is developed. Then in this phase the parts of the sequential program that could be parallelized would be identified. This is the beginning of the specification phase. Although the main objective is to parallelize the prime number generation, but not all part of the program can be parallelized. This is where the partitioning stage of the programming design takes place which is intended to explore the opportunities for parallel execution.

2.2 Development Phase

As mentioned earlier, the parallelization of the algorithm was achieved by using MPI libraries. The parallel program was written incrementally over time which means troubleshooting was done on the program from time to time to avoid error that could not be debugged later on.

2.3 Validation Phase

The program prototype will then go through the validation phase to ensure the project requirements are achieved. If there are still areas that need to be modified and altered, the whole phases will be repeated all over again until the final version of the program is released. Most of the evaluation processes were carried out by the authors.

3 Development Tools

The main reason of choosing C to write the program is because it provides an sequential infrastructure that accommodates mechanism of breaking down the problem into a collection of data structures and operations that is matching the characteristic of parallel processing.

Furthermore, C is also compatible with the concept of partitioning and dynamic memory allocation which, are the concept that is going to be deployed in the parallelization of prime number generation. As mentioned earlier, MPI is used for the parallel processing of the algorithm; a library of subroutine specifications that can be called from C , this is also another reason why the parallel program is written using C. The application that is used to edit the program is Linux gnu[8].

3.1 Libraries

MPI provides all the subroutines that are needed to break the tasks involved in the massive computational process into subtasks that can be distributed to a number of available nodes for processing. The goal of the MPI is to establish a portable, efficient, and flexible standard for

message passing that will be widely used for writing message passing programs. MPI provides an appropriate environment for general purpose message-passing programs, especially programs with regular communication patterns. Fig. 2 shows the general MPI program structure:

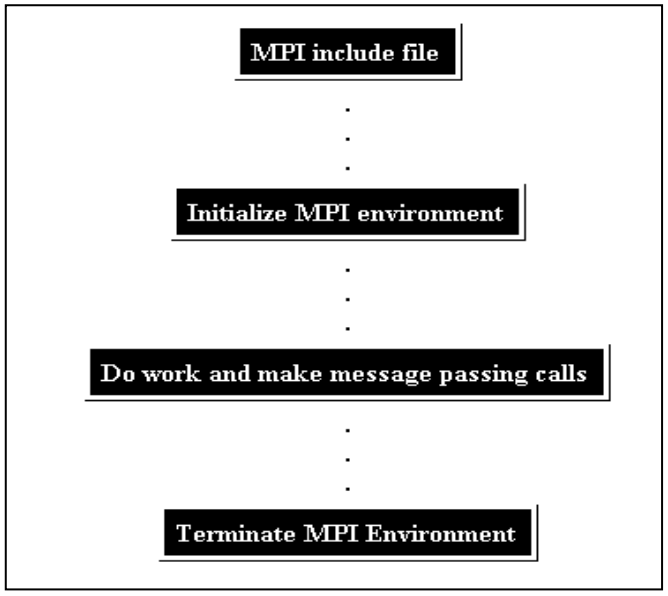

Fig. 2: General MPI Program Structure [8].

MPI contains approximately 125 functions that greatly ease the tasks in implementing common communication structures, such as send-receive, broadcasts and reductions. However, MPI is reasonably easy to learn as a complete message-passing program can be written with just six basic functions.

MPI contains useful communications libraries for applications that need to be ported to various platforms. Different versions of MPI exist for virtually every major platform: message-passing supercomputers, scalable shared-memory machines, symmetric multiprocessors, loosely-coupled workstation clusters, and even individual PCs. With MPI, the programmer can write code *once* and merely recompile it for each new platform.

3.2 Experimental Testbed

Fig 3 shows the experimental cluster set up in the UTP lab which comprised 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 [9]. Although these machines may not be as powerful as the latest cluster-machine in terms of the hardware and performance, the important focus would be the parallelization of the algorithm and how jobs can be disseminated among the processors.

