PDC Project Report
Butterfly Computations

22i-0983 Muhammad Taha

22i-0878 Muhammad Ibrahim Awais

22i-0996 Muhammad Ali

Introduction

Butterflies are the smallest non-trivial subgraphs in bipartite graphs. A butterfly is a (2,2)-biclique, consisting of two vertices from each bipartition,

with all four possible edges present, forming an "hourglass" pattern. Efficiently counting and analyzing butterflies is crucial for applications in social network analysis, spam detection, and discovering patterns within subgraphs, as these structures reveal strong connections and dense regions in data.

The research paper "Parallel Algorithms for Butterfly Computations" by Shi and Shun introduced the PARBUTTERFLY framework, which provides scalable parallel algorithms for butterfly counting and peeling (tip and wing decompositions). These algorithms are designed to exploit modern multicore architectures and provide both theoretical and practical speedups over previous approaches.

In this project, we implement and evaluate parallel butterfly counting and peeling algorithms inspired by the PARBUTTERFLY framework. Our focus is on the ranking and wedge aggregation strategies outlined in the paper, as well as wedge retrieval and both per-vertex and per-edge butterfly counting. We also implement tip and wing peeling. Our implementation supports both sequential and OpenMP-parallel execution. We evaluate scalability and performance on several public bipartite graph datasets.

Algorithm Overview

Butterfly Counting

A butterfly is a (2,2)-biclique: two vertices from each bipartition, with all four possible edges present. Counting butterflies involves finding all such subgraphs in a bipartite graph. The PARBUTTERFLY framework proceeds in four main steps:

1. Ranking: Assign a global order to vertices. This can be done by degree order, side order, or approximate degree order. The choice of ranking affects the number of wedges processed and the efficiency of the algorithm.

2. Wedge Retrieval: Enumerate all wedges (2-paths) where the center and second endpoint have higher rank than the first endpoint. This step reduces redundant work and ensures each butterfly is counted exactly once.

3. Wedge Aggregation: Group wedges by their endpoints to count how many share the same endpoints. This aggregation can be performed

using sorting, hashing, or batching, each with different trade-offs in terms of parallelism and memory usage.

4. Butterfly Counting: Use the wedge counts to compute per-vertex, per-edge, or global butterfly counts. The counts can be used for further analysis or as input to peeling algorithms.

Peeling

Peeling algorithms iteratively remove vertices (tip decomposition) or edges (wing decomposition) with the lowest butterfly count, updating the counts of affected neighbors after each removal. This process helps reveal the hierarchical structure of dense subgraphs and is analogous to k-core decomposition in unipartite graphs. Peeling can be used to identify important or central vertices and edges in the network.

Implementation Details

Code Structure

Our implementation is written in C++ and supports both sequential and parallel execution using OpenMP. The code is organized into several modules, each responsible for a specific part of the algorithm:

1. Ranking: Implements degree order, side order, and approximate degree order. The ranking is used to guide wedge retrieval and aggregation.

2. Wedge Aggregation: Implements three methods: sorting, hashing, and batching. These methods are used to efficiently group wedges by their endpoints in parallel.

3. Wedge Counting: Supports both per-vertex and per-edge counting. The per-vertex method uses the same aggregation strategies as above, while the per-edge method uses a slightly different keying mechanism.

4. Peeling: Uses a bucket data structure to store butterfly counts and supports aggregation using sorting, hashing, or batching.

5. Approximation: Supports optional edge-based or color-based sparsification for approximate counting, which can be useful for very large graphs.

Parallelization

OpenMP: Used for parallelizing loops in ranking, wedge retrieval, aggregation, and peeling. This allows the code to utilize multiple CPU cores efficiently.

MPI (planned): The code is structured to allow future extension to distributed-memory parallelism using MPI. However, the current results focus on shared-memory (OpenMP) and sequential baselines.

