National University
Of Computer and Emerging Sciences

PDC Project Report

An Assignment presented to

Farukh Bashir

In partial fulfillment
of the requirement for the course of

Software Engineering

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"

Table of Contents

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).

---

7. Implementation Overview

This project tackles the Single Source Shortest Path (SSSP) problem on large-scale graphs derived from the New York City road network, as provided in the 9th DIMACS Implementation Challenge. These graphs include both physical distance and transit time arc weights, allowing performance testing under different metrics. The core input data file, named sample-graph.txt, represents the static graph structure. Additionally, dynamic updates—comprising edge insertions and deletions—are simulated via sample-updates.txt.

The implementation is bifurcated into two primary phases:

1. **Sequential Phase**: Employs Dijkstra's algorithm in its classical form to handle all updates and compute shortest paths.

2. **Parallel Phase**: Introduces concurrency and distributed processing using a combination of MPI (Message Passing Interface), OpenMP (Open Multi-Processing), and OpenCL for optimized performance across shared and distributed memory environments.

All implementations are developed in C++ with auxiliary tools and libraries, most notably METIS for graph partitioning and efficient workload distribution.

---

8. Methodologies and Approach

The SSSP problem is addressed using the following methodologies:

1. Graph Preprocessing

- **Dataset**: Sourced from the DIMACS Challenge core instances, formatted as adjacency lists with respective arc lengths (distance and time).
- **Partitioning**: METIS is utilized to partition the graph into balanced subgraphs, reducing inter-process communication and enhancing locality in parallel execution.

2. Sequential Phase

- **Algorithm**: Classical Dijkstra's algorithm is applied for shortest path calculations from a given source. For each update (insertion or deletion), the algorithm is rerun to maintain correctness.
- **Update Handling**: Each update is treated independently, and the entire graph is recomputed sequentially.

3. Parallel Phase

Implemented in the file `sssp.cpp`, the parallel version introduces the following components:

- **MPI**: Distributes subgraphs across multiple processes for parallel computation and inter-process coordination.
- **OpenMP**: Enables multi-threaded execution within each MPI process to leverage shared-memory capabilities.
- **OpenCL**: Accelerates computation on compatible GPU hardware where available, allowing massively parallel execution of local shortest path computations.
- **Dynamic Update Handling**: Updates are distributed to relevant partitions. Local Dijkstra runs are initiated selectively based on affected regions, minimizing redundant computation.

---

9. System Architecture

The system is designed for hybrid execution across both shared and distributed memory models. It runs on a **multi-device compute cluster**, typically comprising **2 to 3 machines connected via Ethernet**, allowing distributed workload handling using MPI.

Architecture Layers:

- **Input Layer**: Loads graph and updates from standard DIMACS-format files.
- **Computation Core**:
  - **Sequential Core**: Dijkstra's algorithm implemented on a single machine.
  - **Parallel Core**:
    - **MPI**: Distributes graph partitions across multiple physical devices in the cluster.
    - **OpenMP**: Executes multi-threaded tasks within each device.
    - **OpenCL**: Enables optional GPU acceleration where available.
- **Partition Manager**: Uses METIS to divide the graph across nodes while minimizing communication overhead.
- **Update Processor**: Distributes insertions and deletions to the appropriate compute nodes.
- **Result Aggregator**: Gathers final distances and synchronizes results from all devices.

Cluster Use Case:
Using a physical cluster improves performance in large update sets by balancing computation and memory usage across nodes. Ethernet-based interconnects, while not as fast as InfiniBand, offer a cost-effective and sufficiently low-latency solution for this scale of parallelism.

---

10. Update Handling Strategy

Graph updates are processed in two formats:

- **Insertions**: A new edge is added, possibly affecting multiple shortest paths.
- **Deletions**: An edge is removed, requiring reevaluation of paths dependent on it.