Fig.3 UTP Cluster [9]

The software stack on all machines is consisting of Linux Ubuntu 5.10 operating system, MPICH-1.2.7p1 and openMosix for Kernel 2.4.26 stable cluster middlewares, parallel High Performance Linpack (HPL) version 1.0a and Flops.c version 2.0 both for parallel benchmark and individual node flops benchmark, GCC-3.3.6 with Basic Linear Algorithm Subroutine (BLAS) version 3.0 as the program compiler and its supporting math library, and lastly is the MPI communication benchmark using mpptest (part of perftest version 1.3b).

The reasoning why we run only HPL C version is by the assumption that the majority of application programs are based on C programming language rather than other programming languages in our implementation [9].

4 The Sequential Solution

This section explains our design of sequential solution. However the serial brute-force algorithm proposed by [10-12] is stated here again for comparison purpose to the authors' algorithm.

Input n: the number of cities,
C : the costmatrix
Output shortesttour

begin

min := infinity;
for all cyclic permutations pi of {1, 2,..., n} do
cost := o;
for i:= 1to n do cost:= cost+ C[i,pi(i)];
*/*here pi(i) is the ith elementof pi*/*
if cost< min then min:= cost; besttour:= pi
output besttour,

end,

The proposed sequential algorithm 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:

Number of possible paths = n!
Where, n = number of cities

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

Table 1: Total number of all possible paths

Number of cities	n!	Number of possible path(s)
3	3!	6
4	4!	24
5	5!	120
6	6!	720
10	10!	3628800
50	50!	3.04×10^{61}

4.1 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.

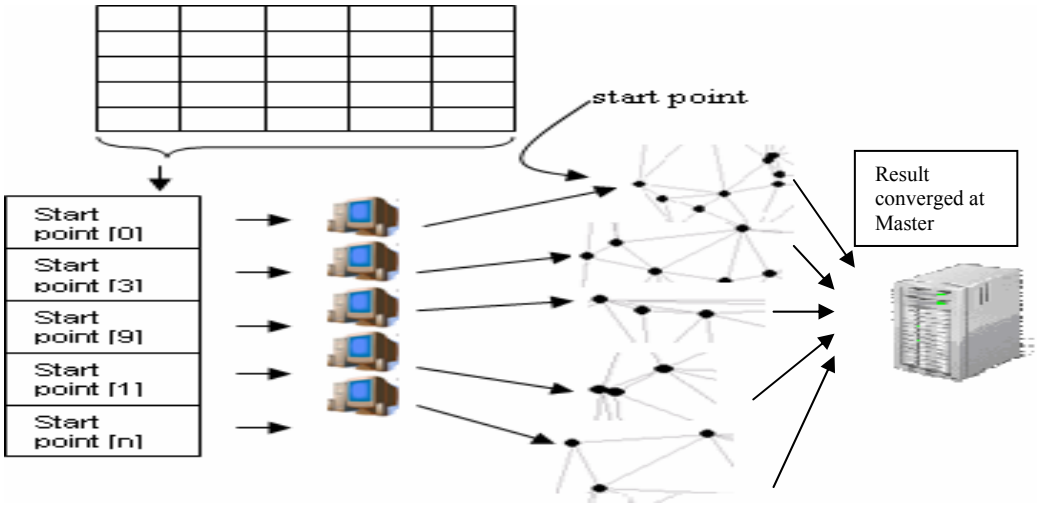

Fig. 4: Dynamic 2D array filling process

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. To further explain the array filling process, refer to the Fig. 4.

5 The Parallel Solution

This section illustrates our design on the parallelization of traveling salesman problem.

5.1 The parallel programming paradigm

We are using the master-slave parallel paradigm for this type of problem. The Master is responsible for dividing the task amongst the other processors called the slaves. All the slaves execute the task given concurrently. In this case the task is to find the distance for each path. The slaves will return the results to the master once they have calculated it. The master will then determine the shortest distance after it has received all the results from all slaves.

5.2 Master-Slave Architecture

The pseudocode mentioned in next subsection was implemented on the Master-Slave HPC architecture. In this setup, the master node acts as the coordinator in terms of load distribution to the other nodes and eventually gathers and stores all the processed data. The slave nodes primary task is to receive the input from the master node and execute the codes destined for the slave nodes. Each of the slave nodes receive one shot record at a time and the entire process depicted in Figure 4 were executed by the slave nodes. The illustration of the architecture is as in Figure 5 below.

