National University
Of Computer and Emerging Sciences

Project Report

An Assignment presented to

Mr. Farrukh Bashir

In partial fulfillment
of the requirement for the course of

CS-3006
Parallel and Distributed Computing

By

Zakariya Abbas (22I-0801)
Muhammad Sibtain(22I-0887)

Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks using shared-memory Platforms

Table of Contents

Introduction

The Single Source Shortest Path (SSSP) problem is fundamental in graph theory with applications in real life problems like transportation and communication networks. While many parallel algorithms exist for static graphs, real-world networks are often dynamic. Finding the SSSP of the entire graph again is too expensive and time consuming. So, we take a look into an algorithm to change only the sub-graph which is affected by the change. This project presents a shared-memory parallel implementation to efficiently update SSSP tress in such dynamic networks.

The work is based on the research paper “A Parallel Algorithm Template for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks” by Khanda et al., including shared-memory multicore CPUs and GPUs.

Key Contributions

The key contribution of the paper includes the design of a platform-independent parallel algorithm that updates only the affected portions of the graph. This approach is built around the idea of maintaining SSSP result as rooted tree structure and updating the shortest paths only for vertices impacted by the network change. Experimental results demonstrated that this method significantly outperforms state-of-the-art recomputation-based solutions like Galois when majority of edge changes are insertions.

Shared-Memory Implementation

The shared-memory approach utilizes OpenMP to parallelise the SSSP update algorithm. After observations, we can see that we can parallelize the detection of affected subgraphs due to the change. It processes batches of edge insertions and deletions to identify affected vertices and update their shortest paths accordingly. The key focus is load-balancing and avoiding synchronization overhead and handling large scale edge changes.

Algorithm Steps

In shared memory implementation, the algorithm is executed in two main steps. First, it identifies the subgraphs affected by edge insertions or deletions. Each changed edge is processed in parallel using OpenMP, and affected vertices are marked using Boolean flags. Edge deletions may disconnect vertices and subtrees from the main SSSP tree, while insertions may introduce shorter paths requiring distance updates.

The second step involves updating these affected subgraphs iteratively. The update algorithm is designed to asynchronously propagate updates in parallel. This approach allows the system to efficiently handle large batches of changes in a scalable manner using OpenMP. The key idea is to identify affected vertices, process them in parallel, and iteratively update their shortest path distances without using locks or critical sections.

The algorithm begins by handling edge deletions before insertions. This ordering is intentional and crucial for maintaining the correctness of the SSSP tree. When an edge is deleted, it may disconnect a vertex or an entire subtree from the original SSSP tree. These vertices must be treated carefully, as their previous paths may no longer be valid. By processing deletions first, the algorithm ensures that any invalid paths are removed before considering the possibility of new, shorter paths introduced by insertions. This prevents incorrect updates caused by temporarily keeping disconnected subtrees intact.

The algorithm maintains two arrays: AffectedDel for vertices affected by deletions and Affected for vertices affected by any change (deletion or insertion). It first traverses the tree starting from deletion-affected vertices. All descendants of a deleted edge are updated by setting their distances to infinity, effectively disconnecting them from the source. This disconnection process is done recursively using a queue and continues until all affected descendants are marked and updated. These steps are executed in parallel with OpenMP using dynamic scheduling to ensure efficient load balancing across threads.

After deletions have been processed and subtrees have been disconnected, the algorithm moves to the insertion and update phase. This step involves recalculating the distances of affected vertices using their neighbours. If a neighbour offers a shorter path to the source, the vertex's distance and parent are updated. This operation is done in parallel across all vertices marked as Affected. The process continues iteratively, as each vertex checks its neighbours, and if a shorter path is found, it updates its distance and marks itself as affected again. This loop continues until no further updates are possible.

A key feature of Algorithm 4 is asynchronous updating, which allows threads to propagate updates without waiting for others to synchronize at every level. This reduces synchronization overhead but possibly introduces more redundant updates, which is still lower than the overhead of synchronization.

Additionally, the algorithm uses dynamic scheduling for OpenMP threads, which helps in balancing work when different affected subtrees vary in size. This is important because some deletion-affected vertices may have large subtrees and assigning them to one thread statically could cause load imbalance and hurt performance.

Finally, to manage large-scale changes effectively, the implementation supports batch processing of edge updates. Instead of applying all changes at once, they are grouped into batches, and the update process is applied to each batch sequentially. This reduces memory contention and improves cache locality, especially when the number of threads is high.

Proposed Parallelization Strategy

In this extended setup, MPI can be employed for inter-node communication, enabling the algorithm to scale across multiple machines in a cluster. METIS would be used to partition the graph so that subgraphs are evenly distributed across nodes, minimizing communication overhead by keeping related computations local to each node. Within each node, OpenMP continues to be responsible for intra-node parallelism, handling updates to the local subgraph in parallel. This hybrid model can effectively combine the high-speed intra-node performance of OpenMP with the scalability of MPI for large, distributed systems.