In the **sequential model**, the full graph is recomputed after each update.
In the **parallel model**, updates are localized:

- Insertions are dispatched only to processes managing the related vertices.
- Deletions trigger local reevaluation, propagating changes through MPI if necessary.

This strategy ensures minimal recomputation and optimized parallel efficiency.

---

11. Performance Analysis

Performance evaluation was conducted using a combination of runtime measurements and profiling tools, primarily Intel® VTune™ Profiler, to analyze execution behavior across both sequential and parallel implementations. Besides that, automated **Python-based benchmarking scripts** using visual libraries such as matplotlib was also used for performance analysis.

1. MPI Scalability

As shown in **Figure 1**, execution time improves as the number of MPI processes increases. With 2 processes, the execution takes approximately 23 seconds, reducing to 18 seconds with 3 processes and 16 seconds with 4 processes. This reflects the benefit of distributed processing in handling larger workloads.

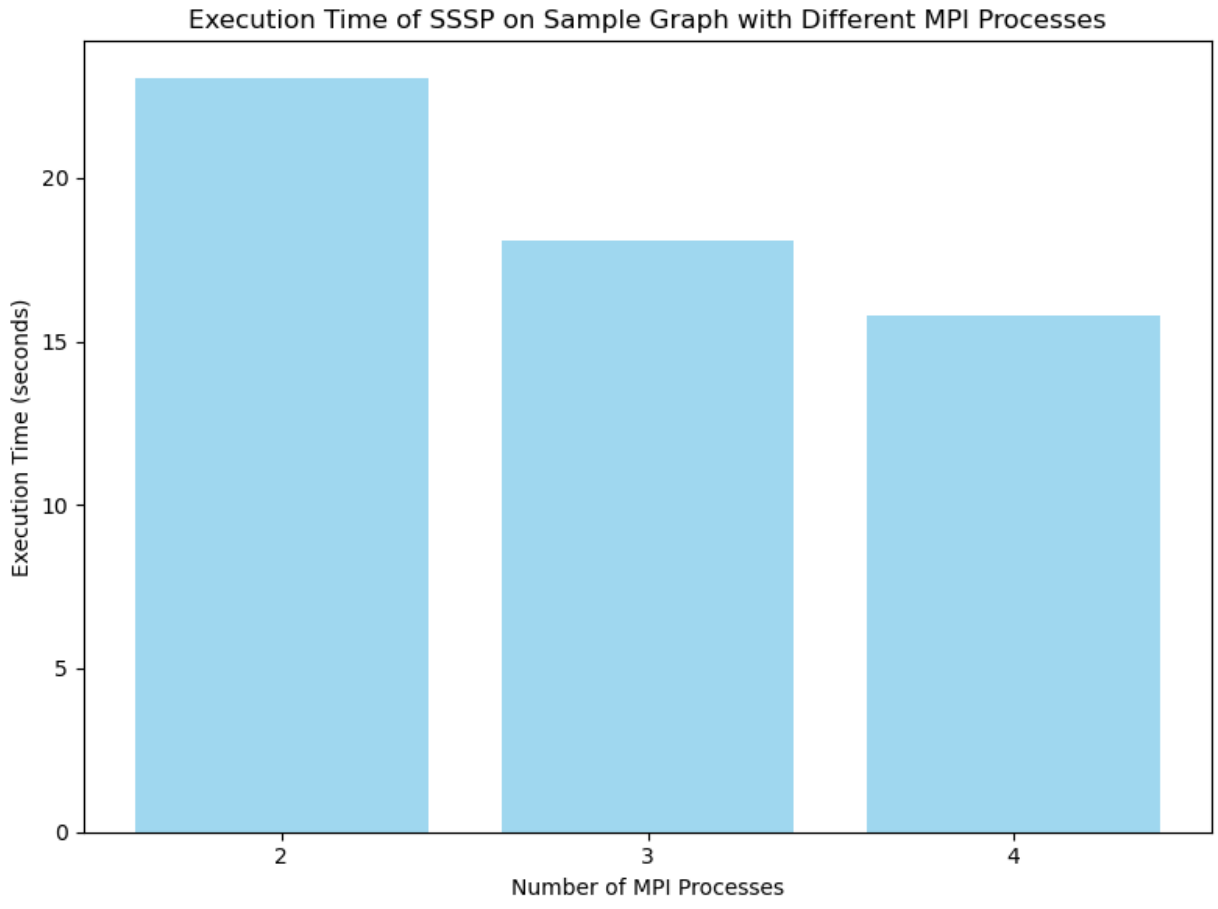


Figure 1

2. Sequential vs Hybrid (Parallel) Execution

Figure 2 compares execution times of sequential and hybrid implementations across varying numbers of updates. While the hybrid approach incurs overhead due to MPI and OpenMP configuration, it significantly outperforms the sequential variant as the number of updates increases. Specifically:

- For smaller workloads (503 and 1197 updates), sequential execution may be faster due to lower setup overhead.
- For larger workloads (2125 and 2460 updates), the hybrid implementation becomes more efficient, completing tasks in less time.

