Project Report: Parallel SSSP Update in Dynamic Networks

Group Members :

Ahmed Ali	22i-0825
Qusai Mansoor	22i-0935
Ahmad	22i-1288

1. Introduction

1.1. Problem Statement

The Single-Source Shortest Path (SSSP) problem is a fundamental challenge in graph theory, aiming to find the shortest paths from a designated source vertex to all other vertices in a weighted graph. In real-world scenarios such as road networks, social networks, or communication systems, these graphs are often dynamic, meaning their structure (addition/deletion of edges) or edge weights can change over time. Recalculating SSSP from scratch after each change is computationally expensive, especially for large-scale networks. Therefore, efficient algorithms for dynamically updating SSSP trees are crucial.

1.2. Our Approach

This project focuses on implementing and evaluating different approaches to update SSSP in dynamic networks. Our methodology involved:

1. Developing a **naive serial implementation** to serve as a baseline for correctness and performance comparison.
2. Implementing a **distributed-memory parallel version using MPI**, where the graph is partitioned and processed across multiple nodes.
3. Implementing a **shared-memory parallel version using OpenMP**, leveraging multi-core architectures for parallel execution on a single node.

The parallel algorithms are inspired by techniques for handling batch updates (multiple edge changes simultaneously) efficiently, aiming to recompute only the affected parts of the SSSP tree.

2. Serial Implementation

The serial implementation (`Serial.cpp`) provides a foundational SSSP computation and update mechanism. The graph is represented using the Compressed Sparse Row (CSR) format.

- **Initial SSSP**: Dijkstra's algorithm, implemented in the `dijkstra` function, is used to compute the initial SSSP tree from a given source vertex. It utilizes a `std::priority_queue` (min-heap) with a custom comparator to efficiently select the vertex with the smallest tentative distance. Distances are initialized to infinity, and parent pointers are set to -1.
- **Batch Updates**: The `updateSSSPBatch` function processes a list of edge changes (insertions and deletions) provided as a `vector<Edge>` and a corresponding `vector<bool>` indicating if each change is an insertion.
    - **Pre-computation for Deletions**: The `g.buildChildren()` method is called to construct a representation of the current SSSP tree (children for each node), which is used by `markDescendants`.
    - **Process Deletions**:
        - The algorithm iterates through the changes. For each deletion, it checks if the edge `(u,v)` was part of the SSSP tree (i.e., if `g.parent[v] == u` or `g.parent[u] == v`).
        - If the deleted edge was in the tree, the `markDescendants` function is invoked. This function performs a Depth-First Search (DFS) starting from the child vertex whose parent link was severed. It sets the distances of all descendants in that subtree to `INFINITY`, resets their parent pointers, and adds them to a priority queue (`pq`) for re-evaluation.
        - The `g.updateEdgeWeight(u, v, INFINITY)` function is called to effectively remove the edge from the graph by setting its weight to infinity.
    - **Process Insertions**:
        - For each insertion (or edge weight update), `g.updateEdgeWeight(u, v, new_weight)` updates the weight of the edge `(u,v)` (and `(v,u)` for undirected graph) in the CSR structure.
        - The algorithm then checks if this new edge (or updated weight) offers a shorter path to `v` via `u` (i.e., `g.dist[u] + w < g.dist[v]`) or to `u` via `v`. If so, the distance and parent of the affected vertex are updated, and the vertex is added to the priority queue (`pq`).
    - **Relaxation (Re-computation)**: After processing all deletions and insertions, a Dijkstra-like relaxation process is performed. It extracts vertices from the priority

queue (`pq`) one by one. For each extracted vertex `z`, it iterates through its neighbors. If a shorter path to a neighbor is found through `z`, the neighbor's distance and parent are updated, and the neighbor is added to `pq`. This continues until `pq` is empty, ensuring that SSSP properties are restored for all affected parts of the graph.

- **Graph Data Structures**:
  - `struct Graph`: Manages graph data using CSR format (`adj` for concatenated adjacency lists, `row_ptr` for start indices of each vertex's edges, `weights` for edge weights). It also stores `dist` (distances from source) and `parent` arrays.
  - `struct Edge`: A simple structure to represent edges for updates (`u`, `v`, `weight`).
- **Error Handling and Input**: The implementation includes robust error handling using `try-catch` blocks for issues like invalid graph dimensions, out-of-bounds accesses during CSR construction, and file I/O errors. The `load_data` and `load_updates` functions are responsible for reading the graph structure and update batches from text files, respectively, with validation checks.

This serial version serves as a correct baseline. However, for large graphs and frequent or large batches of updates, its sequential nature can lead to significant processing times, motivating the need for parallel approaches.

3. MPI Implementation (Distributed Memory)

The MPI implementation aims to scale the SSSP update process by distributing the graph and computation across multiple processes (nodes).

