Comparison of Parallel and Sequential Execution for Shortest Path Algorithms

A Aparna Menon, Manali Shah, Shivani Shrivastava, Sneha Suresh Patil
Department of Information Technology,
National Institute of Technology, Karnataka Surathkal
Mangaluru, 575025
aparna.mnn97@gmail.com , manalis2402@gmail.com , shivanisri8124@gmail.com, psneha716@gmail.com

Dr. Geetha.V
Department of Information Technology,
National Institute of Technology, Karnataka Surathkal
Mangaluru, 575025
geethav@nitk.ac.in

***Abstract*— Shortest Path Algorithms is an important set of algorithms in today's world. It has many applications like Traffic Consultation, Route Finding and Network Design. It is essential for these applications to be fast and efficient as they mostly require real time execution. Sequential execution of shortest path algorithms for large graphs with many nodes is time consuming. On the other hand, parallel execution can make these applications faster. In this paper we suggest algorithms for the parallel execution of three shortest path algorithms, namely Dijkstra's algorithm, Bellman Ford's Algorithm and Floyd Warshall's Algorithm. The parallel model used for this is the input decomposition model.**

Keywords—Shortest Path Algorithms, Dijkstra's Algorithm, Bellman Ford's Algorithm, Floyd Warshall's Algorithm, Parallelization, OpenMP.

I. INTRODUCTION

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. This can be defined for all kinds of graphs- whether undirected, directed, or mixed. Here, we define it for undirected graphs.

Two vertices are adjacent when they are both incident to a common edge. A path in an undirected graph is a sequence of vertices $P=(v_1,v_2,\ldots,v_n)$ ε V x V x V … x V such that v_i is adjacent to v_{i+1} for $1<=i<n$. Such a path P is called a path of length n-1 from v_1 to v_n . (The v_i are variables; their numbering here relates to their position in the sequence and needs not to relate to any canonical labeling of the vertices.)

Let $e_{i,j}$ be the edge incident to both v_i and v_j . Given a real-valued weight function f: E->R, and an undirected (simple) graph G, the shortest path from v to v' is the path $P=(v_1,v_2,\ldots,v_n)$ (where $v_1=v$ and $v_n=v'$) that over all possible minimizes $\sum_{i=1}^{n-1} f(e_{i,i+1)}$.

OpenMP provides a way for thread parallelism, with shared memory concept. Nowadays, as normal standalone systems also come with dual core, quad core etc, it is beneficial to run the algorithm parallel using threads. The OpenMP provides directives to compiler, for identifying the portion of code or algorithm, which can be run in parallel.

For the purpose of analyzing the parallel algorithm with sequential algorithm, we have considered three shortest path algorithms: Dijkstra's Algorithm, Bellman Ford's Algorithm, Floyd Warshall's Algorithnm. The performance analysis of the same with respect to sequential and parallel execution of the shortest path algorithms shows better improvement with parallel execution for big graphs.

The paper is structured as follows: Section II provides details on literature survey, section III explains parallelization of shortest path algorithms and section IV provides details about experimental results and discussion followed by conclusion and reference.

II. LITERATURE SURVEY

A.Background

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) such that the sum of the weights of its constituent edges is minimized. The computational complexities (i.e., computing power and memory requirements) of the Dijkstra, Bellman Ford and Floyd-Warshall algorithms grow with an increasing number of

nodes and edges for a given graph network. In general, there are two computing approaches for solving such a problem i.e. serial computing and parallel computing. In serial computing, a given problem is broken into discrete parts and discrete parts are solved sequentially (i.e., one at a time). In contrast, parallel computing attempts to solve the discrete parts of a problem concurrently.
There are various parallel programming models and parallel programming environments, such as Message Passing Interface (MPI), Open Multi-Processing (Open MP), Parallel Virtual Machine (PVM) and so on. In this thesis, Open Multi-Processing (OpenMP) is used to implement the above mentioned algorithms and analyze their performance. These systems are scalable, i.e., they can be tuned to available budget and computational needs and allowed efficient execution of both demanding sequential and parallel applications. In this paper, we have used OpenMP for the parallelization of the shortest path algorithms.

