

Report on Single-Source Shortest Path (SSSP) in Dynamic Road Networks

Kainat Khalid 22I-2242

Ibrahim Umar 22I-2315

Introduction:

The Single-Source Shortest Path (SSSP) problem aims to find the shortest paths from a source node to all other nodes in a graph. Traditional algorithms like Dijkstra's and Bellman-Ford assume static graphs, where edges and their weights remain constant. However, many real-world networks, such as road networks, are dynamic. In dynamic road networks, traffic conditions change frequently, leading to changes in edge weights (travel times) or even the addition/removal of roads. Recomputing the SSSP from scratch after each change is computationally expensive, especially for large networks. This report analyzes several approaches to address the SSSP problem in dynamic road networks, focusing on parallel computing techniques.

Problem Statement:

The core challenge is to efficiently maintain shortest path information in a road network where edge weights (travel times) change dynamically. Naive recomputation is inefficient. The goal is to develop algorithms and implementations that can incrementally update the shortest paths when changes occur, minimizing the computational cost.

Approaches and Implementations:

We have examined several implementations for solving the SSSP problem, with a focus on adapting them for dynamic road networks.

1. Sequential Implementation:

- **Algorithm:** The provided code implements the Bellman-Ford algorithm. Bellman-Ford can handle negative edge weights, but it's generally less efficient than Dijkstra's for graphs without negative weights.
- **Implementation:**
 - The code reads the road network graph from a file ("road_network.txt"). The file is assumed to contain the number of nodes and edges, followed by the source, destination, and weight for each edge.
 - The `bellmanFord` function initializes the distances to all nodes as infinity, except for the source node, which is set to 0. It then iteratively relaxes the edges, updating the distances if a shorter path is found.

- The main function reads the graph, calls bellmanFord, and prints the distance from the source node (0) to node 10.
- **Dynamic Adaptation:** To adapt this for dynamic updates, we would need to:
 - Store the graph data in a suitable data structure that allows efficient updates (e.g., adjacency list).
 - Identify the edges affected by a change.
 - Rerun the Bellman-Ford algorithm, but potentially only on the affected part of the graph. This is difficult with standard Bellman-Ford. Dijkstra's algorithm with a priority queue would be more efficient for static graphs without negative edge weights, and could be adapted, but still requires significant recomputation in dynamic scenarios.
- **Limitations:** The sequential implementation is inefficient for large, dynamic road networks. Each update requires a complete or partial re-computation.

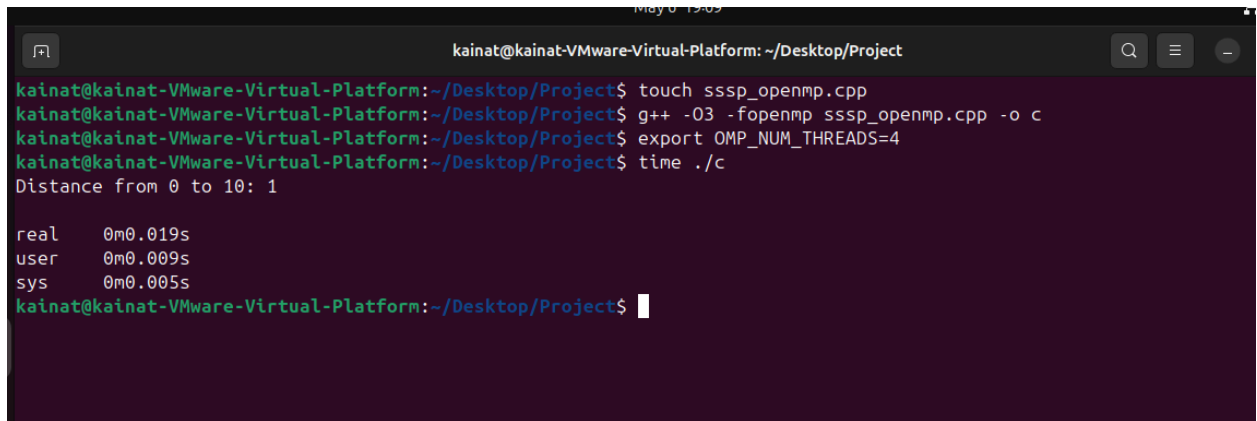
```
kainat@kainat-VMware-Virtual-Platform: ~/Desktop/Project
kainat@kainat-VMware-Virtual-Platform:~/Desktop/Project$ touch sssp_sequential.cpp
kainat@kainat-VMware-Virtual-Platform:~/Desktop/Project$ g++ -O3 sssp_sequential.cpp -o a
g++: error: unrecognized command-line option '-O3'
kainat@kainat-VMware-Virtual-Platform:~/Desktop/Project$ g++ -O3 sssp_sequential.cpp -o a
kainat@kainat-VMware-Virtual-Platform:~/Desktop/Project$ time ./a
Distance from 0 to 10: 1

real    0m0.075s
user    0m0.014s
sys     0m0.015s
kainat@kainat-VMware-Virtual-Platform:~/Desktop/Project$
```

