Theoretically Efficient Parallel Graph Algorithms Can Be Fast And Scalable

Julian Shun (MIT)

Joint work with Laxman Dhulipala and Guy Blelloch (CMU) SPAA 2017 and SPAA 2018
Graphs are becoming very large

<table>
<thead>
<tr>
<th></th>
<th>Asymmetric</th>
<th>Symmetrized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter</td>
<td>41 million vertices</td>
<td>41 million vertices</td>
</tr>
<tr>
<td></td>
<td>1.5 billion edges (6.2 GB)</td>
<td>2.4 billion edges (9.8 GB)</td>
</tr>
<tr>
<td>Yahoo!</td>
<td>1.4 billion vertices</td>
<td>1.4 billion vertices</td>
</tr>
<tr>
<td></td>
<td>6.6 billion edges (38 GB)</td>
<td>12.9 billion edges (63 GB)</td>
</tr>
<tr>
<td>Common Crawl</td>
<td>3.5 billion vertices</td>
<td>3.5 billion vertices</td>
</tr>
<tr>
<td></td>
<td>128 billion edges (540 GB)</td>
<td>225 billion edges (928 GB)</td>
</tr>
</tbody>
</table>

- Need efficient graph processing to do analytics in a timely fashion
Large-Scale Graph Processing

• Write algorithms for large distributed clusters or supercomputer
• Prior results on Common Crawl graph (225B edges):

<table>
<thead>
<tr>
<th>Distributed Algorithms</th>
<th>Hardware</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approx. k-core (Slota et al.)</td>
<td>256 x 32 cores, 16.3TB RAM</td>
<td>363 sec</td>
</tr>
<tr>
<td>Largest Conn. Comp. (Slota et al.)</td>
<td>256 x 32 cores, 16.3TB RAM</td>
<td>63 sec</td>
</tr>
<tr>
<td>Conn. Comp. (Stergiou et al.)</td>
<td>1000 x 24 cores, 128TB RAM</td>
<td>341 sec</td>
</tr>
</tbody>
</table>

• Write algorithms for limited-memory machine that stream graphs from SSDs (TurboGraph, Mosaic, BigSparse)
  • Usually (up to an order of magnitude) slower but much more cost-efficient

What about in-memory computation on a single machine with 1TB RAM?
Multicore Results

- Results on Common Crawl graph (3.5B vertices, 225B edges)
Theoretically-Efficient Practical Algorithms

- Want good performance under many different settings, e.g., different machines and larger datasets

**Work** = number of operations

**Depth** = length of longest sequential dependence

**Running time** \( \leq \frac{\text{Work}}{\#\text{processors}} + \text{Depth} \)

- Goal: Minimize depth without increasing work over best sequential algorithm (work-efficient)
Theoretically-Efficient Practical Algorithms

Hyperlink2012-Host (|V|=102M, |E|=3.9B) on 72 cores

Theoretically-efficient graph algorithms can be fast
### Contributions

- Theoretically-efficient parallel graph algorithms that are practical

<table>
<thead>
<tr>
<th>Breadth-first search</th>
<th>Weighted BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Betweenness centrality</td>
<td>Single-source shortest paths</td>
</tr>
<tr>
<td>Connected components</td>
<td>Low-diameter decomposition</td>
</tr>
<tr>
<td>Biconnected components</td>
<td>Strongly connected components</td>
</tr>
<tr>
<td>Triangle counting</td>
<td>Minimum spanning tree</td>
</tr>
<tr>
<td>k-core decomposition</td>
<td>Maximal matching</td>
</tr>
<tr>
<td>Maximal independent set</td>
<td>Graph coloring</td>
</tr>
<tr>
<td>Approximate set cover</td>
<td></td>
</tr>
</tbody>
</table>

- Extended Ligra framework to support bucketing algorithms
- Theoretically-efficient optimizations
- Experimental evaluation on the largest publicly-available real-world graphs, outperforming existing results
Ligra: Frontier-Based Algorithms

Primitives

- Frontier data-structure (VertexSubset)
- Map over vertices in a frontier (VertexMap)
- Map over out-edges of a frontier (EdgeMap)

Example: Breadth-First Search

Round 1
Round 2
Round 3
Round 4

Some useful graph algorithms cannot be efficiently implemented in frontier-based frameworks
Example: Weighted Breadth-First Search

Given: $G = (V, E, w)$ with positive integer edge weights, $s \subseteq V$

Problem: Compute the shortest path distances from $s$

Frontier-based approach: On each step, update distances of neighbors, place neighbors whose distance decreased onto next frontier
Example: Weighted Breadth-First Search

Frontier: s

Round 1
Example: Weighted Breadth-First Search

Frontier: s

Round 1
Example: Weighted Breadth-First Search

Frontier: 1 4

Round 2
Example: Weighted Breadth-First Search

Frontier: 1, 4

Round 2
Example: Weighted Breadth-First Search

Frontier: 2 3 4 5 6

Round 3
Example: Weighted Breadth-First Search

Takes $O(VE)$ work, which is not work-efficient!
• Sequential algorithm runs in \( O(D + |E|) \) work
• Run Dijkstra’s algorithm, but use buckets instead of a priority queue
• Represent buckets using dynamic arrays
Sequential Weighted BFS