A. Literature Survey

Osama G. Attia [1] All pair shortest path difficulty in unweighted directed graphs reduces the problem for finding the path with shortest distance. Thus, running the breadth-first-search algorithm (BFS) from every node in the graph is an all-pair shortest path algorithm that works on un-weighted graphs. Minimum Spanning Tree algorithms classified on one of two approaches, that of Floyd Algorithm, Dijkstra's Algorithm.

Danny Z. Chen, Every pair of vertices in the graph solving by the all pair shortest path algorithm. The large graph whose edges and node is finite number solve by the sequential manner. It finds the time complexity as well as space complexity. The time complexity takes O(N^3) and space complexity takes O(N^2) [2]

III. SHORTEST PATH ALGORITHMS

A. Sequential Dijkstra's Algorithm

Dijkstra's algorithm shown as *Algorithm 1* is used to find the single source shortest path problem. Consider the single vertex to all other vertices in the graph. Dijkstra's algorithm requires the non-negative weight edge cycle. Applying the Problem on Graph to find the distance between source to destination. For a weighted graph G = (V, E, w), the single-source shortest weighted paths problem is to find the nearest paths from a vertex v is subset of V to every other vertices in V, where V is the finite set of vertices, E is the finite set of edge, w is weight between the nodes of graph. Dijkstra's algorithm solves the single source shortest-path problem on directed and undirected graphs with non-negative weights edgs. Dijkstra's algorithm, find the minimum path from single vertex. Dijkstra's algorithm is similar to Prim's minimum spanning tree algorithm. Resembling Prim's algorithm, it incrementally find the shortest paths from s to the other vertices of graph (G). It is also greedy. i.e., it always selects an edge to a vertex that appeared closest.

The algorithm works as follows:

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.

2. Mark all nodes unvisited. Set the initial node as current. Create a set of the unvisited nodes called the unvisited set consisting of all the nodes except the initial node.

3. For the current node, consider all of its unvisited neighbors and calculate the tentative distances. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be 6+2=8. If this distance is less than the previously recorded tentative distance of B, then overwrite that distance. Even though a neighbor has been examined, it is not marked as "visited" at this time, and it remains in the unvisited set.

4. When complete the neighbors of the current node and mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.

5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal), then stop. The algorithm has finished.

6. Select the unvisited node that is marked with the smallest tentative distance, and set it as the new "current node" then go back to step 3.

Algorithm 1: Sequential Dijkstra's Algorithm

```
1.  Read the number of vertices //V
2. Enter the adjacency matrix //graph[][]
3.  source = 0
4. for each vertex v in Graph: //Initialization distance graph
        dist[v] ← INF
        sptSet[v] ← 0
5.  dist{source]=0
6.  for count from 0 to |V-1|
7.     for v from 0 to |V-1|
8.        if v equals 0
9.          min ← INF
10.       end if
11.       if sptSet[v] equals 0 and dist[v] <= min
12.          min=dist[v]
13.          u=v
14.       end if
```

```
15.  sptSet[u] ← 1
16.  for v from 0 to |V-1|
17.      if (!sptSet[v] && graph[u][v] && dist[u] != INF &&
dist[u]+graph[u][v] < dist[v])
18.          dist[v] = dist[u] + graph[u][v];
```

B. Sequential Bellman Ford's Algorithm

The Bellman–Ford algorithm shown as *Algorithm 2* is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph.

The algorithm works as follows:

1.From the given source vertex, initialize all distances as infinite, except the distance to source itself. The number of times the edges must be processed is equal to the total number of vertices in the graph.

2.Starting from the source node, calculate the distance to the other nodes which are directly connected to the source node. In other words, visit the nodes which are one edge long to the source node and calculate their distance.

3.The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time

4.The second iteration guarantees to give all shortest paths which are at most 2 edges long.

5. Repeat the algorithm for n number of times where n is the total number of edges in the graph and the final result would be the shortest distance from the source node to all the other nodes.

