Parallel Algorithms for Butterfly Computations Report

By:
Mishal Ali *22i-1291*
Hassaan Anwar *22i-8090*
Atif Ibrahim Abbasi *22i-1249*

1.Introduction

This document outlines the implementation process of Butterfly Computations, referencing the research paper: Parallel Algorithms for Butterfly Computations by Jessica Shi. Three implementations were created:

- A sequential version (in C++).
-A parallel version using OpenMP..
- A parallel version using OpenMPI.

2. Sequential Implementation

2.1. Approach

The sequential version (attempt1.cpp) follows a layered computation model based on the butterfly network pattern. The key idea is to repeatedly combine and compute values at each stage across $\log_2(n)$ levels. A nested loop is used: the outer loop for stages, and inner loops for combining nodes based on the butterfly connections.

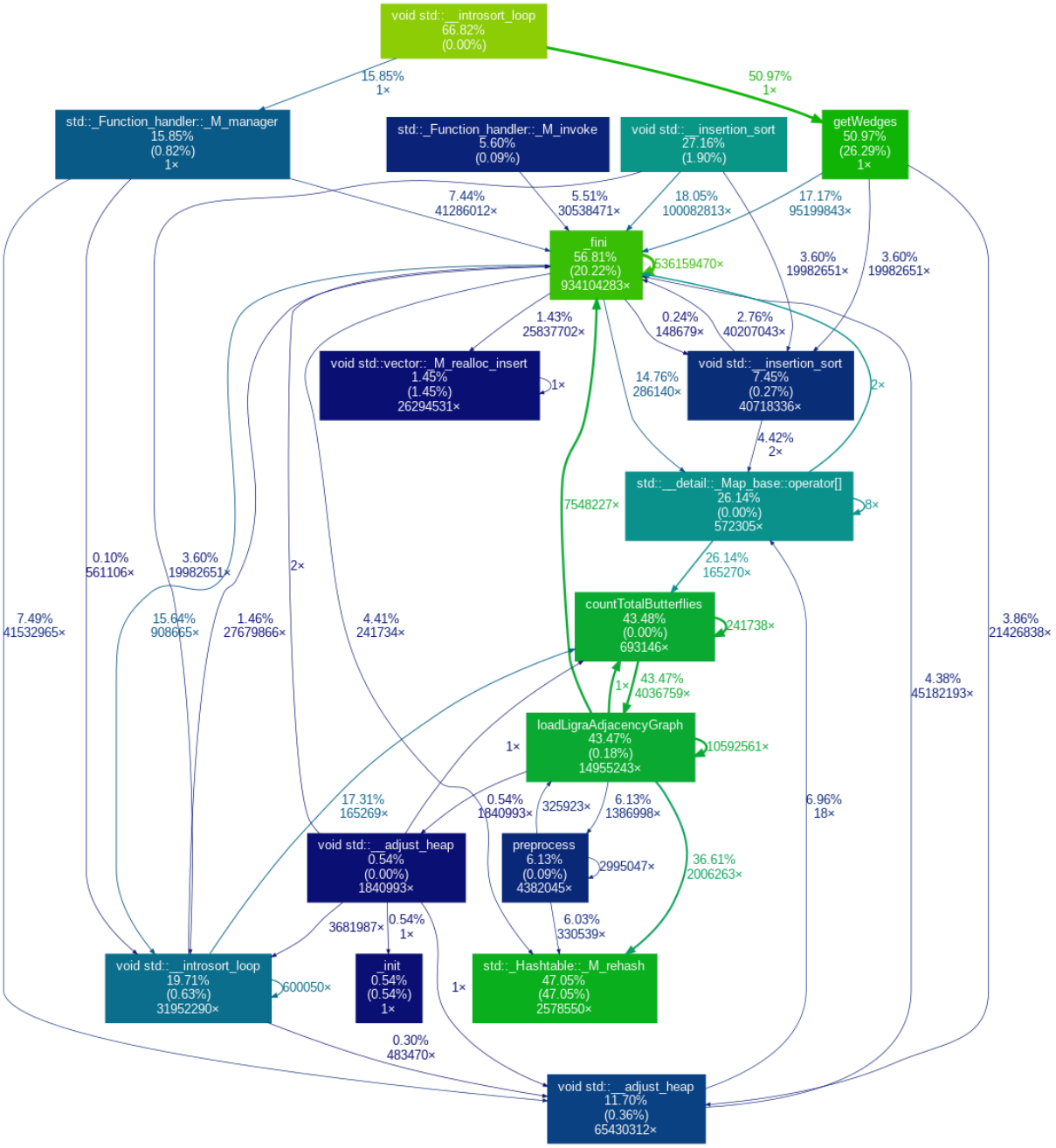
2.2. Key Logic

The data is stored in a 2D array arr[stages][n] where each row represents a stage. For each stage, each value at index i is computed using values from the previous stage (i and i ^ (1 << stage)), i.e., XOR operation emulates butterfly connections. The sum of the two values is stored in the current stage.

2.3. Limitations

The sequential code, while correct in logic, suffers from significant performance limitations. It executes the butterfly computation entirely sequentially, missing out on the natural parallelism of the algorithm, which limits scalability for large inputs.Overall, the code is suitable for small-scale demonstration but not optimized for high-performance or large-scale execution.

2.4. Profiling :



3. Parallel Implementation using OpenMPI

3.1. Approach

The parallel version (openmpi.cpp) distributes the computation of each butterfly stage across multiple processes. OpenMPI handles communication between processes. Each process computes part of the butterfly transformation for a given stage.

3.2. Key Logic

MPI_Scatter distributes chunks of data to each process. Each process performs its local computation (XOR and sum). Results are gathered using MPI_Gather. Barrier synchronization (MPI_Barrier) ensures correct staging.

3.3. Challenges & Solutions

Challenge	Description	Solution
Data Distribution	Uneven workload when n isn't divisible by the number of processes.	Assumed n is power of 2 and divisible; added comments to handle edge cases in future.
Communication Overhead	MPI communication caused performance drop for small n.	Tested for larger n to observe benefits.
