Julienne and the Graph-Based Benchmark Suite

Laxman Dhulipala
MIT (Postdoc)
https://ldhulipala.github.io/

Based on joint work with Guy Blelloch and Julian Shun
Graph Processing: algorithms and systems that enable us to analyze and understand graphs

- Input Graph
- Graph Processing
- Output

- Static
- Dynamic

Algorithms
- Connectivity
- Graph Clustering
- Distance Computations
- Dense Subgraphs

- Understanding
- Visualizations
- Graph-based features
- System-optimization
Large-Scale Graph Processing

WebDataCommons hyperlink graph

- 3.5 billion vertices and 128 billion edges
- ~1TB of memory to store
- Largest publicly available graph

“…[the 2012 graph is the] largest hyperlink graph that is available to the public outside companies such as Google, Yahoo, and Microsoft.”

Year of sourcing vs total number of vertices and edges for real-world graphs from the SNAP and LAW datasets

Large-Scale Graph Processing

WebDataCommons hyperlink graph

- 3.5 billion vertices and 128 billion edges
- ~1TB of memory to store
- Largest publicly available graph

“...[the 2012 graph is the] largest hyperlink graph that is available to the public outside companies such as Google, Yahoo, and Microsoft.”

Parallelism is the key to processing very large graphs in a timely manner

Year of sourcing vs total number of vertices and edges for real-world graphs from the SNAP and LAW datasets

Shared-Memory Parallelism

Shared-Memory Machines

• Cost for a 1TB memory machine with 72 processors is about $20,000.

• Can rent a similar machine (96 processors and 1.5TB memory) for $11/hour on Google Cloud

WebDataCommons Graph

• 3.5 billion vertices and 128 billion edges

A single shared-memory machine can already store the largest publicly available graph datasets, with plenty of room to spare
**Work-Depth Model**

**Work** = total number of vertices in the computation graph

**Depth** = longest directed path in the graph (dependence length)

**Running Time** = \( \frac{\text{Work}}{\# \text{Processors}} + O(\text{Depth}) \)

A *work-efficient* parallel algorithm has work that asymptotically matches that of the best sequential algorithm for the problem

**Goal:** work-efficient and low (polylogarithmic) depth algorithms
Theoretical Efficiency

A parallel algorithm is *theoretically-efficient* if it has good bounds on its work and depth.

Why do we care about theoretical bounds?

Input-agnostic design

- Design codes without worrying too much about your datasets

Robustness to bad inputs

- Perform well even on new classes of graphs
- Understand how they will scale on larger graphs

Work-efficiency matters in practice

- Work-efficient algorithms can be much faster than work-inefficient algorithms

Up to 9x faster using a work-efficient k-core algorithm (described in this talk)
How do we design theoretically-efficient parallel graph algorithms for a certain class of bucketing-based problems
Frontier-Based Algorithms in Ligra

Primitives

- Frontier data-structure (vertexSubset)
- Map over vertices in a frontier (vertexMap)
- Map over out-edges of a frontier to generate new frontier (edgeMap)

Example: Breadth-First Search

- Round 1
- Round 2
- Round 3
- Round 4

- : in frontier
- : unvisited
- : visited
Weighted Breadth-First Search

Given: $G = (V, E, w)$ with positive integer edge weights, and a source $s$

Problem: Compute the shortest path distances from $s$

Frontier-based approach: on each step, visit all neighbors that had their distance decrease
Weighted Breadth-First Search

Given: \( G = (V, E, w) \) with positive integer edge weights, and a source \( s \)

Problem: Compute the shortest path distances from \( s \)

Frontier-based approach: on each step, visit all neighbors that had their distance decrease

Round 1

Frontier: \( s \)

Distances: \( s: 0 \)
Weighted Breadth-First Search

Given: $G = (V, E, w)$ with positive integer edge weights, and a source $s$

Problem: Compute the shortest path distances from $s$

Frontier-based approach: on each step, visit all neighbors that had their distance decrease.
Weighted Breadth-First Search

Given: $G = (V, E, w)$ with positive integer edge weights, and a source $s$

Problem: Compute the shortest path distances from $s$
Weighted Breadth-First Search

Given: $G = (V, E, w)$ with positive integer edge weights, and a source $s$

Problem: Compute the shortest path distances from $s$
Weighted Breadth-First Search

Given: $G = (V, E, w)$ with positive integer edge weights, and a source $s$

Problem: Compute the shortest path distances from $s$

Round 3

Frontier: 4 2 5 6 3

Distances: $s$: 0
           1: 1
           4: 2
           6: 5
Weighted Breadth-First Search

