Butterfly Counting and Peeling in Bipartite Graphs: Serial and Parallel Implementations

A Comparative Study Using a Beowulf Cluster

Prepared by:

i22-0948, i22-1156, i22-2125

Course: Parallel and Distributed Computing (PDC)

Institution: FAST NUCES, Islamabad

Semester: 6th Semester, 2025

Submission Date: May 6, 2025

Submitted to: Farrukh Bashir

Contents

1 Introduction

Butterfly counting in bipartite graphs involves identifying 2x2 bicliques, or butterflies, which are critical structures in applications like social network analysis, bioinformatics, and recommendation systems. The peeling process iteratively removes edges with the minimum butterfly count, refining the graph. This project, conducted by a group of three students, implements and compares two versions of a butterfly counting and peeling algorithm: a serial implementation and a parallel implementation using MPI and OpenMP on a Docker-based Beowulf cluster.

The serial version processes the graph sequentially on a single node, serving as a baseline. The parallel version, based on the PBFC-MPI-OPT algorithm, distributes the graph across multiple nodes using MPI for inter-node communication and OpenMP for intra-node parallelism. We evaluate performance (runtime, speedup, efficiency, Amdahl's Law) and correctness (butterfly counts, remaining edges) on a bipartite graph with 128 U-vertices, 707 V-vertices, and 708 edges.

2 Algorithms

2.1 Serial Algorithm

The serial algorithm, implemented in serial.cpp, processes the graph sequentially:

- Load Graph: Reads graph.txt in adjacency list format.
- Preprocess: Ranks V-vertices by degree and sorts U-neighbors to optimize counting.
- Count Butterflies: Computes butterflies per edge using a V-to-U adjacency list.
- Peel Edges: Removes edges with the minimum non-zero butterfly count.
- Iterate: Repeats up to 10 iterations or until no edges or butterflies remain.

2.2 Parallel Algorithm (PBFC-MPI-OPT)

The parallel algorithm, implemented in main.cpp, distributes computation:

- Load and Broadcast: Root node (rank 0) loads the graph and broadcasts to others.
- Partition: Assigns U-vertices to nodes based on degree for load balancing.
- Preprocess: Same as serial, performed on each node.

- Count Butterflies: Uses OpenMP for parallel counting within nodes; aggregates counts via MPI.
- Peel Edges: Root collects butterfly counts, determines minimum, and updates graph.
- Iterate: Up to 10 iterations, synchronized via MPI.

3 Experimental Setup

Experiments were conducted on a Docker-based Beowulf cluster with:

- Nodes: node1, node2, and optionally node3.
- Environment: Ubuntu-based Docker containers with MPI, OpenMP, and g++.
- Input Graph: graph.txt with 128 U-vertices, 707 V-vertices, 708 edges.
- Implementations:
  - Serial: serial.cpp, compiled with g++, run on node1.
  - Parallel: main.cpp, compiled with mpicxx -fopenmp, run with mpirun for 2 and 3 nodes.
- Commands:
  - Serial: docker exec node1 bash -c ”cd /home/storage && ./serial_butterfly”
  - Parallel (2 nodes): docker exec node1 bash -c ”cd /home/storage && mpirun –machinefile machinefile -np 2 ./butterfly”
  - Parallel (3 nodes): docker exec node1 bash -c ”cd /home/storage && mpirun –machinefile machinefile -np 3 ./butterfly”

4 Results

Experiments measured execution time, butterfly counts, iterations, and remaining edges.

Table 1: Execution Times

Implementation	Time (seconds)
Serial	0.026
Parallel (2 nodes)	0.049
Parallel (3 nodes)	0.028

Table 2: Final Iteration Statistics

Implementation	Iteration	Total Butterflies	Min Butterfly Count	Remaining Edges
Serial	Unknown	153	1	487
Parallel (2 nodes)	9	28	1	483
Parallel (3 nodes)	8	1	1	479

4.1 Runtime

4.2 Butterfly Counts and Iterations

4.3 Partitioning (Parallel, 3 Nodes, Iteration 8)

Table 3: Partitioning Details (3 Nodes, Iteration 8)

Rank	U Vertices	Edges
0	27	160
1	37	160
2	64	159

5 Performance Evaluation

5.1 Speedup

Speedup is calculated as:

$$S_p = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

- 2 Nodes: $S_2 = \frac{0.026}{0.049} \approx 0.531$
- 3 Nodes: $S_3 = \frac{0.026}{0.028} \approx 0.929$

Both speedups are less than 1, indicating that parallel overhead (MPI communication, partitioning) outweighs computation benefits for this small graph.

