National University
Of Computer and Emerging Sciences

"Semester Project Report"

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

In partial fulfillment
of the requirement for the course of

"Parallel and Distributed Computing"
CS-3006

By

Raima Tariq | 22I-1145_C
Rimsha Azam | 22I-1129_C
Jawairia Waseem | 22I-1274_G

Strategy for Updating Single-Source Shortest Paths in Dynamic Networks

This project implements an efficient and parallel strategy for updating Single Source Shortest Paths (SSSP) in large-scale dynamic graphs where edge insertions and deletions occur over time. The approach avoids recomputation from scratch and leverages OpenMP to enhance performance on multi-core systems. The implementation was developed and tested on a dual-boot Ubuntu setup and performance was profiled using `gprof`.

Two-Phase Algorithmic Approach

Our implementation employs a two-step approach for efficiently updating Single-Source Shortest Paths (SSSP) after network changes:

Step 1: Changed Edge Processing

The process_changed_edges function identifies vertices directly affected by edge modifications:

- **For deletions**: When an edge in the SSSP tree is removed, the child vertex has its distance set to infinity, its parent relationship cleared, and is marked for further processing.
- **For insertions**: When a new edge creates a shorter path, the vertex with the higher distance is updated with the new shorter distance, its parent pointer is adjusted, and it's marked as affected.

Step 2: Affected Vertex Propagation

The update_affected_vertices function handles the cascading effects in two sequential phases:

- **Deletion Propagation**: Iteratively processes vertices affected by deletions, invalidating all downstream vertices in the SSSP tree that relied on these paths.
- **Distance Update**: Iteratively processes all affected vertices, examining their neighborhoods to find shorter paths until no further improvements are possible.

Parallel Implementation with OpenMP

Data Structure Optimization

- Graph represented as adjacency lists for efficient neighbor traversal
- Separate tracking arrays for deletion-affected vs. generally affected vertices

Parallelization Techniques

- Parallel initialization of distance and tracking arrays
- Critical sections to protect distance and parent array updates
- Reduction operations to detect convergence across threads
- Collection of affected vertices to avoid repeated scans of entire vertex set

Load Balancing

- Dynamic scheduling via work collection vectors
- Critical regions carefully placed to minimize contention
- Verification performed on sampled vertices to reduce overhead

Performance Optimization

Serial vs. OpenMP Comparison

- OpenMP version utilizes all available threads
- Time measurements at key stages: graph loading, initial SSSP, update, verification
- Partial verification to balance correctness checking with performance

To evaluate and compare the performance of the serial and parallel (OpenMP) implementations of our Single Source Shortest Path (SSSP) algorithm, we used two different profiling approaches due to tool limitations.

Serial Implementation

The serial implementation was profiled using the GNU profiler (`gprof`). The profiling results, however, indicated negligible recorded execution time across all functions. This outcome likely stems from the short runtime of individual functions relative to the sampling resolution of `gprof` (approximately 10 milliseconds per tick). Thus, despite a high frequency of function calls, no meaningful function-level CPU time data was captured.

OpenMP Implementation

For the OpenMP (parallel) version, we initially intended to use `gprof`, but this tool is inherently unsuitable for multithreaded programs as it merges thread data, providing misleading or incomplete insights. Therefore, we employed the Unix `time` command, which gave us the following results:

```
real    19.624s

user    0.031s

sys     0.031s
```

These results show that although the parallel program executed for approximately 19.6 seconds (real time), the actual CPU utilisation (user + system) was only around 0.06 seconds. Such low CPU utilisation suggests significant thread underutilisation and indicates inefficiencies such as high overhead from thread management, synchronisation bottlenecks, or an I/O-bound workload poorly suited for parallel execution.

Overall, the OpenMP parallelisation did not deliver the expected performance improvement over the serial implementation. Instead, it introduced overhead and complexity without sufficient computational workload to justify the parallelisation. Future improvements should focus on optimising parallel granularity, minimising synchronisation overhead, and ensuring computational workloads are sufficient to exploit parallelism effectively.

MPI (Serial) Implementation

To further analyse performance, we measured the execution time of the MPI-based serial version. The output from the `time` command was:

```
real    27.063s

user    4.687s

sys     9.747s
```

Unlike the OpenMP implementation, the MPI version exhibits significantly higher CPU utilisation, with `user + sys` time totalling around 14.43 seconds. This indicates that the MPI program makes effective use of CPU resources, even though it is running in a serial configuration. The higher `sys` time suggests considerable kernel-level operations, possibly due to inter-process communication or file I/O. While the real time is slightly longer than the OpenMP version, the actual work done on the CPU is much greater, indicating that the MPI version is carrying out meaningful computation.

Conclusion

Overall, the OpenMP parallelisation did not deliver the expected performance improvement over the serial implementation. Instead, it introduced overhead and complexity without sufficient computational workload to justify the parallelisation. The MPI implementation, despite being serial in structure, demonstrated better CPU utilisation and more balanced execution. Future improvements should focus on optimising parallel granularity, minimising synchronisation overhead in OpenMP, and potentially combining MPI and OpenMP in a hybrid configuration to fully exploit hardware capabilities.

Adaptive Execution

- Employs Dijkstra's algorithm for initial SSSP computation
- Switches to custom propagation algorithm for incremental updates
- Thread-safe priority queue handling for the initial SSSP phase

Memory Access Patterns

- Careful synchronization to balance parallelism with correctness
- Batched processing of affected vertices to improve locality
- Atomic updates to minimize lock contention