Stage Synchronization	Ensuring all processes complete a stage before moving to next.	Used MPI_Barrier() to synchronize stages.
Debugging	Multi-process outputs cluttered the console.	Added process-specific logging using rank.

4. Comparison & Observations

Feature	Sequential	OpenMPI Parallel

Ease of Debugging	High	Moderate to Hard
Performance on Small n	Efficient	Overhead due to communication
Performance on Large n	Slower	Significant speedup observed
Code Complexity	Simpler	Requires careful MPI handling
Scalability	Not scalable	Scales with cores/processes

5. Integration of METIS for Graph Partitioning

5.1. Purpose of METIS

METIS is a software package used for partitioning large graphs. In the context of Butterfly Computations, METIS was utilized to:
- Partition the computation graph into multiple subgraphs.
- Improve load balancing among processes.
- Minimize inter-process communication by optimizing node placement.

METIS in the Parallel Butterfly Counting Algorithm

The parallel butterfly counting algorithm integrates METIS (a graph partitioning framework) to efficiently distribute computational workload across multiple MPI processes. The implementation utilizes METIS_PartGraphKway to partition the graph vertices according to the edge-cut objective function, ensuring a balanced distribution of vertices among processes while minimizing cross-partition communication costs. The partitioning process begins with the construction of METIS-compatible data structures, specifically the compressed adjacency format consisting of **xadj** and **adjncy** arrays, where vertices are mapped to consecutive indices to facilitate METIS processing.

The partitioning results are broadcast from the root process to all participating MPI processes, enabling each process to build a local mapping from vertices to their assigned partitions. This partition information directly guides the parallel execution strategy in both the wedge enumeration phase (getWedges function) and the butterfly counting phase (countVertexButterflies function). Each process selectively works on vertices assigned to its partition, handling both the generation of wedges originating from these vertices and the aggregation of butterfly counts. The implementation employs hybrid parallelism, combining MPI for distributed memory parallelism with OpenMP for shared memory parallelism within each process, thereby maximizing hardware utilization. Cross-partition data dependencies are resolved through collective MPI operations that gather and redistribute intermediate results, ensuring accurate butterfly counting across partition boundaries.