Given: \( G = (V, E, w) \) with positive integer edge weights, and a source \( s \)

Problem: Compute the shortest path distances from \( s \)

Round 3

Frontier: 4 2 5 6 3

Distances: \( s: 0 \), 1: 1, 4: 2, 6: 5

Not work-efficient!
Sequential Weighted Breadth-First Search

Given: $G = (V, E, w)$ with positive integer edge weights, and a source $s$

Problem: Compute the shortest path distances from $s$

Idea:
- Run Dijkstra’s algorithm, but use buckets instead of a PQ
- Represent buckets using dynamic arrays
- Runs in $O(m + r_{src})$ work
Sequential Weighted Breadth-First Search

Round 1

s

Sequential Weighted Breadth-First Search

Round 1

s

1 1 1 2 3

3 1 1 3

2 5

5

6

7

0 1 2 3 4 5
Sequential Weighted Breadth-First Search

Round 1

0 1 2 3 4 5
Sequential Weighted Breadth-First Search

Round 1

0 1 2 3 4 5
Sequential Weighted Breadth-First Search

Round 2
Sequential Weighted Breadth-First Search

Round 2

0 1 2 3 4 5
Sequential Weighted Breadth-First Search

Round 2
Sequential Weighted Breadth-First Search

Round 2
Sequential Weighted Breadth-First Search

Runs in $O(m + r_{src})$ work
Bucketing

The algorithm uses buckets to *organize work* for future iterations

This algorithm is parallelizable

- In each step:
  1. Process all vertices in the next bucket in parallel
  2. Update buckets of neighbors in parallel
Sequential Weighted Breadth-First Search

Sequential: process vertices one by one

Round 3

0 1 2 3 4 5
Parallel Weighted Breadth-First Search

Round 3

(1) Process vertices in the same bucket in parallel
Parallel Weighted Breadth-First Search

Resulting algorithm performs:

\[ O(m + r_{src}) \text{ work} \]

\[ O(r_{src} \log n) \text{ depth} \]

(assuming efficient bucketing)

(2) Insert neighbors into buckets in parallel
Parallel Bucketing

Bucketing is useful for more than just wBFS

Parallel Approximate Set Cover [BPT’12]

Parallel Shortest Paths [MB’03]

Parallel k-Tip Decomposition [SS’20]
Parallel Bucketing

Bucketing is useful for more than just wBFS

Goals

• Simplify expressing algorithms 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: Results

Shared memory framework for *bucketing-based algorithms*

Extend Ligra with an interface for bucketing
- Theoretical bounds for primitives
- Fast implementations of primitives

Can implement a bucketing algorithm with
- $n$ vertices
- $T$ total buckets
- $U$ updates

over $K$ Update calls, and $L$ calls to NextBucket

\[ O(n + T + U) \] expected work and
\[ O((K + L) \log n) \] depth w.h.p.

Bucketing implementation is work-efficient
Bucketing Interface:

(1) Create bucket structure
(2) Get the next bucket (vertexSubset)
(3) Update buckets of a subset of identifiers
MakeBuckets : buckets
\[ n : \text{int} \]
\[ D : \text{identifier} \rightarrow \text{bucket\_id} \]
\[ O : \text{bucket\_order} \]

Initialize bucket structure
Bucketing Interface

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

MakeBuckets : buckets
\[
\begin{align*}
  n & \text{: int} \\
  D & \text{: identifier → bucket_id} \\
  O & \text{: bucket_order}
\end{align*}
\]

Initialize bucket structure
Bucketing Interface

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

MakeBuckets : buckets

$n : \text{int}$

$D : \text{identifier} \rightarrow \text{bucket_id}$

$O : \text{bucket_order}$

Initialize bucket structure
Extractor: Bucketing Interface

NextBucket : bucket

Extract identifiers in the next non-empty bucket
Bucketing Interface

Order: increasing

NextBucket : bucket

Extract identifiers in the next non-empty bucket
Bucketing Interface

Order: increasing

NextBucket : bucket

Extract identifiers in the next non-empty bucket
Bucketing Interface

Update buckets for k identifiers

UpdateBuckets

$k : \text{int}$

$F : \text{int} \rightarrow (\text{identifier, bucket\_dest})$
Bucketing Interface

Update buckets for \( k \) identifiers

\[ F : \text{int} \rightarrow (\text{identifier, bucket}_\text{dest}) \]

