Parallelization of Dijkstra's Shortest Path Algorithm using OpenMP and MPI with Performance Analysis

Jared Reiling
CMSE 822: Parallel Computing
Department of Computational Mathematics, Science, and Engineering, Michigan State University

Introduction

- Finding the shortest path in a graph is a necessary algorithm implemented in computational sciences
- Applications include computational neuroscience, topological data analysis, and supply-chain management [1]
- Algorithm works by calculating shortest path from source node to all other nodes in the graph. Returns the distances between all nodes from source and shortest path [2]
- Calculated Dijkstra's algorithm for every node as a source point

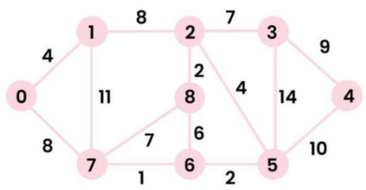
Key Points

- Significant speedup occurred upon implementation of OpenMP and MPI
- Implemented balanced workload in threads/ranks through domain decomposition
- All MPI implementations have exponential decrease in run time
- Graphs using OpenMP with size 100 and 1000, number of threads exponentially decrease runtime
- Graphs of size 10 with OpenMP did not experience speedup with parallelization

References

1. Mengsen Zhang, Samir Chowdhury, Manish Saggar; Temporal Mapper: Transition networks in simulated and real neural dynamics. Network Neuroscience 2023; 7 (2): 431–460. doi: https://doi.org/10.1162/netn_a_00301
2. "Dijkstra's algorithm." wikipedia. Wikimedia Foundation, 20 April 2024, https://en.wikipedia.org/wiki/Dijkstra's_algorithm
3. "How to find Shortest Paths from Source to all Vertices using Dijkstra's Algorithm." geeksforgeeks, 20 April 2024, https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

Implementation of Dijkstra's Algorithm

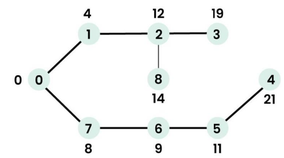


Common graph between serial and parallel algorithms [3]

Implementation of Dijkstra's algorithm [2]

```
function Dijkstra(Graph, source):

    for each vertex v in Graph.Vertices:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q

        for each neighbor v of u still in
Q:
            alt ← dist[u] + Graph.Edges(u,
v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

- Serial
- Parallel (OpenMP)
- Parallel (MPI)



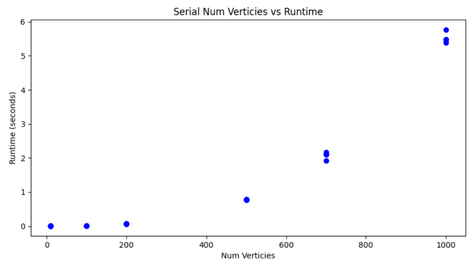
Calculate shortest path

```
Shortest Path for source node = 0
Vertex              Distance from Source
0                                   0
1                                   4
2                                   12
3                                   19
4                                   21
5                                   11
6                                   9
7                                   8
8                                   14
Time for shortest path distance: 0.000002 seconds
```

Verify correct shortest path calculation for serial and parallel implementation with changing ranks/threads

Parallelization produced significant speedup with larger graphs

Serial Implementation



Serial:

- Time complexity of algorithm is $O(V^2)$ where V is the number of nodes. Performance is as expected.

MPI:

- Discretized domain where each thread has a balanced workload
- For larger node graphs, exponential decrease in run time as number of threads increase

OpenMP:

- For graph size = 10, linear relationship between run time and number of threads
- Graph sizes = 100 and graph sizes = 1000, exponential decrease in run time as number of threads increases
- Outliers due to jitter when "fighting" for resources in HPCC

Parallel Implementations