Parallel Batch-Dynamic Algorithms for $k$-Core Decomposition and Related Problems

Quanquan C. Liu
Northwestern University

Jessica Shi
MIT

Shangdi Yu
MIT

Laxman Dhulipala
University of Maryland

Julian Shun
MIT
k-Core
$k$-Core Decomposition

Coreness or Core Number of Node $\nu$: Maximum Core Value of a Core Containing $\nu$
Coreness or Core Number of Node $\nu$:
Maximum Core Value of a Core Containing $\nu$
Approximate $k$-Core Decomposition

Approx. Core Number: 2

Approx. core number of every node: 3

$c$-Approx. Core Number: Value lower bounded by $\frac{\text{core}(v)}{c}$ and upper bounded by $c \times \text{core}(v)$
Approximate $k$-Core Decomposition

$c$-Approx. Core Number: Value lower bounded by $\text{core}(v)/c$ and upper bounded by $c \times \text{core}(v)$
Approximate $k$-Core Decomposition

$c$-Approx. Core Number: Value lower bounded by $\frac{\text{core}(v)}{c}$ and upper bounded by $c \times \text{core}(v)$

Approx. Core Number : 2

Approx. core number of every node: 3

$\frac{3}{2}$-approx

$3/2$-approximations in this paper
Applications of $k$-Core Decomposition

- Graph clustering
- Community detection
- Graph visualizations
- Protein network analysis
- Modeling of disease spread
- Approximating network centrality measures
- Much interest in the machine learning, database, graph analytics, and other communities
Applications of $k$-Core Decomposition

- Graph clustering
- Community detection
- Graph visualizations
- Protein network analysis
- Modeling of disease spread
- Approximating network centrality measures
- Much interest in the machine learning, database, graph analytics, and other communities

**Static, Sequential Setting:** $O(n)$ time

Billions or Even Trillions of Edges

**Too Much Time** to Process Staticaly and Sequentially
# Large Graphs

<table>
<thead>
<tr>
<th>Platform</th>
<th>Edge Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facebook</td>
<td>~ 92.5 million edges</td>
</tr>
<tr>
<td>Friendster</td>
<td>~ 1.8 billion edges</td>
</tr>
<tr>
<td>Twitter</td>
<td>~ 2 billion edges</td>
</tr>
<tr>
<td>Common Crawl</td>
<td>~ 128 billion edges</td>
</tr>
<tr>
<td>Google</td>
<td>~ 6 trillion edges</td>
</tr>
</tbody>
</table>
Large Graphs

- Facebook: ~ 92.5 million edges
- Friendster: ~ 1.8 billion edges
- Twitter: ~ 2 billion edges
- Common Crawl: ~ 128 billion edges
- Google: ~ 6 trillion edges

Graphs are rapidly changing:
- 3M emails/sec
- 486K WhatsApp messages/sec
- 500M tweets/day
- 547K new websites/day
Work-Depth Model

• **Work:**
  - Total number of operations executed by algorithm
  - **Work-efficient:** work asymptotically the same as *best-known* sequential algorithm

• **Depth:**
  - Longest chain of sequential dependencies in algorithm

• **Other Characteristics:**
  - Arbitrary forking
  - Concurrent read, concurrent write to the same shared memory
Batch-Dynamic Model Definition

$G_i$  
Initial $k$-Core Decomposition

$B$ Edge Insertions/Deletions

$G_{i+1}$  
New $k$-Core Decomposition
Batch-Dynamic Graph Algorithms

• Triangle counting [Ediger et al. ‘10, Makkar et al. ’17, Dhulipala et al. ‘20]
• Euler Tour Trees [Tseng et al. ‘19]
• Connected Components [Ferragina and Lucio ‘94, McColl et al. ‘13; Acar et al. ’19, Nowicki and Onak ‘21]
• Rake-Compress Trees [Acar et al. ‘20]
• Incremental Minimum Spanning Trees [Anderson et al. ‘20]
• Minimum Spanning Forest/Graph Clustering [Nowiki and Onak ‘21, Tseng et al. ‘22]
• Graph Connectivity [Dhulipala et al. ‘20]
• Maximal Matching [Nowicki and Onak ‘21]
Why Approximate $k$-Core Decomposition