Update buckets for \( k \) identifiers
Bucketing Interface

Update buckets for k identifiers

UpdateBuckets

\[ k : \text{int} \]

\[ F : \text{int} \rightarrow (\text{identifier}, \text{bucket}_{\text{dest}}) \]

\[ [(1,1), (7,2), (6,2)] \]
Bucketing Interface

[(1,1), (7,2), (6,2)]

UpdateBuckets

\[ k : \text{int} \]
\[ F : \text{int} \rightarrow (\text{identifier}, \text{bucket\_dest}) \]

Update buckets for k identifiers
Bucketing Interface

[(1,1), (7,2), (6,2)]

UpdateBuckets

\( k : \text{int} \)
\( F : \text{int} \rightarrow (\text{identifier, bucket}_\text{dest}) \)

Update buckets for \( k \) identifiers
Sequential Bucketing

Can implement sequential bucketing with:

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

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

Idea:

- Use dynamic arrays that are updated lazily
Parallel Bucketing

Can implement parallel bucketing with:

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

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

\[
O((K + L) \log n) \text{ depth w.h.p.}
\]

Idea:

- Use dynamic arrays
- MakeBuckets: call UpdateBuckets. NextBucket: parallel filter
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 w.h.p.

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

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

All ids going to bucket 1

• Prefix sum to compute ids going to each bucket
• Resize buckets and inject all ids in parallel

Can see paper for details on practical implementation and optimizations
k-Core Decomposition

**k-core**: maximal connected subgraph of G where all vertices have degree at least k *within the subgraph*

**coreness**: largest k-core that a given vertex participates in

*Widely used in network analysis tasks such as unsupervised clustering of social and biological networks*
The Peeling Algorithm

- Current degree of remaining vertices decreases as vertices are *peeled* from the graph

- Once a vertex’s current degree is less than or equal to the current core number, it gets peeled
The Peeling Algorithm

$k = 1$

$k = 2$

$k = 3$

$k = 1$

$k = 2$

$k = 3$
The Peeling Algorithm

Classic sequential implementations of this peeling algorithm run in $O(m)$ time [AM’84, BZ’03]
Parallel Peeling

Remove all vertices with degree less than or equal to the current core number in parallel
Insert vertices in bucket structure by degree
Insert vertices in bucket structure by degree
Insert vertices in bucket structure by degree

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

While not all vertices have been processed yet:

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

While not all vertices have been processed yet:

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

While not all vertices have been processed yet:
  1. Extract the next bucket, set core numbers
Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next 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 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 bucket, set core numbers
2. Sum edges removed from each neighbor of this frontier
3. Compute the new buckets for the neighbors
Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next bucket, set core numbers
2. Sum edges removed from each neighbor of this frontier
3. Compute the new buckets for the neighbors
Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next 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 (neighbors, buckets)
Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next 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 (neighbors, buckets)
Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next 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 (neighbors, buckets)
Insert vertices in bucket structure by degree

While not all vertices have been processed yet:

1. Extract the next 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 (neighbors, buckets)
We process each edge at most once in each direction:

# updates = \( O(|E|) \)

# buckets \( \leq |V| \)

# calls to NextBucket = \( \rho \)

# calls to UpdateBuckets = \( \rho \)

Therefore the algorithm runs in:

\[ O(|E| + |V|) \] expected work

\[ O(\rho \log |V|) \] depth w.h.p.

On the largest graph we test on, \( \rho = 130,728 \)

On 72 cores, our code finishes in a few minutes, but the work-inefficient algorithm does not terminate within 3 hours
We process each edge at most once in each direction:

\[
\text{# updates} = O(|E|)
\]

\[
\text{# buckets} \leq |V|
\]

\[
\text{# calls to NextBucket} \leq \text{# calls to UpdateBuckets}
\]

Therefore the algorithm runs in:

\[O(|E| + |V|)\text{ depth w.h.p.}\]

On the largest graph we test on, \(= 130728\),

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
A Work-Efficient k-core Decomposition Algorithm

Julienne Algorithm in GBBS

- Actual code is under 50 lines of C++
- Parallel cost:
  \[ O(m + n) \] expected work
  \[ O(\rho \log n) \] depth whp

where \( \rho \) is the number of peeling rounds

Our algorithm is the first work-efficient algorithm for k-core decomposition with non-trivial parallelism
## Work and Depth of Algorithms in Julienne

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Work</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-core</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>wBFS</td>
<td>$O(D +</td>
<td>E</td>
</tr>
<tr>
<td>Delta-stepping</td>
<td>$O(w_\Delta)$</td>
<td>$O(d_\Delta \log</td>
</tr>
<tr>
<td>Approx Set Cover</td>
<td>$O(M)$</td>
<td>$O(\log^3 M)$</td>
</tr>
</tbody>
</table>