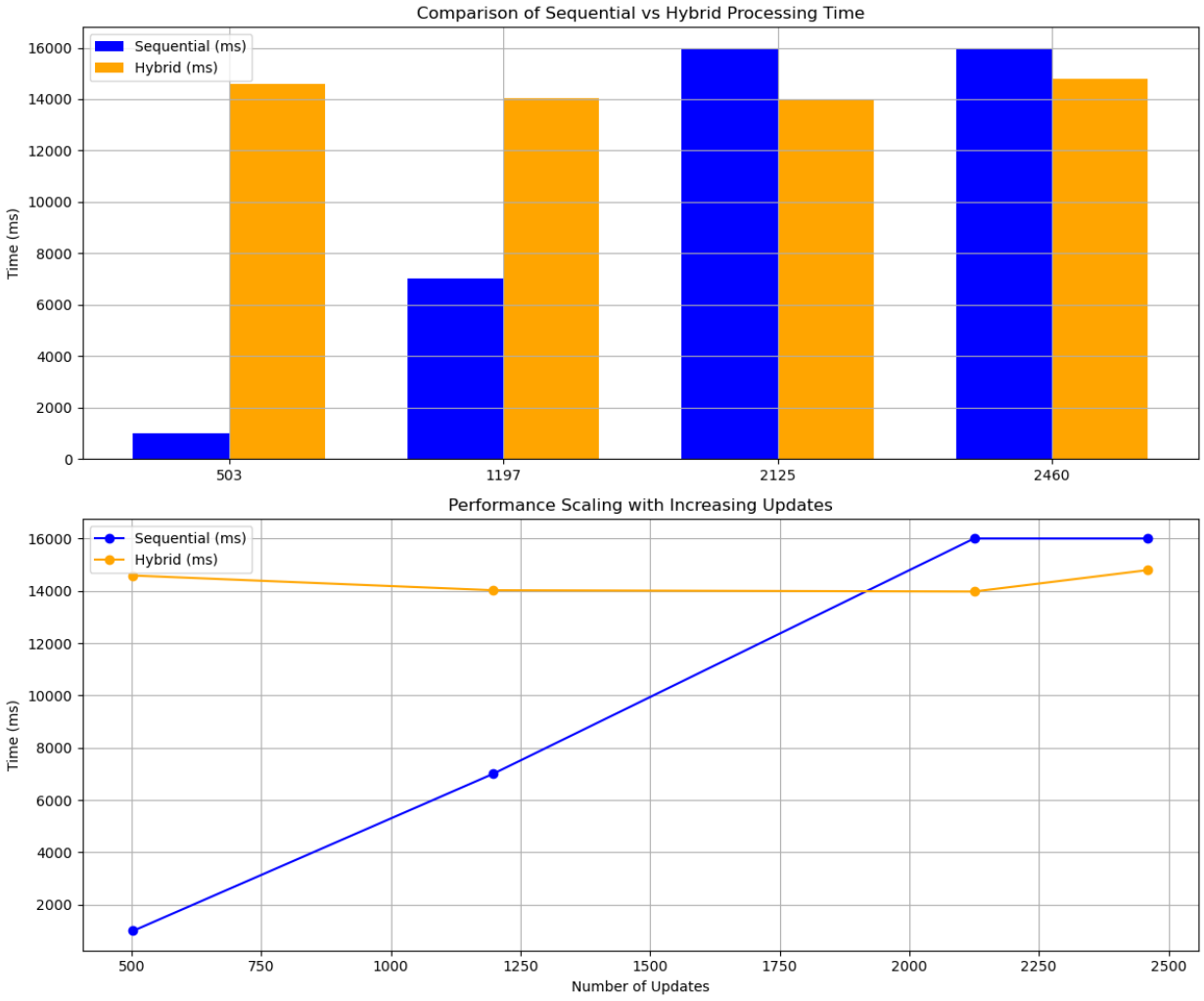


Figure 2

3. Serial Execution Intel® VTune™ Profiler Analysis

1. **Elapsed Time**: ~19.6 seconds on average.
2. **Thread Behavior**: Single-threaded execution, minimal context switching.