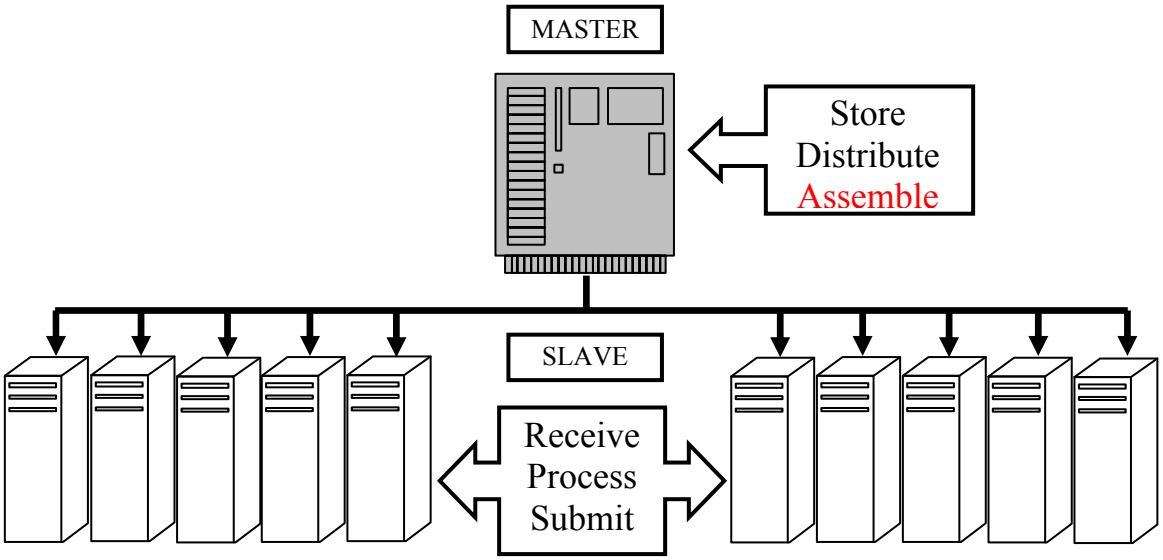

Fig 5: Master-Slave architecture

5.3 The parallel algorithm

The algorithm of the parallel program is outlined as follows:

Start

Master open file and get the input for the coordinates of a node

Master initializes the source and destination node
Master initialize dynamic 2D array
Master compute for all the possible path using permutation algorithm and stores in the dynamic 2D array
Master divide the number of possible path (rows) with the number of processors

Master sends the number of rows to each slave
Each slave will receive an initialized row from master
Each slave will compute for the distance for all possible paths
Each slave will compare the distance to find the shortest path
Each slave will return shortest distance and shortest path to Master

Master waits for results from slaves
Master receives shortest distance and shortest path from each slave
Master compares the shortest distance hence finds the shortest path

Master display shortest distance and shortest path

End

When the code runs on the grid cluster, master will create a table of dynamic 2D array that later populates all the possible paths. As the number of nodes increases, the number of possible path would increase excessively. Therefore, it is significant to use the dynamic array that can easily expand to a very large size and only takes memory spaces that it needed.

A 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. 6.

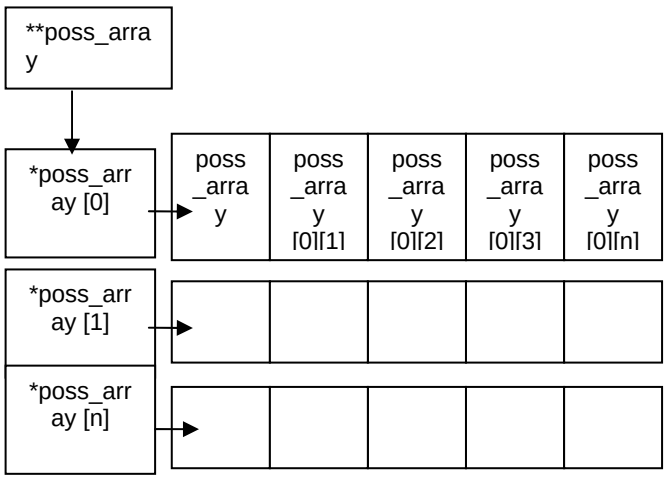

Fig. 6: Master creates dynamic 2D array

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. 7.