- **Graph Partitioning**: Although the specific `main.cpp` code for MPI was not detailed in the final inputs, a common and effective strategy for distributed graph processing, which you mentioned considering, is using a library like **METIS**. METIS partitions the graph into roughly equal-sized subgraphs while minimizing edge cuts (edges connecting vertices in different partitions). This helps in balancing the computational load and reducing inter-process communication. Each MPI process would then be responsible for a subgraph.
- **Data Distribution**:
-
  1. Each MPI process stores its local portion of the graph (vertices and their outgoing edges).
  2. Distance (`dist`) and parent (`parent`) arrays are also distributed, with each process managing these values for its assigned vertices.
- **Parallel SSSP Update (General Approach)**: The update process in a distributed setting typically involves iterative local computations and global communication phases:
  1. **Broadcast of Global Information**:

        - Initial graph data or global updates (like the batch of edge changes) are often broadcast from a root process to all other processes using `MPI_Bcast`. The `mpiP` profiler output shows `Bcast` operations, indicating its use for disseminating information.
    2. **Local Processing**: Each process handles updates affecting its local vertices.
        - For deletions, if an edge local to a process (or connecting to its local vertex) that was part of the global SSSP tree is removed, the process updates its local distances and identifies affected local vertices.
        - For insertions, new local edges might create shorter paths for local vertices.
    3. **Communication and Synchronization**:
        - **`MPI_Allgather`**: To propagate changes or share necessary data (e.g., updated distances of boundary vertices, lists of affected vertices), `MPI_Allgather` is often used. This allows each process to receive relevant information from all other processes. The `mpiP` log confirms `Allgather` calls.
        - **`MPI_Reduce`** / **`MPI_Allreduce`**: Global reduction operations are essential for collective decisions or computations. For instance, `MPI_Allreduce` can be used to check if any process made a change in an iteration (e.g., by checking a global "changed" flag), which helps in determining convergence. The `mpiP` log shows `Reduce` and `Allreduce` calls.
        - **`MPI_Barrier`**: Used for synchronization between phases, ensuring all processes complete a certain step before proceeding to the next. This is also evident in the `mpiP` profile.
    4. **Iterative Refinement**: The update process is typically iterative. Processes exchange information about distance updates at their partition boundaries and re-calculate paths. This continues until no more improvements can be made, and the distributed SSSP tree converges to a stable state.
- **Profiling with `mpiP`**: The `mpiP` output provides insights into the MPI communication patterns. It highlights the time spent in various MPI calls (e.g., `Bcast`, `Allgather`, `Reduce`, `Barrier`) and the volume of data exchanged. This information is crucial for identifying communication bottlenecks and understanding the scalability of the MPI implementation. For example, the report shows significant time spent in `Bcast` and `Allgather`, which is common in distributed iterative algorithms where data needs to be shared and synchronized frequently.

4. OpenMP Implementation (Shared Memory)

The OpenMP implementation (`Parallel.cpp`) parallelizes the SSSP update process on a multi-core shared-memory system, drawing inspiration from the algorithms presented in the reference paper.

- **Parallelization Strategy**: OpenMP pragmas (`#pragma omp parallel for`) are used to parallelize the computationally intensive loops over vertices or edges. The `updateSSSPBatch_Paper` function orchestrates the update.
- **Algorithm Overview (based on `Parallel.cpp` and paper concepts)**:
  - **Initialization**:
    - The state of the SSSP tree (parent array) *before* updates is saved (`initial_parent`).
    - `affected_del` (vertices affected by deletion propagation) and `affected` (vertices needing re-evaluation) flags are reset in parallel.
  - **Algorithm 2: Identify Affected Vertices**:
    - **Process Deletions**: A parallel loop iterates through edge deletions.
      - If a deleted edge `(u,v)` was in the `initial_parent` tree, the child vertex is marked: its distance is set to `INFINITY_VAL`, and it's flagged in `affected_del` and `affected`.
      - The edge is marked as deleted in the CSR graph representation (e.g., by setting its target `adj[idx]` to -1 or weight to `INFINITY_VAL`).
    - **Process Insertions**: A parallel loop iterates through edge insertions.
      - Edge weights are updated in the CSR structure.
      - If an insertion `(u,v)` with weight `w` potentially creates a shorter path (e.g., `g.dist[u] + w < g.dist[v]`), `g.dist[v]` and `g.parent[v]` are updated, and `v` is marked in `affected`.
  - **Algorithm 3: Update Affected Subgraph**:
    - **Part 1: Propagate Deletions**:
      - This part iteratively propagates the effect of deletions through the `initial_parent` tree.
      - A `while(deletion_propagated)` loop continues as long as changes occur.
      - Inside, a `#pragma omp parallel for` loop processes vertices currently in `affected_del_snapshot` (a copy of `affected_del` for stable iteration). For each such vertex `v`, its children in the `initial_children` tree (precomputed from `initial_parent`) that haven't already been set to infinity are updated: their distances become `INFINITY_VAL`, parents are nullified, and they are added to `affected_del` and `affected` for the next iteration.
    - **Part 2: Iterative Relaxation**:
      - This part performs a Bellman-Ford-like relaxation for all vertices marked in `affected`.
      - A `while(changed_in_iteration)` loop continues until no more distance improvements are found.