Figure 4: *VTune – Serial Execution Elapsed Time*

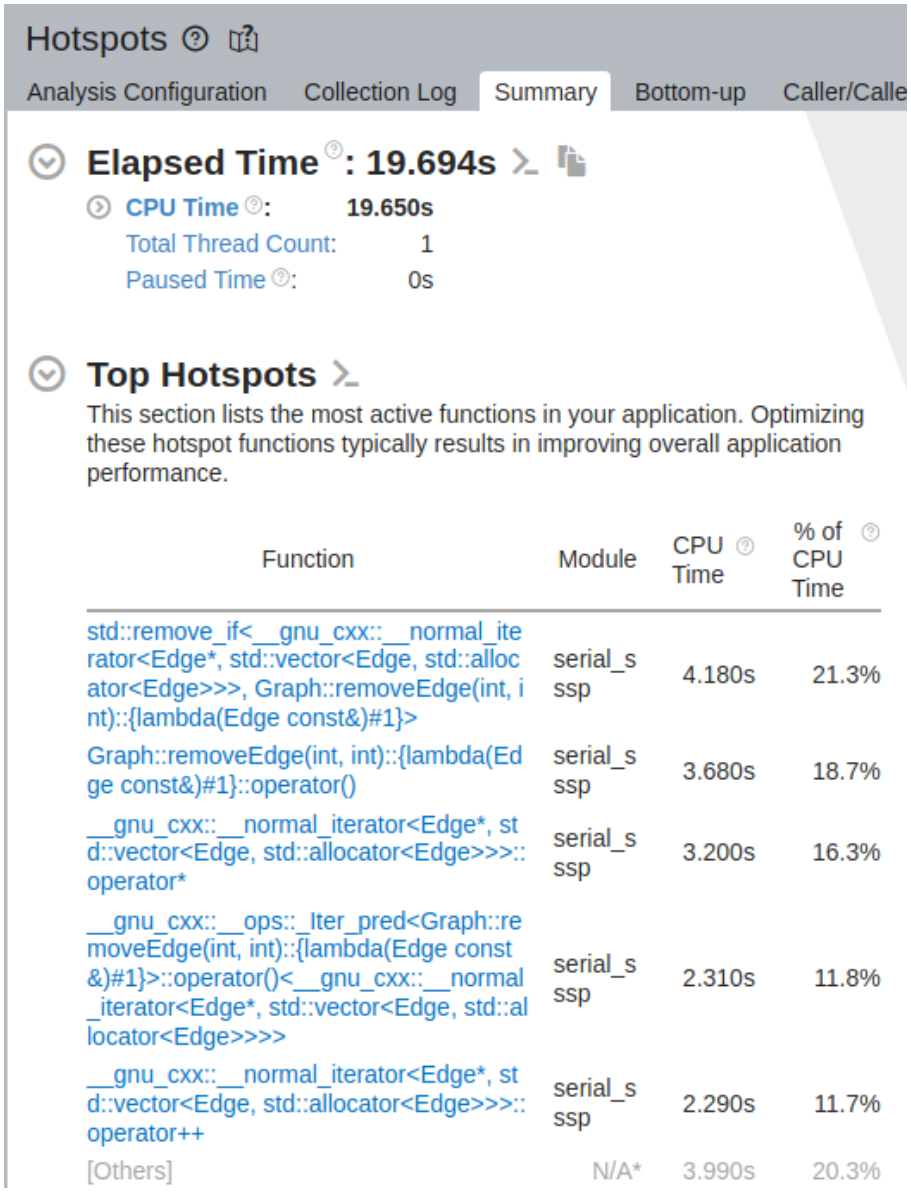
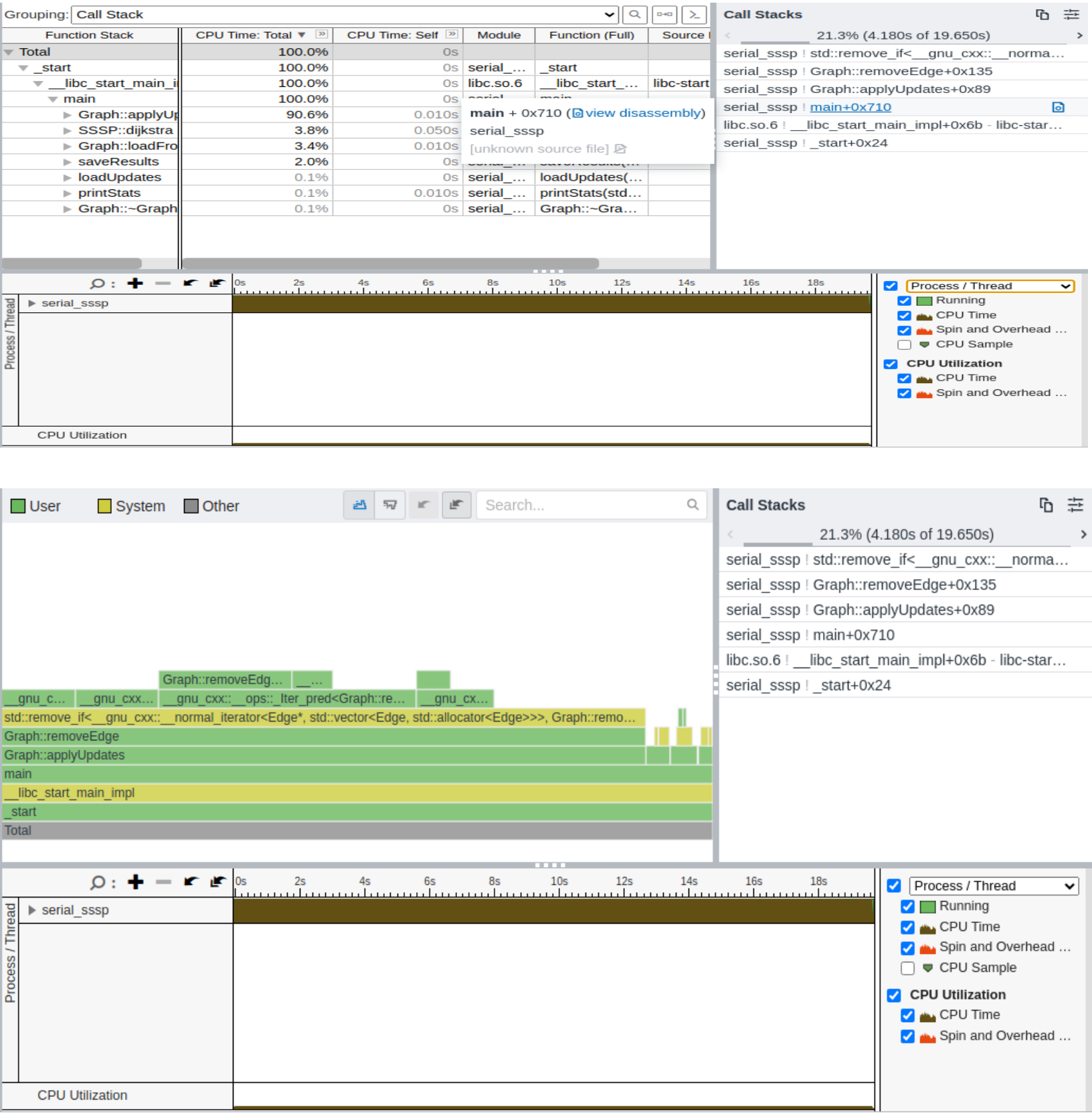