Algorithm 2: Sequential Bellman Ford's Algorithm

```
1. Read the number of vertices //V
2. Read the number of edges //E
3. Read the source vertex, destination vertex and weight of
the edge
4. Initialise the distance array
for i from 0 to |V-1|
   dist[i]=INT_MAX
dist[src]=0
5. Update the distance array
for i from 1 to |V-1|
    for j from 0 to |E-1|
       u ← graph->edge[j].src
       v ←graph->edge[j].dest
      if dist[u]!=INT_MAX and dist[u]+weight<dist[v]
            dist[v]=dist[u]
      end if
6. Detection for negative weight cycle
for i from 0 to E
   u ← graph->edge[j].src
   v ←graph->edge[j].dest
   if dist[u]!=INT_MAX and dist[u]+weight<dist[v]
      print "Negative weight cyle is there "
7. Print the final distance array
for i from 0 to |V-1|
   print dist[i]
```

C. Sequential Floyd Warshall's Algorithm

The Floyd–Warshall algorithm shown as *Algorithm 3* is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices. Although it does not return details of the paths themselves.

The algorithm works as follows:

1. This algorithm find the shortest path by constructing the distance matrix from the source node to all the other nodes. The distance matrix follow the following things
i. If i=j, distance should be 0
ii. Distance[i][j] should be equal to Distance[j][i].
iii. If two nodes are not connected, distance should be infinity i.e. a very large number.

2.Then the weighted matrix is generated according to the k formula W ij = min (W ij , W ik + W jk).

3. Now, construct a matrix D1, in which first row and column will remain the same and the other values will be filled by the above mentioned formula.

4. Now, construct matrix D2, in which second row and second column of D1 will remain the same and other values will be filled by the above formula.

5. Repeat this for all the rows and columns i.e. repeat this n times where n is equal to the number of edges.

The final matrix will be shortest distance matrix.

Algorithm 3: Sequential Floyd Warshall's Algorithm

```
1. Read the number of vertices //V
2. Read the adjacency matrix for the graph //graph[][]
3. let dist be a |V| × |V| array of minimum distances initialized to graph[][]
for i from 0 to |V-1|
   for j from 0 to |V-1|
      dist[i][j] ← graph[i][j]
4. for k from 0 to |V-1|
      for i from 0 to |V-1|
         for j from 0 to |V-1|
            if dist[i][k] + dist[k][j] < dist[i][j]
               dist[i][j] ← dist[i][k] + dist[k][j]
            end if
5. for i from 0 to |V-1|
      for j from 0 to |V-1|
            if dist[i][j] = INF
                  print INF
            else
                  print dist[i][j]
            end if
```

IV. WORK DONE AND RESULT ANALYSIS

The sequential shortest path algorithms are implemented in C. The parallelization of the algorithm is performed using OpenMP. OpenMP is a usage of multithreading, a strategy for parallelizing whereby an expert string forks a predetermined number of slave strings and the framework separates an errand among them. The strings then run simultaneously, with the runtime environment assigning strings to distinctive processors.

The segment of code that is intended to keep running in parallel is stamped likewise, with a preprocessor order that will bring about the strings to shape before the segment is executed. Each string has an id appended to it which can be acquired utilizing a method *omp_get_thread_num()*. The string id is a whole number, and the master string has an id of 0. After the execution of the parallelized code, the strings join over into the master string, which proceeds with forward to the end of the system.

As a matter of course, every string executes the parallelized segment of code freely. Work-sharing develops can be utilized to partition an assignment among the strings so that every string executes it's apportioned a portion of the code. Both undertaking parallelism and information parallelism can be accomplished utilizing OpenMP as a part of along these lines.

The runtime environment assigns strings to processors contingent upon use, machine burden and different variables. The runtime environment can dole out the quantity of strings in view of environment variables, or the code can do as such utilizing capacities. The OpenMP capacities are incorporated into a header document named omp.h in C/C++.

The performance of the algorithm is analyzed with respect to speedup. The speedup is calculated as shown in equation 1.

Equation 1:

Speedup = Time for serial execution/Time for parallel execution

The results are analyzed for three different algorithms: Dijkstra's, Bellman Ford's and Floyd Warshall's algorithm.

Parallelizing algorithms

All the above three algorithms use a parallel formulation of the shortest path problem to increase concurrency. These algorithms are parallelized using Open MP. An Open MP application begins with a single thread called the master thread. As the parallel program executes, the master thread encounters parallel regions in which the master thread creates thread teams (which include the master thread). At the end of a parallel region, the thread teams are stopped and all the thread combine to form the master thread and continues execution. Since Open MP is primarily a pragma or directive based standard. To start, serial code of the algorithms were written and then were parallelized keeping in mind the regions which are independent and can be parallelized. For parallelization using Open MP, several constructs like work sharing constructs (for directive, section directive and single directive), synchronization constructs (critical directive and barrier directive) and clauses like no wait, private, shared are used.OpenMP uses the fork join model shown in Figure 1,2 and 3.

