Parallelism Background and Fast Parallel Spatial Clustering Algorithms

Laxman Dhulipala
University of Maryland, College Park
cs.umd.edu/~laxman
My Research
My Research

High-performance systems and algorithms that provide theoretical guarantees and are easy to use
My Research

High-performance systems and algorithms that provide theoretical guarantees and are easy to use
High-performance systems and algorithms that provide theoretical guarantees and are easy to use

My Research

Practice

Provably-Efficient and Scalable Systems

Parallel Algorithm Design and Analysis

New Models for Parallel Computation

Theory
My Research

High-performance systems and algorithms that provide theoretical guarantees and are easy to use

Practice

Provably-Efficient and Scalable Systems

Parallel Algorithm Design and Analysis

New Models for Parallel Computation

Theory
Large-Scale Graph Processing

WebDataCommons hyperlink graph

• 3.5 billion vertices and 128 billion edges
• ~1TB of memory to store
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.”
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

Sources:
- https://snap.stanford.edu/data/
- http://law.di.unimi.it/datasets.php
- http://webdatacommons.org/hyperlinkgraph/
Parallelism

Parallel machines are everywhere!
Parallelism

*Parallel machines are everywhere!*
Parallelism

Parallel machines are everywhere!
Parallelism

*Parallel machines are everywhere!*
Parallelism

Parallel machines are everywhere!
Parallelism

*Parallel machines are everywhere!*

Main focus of my work is *shared-memory parallelism*
Shared-Memory Parallelism
Shared-Memory Parallelism

Shared-Memory Machines

- Cost for a 1 TB memory machine with 72 processors is about $20,000.
Shared-Memory Parallelism

Shared-Memory Machines

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

Computation Graph
Work-Depth Model

\[ \text{Work} = \text{total number of vertices in the computation graph} \]
Work-Depth Model

Computation Graph

Work = total number of vertices in the computation graph

Depth = longest directed path in the graph (dependence length)
Work-Depth Model

Work = total number of vertices in the computation graph

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

Running Time = Work/#Processors + \( O(\text{Depth}) \)
**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.
Work-Depth Model

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

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

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

**Running Time** = 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.
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?*
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
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
Theoretical Efficiency

