SERIAL AND PARALLEL IMPLEMENTATION OF
SHORTEST PATH ALGORITHM IN THE OPTIMIZATION
OF PUBLIC TRANSPORT TRAVEL

Guruprasad Nagraj¹ & Dr. Y S Kumaraswamy²
¹ Research Scholar Dept. of Computer Science, Dr. MGR Deemed University, Maduravoyal, Chennai, INDIA ngp_jyothi@yahoo.co.in
² Professor & Head, Dept of Master of Computer Applications, Dayanand Sagar College of Engineering, Bangalore, INDIA yskldswamy@yahoo.co.in

ABSTRACT

Traffic congestion is becoming a serious problem in more and more modern cities. Encouraging more private vehicle drivers to use public transportation is one of the more effective and economical ways to reduce the ever increasing congestion problem on the streets (Hartley and Bargiela 2001). With the research and application of Intelligent Transportation System and the popularization of dual-core computer, there is a higher requirement for solving the shortest path algorithm in large scale transportation networks in real-time by using multi core technology.

This paper basically analyzes the performance of the program execution in sequential and parallel way in multi core machines where in the algorithms designed were executed for large set of nodal points (upto 500) where each nodal point basically represents a source or destination or it can be even a transit point between any source and destination with respect to public transport of Bangalore Metropolitan Transport Corporation.

Keywords: Shortest path, Multi core, Public transport, Congestion problem, Transit point.
1. INTRODUCTION

With the increase of private vehicles on the streets more and more congestion occurs. This traffic congestion not only causes a monetary problem but also a pollution problem at the same time. (Peytchev 1999) Policy debates promoted by publication of the Transport White Papers at UK and Scottish levels, have identified the need to reduce the number of private vehicles journey and encourage them to go for public transport usage (Hine and Scott 2000). As public transport services become more popular, the users of public transport need route information to help them plan journeys in a much better and more efficient manner. One of the most important piece of information to be delivered to public transportation users is quick bus route(s) service between their source and destination. In our case quick actually means less distance to be travelled.

A shortest path problem is for finding a path with minimum travel cost and distance from one or more origins to one or more destinations through a connected network. It is an important issue because of its wide range of applications in transportations. In some applications, it is also beneficial to know the next shortest path between any two nodes. For instance, in order to improve the effectiveness of travel information provision, there is a need to provide some rational alternative path (route) for users of public transport.

In this paper we are focusing on understanding the multi core processing and programming methods. And applying the same to the three well known algorithms (Dijkstra’s, Bellman Ford and Ant Colony) to solve for public transport travel. We have produced a comparative study by monitoring and analyzing the performance both in sequential and parallel way.

2. MULTI CORE PROGRAMMING ARCHITECTURE

Processors were originally developed with only one core. The core is the part of the processor that actually performs the reading and executing of
instructions. Single-core processors can process only one instruction at a time. (To improve efficiency, processors commonly utilize pipelines internally, which allow several instructions to be processed together; however, they are still consumed into the pipeline one at a time.)

A multi-core processor is composed of two or more independent cores. One can describe it as an integrated circuit which has two or more individual processors (called cores in this sense). Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP), or onto multiple dies in a single chip package. A many-core processor is one in which the number of cores is large enough that traditional multi-processor techniques are no longer efficient largely due to issues with congestion supplying sufficient instructions and data to the many processors. This threshold is roughly in the range of several tens of cores and probably requires a network on chip.

### 3. SHORTEST PATH ALGORITHM

Efficient management of networks requires that the shortest route (path) from one point (node) to another is known; this is termed as shortest path. Is is quite often necessary to be able to determine alternative routes through the network, in case any part of the shortest path is damaged or busy. The analysis of transportation networks is one of many application areas in which the computation of shortest paths is one of the most fundamental problems. These have been the subject of extensive research for many years. The shortest path problem was one of the first network problems studied in terms of Operations Research.

If one represents a nondeterministic abstract machine as a graph where vertices describe states and edges describe possible transitions, shortest path algorithms can be used to find an optimal sequence of choices to reach a certain goal state, or to establish lower bounds on the time needed to reach a given state.
Suppose a motorist wishing to drive from city A to city B would be interested in answers to the following questions:

1) Is there a path from A to B?

2) If there is more than one path from A to B, which is the shortest path?

A shortest path algorithm is a program, or set of directions that can be executed to provide the shortest path between locations given certain conditions and paths. Conditions such as traffic density, speed of travel, and others, as well as geographic obstacles can be factored in to help the algorithm execute and display the shortest path. The latest algorithms being developed adjust to conditions and dynamically execute to give new shortest paths based not only on distance, but also on time.