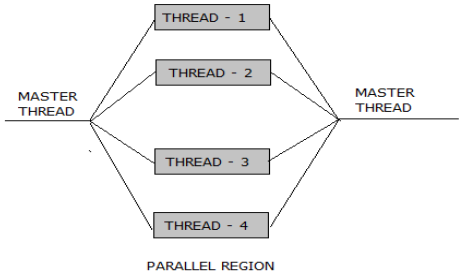

Figure 1

The directives and clauses used in the parallel algorithms are:

A ***parallel*** region is a block of code that will be executed by multiple threads.

The ***single*** directive specifies that the enclosed code is to be executed by only one thread in the team.

The ***for*** directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

The ***schedule*** clause describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.

In ***dynamic*** scheduling loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another.

The ***default*** clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region.

The ***critical*** directive specifies a region of code that must be executed by only one thread at a time.

The ***atomic*** directive specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it. In essence, this directive provides a mini-critical section.

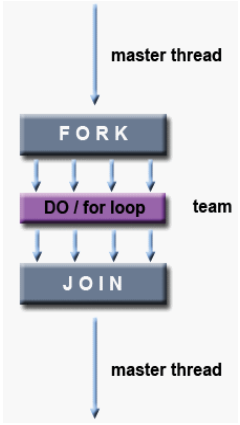

Figure 2

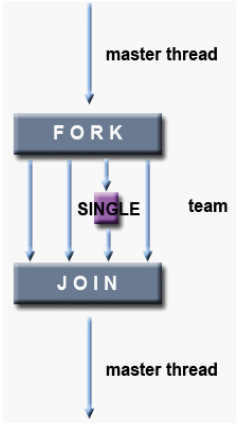

Figure 3

Algorithm 4 : Dijkstra's Algorithm Parallel

DIJKSTRA'S PARALLEL ALGORITHM

```
#pragma omp single
1.  Read the number of vertices //V
2. Enter the adjacency matrix //graph[][]
3.  source = 0
4. for each vertex v in Graph: //Initialization distance graph
        dist[v] ← INF
        sptSet[v] ← 0
5.  dist{source]=0
#pragma omp parallel for schedule(dynamic,chunk)
collapse(2)
6.  for count from 0 to |V-1|
7.     for v from 0 to |V-1|
         #pragma omp critical
8.        if v equals 0
9.          min ← INF
10.       end if
11.       if sptSet[v] equals 0 and dist[v] <= min
12.          min=dist[v]
13.          u=v
14.       end if
15.  sptSet[u] ← 1
 #pragma omp parallel for schedule(dynamic,chunk)
16.  for v from 0 to |V-1|
17.     if (!sptSet[v] && graph[u][v] && dist[u] != INF
&&  dist[u]+graph[u][v] < dist[v])
            #pragma omp atomic
18.         dist[v] = dist[u] + graph[u][v];
```

Algorithm 5 : Bellman Ford Parallel Algorithm

BELLMAN FORD PARALLEL

```
#pragma omp single
1. Read the number of vertices //V
2. Read the number of edges //E
3. Read the source vertex, destination vertex and      weight
of the edge
4. Initialise the distance array
#pragma omp parallel for schedule(dynamic,chunk)
for i from 0 to |V-1|
   dist[i]=INT_MAX
dist[src]=0
5. Update the distance matrix
#pragma omp parallel for schedule(dynamic,chunk)
collapse(2)
```

```
for i from 1 to |V-1|
    for j from 0 to |E-1|
       u ← graph->edge[j].src
        v ←graph->edge[j].dest
       #pragma omp critical
if dist[u]!=INT_MAX and dist[u]+weight<dist[v]
                dist[v]=dist[u]
       end if
6. Detection for negative weight cycle
#pragma omp parallel for schedule(dynamic,chunk)
for i from 0 to E
   u ← graph->edge[j].src
   v ←graph->edge[j].dest
   #pragma omp critical
   if dist[u]!=INT_MAX and dist[u]+weight<dist[v]
      print "Negative weight cyle is there "
7. Print the final distance array
for i from 0 to |V-1|
   print dist[i]
```

