Parallel Dynamic SSSP Update Framework

Team Members:

Mesam E Tamaar Khan	22i-1304	CS-D
Aamna Saeed	22i-1179	CS-D
Syed Tashfeen Hassan	22i-0860	CS-D

Parallel Dynamic SSSP Update Framework

This document outlines sequential and parallel approaches for maintaining and updating the Single Source Shortest Path (SSSP) tree in a dynamic graph, particularly under edge insertions and deletions.

---

1. Sequential SSSP Update Algorithm

- **Applicable Changes**: Only edge insertions and deletions.
- **Process**:
    1. Identify affected vertices and insert them into a priority queue.
    2. Use Dijkstra's algorithm to recompute and update distances.

---

2. CPU-Based Implementation

- **Approach**: Shared memory implementation using OpenMP [7].
- **References**: [16], [17], [18], [19] provide relevant background and optimization details.
- **Features**:
    - Asynchronous updates.
    - Efficient for moderate-sized graphs.
    - Capable of processing batches of graph changes.

---

3. GPU-Based Implementation

- **Framework**: Gunrock-based architecture (Advance, Filter, Compute).
- **Algorithm**: Modified Bellman-Ford using two queues on Kepler GPU.
- **Key Paper**: [15]—efficient for small-scale updates (<10%).

Core Concepts

- **SIMT Model**: Single Instruction, Multiple Threads using CUDA.
- **Graph Storage**: Compressed Sparse Row (CSR) format for memory efficiency.
- **Parallel Functions**:
    - `ProcessDel`: Handles deletions, marks and disconnects vertices.
    - `ProcessIns`: Processes insertions and updates vertex distances.
    - `DisconnectC`: Recursively disconnects subtrees.

- `ChkNbr`: Updates distances by checking neighbors.

Vertex-Marking Functional Block (VMFB)

A three-phase mechanism for marking affected vertices:

1. **Vertex Marking**: Each thread marks affected vertices.
2. **Global Synchronization**: Barrier to ensure phase completion.
3. **Filter Phase**: Uses `ballot_sync` and `atomicAdd` to collect final affected set.

GPU Update Algorithm (Algorithm 5)

1. Initialize flags to track affected vertices.
2. **First VMFB**: Run `ProcessDel` to mark deletion-affected vertices (AffDel).
3. **Second VMFB**: Run `ProcessIns` to mark insertion-affected vertices (AffIns).
4. **Subtree Disconnection**: Use `DisconnectC` to propagate deletion impact.
5. **Distance Update Loop**:
    - Use `ChkNbr` iteratively until convergence is achieved.

Design Advantages

- Minimizes atomic operations.
- Promotes modularity via VMFB.
- Scalable to large updates (e.g., 100M edges).
- Ensures correctness by:
    - Avoiding race conditions.
    - Preventing cycle formation.

---

4. Parallel Dynamic Framework

This unified parallel model handles dynamic SSSP updates in two stages:

Step 1: Identify Affected Subgraph

- **Parallel Edge Processing**: Every changed edge is processed to detect affected vertices.
- **Edge Deletion**:
    - If edge `(u,v)` is not part of SSSP tree → No effect.
    - If part of SSSP tree:
        - Assume `u` is parent of `v`.

- Set `Dist[v] = ∞`, `Parent[v] = null`.
- Mark `Affected_Del[v]` and `Affected[v]` as `true`.

- **Edge Insertion**:
  - For inserted edge `(u,v)` with weight `w(u,v)`:
    - If `Dist[u] + w(u,v) < Dist[v]`:
      - Update `Dist[v]`, set `Parent[v] = u`.
      - Mark `Affected[v] = true`.

Step 2: Update SSSP Tree

Part 1: Subtree Disconnection

- When a vertex is disconnected (due to edge deletion), all its descendants are also considered disconnected.
- Traverse subtrees from affected vertices and mark them accordingly.

Part 2: Distance Propagation

- For each `Affected[v]`, evaluate neighbors `n`.
  - If `Dist[v] + w(v,n) < Dist[n]`, update `Dist[n]` and set `Parent[n] = v`.
  - Similarly, update `v` if distance through `n` improves.
- Repeat until no further improvements.

---

5. Data Structures

- **SSSP Tree**: Stored as an adjacency list.
- **Per-Vertex Metadata** (arrays of size `V`):
  - `Parent[v]`: Parent of vertex `v`.
  - `Dist[v]`: Distance from source.
  - `Affected[v]`: True if vertex is impacted by any change.
  - `Affected_Del[v]`: True if affected specifically by deletion.