A parallel algorithm is \textit{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
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
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)
<table>
<thead>
<tr>
<th>Graph Systems: examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pregel</td>
</tr>
<tr>
<td>PowerGraph</td>
</tr>
<tr>
<td>PowerLyra</td>
</tr>
<tr>
<td>Parallel BGL</td>
</tr>
<tr>
<td>GraphLab</td>
</tr>
<tr>
<td>Green-Marl</td>
</tr>
<tr>
<td>GraphMat</td>
</tr>
<tr>
<td>Ringo</td>
</tr>
<tr>
<td>SNAP</td>
</tr>
<tr>
<td>GraphIt</td>
</tr>
<tr>
<td>Ligra</td>
</tr>
<tr>
<td>Julienne</td>
</tr>
<tr>
<td>GBBS</td>
</tr>
<tr>
<td>STAPL</td>
</tr>
<tr>
<td>GraphX (Spark)</td>
</tr>
<tr>
<td>ASPIRE</td>
</tr>
<tr>
<td>GoFFish</td>
</tr>
<tr>
<td>Presto</td>
</tr>
<tr>
<td>GraphChi</td>
</tr>
<tr>
<td>Blogel</td>
</tr>
<tr>
<td>GraM</td>
</tr>
<tr>
<td>Giraph</td>
</tr>
<tr>
<td>PAGE</td>
</tr>
<tr>
<td>MOCgraph</td>
</tr>
<tr>
<td>GraphChi</td>
</tr>
<tr>
<td>GridGraph</td>
</tr>
<tr>
<td>NXgraph</td>
</tr>
<tr>
<td>Chaos</td>
</tr>
<tr>
<td>FlashGraph</td>
</tr>
<tr>
<td>Graphine</td>
</tr>
<tr>
<td>Sage</td>
</tr>
<tr>
<td>Graphite</td>
</tr>
<tr>
<td>GraFBoost</td>
</tr>
<tr>
<td>X-Stream</td>
</tr>
<tr>
<td>TurboGraph</td>
</tr>
<tr>
<td>TurboGraph++</td>
</tr>
<tr>
<td>Ligra+</td>
</tr>
<tr>
<td>MMap</td>
</tr>
<tr>
<td>PathGraph</td>
</tr>
<tr>
<td>GridGraph</td>
</tr>
<tr>
<td>NXgraph</td>
</tr>
<tr>
<td>Chaos</td>
</tr>
<tr>
<td>FlashGraph</td>
</tr>
<tr>
<td>Graphene</td>
</tr>
<tr>
<td>Sage</td>
</tr>
<tr>
<td>Graphite</td>
</tr>
<tr>
<td>GraFBoost</td>
</tr>
<tr>
<td>X-Stream</td>
</tr>
<tr>
<td>TurboGraph</td>
</tr>
<tr>
<td>TurboGraph++</td>
</tr>
<tr>
<td>Ligra+</td>
</tr>
<tr>
<td>MMap</td>
</tr>
<tr>
<td>PathGraph</td>
</tr>
<tr>
<td>GridGraph</td>
</tr>
<tr>
<td>NXgraph</td>
</tr>
<tr>
<td>Chaos</td>
</tr>
<tr>
<td>FlashGraph</td>
</tr>
<tr>
<td>Graphene</td>
</tr>
<tr>
<td>Sage</td>
</tr>
<tr>
<td>Graphite</td>
</tr>
<tr>
<td>GraFBoost</td>
</tr>
<tr>
<td>X-Stream</td>
</tr>
<tr>
<td>TurboGraph</td>
</tr>
<tr>
<td>TurboGraph++</td>
</tr>
<tr>
<td>Ligra+</td>
</tr>
<tr>
<td>MMap</td>
</tr>
<tr>
<td>PathGraph</td>
</tr>
<tr>
<td>GridGraph</td>
</tr>
<tr>
<td>NXgraph</td>
</tr>
<tr>
<td>Chaos</td>
</tr>
<tr>
<td>FlashGraph</td>
</tr>
<tr>
<td>Graphene</td>
</tr>
<tr>
<td>Sage</td>
</tr>
<tr>
<td>Graphite</td>
</tr>
<tr>
<td>GraFBoost</td>
</tr>
<tr>
<td>X-Stream</td>
</tr>
<tr>
<td>TurboGraph</td>
</tr>
<tr>
<td>TurboGraph++</td>
</tr>
<tr>
<td>Ligra+</td>
</tr>
<tr>
<td>MMap</td>
</tr>
<tr>
<td>PathGraph</td>
</tr>
<tr>
<td>GridGraph</td>
</tr>
<tr>
<td>NXgraph</td>
</tr>
<tr>
<td>Chaos</td>
</tr>
<tr>
<td>FlashGraph</td>
</tr>
<tr>
<td>Graphene</td>
</tr>
<tr>
<td>Graph Systems: examples</td>
</tr>
<tr>
<td>-------------------------</td>
</tr>
<tr>
<td>Pregel</td>
</tr>
<tr>
<td>PowerGraph</td>
</tr>
<tr>
<td>PowerLyra</td>
</tr>
<tr>
<td>Parallel PC</td>
</tr>
<tr>
<td>GraphLab</td>
</tr>
<tr>
<td>Green-Marl</td>
</tr>
<tr>
<td>GraphMat</td>
</tr>
<tr>
<td>Ring</td>
</tr>
<tr>
<td>SNAP</td>
</tr>
<tr>
<td>GraphIt</td>
</tr>
<tr>
<td>Ligra</td>
</tr>
<tr>
<td>Julienne</td>
</tr>
<tr>
<td>GBBS</td>
</tr>
<tr>
<td>STAPL</td>
</tr>
</tbody>
</table>

Unfortunately existing graph systems typically study a very small set of simple problems, such as BFS.

