A Parallel Algorithm Template for Updating

Single-Source Shortest Paths in

Large-Scale Dynamic Networks

Name: Muhammad Masab Hammad, Muhammad

Hashim Awan, Fahad Faisal

Roll Number: i22-1004, i22-0976, i22-0817

Section: D

Table of Contents

Dynamic SSSP with Dijkstra (Serial Implementation)— Overview
This C++ program implements a dynamic Single-Source Shortest Path (SSSP) algorithm using Dijkstra's method on an undirected, weighted graph. It efficiently handles real-time graph updates (edge insertions and deletions), making it suitable for evolving networks where rapid recomputation is necessary.

? Dynamic Graph Handling
The graph is loaded from a dataset file, with each edge defined by source, destination, and weight.

- `addEdge` and `removeEdge` allow for dynamic updates to the graph structure.
- After each batch of modifications, the algorithm recomputes shortest paths without rebuilding the entire tree.

⚙SSSP Computation
The `DynamicSSSP` class maintains an adjacency list along with `distance` and `parent` vectors.

- Initial shortest paths from vertex 0 are computed using a priority queue-based Dijkstra's algorithm.
- The shortest path tree is then printed, showing vertex distances and their respective parents.

☐Performance Measurement
Execution time before and after dynamic updates is recorded using the `<chrono>` library, enabling performance comparisons across static and dynamic phases.

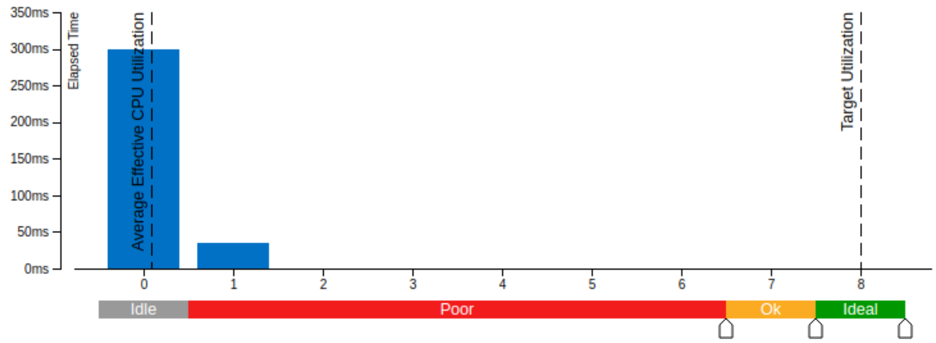
? Use Case Suitability
This implementation serves as a flexible and efficient template for applications such as:

- Real-time routing in communication networks
- Dynamic analysis of social networks
- Rapid prototyping of dynamic graph algorithms

Its modular structure and efficient recomputation strategy make it ideal for large-scale, real-time systems.

