Parallel Implementation of Graph Algorithms

Bhagyashri Bhamare
Dept. of Information Technology
NITK Surathkal
Mangalore, India
bhamare.bhagyashri1999@gmail.com

Chinmayi C. R.
Dept. of Information Technology
NITK Surathkal
Mangalore, India
chinmayicr27@gmail.com

K. Keerthana
Dept. of Information Technology
NITK Surathkal
Mangalore, India
keerthanakanapuram@gmail.com

Utkarsh Meshram
Dept. of Information Technology
NITK Surathkal
Mangalore, India
utkarsh.meshram2000@gmail.com

***Abstract*—Graphs are basically collections of vertices and edges and have wide applications. Problems that arise in the real life network imply the finding of the shortest path and its distances from source to one or more destinations. Real world involves large graphs involving millions of vertices are common and are challenging to process. Data is growing exponentially and sequential execution proves inefficient. So we need a faster approach. We have four approaches to solve the single-source shortest path problem: Dijkstra's algorithm, Bellman Ford algorithm, Floyd Warshall algorithm and Prim's algorithm.**

***Index Terms*—Bellman-Ford, Dijkstra, Floyd Warshall, Prim, single-source shortest path, Parallel graph algorithms**

I. Introduction

Dijkstra's algorithm is a graph search algorithm that solves 6 single-source shortest path for a graph with non-negative weights. Parallel implementation is the solution for large inputs. Bellman Ford algorithm, despite being slower than Dijkstra's algorithm is more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. The input graph G(V, E) being considered in this problem is connected, directed and may contain negative weights. The algorithm finds a shortest path from a specified vertex to every other vertex in the graph. If there is a negative cycle (a cycle on which the sum of weights is negative) in the graph, there will be no shortest path.The Floyd Warshall is a classic dynamic programming algorithm that solves the all-pairs shortest path (APSP) problem on directed weighted graphs. Floyd Warshall can be applied to graphs with negative weight edges to determine whether the graph has negative cycles or not. Floyd Warshall is used in many real-life applications, such as bioinformatics for clustering correlated genes,in database systems or in data mining.Prim's algorithm is a popular greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. The parallel implementation uses the cut property of graph to find the minimum weight edge using multiple threads.

II. Literature Survey

Many approaches have been adopted to optimise shortest path problem in graphs.

A. Literature Survey 1

F. Busato and N. Bombieri [1] have presented a parallel implementation of the Bellman-Ford algorithm based on frontier propagation. This is different from all the other approaches. In this paper, a frontier data structure in which all and only active nodes are processed in parallel is used. The parallel processing of active nodes does preserve the semantics of the algorithm.The article also presents different optimizations to the implementation, which are oriented both to the algorithm and to the architecture.

B. Literature Survey 2

Zhiyuan Yan and Qifan Song [2] have presented a parallel implementation of the Floyd-Warshall algorithm based on Hybrid MPI and OpenMp to exploit the parallelism inside a multi-core node computer. Different schemes of PF based on diverse matrix partitions are implemented and the efficiencies of PF with multi-thread and multi-process are analyzed by using of Chinese traffic network data. The result indicates that the efficiency of the implementation with multi-process and 2 threads in each process is about 80 % higher than multi-process single-thread one.

C. Literature Survey 3

H. Ortega-Arranz et. al. [3] has also solved this problem by using Dijkstra's algorithm. The time complexity of this algorithm is O(V2). In this approach the internal operations of the sequential Dijkstra algorithm are parallelized. The idea behind this is that the parallelization of a single sequential Dijkstra algorithm resides in the inherent parallelism of its loops. For each iteration of Dijkstra's algorithm, the outer loop selects a node to compute new distance labels. Inside this loop, the algorithm relaxes its outgoing edges in order to update the old distance labels, that is the inner loop. Parallelizing the inner loop implies to traverse simultaneously the outgoing edges of the frontier node.

D. Literature Survey 4

Vladimir Lonc˘ar, Srdjan Škrbic´and Antun Balaz˘ [4] present their parallelization targeting message passing parallel machine with distributed memory. Their work considers large graphs that can not fit into memory of one process. It is believed that only two steps can be parallelized: selection of the minimum-weight edge connecting a vertex not in MST to a vertex in MST, and updating array after a vertex is added to MST.