- $\rho$ : number of rounds of parallel peeling
- $D$ : diameter
- $w_\Delta, d_\Delta$ : work and number of rounds of the delta-stepping algorithm
- $M$ : sum of sizes of sets

[2] Blelloch, Peng, Tangwongsan: Linear-work greedy parallel approximate set cover and variants
Experimental Results

Across all inputs:

- Between 4-41x speedup over sequential peeling
- Speedups are smaller on small graphs with large \( \rho \)
- 2-9x faster than work-inefficient implementation
Experimental Results: Hyperlink Graphs

Hyperlink graphs extracted from Common Crawl Corpus

| Graph  | |V|   | |E|   | |E|(symmetrized) |
|--------|--------|--------|--------|
| HL2014 | 1.7B   | 64B    | 124B   |
| HL2012 | 3.5B   | 128B   | 225B   |

- Previous analyses use supercomputers [1] or external memory [2]
- HL2012-Sym requires ~1TB of memory uncompressed

[1] Slota et al., 2015, Supercomputing for Web Graph Analytics
[2] Zheng et al., 2015, FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs
Experimental Results: Hyperlink Graphs

<table>
<thead>
<tr>
<th>Graph</th>
<th>k-core</th>
<th>wBFS</th>
<th>Set Cover</th>
</tr>
</thead>
<tbody>
<tr>
<td>HL2014</td>
<td>97.2</td>
<td>9.02</td>
<td>45.1</td>
</tr>
<tr>
<td>HL2012</td>
<td>206</td>
<td>—</td>
<td>104</td>
</tr>
</tbody>
</table>

Running time in seconds on 72 cores with hyperthreading

- Able to process in main-memory of 1TB machine by compressing
- 23-43x speedup across applications
## k-Core Decomposition on the WebDataCommons Graph

- **k-core**: maximal connected subgraph of $G$ s.t. all vertices have degree at least $k$

<table>
<thead>
<tr>
<th>BlueWaters [SRM’16]</th>
<th>GBBS [DBS’18]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
<td>363 seconds</td>
</tr>
<tr>
<td><strong>Processors</strong></td>
<td>8192</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>16 TB</td>
</tr>
<tr>
<td><strong>Quality</strong></td>
<td>Approximate</td>
</tr>
<tr>
<td><strong>Cost</strong></td>
<td>Very Expensive</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>184 seconds</td>
</tr>
<tr>
<td><strong>Processors</strong></td>
<td>72</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>1 TB</td>
</tr>
<tr>
<td><strong>Quality</strong></td>
<td>Exact</td>
</tr>
<tr>
<td><strong>Cost</strong></td>
<td>Highly Affordable</td>
</tr>
</tbody>
</table>

1.95x faster than the approximate distributed result by SRM’16, using 56.8x fewer hyper-threads and 16.3x less memory
Summary: Julienne

Julienne: framework for *bucketing-based algorithms*

- Codes:
  - Simple (< 100 lines each)
  - Theoretically efficient (strong bounds on work and depth)
  - Good performance in practice
  - Code included as part of the GBBS library
Theoretically-Efficient Parallel Graph Algorithms can be Fast and Scalable [DBS’18]

Can we solve a broad set of fundamental graph problems on the largest graphs, affordably and quickly?
The Graph-Based Benchmark Suite (GBBS)

- Introduce a benchmark suite for graph problems with over 20 important problems
- GBBS algorithms achieve state-of-the-art results on the largest publicly available graphs

**Connectivity Problems**
- Low-Diameter Decomposition
- Connectivity
- Spanning Forest
- Biconnectivity
- Minimum Spanning Forest
- Strongly Connected Components

**Subgraph Problems**
- k-Core Decomposition
- k-Truss Decomposition
- Apx. Densest Subgraph
- Triangle Counting
- Higher-Clique Counting

**Covering Problems**
- Maximal Ind. Set
- Maximal Matching
- Apx. Set Cover
- Graph Coloring

**Shortest Path Problems**
- Breadth-First Search
- Betweenness Centrality
- Bellman-Ford
- General Weight SSSP
- Integral Weight SSSP
- SS Widest Path
- k-Spanner

**Eigenvector Problems**
- PageRank
- Personalized PageRank
- Personalized SimRank

[github.com/paralg/gbbs](https://github.com/paralg/gbbs)
Benchmarking Connectivity on WebDataCommons Graph

