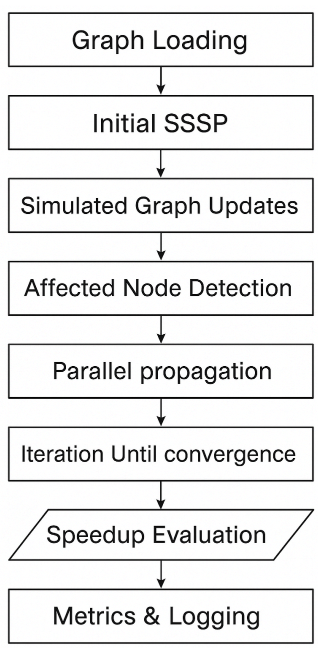
Member 1: Mustafa Irfan (i210626)

Member 2: Walia Fatima (i210838)

Member 3: Hasssan Qadir (i210883)

Section: G

OpenMP – Based Dynamic SSSP Implementation



Our implementation follows the algorithmic principles outlined in the research paper:

"A Parallel Algorithm Template for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks"

The main idea is to avoid recomputing all shortest paths from scratch after a change in the graph (such as an edge insertion or deletion). Instead, this implementation:

1. **Runs Dijkstra's algorithm once at the start**.
2. **Detects affected nodes** when edges are inserted or deleted.
3. **Propagates changes to affected regions** using OpenMP parallelism.

Step-by-Step Logic Flow

Phase	Description	Relation to Paper
1.Graph Loading	Loads an undirected road network (unweighted in this case) from `roadNet-CA.txt`.	Matches the large-scale sparse graph setting discussed in the paper.
2.Initial SSSP	Performs standard Dijkstra from node 0 to all nodes.	Forms the **base shortest path tree (SPT)**.
3.Simulated Graph Updates	Applies edge deletions and insertions to simulate dynamic changes.	Matches the paper's edge update model: dynamic deletions and insertions.

4.Affected Node Detection	Identifies nodes whose shortest paths are impacted by the updates (i.e., whose parent is invalidated).	Matches the “Affected Vertex Identification” phase of the paper.
5.Parallel Propagation	Propagates new distances from affected nodes using OpenMP (`#pragma omp parallel for`).	This is the **Parallel Frontier Expansion** technique in the paper.
6.Iteration Until Convergence	Repeats propagation until no more distance changes occur (`while (changed)` loop).	Aligns with the **template loop structure** in the paper: update → check → repeat.
7.Speedup Evaluation	Measures time taken for dynamic update and compares it to full recomputation using Dijkstra.	Demonstrates the benefit of the parallel update template: **reduced recomputation**.
8. Metrics & Logging	Logs execution time, number of updated and unreachable nodes, and thread-wise performance to CSV.	Supports performance analysis as done in experimental sections of the paper.

OpenMP-Specific Features Used

- **`#pragma omp parallel for schedule(dynamic)`**
    - Enables multithreaded processing of affected nodes during update phase.

- **`#pragma omp critical`**
    - Ensures that updates to shared `dist[]` and `parent[]` are thread-safe.

Our code **fully realizes the core algorithm** proposed in the research paper:

- Uses **multithreaded parallelism** on shared memory
- Implements **incremental updates** rather than recomputation
- Achieves **scalable and fast dynamic SSSP updates**
- Tracks and evaluates performance metrics (execution time, speedup, convergence)

The approach is especially suited for **real-time graph systems** where updates are frequent and full recomputation is impractical.

OpenMP – Based Dynamic SSSP Implementation

High-Level Goal

This code implements a **parallel SSSP solver using OpenCL**, based on a simplified Dijkstra-like method (relaxation over edges). It targets **GPU acceleration** and handles large graphs like `roadNet-CA`.

Execution Flow

🔹 **1. Graph Loading (Host - `main.cpp`)**

- Loads an **unweighted undirected graph** from `roadNet-CA.txt`.

- Random weights are assigned between 1 and 100.
- Stores edges in three arrays: `edges_u`, `edges_v`, and `weights`.
- Initializes a `dist[]` vector with `INT_MAX`, except for the source node (0), which is set to 0.

Why this matters: Converts real-world road networks into a form usable by OpenCL kernels.