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

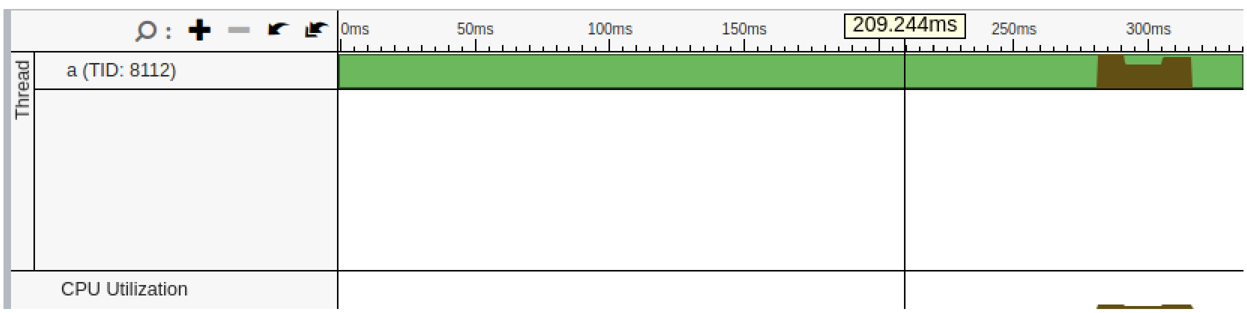


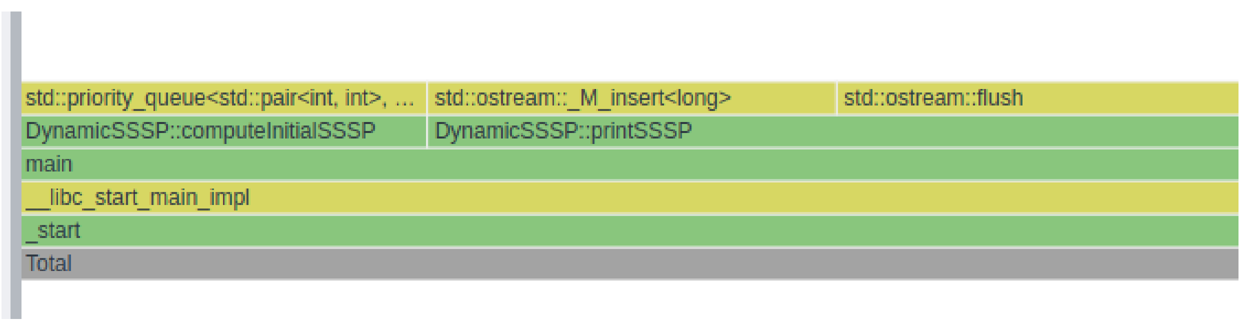
Elapsed Time: 0.335s

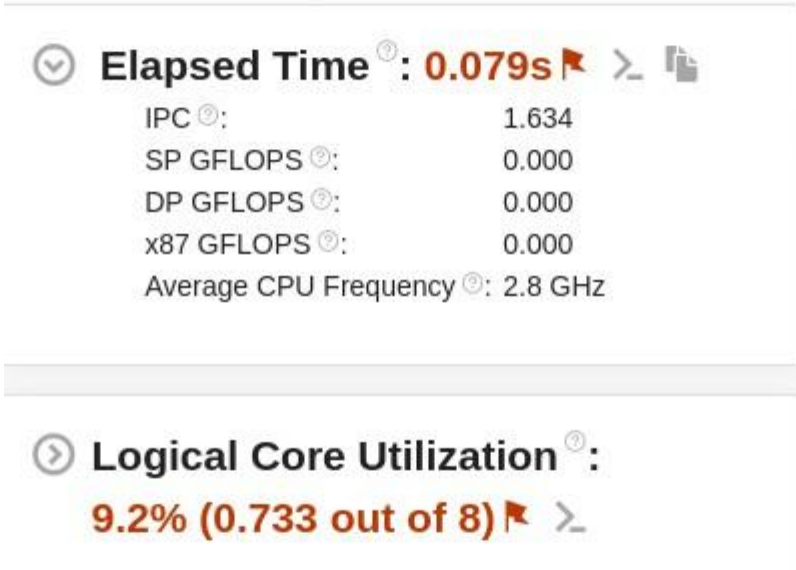
CPU Time: 0.030s

Total Thread Count: 1

Paused Time: 0s







PART 2 (openmp)

Dynamic SSSP with Parallelism — Overview

The provided C++ code implements a **Dynamic Single Source Shortest Path (SSSP)** algorithm tailored for efficiently computing shortest paths in **large-scale, evolving graphs**. This solution supports both **static graph computation** and **dynamic updates** (such as edge insertions and deletions) by leveraging **OpenMP** for parallel execution, resulting in faster performance for real-time applications.

□ Dynamic Graph Handling

The algorithm supports runtime modifications to the graph through:

- `addEdge` and `removeEdge`: Functions that dynamically update the graph's structure.
- `processIncomingUpdates` and `updateAffectedVertices`: Efficiently propagate the effects of structural changes by identifying and recalculating only **affected vertices**, rather than recomputing the entire graph.

This **incremental update mechanism** is built upon the **delta-stepping strategy**, which helps reduce redundant computations after each update.

⚙Parallelism with OpenMP

To enhance performance, **OpenMP** is employed for:

- Parallelizing loops (e.g., finding the minimum-distance vertex) using `#pragma omp parallel for`.
- Ensuring thread-safe modifications to shared data (like `distance[]` and `parent[]`) via `#pragma omp critical`.