Benchmarks are based on I/O specifications, e.g.,

Maximal Independent Set
Input: $G(V, E)$ an undirected graph
Output: $U \subseteq V$, a set of vertices such that no two vertices in $U$ are neighbors, and all vertices in $V \setminus U$ have a neighbor in $U$

k-core (Coreness)
Input: $G(V, E)$ an undirected graph
Output: A mapping from each vertex to its coreness value (the maximum $k$ such that the vertex is in a non-empty $k$-core)

I/O specification makes it easy to compare different algorithm implementations
A broad set of fundamental graph problems can be solved on a graph with over 200 billion edges in 3 minutes.
Work and Depth of GBBS Results

<table>
<thead>
<tr>
<th>Problem</th>
<th>Work</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-First Search (BFS)</td>
<td>$O(m)$</td>
<td>$\tilde{O}(\text{diam}(G))$</td>
</tr>
<tr>
<td>Integral-Weight SSSP (weighted BFS)</td>
<td>$O(m)^\dagger$</td>
<td>$\tilde{O}(\text{diam}(G))^*$</td>
</tr>
<tr>
<td>General-Weight SSSP (Bellman-Ford)</td>
<td>$O(\text{diam}(G) \cdot m)$</td>
<td>$\tilde{O}(\text{diam}(G))$</td>
</tr>
<tr>
<td>Single-Source Widest Path (Bellman-Ford)</td>
<td>$O(\text{diam}(G) \cdot m)$</td>
<td>$\tilde{O}(\text{diam}(G))$</td>
</tr>
<tr>
<td>Single-Source Betweenness Centrality (BC)</td>
<td>$O(m)$</td>
<td>$\tilde{O}(\text{diam}(G))$</td>
</tr>
<tr>
<td>O(k)-Spanner</td>
<td>$O(m)$</td>
<td>$\tilde{O}(k \log n)^*$</td>
</tr>
<tr>
<td>Low-Diameter Decomposition (LDD)</td>
<td>$O(m)$</td>
<td>$O(\log^2 n)^*$</td>
</tr>
<tr>
<td>Connectivity (CC)</td>
<td>$O(m)^\dagger$</td>
<td>$O(\log^3 n)^*$</td>
</tr>
<tr>
<td>Spanning Forest</td>
<td>$O(m)^\dagger$</td>
<td>$O(\log^3 n)^*$</td>
</tr>
<tr>
<td>Biconnectivity</td>
<td>$O(m)^\dagger$</td>
<td>$O(\max(\text{CC}, \text{BFS}))$</td>
</tr>
<tr>
<td>Strongly Connected Components (SCC)</td>
<td>$O(m \log n)^\ddagger$</td>
<td>$\tilde{O}(\text{diam}(G))^*$</td>
</tr>
<tr>
<td>Minimum Spanning Forest (MSF)</td>
<td>$O(m \log n)$</td>
<td>$O(\log^2 n)$</td>
</tr>
<tr>
<td>Maximal Independent Set (MIS)</td>
<td>$O(m)^\dagger$</td>
<td>$O(\log^2 n)^*$</td>
</tr>
<tr>
<td>Maximal Matching (MM)</td>
<td>$O(m)^\dagger$</td>
<td>$O(\log^2 n)^*$</td>
</tr>
</tbody>
</table>

**Main Challenge:**

How do we build simple and provably-efficient implementations of these algorithms that work on the largest real-world graphs?
GBBS Library

- High-level graph processing interface in the lineage of Ligra [SB’12]
- Provides many useful primitives
- Compressed graph representations based on extending Ligra+

**Vertex Operations**
- Map
- Reduce
- Filter
- Pack
- Intersect

**Graph Operations**
- Filter
- Pack
- Contract

**Core GBBS Interfaces**

- **Bucketing**
- **Graph**
- **Vertex**
- **ParlayLib**
- **Parallel Runtime**

**Graph Representations**

- Cilk, OpenMP, TBB, Homegrown

**Compression Library**

**Table**

| Graph                      | |V|   | |E|   | Size (CSR) | Compressed | Bytes/edge |
|----------------------------|----------------|------|----------------|------------|------------|
| WDC Hyperlink              | 3.5B           | 128B | 1080GB         | 446GB      | 1.74       |
| WDC Hyperlink (Sym)        | 3.5B           | 225B | 928 GB         | 351GB      | 1.56       |
## Vertex Interface