• Dynamic exact $k$-core decomposition:
  • $\Omega(n)$ work, $\Omega(n)$ depth, parallel [Aridhi et al. ‘16, Gabert et al. ‘21, Hua et al. ‘20, Jin et al. ‘18, Wang ‘17]
  • One update can cause $\Omega(n)$ coreness changes
Why Approximate $k$-Core Decomposition

- Dynamic exact $k$-core decomposition:
  - $\Omega(n)$ work, $\Omega(n)$ depth, parallel [Aridhi et al. ‘16, Gabert et al. ‘21, Hua et al. ‘20, Jin et al. ‘18, Wang ‘17]
  - One update can cause $\Omega(n)$ coreness changes
Why Approximate $k$-Core Decomposition

• Dynamic exact $k$-core decomposition:
  • $\Omega(n)$ work, $\Omega(n)$ depth, parallel [Aridhi et al. ‘16, Gabert et al. ‘21, Hua et al. ‘20, Jin et al. ‘18, Wang ‘17]
  • One update can cause $\Omega(n)$ coreness changes
Why Approximate $k$-Core Decomposition

• Dynamic exact $k$-core decomposition:
  • $\Omega(n)$ work, $\Omega(n)$ depth, parallel [Aridhi et al. ‘16, Gabert et al. ‘21, Hua et al. ‘20, Jin et al. ‘18, Wang ‘17]
  • One update can cause $\Omega(n)$ coreness changes
Why Approximate $k$-Core Decomposition

• Dynamic exact $k$-core decomposition:
  • $\Omega(n)$ work, $\Omega(n)$ depth, parallel [Aridhi et al. ‘16, Gabert et al. ‘21, Hua et al. ‘20, Jin et al. ‘18, Wang ‘17]
  • One update can cause $\Omega(n)$ coreness changes
Why Approximate $k$-Core Decomposition

• Dynamic approximate $k$-core decomposition:
  • $O(\log^2 n)$ time amortized, sequential, $(2 + \varepsilon)$-approximation
    [Sun et al. ‘20]
  • Can accumulate error, charge time to updates
  • Threshold peeling procedure
Why Approximate $k$-Core Decomposition

- **Dynamic approximate $k$-core decomposition:**
  - $O(\log^2 n)$ time amortized, sequential, $(2 + \varepsilon)$-approximation
    - [Sun et al. ‘20]
  - Can accumulate error, **charge time to updates**
  - **Threshold peeling** procedure

Does not use parallelism
**One update** at a time
Why Approximate $k$-Core Decomposition

- Dynamic approximate $k$-core decomposition:
  - $O(\log^2 n)$ time amortized, sequential, $(2 + \varepsilon)$-approximation [Sun et al. ‘20]
  - Can accumulate error, charge time to updates
  - Threshold peeling procedure

Caveat: amortized $O(\log^2 n)$ depth, worst-case $\Omega(n)$ depth

Want: worst-case poly$(\log n)$ depth
Batch Dynamic $k$-Core Decomposition

- $(2 + \epsilon)$-approximation for coreness of every vertex
Batch Dynamic $k$-Core Decomposition

- (2 + $\varepsilon$)-approximation for coreness of every vertex

- $O(B \log^2 n)$ amortized work and $O(\log^2 n \log \log n)$ depth with high probability, size $B$ batch
Batch Dynamic $k$-Core Decomposition

- $(2 + \epsilon)$-approximation for coreness of every vertex

- $O(B \log^2 n)$ amortized work and $O(\log^2 n \log \log n)$ depth with high probability, size $B$ batch

- Is work-efficient, matches Sun et al. ‘20
Batch Dynamic $k$-Core Decomposition

- $(2 + \epsilon)$-approximation for coreness of every vertex

- $O(B \log^2 n)$ amortized work and $O(\log^2 n \log \log n)$ depth with high probability, size $B$ batch

- Is work-efficient, matches Sun et al. ‘20

- Based on a parallel level data structure (PLDS)
Batch Dynamic $k$-Core Decomposition + Others!

