Name: Kainat Khalid ***Roll-no: 22i-2242***

Name: Ibrahim Umar Malik ***Roll-no: 22i-2315***

Section:CS-C

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

1. Problem Statement

- **Core Challenge:** Traditional SSSP algorithms (e.g., Dijkstra's) assume static graphs, but real-world networks (e.g., social, transportation) are dynamic—edges/weights change over time.
- **Gap:** Recomputing SSSP from scratch after each change is inefficient for large-scale dynamic networks.
- **Goal:** Develop a parallel framework to *update* SSSP incrementally when edges are added/deleted, minimizing redundant computations.

2. Proposed Parallel Algorithm

Key Idea:

- **Step 1:** Identify subgraphs affected by edge changes (parallelizable).
- **Step 2:** Update only affected subgraphs iteratively (avoids global recomputation).

Parallelization Strategy:

- **MPI (Inter-node):**

- Partition graph using **METIS** (not explicitly mentioned but implied for distributed memory).
- Each process handles a subset of vertices/edges.

- **OpenMP (Intra-node):**
    - Parallelize edge relaxation and tree updates within a node using dynamic scheduling.
- **GPU (Alternative):**
    - Uses CUDA with a **Vertex-Marking Functional Block (VMFB)** to avoid atomic operations and reduce synchronization overhead.

Data Structures:

- **SSSP Tree:** Stored as an adjacency list with `Parent`, `Dist`, and `Affected` flags per vertex.
- **Dynamic Updates:**
    - **Edge Insertion:** Update distances if a shorter path is found.
    - **Edge Deletion:** Disconnect affected subtrees and mark vertices for reprocessing.

3. Results

- **Speedup:**
    - **GPU:** Up to **5.6× faster** than Gunrock (static recomputation) for 100M edge changes (≥50% insertions).
    - **CPU (OpenMP):** Outperforms Galois by **up to 5×** for similar conditions.
- **Scalability:**
    - Efficiently handles graphs with **millions of vertices/edges** (e.g., LiveJournal: 12.6M vertices, 161M edges).
    - Performance degrades if >75% changes are deletions (recomputation becomes better).

4. Key Contributions

1. **Unified Framework:** Works for both CPUs (shared memory) and GPUs.
2. **Iterative Convergence:** Avoids locks by iteratively updating distances until stability.

3. **Load Balancing:** Uses dynamic scheduling for uneven workloads (e.g., varying subtree sizes).
4. **Batch Processing:** Handles **100M edge changes** efficiently by processing in batches.

5. Tools/Datasets Used

- **Graphs:** Real-world (e.g., LiveJournal, Orkut) and synthetic R-MAT networks.
- **GPU:** NVIDIA Tesla V100 (CUDA).
- **CPU:** Intel Xeon Gold (OpenMP).

Parallelization Strategy for Dynamic SSSP Updates

(Based on Paper 3: "A Parallel Algorithm Template for Updating SSSP in Dynamic Networks")

1. MPI Strategy (Inter-Node Parallelism)

- **Graph Partitioning with METIS:**
  - Use **METIS** to split the graph into `k` partitions (one per MPI process).
  - **Partitioning Goal:** Minimize edge cuts while balancing vertex counts across nodes.
  - **Output:** Each MPI process gets a subgraph + boundary vertices (shared with neighbors).
- **Communication Patterns:**
  - **Master-Worker:** Designate one process (e.g., rank 0) to coordinate batch updates.
  - **All-to-All:** Sync `Dist`/`Parent` arrays for boundary vertices after each iteration (MPI_Allreduce).
  - **Sparse Updates:** Only communicate changes to affected vertices (MPI_Isend/MPI_Irecv).

2. OpenMP Strategy (Intra-Node Parallelism)

- **Thread-Level Parallelism:**
  - Parallelize loops in **Step 1** (identifying affected vertices) and **Step 2** (updating distances) using `#pragma omp parallel for schedule(dynamic)`.

- Dynamic Scheduling: Handles uneven workloads (e.g., subtrees of varying sizes).