III. Methodology

The four algorithms used for the shortest path problem are: Dijkstra's algorithm, Bellman Ford algorithm, Floyd Warshall algorithm and Prim's algorithm.

A. Dijkstra's algorithm

Serial Region:

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:
        dist[v] = infinity
        previous[v] = undefined
        dist[source] = 0
        Q = the set of all nodes in Graph
    while Q is not empty:
        u = node in Q with smallest dist[ ]
        remove u from Q
        for each neighbor v of u:
            alt = dist[u] + dist_between(u, v)
            if alt < dist[v]
                dist[v] = alt
                previous[v] = u
    return previous[ ]
```

Parallel region:

For a given source node in the graph, Dijkstra's algorithm finds the shortest path between that node and every other.For openmp, parallel finding the next node and update distance parts. Which means, for outer loop, renew the set which contains nodes that already in shortest new graph, this part stay still, can't be paralleled. For two inner loops, which are finding the next node and update the distance for this new node, this part can be paralleled.

```
#pragma omp parallel for schedule(runtime) private(i)
 for(i to V)
    if(vertices[i].visited == FALSE)
        int c = findEdge( u, vertices[i], edges, weights)
            len[vertices[i].title] = minimum(len[vertices[i].title],len[u.title]+ c)
```

B. Bellman Ford algorithm

Bellman Ford -algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths. By doing this repeatedly for all vertices, we can guarantee that the result is optimized.

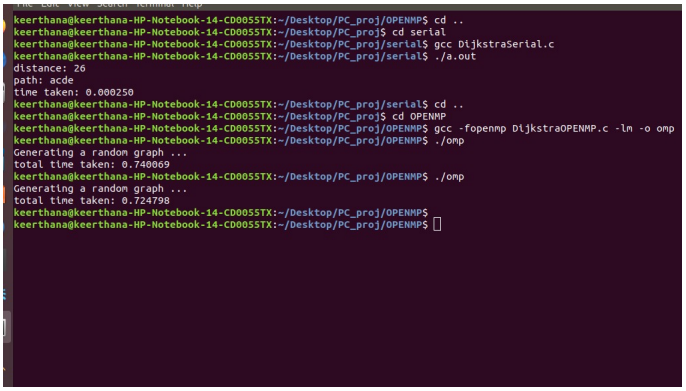

Fig. 1. Output: Dijkstra's algorithm

Serial Algorithm:

```
Algorithm BellmanFord (s,Dist,Cost,n)
    for i=1 to n do
        Dist[i] = Cost[s,i];
    End for
    for k=1 to n-1 do
        for each (u,v) in E do
            Relax(u,v)
        End for
    End for
Relax (u,v):
    if Dist[v]> Dist[u] + Cost[u,v]
        Dist[v] = Dist[u] + Cost[u,v]
```

Parallel region:OpenMP

```
function bellmanFord:
  omp_se_num_threads(p)
  while(!queue.empty() and no negative cycle):
  u = queue.front()
  queue.pop()
  In_queue[u] = false
  #pragma omp parallel for
      for (v to n):
          weight = mat[u * n + v];
          if (weight < INF):
              new_dist = weight + dist[u];
              if (new_dist < dist[v]):
                  dist[v] = new_dist;
                  enqueue_counter[v]++;
                  if (in_queue[v] == false)
                      in_queue[v] = true;
                      if(enqueue_counter[v]>=n)
                        *has_negative_cycle = true;
  #pragma omp critical
      queue.push(v)
```

SPFA assumes the distances to all nodes to be infinity before doing anything. It is as if all other nodes are disconnected to the starting node. Then, by exploring the edges, SPFA gradually shortens the distances until there is no room for further optimization. In Bellman Ford algorithm using queue optimisation, SPFA maintains an examine queue, which is the heart of the algorithm. The queue stores nodes that have to be examined for distance update. And SPFA terminates when the queue empties, which indicates that there is no further possible optimization.

Parallel region:MPI