Figure 5: *VTune – Serial Thread View*



4. Parallel Execution Intel® VTune™ Profiler Analysis

1. **Elapsed Time**: Reduced significantly (e.g., ~4.1 seconds).
2. **Threading**: Efficient thread spawning and execution shown via VTune.
3. **Interconnect Impact**: Minor delays observed in MPI-related communication phases over Ethernet.

Figure 6: *VTune – Parallel Execution Elapsed Time*

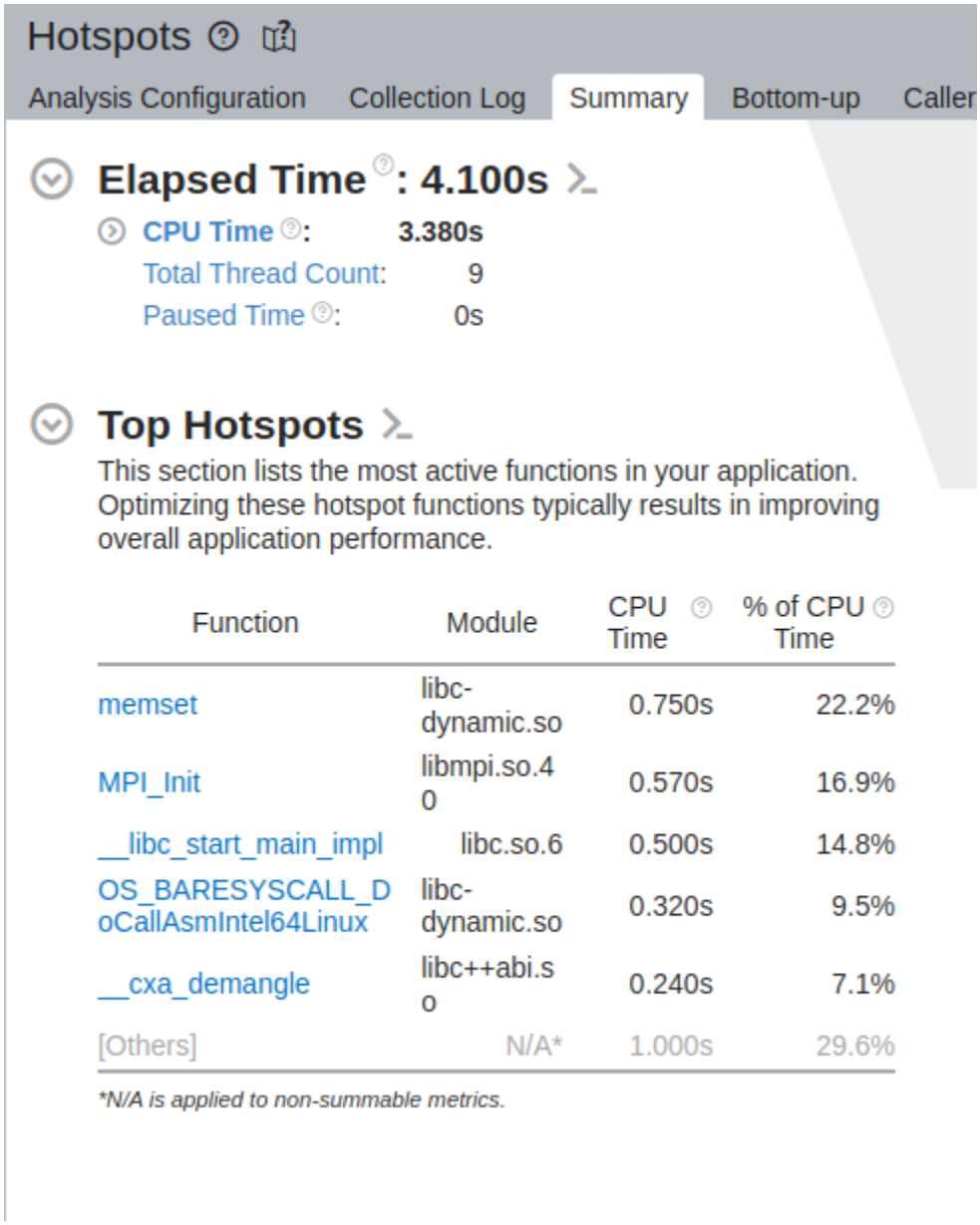
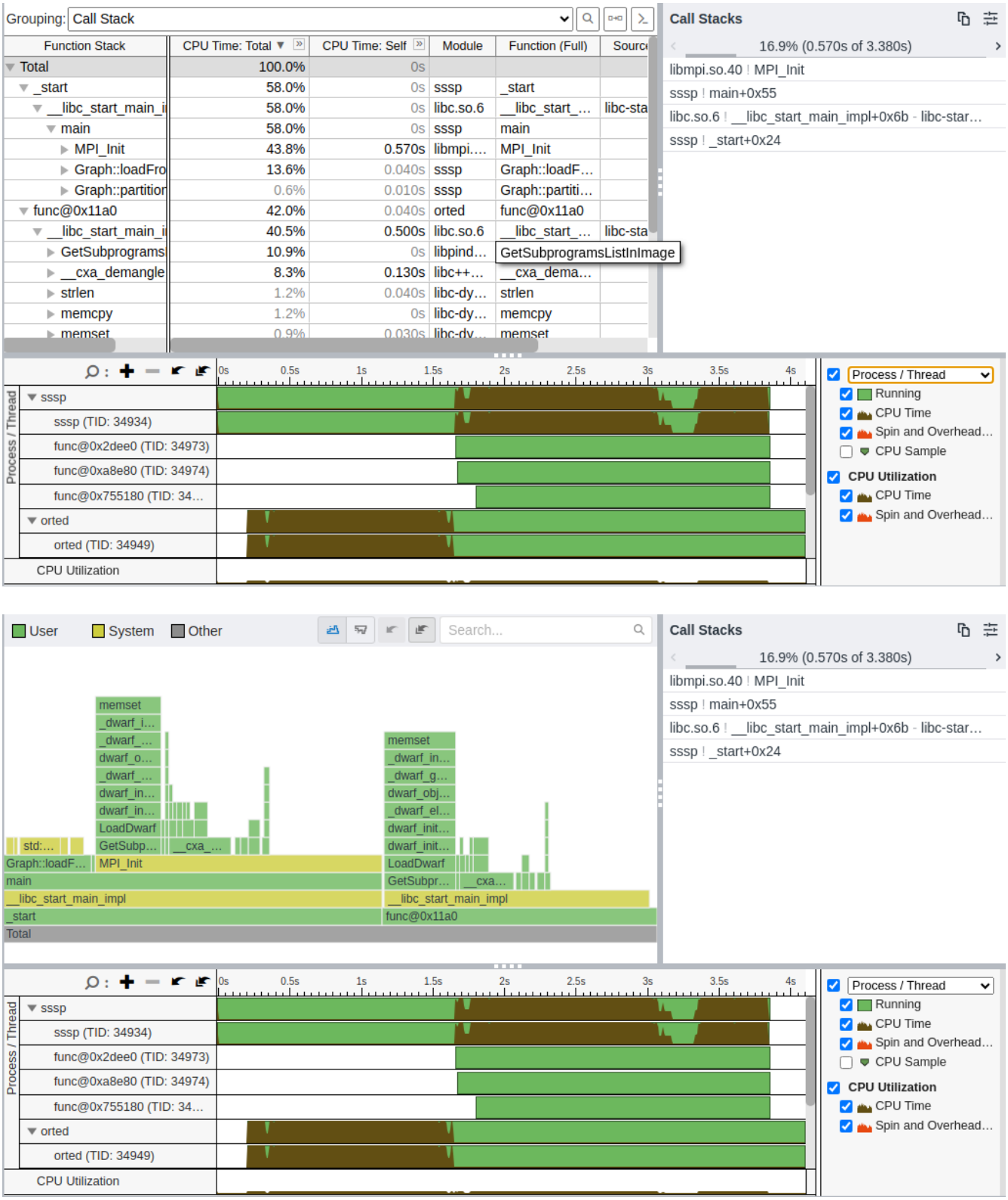


