**Parallel Programming Lab Project**

**Parallel Johnson’s Algorithm**

***Submitted by***

Hazlin Dsouza 160905013

Dia Majumder 160905100

Keya Sukthankar 160905186

VI Sem, CSE B

Department of Computer Science & Engineering



**Abstract:**

The problem is to find shortest paths between every pair of vertices in a given weighted directed graph. Using Johnson’s algorithm, we can find all pair shortest paths. Johnson’s algorithm uses both Dijkstra and Bellman-Ford algorithms as subroutines. In this problem, even edges with negative weights may be present. The basic idea of this algorithm is to assign a new weight to every edge using shortest distance values. This increases all set of paths between two vertices by the same amount and all negative edges become non negative. To calculate the vertex weights, the Bellman Ford algorithm is used. In this project, we parallelize Dijkstra’s algorithm to enhance its performance.

**Objective:**

The objective of the project is to demonstrate the computational performance of the parallelized Dijkstras algorithm to find shortest paths between every pair of vertices. Dijkstras algorithm is implemented in Message Passing Interface and CUDA to be parallelized.

We can find the shortest paths between every pair of vertices sequentially. This sequential approach however is not feasible when scale is galactic, necessitating the use of parallel, high performance computing and sophisticated codes.

The parallel implementation of this problem is studied by relevant comparisons between execution time that are made to realize the improvement in computational performance.

**Introduction:**

Johnson’s algorithm is an all pair shortest path problem, which can even take care of negative edge weights in directed acyclic graphs. This algorithm uses Bellman ford algorithm and Dijkstra’s algorithm as subroutines. The idea of the algorithm is to iteratively apply Dijkstra’s algorithm to each of the graph vertices, so one can find the shortest paths between any pair of vertices. Although Dijkstra’s algorithm disadvantage is the inability of its application for graphs with negative edge weights. For correct execution of the Johnson’s algorithm, the input graph must not have negative-weight cycles. Bellman-Ford algorithm is executed

to remove the non-negative values and to get a modified graph with non-negative edge weights.

As the number of vertices increases, the complexity of the algorithm also increase. The time complexity of the serial implementation of Johnson’s algorithm is O(V2log V+VE). Individually, the complexities of Bellman Ford and Dijkstra algorithms are O(VE) and O(VlogV) respectively.

**Literature Review:**

1. <https://www.geeksforgeeks.org/johnsons-algorithm/>
2. Dijkstra's shortest path algorithm serial and parallel execution performance analysis

By Nadira Jasika, Naida Alispahic, Arslanagic Elma, Kurtovic Ilvana, Lagumdzija Elma, Novica Nosovic.

<https://ieeexplore.ieee.org/abstract/document/6240942>

**Methodology:**

MPI CODE:

We take directed acyclic graph as input and assign dist[ ] array to infinity(999) and todo[ ] array to -1 check vertex is visited or not to. Select a source.

First step is creation of new graph using bellman-for algorithm to replace initial weights with new weights such that shortest path remains the same and weights of all edges in a graph are non-negatives.

Modified graph is broadcasted to all process and each process takes care of one vertex.

The dijkstra’s algorithm is implemented parallelly in MPI. Dijkstra’s algorithm distinguishes two types of vertices: visited or unvisited. If the vertex is unvisited and distance of source to vertex directly is greater than distance of source to vertex by alternating paths then the distance from source to vertex is replaced by alternating paths to get shortest distance between source and the vertex. The result is calculated in last process and calculation is done in synchronised manner. Algorithm allows efficient calculation of shortest path from source to all vertices in given graph.

CUDA CODE:

We have two kernel functions :

\_\_global\_\_ void closetNode( ) helps to find closet vertex using 1D grid with 1D block

\_\_global\_\_ void relax( ) helps to calculate the path between vertex and its closet vertex and using 3D grid with 3D block where ,dim3 ngrid(1,1,1),dim3 nblock(V,1,1) here V-number of vertices.

We take input graph and is subjected to bellman-ford algorithm to get modified graph, node\_dist[ ] array gives distance between vertices and visited\_node[] array tells if vertex was visited or not. We use cudaEvent( ) to calculate execution time.

In this also we parallelize the dijkstra’s algorithm and works similar way as in MPI.

void bellman\_ford( ) is non kernel function which does same work as in MPI to get new graph with non-negative weights.

In terms of optimization and acceleration of algorithm, we emphasised on accelerating Dijkstra’s algorithm, because majority of calculation are executed in it and Bellman-ford algorithm is only used once when initially input graph is re-weighted.

**Results:**

|  |  |  |
| --- | --- | --- |
| Method of implementation | Execution time with 4 vertices (in ms) | Execution time with 5 vertices (in ms) |
| Serial | 0.284000 | 0.640800 |
| MPI | 0.229343 | 6.008369 |
| CUDA | 0.072488 | 0.077656 |

Note: Source is taken as node 0.

**Conclusions:**

The shortest path from source node to all the nodes in the graph are found and the efficiency of the parallel algorithm is higher.

**Limitations and Possible Improvements:**

1. We have only computed shortest paths from a single source node to every node while we should be able to compute shortest path from every node to every other node.
2. The algorithm does not work for cyclic graphs and may fail to give correct values of distances.

**References :**