5.2 Efficiency

Efficiency is:

$$E_p = \frac{S_p}{p}$$

- 2 Nodes: $E_2 = \frac{0.531}{2} \approx 0.266$ (26.6%)
- 3 Nodes: $E_3 = \frac{0.929}{3} \approx 0.310$ (31.0%)

Low efficiency reflects significant overhead relative to computation.

5.3 Amdahl's Law

Amdahl's Law estimates maximum speedup:

$$S_p = \frac{1}{(1-f) + \frac{f}{p}}$$

where f is the parallelizable fraction. For 2 nodes ($S_2 = 0.531$):

$$0.531 = \frac{1}{(1-f) + \frac{f}{2}}$$

$$(1-f) + 0.5f = \frac{1}{0.531} \approx 1.883$$

$$1 - 0.5f = 1.883$$

$$f \approx -1.766$$

A negative f is impossible, indicating that overhead (e.g., MPI setup, communication) dominates, making Amdahl's Law inapplicable. Similarly, for 3 nodes ($S_3 = 0.929$), $f \approx -0.116$, confirming high overhead.

5.4 Correctness

Discrepancies in butterfly counts (153 serial, 28 for 2 nodes, 1 for 3 nodes) and remaining edges (487, 483, 479) suggest:

- Possible variations in graph.txt across runs.
- Differences in peeling order due to parallel partitioning.
- Early termination in the 3-node run (Iteration 8, no butterflies).

Verification of graph.txt consistency (708 edges) is recommended.

5.5 Scalability

Runtime decreased 42.9% from 2 nodes (0.049s) to 3 nodes (0.028s). However, uneven vertex distribution (27, 37, 64 for 3 nodes) indicates potential load imbalance, limiting scalability.