- $(2 + \epsilon)$-approximation for coreness of every vertex

- $O(B \log^2 n)$ amortized work and $O(\log^2 n \log \log n)$ depth with high probability, size $B$ batch

- Is work-efficient, matches Sun et al. ’20

- Based on a parallel level data structure (PLDS)

**Static $k$-Core Decomposition**

- Low Out-Degree Orientation
- Maximal Matching
- Clique Counting
- Vertex Coloring
Sequential Level Data Structures for Dynamic Problems

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


- Clustering [Wulff-Nilsen ‘12]

- Low out-degree orientation [Solomon-Wein 20, Henzinger-Neumann-Weiss ‘20]

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

Vertices partitioned into levels

$O(\log^2 n)$

Bhattacharya-Henzinger-Nanongkai-Tsourakakis STOC 2015
Henzinger-Neumann-Weiss 2020
Sequential Level Data Structure (LDS)

Vertices partitioned into levels

\[ O(\log^2 n) \]

Bhattacharya-Henzinger-Nanongkai-Tsourakakis STOC 2015
Henzinger-Neumann-Weiss 2020
Sequential Level Data Structure (LDS)

$O(\log^2 n)$

Vertices partitioned into levels

$O(\log n)$ levels
Cut-off: $(1 + \epsilon)^i$

...
Sequential Level Data Structure (LDS)

$O(\log^2 n)$

Vertices partitioned into levels

- Used for **Vertex Coloring** and **Densest Subgraphs**
- **Not used for k-core decomposition prior to our work**

Bhattacharya-Henzinger-Nanongkai-Tsourakakis STOC 2015
Henzinger-Neumann-Weiss 2020
Sequential Level Data Structure (LDS)

$O(\log^2 n)$

Vertices partitioned into levels

# neighbors: $> 2.1(1 + \epsilon)^i$

= edge insertion

Sequential Level Data Structure (LDS)

Vertices partitioned into levels

$O(\log^2 n)$

# neighbors: $> 2.1(1 + \epsilon)^i$

Sequential Level Data Structure (LDS)

$O(\log^2 n)$

Vertices partitioned into levels

# neighbors: $< (1 + \epsilon)^i$

Sequential Level Data Structure (LDS)

Vertices partitioned into levels

$O(\log^2 n)$

Difficulties with Parallelization

- Large sequential dependencies
- Large depth
Difficulties with Parallelization

- Large sequential dependencies
- Large depth
Difficulties with Parallelization

- Large sequential dependencies
- Large depth
Difficulties with Parallelization

Large sequential dependencies

Large depth
Difficulties with Parallelization

- Large sequential dependencies
- Large depth
Difficulties with Parallelization

- Large sequential dependencies
- Large depth
Difficulties with Parallelization

- Large sequential dependencies
- Large depth
Difficulties with Parallelization

Large sequential dependencies

Large depth
Difficulties with Parallelization

Large sequential dependencies

Large depth
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

Deletions

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

Deletions

Only lower bound cutoff, $(1 + \epsilon)^i$, ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

Deletions

Calculate \textit{desire-level}: closest level that satisfies cutoffs

Only lower bound cutoff, \((1 + \epsilon)^i\), ever violated.

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

Deletions

- Calculate *desire-level*: closest level that satisfies cutoffs
- Iterate from bottommost level to top level and move vertices to desire-level
- Only lower bound cutoff, \((1 + \epsilon)^i\), ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

Deletions

Calculate *desire-level*: closest level that satisfies cutoffs

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

Only lower bound cutoff, \((1 + \epsilon)^i\), ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

Deletions

Calculate *desire-level*: closest level that satisfies cutoffs

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

Only lower bound cutoff, \((1 + \varepsilon)^i\), ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

Deletions

Calculate *desire-level*: closest level that satisfies cutoffs

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

Only lower bound cutoff, $(1 + \varepsilon)^i$, ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

Calculate *desire-level*: closest level that satisfies cutoffs

Deletions

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

Only lower bound cutoff, \((1 + \epsilon)^i\), ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

Deletions

Calculate *desire-level*: closest level that satisfies cutoffs

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

Only lower bound cutoff, \((1 + \varepsilon)^i\), ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

Deletions

Calculate *desire-level*: closest level that satisfies cutoffs

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

Only lower bound cutoff, \((1 + \epsilon)^i\), ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

Deletions

Vertices need to move at most ONCE, unlike sequential LDS!