<table>
<thead>
<tr>
<th>Neighborhood operators:</th>
<th>map ((\text{edge} \rightarrow \text{void}) \rightarrow \text{void})</th>
<th>reduce ((\text{edge} \rightarrow \text{E}) \ast \text{E monoid} \rightarrow \text{E})</th>
<th>scan ((\text{edge} \rightarrow \text{E}) \ast \text{E monoid} \rightarrow \text{E})</th>
<th>count ((\text{edge} \rightarrow \text{bool}) \rightarrow \text{int})</th>
<th>filter ((\text{edge} \rightarrow \text{bool}) \rightarrow \text{E seq})</th>
<th>pack ((\text{edge} \rightarrow \text{bool}) \rightarrow \text{void})</th>
<th>iterate ((\text{edge} \rightarrow \text{bool}) \rightarrow \text{void})</th>
<th>i-th (\text{int} \rightarrow \text{edge})</th>
<th>degree (\text{unit} \rightarrow \text{int})</th>
<th>getNeighbors (\text{unit} \rightarrow \text{nghlist})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(</td>
<td>N(v)</td>
<td>)) (O(\log n))</td>
<td>(O(\text{d}<em>{\text{it}})) (O(\text{d}</em>{\text{it}}))</td>
<td>(O(1)) (O(1))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vertex-Vertex operators:</th>
<th>intersection ((\text{nghlist} \ast \text{nghlist}) \rightarrow \text{int})</th>
<th>union ((\text{nghlist} \ast \text{nghlist}) \rightarrow \text{int})</th>
<th>difference ((\text{nghlist} \ast \text{nghlist}) \rightarrow \text{int})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(l \log (h/l + 1)))</td>
<td>(O(\log n))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Provides **functional primitives** for commonly used vertex operations with good theoretical bounds on their cost.
Provides functional primitives for performing whole-graph operations, and for operations that consume and produce vertexSubsets.
Consider \((u, v) \in E\) s.t. \(u \in U\) and \(C(v)\)

If \(F(u, v) = \text{True}\) return \(v\) in output, \(O\)

Operator specification doesn’t insist on a particular implementation. Thus, Ligra (and GBBS) can implement direction-optimization “under the hood”
Generalizing edgeMap to Other Graph Operations

**Inputs**

- vertexSubset $U$
- Map function $F : \text{edge} \rightarrow O$
- Combine function $M : O \text{ monoid } (O \rightarrow O \rightarrow O, \text{identity})$
- Condition function $C : \text{vtxid} \rightarrow \text{bool}$

**Output**

- $O \text{ vertexSubset } R$

Aggregating results at the source vertices yields a **src**- version

Aggregating results at the neighbor vertices yields a **ngh**- version
Example: Updating Induced Degrees in Parallel using nghCount

$k = 1$

Our Implementation

- We provide a provably-efficient implementation of nghCount that takes

$$O\left( |U| + \sum_{u \in U} d(u) \right)$$

expected work

$$O(\log n)$$ depth whp
Connectivity Problems in GBBS

- Connectivity and related problems are probably the best studied problems in the parallel algorithms literature.
- Practical work-efficient implementations of these problems are absent in the experimental literature.