```
function bellman_ford(my_rank, p, MPI_Comm comm, n, *mat, s, *dist, bool *has_negative_cycle):
    for (int i = 0 to loc_n)
        loc_dist[i] = INF;
    loc_dist[s] = 0;
    MPI_Barrier(comm);
    int loc_iter_num = 0;
    if (my_rank == 0):
        loc_n = n;
    MPI_Bcast(&loc_n, 1, MPI_INT, 0, comm);
    if (my_rank == p - 1) :
        loc_end = loc_n;
    MPI_Bcast(loc_mat, loc_n * loc_n, MPI_INT, 0, comm);
    for (iter = 0 to loc_n - 1) :
        loc_has_change = false;
        loc_iter_num++;
            for (int u = loc_start to loc_end):
                for (int v = 0 to loc_n) :
                  weight = loc_mat[utils::convert_dimension_2D_1D(u, v, loc_n)];
                  if (weight < INF) :
                    if (loc_dist[u] + weight < loc_dist[v]) :
                      loc_dist[v] = loc_dist[u] + weight;
                      loc_has_change = true;
                            MPI_Allreduce(MPI_IN_PLACE, &loc_has_change, 1, MPI_CXX_BOOL, MPI_LOR, comm);
            if (!loc_has_change):
                break;
            MPI_Allreduce(MPI_IN_PLACE, loc_dist, loc_n, MPI_INT, MPI_MIN, comm);
            if (loc_iter_num == loc_n - 1) :
                loc_has_change = false;
                for (int u = loc_start; u < loc_end; u++) :
                  for (int v = 0; v < loc_n; v++) :
                              int weight = loc_mat[utils::convert_dimension_2D_1D(u, v, loc_n)];
                    if (weight < INF) :
                      if (loc_dist[u] + weight < loc_dist[v]) :
                      loc_dist[v] = loc_dist[u] + weight;
                      loc_has_change = true;
                      break;
                            MPI_Allreduce(&loc_has_change, has_negative_cycle, 1, MPI_CXX_BOOL, MPI_LOR, comm);
```

Fig. 2. Output: Bellman Ford Algorithm

```
    if(my_rank == 0):
        memcpy(dist, loc_dist, loc_n * sizeof(int);
```

C. Floyd Warshall algorithm

In Floyd Warshall Algorithm, the solution matrix is initialised the same way as the input graph matrix in the first step. Then the solution matrix is updated by considering each vertex as an intermediate vertex. The idea is to pick all the vertices one by one and update all the shortest paths which includes the picked vertex as an intermediate vertex in the shortest path.
A parallel region wherein each thread will get a particular number of rows of intermediate matrix. And those row values are updated parallelly. In the second version, Parallelized Tiled FW Algorithm, the input matrix is divided into tiles of size B. During the k-th block iteration, the algorithm updates the k-th diagonal tile first, then it updates the tiles in the remainder of the k-th block row (East and West Tiles) parallelly and k-th block column (North-South tiles) parallelly and finally it updates the remaining tiles of the matrix (North East, North West, South East, South West tiles) parallelly. Each thread takes one tile of size BXB and applies the FW algorithm to execute in parallel.

Serial Region:

```
FloydWarshall(Ak,n):
  n = no of vertices
  A = matrix of dimension n*n
  for k = 1 to n
      for i = 1 to n
          for j = 1 to n
              A[i, j] = min (Ak-1[i, j], Ak-1[i, k] + Ak-1[k, j])
return A
```

Parallel Region:OpenMP

In parallel region k acts as middle node to reach a particular node j where i acts as source. Here we move into parallel region with first for loop for k. Also there's another parallel region for i and j.