Calculate desire-level: closest level that satisfies cutoffs

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

Only lower bound cutoff, \((1 + \epsilon)^i\), ever violated.
Our Parallel Batch-Dynamic Level Data Structure (PLDS)

Deletions

- Vertices need to move at most ONCE, unlike sequential LDS!
- Calculate desire-level: closest level that satisfies cutoffs

- $O(\log^2 n \log \log n)$ depth w.h.p

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

- Only lower bound cutoff, $(1 + \epsilon)^i$, ever violated.
Obtaining the Coreness Estimate

- Set the coreness estimate: $(1 + \delta) \max(\lfloor \text{level}(v) + 1 \rfloor / (4 \lfloor \log_{1+\delta} n \rfloor) - 1, 0)$
- Each group has $4 \lfloor \log_{1+\delta} n \rfloor$ levels

$O(\log^2 n)$
Obtaining the Coreness Estimate

- Set the coreness estimate: 
  \[(1 + \delta)^{\max\left(\left\lfloor\frac{\text{level}(v) + 1}{4 \left\lceil \log_{1+\delta} n \right\rceil} \right\rfloor - 1, 0\right)}\]

- Each group has \(4 \left\lceil \log_{1+\delta} n \right\rceil\) levels

- Intuitively, exponent is group number of highest group where node above topmost level

\(O(\log^2 n)\)
Obtaining the Coreness Estimate

- Set the coreness estimate: $(1 + \delta)^{\max\left(\frac{\text{level}(v) + 1}{4 \lceil \log_{1+\delta} n \rceil} - 1, 0\right)}$
- Each group has $4 \lceil \log_{1+\delta} n \rceil$ levels
- Intuitively, exponent is group number of highest group where node above topmost level
Obtaining the Coreness Estimate

- Set the coreness estimate: \((1 + \delta)^{\max(\left\lfloor \text{level}(v) + 1 \right\rfloor / (4 \left\lceil \log_{1+\delta} n \right\rceil) - 1, 0)}\)
- Each group has \(4 \left\lceil \log_{1+\delta} n \right\rceil\) levels
- Intuitively, exponent is group number of highest group where node above topmost level

\(O(\log^2 n)\)

\((1 + \delta)^0 = 1\)
Obtaining the Coreness Estimate

- Set the coreness estimate: $(1 + \delta)^{\max(\lfloor(level(v)+1)/(4\lceil\log_{1+\delta} n\rceil)\rfloor-1,0)}$
- Each group has $4 \lceil\log_{1+\delta} n\rceil$ levels
- Intuitively, exponent is group number of highest group where node above topmost level

Inductive proof that only uses **cutoffs of levels**

Requires number of levels per group $\Omega(\log n)$

$O(\log^2 n)$

$\Omega(\log n)$
Proof of Our Approximation Factor: Upper Bound

Coreness Estimate: $(1 + \epsilon)^i$

Invariant: $\leq 2.1(1 + \epsilon)^i$
Proof of Our Approximation Factor: Upper Bound

Coreness Estimate: $(1 + \epsilon)^i$

Invariant: $\leq 2.1(1 + \epsilon)^i$
Proof of Our Approximation Factor: Upper Bound

Invariant: \( \leq 2.1 \)

Estimate: \((1 + \epsilon)^i\)

\(\text{core}(v) \leq 2.1(1 + \epsilon)^i\)
Proof of Our Approximation Factor: Upper Bound

Estimate: \( (1 + \epsilon)^i \)

Invariant: \( \leq 2.1(1 + \epsilon)^i \)

\[ \text{core}(v) \leq 2.1(1 + \epsilon)^i \]

**Key Proof:** Lower bound proof only requires lower bound invariant and definition of \( k\)-core.
Complexity Analysis

• $O(\log^2 n)$ levels
Complexity Analysis

- $O(\log^2 n)$ levels
- $O(\log \log n)$ depth per level to calculate desire-levels using doubling search
Complexity Analysis