GBBS provides simple and high-level implementations of connectivity problems based on low-diameter decomposition.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Work</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-First Search (BFS)</td>
<td>$O(m)$</td>
<td>$\tilde{O}(\text{diam}(G))$</td>
</tr>
<tr>
<td>Integral-Weight SSSP (weighted BFS)</td>
<td>$O(m)$</td>
<td>$\tilde{O}(\text{diam}(G))$</td>
</tr>
<tr>
<td>General-Weight SSSP (Bellman-Ford)</td>
<td>$O(\text{diam}(G) \cdot m)$</td>
<td>$\tilde{O}(\text{diam}(G))$</td>
</tr>
<tr>
<td>Single-Source Widest Path (Bellman-Ford)</td>
<td>$O(\text{diam}(G) \cdot m)$</td>
<td>$\tilde{O}(\text{diam}(G))$</td>
</tr>
<tr>
<td>Single-Source Betweenness Centrality (BC)</td>
<td>$O(m)$</td>
<td>$\tilde{O}(\text{diam}(G))$</td>
</tr>
<tr>
<td>O(k)-Spanner</td>
<td>$O(m)$</td>
<td>$\tilde{O}(k \log n)$</td>
</tr>
<tr>
<td>Low-Diameter Decomposition (LDD)</td>
<td>$O(m)$</td>
<td>$O(\log^2 n)$</td>
</tr>
<tr>
<td>Connectivity (CC)</td>
<td>$O(m)$</td>
<td>$O(\log^3 n)$</td>
</tr>
<tr>
<td>Spanning Forest</td>
<td>$O(m)$</td>
<td>$O(\log^3 n)$</td>
</tr>
<tr>
<td>Biconnectivity</td>
<td>$O(m)$</td>
<td>$O(\max(\text{CC}, \text{BFS}))$</td>
</tr>
<tr>
<td>Strongly Connected Components (SCC)</td>
<td>$O(m \log n)$</td>
<td>$\tilde{O}(\text{diam}(G))$</td>
</tr>
<tr>
<td>Minimum Spanning Forest (MSF)</td>
<td>$O(m \log n)$</td>
<td>$O(\log^2 n)$</td>
</tr>
<tr>
<td>Maximal Independent Set (MIS)</td>
<td>$O(m)$</td>
<td>$O(\log^2 n)$</td>
</tr>
<tr>
<td>Maximal Matching (MM)</td>
<td>$O(m)$</td>
<td>$O(\log^2 n)$</td>
</tr>
<tr>
<td>Graph Coloring</td>
<td>$O(m)$</td>
<td>$O(\log n + L \log \Delta)$</td>
</tr>
<tr>
<td>k-core</td>
<td>$O(m)$</td>
<td>$O(\rho \log n)$</td>
</tr>
<tr>
<td>Approximate Set Cover</td>
<td>$O(m)$</td>
<td>$O(\log^3 n)$</td>
</tr>
<tr>
<td>Triangle Counting (TC)</td>
<td>$O(m^{3/2})$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Approximate Densest Subgraph</td>
<td>$O(m)$</td>
<td>$O(\log^2 n)$</td>
</tr>
<tr>
<td>PageRank Iteration</td>
<td>$O(n + m)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

\[ \dagger: \text{in expectation} \quad *: \text{whp} \]
Connectivity Problems in GBBS using LDD

Low-Diameter Decomposition [MPX’13]
- $O(m + n)$ work
- $O(\log^2 n)$ depth whp
- Max diameter is $O(\log n/\beta)$
- Total cut edges $\leq \beta m$

Spanning Forest [SDB’14]
- $O(m + n)$ expected work
- $O(\log^3 n)$ depth whp

Undirected Connectivity [SDB’14]
- $O(m + n)$ expected work
- $O(\log^3 n)$ depth whp

Biconnectivity [SV’87]
- $O(m + n)$ expected work
- $O(\text{diam}(G) + \log^3 n)$ depth whp

Graph Contraction

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>graph $G(V, E)$</td>
<td>graph $G'(V', E')$</td>
</tr>
<tr>
<td>Cluster labels $F : \text{vtxid} \rightarrow \text{int}$</td>
<td></td>
</tr>
</tbody>
</table>

89
“Hard” Problems in GBBS

- Work-efficient, polylog depth algorithms not known for these problems
- Instead, focus on work-efficiency at the expense of parametrizing depth in terms of some other graph parameter (usually diameter)

