Parallel Batch-Dynamic $k$-Core Decomposition

Julian Shun (MIT CSAIL)

Joint work with Quanquan Liu, Jessica Shi, Shangdi Yu, and Laxman Dhulipala
Project Presentations

- Final project presentations in class on 5/14
  - Problem and motivation
  - Prior work
  - Your technical contributions
  - Challenges encountered
  - Experimental results

- 5 minutes per person (10 minutes for groups of 2)
- ~1 minute Q&A

- Please send your slides to indsayung@mit.edu by 9am on 5/14
- We will play all slides from the same computer to minimize transition time
- Project report due 5/14
Graphs are becoming very large

**Size**

- **Common Crawl**: 3.5 billion vertices, 128 billion edges
- **Largest publicly available graph**: 272 billion vertices, 5.9 trillion edges
- **Proprietary graph**: > 100 billion vertices, 6 trillion edges

Graphs are rapidly changing

*(500M tweets/day, 547K new websites/day)*
Parallelism and Dynamic Algorithms for High Performance

- Take advantage of parallel machines

- Design dynamic algorithms to avoid unnecessary work on updates
Parallel Batch-Dynamic Algorithms

• Process updates in batches, and use parallelism within each batch

A **batch** of edge insertions/deletions

Current graph + Current statistics

Updated graph + Updated statistics

- **Insertion**
- **Deletion**
Our Parallel Batch-Dynamic Algorithms

- **k-core decomposition**
- Clique counting
- Low out-degree orientation
- Maximal matching
- Graph coloring
- Minimum spanning forest
- Single-linkage clustering
- Closest pair

**Theory**

- $O(n \log n)$
- $O(n)$
- $O(\log n)$

**Practice**

Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, Julian Shun, “Parallel Batch-Dynamic Algorithms for k-Core Decomposition and Related Graph Problems,” SPAA 2022
Related Work on Parallel Batch-Dynamic Algorithms

• Triangle Counting [EB10, MBG17]
• Euler Tour Trees [TDB19]
• Connected Components [FL94, MGB13, AABD19]
• Rake-Compress Trees [AABDW20]
• Incremental Minimum Spanning Trees [ABT20]
$k$-Core Decomposition
**k-Core Decomposition**

*k*-core: maximal connected subgraph of G such that all vertices have induced degree $\geq k$

Coreness($v$): largest value of $k$ where $v$ participates in the $k$-core

Coreness($v$) = 3

Goal: compute coreness for all vertices
Approximate $k$-Core Decomposition

$k$-core: maximal connected subgraph of $G$ such that all vertices have induced degree $\geq k$

c-Approx-Coreness($v$): value within multiplicative $c$ factor of Coreness($v$)
Applications of $k$-core Decomposition

- Graph clustering
- Community detection
- Graph visualization
- Protein network analysis
- Approximating network centrality
Work-Span Model

Work = number of operations
Span = length of longest sequential dependence

Running time ≤ (Work/\#processors) + O(Span)

• Goal: Design low-span parallel algorithms that are work-efficient (work asymptotically matches that of the best sequential algorithm)
Our Results for \( k \)-core Decomposition

- Our algorithm dynamically maintains a \((2 + \epsilon)\)-approximation for coreness of every vertex.

- A batch of \( B \) updates takes \( O(B \log^2 n) \) amortized work and \( O(\log^2 n \log \log n) \) span with high probability.

- Our algorithm is work-efficient, matching the work of the state-of-the-art sequential algorithm by Sun et al.