Can we solve a broad set of static graph problems on very large graphs?
Theoretically-Efficient Parallel Graph Algorithms can be Fast and Scalable [D, Blelloch, Shun, SPAA’18 Best Paper]

- Introduce the Graph-Based Benchmark Suite (GBBS) 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]
Benchmarking Connectivity on WebDataCommons Graph

![Graph 1: Memory Used vs. Running Time](image1)

- **GraFBoost**
- **Mosaic**
- **FlashGraph**
- **Gluon**
- **FastSV**

- **External Memory**
- **Distributed Memory**
- **Shared Memory**

![Graph 2: Number of Processors vs. Running Time](image2)

- **GraFBoost**
- **Mosaic**
- **FlashGraph**
- **Gluon**
- **FastSV**

- **External Memory**
- **Distributed Memory**
- **Shared Memory**
Outperform external memory results by orders of magnitude using comparable hardware.
Benchmarking 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.
GBBS can analyze $O(100B)$ edge graphs in seconds to minutes.
GBBS can analyze O(100B) edge graphs in seconds to minutes

A broad set of fundamental graph problems can be solved on a graph with over 200 billion edges in 3 minutes
<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>
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>$O(m)$</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>$O(\max(\text{CC, BFS}))$</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>Approximate Densest Subgraph</td>
<td>$O(m)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>PageRank Iteration</td>
<td>$O(n + m)$</td>
<td>$O(\log n)$</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]
GBBS Library

- High-level graph processing interface in the lineage of Ligra [SB’12]

- Provides many useful primitives

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

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

Diagram:
- Core GBBS Interfaces
  - Bucketing
  - Graph
  - Vertex
  - ParlayLib
  - Parallel Runtime
  - Graph Representations
  - Compression Library

Cilk, OpenMP, TBB, Homegrown
GBBS Library

- High-level graph processing interface in the lineage of Ligra [SB’12]

- Provides many useful primitives

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

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

- Compressed graph representations

### Core GBBS Interfaces

- Bucketing
- Graph
- Vertex
- ParlayLib
- Parallel Runtime
- Graph Representations
- Compression Library

#### Cilk, OpenMP, TBB, Homegrown

### Graph

| 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 |
k-Core Decomposition
k-Core Decomposition

k-core: maximal connected subgraph of G where all vertices have degree at least k \textit{within the subgraph}
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
k-Core Decomposition

k-core: maximal connected subgraph of G where all vertices have degree at least k \textit{within the subgraph}

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

$k = 1$
The Peeling Algorithm

$k = 1$

$k = 1$
The Peeling Algorithm

- $k = 1$
- $k = 1$
- $k = 1$
The Peeling Algorithm

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

- 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.

All vertices “below threshold” can be peeled in parallel.
Our contribution is to give a general interface for bucketing.
A Work-Efficient k-core Decomposition Algorithm

GBBS Algorithm

- Actual code in GBBS 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

---

**Algorithm 1** k-core (Coreness)

1: Coreness[0, \ldots, n] := 0
2: procedure Coreness(G(V, E))
3: \( F = \text{maxBuckets}(V, \text{Coreness}, \text{increasing}) \) \( \triangleright \) initialized to initial degrees
4: \( \text{Finished} := 0 \)
5: while \( \text{Finished} < |V| \) do
6:   \( \text{Finished} := \text{Finished} + |\text{ids}| \)
7:   \( \text{cond} \triangleright \text{fn} v \rightarrow \text{return true} \)
8:   \( \text{induced}D := D[v] \) \( \triangleright \) current core number, and vertices peeled this step
9:   if \( \text{induced}D > k \) then
10:      \( \text{new}D := \max(\text{induced}D - \text{edges} - \text{edges} \text{removed}, k) \)
11:      Coreness[\text{inds}] := \text{new}D
12:      \( \text{bkt} := \text{Bucket}(\text{induced}D, \text{new}D) \)
13:      if \( \text{bkt} \neq \text{null} \) then
14:         \( \text{return Some(bkt)} \)
15:      \text{Moved} := \text{moveCount}(G, \text{ids}, \text{cond} \triangleright \text{fn}) \) \( \triangleright \) Moved is an bkktdest vertexSubset
16:      \( \text{UpdateBuckets(Moved)} \) \( \triangleright \) update the buckets of vertices in Moved
17:      \text{return Coreness}
A Work-Efficient k-core Decomposition Algorithm