```
#pragma omp parallel
```

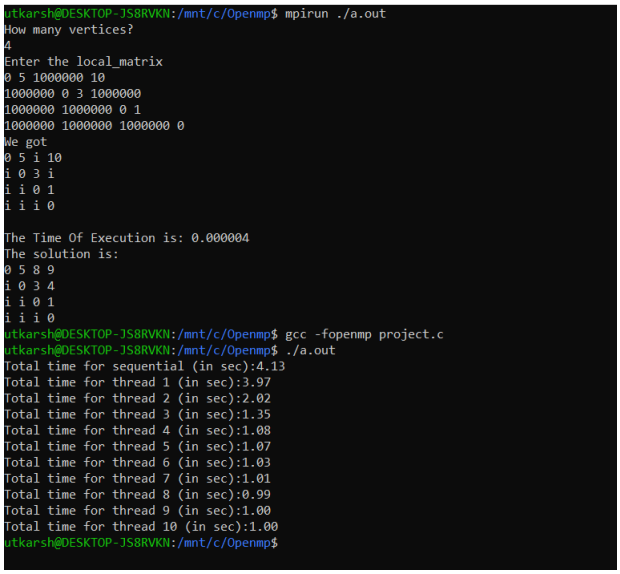

Fig. 3. Output: FLoyd Warshall

```
for (k = 0 ; k < N; k++)
dm = Ak[k]
#pragma omp parallel
    for (i=0 ; i < N ; i++)
    ds = Ak[i]
        for (j=0; j < N;j++)
        ds[j] = min (ds[j],ds[k]+dm[j])
```

Parallel region:MPI

```
Floyd(local_mat[], n, my_rank, p, MPI_Comm comm):
 row_k = malloc(n*sizeof(int));
 for (global_k = 0; global_k < n; global_k++):
    root = Owner(global_k, p, n);
    if (my_rank == root):
        Copy_row(local_mat, n, p, row_k, global_k);
    MPI_Bcast(row_k, n, MPI_INT, root, comm);
    for (local_i = 0 to n/p):
        for (global_j to n):
            temp = local_mat[local_i*n + global_k] +
row_k[global_j];
            if (temp < local_mat[local_i*n+global_j]):
        local_mat[local_i*n + global_j] = temp;
```

The adjacency matrix is distributed by block rows.Owner function returns rank of process that owns global row k.

D. Prim's algorithm

Prim's algorithm is a greedy algorithm, and it starts by selecting a random vertex as the root of the tree. It then grows the tree by adding a vertex that is closest to current tree. The algorithm ends when all vertices have been added to the tree. The sum of all added edges is the cost of MST. The computational complexity of this serial algorithm is O(N2), where N is number of vertices which can be reduced using the parallel implementation.

Serial region:

function Prim:

T =∅

$U = \{1\}$

$while(U \neq V)$

$let(u, v)$ *be the lowest cost edge such that* $u \in U$ *and* $v \in V - U$

$T = T \cup (u, v)$

$U = U \cup v$

Parallel region:OpenMP

The parallel implementation can be divided into two steps:

1. Parallelizing the code for finding minimum key for the vertices by finding local minimum and then global minimum.

2. Parallelizing the logic for updating the values of the key vector after finding the minimum key. Prim's algorithm is iterative. Each iteration adds a new vertex to the minimum spanning tree. Since the value of key[v] for a vertex v may change every time a new vertex u is added in VT, it is hard to select more than one vertex to include in the minimum spanning tree. However, the method of finding index of the vertex with minimum key value can be parallelized.