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

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

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

• Total depth: $O(\log^2 n \log \log n)$

• $O(B \log^2 n)$ amortized work is based on potential argument
  • Vertices and edges store potential based on their levels
Experimental Implementation Details

• Designed an optimized multicore implementation
Experimental Implementation Details

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

Graphs from Stanford SNAP database, DIMACS Shortest Paths challenge, and Network Repository—including some temporal

<table>
<thead>
<tr>
<th>Graph</th>
<th>Num. Vertices</th>
<th>Num. Edges</th>
<th>Max k</th>
</tr>
</thead>
<tbody>
<tr>
<td>dblp</td>
<td>425,957</td>
<td>2,099,732</td>
<td>101</td>
</tr>
<tr>
<td>brain-network</td>
<td>784,262</td>
<td>267,844,669</td>
<td>1200</td>
</tr>
<tr>
<td>wikipedia</td>
<td>1,140,149</td>
<td>2,787,967</td>
<td>124</td>
</tr>
<tr>
<td>youtube</td>
<td>1,138,499</td>
<td>5,980,886</td>
<td>51</td>
</tr>
<tr>
<td>stackoverflow</td>
<td>2,601,977</td>
<td>28,183,518</td>
<td>163</td>
</tr>
<tr>
<td>livejournal</td>
<td>4,847,571</td>
<td>85,702,474</td>
<td>329</td>
</tr>
<tr>
<td>orkut</td>
<td>3,072,627</td>
<td>234,370,166</td>
<td>253</td>
</tr>
<tr>
<td>usa-central</td>
<td>14,081,816</td>
<td>16,933,413</td>
<td>2</td>
</tr>
<tr>
<td>usa-road</td>
<td>23,072,627</td>
<td>28,854,312</td>
<td>3</td>
</tr>
<tr>
<td>twitter</td>
<td>41,652,231</td>
<td>1,202,513,046</td>
<td>2484</td>
</tr>
<tr>
<td>friendster</td>
<td>65,608,366</td>
<td>1,806,067,135</td>
<td>304</td>
</tr>
</tbody>
</table>
## Tested Graphs

Graphs from Stanford SNAP database, DIMACS Shortest Paths challenge, and Network Repository—including some temporal graphs.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Num. Vertices</th>
<th>Num. Edges</th>
<th>Max k</th>
</tr>
</thead>
<tbody>
<tr>
<td>dblp</td>
<td>425,957</td>
<td>2,099,732</td>
<td>101</td>
</tr>
<tr>
<td>brain-network</td>
<td>784,262</td>
<td>267,844,669</td>
<td>1200</td>
</tr>
<tr>
<td>wikipedia</td>
<td>1,140,149</td>
<td>2,787,967</td>
<td>124</td>
</tr>
<tr>
<td>youtube</td>
<td>1,138,499</td>
<td>5,980,886</td>
<td>51</td>
</tr>
<tr>
<td>stackoverflow</td>
<td>2,601,977</td>
<td>28,183,518</td>
<td>163</td>
</tr>
<tr>
<td>livejournal</td>
<td>4,847,571</td>
<td>85,702,474</td>
<td>329</td>
</tr>
<tr>
<td>orkut</td>
<td>3,072,627</td>
<td>234,370,166</td>
<td>253</td>
</tr>
<tr>
<td>usa-central</td>
<td>14,081,816</td>
<td>16,933,413</td>
<td>2</td>
</tr>
<tr>
<td>usa-road</td>
<td>23,072,627</td>
<td>28,854,312</td>
<td>3</td>
</tr>
<tr>
<td>twitter</td>
<td>41,652,231</td>
<td>1,202,513,046</td>
<td>2484</td>
</tr>
<tr>
<td>friendster</td>
<td>65,608,366</td>
<td>1,806,067,135</td>
<td>304</td>
</tr>
</tbody>
</table>
Experiments

• 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

• Timeout: 3 hours

• 3 different types of batches:
Experiments

- **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 hyper-threading
  - 1433.6 GB memory

- Timeout: 3 hours

- 3 different types of batches:
  - All Batched Insertions
  - All Batched Deletions
  - Mixed Batches of Both Insertions and Deletions
Experiments