**Transitive Closure Bottleneck:** See book chapter by Karp and Ramachandran

<table>
<thead>
<tr>
<th>Problem</th>
<th>Work</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-First Search (BFS)</td>
<td>(O(m))</td>
<td>(\tilde{O}(\text{diam}(G)))</td>
</tr>
<tr>
<td>Integral-Weight SSSP (weighted BFS)</td>
<td>(O(m))†</td>
<td>(\tilde{O}(\text{diam}(G)))*</td>
</tr>
<tr>
<td>General-Weight SSSP (Bellman-Ford)</td>
<td>(O(\text{diam}(G) \cdot m))</td>
<td>(\tilde{O}(\text{diam}(G)))</td>
</tr>
<tr>
<td>Single-Source Widest Path (Bellman-Ford)</td>
<td>(O(\text{diam}(G) \cdot m))</td>
<td>(\tilde{O}(\text{diam}(G)))</td>
</tr>
<tr>
<td>Single-Source Betweenness Centrality (BC)</td>
<td>(O(m))</td>
<td>(\tilde{O}(\text{diam}(G)))</td>
</tr>
<tr>
<td>(O(k))-Spanner</td>
<td>(O(m))</td>
<td>(\tilde{O}(k \log n))*</td>
</tr>
<tr>
<td>Low-Diameter Decomposition (LDD)</td>
<td>(O(m))†</td>
<td>(O(\log^2 n))*</td>
</tr>
<tr>
<td>Connectivity (CC)</td>
<td>(O(m))†</td>
<td>(O(\log^3 n))*</td>
</tr>
<tr>
<td>Spanning Forest</td>
<td>(O(m))†</td>
<td>(O(\log^3 n))*</td>
</tr>
<tr>
<td>Biconnectivity</td>
<td>(O(m))†</td>
<td>(O(\max(\text{CC}, \text{BFS})))</td>
</tr>
<tr>
<td>Strongly Connected Components (SCC)</td>
<td>(O(m \log n))†</td>
<td>(\tilde{O}(\text{diam}(G)))*</td>
</tr>
<tr>
<td>Minimum Spanning Forest (MSF)</td>
<td>(O(m \log n))</td>
<td>(O(\log^2 n))</td>
</tr>
<tr>
<td>Maximal Independent Set (MIS)</td>
<td>(O(m))†</td>
<td>(O(\log^2 n))*</td>
</tr>
<tr>
<td>Maximal Matching (MM)</td>
<td>(O(m))†</td>
<td>(O(\log^2 n))*</td>
</tr>
<tr>
<td>Graph Coloring</td>
<td>(O(m))</td>
<td>(O(\log n + L \log \Delta))</td>
</tr>
<tr>
<td>(k)-core</td>
<td>(O(m))†</td>
<td>(O(\rho \log n))*</td>
</tr>
<tr>
<td>Approximate Set Cover</td>
<td>(O(m))†</td>
<td>(O(\log^3 n))*</td>
</tr>
<tr>
<td>Triangle Counting (TC)</td>
<td>(O(m^{3/2}))</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>Approximate Densest Subgraph</td>
<td>(O(m))</td>
<td>(O(\log^2 n))</td>
</tr>
<tr>
<td>PageRank Iteration</td>
<td>(O(n + m))</td>
<td>(O(\log n))</td>
</tr>
</tbody>
</table>

\(\dagger\): in expectation  \(*\): whp
Case Study: Connectivity on WebDataCommons Graph

Outperform external memory results by orders of magnitude using comparable hardware.

Outperform distributed memory results using orders of magnitude less hardware.
Recent Results that use GBBS

**SAGE**

Semi-Asymmetric Graph Engine

Design extensions of GBBS algorithms to a semi-asymmetric setting for NVRAM machines, and achieve state-of-the-art running times (VLDB’20)

with Charles McGuffey, Hongbo Kang, Yan Gu, Guy Blelloch, Phil Gibbons, and Julian Shun

**ConnectIt**

Framework for parallel connectivity, spanning forest, and incremental connectivity (VLDB’21)

with Changwan Hong and Julian Shun

**ArbClique**

Implement state-of-the-art k-clique counting (exact+approximate), and k-clique densest-subgraph algorithms in GBBS (in submission)

with Jessica Shi and Julian Shun

Lots of other ongoing work!

**Efficient parallel graph algorithms for motifs (cycles, cliques)**

with Changwan Hong and Julian Shun

**Shared-memory parallel graph embedding**

**Parallel Graph Clustering (SCAN, Hierarchical Agglomerative Clustering)**

**Parallel Batch-Dynamic k-Core Decomposition, Triangle Counting**

with Jessica Shi and Julian Shun
Summary: Julienne

Julienne: framework for *bucketing-based algorithms*

- Codes:
  - Simple (< 100 lines each)
  - Theoretically efficient (strong bounds on work and depth)
  - Good performance in practice
  - Code included as part of the GBBS library
Summary: The Graph-Based Benchmark Suite (GBBS)

- Introduced a benchmark suite for graph problems with over 20 important problems
- Introduced GBBS library, which extends Ligra, Ligra+, and Julienne to simplify expressing theoretically-efficient parallel graph algorithms
- GBBS implementations achieve state-of-the-art results on the largest publicly available graphs

**Connectivity Problems**
- Low-Diameter Decomposition
- Connectivity
- Spanning Forest
- Biconnectivity
- Minimum Spanning Forest
- Strongly Connected Components

**Subgraph Problems**
- k-Core Decomposition
- k-Truss Decomposition
- Apx. Densest Subgraph
- Triangle Counting
- Higher-Clique Counting

**Covering Problems**
- Maximal Ind. Set
- Maximal Matching
- Apx. Set Cover
- Graph Coloring

**Shortest Path Problems**
- Breadth-First Search
- Betweenness Centrality
- Bellman-Ford
- General Weight SSSP
- Integral Weight SSSP
- SS Widest Path
- k-Spanner

**Eigenvector Problems**
- PageRank
- Personalized PageRank
- Personalized SimRank

[github.com/paralg/gbbs](https://github.com/paralg/gbbs)