- Our algorithm is based on a parallel level data structure.
Sequential Level Data Structures for Dynamic Problems

- Maximal Matching [Baswana-Gupta-Sen ‘18, Solomon ‘16]


- Clustering [Wulff-Nilsen ‘12]


- Densest subgraph [Bhattacharya-Henzinger-Nanongkai-Tsourakakis ‘15]
Sequential Level Data Structure (LDS)

- Described by Bhattacharya, Henzinger, Nanongkai, Tsourakakis [2015] and Henzinger, Neumann, Wiese [2020]

- Maintain invariants per vertex, which give upper/lower bounds on roughly its number of “up-neighbors” (neighbors at around its level and above)

- We prove that levels translate to coreness estimates
Sequential Level Data Structure (LDS)

Vertices partitioned into levels

$O(\log^2 n)$

$\# \text{ up-neighbors: } > 2.1(1 + \delta)^i$

= edge insertion
Sequential Level Data Structure (LDS)

Vertices partitioned into levels

$O(\log^2 n)$

# up-neighbors: $> 2.1(1 + \delta)^i$
Sequential Level Data Structure (LDS)

$O(\log^2 n)$

Vertices partitioned into levels

# up-neighbors: $< (1 + \delta)^i$

= edge deletion
Sequential Level Data Structure (LDS)

Vertices partitioned into levels

$O(\log^2 n)$

# up-neighbors: < $(1 + \delta)^i$
Difficulties with Parallelization

Large sequential dependencies

Large span
Difficulties with Parallelization

- Large sequential dependencies
- Large span
Difficulties with Parallelization

Large sequential dependencies

Large span
Difficulties with Parallelization

Large sequential dependencies

Large span

Diagram showing interdependent nodes.
Difficulties with Parallelization

Large sequential dependencies

Large span
Difficulties with Parallelization

- Large sequential dependencies
- Large span
Difficulties with Parallelization

- Large sequential dependencies
- Only processes one update at a time
- Large span
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

Deletions

= edge deletion
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

- Vertices only need to move down, and never up

Deletions

Only the lower bound invariant is ever violated.
For vertices incident to updated edges, calculate *desire-level* (dl): closest level that satisfies invariants.

Only the lower bound invariant is ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

For vertices incident to updated edges, calculate desire-level ($dl$): closest level that satisfies invariants.

Iterate from bottommost level to top level and move vertices to desire-level.

Only the lower bound invariant is ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

For vertices incident to updated edges, calculate desire-level (dl): closest level that satisfies invariants

Deletions

Iterate from bottommost level to top level and move vertices to desire-level

Only the lower bound invariant is ever violated.

To achieve parallelism (low span), we need to move all vertices together for each desire-level
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

For vertices incident to updated edges, calculate desire-level (dl): closest level that satisfies invariants.

Iterate from bottommost level to top level and move vertices to desire-level.

Only the lower bound invariant is ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

For vertices incident to updated edges, calculate desire-level (dl): closest level that satisfies invariants.

Iterate from bottommost level to top level and move vertices to desire-level.

Only the lower bound invariant is ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

For vertices incident to updated edges, calculate desire-level (dl): closest level that satisfies invariants.

Iterate from bottommost level to top level and move vertices to desire-level.

Only the lower bound invariant is ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

For vertices incident to updated edges, calculate desire-level \( (dl) \): closest level that satisfies invariants.

Iterate from bottommost level to top level and move vertices to desire-level.

Only the lower bound invariant is ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

Deletions

For vertices incident to updated edges, calculate desire-level (dl): closest level that satisfies invariants.

Iterate from bottommost level to top level and move vertices to desire-level.

Only the lower bound invariant is ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

For vertices incident to updated edges, calculate desire-level (dl): closest level that satisfies invariants.

- Each vertex moves only once, unlike in sequential LDS.

Deletions

Iterate from bottommost level to top level and move vertices to desire-level.