        - Inside, a `#pragma omp parallel for` loop processes vertices in `affected_snapshot`. For each such vertex `v`, it relaxes its outgoing edges: if a shorter path to a neighbor `n_node` via `v` is found, `dist[n_node]` and `parent[n_node]` are updated, and `n_node` is marked in `affected` for the next iteration. The `changed_in_iteration` flag is updated using an OpenMP `reduction`.

- **Load Balancing**: The `schedule(dynamic)` clause is used with `#pragma omp parallel for`. This allows threads to dynamically pick chunks of loop iterations. This is beneficial when the workload per iteration is uneven (e.g., vertices have different degrees, or the impact of an update varies), helping to keep all threads busy and improving overall efficiency.
- **Synchronization**:
    - Implicit barriers at the end of `omp for` loops ensure that all threads complete the loop before proceeding.
    - The `reduction(||:variable)` clause is used to safely update shared flags like `deletion_propagated` or `changed_in_iteration` across threads.
    - Using snapshots of `affected_del` and `affected` arrays at the beginning of each iteration of the `while` loops ensures that threads operate on a consistent view of which vertices to process for that specific iteration.

5. Results and Performance

The performance of the implemented SSSP update algorithms was evaluated by measuring their execution time on various datasets and update batch sizes. The speedup of the parallel implementations (MPI and OpenMP) is calculated relative to the serial implementation.

366923 updates (m/2)

Serial	MPI	OpenMP (4 Threads)
1100ms	2400ms (Not on cluster)	700ms

6. Scalability Analysis

- **Impact of Dataset Size and Updates**: It is generally observed that the benefits of parallelization become more pronounced with larger datasets (more vertices and edges) and larger batches of updates.
    - For **large graphs**, the computational workload is substantial, providing more opportunities for parallel execution to outperform serial computation.

    - For **large update batches**, parallel algorithms can process multiple changes concurrently, significantly reducing the update time compared to a serial approach that handles them sequentially or with less concurrency.
- **MPI Scalability**:
    - The MPI version is designed for distributed memory systems and can, in principle, scale to very large graphs that may not fit into the memory of a single machine.
    - Scalability depends on factors like the efficiency of graph partitioning (e.g., by METIS), the ratio of computation to communication, and the network latency/bandwidth between nodes.
    - The `mpiP` profiling helps in understanding these aspects by showing time spent in communication versus computation. High communication overhead can limit scalability.
- **OpenMP Scalability**:
    - The OpenMP version scales with the number of available cores on a shared-memory machine.
    - Its performance is often limited by memory bandwidth and synchronization overhead (e.g., managing shared flags, implicit barriers).
    - The use of `schedule(dynamic)` helps in load balancing, which is crucial for good scalability, especially when work distribution is irregular.
    - For problems that fit within a single machine's memory, OpenMP can offer good speedups with lower programming complexity compared to MPI.
- **Overall Observation**: When a large number of updates are processed on a large dataset, both MPI and OpenMP approaches are expected to demonstrate a considerable speedup over the serial version. The choice between MPI and OpenMP often depends on the scale of the problem and the available hardware infrastructure.

7. Conclusion

This project successfully implemented and explored serial, MPI-based distributed, and OpenMP-based shared-memory parallel algorithms for updating Single-Source Shortest Paths in dynamic networks. The parallel implementations, inspired by established algorithms, aim to efficiently handle batch updates by focusing computation on the affected regions of the graph.

The results (to be filled in the table) are anticipated to show that parallel approaches offer significant performance improvements over the naive serial method, especially for large-scale graphs and substantial numbers of updates. The choice of MPI or OpenMP depends on the specific hardware and the scale of the network being analyzed. Profiling tools like `mpiP` proved valuable in understanding the behavior and potential bottlenecks of the distributed MPI implementation.

This work underscores the importance of parallel computing techniques for tackling complex graph problems in the context of ever-changing, large-scale network data.

Running the code:

To run serial: cd Serial && g++ Serial.cpp -o Serial && ./Serial To run MPI: cd MPI && mpicxx main.cpp -o main -lmetis && mpirun -np 3 ./main To run Parallel: cd OpenMP && g++ Parallel.cpp -o Parallel && ./Parallel