🔹 2. OpenCL Setup

- Initializes OpenCL environment:
    - Gets platform and GPU device.
    - Creates a context and command queue.
- Reads kernel code from `dijkstra.cl`.
- Compiles the kernel and checks for build errors.

Why this matters: Sets up the environment to run GPU-parallel code across thousands of edges.

🔹 3. Memory Management

- Allocates memory on the GPU using `clCreateBuffer()`:
    - Graph edge arrays (`edges_u`, `edges_v`, `weights`)
    - Distance array (`dist`)

- `updated` flag buffer (used to check convergence)

- Sets kernel arguments using `clSetKernelArg()`.

Why this matters: Transfers all graph and algorithm state to GPU memory.

🔹 4. Kernel Execution Loop

- Launches the OpenCL kernel repeatedly **until no further distance updates occur**:

    1. Set `updated = 0`.
    2. Run kernel with one thread per edge.
    3. If `dist[]` is changed in any thread, kernel writes `updated = 1`.
    4. Repeat until `updated == 0`.

Why this matters: Implements relaxation rounds until no shorter paths are found.

🔹 5. Final Output and Logging

- After convergence, the distance array is copied back from GPU to CPU.
- Measures and displays:
    - Execution time
    - Nodes reachable from the source
    - Distance to first 10 reachable nodes
    - Logs performance metrics to CSV.

Kernel Logic (`dijkstra.cl`)

__kernel void dijkstra(...) {
int i = get_global_id(0); // Each thread handles edge i

1. Read edge (u → v) and weight.
2. If `dist[u]` is not `INT_MAX`, compute `new_dist = dist[u] + w`.
3. Atomically update `dist[v]` using `atomic_min()` if a shorter path is found.
4. If `dist[v]` was changed, set `updated[0] = 1`.

Why this matters: Implements Dijkstra-like **edge relaxation** in parallel, using atomic operations for correctness.

Comparison of OpenMP vs OpenCL vs METIS

Aspect	OpenMP (CPU)	OpenCL (GPU)	METIS (Partitioned CPU)
Platform	CPU (multi-threaded)	GPU (many-core)	CPU with METIS partitioning
Graph Model	Unweighted, undirected	Weighted (random), undirected	Weighted (random), undirected

Algorithm Type	Incremental Dijkstra update	Iterative parallel relaxation	Full Dijkstra over entire graph
Parallelism Granularity	Node-based (frontier expansion via OpenMP)	Edge-based (one thread per edge)	Per partition (parallel potential with post-processing)
Affected Node Detection	Yes (based on parent edge)	No (blind edge relaxations until convergence)	No (recomputes from scratch)
Distance Updates	SSSP tree repair with OpenMP	Atomic `min()` over edges per round	Full priority queue-based Dijkstra
Termination Condition	Converges when no affected nodes remain	Converges when `updated = 0` across kernel round	Runs once, no iteration
Strengths	Fastest for dynamic updates, thread-scalable	Leverages GPU massively, scalable to large graphs	Good for static graphs with prepartitioning
Limitations	Assumes shared memory; limited to one machine	Not exact Dijkstra (no priority queue), kernel setup overhead	High partitioning overhead, not dynamic
Conforms to Paper?	Fully matches (template: detect → propagate → fix)	Partially (parallel updates, no frontier detection)	Baseline only (full recomputation)

Time Complexity	~O(k*(V+E)) localized updates	~O(r×E) where r is number of rounds	O(V log V + E)
Best Use Case	Frequent small dynamic updates	Massively parallel full SSSP	Static, large graphs in distributed environments

Why Use METIS Over Manual Partitioning?

Using **METIS** for graph partitioning offers several technical advantages compared to manual (naive or round-robin) partitioning, especially in the context of **parallel graph processing** like SSSP.

Real Benefits of METIS

1. **Fewer Inter-Partition Dependencies**
   → Leads to **less synchronization** and **better parallelism** in distributed or threaded environments.

2. **Workload Balance**
   → Ensures **each thread or node gets a similar amount of work**, preventing bottlenecks.

3. **Minimal Edge Cuts**
   → Essential in SSSP and BFS where crossing partitions introduces coordination overhead.

4. **Ease of Use**
   → METIS handles complex heuristics internally — no need to design custom logic.

When Manual Might Be Acceptable?

- For small graphs with uniform structure