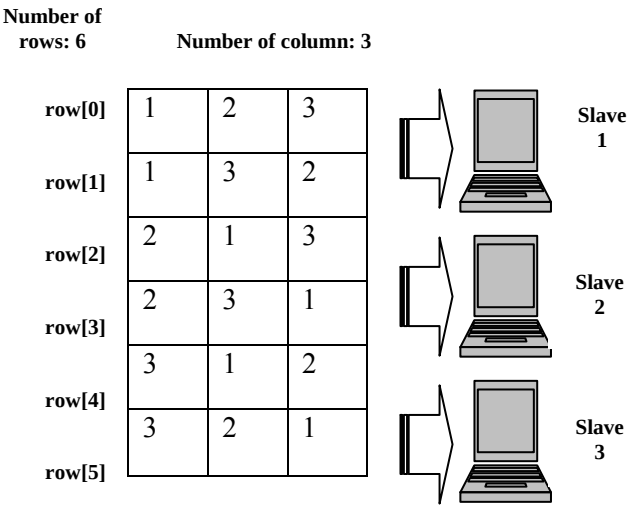

Fig. 7: Example of assigning 6 rows to 3 slaves

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

6 Results and Discussion

6.1 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 10 cities.

Table 2: Results of parallel execution time for 10 cities

Number of processors/nodes	Execution Time (seconds)
1	17+
2	11+
3	9+
5	7+
10	6+
32	5+

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.

The main goal we want to reach with parallelization is to gain a good speedup. A good speedup means to be nearly *n* times faster with *n* processors.
Speedup is the ratio between sequential execution time and parallel execution time and can be calculated using the formula below:

$$Speedup = \frac{sequential\ \ execution\ \ time}{parallel\ \ execution\ \ time}$$

Sequential execution time is the time taken for a processor to perform the required computation. Parallel execution time is capturing the time taken when master starts to divide the tasks until it receives the last result from the slave.

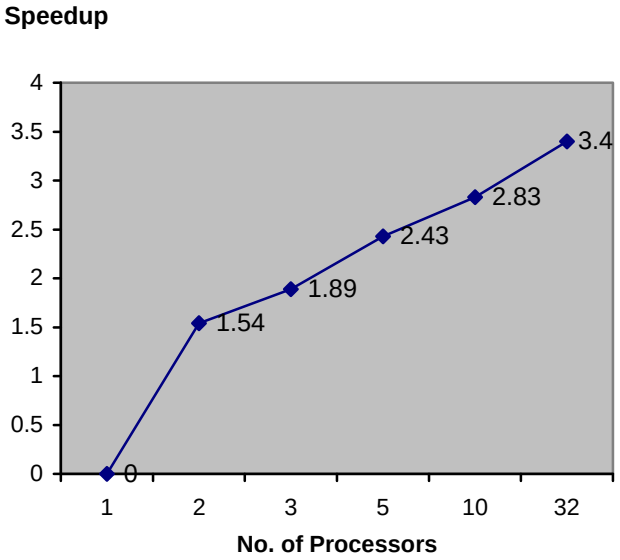

Fig. 8 Speedup for the algorithm

Fig. 8 shows the speedup obtained for the proposed parallel algorithm with various numbers of processors. It can be seen that the gap between each speedup is getting smaller as the number of processors increases. Therefore to determine the cause of such speedup we use The Karp-Flatt Metric [x] which is called experimentally determined serial fraction, e and calculated using formula below:

$$e = \frac{\frac{1}{speedup} - \frac{1}{no._of_processors}}{1 - \frac{1}{no._of_processors}}$$

Fig. 9 depicts the serial fraction obtained for each processor and it shows that the experimentally determined serial fraction is steadily increasing as the number of processors increases. Based on the [x], it can be inferred that the principal reason for smaller gap in speedup is due to parallel overhead. The parallel overhead is actually due to time spent in process startup, communication and synchronization between the master and the slaves.