This multi-threaded design drastically reduces computation time, especially on multi-core systems, while maintaining correctness.

✅Validation and Consistency

To guarantee the reliability of computed paths:

- `validateSSSP` is used post-update to **verify the consistency** of parent-child relationships and distance values across the graph.
- Timestamps track the recency of each edge and vertex update, ensuring updates are processed in the correct sequence and avoiding stale information.

⚡Efficient Update Propagation

The update engine uses:

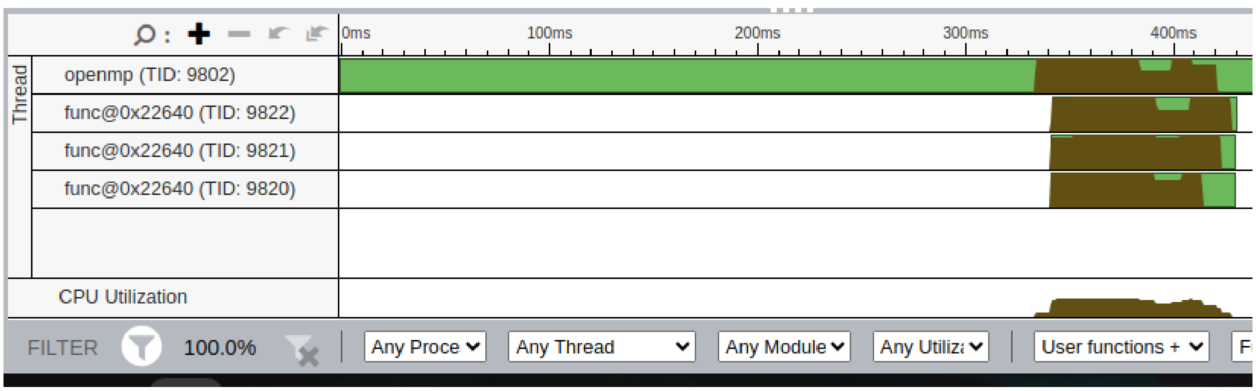
- A **priority queue (min-heap)** to prioritize changes affecting vertices with **shorter distances**, thus avoiding unnecessary path expansions.
- A tunable parameter `batchSize` to control the **number of updates** processed simultaneously, balancing **throughput** and **system overhead**.