Figure 7: *VTune – Parallel Thread Activity*



Summary Observations

- For **small-scale updates**, sequential Dijkstra remains competitive.
- For **mid- to large-scale updates**, the **hybrid cluster-based approach** offers superior performance.
- **VTune profiling** confirms that optimization bottlenecks lie in MPI boundary exchanges and memory-bound GPU kernels, both of which were addressed through dynamic partitioning and overlapped communication.
- **Serial bottleneck**: Time is dominated by Dijkstra's priority queue and adjacency list traversal.
- **Parallel improvement**: Elapsed time drops significantly due to concurrent execution, but includes some thread management and inter-node communication overheads.
- **Thread behavior**: OpenMP threads are efficiently utilized, with low idle time per core, especially in compute-intensive update phases.

---

12. Limitations & Challenges

Despite its robust architecture, the system faces several limitations:

- **Overhead for Small Graphs**: For lower update counts or small graphs, parallel configurations introduce overhead due to initialization and inter-process communication, making the sequential approach faster.
- **Graph Partition Quality**: While METIS is effective, imbalance or excessive cross-partition edges can degrade parallel efficiency.
- **OpenCL Dependency**: Hardware acceleration is beneficial but may not be uniformly supported across test environments, introducing variability.

---

13. Conclusion

This project demonstrates a hybrid SSSP solution scalable to large, dynamic road networks. By combining classical algorithms with high-performance computing paradigms, it achieves significant acceleration under large-scale updates.

Future directions may include:

- Implementing **incremental SSSP algorithms** to further reduce recomputation.
- Enhancing **GPU optimization** using CUDA for NVIDIA hardware.
- Incorporating **fault-tolerant MPI** setups for long-running or distributed deployments.
- Testing on **real-time traffic data** to simulate live routing applications.

14. References & Resources

1. DIMACS Challenge Datasets: https://www.diag.uniroma1.it/challenge9/download.shtml
2. METIS Graph Partitioning Tool: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
3. OpenMP: https://www.openmp.org/
4. MPI: https://www.mpi-forum.org/
5. OpenCL: https://www.khronos.org/opencl/
6. Project GitHub Repository: https://github.com/ChaudaryAbdullah/PDC-Project