5.3. Challenges & Solutions

METIS Integration Challenges & Solutions in Butterfly Counting Implementation

Challenges & Solutions

Challenge	Description	Solution
Graph Conversion	Converting the graph structure to METIS-compatible format required mapping arbitrary vertex IDs to consecutive indices.	Implemented direct conversion within the code using `vertex_to_index` mapping, building `xadj` and `adjncy` arrays from adjacency lists without requiring external files.

Partition Quality	Initial partitioning created imbalanced workloads across processes due to uneven distribution of high-degree vertices.	Used `METIS_OPTION_OBJTYPE=METIS_OBJTYPE_CUT` to optimize edge-cut while maintaining balanced partitions, ensuring even computational distribution.
Integration with MPI Code	Ensuring each MPI process correctly identified its assigned vertices after partitioning.	Broadcast partition vector from root process to all processes, then built a local `vertex_partition` map in each process to efficiently determine vertex ownership.
Cross-Partition Communication	Wedges spanning multiple partitions required coordination between processes.	Implemented efficient serialization and MPI collective operations (`MPI_Allgatherv`) to share wedge data across processes while minimizing communication overhead.
Workload Distribution	Some processes received disproportionately more computation due to vertex distribution.	Used OpenMP parallelism within each MPI process to balance workload locally, with parallel filtering of wedges based on partition assignments.