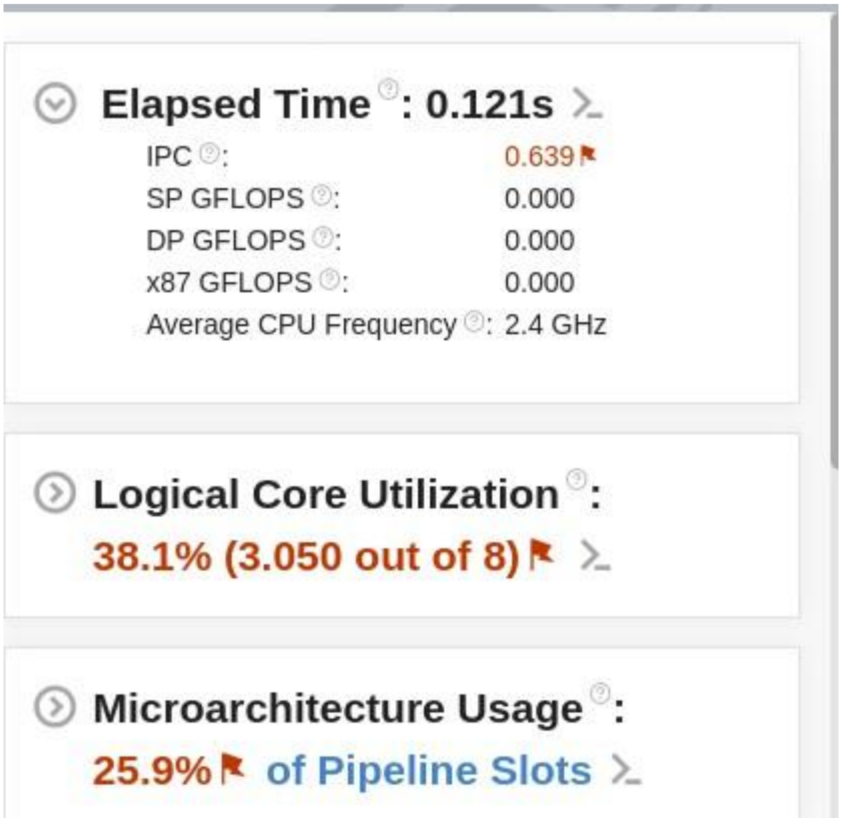
? Use Case Suitability

This approach is well-suited for scenarios where graphs change frequently, such as:

- **Real-time network routing**
- **Social network analysis**
- **Urban traffic simulation**
- **Dynamic dependency graphs in software systems**

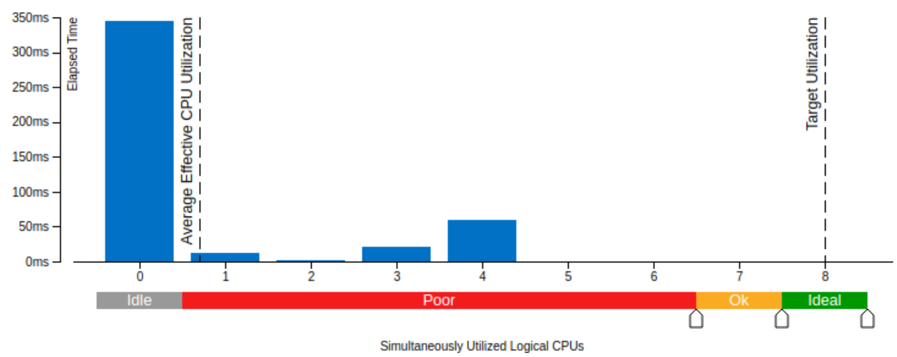
Its efficient, parallel, and incremental design offers a practical balance between **accuracy**, **speed**, and **scalability**.





Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



PART 3 (MPI + OpenMP)

Dynamic SSSP with Hybrid Parallelism — Overview
The provided C++ implementation extends dynamic Single Source Shortest Path (SSSP) computation to distributed environments by combining **MPI for inter-process communication** and **OpenMP for intra-process parallelism**. This hybrid-parallel approach ensures scalability across multiple nodes and efficient utilization of multi-core architectures for large and evolving graphs.

� Hybrid Parallelism with MPI and OpenMP
To handle both distributed memory and shared memory architectures:

- **MPI** partitions the graph across processes, with each process responsible for a subset of vertices.
- **OpenMP** accelerates intra-process computations like local minimum selection and edge relaxation using `#pragma omp parallel for`.
- **Thread safety** is ensured using `#pragma omp critical` during shared state updates (e.g., distance and parent arrays).

This dual-layered parallelism maximizes performance on high-performance computing (HPC) systems.

� Dynamic Graph Handling
This solution supports runtime updates via:

- `addEdge` and `removeEdge`: Dynamically modify the graph structure.
- `processIncomingUpdates` and `updateAffectedVertices`: Propagate the effects of edge insertions and deletions to relevant parts of the graph.

Instead of recomputing the full shortest path tree, only **affected vertices are incrementally updated**, significantly reducing recomputation costs in dynamic environments.

▯ Efficient Distributed Communication

MPI-based communication ensures consistent global state:

- **MPI_Allreduce** is used to determine the global minimum-distance vertex across all processes.
- **MPI_Bcast** propagates updated vertex data to ensure synchronized state.
- **MPI_Barrier** enforces alignment between processes during update phases.

These collective operations maintain consistency and convergence across distributed partitions while minimizing latency overhead.

▯ Debugging and Robustness Features

The implementation includes:

- `DEBUG_PRINT` macros to trace computation, monitor update propagation, and detect anomalies or hangs.
- **Timestamps** to sequence updates correctly and prevent processing outdated information.
- **Iteration limits** to guard against infinite loops in dynamic update scenarios.