• 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

• Timeout: 3 hours

• 3 different types of batches:
  • All Batched Insertions
  • All Batched Deletions
  • Mixed Batches of Both Insertions and Deletions
Runtimes/Accuracy Against State-of-the-Art Algorithms

**Benchmarks**

- **Sun et al. TKDD**: sequential, approx., dynamic algorithm
- **LDS**: sequential, approx., dynamic LDS of Henzinger et al.
- **Zhang and Yu SIGMOD**: sequential, exact, dynamic algorithm
- **Hua et al. TPDS**: parallel, exact, dynamic algorithm

**Versions of PLDS**

- **PLDS**: exact theoretical algorithm
- **PLDSOpt**: code-optimized PLDS
Runtimes/Accuracy Against State-of-the-Art Algorithms

Benchmarks

- **Sun et al. TKDD**: sequential, approx., dynamic algorithm
- **LDS**: sequential, approx., dynamic LDS of Henzinger et al.
- **Zhang and Yu SIGMOD**: sequential, exact, dynamic algorithm
- **Hua et al. TPDS**: parallel, exact, dynamic algorithm

Versions of PLDS

- **PLDS**: exact theoretical algorithm
- **PLDSSOpt**: code-optimized PLDS

Key Optimization Feature: **Reduce number of levels** per group
Runtimes/Accuracy Against State-of-the-Art Algorithms

- **DBLP**: 425K vertices, 2.1M edges
- **LJ (LiveJournal)**: 4.8M vertices, 85.7M edges
Runtimes/Accuracy Against State-of-the-Art Algorithms

**PLDSOpt**: 19.04–544.22x speedup over Sun

**LJ (LiveJournal)**: 2.49–24.41x speedup over Hua
Number of Hyper-Threading

Faster than all other algorithms at 4 cores!

**PLDSOpt:** 33.02x
self-relative speedup

**PLDS:** 26.46x
self-relative speedup

**Hua:** 3.6x
self-relative speedup
Speedups On a Variety of Graphs

- Speedups against dynamic benchmarks: Hua, Zhang, and Sun

**Graphs ordered by size (left to right)**

Speedups on **all graphs** against **all benchmarks**

Speedups up to: **91.95x** for Hua, **35.59x** for Sun, **723.72x** for Zhang
Speedups On a Variety of Graphs

- Speedups against dynamic benchmarks: Hua, Zhang, and Sun

**Graphs ordered by size (left to right)**

- Speedups on all graphs against all benchmarks
- Speedups up to: 91.95x for Hua, 35.59x for Sun, 723.72x for Zhang
Speedups Against Parallel Static Algorithms

- Parallel exact $k$-core decomposition [Dhulipala et al. ‘18]
- Parallel $(2 + \varepsilon)$-approximate $k$-core decomposition

### Graphs ordered by size (left to right)

- dblp
- youtube
- wiki
- ctr
- usa
- stackoverflow
- livejournal
- orkut
- brain
- twitter
- friendster

**Batch size = $10^6$**

**Speedups over Static Algorithms**

**Speedups over Exact**

**Speedups over Approx**
Speedups Against Parallel Static Algorithms

- Parallel exact $k$-core decomposition [Dhulipala et al. ‘18]
- 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)
## Other Results

<table>
<thead>
<tr>
<th>Problem</th>
<th>Approx</th>
<th>Work</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static $k$-Core</td>
<td>$2 + \varepsilon$</td>
<td>$O(m + n)$</td>
<td>$\tilde{O}(\log^2 n)$</td>
</tr>
<tr>
<td>Low Out-Degree</td>
<td>$4 + \varepsilon$</td>
<td>$O(</td>
<td>B</td>
</tr>
<tr>
<td>Maximal Matching</td>
<td>Maximal</td>
<td>$O(</td>
<td>B</td>
</tr>
<tr>
<td>Clique Counting</td>
<td>Exact</td>
<td>$O(</td>
<td>B</td>
</tr>
<tr>
<td>Explicit Coloring</td>
<td>$O(k \log n)$</td>
<td>$O(</td>
<td>B</td>
</tr>
<tr>
<td>Implicit Coloring</td>
<td>$O(2^k)$</td>
<td>$O(</td>
<td>B</td>
</tr>
</tbody>
</table>
PLDS to Other Results