Following algorithms have been implemented in our work:

1) Dijkstra’s algorithm
2) Bellman Ford algorithm
3) Ant colony algorithm

3.1 Dijkstra’s algorithm

Dijkstra’s algorithm (named after its discover, E W Dijkstra) solves the problem of finding the shortest path from a point in a graph (the source) to a destination. It turns out that one can find the shortest paths from a given source to all points in a graph in the same time, hence this problem is sometimes called the single-source shortest paths problem.

The single source shortest path problem can be described as follows:

Let G= \{V, E\} be a directed weighted graph with V having the set of vertices. The special vertex s in V, where s is the source and let for any edge e in E, EdgeCost(e) be the length of edge e. All the weights in the graph should be
non-negative. Before going in depth about Dijkstra’s algorithm let’s talk in detail about directed-weighted graph.

Directed graph can be defined as an ordered pair \( G = (V,E) \) with \( V \) is a set, whose elements are called vertices or nodes and \( E \) is a set of ordered pairs of vertices, called directed edges, arcs, or arrows.

The following pseudo-code gives a brief description of the working of the Dijkstra’s algorithm.

**Procedure**

Dijkstra \( (V: \text{set of vertices 1... n} \{\text{Vertex 1 is the source}\}) \)

\( \text{Adj[1…n]} \) of adjacency lists;

\( \text{EdgeCost(u, w)}: \text{edge – cost functions;} \)

\textbf{Var:} \( \text{sDist[1…n]} \) of path costs from source (vertex 1);

\{\( \text{sDist[j]} \) will be equal to the length of the shortest path to \( j \)\}

**Begin:**

**Initialize**

\{Create a virtual set Frontier to store \( i \) where \( \text{sDist}[i] \) is already fully solved\}

Create empty Priority Queue \( \text{New Frontier}; \)

\( \text{sDist[1]} ← 0; \) \{The distance to the source is zero\}

\textbf{forall} vertices \( w \) in \( V – \{1\} \) \{no edges have been explored yet\}

\( \text{sDist}[w] ← \infty \)

\textbf{end for;}

Fill \( \text{New Frontier} \) with vertices \( w \) in \( V \) organized by priorities \( \text{sDist}[w]; \)

**endInitialize;**

**repeat**

\( v ← \text{DeleteMin}[\text{New Frontier}]; \) \{\( v \) is the new closest; \( \text{sDist}[v] \) is already correct\}
forall of the neighbors w in Adj[v] do
if sDist[w] > sDist[v] + EdgeCost(v, w) then
sDist[w] ← sDist[v] + EdgeCost(v, w)
update w in New Frontier {with new priority sDist[w]}
endif
endfor
until New Frontier is empty
endDijkstra;

3.2 Bellman Ford algorithm

The Bellman–Ford algorithm computes single-source shortest paths in a weighted digraph. For graphs with only non-negative edge weights, the faster Dijkstra’s algorithm also solves the problem. Thus, Bellman–Ford is used primarily for graphs with negative edge weights. The algorithm is named after its developers, Richard Bellman and Lester Ford Jr.

procedure BellmanFord(list vertices, list edges, vertex source)

This implementation takes in a graph, represented as lists of vertices and edges, and modifies the vertices so that their distance and predecessor attributes store the shortest paths.

Step 1: Initialize graph

for each vertex v in vertices:
    if v is source then v.distance := 0
    else v.distance := infinity
    v.predecessor := null

Step 2: relax edges repeatedly
for i from 1 to size(vertices)-1:
  for each edge uv in edges:
    u := uv.source
    v := uv.destination // uv is the edge from u to v
    if v.distance > u.distance + uv.weight:
      v.distance := u.distance + uv.weight
      v.predecessor := u

Step 3: check for negative-weight cycles
  for each edge uv in edges:
    u := uv.source
    v := uv.destination
    if v.distance > u.distance + uv.weight:
      error "Graph contains a negative-weight cycle"

3.3 Ant colony algorithm

In computer science and operations research, the ant colony optimization algorithm (ACO) is a probabilistic technique for solving computational problems which can be reduced to finding good paths through graphs.
Pseudo – Code

In fact the algorithm uses a set of artificial ants (individuals) which cooperate to the solution of a problem by exchanging information via pheromone deposited on graph edges. The ACO algorithm is employed to imitate the behaviour of real ants and is as follows:

```pseudo
procedure shortestPath(in: previousNode, in: currentNode, in/out: path)
    add currentNode to end of path
    if current node is food
        return
    else
        greatestPheromoneTrail=0
        nextNode=null
        for all arcs connected to currentNode, except the one connected to previousNode
            if this arc’s pheromone trail > greatestPheromoneTrail
                greatestPheromoneTrail = this arc’s pheromone trail
                nextNode = node at the end of this arc
            end if
        end for
        shortestPath(currentNode, nextNode, path)
    end if
end procedure
```
4. IMPLEMENTATION

Open Multi Processing API is been used in the code to support multi-platform shared memory multiprocessing programming in C & C++. We have used set of compiler directives from the Open MP, which influence the run-time behaviors.