⚡Scalability and Load Balancing

To enhance scalability:

- Updates are processed in **batches** (configurable via `batchSize`) to reduce communication frequency and allow amortized processing.
- Dynamic repartitioning or intelligent edge assignment may be considered to mitigate **load imbalance** in high-skew graphs.

This makes the approach effective on large, frequently changing graphs distributed over multiple compute nodes.

⍰ Use Case Suitability

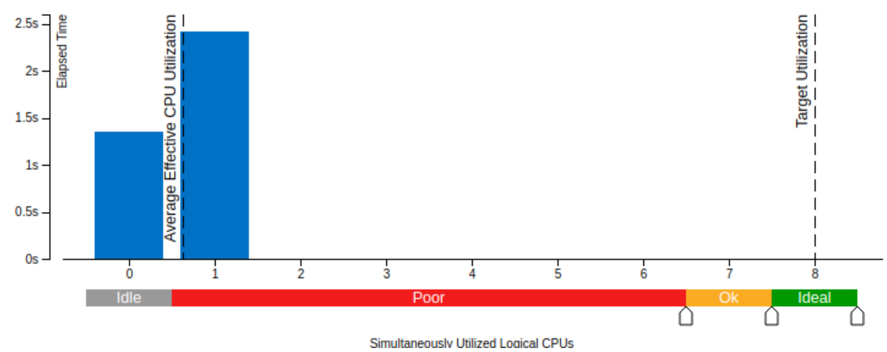
This implementation is designed for real-time, large-scale applications where graphs evolve rapidly:

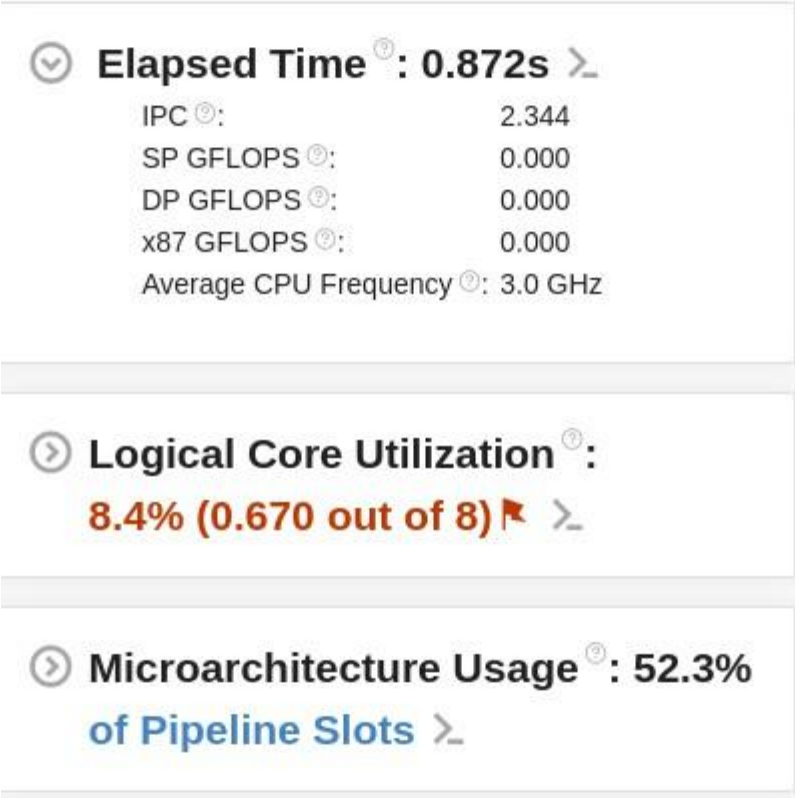
- **Traffic network routing** with live updates (e.g., road closures or congestion)
- **Social network evolution** and influence analysis
- **Communication network resilience and rerouting**
- **Dynamic dependency resolution** in distributed software systems

