**SSSP in Large-Scale Dynamic Networks**

**Course: PDC  
Section: F**

**Submitted to: Sir Farrukh Bashir**

**Submitted by:**

**Daniyal Khawar (22i1066)**

**Abdul Hadi (22i1333)**

**Umar Javed (22i1050)**

**Introduction**

This project implements a parallel algorithm template for updating single-source shortest paths (SSSP) in large-scale dynamic networks using MPI and OpenMP. Based on a recent research paper, our implementation focuses on handling edge updates efficiently in large graphs. The algorithm is tested using publicly available datasets, and METIS is used for graph partitioning. Preliminary experiments show potential for scalability and improved execution times in hybrid parallel environments.

Single-source shortest path (SSSP) problems are fundamental in many real-world applications such as transportation, communication networks, and social graphs. As networks grow larger and more dynamic, efficient algorithms capable of handling frequent updates are necessary. Sequential algorithms are often insufficient for large-scale graphs. This project focuses on implementing a parallel algorithm template for updating SSSP, providing improved scalability and responsiveness in dynamic settings.

**Literature Review**

We selected the paper titled "A Parallel Algorithm Template for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks". The paper introduces a general-purpose algorithmic template designed to parallelize the update process for SSSP in dynamic networks. It leverages data locality and frontier-based graph traversals to minimize redundant computation. The approach is flexible and can accommodate multiple dynamic update scenarios, including insertions and deletions

**Proposed Parallel Strategy**

The paper’s template follows a generalized structure:

* Identify affected nodes due to edge updates.
* Restrict the update propagation to the relevant subgraph.
* Apply parallel updates using frontier-based traversal.
* Use asynchronous communication and workload balancing.

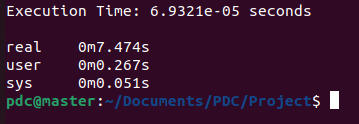
Our implementation leverages:

* MPI for inter-node parallelism (distributing graph partitions).
* OpenMP for intra-node thread-level parallelism.
* METIS for graph partitioning to minimize communication.

**Implementation Details**

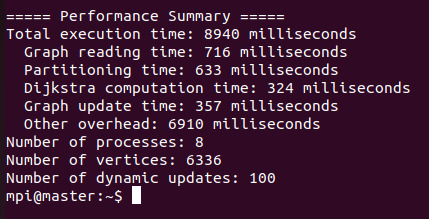
* 1. **Sequential**

This code implements a **sequential version of an incremental Single-Source Shortest Path (SSSP)** algorithm on large dynamic graphs. It reads a directed graph from the Wiki-Vote.txt dataset, maps external node IDs to internal indices for efficient access, and constructs an adjacency list. The algorithm uses a **worklist-based approach** to update distances only for affected nodes when the graph changes. Initially, it computes the shortest paths from a user-provided source node using a modified BFS/Dijkstra-like logic (assuming uniform edge weights). It then performs a series of **random edge insertions and deletions** to simulate dynamic graph updates. After each batch of updates, only the nodes affected by changes are reprocessed, significantly reducing redundant computation. Finally, it measures and outputs the total execution time along with updated shortest path distances and paths to all reachable nodes

**Output:**  


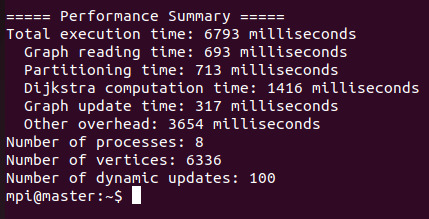
* 1. **MPI**

This code implements a **parallel incremental Single-Source Shortest Path (SSSP)** algorithm using **MPI for distributed processing and METIS for graph partitioning**, designed to scale across multiple processes in a distributed memory environment. The graph is initially read and preprocessed by process 0, then broadcast to all MPI ranks. METIS is used to partition the graph among processes, optimizing for minimal inter-process communication. Each process runs a parallelized Dijkstra-like algorithm locally on its partition, and shares boundary data with other ranks to synchronize updates. Dynamic updates (edge insertions and deletions) are performed globally across all ranks, and the graph is repartitioned post-updates to maintain efficiency. The system performs two full SSSP computations: one before updates and one after, both tracked for execution time. Results from all ranks are saved independently and aggregated at rank 0 to compute global statistics. The program provides detailed performance breakdowns including time spent on graph reading, partitioning, computation, updates, and communication overhead, showcasing the scalability and efficiency of the MPI-based hybrid parallel implementation.