Few parallel construct used in the program –

#pragma omp for - Distribute Iterarions over threads
#pragma omp sections - Distribute independent work units
#pragma omp single - Only one thread executes the code block.

In C & C++ we have to include the header file omp.h to use the above mentioned directives. Sequential code has been written without using the open mp compiler directives whereas parallel code has been used the open mp compiler to distribute the task to process parallel in form of threads.

5. TESTING

Below tabulation & graph has been derived has part of test result in the different set of node –

Dijikstra Algorithm Tabulation

<table>
<thead>
<tr>
<th></th>
<th>Parallel Execution - 4 core</th>
<th>Parallel Execution - 2 core</th>
<th>Sequential Execution - Single Core</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>0.016</td>
<td>0.016</td>
<td>0.017</td>
</tr>
<tr>
<td>200</td>
<td>0.026</td>
<td>0.031</td>
<td>0.027</td>
</tr>
<tr>
<td>300</td>
<td>0.047</td>
<td>0.047</td>
<td>0.049</td>
</tr>
<tr>
<td>400</td>
<td>0.067</td>
<td>0.074</td>
<td>0.067</td>
</tr>
<tr>
<td>500</td>
<td>0.075</td>
<td>0.074</td>
<td>0.085</td>
</tr>
</tbody>
</table>
Calculating for 500 nodes -

**Average Speedup ratio** (Sp) = Average serial execution time / Average parallel execution time

\[
\text{Sp} = \frac{0.4107}{0.146} \quad \text{Sp} = \frac{0.4107}{0.0780}
\]

Sp = 2.8127 (Dual Core)  \quad \text{Sp} = 5.2649 (Quad Core)

**Efficiency** (Ep) = Speedup ratio (Sp) / No. of Processor (P)

Ep = 2.8127 / 2 \quad \text{Ep} = 5.2649/4

Ep = 1.406(Dual Core)  \quad \text{Ep} = 1.316 (Quad Core)

Therefore, Average Operating Efficiency is 140.6% for Dual core and 131.6% for Quad core machines.
Bellman – Ford Algorithm Tabulation

Table 2 Bellman Ford Algorithm Parallel Execution

<table>
<thead>
<tr>
<th></th>
<th>Parallel Execution - 4 core</th>
<th>Parallel Execution - 2 core</th>
<th>Sequential Execution - Single Core</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3 Avg Time</td>
</tr>
<tr>
<td>100</td>
<td>0.015</td>
<td>0.015</td>
<td>0.015</td>
</tr>
<tr>
<td>200</td>
<td>0.047</td>
<td>0.047</td>
<td>0.068</td>
</tr>
<tr>
<td>300</td>
<td>0.14</td>
<td>0.148</td>
<td>0.1483</td>
</tr>
<tr>
<td>400</td>
<td>0.281</td>
<td>0.321</td>
<td>0.306</td>
</tr>
<tr>
<td>500</td>
<td>0.656</td>
<td>0.609</td>
<td>0.65</td>
</tr>
</tbody>
</table>

Fig. 2 Bellman Ford Algorithm Avg Time Graph

Calculating for 500 nodes -

Average Speedup ratio (Sp) = Average serial execution time / Average parallel execution time

Sp = 1.589 / 0.998  Sp = 1.589 / 0.638

Sp = 1.5916 (Dual Core)  Sp = 2.489 (Quad Core)
Efficiency (Ep) = Speedup ratio (Sp) / No. of Processor (P)

\[
\text{Ep} = \frac{1.5916}{2} \quad \text{Ep} = \frac{2.489}{4} \\
\text{Ep} = 0.7982 \text{ (Dual Core)} \quad \text{Ep} = 0.6223 \text{ (Quad Core)}
\]

Therefore, Average Operating Efficiency is 79.8% for Dual core and 62.2% for Quad core.