- **Critical Sections:**
  - Avoid locks by using **iterative convergence** (no atomic updates; recompute until stable).
  - Example:

cpp

Copy

Download

```cpp
#pragma omp parallel for
for (Vertex v : AffectedVertices) {
  relax_neighbors(v); // No locks; redundant updates allowed
}
```

3. METIS Integration

- **Preprocessing:**
  - Run METIS on the initial graph to generate partitions (`mpmetis -kway graph.txt N`, where `N` = MPI ranks).
- **Dynamic Updates:**
  - After edge changes, re-partition only if imbalance exceeds a threshold (e.g., >20% vertex count variance).
  - **Optimization:** Cache partition data to avoid recomputing for small changes.

4. OpenCL Alternative (GPU Acceleration)

(Optional, if GPUs are available)

- **Kernel Design:**
  - **Step 1 (Edge Processing):** Launch one GPU thread per changed edge (massively parallel).
  - **Step 2 (Vertex Updates):** Use **VMFB** (Vertex-Marking Functional Block) to:

1. Mark affected vertices in parallel (no atomics).
2. Filter/Sync via GPU ballot operations.

- **Data Structure:** Store graph in CSR format for coalesced memory access.

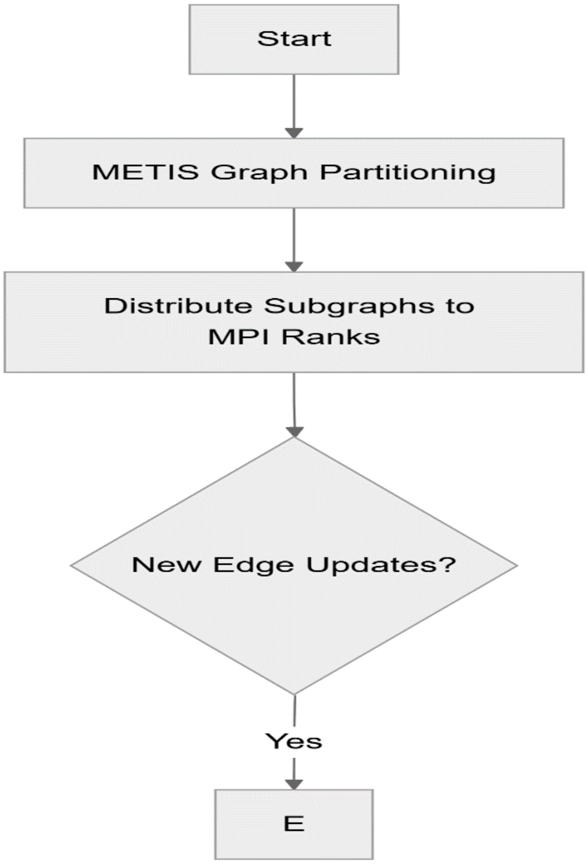
Implementation Workflow

1. **Initialization:**
   - Partition graph with METIS → Distribute subgraphs to MPI processes.
   - Compute initial SSSP sequentially (Dijkstra's) on each partition.
2. **Dynamic Update:**
   - **MPI:** Broadcast batch of edge changes (`Ins_k`, `Del_k`).
   - **OpenMP:** Parallelize Steps 1–2 within each node.
   - **Sync:** Aggregate updates to boundary vertices via MPI.
3. **Termination:**
   - Check global convergence (no further distance updates).

Challenges & Mitigations

- **Load Imbalance:**
  - Use METIS to balance partitions; dynamic scheduling in OpenMP.
- **Synchronization Overhead:**

- Limit MPI syncs to boundary vertices; tolerate redundant OpenMP updates.



Key Interactions Explained:

1. **METIS (Yellow):**

- Partitions the initial graph into `k` subgraphs (1 per MPI rank).
- *Output:* Balanced partitions with minimal edge cuts.

2. **MPI (Purple):**

- Distributes subgraphs to ranks.
- Broadcasts edge updates (`Ins_k/Del_k`) to all ranks.
- Synchronizes boundary vertex distances after local updates.

3. **OpenMP (Blue):**

- Within each MPI rank:
  - **Step 1:** Parallel edge processing (mark affected vertices).
  - **Step 2:** Parallel distance updates (dynamic scheduling).

4. **Convergence Check:**

   - Iterates until no further distance updates occur (global sync via MPI).