```
function minKey(key[], visited[]):
    min = INT_MAX, index, i;
    #pragma omp parallel
        index_local = index;
        min_local = min
    #pragma omp for nowait
        for (i to n)
            if (visited[i] == false and key[i] <
min_local)
                min_local = key[i]
                index_local = i
    #pragma omp critical
        if (min_local < min)
            min = min_local
            index = index_local
    return index
```

For further optimization, the logic of updating the values of key vector after finding the minimum key value can also be parallelized using static scheduling for parallel for loop for updating the values of key vector after we find the global minimum from the previous parallel for loop.

IV. Conclusion

We used four algorithms to experiment and analyse different input size and different number of threads to understand the best algorithm for a particular input with the help of openmp and mpi. There is a major difference in the execution time for sequential and parallel implementation as the input size increases. Floyd Warshall shows a huge difference in the sequential and parallel implementation. As the number

Fig. 4. Graph: Prim's Algorithm

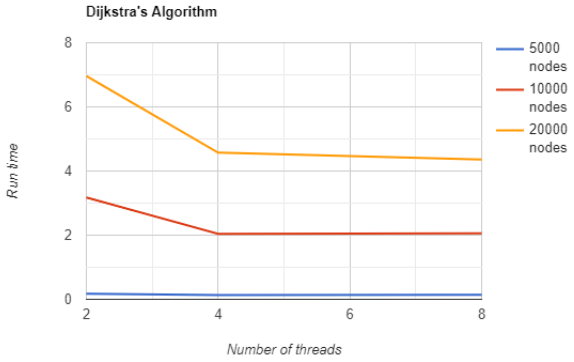

Fig. 5. Graph: Dijkstra's Algorithm

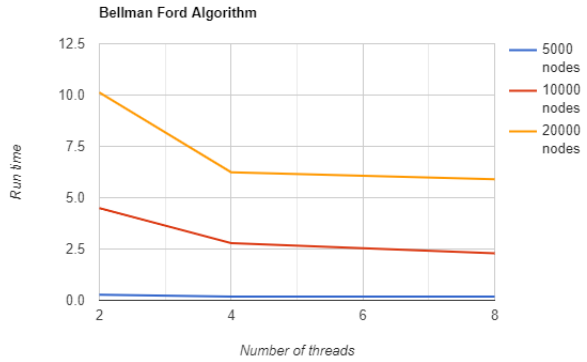

Fig. 6. Graph: Bellman Ford Algorithm

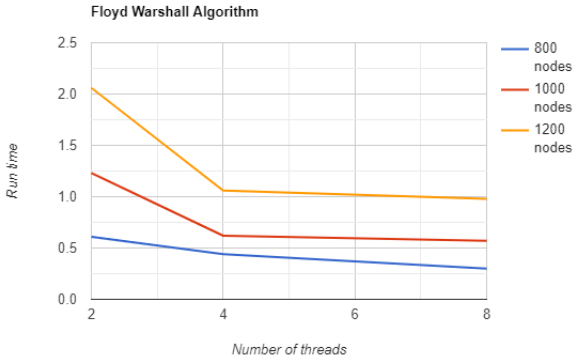

Fig. 7. Graph: Floyd Warshall Algorithm

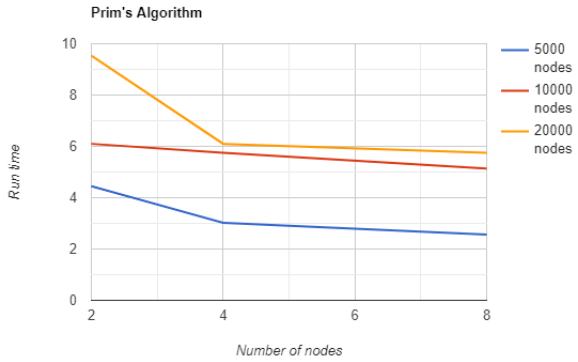

Fig. 8. Graph: Prim's Algorithm

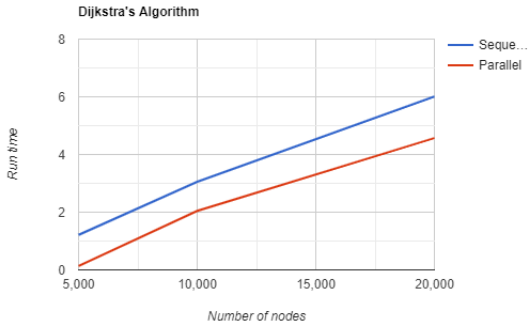

Fig. 9. Graph: Dijkstra's Algorithm

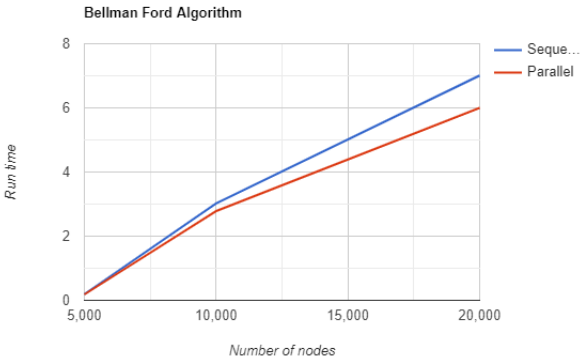

Fig. 10. Graph: Bellman Ford Algorithm

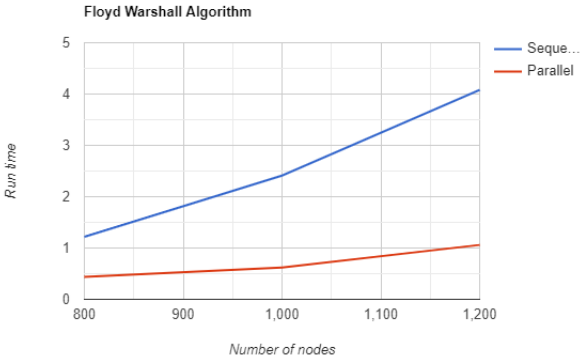

Fig. 11. Graph: Floyd Warshall Algorithm

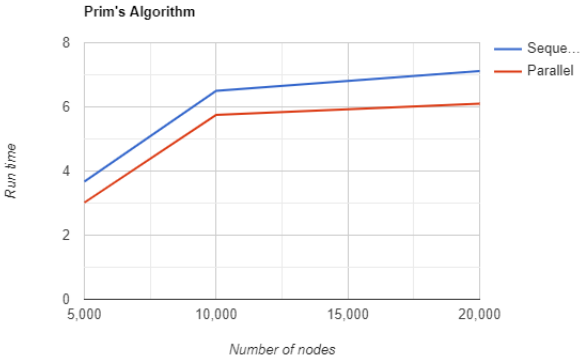

Fig. 12. Graph: Prim's Algorithm

Dijkstra's Algorithm	2	4	8
5000	0.1748	0.1342	0.1448
10000	3.1735	2.0452	2.0573
20000	6.9632	4.571	4.3563