Algorithm 6 : Floyd Warshall Parallel Algorithm

FLOYD WARSHALL PARALLEL

```
#pragma omp single
1. Read the number of vertices //V
2. Read the adjacency matrix for the graph //graph[][]
3. let dist be a |V| × |V| array of minimum distances initialized to graph[][]
#pragma omp parallel schedule(dynamic,chunk) collapse(2)
for i from 0 to |V-1|
   for j from 0 to |V-1|
      dist[i][j] ← graph[i][j]
4. Update the dist[][] matrix
#pragma omp parallel schedule(dynamic,chunk) collapse(3)
for k from 0 to |V-1|
    for i from 0 to |V-1|
        for j from 0 to |V-1|
          #pragma omp critical
           if dist[i][k] + dist[k][j] < dist[i][j]
              dist[i][j] ← dist[i][k] + dist[k][j]
           end if
5. Print the final matrix
#pragma omp parallel for schedule(dynamic,chunk)
collapse(2)
for i from 0 to |V-1|
     for j from 0 to |V-1|
        #pragma omp critical
        if dist[i][j] = INF
           print INF
        else
           print dist[i][j]
        end if
```

V RESULT AND DISCUSSION

The three algorithms were run, both, in parallel and sequential. The inputs to the graph for Dijkstra's as well as Floyd Warshall's algorithms were given randomly in the code, for number of nodes =100, 200, 500 and 1000.
For Bellman Ford, the number of nodes was kept restricted to 8, but the number of threads were varied as 2,4 and 8. The results obtained in the execution, are given as follows, in tables 1, 2 and 3.

Table 1: Results for Dijkstra's Algorithm (in seconds)

	n=100	n=200	n=500	n=1000
Sequential	0.000274	0.00261	0.003334	0.008866
Parallel	0.009248	0.00662	0.084585	0.050105

Table 2: Results for Floyd Warshall's Algorithm (in seconds)

	n=100	n=200	n=500	n=1000
Sequential	0.013221	0.20379	0.759469	4.373039
Parallel	0.129361	0.39927	5.405254	41.64940

Table 3: Results for Bellman Ford's Algorithm (in seconds)

	Execution Time in Parallel
Number of threads=2	0.000474
Number of threads=4	0.000442
Number of threads=8	0.00091

From this, we observe that the execution time for the parallel execution is greater than that of sequential execution for Dijkstra's algorithm and Floyd Warshall's algorithm.
This is because in these algorithms, most of the statements executed are put into the critical section, where only one thread can execute it. This is similar to sequential execution. But parallel execution also has overheads for the communication between threads.

The execution time of the Bellman Ford Algorithm improves as the number of threads increases from 2 to 4, but decreases from 4 to 8.

VI CONCLUSION

In this paper, the shortest path algorithms to solve the all pair shortest path and single source shortest path of graph were analyzed. The single source shortest path and all pair shortest path using three serial algorithms and three parallel algorithms were compared, it can be converted into parallel.
It was observed that for small inputs, the time taken by sequential algorithms is more than the time taken by parallel algorithms. This happens because of the overhead, as the threads must communicate with each other. Since the threads are working on the same data, they need to synchronize amongst themselves. Hence this overhead overshoots the parallel execution time, thereby resulting in sequential algorithms take lesser time.

There is a lot of suite for experimentation with the parallel search algorithms. In this work, we explored the performance of parallel shortest path algorithms approach over serial execution.

REFERENCES

[1] Osama G. Attia, Alex Grieve, Kevin R. Townsend, Phillip Jones, and Joseph Zambreno ,"Accelerating All-Pairs Shortest Path Using a Message-Passing Reconfigurable Architecture",2015.

[2] Danny Z. Chen, "Solving the All-Pair Shortest Path Problem on Interval and Circular-Arc Graphs.(Preliminary Version)",The National Science Foundation,1994

[3] Rajashri Awari, "Parallelization of Shortest Path Algorithm Using OpenMP and MPI", 2017

[4] Uditi, "Comparative Analysis of Parallelised Shortest Path Algorithms using Open MP", 2017