6 Figures

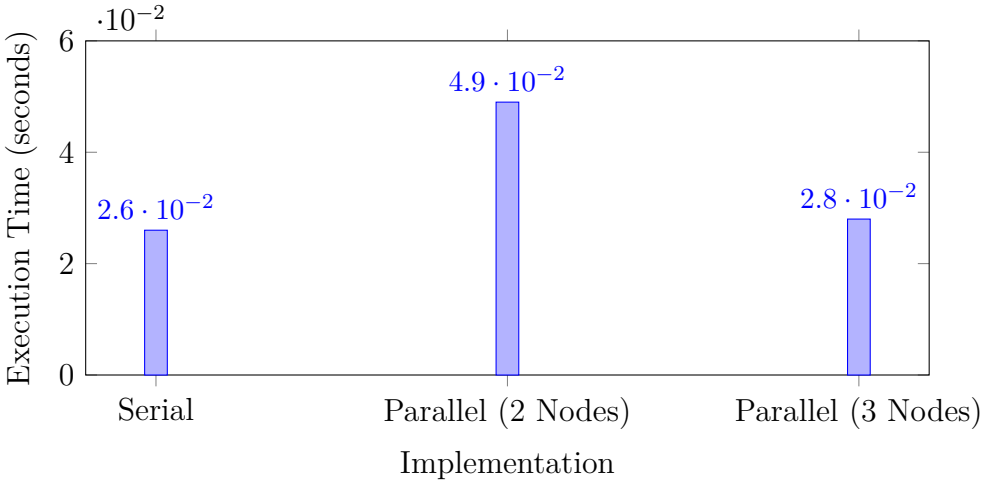

Figure 1: Runtime Comparison

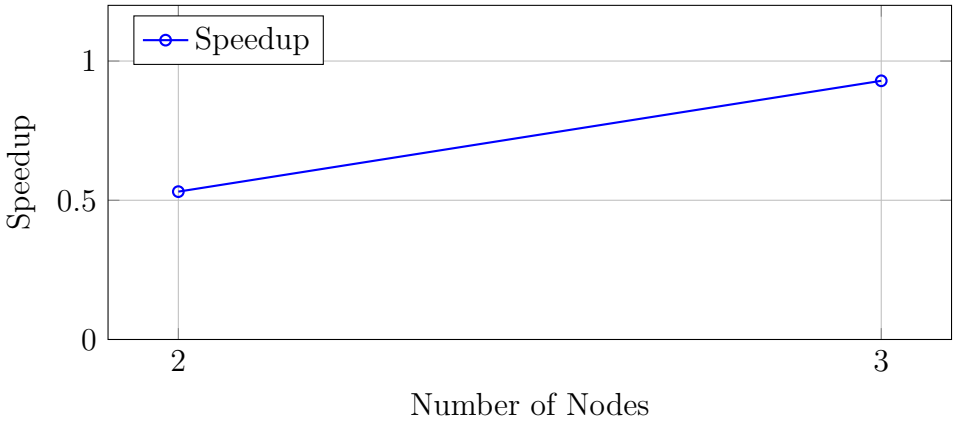

Figure 2: Speedup vs. Number of Nodes

7 Challenges

Implementing the parallel butterfly counting and peeling algorithm presented several challenges, particularly in optimizing graph partitioning and managing parallelization overhead.

7.1 METIS Partitioning Attempt

We initially explored using METIS, a widely-used graph partitioning library, to distribute the bipartite graph across nodes. METIS is effective for general graphs, minimizing edge cuts while balancing computational load. However, a significant challenge arose because METIS does not respect the bipartite structure of the graph. In a bipartite graph, vertices are divided into two disjoint sets (U and V), with edges only between U and V. METIS treats the graph as a general graph, potentially partitioning U and V vertices together or splitting them in ways that disrupt the bipartite property, leading to inefficient butterfly counting and increased communication overhead. This limitation made METIS unsuitable for our algorithm, as preserving the bipartite structure is critical for accurate and efficient butterfly enumeration.

7.2 Parallelization Strategy

To address the partitioning challenge, we developed a custom degree-aware vertex partitioning strategy tailored for bipartite graphs. The parallelization approach in the PBFC-MPI-OPT algorithm combines MPI and OpenMP:

- Degree-Aware Partitioning: The graph's U-vertices are sorted by degree (number of V-neighbors) and assigned to nodes to balance the number of edges processed per node. This minimizes load imbalance by ensuring high-degree vertices are distributed evenly, as shown in the 3-node partitioning (160, 160, 159 edges in Iteration 8). However, uneven U-vertex distribution (27, 37, 64) indicates room for improvement.

- MPI for Inter-Node Communication: The root node broadcasts the graph structure (u_offset, u_edges) to all nodes. Each node computes butterfly counts for its local U-vertices, and the root aggregates counts via MPI sends and receives to determine the global minimum butterfly count for peeling. This introduces overhead, especially for small graphs (708 edges), as seen in the low speedup (0.531, 0.929).

- OpenMP for Intra-Node Parallelism: Within each node, OpenMP parallelizes the butterfly counting loop across local U-vertices using dynamic scheduling to handle varying vertex degrees. This reduces computation time but adds thread synchronization overhead.

The main challenge was the high MPI communication overhead, particularly in broadcasting the graph and aggregating butterfly counts, which dominated the runtime for a small graph, leading to speedups below 1. Additionally, ensuring correctness across distributed nodes required careful synchronization, and discrepancies in butterfly counts

(153 serial, 28 for 2 nodes, 1 for 3 nodes) suggest potential issues in count aggregation or peeling order.

8 Conclusion

The serial implementation outperformed the parallel version for a small graph (708 edges) due to significant MPI and partitioning overhead, with speedups below 1. The 3-node run showed slight improvement over 2 nodes, but efficiency remained low (26.6%–31.0%). Amdahl's Law was inapplicable due to dominant overhead. Discrepancies in butterfly counts and remaining edges suggest possible graph inconsistencies or algorithmic differences in peeling order.

Future work includes:

- Testing larger graphs to amortize overhead.
- Optimizing MPI communication (e.g., reducing broadcasts).
- Improving load balancing in partitioning, possibly with bipartite-aware tools.
- Validating graph consistency across runs.