PDC Project Phase 1

An Assignment presented to

Sir Farukh Bashir

In partial fulfillment
of the requirement for the course of

PDC

By

Arban Arfan(22I-0981), Abdullah Shakir(22I-1138), Messam Raza (22I-1194)

BS(CS)
SECTION-E

Title: A Comprehensive Analysis and Hybrid Parallelization Strategy for

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

---

1. Introduction

The dynamic nature of real-world networks—ranging from communication infrastructures to biological systems—demands algorithms that not only compute efficiently but also **adapt quickly** to changes in network topology. A core operation in graph analytics is the **Single Source Shortest Path (SSSP)** problem.

While static algorithms like Dijkstra's perform well for non-changing graphs, they fall short when edge insertions or deletions occur frequently. The paper under review proposes a platform-independent **parallel framework for updating SSSP trees incrementally**, eliminating the need to recompute from scratch after every change. This report presents a formal analysis of the paper, summarizes its key ideas, and **proposes an extended parallelization strategy** using MPI, OpenMP/OpenCL, and METIS to enable scalability across distributed-memory and heterogeneous systems.

---

2. In-Depth Study of the Paper

1. **Problem Motivation**
   Real-world networks (social, communication, cyber-physical) are both large (millions of vertices, billions of edges) and dynamic (edges inserted/deleted over time). Conventional SSSP algorithms assume static graphs, resulting in expensive re-computation upon each change. The paper addresses **incremental** updates to the SSSP tree efficiently, without recomputing from scratch .
2. **Algorithmic Framework**
   The proposed framework decouples *algorithm design* from *implementation platform* and proceeds in two major phases:
   - **Step 1: Identify Affected Subgraph**
     Process each inserted/deleted edge in parallel to mark vertices whose shortest-path distances may change. Deletions sever tree links (setting distances to ∞), while insertions tentatively relax affected vertices (Algorithms 2: ProcessCE) .
   - **Step 2: Update Affected Subgraph**
     Iteratively traverse only the marked vertices and their neighbors to converge to the new shortest distances (Algorithm 3: UpdateAffectedVertices). No global priority queue is used; instead, the method relies on repeated, lock-free

relaxations that converge to the minimum distances without fine-grained synchronization .

3. **Data Structures**
    - **SSSP Tree Representation**: Adjacency list storing, for each vertex *v*, its parent in the SSSP tree, current distance `Dist[v]`, and Boolean flags `Affected`/`Affected_Del`.
    - **Dynamic Updates**: Arrays of changed edges (`Del_k`, `Ins_k`) and auxiliary Boolean arrays for marking subtrees.
4. **Scalability Challenges and Solutions**
    - **Load Balancing**: Subtrees rooted at deleted vertices vary in size; dynamic scheduling (OpenMP) or functional blocks (GPU) mitigate imbalance.
    - **Synchronization**: Iterative, asynchronous relaxations replace costly locks; vertices only move toward shorter distances, guaranteeing convergence.
    - **Cycle Avoidance**: Deletion-affected subtrees are fully disconnected before reattachment, preventing transient cycles when concurrently processing multiple insertions.

---

3. Summary of Key Aspects

Aspect	Description
Dynamic Model	Edge-insertions and deletions only; vertex changes reduce to edge changes.
Core Idea	*Locality*: Only update portions of the SSSP tree affected by changes. *Iteration*: Repeatedly relax edges until convergence, obviating explicit locks.
Parallelization Paradigm	Two-phase, data-parallel processing of edges/subgraphs, independent of platform (shared-memory vs. GPU).
Performance Metrics	Speedup over re-computation (Gunrock on GPU, Galois on CPU) up to 5.6×–8.5× for moderate update sizes; benefits diminish if >75% of edges deleted (small affected set).
Platforms Evaluated	NVIDIA V100 GPU (functional block approach), OpenMP on multicore CPU (asynchronous scheduling); experiments on real-world and synthetic graphs (up to ~16 M vertices, 250 M edges).