Fig. 13. Table: Dijkstra's Algorithm

Bellman Ford Algorithm	2	4	8
5000	0.2672	0.1832	0.1862
10000	4.49	2.783	2.2951
20000	10.122	6.2421	5.8995

Fig. 14. Table: Bellman Ford Algorithm

Floyd Warshall Algorithm	2	4	8
800	0.61	0.44	0.30
1000	1.23	0.62	0.57
1200	2.06	1.06	0.98

Fig. 15. Table: Floyd Warshall Algorithm

of threads is increased for larger inputs it tends to work better.Bellman Ford Algorithm shows a major decline in the execution time as the number of threads is increased. Dijkstra's algorithm works better in the parallel implementation for most of the inputs. Prim's algorithm works best in parallel for larger inputs.

V. Future Work

Add features that help detect automatically the best algorithm for a type of graph. To improve efficiency of output by choosing the optimum algorithm with respect to number of vertices, processors and threads.

VI. INDIVIDUAL CONTRIBUTION

Bhagyashri Bhamare 181IT111	Bellman Ford Algorithm: Serial, OpenMP and MPI
Chinmayi C. R. 181IT113	Prim's Algorithm: Serial and OpenMP Bellman Ford Algorithm: MPI
K. Keerthana 181IT221	Dijkstra's Algorithm: Serial and OpenMP Floyd Warshall Algorithm: MPI
Utkarsh Meshram 181IT250	Floyd Warshall: Serial, OpenMP and MPI

References

[1] S. F. Busato and N. Bombieri, "An Efficient implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures," vol. 27, no. 8, pp. 2222–2233, 2016.

[2] Zhiyuan Yan and Qifan Song "Parallel Implementation of the Floyd-Warshall algorithm based on Hybrid MPI and OpenMp." October 2012

[3] H. Ortega-Arranz, Y. Torres, D. Llanos, and A.Gonzalez-Escri-bano, "A new GPU-based approach to the shortest path problem," in Proc. Int. Conf.High Perform. Comput. Simul. 2013, pp. 505–511

[4] Vladimir Lonc˘ar, Srdjan Škrbic´and Antun Balaz˘, "Parallelization of Minimum Spanning Tree Algorithms Using Distributed Memory Architectures". 2012

Prim's Algorithm	2	4	8
5000	4.448	3.018	2.5623
10000	6.0989	5.75	5.1331
20000	9.5345	6.098	5.75

Fig. 16. Table: Prim's Algorithm