- $k$-Core Decomposition
- $O(\alpha)$ Out-Degree Orientation
- $O(\alpha \log n)$-Coloring
- Maximal Matching
- $k$-Clique Counting
- Implicit $O(2^\alpha)$-Coloring
## Other Results + Future Work

<table>
<thead>
<tr>
<th>Problem</th>
<th>Approx</th>
<th>Work</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static $k$-Core</td>
<td>$2 + \varepsilon$</td>
<td>$O(m + n)$</td>
<td>$\tilde{O}(\log^2 n)$</td>
</tr>
<tr>
<td>Low Out-Degree</td>
<td>$4 + \varepsilon$</td>
<td>$O(</td>
<td>B</td>
</tr>
<tr>
<td>Maximal Matching</td>
<td>Maximal</td>
<td>$O(</td>
<td>B</td>
</tr>
<tr>
<td>Clique Counting</td>
<td>Exact</td>
<td>$O(</td>
<td>B</td>
</tr>
<tr>
<td>Explicit Coloring</td>
<td>$O(k \log n)$</td>
<td>$O(</td>
<td>B</td>
</tr>
<tr>
<td>Implicit Coloring</td>
<td>$O(2^k)$</td>
<td>$O(</td>
<td>B</td>
</tr>
</tbody>
</table>
Conclusion

- New parallel level data structure (PLDS)

- Parallel batch-dynamic algorithms for $k$-core decomposition and related problems (low out-degree orientation, maximal matching, clique counting, graph coloring)

- Our $k$-core algorithm achieves significant improvements over state-of-the-art solutions in practice

- Source code available at https://github.com/qqliu/batch-dynamic-kcore-decomposition
Extra Slides
Proof of Our Approximation Factor: Lower Bound

Assume for Contradiction:
\[ c(v) < \frac{(1 + \epsilon)^i}{2.1(1 + \epsilon)} \]

Estimate: \((1 + \epsilon)^i\)

Last level of group \(i\)
Proof of Our Approximation Factor: Lower Bound

Assume for Contradiction:
\[ c(v) < \frac{(1 + \epsilon)^i}{2.5} \]

Estimate: \((1 + \epsilon)^i\)

Last level of group \(i\)
Proof of Our Approximation Factor: Lower Bound

Estimate: \((1 + \epsilon)^i\)

Assume for Contradiction:
\[ c(v) < \frac{(1 + \epsilon)^i}{2.5} \]

Last level of group \(i\)

# nodes at or above level below \(v\) is: \(\geq (1 + \epsilon)^i\)
Proof of Our Approximation Factor: Lower Bound

Assume for Contradiction:
\[ c(v) < \frac{(1 + \epsilon)^i}{2.5} \]

# nodes at or above level of \( v \) is: \( \geq (1 + \epsilon)^i \)

Pruning Procedure
Remove all \( w \) where
\[ d_{S_1}(w) < \frac{(1+\epsilon)^i}{2.5} \]
Proof of Our Approximation Factor: Lower Bound

Assume for Contradiction:

\[ c(v) < \frac{(1 + \epsilon)^i}{2.5} \]

At least \( (1 + \epsilon)^i - \frac{(1+\epsilon)^i}{2.5} \) edges must be pruned.

Pruning Procedure
Remove all \( w \) where \( d_{S_1}(w) < \frac{(1+\epsilon)^i}{2.5} \).

\# nodes at or above level of \( v \) is: \( \geq (1 + \epsilon)^i \).
Proof of Our Approximation Factor: Lower Bound

Assume for Contradiction:
\[ c(v) < \frac{(1 + \epsilon)^i}{2} \]

At least \( \frac{(1 + \epsilon)^i}{2} \) edges must be pruned

Pruning Procedure
Remove all \( w \) where
\[ d_{S_1}(w) < \frac{(1 + \epsilon)^i}{2.5} \]

\# nodes at or above level of \( v \) is: \( \geq (1 + \epsilon)^i \)
Proof of Our Approximation Factor: Lower Bound

Assume for Contradiction:

\[ c(v) < \frac{(1 + \epsilon)^i}{2.5} \]

At least \( \left( \frac{1 + \epsilon}{2} \right)^i \) edges must be pruned.