Round 1

0 1 2 3 4 5
Sequential Weighted BFS

Round 1

s

0 1 2 3 4 5
Sequential Weighted BFS

Round 1
Sequential Weighted BFS

Round 1

0 1 2 3 4 5
Sequential Weighted BFS

Round 2

0 1 2 3 4 5
Sequential Weighted BFS

Round 2
Sequential Weighted BFS

Round 2

0 1 2 3 4 5

1 4 2

2 3 1

1 2

0 1 2 3 4 5

1 4 2

2 3 1

1 2

0 1 2 3 4 5

1 4 2

2 3 1

1 2

0 1 2 3 4 5

1 4 2

2 3 1

1 2

0 1 2 3 4 5

1 4 2

2 3 1

1 2

0 1 2 3 4 5

1 4 2

2 3 1

1 2

0 1 2 3 4 5

1 4 2

2 3 1

1 2

0 1 2 3 4 5

1 4 2

2 3 1

1 2

0 1 2 3 4 5

1 4 2

2 3 1

1 2

0 1 2 3 4 5

1 4 2

2 3 1

1 2
Sequential Weighted BFS

Round 2
Sequential Weighted BFS

Round 3
Sequential Weighted BFS

Round 3

0 1 2 3 4 5
Sequential Weighted BFS

$O(D + |E|)$ work where $D$ is the graph diameter.
**Bucketing**

The algorithm uses buckets to organize work for future iterations.

This algorithm is actually parallelizable:

- In each step:
  1. Process all vertices in the next non-empty bucket in parallel
  2. Update buckets of neighbors in parallel
Sequential Weighted BFS

Sequential: process vertices one by one

Round 3

0 1 2 3 4 5
Parallel Weighted BFS

(1) Process vertices in the same bucket in parallel

Round 3

0 1 2 3 4 5
Parallel Weighted BFS

Resulting algorithm performs:

\[ O(D + |E|) \] work
\[ O(D \log |V|) \] depth

(assuming efficient bucketing)

(2) Insert neighbors into buckets in parallel
Parallel Bucketing

Bucketing is useful for more than just weighted BFS

- k-core (coreness)
- Delta Stepping for Single-Source Shortest Paths
- Parallel Approximate Set Cover

Goals

- Simplify expressing algorithms by using an interface
- Theoretically efficient, reusable implementation

Challenges

1. Multiple vertices insert into the same bucket in parallel
2. Possible to make work-efficient parallel implementations?
Julienne Framework

Bucketing Interface:

1. **MakeBuckets**: Create bucket structure
2. **NextBucket**: Return the next non-empty bucket (as a VertexSubset)
3. **UpdateBuckets**: Update buckets of a subset of vertices
MakeBuckets

\[ D(1) = 0, D(2) = 1, D(3) = 4, \ldots \]

Initialize bucket structure
MakeBuckets

$D(1) = 0, D(2) = 1, D(3) = 4, \ldots$

Initialize bucket structure
NextBucket

Extract vertices in the next non-empty bucket
NextBucket

Extract vertices in the next non-empty bucket
NextBucket

Extract vertices in the next non-empty bucket
Move vertices to new buckets

Input: array of (vertex, destination bucket) pairs
UpdateBuckets

Move vertices to new buckets

\[ [(1,3), (7,2), (6,2)] \]
UpdateBuckets

Move vertices to new buckets

\[[(1,3), (7,2), (6,2)]\]
Sequential Bucketing

Can implement sequential bucketing with:

- $n$ vertices
- $T$ total buckets
- $K$ calls to `UpdateBuckets`, where each updates the vertices in $S_i$

$$\text{in } O(n + T + \sum_{i=0}^{K} |S_i|) \text{ work}$$

Implementation:

- Use dynamic arrays
- Update lazily
  - When deleting, leave vertex in bucket
  - When encountering a vertex, check if it has already been processed
Parallel Bucketing

Can implement parallel bucketing with:

- n vertices
- T total buckets
- K calls to UpdateBuckets, where each updates the vertices in $S_i$
- L calls to NextBucket

\[ O(n + T + \sum_{i=0}^{K} |S_i|) \] expected work and

\[ O((K + L) \log n) \] depth with high probability

Implementation:

- Use dynamic arrays, delete lazily
- NextBucket: filter out already processed vertices (uses parallel prefix sum, which takes linear work and logarithmic depth)
**Parallel Bucketing**

UpdateBuckets:

- Use work-efficient semisort [Gu et al. 2015]
- Given $k$ (key, value) pairs, semisorts in $O(k)$ expected work and $O(\log k)$ depth with high probability

$([(3,9), (2,1), \ldots, (4,7), (1,1)]$

\[\downarrow\]