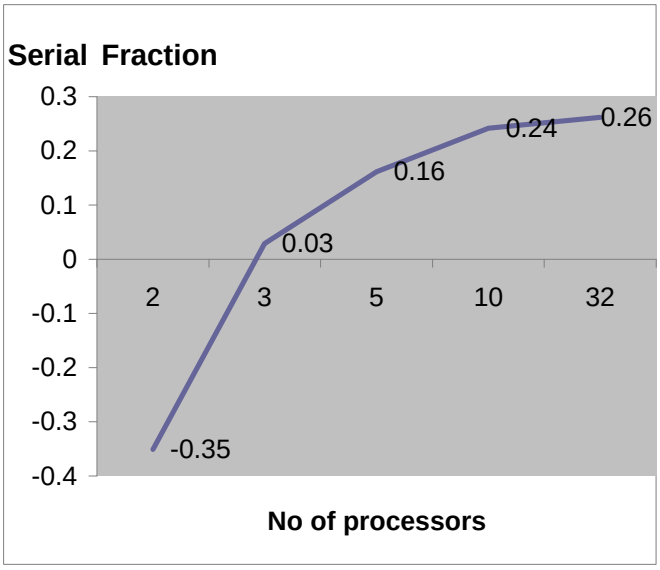

Fig. 9 Experimentally Serial Fraction of the Parallel Program

Fig. 10 presents the efficiency of the 32 processors in solving the parallel algorithm. The efficiency of a parallel program is a measure of processor utilization and is calculated using the formula below:

$$Efficiency = \frac{Sequential\ execution\ time}{processors\ used \times Parallel\ execution\ time}$$

It has been observed that the efficiency decreases as the number of processors are increased. This is because as the more processors involved in performing computation, the less task was assigned to each processor. Therefore, to maintain the same level of efficiency for each processor, problem size should be increased as the number of processors increased.

A possible solution to this problem is to derive an algorithm by incorporating a proper scheduling technique. Having this, job can be decomposed effectively, hence allowing greater efficiency

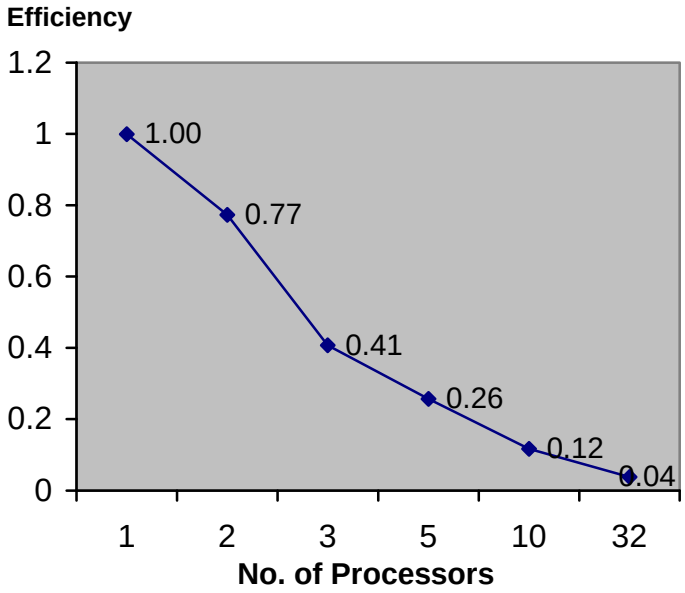

Fig. 10 Efficiency of the Parallel Algorithm

6.2 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 3: Number of possibilities/rows for 11 and 12 cites

Number of cites/points	Number of possibilities/rows
11	11! = 39, 916, 800
12	12! = 479, 001, 600

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 ≈ 159 MB

(12 cols x 479001600rows) x 4 bytes = 1916006400 bytes ≈ 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 Recommendation

As suggested in [13], performance of a cluster can significantly be improved by using a Generalized Shared Memory, which is maintained in a consistent state by a hardware-based coherency mechanism that operates on shared objects, wherever they happen to be located. This increases both the performance and the versatility of the architectures by permitting the composition of private vs. shared memory to be of arbitrary size and dynamically variable on different computer nodes in the cluster.