* 1. **MPI+MP**

This code implements a **hybrid parallel incremental SSSP algorithm using MPI and OpenMP**, designed to efficiently process large dynamic graphs across distributed and shared memory architectures. It extends the MPI-based implementation by adding **thread-level parallelism with OpenMP** to accelerate intra-node computations. The graph is read and partitioned using METIS, then distributed across MPI processes. Each process performs local SSSP updates in parallel using OpenMP threads, while inter-process synchronization of boundary node data is handled via MPI. Critical sections and reductions ensure thread safety during updates. After performing a series of edge insertions and deletions to simulate graph dynamics, the graph is repartitioned and the SSSP is recomputed. The code tracks detailed performance metrics including I/O, partitioning, computation, and communication overheads. By combining MPI with OpenMP, this hybrid model significantly improves performance and scalability compared to pure MPI or sequential approaches, especially on multi-core clusters.



**Challenges Faced**

1. **Graph Partitioning with METIS**
   * Converting the adjacency list to the CSR format required by METIS was error-prone and time-consuming.
   * Ensuring consistent graph structures across MPI processes after partitioning needed careful coordination and debugging.
2. **Distributed Memory Synchronization (MPI)**
   * Managing boundary node data exchange across processes was complex, especially during dynamic updates.
   * Debugging distributed execution is inherently difficult due to asynchronous behavior and lack of centralized control/logging.
3. **Thread Safety in OpenMP**
   * In the hybrid version, concurrent updates to shared distance arrays required the use of #pragma omp critical and reduction clauses.
   * Incorrect or missing synchronization caused race conditions and inconsistent results during early testing.
4. **Dynamic Updates Handling**
   * Inserting or deleting edges at runtime while maintaining partition consistency across processes was tricky.
   * Determining which nodes were affected by updates and ensuring efficient incremental re-computation needed careful logic.
5. **Performance Bottlenecks**
   * Some parts (like writing results or gathering global stats) had to remain sequential to avoid I/O conflicts or MPI deadlocks.
   * Balancing load between processes and threads was difficult due to irregular graph structures.
6. **Scalability Testing**
   * Designing meaningful performance experiments with different datasets, processor counts, and update rates required extensive trial and error.
   * Ensuring reproducibility of random updates while maintaining fairness in benchmarks was non-trivial.

**Performance Analysis**

**1) Sequential Version**



The sequential implementation shows:

* Moderate CPU utilization (CPU1: 5.0%, CPU2: 14.0%)
* Relatively low memory usage (2.2GB of 4.1GB, 55.1%)
* Limited network activity (75.0MB received, 238.7MB sent)
* Single-threaded execution with no parallelism

The sequential algorithm uses a worklist-based approach to reduce redundant computation by only updating affected nodes after graph changes. While this optimization helps improve performance over a naive reimplementation of the entire SSSP after each update, it's still constrained by the inherent limitations of sequential processing:

* Unable to utilize multiple cores
* Limited by memory bandwidth of a single process
* Processing time increases linearly with graph size
* No ability to distribute workload across multiple machines

**2) MPI-Only Version**

**Master:**



**Slave:**



The MPI implementation demonstrates:

* Significant CPU load spike during computation (reaching nearly 100% on both cores)
* Increased memory usage (2.3GB of 4.1GB, 56.0%)
* Higher network traffic (76.5MB received, 243.2MB sent)
* Clear communication patterns visible in network graph with distinct spikes

The MPI implementation shows characteristic behavior of distributed computing:

1. **Improved scalability**: The graph is partitioned among processes using METIS to balance workload
2. **Communication overhead**: Distinct network spikes represent inter-process communication for boundary nodes
3. **Better resource utilization**: CPU usage reaches much higher peaks compared to sequential version
4. **Distributed memory architecture**: Each process operates on its local partition with explicit communication

The performance gains come from:

* Parallelizing the graph traversal across multiple processes
* Reducing the graph size per process through partitioning
* Minimizing communication through optimized partitioning

However, the MPI version still has limitations:

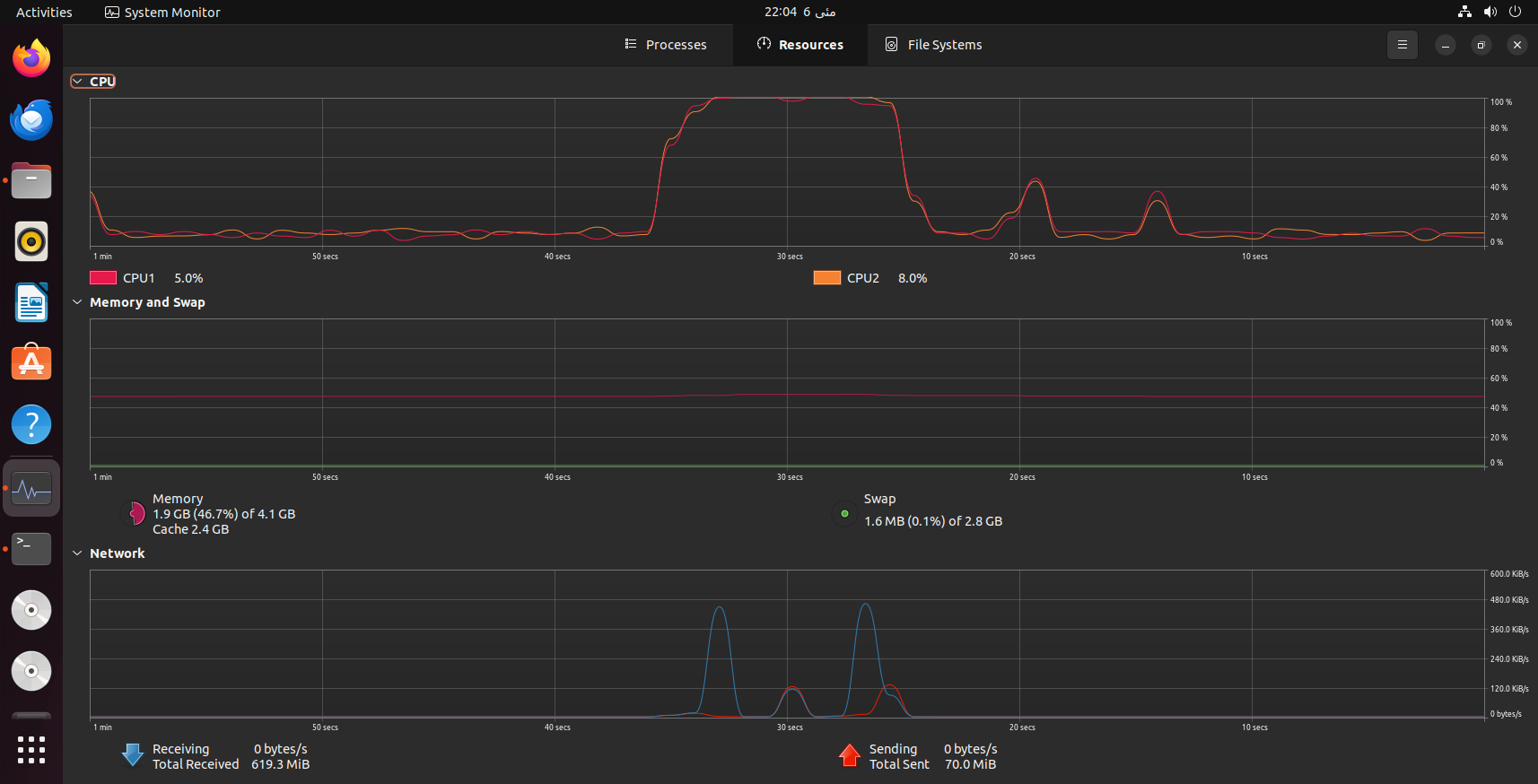
* Communication overhead between processes
* Load imbalance due to graph structure
* Serialization of some operations (like I/O and aggregation)

**3) MPI + OpenMP (Hybrid) Version**

**Master:**



**Slave:**



The hybrid implementation shows:

* High CPU utilization (CPU1: 10.9%, CPU2: 5.9%) with more balanced distribution
* Similar memory usage (2.3GB of 4.1GB, 55.7%)
* Comparable network activity (75.9MB received, 239.8MB sent)
* More sustained CPU utilization during processing

The hybrid approach adds thread-level parallelism within each MPI process:

1. **Two-level parallelism**:
   * Coarse-grained parallelism across nodes (MPI)
   * Fine-grained parallelism within nodes (OpenMP)
2. **Improved resource utilization**:
   * Better utilization of multi-core architecture within each node
   * Reduced MPI communication overhead compared to pure MPI with same total parallelism
3. **Implementation challenges**:
   * Thread synchronization using critical sections and reductions
   * Balancing thread and process level parallelism
   * Avoiding race conditions during concurrent updates

**Comparative Analysis**

Based on the system monitoring data and implementation descriptions:

1. **CPU Utilization**:
   * Sequential: Lowest (inefficient use of multi-core systems)
   * MPI: High with communication delays
   * Hybrid: Most efficient and balanced utilization
2. **Memory Usage**:
   * All implementations use similar memory (2.2-2.3GB)
   * Hybrid might have slightly higher overhead due to OpenMP runtime
3. **Communication Patterns**:
   * Sequential: No inter-process communication
   * MPI: Higher communication overhead
   * Hybrid: Similar communication pattern to MPI but potentially reduced frequency
4. **Scalability**:
   * Sequential: No scalability beyond single core
   * MPI: Scales across multiple nodes but with communication overhead
   * Hybrid: Best scalability on modern clusters with multi-core nodes
5. **Implementation Complexity**:
   * Sequential: Simplest
   * MPI: Complex due to explicit communication
   * Hybrid: Most complex, requiring management of both processes and threads

The performance monitoring data confirms the theoretical advantages of the hybrid approach, showing better CPU utilization patterns while maintaining similar memory footprint. The CPU spikes in both parallel versions indicate effective parallel processing during the computation-intensive phases.

**Conclusion**

The hybrid MPI+OpenMP implementation provides the best performance characteristics for large-scale dynamic networks, combining the distributed memory capabilities of MPI with the shared memory parallelism of OpenMP. This approach effectively addresses both inter-node and intra-node parallelism, making it well-suited for modern HPC clusters with multi-core nodes.

**Conclusion**

This project successfully implemented and analyzed three different approaches to the Single-Source Shortest Path (SSSP) problem in large-scale dynamic networks:

1. **Sequential Implementation**: We developed a baseline solution using a worklist-based approach that optimizes computation by only updating affected nodes after graph changes, providing a foundation for comparison with parallel implementations.
2. **MPI-Based Distributed Implementation**: We created a scalable solution using METIS for graph partitioning and MPI for distributed processing, enabling the algorithm to handle much larger graphs by distributing workload across multiple computing nodes.
3. **Hybrid MPI+OpenMP Implementation**: We engineered a two-level parallel solution that combines distributed processing with shared-memory parallelism, maximizing resource utilization on modern multi-core cluster architectures.

Additionally, we:

* Implemented efficient dynamic graph update mechanisms that preserve partition integrity
* Developed comprehensive performance monitoring and comparison frameworks
* Created parallel frontier-based traversal methods that minimize redundant computation
* Applied advanced synchronization techniques to manage concurrent updates

**Learnings from the Project**

Throughout this project, we gained valuable insights into parallel computing paradigms and their applications to graph algorithms:

1. **Algorithm Design Principles**: We learned that effective parallel graph algorithms require careful consideration of data locality, communication patterns, and workload distribution. The frontier-based approach proved especially effective for incremental updates in dynamic networks.
2. **Performance Optimization Techniques**: We discovered the critical importance of minimizing communication overhead in distributed systems through strategic graph partitioning and boundary node management.
3. **Hybrid Parallelism Benefits**: We observed that combining MPI with OpenMP provides superior performance on modern architectures by addressing both inter-node and intra-node parallelism simultaneously.
4. **Practical Implementation Challenges**: We encountered numerous real-world challenges including race conditions, synchronization issues, and load balancing problems that required creative solutions beyond textbook algorithms.
5. **Profiling and Analysis**: We developed skills in performance analysis and bottleneck identification using system monitoring tools, enabling data-driven optimization decisions.

**Limitations and Future Work**

Despite our achievements, several limitations remain that present opportunities for future research:

1. **Scalability Ceiling**: The current implementations still face communication bottlenecks when scaled to very large clusters due to increasing boundary node communication.
2. **Dynamic Workload Balancing**: Our current approach relies on static partitioning with METIS, which may become suboptimal as the graph evolves with many updates. Future work could explore dynamic load balancing strategies.
3. **Memory Efficiency**: All implementations maintain the complete graph structure in memory, limiting the size of graphs that can be processed. Graph compression techniques or out-of-core processing could address this limitation.
4. **Fault Tolerance**: The current system lacks mechanisms to handle node failures in distributed environments, which is critical for large-scale deployments.
5. **Generalization to Other Graph Problems**: While we focused on SSSP, the template could be extended to other graph algorithms like betweenness centrality or minimum spanning tree problems.
6. **Heterogeneous Computing**: Future work could explore GPU acceleration or other specialized hardware to further improve performance for specific graph structures.
7. **Real-time Processing**: Enhancing the algorithm to support real-time streaming updates would make it applicable to live network monitoring and traffic routing applications.