---

4. Key Contributions

The authors clearly articulate three primary contributions:

1. **Platform-Independent Framework**
   A novel two-step **update** algorithm for dynamic SSSP that cleanly separates the *logical* update procedure from *physical* implementation details, enabling deployment on both CPUs and GPUs .
2. **Shared-Memory Extension**
   An enhanced OpenMP implementation that processes up to 100 M changes in batches—tenfold the prior work—while preserving scalability through dynamic scheduling and tunable asynchrony .
3. **GPU Functional-Block Design**
   The **Vertex-Marking Functional Block (VMFB)** abstraction breaks complex graph operations into simple kernels for edge deletion, insertion, disconnection, and neighbor-relaxation, minimizing atomics and maximizing warp efficiency .
4. **Experimental Validation**
   Achieves up to **8.5x speedup over Gunrock** and **5x over Galois** under certain conditions.

---

5. Proposed Parallelization Strategy

To deploy the dynamic SSSP update algorithm on a **distributed-memory cluster** with multi-core CPUs and/or GPUs per node, we recommend the following three-layered approach:

5.1 Graph Partitioning with METIS / ParMETIS

- **Objective**: Minimize inter-node communication by reducing the edge-cut, while balancing vertex counts (and anticipated update loads) across MPI ranks.
- **Offline Partitioning**:
    1. Apply **METIS** (for static partition) or **ParMETIS** (for scalable, parallel partitioning) to the initial graph snapshot, producing k partitions for k MPI processes.
    2. Assign each process a block of contiguous vertex IDs plus ghost-vertex buffers for neighbors in other partitions.
- **Dynamic Repartitioning** (optional):
    1. If update patterns become highly skewed (e.g., hotspot insertions/deletions), invoke incremental repartitioning (METIS's adaptive modes) at controlled intervals to restore load balance.

5.2 Inter-Node Parallelism with MPI

- **Edge-Update Broadcast**:
  - Each MPI rank collects the subset of changed edges whose endpoints lie in its local partition. Use `MPI_Alltoallv` to route edge-updates so that both endpoints' owners receive relevant changes.
- **Global Iterations**:
  - For each round of **Step 1**, processes independently mark local affected vertices.
  - Exchange boundary-vertex "affected" flags via nonblocking `MPI_Ineighbor_allgather` or one-sided RMA (Remote Memory Access) to propagate cross-partition dependencies.
- **Convergence Detection**:
  - After each **Step 2** iteration (local relaxations), perform an `MPI_Allreduce` (logical OR) on a local `Changed` flag to determine whether further global iterations are required.

5.3 Intra-Node Parallelism with OpenMP

- **Shared-Memory Phase**:
  - Within each MPI process, spawn an OpenMP team to execute local portions of both **Step 1** (processing local edge-updates) and **Step 2** (iterative relaxations) in parallel.
- **Work-Stealing & Dynamic Scheduling**:
  - Use `#pragma omp parallel for schedule(dynamic,chunk)` for loops over edges or vertices, letting the runtime balance variable-sized subtrees.
- **Tunable Asynchrony**:
  - Adapt the "level of asynchrony" parameter (i.e., number of relaxation hops per synchronization) to trade redundant computation versus barrier overhead, tailored to intra-node core counts and memory bandwidth.

5.4 GPU Acceleration with OpenCL

- **Kernel Mapping**:
  - On nodes equipped with GPUs, offload **Step 1** and **Step 2** routines as OpenCL kernels mirroring the VMFB design (ProcessDel, ProcessIns, DisconnectC, ChkNbr).
- **Hybrid Execution**:
  - Partition per-rank local subgraph further into "GPU-resident" and "CPU-resident" subsets if GPU memory is constrained.

- Use OpenCL events and nonblocking MPI to overlap inter-node exchanges with GPU compute.