**Ant Colony Algorithm Tabulation**

**Table 3 Ant Colony Algorithm Parallel Execution**

<table>
<thead>
<tr>
<th>Parallel Execution - 4 core</th>
<th>Parallel Execution - 2 core</th>
<th>Sequential Execution - Single Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.017</td>
<td>0.015</td>
</tr>
<tr>
<td>200</td>
<td>0.068</td>
<td>0.055</td>
</tr>
<tr>
<td>300</td>
<td>0.287</td>
<td>0.29</td>
</tr>
<tr>
<td>400</td>
<td>0.656</td>
<td>0.808</td>
</tr>
<tr>
<td>500</td>
<td>1.1</td>
<td>1.19</td>
</tr>
</tbody>
</table>

**Fig. 3 Ant Colony Algorithm Avg Time Graph**
Calculating for 500 nodes -

**Average Speedup ratio (Sp)** = Average serial execution time / Average parallel execution time

\[ Sp = \frac{4.473}{1.528} \]
\[ Sp = \frac{4.473}{1.11} \]
\[ Sp = 2.927 \text{ (Dual Core)} \]
\[ Sp = 4.017 \text{ (Quad Core)} \]

**Efficiency (Ep)** = Speedup ratio (Sp) / No. of Processor (P)

\[ Ep = \frac{2.927}{2} \]
\[ Ep = 4.017 \]
\[ Ep = 1.467 \text{ (Dual Core)} \]
\[ Ep = 1.004 \text{ (Quad Core)} \]

Therefore, Average Operating Efficiency is 146.7% for Dual core and 100.4% for Quad core.

**Parallel Programming comparison between algorithms – Dual Core**

*Table 4 Parallel Programming comparison – Dual Core*

<table>
<thead>
<tr>
<th></th>
<th>Dijkstra</th>
<th></th>
<th>Bellman</th>
<th></th>
<th>Ant colony</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3 Avg Time</td>
<td>1</td>
<td>2</td>
<td>3 Avg Time</td>
</tr>
<tr>
<td>100</td>
<td>0.031</td>
<td>0.031</td>
<td>0.015</td>
<td>0.0257</td>
<td>100</td>
<td>0.016</td>
</tr>
<tr>
<td>200</td>
<td>0.063</td>
<td>0.063</td>
<td>0.078</td>
<td>0.0680</td>
<td>200</td>
<td>0.109</td>
</tr>
<tr>
<td>300</td>
<td>0.078</td>
<td>0.093</td>
<td>0.095</td>
<td>0.0880</td>
<td>300</td>
<td>0.343</td>
</tr>
<tr>
<td>400</td>
<td>0.124</td>
<td>0.124</td>
<td>0.109</td>
<td>0.1190</td>
<td>400</td>
<td>0.639</td>
</tr>
<tr>
<td>500</td>
<td>0.156</td>
<td>0.141</td>
<td>0.141</td>
<td>0.1460</td>
<td>500</td>
<td>0.998</td>
</tr>
</tbody>
</table>
Fig. 4 Parallel Programming comparison – Dual Core

For 100 and 200 nodes all the algorithms are taking almost same amount of time. Since processor time is used for thread creation and memory allocation for variables. More than 200 nodes, Dijkstra’s best fit algorithm shows a tremendous performance when implemented in parallel programming.

Parallel Programming comparison between algorithms – Quad Core

Table 5 Parallel Programming comparison – Quad Core

<table>
<thead>
<tr>
<th></th>
<th>Dijkstra</th>
<th>Bellman</th>
<th>Ant colony</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.016</td>
<td>0.016</td>
<td>0.017</td>
</tr>
<tr>
<td>200</td>
<td>0.026</td>
<td>0.031</td>
<td>0.027</td>
</tr>
<tr>
<td>300</td>
<td>0.047</td>
<td>0.047</td>
<td>0.049</td>
</tr>
<tr>
<td>400</td>
<td>0.067</td>
<td>0.074</td>
<td>0.067</td>
</tr>
<tr>
<td>500</td>
<td>0.075</td>
<td>0.074</td>
<td>0.085</td>
</tr>
</tbody>
</table>
For 100 and 200 nodes all the algorithms are taking almost same amount of time. Since processor time is used for thread creation and memory allocation for variables. More than 200 nodes, Dijkstra’s best fit algorithm shows a tremendous performance when implemented in parallel programming.

6. CONCLUSIONS

This work proposes the use of multi-threaded OpenMP parallel optimization to improve serial algorithms on multi-core systems, and applies to the actual transportation network to achieve the expected results. The tabulations shows that the time cost of multithreaded parallel algorithms on dual-core system are much faster than the serial algorithms. The parallel running speed can be improved with the increase of the number of cores.

REFERENCES


4. Qiujin Wu, Joanna Hartley & David AL-Dabass, “Time Dependent Stochastic shortest paths algorithms for a scheduled transportation network”.

5. Qiujin Wu, Joanna Hartley & David AL-Dabass, “Using K shortest path algorithms to accommodate user preferences in the optimization of public transport travel”.