2. Parallel Implementation with OpenMP:

- **Algorithm:** This code also uses the Bellman-Ford algorithm but parallelizes it with OpenMP.
- **Implementation:**
 - The structure is similar to the sequential version, but the core Bellman-Ford loop is parallelized using `#pragma omp parallel for`.
 - A critical section (`#pragma omp critical`) is used to protect updates to the dist array, ensuring thread safety. An updated flag and an outer if condition are used to reduce the number of times the critical section is entered.
 - The code includes an optimization to exit the loop early if no distances are updated in an iteration, potentially improving performance.
- **Dynamic Adaptation:**
 - Similar to the sequential version, efficient dynamic updates require identifying affected areas and applying targeted updates. OpenMP helps with the update phase, but the identification of the affected region would need to be handled.

- **Advantages:** OpenMP provides a relatively straightforward way to parallelize the Bellman-Ford algorithm, potentially leading to significant speedups on multi-core systems.
- **Limitations:** OpenMP is suitable for shared-memory systems. For very large road networks, a distributed-memory approach is needed. The critical section can become a bottleneck if there is high contention. Bellman-Ford is still less efficient than Dijkstra's (for static graphs) and more complex to adapt to dynamic changes.



```

kainat@kainat-VMware-Virtual-Platform: ~/Desktop/Project
kainat@kainat-VMware-Virtual-Platform:~/Desktop/Project$ touch sssp_openmp.cpp
kainat@kainat-VMware-Virtual-Platform:~/Desktop/Project$ g++ -O3 -fopenmp sssp_openmp.cpp -o c
kainat@kainat-VMware-Virtual-Platform:~/Desktop/Project$ export OMP_NUM_THREADS=4
kainat@kainat-VMware-Virtual-Platform:~/Desktop/Project$ time ./c
Distance from 0 to 10: 1

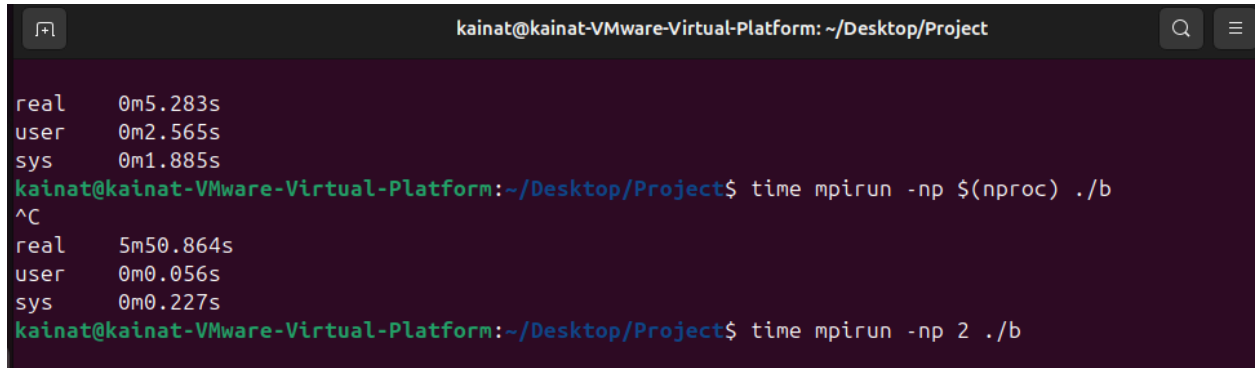
real    0m0.019s
user    0m0.009s
sys     0m0.005s
kainat@kainat-VMware-Virtual-Platform:~/Desktop/Project$

```

3. Parallel Implementation with MPI:

- **Algorithm:** This implementation uses MPI for distributed-memory parallelism. It employs a simplified version of the Bellman-Ford algorithm.
- **Implementation:**
 - The graph is partitioned, and each MPI process works on a subset of the edges. The code assumes that the graph has already been partitioned (using METIS, as mentioned in the "Literature Review Summary.pdf"). Each process reads its portion of the graph from a file (e.g., "partition_0.txt").
 - MPI_Allreduce is used to ensure that all processes have the latest distance information. This is a global synchronization point.
- **Dynamic Adaptation:**
 - MPI allows for distributing the graph across multiple machines, which is essential for very large road networks.
 - When an edge changes, the change needs to be communicated to all processes that contain edges affected by the change. This communication can be a significant overhead.
 - The MPI_Allreduce call in each iteration acts as a global synchronization point, which can be a performance bottleneck.
- **Advantages:** MPI enables the processing of massive graphs that cannot fit into the memory of a single machine.

