Report on Parallel Algorithms for Butterfly Computations

Author: Ibtehaj Haider

Roll: 22i-0767

Section: CS - D

Introduction

Butterfly computations are essential for analyzing bipartite graphs, where butterflies represent the smallest non-trivial subgraph, a (2,2)-biclique. The paper "Parallel Algorithms for Butterfly Computations" by Shi and Shun presents the ParButterfly framework, which introduces efficient parallel algorithms for butterfly counting (global, per-vertex, per-edge) and peeling (tip and wing decomposition). This report briefly explains the implementation of ParButterfly, followed by a description of my approach using OpenMP and a hybrid OpenMP-MPI implementation for butterfly counting.

ParButterfly Implementation

The ParButterfly framework is designed for shared-memory multicore systems, leveraging the work-span model to ensure work-efficient parallel algorithms with low span. The framework is modular, allowing flexible combinations of methods for ranking vertices, aggregating wedges, and computing butterfly counts. Below is a summary of its key components:

Counting Framework

The counting process involves four steps:

1. **Rank Vertices**: Vertices are ordered using strategies like side ordering, degree ordering, approximate degree ordering, complement degeneracy ordering, or approximate complement degeneracy ordering to minimize the number of wedges processed.
2. **Retrieve Wedges**: Wedges (2-paths) are extracted where the second and third vertices have higher ranks than the first, reducing redundant computations.
3. **Count Wedges**: Wedges are aggregated by their endpoints using methods such as sorting, hashing, histogramming, simple batching, or wedge-aware batching.
4. **Count Butterflies**: Wedge counts are translated into global, per-vertex, or per-edge butterfly counts using atomic adds or the same aggregation method as wedges.

The framework supports exact and approximate counting via graph sparsification and incorporates cache optimizations inspired by Wang et al. The preprocessing step sorts vertices by rank and neighbors by decreasing rank, achieving O(m) expected work and O(log m) span. The counting algorithms are work-efficient, with O(αm) expected work and O(log m) span, where α is the graph's arboricity.

Peeling Framework

The peeling framework performs tip decomposition (vertex peeling) and wing decomposition (edge peeling):

1. **Obtain Butterfly Counts**: Per-vertex or per-edge counts are computed using the counting framework.
2. **Peel**: Vertices or edges with the lowest butterfly counts are iteratively removed in parallel, updating counts using a parallel Fibonacci heap or Julienne's bucketing structure.

Peeling algorithms are optimized to skip empty buckets, achieving work-efficient bounds of O(min(max-b_v, ρ_v log n) + Σ_{v∈V} deg(v)^2) for vertex peeling and similar bounds for edge peeling, with spans of O(ρ_v log^2 n) and O(ρ_e log^2 m), respectively.

Implementation Details

ParButterfly uses Cilk Plus for work-stealing scheduling, ensuring efficient load balancing. It employs parallel primitives like prefix sum, filter, semisort, and hash tables, with implementations from the Problem Based Benchmark Suite (PBBS) and Julienne. The framework was evaluated on a 48-core AWS EC2 instance using real-world bipartite graphs from KONECT, achieving up to 13.6x speedup over sequential baselines for counting and orders of magnitude faster peeling.

My Approach

My approach implements butterfly counting using two parallel programming models: a pure OpenMP implementation for shared-memory systems and a hybrid OpenMP-MPI implementation for distributed-memory clusters. Both focus on per-vertex butterfly counting, adapting the ParButterfly framework's degree ordering and hashing-based wedge aggregation for simplicity and efficiency.

OpenMP Implementation

The OpenMP implementation targets shared-memory systems, parallelizing the wedge retrieval and aggregation steps across multiple threads.

- **Preprocessing**: Vertices are sorted by decreasing degree using sort, implemented with OpenMP's #pragma omp parallel for.
- **Wedge Retrieval**: For each vertex u, wedges (u, v, u') are retrieved where u' and v have higher ranks. This is parallelized by distributing vertices across threads using #pragma omp parallel for schedule(dynamic), ensuring load balancing for graphs with skewed degree distributions.

