Parallel algorithms for butterfly computations

Jessica Shi (MIT CSAIL)
Julian Shun (MIT CSAIL)
What are butterflies?

**Butterflies** = 4-cycles = $K_{2,2}$

Think of these as the bipartite analogue of triangles ($K_3$)

*Note*: Bipartite graphs contain no triangles
Finding dense bipartite subgraphs

- **Finding dense subgraphs: (not bipartite)**
  - **K-core**: Repeatedly find + delete min degree vertex
  - **Triangle peeling** (triangle densest subgraph): approx by repeatedly find + delete vertex containing min # triangles

- **Butterfly peeling**: Repeatedly find + delete vertex containing min # of butterflies\(^1\)

- **Applications:**
  - **Link spam detection**: External links to a spam page, for self-promotion in search rankings

[1] Sariyuce and Pinar (18)
Link spam detection

- **Nodes** = webpages, **Edges** = links
- **Web communities** = dense bipartite subgraphs
  - **Bipartitions** = topics, page-creators interested in topics

![Graph Diagram]

- Dog training tips
- Professional dog grooming
- Dogs wiki
- American kennel club
- Corgi blogger
- Cats of NYC
Main goal: Build a framework ParButterfly to count and peel butterflies

New parallel algorithms for butterfly counting + peeling
ParButterfly framework with modular settings
  - Tradeoff b/w theoretical bounds + practical speedups
Comprehensive evaluation
  - Counting outperforms fastest seq algorithms by up to 13.6x
  - Peeling outperforms fastest seq algorithms by up to 10.7x
Important paradigms

- **Parallelization**
  - Shared memory
  - Work-span model:
    - \( \text{Work} = \text{total # operations} \)
    - \( \text{Span} = \text{longest dependency path} \)

- **Strong theoretical bounds**
  - \( \text{Work-efficient} = \text{work matches sequential time complexity} \)

- **Fast in practice**
ParButterfly counting framework
How do we count butterflies? (per vertex)

Wedge $= P_2 =$

Endpoints $\rightarrow$ Center
How do we count butterflies? (per vertex)

Wedge = $P_2 = \begin{array}{c}
\text{Wedges with the same endpoints form butterflies:}
\end{array}$

- # wedges w/endpoints $\bullet \quad \bullet = w = 3$
- # butterflies on endpoints $\bullet \quad \bullet = \binom{w}{2} = \binom{3}{2} = 3$
- # butterflies on each center $\bullet = w - 1 = 3 - 1 = 2$
1. Retrieve wedges
2. **Aggregate wedges**: For each pair of endpoints, count # wedges \( w \)
3. **Compute butterfly counts**:
   - + \( \binom{w}{2} \) for each endpoint
   - + \( w - 1 \) for each center

**One question**: How do we aggregate wedges?
(will discuss wedge retrieval after)
Method 1: Semisorting (on endpoints)
Method 1: **Semisorting** (on endpoints)
Method 2: **Hashing** (keys = endpoints)
Method 2: **Hashing** (keys = endpoints)
Wedge aggregating

- Method 3: Histogramming (frequencies of endpoints)

- \[ \text{green } = 3 \]
- \[ \text{blue } = 2 \]
- \[ \text{yellow } = 1 \]
Semisorting\textsuperscript{[1]}, hashing\textsuperscript{[2]}, and histogramming\textsuperscript{[3]} are all work-efficient

\[ w = \# \text{ of wedges} \]

\[ O(w) \text{ expected work, } O(\log w) \text{ span whp} \]

\[ \text{[1]} \text{ Gu, Shun, Sun, and Blelloch (15)} \]
\[ \text{[2]} \text{ Shun and Blelloch (14)} \]
\[ \text{[3]} \text{ Dhulipala, Blelloch, and Shun (17)} \]
Butterfly counts from wedge counts

Each wedge produces butterfly counts per vertex

Another question: How do we handle butterfly counts on the same vertex in parallel?

1. Use atomic adds
2. Aggregate counts in the same way we aggregated wedge counts (semisorting, hashing, histogramming)
Counting framework so far

1. Retrieve wedges
2. Aggregate wedges:
   - Semisort, Hash, Histogram
3. Compute butterfly counts:
   - Semisort, Hash, Histogram, Atomic add

One more way to count wedges: Batching
(not with polylogarithmic span, but fast in practice)
Wedge aggregating (batching)

- **Main idea**: Process a subset of vertices in parallel, finding all wedges where those vertices are endpoints.

Array of size $|V|$: 0 1 0 3
Wedge aggregating (batching)

- Each vertex requires linear memory ➔
- How many vertices do we process in parallel?
  - Simple: Fixed # based on memory available
  - Wedge-aware: Dynamically choose based on how many wedges will be processed per vertex
Counting framework so far

1. Retrieve wedges
2. Aggregate wedges:
   - Semisort, Hash, Histogram, Batch (Simple + Wedge-aware)
3. Compute butterfly counts:
   - Semisort, Hash, Histogram, Atomic add

More questions:
- How do we retrieve wedges?
- How many wedges are there?
- Method 1: Process wedges w/endpoints from one bipartition (Side) \([1]\)

It depends!

- Is this optimal (min # wedges)? Not always.

[1] Sani-Mehri, Sariyuce, Tirthapura (18)
Regardless of which side we pick, butterfly count does not change – only some “useful” wedges create butterflies.