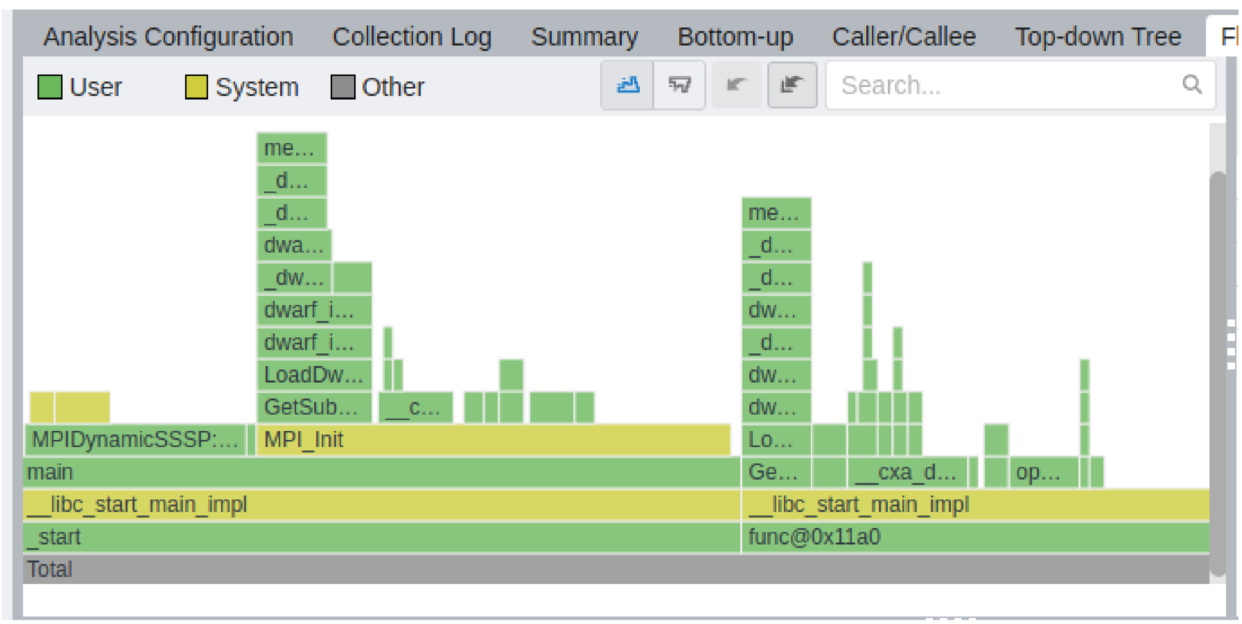
By integrating both distributed and shared-memory paradigms, this approach delivers high performance, accuracy, and responsiveness in complex, evolving environments.

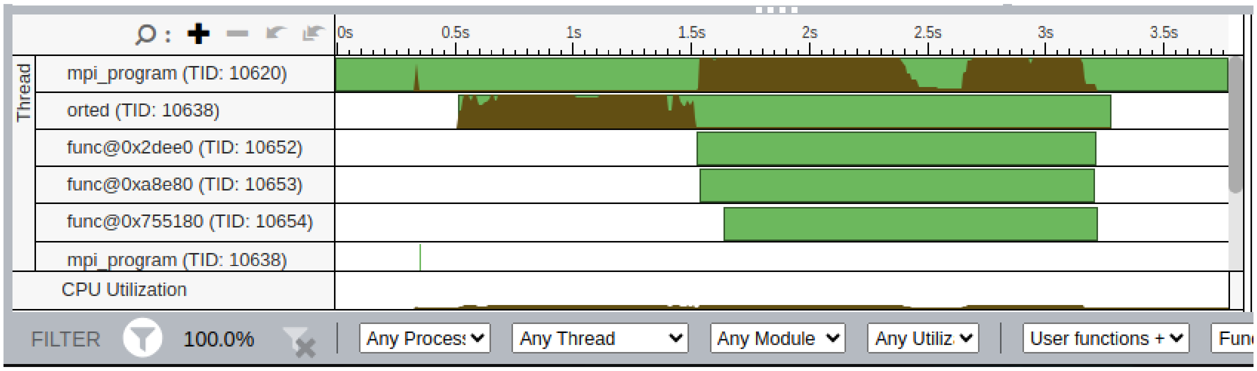
Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.