5.5 Integration and Hybrid Workflow

1. **Initialization**:
   - **METIS** → static partition → MPI ranks set up local subgraphs + ghost regions.
2. **Per-Batch Update** (for a batch of edge changes):
   - **MPI**: Distribute edge-updates to owning ranks.
   - **Step 1** (local):
     - OpenMP/OpenCL kernels: mark affected vertices, perform tentative parent updates.
   - **MPI**: Exchange boundary flags to capture cross-partition effects.
   - **Step 2** (local iterated):
     - OpenMP/OpenCL: relax local edges until no local change;
     - MPI: global OR to test convergence across all ranks.
3. **Termination**:
   - Once converged, each rank holds its portion of the updated SSSP tree; optionally merge or query results.

5.6 Benefits and Advantages

- **Reduced Communication**
  - METIS-minimized cuts shrink ghost-exchange volume, and only affected flags traverse partitions.
- **Load Balance**
  - Dynamic scheduling and asynchrony tuning ensure even work distribution despite irregular update patterns.
- **Scalability Across Scales**
  - Localized updates and iterative relaxations scale from single-node to multi-thousand-node clusters.
- **Heterogeneous Utilization**
  - Seamless offload to GPUs (via OpenCL) and CPUs (via OpenMP) maximizes resource utilization.
- **Lower Recompute Overhead**
  - Incremental updates avoid full SSSP re-computation, yielding significant speedups when updates are sparse (< 75 % deletions).

---

6. Implementation Considerations

- **Communication Minimization**
  Leverage METIS's minimized edge-cut to reduce ghost-region exchanges and overall MPI message volume.
- **Fault Tolerance & Checkpointing**
  Periodically persist per-rank SSSP tree state and batch offsets to durable storage, enabling rollback on failure.
- **Tuning Parameters**
  - **Batch Size**: Larger batches amortize MPI startup costs but increase local workload and memory pressure.
  - **Asynchrony Level**: Trade redundant computation versus barrier overhead by tuning the number of local relaxation hops per synchronization.
- **Library Support & Toolchain**
  - **ParMETIS/METIS** for graph partitioning.
  - **MPI-3** (neighborhood collectives, RMA) for inter-node coordination.
  - **OpenMP 5.0** for intra-node CPU parallelism .
  - **OpenCL 2.0+** or **CUDA 11.x** (with matching NVIDIA driver 450+) for GPU offload .
- **Dependencies Management**
  All external libraries, compilers, and runtimes must be specified explicitly and managed via the site's module or package-manager system (e.g., Lmod/Environment Modules, Spack) or via a reproducible container image (e.g., Singularity, Docker). In particular, the following *exact* components and versions (or newer, ABI-compatible releases) are required:
  - **C++ compiler** (GCC 9.x or later, Clang 10.x or later) with full C++17 support and OpenMP 4.5+ offload
  - **OpenMP runtime** for shared-memory parallelism
  - **MPI library** (e.g., OpenMPI 4.x or MPICH 3.x) with CUDA-aware support for inter-node communication
  - **CUDA Toolkit** 11.x and matching NVIDIA driver (450+ series) for GPU kernels
  - **METIS** 5.1.0 or later for graph partitioning on both CPU and GPU workloads
  - **OpenCL** (if used instead of OpenMP on accelerators), version 2.2 or higher

---

7. Conclusion

By combining **METIS**-driven partitioning, **MPI** for distributed coordination, and **OpenMP**/**OpenCL** for on-node parallelism, the dynamic SSSP update framework of Khanda *et al.* can be scaled to large heterogeneous clusters. This strategy maintains the algorithm's two-phase locality—identifying and updating only affected subgraphs—while minimizing inter-process communication and synchronizations. Tunable asynchrony and batch processing further adapt the implementation to varying graph dynamics and hardware characteristics, ensuring both high performance and resource efficiency.

References

Khanda, A., Srinivasan, S., Bhowmick, S., Norris, B., & Das, S. K. (2022). A parallel algorithm template for updating single-source shortest paths in large-scale dynamic networks. *IEEE Transactions on Parallel and Distributed Systems*, 33(4), 929–942.