By Induction:

At least \( \left( \frac{1 + \epsilon}{2} \right)^i \) edges must be pruned.

Pruning Procedure:
Remove all \( w \) where

\[ d_{S_j}(w) < \frac{(1 + \epsilon)^i}{2.5} \]

# nodes at or above level of \( v \) is: \( \geq (1 + \epsilon)^i \)
Proof of Our Approximation Factor: Lower Bound

By Induction:

At least \((\frac{1+\epsilon}{2})^j\) edges must be pruned

At least \(\frac{(1+\epsilon)^i}{2} \cdot \frac{1}{(1+\epsilon)^i} \cdot \frac{1}{2.5}\) nodes must be pruned

Assume for Contradiction:

\(c(v) < \frac{(1 + \epsilon)^i}{2.5}\)

Pruning Procedure

Remove all \(w\) where \(d_{S_j}(w) < \frac{(1+\epsilon)^i}{2.5}\)

Number of nodes at or above level of \(v\) is: \(\geq (1 + \epsilon)^i\)
Proof of Our Approximation Factor: Lower Bound

By Induction:

At least \( \left( \frac{1+\epsilon}{2} \right)^j \) edges must be pruned

At least \( \left( \frac{1+\epsilon}{2} \right)^{j-1} \) nodes must be pruned

Assume for Contradiction:

\[ c(v) < \frac{(1 + \epsilon)^i}{2.5} \]

Pruning Procedure

Remove all \( w \) where

\[ d_{S_j}(w) < \frac{(1+\epsilon)^i}{2.5} \]

\# nodes at or above level of \( v \) is: \( \geq (1 + \epsilon)^i \)
Proof of Our Approximation Factor: Lower Bound

Assume for Contradiction:
\[ c(v) < \frac{(1 + \epsilon)^i}{2.5} \]

By Induction:
\[ \text{At least } \left( \frac{(1+\epsilon)^i}{2} \right)^j \text{ edges must be pruned} \]

\[ \left( \frac{(1 + \epsilon)^i}{2} \right)^{j-1} \leq n \]

\[ j \leq \log_{(1+\epsilon)^i/2}(n) \]

Pruning Procedure
Remove all \( w \) where
\[ d_{S_j}(w) < \frac{(1+\epsilon)^i}{2.5} \]

\[ \# \text{ nodes at or above level of } v \text{ is: } \geq (1 + \epsilon)^i \]
Proof of Our Approximation Factor: Lower Bound

By Induction:

At least \( \left(\frac{(1+\epsilon)^i}{2}\right)^j \) edges must be pruned.

\[
\left(\frac{(1+\epsilon)^i}{2}\right)^{j-1} \leq n
\]

\( j \leq \log\left(\frac{(1+\epsilon)^i}{2}\right) (n) \)

Assume for Contradiction:

\( c(v) < \frac{(1+\epsilon)^i}{2.5} \)

Pruning Procedure
Remove all \( w \) where

\[
d_{S_j(w)} < \frac{(1+\epsilon)^i}{2.5}
\]

# nodes at or above level of \( v \) is: \( \geq (1 + \epsilon)^i \)

Run out of vertices before first level of the group.
Proof of Our Approximation Factor: Lower Bound

By Induction:

At least \( \left( \frac{(1+\epsilon)^i}{2} \right)^j \) edges must be pruned.

\[
\left( \frac{1 + \epsilon}{2} \right)^{j-1} \leq n
\]

\( j \leq \log_{(1+\epsilon)/2} (n) \)

Must be the case that:

\[
c(v) \geq \frac{(1 + \epsilon)^i}{2.5}
\]

Pruning Procedure

Remove all \( w \) where \( d_{S_j}(w) < \frac{(1+\epsilon)^i}{2.5} \)

# nodes at or above level of \( v \) is: \( \geq (1 + \epsilon)^i \)

Run out of vertices before first level of the group.