Efficiency of the algorithm can also be improved by executing it on a cluster with better interconnects as suggested in [14]. A thorough study is recommended to investigate the suitable interconnects to execute the algorithm in order to yield optimum result.

6 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. The speed up shows an increased in processing time however the efficiency measured at a declined rate, this is possibly due to the network and communication latency among processors or compute nodes. It also has shown some limitations as we increased the number of processors and this will be further investigated in the future work.

References:

[1] Selim G Aki, Stefan D Bruda, Improving A Solution's Quality Through Parallel Processing. *The Journal of Supercomputing* archive.Volume 19 , Issue 2 (June 2001).

[2] MPI Retrieved on May, 17 2008 from http://www-unix.mcs.anl.gov/mpi/

[3] Dani Adhipta, Izzatdin Bin Abdul Aziz, Low Tan Jung, Nazleeni Binti Haron .Performance Evaluation on Hybrid Cluster: The Integration of Beowulf and Single System Image, *Proceedings of ICTS*,Jakarta. August 2006.

[4] Delisle P., Krajecki M. et al, "Parallel implementation of an ant colony optimization metaheuristic with OPENMP", *Proceedings of the 3rd European Workshop on OPENMP*, Spain, Sep, 2001

[5] Baraglia, R., Hidalgo, J. I., & Perego, R. A parallel hybrid heuristic for the TSP. In Proceedings of EvoCOP2001*, the First European Workshop on Evolutionary Computation in Combinatorial Optimization,* 193-202. 2001.

[6] Ling Chen, Hai-Ying Sun, & Shu Wang, Parallel implementation of ant colony optimization on MPP. *Proc. International Conference on Machine Learning and Cybernetics,* Boading, 2008.

[7] Tschoke, S., Luling, R., & Monien, B. (1995). Monien: Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 Processor Network. *Proc. 9th Int. Parallel Processing Symp. (IPPS '95)*, 182-189.

[8] Izzatdin Aziz, Nazleeni Haron, Low Tan Jung, WAN Rahaya Wan Dagang (2007)"Parallelization of Prime Number Generation Using Message Passing Interface" WSEAS Journal Transaction of Computers Volume 7 2008 ISSN: 1109-2750.

[9] Dani Adhipta, Izzatdin Bin Abdul Aziz, Low Tan Jung, Nazleeni Binti Haron .Performance Evaluation on Hybrid Cluster: The Integration of Beowulf and Single System Image, Journal PLATFORM Volume 5 Number 1 January –June 2007, ISSN 1511-6794.

[10] Yang L., Jin L., Integrating Parallel algorithm Design With Parallel Machine Models, *ACM SIGCSE Bulletin, Vol. 27, Issue 1, Pg: 131 – 135,* March 1995.

[11] Manohar R., Zary S., Dalibor V., Implementation Machine Paradigm For Parallel Programming. In proceedings of *ACM/IEEE 1990 Conference On Super-computing*, Nov 1990.

[12] A.J Sanchez Santiago,A.J Yuste, J.E Munoz Exposito, S Garcia Galan, J.M Maqueira Marin, S Bruque, "A Dynamic Balanced Scheduler for Genetic Algorithms for GRID Computing" Journal WSEAS Transaction on Computers, Issue 1 Volume 8, ISSN **1109-2750**

[13] Free Patents Online "Generalized shared memory in a cluster architecture for a computer system" <Access date: 16 March 2009> http://www.freepatentsonline.com/EP0603801.html.

[14] Weikuan Yu, Ranjit Noronha, Shuang Liang, Dhabaleswar K.Panda "Benefis of High Speed Interconnects to Cluster File Systems:A Case Study with Lustre" IEEEXplore, <Accessed Date: 16 March 2009> http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1639564