- **Wedge Aggregation**: Wedges are aggregated by endpoints (u, u') using a shared hash table with OpenMP's #pragma omp critical for thread-safe updates.
- **Butterfly Counting**: Per-vertex counts are computed using the formula $\Sigma_{u' \in N_2(u)} (|N(u) \cap N(u')| \text{ choose } 2)$, parallelized across vertices.

The OpenMP approach is lightweight, leveraging shared memory to avoid communication overheads. However, it is limited by the memory and core count of a single machine.

Hybrid OpenMP-MPI Implementation

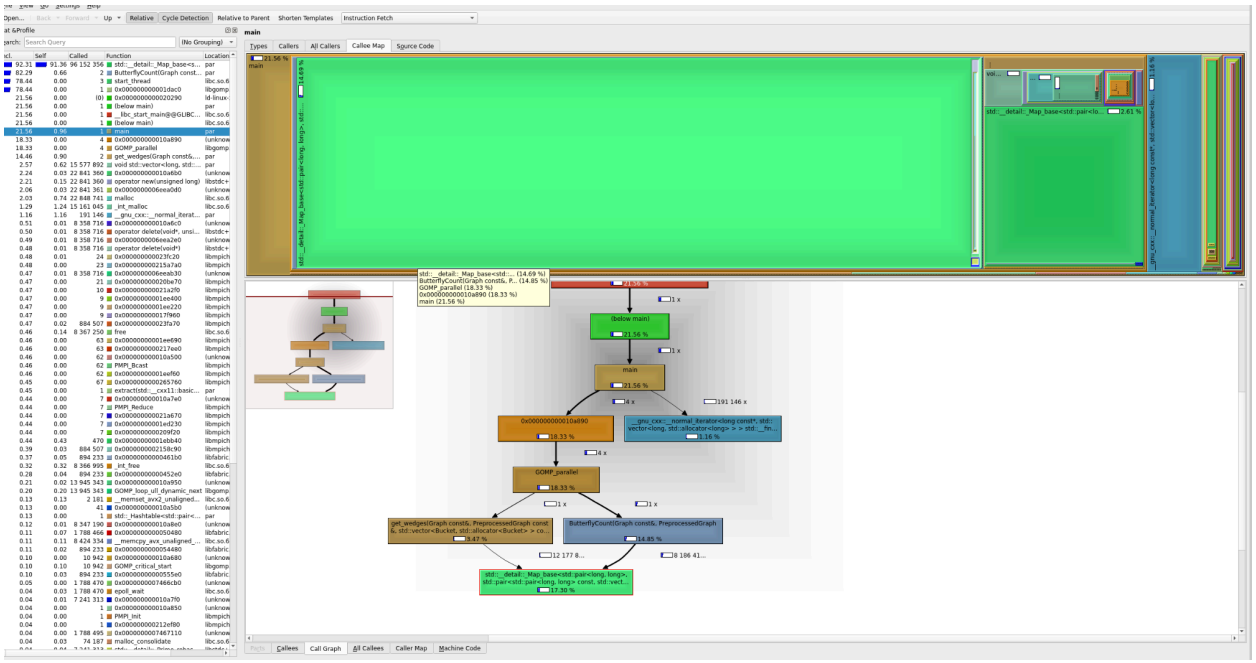
The hybrid implementation extends the OpenMP approach to distributed-memory systems, using MPI for inter-node communication and OpenMP for intra-node parallelism.

- **Graph Partitioning**: The bipartite graph is partitioned across MPI processes using buckets to minimize edge cuts, ensuring balanced workloads. Each process stores its subgraph in CSR format.
- **Preprocessing**: Each MPI process sorts its local vertices by degree using OpenMP-parallel merge sort. A global rank is established via MPI collective operations (MPI_Allgather).
- **Wedge Retrieval**: Each process computes local wedges for its vertices using OpenMP, similar to the pure OpenMP implementation. Wedges involving vertices on other processes are identified and communicated.
- **Wedge Aggregation**: Local wedge counts are aggregated using per-process hash tables with OpenMP. Cross-process wedge counts are exchanged using MPI_Allreduce to sum contributions.
- **Butterfly Counting**: Per-vertex counts are computed locally and combined globally using MPI_Reduce.