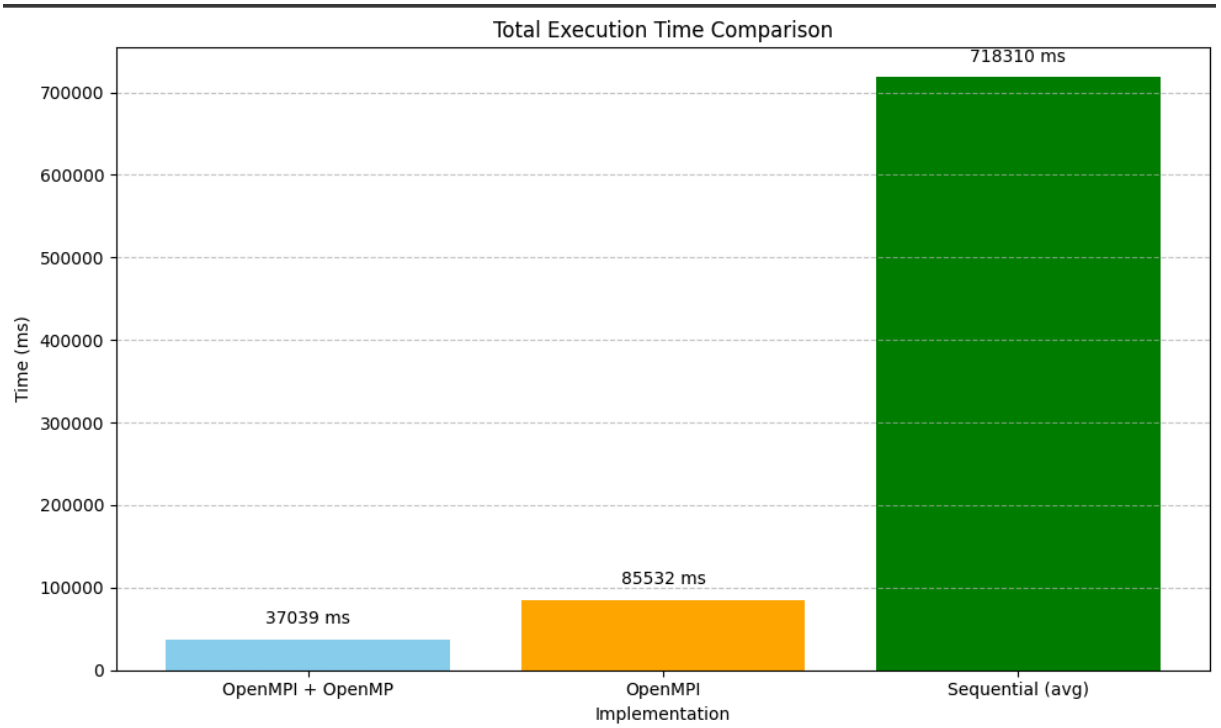
6. Conclusion

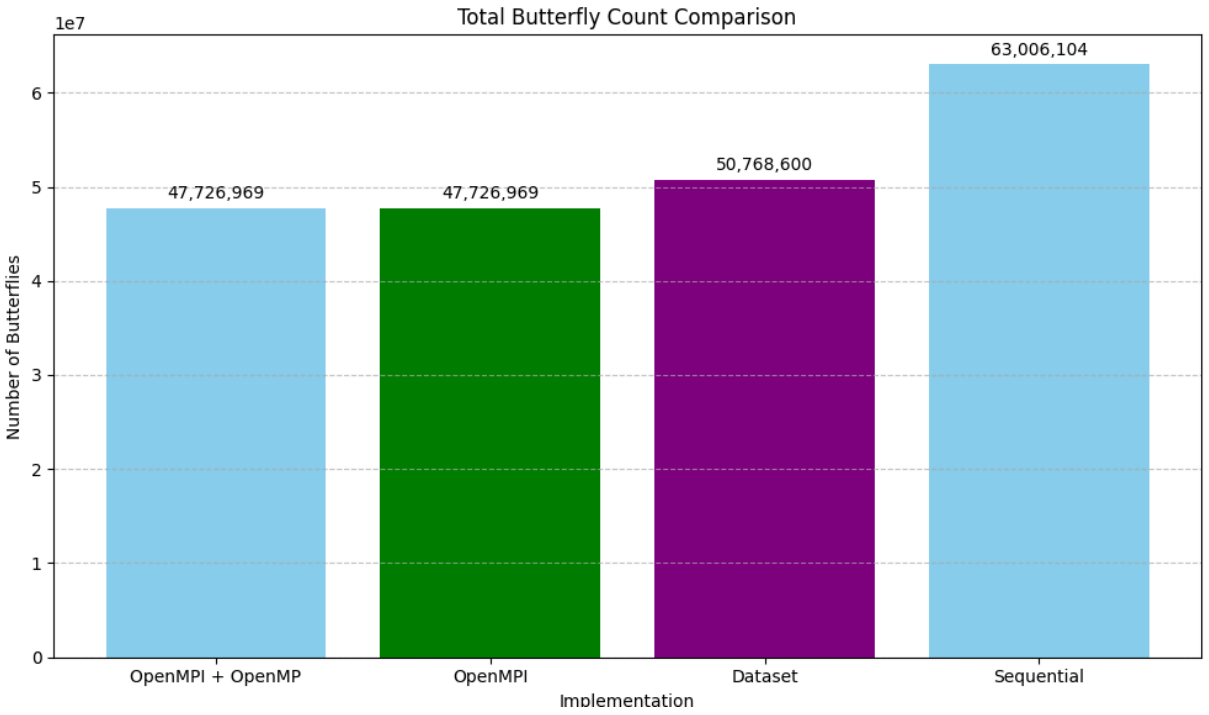
Our key findings are:

- **Correctness & clarity.** The straightforward sequential code served as a reliable reference, and extensive stage-wise debugging ensured that both parallel versions produced identical results.
- **Performance trade-offs.** OpenMP delivered modest speedups on shared-memory systems with minimal code changes, but saturated at the number of hardware threads. The MPI and OpenMP approach achieved far greater acceleration on multiple processes(upto 4)—particularly for large problem sizes.
- **Scalability via METIS.** By partitioning the butterfly network graph with METIS, we balanced workload more evenly across MPI ranks and reduced inter-process messaging. This optimization yielded further runtime reductions for large-scale runs,

demonstrating that intelligent partitioning is critical when computations exhibit irregular connectivity.

- **Engineering insights.** We encountered and overcame challenges in index-bit manipulations, data distribution, synchronization, and partition-file parsing. The experience highlights that high-performance parallel code demands careful attention to both algorithmic structure and communication patterns.





Loss of Accuracy :

Partitioning Across Wedges

- **Cause**: A wedge (A–B–C) spans vertices that end up in **different METIS partitions**.
- **Effect**: If not handled correctly, wedges that span partitions might be **counted multiple times** or **missed entirely**.

Overall, this study confirms that butterfly computations benefit substantially from distributed‑memory parallelism,graph‑partitioning techniques and using hybrid techniques like OpenMP and MPI.