---

6. Scalability Challenges

- **Load Balancing**: Subgraphs vary in size—requires dynamic workload balancing.
- **Synchronization**: Avoid race conditions during updates by using lock-free, iterative strategies.

- **Cycle Avoidance**: Ensure cycle-free updates by first disconnecting subtrees before processing insertions.

---

7. Experimental Evaluation

7.1 Platforms

- **CPU**: Intel Xeon Gold 6148 (384GB RAM), using OpenMP.
- **GPU**: NVIDIA Tesla V100 (32GB), dual AMD EPYC CPUs.

7.2 Datasets

- **Real-World Graphs**: Orkut, LiveJournal, BHJ.
- **Synthetic Graphs**: RMAT, Graph500.
- **Scale**: Graphs range from 1.8M to 16M vertices and up to 258M edges.

7.3 GPU Implementation Results

- **Workload**: Tested with 50M and 100M edge updates across varying insertion/deletion ratios.

Key Observations:

- **100% Insertions**: Achieved faster update times due to bypassing subtree disconnection routines.
- **Higher Deletion Ratios**: Increased update time due to extra overhead from disconnection and reconnection logic.
- **Subgraph Overlap**: Helps avoid redundant computations, boosting performance.

7.4 GPU Performance Comparison

- **Benchmark**: Compared against **Gunrock**, a leading GPU SSSP library (static).

Speedups:

- Up to **8.5×** faster with 50M changes (insertion-dominant workloads).
- Up to **5.6×** faster with 100M edge changes.

Additional Insights:

- When **deletions dominate** (>75%), Gunrock may outperform due to simpler recomputation.
- **Recomputing** becomes preferable when more than 50% of total graph edges are impacted and most changes are deletions.

7.5 Shared-Memory (CPU) Implementation Results

- **Comparison Tool**: **Galois**, a shared-memory SSSP recomputation framework.
- **Test Case**: 100M edge changes, varying insertion percentages.

Performance:

- Outperforms Galois across most networks.
- Performance degrades when >85% of nodes are affected.

Scalability:

- Runtime improves significantly with thread count (especially effective up to **72 threads**).
- Exceptions observed when affected node count was low (<15%).

7.6 Performance Tuning Experiments

Asynchronous Update Level (Figure 9)

- **Higher asynchrony** (i.e., more relaxed synchronization) leads to reduced execution times.
- **Exception**: Slight performance dip on the Orkut dataset with 75% insertions.

Batch Processing of Edge Changes (Figure 10)

- **Batch Sizes**: 15, 30, 50 edge changes per batch.
- **Benefits**:
    - Outperforms non-batched implementations.
    - Particularly effective at **64–72 thread** configurations.
    - Marginal gains at lower thread counts.

---

8. Conclusion

Summary of Contributions

This work presents a novel, **parallel framework** for efficiently updating **Single-Source Shortest Paths (SSSP)** in large **dynamic graphs**. The framework supports both:

- **Shared-memory CPU environments** using OpenMP.
- **GPU architectures** via CUDA, leveraging **Vertex-Marking Functional Blocks (VMFBs)** for scalable parallelism.

The design emphasizes modularity, load balancing, and minimizing synchronization overhead.

Performance Highlights

The proposed update-based framework demonstrates substantial speedups over traditional **recomputation-based approaches**:

- **GPU Implementation**: Achieves up to **8.5× speedup** over **Gunrock** in insertion-dominant scenarios.
- **CPU Implementation**: Outperforms **Galois** across a wide range of datasets and configurations.

The system is particularly effective when the majority of edge updates are insertions and the affected region is relatively sparse.

Main Insight

The framework's **update-based approach** is ideal when:

- **Insertions** dominate the workload.
- Fewer than **~80% of nodes** are affected by the changes.

However, for workloads with **extensive deletions** or when **a majority of the graph is affected**, **full recomputation** can outperform incremental updates.

Future Work

Several promising directions are identified for future exploration:

- **Hybrid Strategy**:
  - Dynamically select between **update-based** and **recomputation-based** approaches depending on the nature of each batch of changes.
- **Predictive Optimizations**:
  - Use prior knowledge or historical patterns of edge changes to proactively optimize update routines.
- **Workload-Aware Performance Tuning**:
  - Study behavior with **non-random** or **localized edge change batches**, which could better reflect real-world scenarios.