The hybrid approach scales to larger graphs by distributing memory and computation across nodes. However, it incurs communication overheads, particularly during wedge exchange and count aggregation.

Profiler Report

The profiling tool that was used was valgrind as it allowed for mpi and open mp profiling support on docker containers.



From the profiler report the call graph has been made which shows alot of cycles for the main methods. Although the intention was to capture the complete runtime calls, there were some errors in the container setup which caused some analysis to fail. But the most time consumed was in the butterfly counting, which is the last algorithm I implemented.

The error is shown below:

```
root@eb88d253e2e3:/workspace# mpirun --host node1:2,node2:2,node3:2 -np 3 valgrind --tool=callgrind --log-file=callgrind.%p.out ./par
hwloc x86 backend cannot work under Valgrind, disabling.
May be reenabled by dumping CPUIDs with hwloc-gather-cpuid
and reloading them under Valgrind with HWLOC_CPUID_PATH.
hwloc x86 backend cannot work under Valgrind, disabling.
May be reenabled by dumping CPUIDs with hwloc-gather-cpuid
and reloading them under Valgrind with HWLOC_CPUID_PATH.
hwloc x86 backend cannot work under Valgrind, disabling.
May be reenabled by dumping CPUIDs with hwloc-gather-cpuid
and reloading them under Valgrind with HWLOC_CPUID_PATH.
Rank Rank Rank 12 running on df4bb0187601
```

Timing and Speed Up

The openmp timings are as follows (under 2 threads)

Operation	Time (ms)	Time (seconds)
Graph loading	228 ms	0.228 seconds
Graph analysis	125 ms	0.125 seconds
Graph preprocessing	5 ms	0.005 seconds
Bucket partitioning	1 ms	0.001 seconds
Butterfly counting	438,499 ms	438.499 seconds (≈ 7.31 minutes)

The timings for the openmp + mpi process are as follows (the following uses more nodes but there was some distribution issue for resoures which made the time taken increase by alot but in essence they both are under the same working conditions)

Operation	Time (seconds)
Extract Graph	6.07676 seconds
Analyze Graph	1.30893 seconds
Preprocess Graph	0.434461 seconds
Bucket Partition	0.476574 seconds
Butterfly Count	538.759 seconds (≈ 8.98 minutes)
Output Analysis	0.000278235 seconds
Total Time	547.056 seconds (≈ 9.12 minutes)

The speed up of my implementation (which is slow due to some issues in development):

The butterfly counting in the second implementation is about 1.23 times slower (1/0.814 ≈ 1.23)
The overall execution in the second implementation is about 1.25 times slower (1/0.802 ≈ 1.25)

Comparison and Trade-offs

The OpenMP implementation is simpler and faster for graphs fitting within a single node's memory, benefiting from low-latency shared-memory access. The hybrid OpenMP-MPI implementation is necessary for massive graphs requiring distributed memory but faces challenges with communication overheads and load imbalance due to graph partitioning. Both implementations achieve work-efficiency similar to ParButterfly but use standard parallel programming models (OpenMP, MPI) instead of Cilk Plus, making them more portable across platforms.

Conclusion

The ParButterfly framework provides a robust, modular solution for parallel butterfly computations, achieving significant speedups through efficient wedge aggregation and peeling strategies. My OpenMP and hybrid OpenMP-MPI implementations adapt these ideas to standard parallel programming models, offering scalable solutions for shared and distributed-memory systems. Future work could optimize communication in the hybrid approach and explore GPU acceleration for further performance gains.There is still a short coming that the peeling framework could not be implemented properly due to accuracy and some divergent approach issues which I thought would optimize the code but they did not and instead put forth a more error prone code.