2 “useful” wedges = 1 butterfly

(Note: Butterfly count remains the same)
Method 2: *Degree* ranking

Main idea:
Once we obtain all wedges with endpoint v, we do not have to consider wedges with endpoint v again.
Method 2: Degree ranking

1. Order vertices by non-increasing degree
2. For each vertex v, only consider wedges with endpoint v that is formed by vertices later in the ordering than v
Retrieves wedges

- Method 2: Degree ranking

![Diagram showing two graphs with nodes 1, 2, 3, 4, 5, 6, 7, 8, with red lines indicating wedges. The left graph has red edges between nodes 3 and 4, and the right graph has red edges between nodes 1 and 3. Both graphs show 2 wedges.](image-url)
- Method 2: **Degree** ranking

![Graph demonstrating degree ranking]
Retrieve wedges

- Method 2: **Degree** ranking

We only processed 4 wedges!
# wedges processed using degree order = $O(\alpha m)$
- $\alpha =$ arboricity ($O(\sqrt{m})$)
- $m =$ # edges

Therefore: (using work-efficient options)
- Ranking vertices = $O(m)$ expected work, $O(\log m)$ span whp
- Retrieving wedges = $O(\alpha m)$ expected work, $O(\log m)$ span whp
- Counting wedges = $O(\alpha m)$ expected work, $O(\log m)$ span whp
- Computing butterfly counts = $O(\alpha m)$ expected work, $O(\log m)$ span whp

Total = $O(\alpha m)$ expected work, $O(\log m)$ span whp

[1] Chiba and Nishizeki (85)
Other rankings

- **Approximate degree order**
  - Log degree

- **Complement degeneracy order**
  - Ordering given by repeatedly finding + deleting greatest degree vertex

- **Approximate complement degeneracy order**
  - Complement degeneracy order, but using log degree

We show these are all work-efficient
Counting framework

1. **Rank vertices:**
   - Side, Degree, Approx Degree, Co Degeneracy, Approx Co Degeneracy

2. **Retrieve wedges**

3. **Aggregate wedges:**
   - Semisort, Hash, Histogram, Batch (Simple + Wedge-aware)

4. **Compute butterfly counts:**
   - Semisort, Hash, Histogram, Atomic add

\[ O(\alpha m) \text{ expected work, } O(\log m) \text{ span whp} \]
ParButterfly peeling framework
How do we peel butterflies?

- **Goal**: Iteratively remove all vertices with min butterfly count

  - **Subgoal 1**: A way to keep track of vertices with min butterfly count
  - **Subgoal 2**: A way to update butterfly counts after peeling vertices

**Note**: We’ve already done subgoal 2 in counting framework

For subgoal 1, we give a work-efficient batch-parallel Fibonacci heap which supports batch insertions/decrease-keys (see paper).
Peeling framework

1. Obtain butterfly counts
2. Iteratively remove vertices with min butterfly count
   - Use *batch-parallel Fibonacci heap* to find vertex set $S$
   - Count wedges with endpoints in $S$
     - *Semisort, Hash, Histogram, Batch (Simple + Wedge-aware)*
   - Compute updated butterfly counts
     - *Semisort, Hash, Histogram*
Peeling framework bounds

- **By vertex**: ($\rho_v = \text{number of peeling rounds across all vertices}$)
  
  $O(\rho_v \log m + \sum \text{degree}(v)^2)$ expected work, $O(\rho_v \log^2 m)$ span whp

- **By edge**: ($\rho_e = \text{number of peeling rounds across all edges}$)
  
  $O(\rho_e \log m + \sum_{(u,v)} \sum_{u' \in N(u)} \min(\text{degree}(u), \text{degree}(u')))$ expected work,

  $O(\rho_e \log^2 m)$ span whp
Evaluation
m5d.24xlarge AWS EC2 instance: 48 cores (2-way hyper-threading), 384 GiB main memory

Cilk Plus\(^1\) work-stealing scheduler

Koblenz Network Collection (KONECT) bipartite graphs

Some modifications:
- Julienne\(^2\) instead of batch-parallel Fibonacci heap
- Cannot hold all wedges in memory – batch wedge retrieval

\(^1\) Leiserson (10)
\(^2\) Dhulipala, Blelloch, and Shun (17)
Counting: Best aggregation method:

**Batching**

![Graph showing the comparison of different aggregation methods: Sort, Hash, Hist, and Batch. The graph depicts the multiplicative slowdown for the discogs, enwiki, and web datasets. The x-axis represents the datasets, while the y-axis represents the multiplicative slowdown. The bars indicate the performance of each method for each dataset.](image-url)
Counting: Best ranking method:

Approx Complement Degeneracy / Approx Degree

![Graph showing comparison of different ranking methods.](image)
Butterfly counting results

- 6.3 – 13.6x speedups over best seq implementations\(^1\) \(^2\)
- 349.6 – 5169x speedups over best parallel implementations\(^3\)
  - Due to work-efficiency
- 7.1 – 38.5x self-relative speedups
- Up to 1.7x additional speedup using a cache-optimization\(^4\)

---

[1] Sanei-Mehri, Sariyuce, Tirthapura (18)
Peeling: Best aggregation method: Histogramming
Butterfly peeling results

- **1.3 – 30696x** speedups over best seq implementations\[^1\]
  - Depends heavily on peeling complexity
  - Largest speedup due to better work-efficiency for some graphs

- **Up to 10.7x** self-relative speedups
  - No self-relative speedups if small # of vertices peeled

\[^1\] Sariyuce and Pinar (18)
Conclusion
New parallel algorithms for butterfly counting/peeling

Modular ParButterfly framework w/ranking + aggregation options

Strong theoretical bounds + high parallel scalability

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

Future work:
- Cycle counting extensions
- Better work bounds for butterfly peeling
Thank you
Priority queue for butterfly counts

Batch-parallel Fibonacci heap:

- **$k$ insertions**: $O(k)$ amortized expected work, $O(\log(n+k))$ span whp
- **$k$ decrease-keys**: $O(k)$ amortized work, $O(\log^2 n)$ span whp
- **delete-min**: $O(\log n)$ amortized expected work, $O(\log n)$ span whp