- **Limitations:** MPI introduces communication overhead, especially with MPI_Allreduce. Efficiently handling dynamic updates requires careful management of data distribution and communication. Load balancing across processes is crucial.



```

kainat@kainat-VMware-Virtual-Platform: ~/Desktop/Project
real    0m5.283s
user    0m2.565s
sys     0m1.885s
kainat@kainat-VMware-Virtual-Platform:~/Desktop/Project$ time mpirun -np $(nproc) ./b
^C
real    5m50.864s
user    0m0.056s
sys     0m0.227s
kainat@kainat-VMware-Virtual-Platform:~/Desktop/Project$ time mpirun -np 2 ./b

```

4. Parallel Implementation with OpenCL:

- **Algorithm:** The OpenCL implementation uses a kernel function (sssp_update) to perform the distance updates on a GPU. It assumes the graph is stored in Compressed Sparse Row (CSR) format, which is efficient for GPU processing.
- **Implementation:**
 - The C++ host code sets up the OpenCL environment, loads the kernel, reads the graph data, transfers the data to the GPU, executes the kernel, and reads the results back.
 - The OpenCL kernel performs the distance updates. It uses atomic_min to ensure thread-safe updates to the distance array.
- **Dynamic Adaptation:**
 - For dynamic updates, the OpenCL buffers on the GPU would need to be updated. This involves transferring the changed edge data from the host to the GPU.
 - The kernel would then need to be re-executed. Efficiently handling dynamic updates requires minimizing data transfers between the host and the GPU.
- **Advantages:** GPUs can provide massive parallelism, potentially leading to significant speedups for SSSP calculations. CSR format is well-suited for GPU processing.
- **Limitations:** OpenCL introduces the overhead of transferring data between the host and the GPU. Atomic operations, while thread-safe, can introduce some performance overhead. Efficiently handling dynamic updates requires minimizing these transfers and kernel launches.

Analysis:

- **Problem:** Traditional SSSP algorithms are inefficient for dynamic networks.
- **Proposed Solution:** An incremental update approach that focuses on updating only the affected subgraphs.
- **Parallelization:**
 - MPI for inter-node parallelism (graph partitioning and distributed processing).
 - OpenMP for intra-node parallelism (parallelizing the update steps within each node).
- **GPU (Alternative):** Using CUDA and a Vertex-Marking Functional Block (VMFB) to optimize GPU execution.
- **Data Structures:** Storing the graph in CSR format for efficient memory access on GPUs.
- **Challenges and Mitigations:**
 - **Load Imbalance:** Using METIS for balanced graph partitioning and dynamic scheduling in OpenMP.
 - **Synchronization Overhead:** Limiting MPI synchronization to boundary vertices and tolerating redundant OpenMP updates.

Comparison and Discussion:

Feature	Sequential (Code 2)	OpenMP (Code 3)	MPI (Code 4)	OpenCL (Codes 5 & 6)
Parallelism	No	Shared Memory	Distributed Memory	GPU
Scalability	Poor	Limited	Excellent	Very Good
Dynamic Updates	Inefficient	Moderately	Complex	Complex
Implementation	Simple	Moderate	Complex	Complex
Suitable For	Small graphs	Medium graphs	Very large graphs	Computation-intensive
Memory Model	Single Node	Shared Memory	Distributed Memory	Device Memory
Key Bottleneck	Computation	Thread contention	Communication	Data Transfer, Atomic Ops

Conclusion:

For dynamic road networks, the sequential implementation is not practical due to its inefficiency in handling updates. Parallel approaches are necessary to achieve acceptable performance.

- OpenMP can provide good speedups for medium-sized networks on shared-memory systems, but it may face limitations in scalability and thread contention for very large graphs.
- MPI offers excellent scalability for massive road networks by distributing the graph across multiple machines. However, it introduces communication overhead and requires careful management of data distribution and synchronization.
- OpenCL can leverage the massive parallelism of GPUs, making it suitable for computation-intensive SSSP calculations. However, it involves the overhead of transferring data between the host and the GPU, and efficient handling of dynamic updates requires minimizing these transfers.

The most promising approach for large-scale dynamic road networks is a hybrid approach that combines MPI and OpenMP, as outlined in the "Literature Review Summary.pdf". This approach leverages MPI for distributed-memory parallelism and OpenMP for shared-memory parallelism within each node. Efficiently handling dynamic updates in such a system requires careful consideration of:

- **Graph Partitioning:** Using tools like METIS to partition the graph and minimize edge cuts.
- **Data Distribution:** Distributing the graph data across MPI processes.
- **Communication:** Minimizing communication between MPI processes, especially global synchronization.
- **Incremental Updates:** Developing algorithms that can efficiently update the shortest paths when edge weights change, without recomputing the entire graph.
- **Load Balancing:** Ensuring that the workload is evenly distributed across all processes and threads.

Further research and development are needed to optimize these parallel algorithms and data structures for dynamic road networks.