$[(3,9), \ldots,(2,1), (1,1), (7,1), \ldots,(4,7), (6,7), \ldots]$

*All vertices going to bucket 1*

- Compute num. vertices going to each bucket (parallel prefix sum)
- Resize buckets and copy over all vertices in parallel
Example: k-core and Coreness

k-core: maximal connected subgraph of G s.t. all vertices have degree $\geq k$

$\lambda(v)$: largest k-core that $v$ participates in

Can efficiently compute k-cores after computing coreness
Sequential Peeling

Sequential Peeling:

- Bucket sort vertices by degree
- Remove the minimum degree vertex, set its core number
  - Update the buckets of its neighbors

Each vertex and edge is processed exactly once:

\[ W = O(|E| + |V|) \]
Parallel Peeling

Existing parallel algorithms:

- Remove all vertices with minimum degree from graph and set their core numbers

Existing parallel algorithms will scan all remaining vertices on each round to find the ones with minimum degree

\[
W = O(|E| + \rho|V|) \\
D = O(\rho \log |V|)
\]

\(\rho = \text{number of peeling steps done by the parallel algorithm}\)

Not work-efficient!
Work-Efficient Peeling

Insert vertices in bucket structure by degree
While not all vertices have been processed yet:

1. Extract the next non-empty bucket, set core numbers
Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next non-empty bucket, set core numbers
Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next non-empty bucket, set core numbers
Insert vertices in bucket structure by degree
While not all vertices have been processed yet:
1. Extract the next non-empty bucket, set core numbers
2. Sum edges removed from each neighbor of this frontier
Insert vertices in bucket structure by degree

While not all vertices have been processed yet:
1. Extract the next non-empty bucket, set core numbers
2. Sum edges removed from each neighbor of this frontier
3. Compute the new buckets for the neighbors
4. Update the bucket structure with the (neighbor id, dest bucket)
Work-Efficient Peeling Analysis

We process each edge at most once in each direction:

- \# updates = \(O(|E|)\)
- \# buckets \leq 59K
- \# calls to NextBucket = \(O(|E|)\)
- \# calls to UpdateBuckets = \(O(|E|)\)

Therefore the algorithm runs in:

\[ \text{expected work} \times \text{depth} \quad \text{with high probability} \]

On the Common Crawl graph (225M edges), \# buckets = 59K

On 72 cores, our code finishes in a few minutes, but the work-inefficient algorithm does not terminate within 3 hours.

Efficient peeling using Julienne
Experiments: k-core

- 2-9x faster than work-inefficient implementation
- Between 4-41x speedup on 72 cores over sequential peeling
- Speedups are smaller on small graphs with large $\rho$

Friendster
($|V| = 121M, |E| = 3.6B, \rho = 10K$)

Hyperlink2012-Host
($|V| = 102M, |E| = 3.9B, \rho = 19K$)
Single-Source Shortest Paths

- 1.8-5.2x faster than (work-inefficient) Bellman-Ford
- Competitive with hand-optimized Single-Source Shortest Paths implementations
- On 72 cores, 18-32x self-relative speedup, 17-30x speedup over DIMACS solver
More Graph Algorithms

• Theoretically-efficient implementations of over a dozen other graph algorithms
• Compression was crucial in running on 1TB machine
  • Compressed edge lists using delta encoding and variable-length codes
• Theoretically-efficient parallel primitives on compressed edge lists
  • Map, Map-Reduce, Filter, Pack, Intersect
Scaling to Largest Graph

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-core</td>
<td>193 sec</td>
</tr>
<tr>
<td>Weighted BFS</td>
<td>58 sec</td>
</tr>
<tr>
<td>Biconnected components</td>
<td>201 sec</td>
</tr>
<tr>
<td>Strongly connected components</td>
<td>182 sec</td>
</tr>
<tr>
<td>Minimum spanning forest</td>
<td>228 sec</td>
</tr>
<tr>
<td>Maximal independent set</td>
<td>34 sec</td>
</tr>
<tr>
<td>Maximal matching</td>
<td>126 sec</td>
</tr>
<tr>
<td>Triangle counting</td>
<td>1470 sec</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-first search</td>
<td>12 sec</td>
</tr>
<tr>
<td>Connected components</td>
<td>38 sec</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>53 sec</td>
</tr>
<tr>
<td>Betweenness centrality (1 source)</td>
<td>40 sec</td>
</tr>
<tr>
<td>Low-diameter decomposition</td>
<td>18 sec</td>
</tr>
<tr>
<td>Graph coloring</td>
<td>174 sec</td>
</tr>
<tr>
<td>Approximate set cover</td>
<td>104 sec</td>
</tr>
<tr>
<td>PageRank (1 iteration)</td>
<td>28 sec</td>
</tr>
</tbody>
</table>

- Outperforms reported numbers for this graph
- For many algorithms, no published results for this graph
Conclusion

- Theoretically-efficient parallel algorithms can be fast and scalable
- Can process largest graphs on a single multicore server with 1TB of RAM

- Julienne framework available at https://github.com/jshun/ligra
- All of our theoretically-efficient graph algorithms are available at https://github.com/ldhulipala/gbbs