Challenges

Achieving correct butterfly counts required careful handling of the modifiedNeighbors vector. Initially, incorrect pruning of neighbors led to undercounting. By strictly following the paper's algorithm and ensuring that the pruning condition (rank(w) > rank(v) > rank(u)) was applied correctly, we obtained accurate results.

Ensuring thread safety in aggregation was essential. We used atomic operations and critical sections to prevent race conditions during parallel updates.

Efficiently updating butterfly counts during peeling required careful management of data structures to avoid redundant work.

Datasets

We evaluated our implementation on three bipartite graph datasets, each with different sizes and characteristics. All datasets were preprocessed to remove self-loops and duplicate edges.

Dataset Name	|U|	|V|	|E|	#Butterflies	Notes
dataset1	128	128	708	712	Small test graph
github	56,519	120,867	440,237	50,894,505	KONECT, user-repo
pinterestSet	55,187	9,916	186,137	1,807,806	Pinterest, user-board

Results and Analysis

Performance Comparison

We measured execution time for three solutions on each dataset:

- Sequential: Our own sequential implementation.
- Parallel (Ours): Our OpenMP-based parallel implementation.
- PARButterfly: The reference parallel implementation from Shi and Shun.

The main metric is speedup, defined as the Sequential speed over the Parallelized speed.

Performance Comparison Table

Dataset	|U|	|V|	|E|	#Butterflies	Version	#Threads	Time (s)	Speedup
dataset1	128	128	708	712	Sequential	1	0.003	1.0
					Parallel (Ours)	6	0.00684	0.44
					PARButterfly	6	0.00149	2.01
github	56,519	120,867	440,237	50,894,505	Sequential	1	5.63	1.0
					Parallel (Ours)	6	1.36	4.14
					PARButterfly	6	0.675	8.34
pinterestSet	55,187	9,916	186,137	1,807,806	Sequential	1	0.615	1.0
					Parallel (Ours)	6	0.213	2.89
					PARButterfly	6	0.0634	9.70

Notes:

- All implementations produced the same butterfly count for each dataset, confirming correctness.
- For the smallest dataset, parallel overheads make our parallel version slower than sequential, but PARButterfly is fastest.
- For larger datasets, both our parallel and PARButterfly implementations show significant speedup over sequential, with PARButterfly consistently the fastest.

Scalability

On the github and pinterestSet datasets, our parallel implementation achieves between 2.9x and 4.1x speedup with 6 threads. PARButterfly achieves between 8.3x and 9.7x speedup (the number of threads is not specified, but is likely similar or higher). On the smallest dataset, parallel overhead outweighs the benefits, as expected. This demonstrates strong scaling for larger datasets.

Discussion and Findings

Parallelization Impact: For real-world datasets, OpenMP parallelization provides substantial speedup. PARButterfly's highly optimized implementation achieves even greater speedup, likely due to more advanced parallelization techniques and lower overhead.

Bottlenecks: On small graphs, parallel overhead dominates, making parallel execution slower than sequential. On larger graphs, memory bandwidth and synchronization become limiting factors as the number of threads increases.

Algorithmic Efficiency: The batching aggregation method and careful ranking (using approximate degree order) are effective, as demonstrated by both our results and those of PARButterfly.

Correctness: All implementations agree on the butterfly count, confirming the correctness of our approach.

Future Work: Extending the implementation to distributed-memory parallelism using MPI and further optimizing memory usage are promising directions for future research.

Conclusion

We implemented and evaluated parallel butterfly counting and peeling algorithms inspired by the PARBUTTERFLY framework. Our OpenMP implementation achieves substantial speedup over the sequential baseline, especially on large bipartite graphs. The batching aggregation method is the most efficient in practice. Future work includes extending to distributed-memory (MPI) and further optimizing memory usage for even larger graphs.

References

Shi, J., & Shun, J. (2020). Parallel Algorithms for Butterfly Computations. SIAM Journal on Scientific Computing, 42(5), C320-C349.

http://konect.cc/networks/

https://github.com/jeshi96/parbutterfly/

Appendix

https://github.com/i220878/PDC_Project