Performance Summary

In terms of performance, the shared-memory implementation shows up to 5× speedup over traditional recomputation methods, especially when edge changes consist mostly of insertions. Scalability tests confirm that the implementation benefits from additional threads, although the performance gain is sensitive to the percentage of the graph affected by changes. If the changes impact more than 80% of the graph, recomputation may become more efficient. Additional experiments also show that increasing the level of asynchrony—i.e., the number of vertices visited before synchronization—can further improve execution time. Similarly, processing edge changes in batches offers performance improvements at high thread counts.

Conclusion

In conclusion, the shared-memory implementation of the proposed SSSP update algorithm presents an efficient, scalable approach for maintaining shortest path information in large, dynamic graphs. The algorithm's design, which avoids strict synchronization and favors localized updates, makes it particularly suitable for modern multicore architectures. The framework not only provides performance benefits over recomputation but also lays the foundation for future hybrid distributed solutions using MPI and METIS.

GPU Implementation

Proposed Parallelization Strategy

The GPU implementation exploits the **Single Instruction Multiple Threads (SIMT)** execution model provided by modern NVIDIA GPUs. Specifically, the implementation assigns one CUDA thread to each element in a large array of operands to perform graph-related operations simultaneously. When the operand array exceeds the available CUDA threads, the implementation utilizes **Grid-Stride Loops** to ensure all elements are processed efficiently.

The algorithm employs the **Compressed Sparse Row (CSR)** format for storing graph data, given its efficient memory utilization and suitability for GPU computations. The graph structure and its properties for the SSSP algorithm are maintained within an array data structure T, of size |V|, where each element T[i] holds the following properties:

- **Parent**: The parent vertex of vertex i.
- **Dist**: The distance from the source vertex to vertex i.
- **Flag**: A Boolean indicator marking if vertex i has been affected by recent graph changes and needs reprocessing.

To address the challenges posed by concurrent GPU thread operations, the implementation introduces a novel component named **Vertex-Marking Functional Block (VMFB)**. This block specifically minimizes the necessity for CUDA atomic operations, significantly reducing serialization of computations. VMFB manages graph-related parallel operations effectively and includes three main steps:

Vertex Marking

Each CUDA thread independently executes a user-defined function F_x on elements of the operand array and updates the shared flags array asynchronously. Due to the nature of updates (always setting flags to a constant value when certain conditions are met), the correctness of computations remains intact despite asynchronous updates.

Synchronization

After marking vertices asynchronously, a global synchronization barrier is used. This ensures all threads complete their execution of marking operations before proceeding, maintaining consistency and correctness of the data.

Filter

Following synchronization, this step utilizes **CUDA ballot synchronization** to efficiently consolidate marked vertices into a single array without duplication. This filtering significantly reduces redundant computations, especially in scenarios where affected subgraphs overlap, enhancing the overall performance and memory efficiency.

Algorithmic Implementation on GPU

The GPU implementation detailed as Algorithm 5 of the article involves four core user-defined functions:

- **ProcessIns**: Processes inserted edges, marking first-level affected vertices.
- **ProcessDel**: Handles deleted edges, identifying first-level deletion-affected vertices.
- **DisconnectC**: Disconnects vertices affected by deletions from their respective parent vertices.
- **ChkNbr**: Checks neighbors to reconnect disconnected vertices and updates their distances accordingly.

Each invocation of VMFB systematically handles arrays of vertices or edges using CUDA threads, where each thread independently applies the given function, flags affected vertices, and subsequently filters to maintain only unique vertices needing further processing.

Performance Summary

The GPU implementation is demonstrated to be efficient and scalable, offering significant performance improvements. Comparative performance experiments conducted on NVIDIA Tesla V100 GPUs indicate substantial speedups when compared with existing GPU-based static recomputation methods such as Gunrock. Specifically, this dynamic GPU approach achieves **up to 8.5× speedup for 50 million changed edges and up to 5.6× speedup for 100 million changed edges**, especially when changes are dominated by insertions rather than deletions.

Performance evaluations suggest that when more than 50% of the network undergoes deletion-based changes, a static recomputation approach may become more efficient. Conversely, when insertions predominate, the proposed GPU updating algorithm consistently outperforms recomputation strategies.

Conclusion

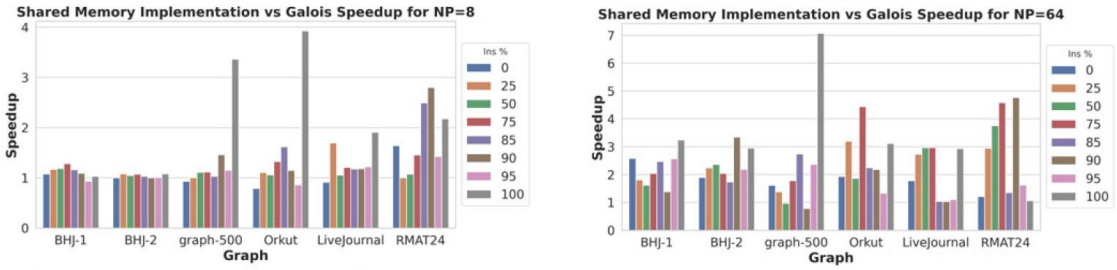
In summary, the GPU-based implementation of the dynamic SSSP updating algorithm harnesses the parallel capabilities of CUDA efficiently. By utilizing structured functional blocks (VMFB), minimizing atomic operations, and effectively managing synchronization and filtering, the

implementation achieves substantial performance gains over traditional recomputation methods. This makes the algorithm highly suitable for real-time applications involving large-scale dynamic networks.

