**Performance Analysis: Parallelizing Dijkstra’s Algorithm**

**Submitted by : Kamalnadh Kundla**

**Introduction**

This report presents the implementation and performance analysis of a parallelized version of Dijkstra's algorithm for shortest path computation in a weighted graph. The algorithm leverages OpenMP for parallelization, comparing its performance against a sequential implementation. The goal is to evaluate the speedup and runtime efficiency for graphs of varying sizes.

**Parallelization Strategy**

1. **Key Parallelized Steps**:
   * **Finding the Minimum Distance Node**:
     + Implemented using #pragma omp parallel for reduction(min:), which allows threads to collaboratively identify the node with the smallest distance.
   * **Updating Neighbor Distances**:
     + Each thread updates the distances for neighbors of the current node. Synchronization is ensured using a critical section (#pragma omp critical).
2. **Workload Balancing**:
   * Dynamic scheduling (schedule(dynamic, chunk\_size)) ensures even distribution of work across threads, especially for graphs with varying node degrees.
3. **Graph Representation**:
   * **Adjacency List**:
     + Efficient traversal of neighbors reduces unnecessary computations and improves thread utilization.
4. **Connectivity Check**:

Check with the DFS algorithm whether the graph is connected or not, if the graph is connected with all other nodes, then its find the shortest path to every node from the source else some of the nodes would be carried out to Infinity.

**Performance Analysis**

The algorithm was tested on graphs of varying sizes. Both **sequential** and **parallel** implementations were evaluated based on execution time and speedup.

| **Graph Size (Nodes, Edges)** | **Sequential Time (s)** | **Parallel Time (s)** | **Speedup** |
| --- | --- | --- | --- |
| 1000, 5000 | 0.008970 | 0.090991 | 0.10 |
| 5000, 20000 | 0.121640 | 0.668216 | 0.18 |
| 10000, 100000 | 0.493490 | 1.712201 | 0.29 |
| 100000, 5000000 | 41.458297 | 19.537060 | 2.12 |

**Graphs**

1. **Execution Time Comparison**:
   * The graph below shows the execution times for sequential and parallel implementations.
   * A graph with blue and orange lines

     Description automatically generated
2. **Speedup Achieved**:
   * The graph below illustrates the speedup achieved for different graph sizes.
   * A graph with blue bars

     Description automatically generated

**Challenges**

1. **Synchronization Overhead**:
   * The use of critical sections for updating shared data (e.g., distances array) introduced delays, especially for small graphs.
2. **Sparse Graphs**:
   * Many threads were underutilized when processing sparse graphs due to a lack of neighboring nodes to process.
3. **Memory Contention**:
   * Shared access to arrays like distances[] and visited[] resulted in contention among threads, limiting scalability.

**Lessons Learned**

1. **Dynamic Scheduling**:
   * Significantly improved performance by balancing workload dynamically across threads.
2. **Critical Sections vs. Atomic Operations**:
   * Critical sections ensured correctness but added overhead. Exploring atomic operations or lock-free designs could reduce synchronization costs.
3. **Graph Representation**:
   * The adjacency list structure reduced memory usage and unnecessary computations, improving efficiency for larger graphs.

**Conclusion**

The parallel implementation demonstrated significant improvements in runtime for larger graphs. A speedup of **2.12x** was achieved for the largest tested graph (100,000 nodes, 5,000,000 edges). However, for smaller graphs, the overhead of parallelization outweighed the benefits. Further optimizations such as lock-free data structures, hybrid parallelism (e.g., OpenMP + MPI), and graph partitioning could yield better results.

**Screenshots**

A white background with black text

Description automatically generated

A screenshot of a computer program

Description automatically generated