Elapsed Time: 0.872s
IPC: 2.344
SP GFLOPS: 0.000
DP GFLOPS: 0.000
x87 GFLOPS: 0.000
Average CPU Frequency: 3.0 GHz
Logical Core Utilization:
8.4% (0.670 out of 8)
Microarchitecture Usage: 52.3%
of Pipeline Slots


Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
User
System
Other
Search...
me…
_d…
dwa…
_dw…
dwarf_i…
LoadDw…
GetSub…
__c…
MPIDynamicSSSP:…
MPI_Init
main
__libc_start_main_impl
_start
Total
dw…
Lo…
Ge…
__cxa_d…
op…
func@0x11a0



ALL IN 1

PART 4 (MPI + OpenMP + METIS)

Hybrid Dynamic SSSP with Distributed Partitioning — Overview

This C++ implementation introduces a hybrid parallel algorithm for solving the **Dynamic Single Source Shortest Path (SSSP)** problem in large-scale, evolving graphs. It integrates **MPI** for distributed computing, **OpenMP** for shared-memory parallelism, and **METIS** for graph partitioning. The core class `HybridDynamicSSSP` handles dynamic updates and initial path computation efficiently, making it ideal for high-performance, real-time graph applications.

▯ METIS-Based Graph Partitioning

To ensure balanced workloads and minimize inter-process communication:

- **METIS** partitions the graph to reduce edge cuts and evenly distribute vertices across MPI processes.
- Each process is responsible for a subset of the vertices, maintaining a local view of the graph and improving scalability.

This preprocessing step ensures that communication overhead is minimized during parallel execution.

▯ Hybrid Parallelism with MPI and OpenMP

The algorithm combines distributed and shared-memory models for optimal performance:

- **MPI** handles data distribution and coordination across compute nodes.
- **OpenMP** accelerates local operations such as distance updates and neighbor relaxation with `#pragma omp parallel for`.
- **Thread safety** is enforced using `#pragma omp critical` during updates to shared structures like `distance[]` and `parent[]`.

This hybrid design fully utilizes modern multi-core, multi-node architectures.

Dynamic Graph Handling with Delta-Stepping
The system supports real-time updates to the graph:

- `addEdge` and `removeEdge`: Modify the graph topology dynamically.
- `updateAffectedVertices`: Triggers **incremental recalculation** of shortest paths using a delta-stepping approach.

Affected vertices are prioritized using a **queue**, avoiding full recomputation and enabling low-latency responses to structural changes.

Efficient Communication and Synchronization
MPI ensures synchronized state across distributed processes using:

- `MPI_Allreduce`: Determines global minima in distance values.
- `MPI_Bcast`: Broadcasts shared updates (e.g., changed vertex distances) to all processes.
- `MPI_Barrier`: Aligns computation stages across nodes.

Debugging support via `DEBUG_PRINT` macros helps trace execution flow and identify communication bottlenecks or deadlocks.

✅**Validation and Robustness**
To maintain correctness and prevent errors:

- `validateSSSP`: Verifies distance consistency and parent relationships after each update.
- **Iteration caps** avoid infinite loops in dynamic update scenarios.
- **Batch processing** of updates smooths load and reduces communication overhead.

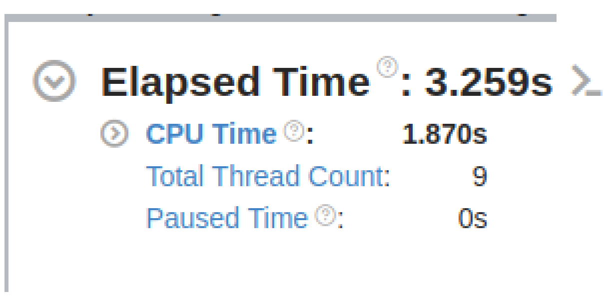
This contributes to robust and predictable system behavior in real-time contexts.

⍰ Use Case Suitability

This hybrid parallel SSSP solution is designed for dynamic, high-throughput environments such as:

- **Traffic management systems** with real-time incident updates
- **Social network analysis** where relationships evolve frequently
- **Distributed systems dependency tracking**
- **Telecommunication networks with live rerouting**

Its ability to combine **scalability, speed, and adaptability** makes it highly effective for modern graph-centric applications.



Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Id CPU utilization value.