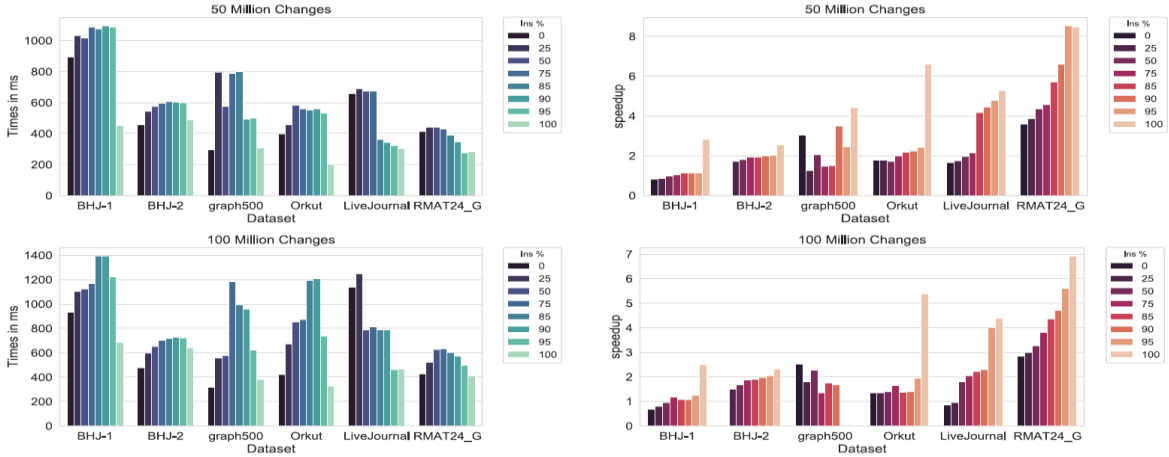
Conclusion

In conclusion, both the shared-memory (CPU) and GPU implementations of the proposed algorithm for updating Single Source Shortest Paths (SSSP) demonstrate significant performance advantages over traditional recomputation-based methods, such as Galois (CPU) and Gunrock (GPU). While sharing the common approach of efficiently updating only the affected portions of the graph, each implementation optimally leverages its respective hardware architecture.

The **shared-memory CPU implementation** excels in scenarios requiring flexible load balancing and asynchronous processing. It leverages OpenMP's dynamic scheduling and asynchronous updating to efficiently handle large batches of edge changes. This approach is highly beneficial when the affected subgraphs vary significantly in size. Moreover, the use of batch processing and asynchronous propagation of updates helps to further mitigate synchronization overhead. It is particularly effective for cases where the edge changes are dominated by insertions, achieving up to a 5× speedup over recomputation. However, when the network experiences extensive changes, particularly when over 80% of nodes are affected, the recomputation method may offer better performance.



The **GPU implementation**, utilizing NVIDIA's CUDA and its SIMT execution model, significantly reduces computation serialization through the innovative use of Vertex-Marking Functional Blocks (VMFB). These blocks efficiently manage concurrent operations, reducing reliance on atomic operations and improving overall parallel performance. The GPU approach is especially advantageous when the updates involve large numbers of insertions, achieving up to 8.5× speedup for 50 million changes and up to 5.6× speedup for 100 million changes compared to Gunrock's static recomputation. Nonetheless, in scenarios where a high percentage of changes involves deletions (typically over 75%), recomputing from scratch may still be more efficient.



Recommended usage scenarios:

- Use the **shared-memory CPU implementation** for:
  - Applications requiring flexible, asynchronous updates.
  - Situations with dynamic workloads and varying subgraph sizes.
  - Scenarios dominated by insertion operations.
- Use the **GPU implementation** for:
  - Real-time applications demanding maximum throughput and high parallelism.
  - Situations where updates are predominantly insertions, benefiting significantly from parallel GPU operations.
  - Large-scale problems where the overhead of atomic operations can be effectively minimized by GPU-specific optimizations.

Both implementations provide robust, efficient, and scalable solutions tailored to their specific computing environments. Thus, the choice between GPU and CPU approaches should consider the specific nature of network changes, computational resources, and real-time performance requirements. The flexibility and efficiency provided by this parallel algorithm template not only improve upon traditional approaches but also lay a solid foundation for future hybrid and distributed implementations involving MPI and METIS for even larger and more complex dynamic networks.