Only the lower bound invariant is ever violated.
We set the coreness estimate of a vertex to be
\[(1 + \delta)^{\max\left(\left\lfloor \frac{\text{level}(v) + 1}{4 \left\lfloor \log_{1+\delta} n \right\rfloor}\right\rfloor - 1, 0}\right)}\]

- Exponent is roughly the group number
- Higher vertices have higher coreness estimates
- This gives a \((2 + \epsilon)\)-approximation
- Getting better than a 2-approximation is P-complete
- Automatically get \((4 + \epsilon)\)-approximation to densest subgraph value
Implementation Details

- Designed an optimized multicore implementation
- Used parallel primitives and data structures from the Graph Based Benchmark Suite [Dhulipala et al. ‘20]
- Maintain concurrent hash tables for each vertex $v$
  - One for storing neighbors on levels $\geq \text{level}(v)$
  - One for storing neighbors on every level $i$ in $[0, \text{level}(v)-1]$
- Moving vertices around in the PLDS requires carefully updating these hash tables for work-efficiency
Complexity Analysis

- $O(\log^2 n)$ levels
  - $O(\log \log n)$ span per level to calculate desire-levels using doubling search
  - $O(\log^* n)$ span with high probability for hash table operations
- Total span: $O(\log^2 n \log \log n)$

- $O(B \log^2 n)$ amortized work is based on potential argument
  - Uses very similar analysis to Bhattacharya, Henzinger, Nanongkai, Tsourakakis [2015]
  - Vertices and edges store potential based on their levels in PLDS, which is used to pay for the cost of moving vertices around
  - We need to map parallel operations to an equivalent set of sequential operations
Experiments
Experimental Setup

- **c2-standard-60 Google Cloud instances**
  - 30 cores with two-way hyper-threading
  - 236 GB memory

- **m1-megamem-96 Google Cloud instances**
  - 48 cores with two-way hyperthreading
  - 1433.6 GB memory

- **3 different types of batches:**
  - All batches of insertions
  - All batches of deletions
  - Mixed batches of both insertions and deletions
Runtimes/Accuracy vs. State-of-the-Art Algorithms

**PLDS:** our algorithm
**PLDSOpt:** optimized PLDS

**Hua et al.:** parallel, exact, dynamic algorithm
**Sun et al.:** sequential, approx., dynamic algorithm

- **PLDSOpt:** 19–544x speedup over Sun et al.
- **PLDSOpt:** 2.5–25x speedup over Hua et al.

![Graphs showing runtimes and accuracy for DBLP and LJ datasets.](image)
Scalability vs. # Hyper-threads

PLDS: our algorithm
PLDSOpt: optimized PLDS
Hua et al.: parallel, exact, dynamic algorithm
Sun et al.: sequential, approx., dynamic algorithm

- Self-relative parallel speedups
  - PLDSOpt: 33x, PLDS: 26x, Hua: 3.6x
- PLDSOpt is faster than all of the other algorithms at 4 or more cores
Runtime vs. Batch Size

PLDS: our algorithm
PLDSOpt: optimized PLDS

Hua et al.: parallel, exact, dynamic algorithm
(Sun et al. does not have a batch method)

- PLDSOpt achieves 2.5-115x speedup over Hua et al.
Runtime vs. Static Algorithms

- Parallel exact $k$-core decomposition [Dhulipala, Blelloch, Shun 2018]
- Parallel $(2 + \epsilon)$-approximate $k$-core decomposition

We achieve speedups for all but the smallest graphs

Speedups of up to 122x for Twitter (1.2B edges) and Friendster (1.8B edges)
Conclusion

• Theoretically-efficient and practical batch-dynamic $k$-core decomposition algorithm

• Using our PLDS, we designed parallel batch-dynamic algorithms for several other problems:
  • Low out-degree orientation
  • Maximal matching
  • Clique counting
  • Graph coloring

• Source code available at
  https://github.com/qqliu/batch-dynamic-kcore-decomposition

Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, Julian Shun, “Parallel Batch-Dynamic Algorithms for k-Core Decomposition and Related Graph Problems,” SPAA 2022
Course Summary

- Congratulations on making it through all the lectures!
- Lots of exciting research going on in algorithm and performance engineering
- Look out for relevant seminars
  - CSAIL seminars mailing list: seminars@csail.mit.edu
- Relevant conferences: SPAA, PPoPP, ACDA, ALENEX, ESA, SEA, PODC, IPDPS, SC, VLDB, SIGMOD, and more