GBBS Algorithm

- Actual code in GBBS 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
### k-Core Decomposition on the WebDataCommons Graph

**BlueWaters [SRM’16]**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>363 seconds</td>
</tr>
<tr>
<td>Processors</td>
<td>8192</td>
</tr>
<tr>
<td>Memory</td>
<td>16 TB</td>
</tr>
<tr>
<td>Quality</td>
<td>Approximate</td>
</tr>
<tr>
<td>Cost</td>
<td>Very Expensive</td>
</tr>
</tbody>
</table>

**k-core** : maximal connected subgraph of G s.t. all vertices have degree at least k
### k-Core Decomposition on the WebDataCommons Graph

**k-core** is a maximal connected subgraph of $G$ such that all vertices have a degree of at least $k$.

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

**BlueWaters**
- **Time**: 363 seconds
- **Processors**: 8192
- **Memory**: 16 TB
- **Quality**: Approximate
- **Cost**: Very Expensive

**GBBS**
- **Time**: 184 seconds
- **Processors**: 72
- **Memory**: 1 TB
- **Quality**: Exact
- **Cost**: Highly Affordable
k-Core Decomposition on the WebDataCommons Graph

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

BlueWaters [SRM’16] vs. GBBS [DBS’18]

<table>
<thead>
<tr>
<th></th>
<th>BlueWaters [SRM’16]</th>
<th>GBBS [DBS’18]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>363 seconds</td>
<td>184 seconds</td>
</tr>
<tr>
<td>Processors</td>
<td>8192</td>
<td>72</td>
</tr>
<tr>
<td>Memory</td>
<td>16 TB</td>
<td>1 TB</td>
</tr>
<tr>
<td>Quality</td>
<td>Approximate</td>
<td>Exact</td>
</tr>
<tr>
<td>Cost</td>
<td>Very Expensive</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
GBBS as a Research Repository 5 Years On

Basis for many other parallel graph projects:

- Fast Parallel Graph Connectivity [DHS’21]
- Parallel k-clique enumeration [SDS’21]
- Graph Embedding [QDTPW’21]
- Structural Graph Clustering [TDS’21]
- Batch-Dynamic Graph Orientation [LSYDS’22]
GBBS as a Research Repository 5 Years On

Basis for many other parallel graph projects:

❖ Fast Parallel Graph Connectivity [DHS’21]
❖ Parallel k-clique enumeration [SDS’21]
❖ Graph Embedding [QDTPW’21]
❖ Structural Graph Clustering [TDS’21]
❖ Batch-Dynamic Graph Orientation [LSYDS’22]

Used at Google:

❖ Fast and scalable implementations of parallel graph clustering algorithms (e.g., Affinity Clustering)
❖ Being used to develop and evaluate parallel hierarchical agglomerative clustering (HAC) algorithms
Faster k-Means to Accelerate ANNS
Clustering

❖ Given a set of points $P$ with a notion of distance between the points, group the points into a number of clusters so that:
  ❖ Members of the same cluster are close / similar to each other
  ❖ Members of different clusters are dissimilar

Usually:
❖ Points are in a high-dimensional space, e.g., $P \in \mathbb{R}^d, d \geq 100$
❖ Distance is measured using Euclidean distance, but other measures also possible (e.g., Jaccard, edit-distance, etc)
Clustering
Clustering Problem: Building Bucketing-Based Indexes

- Exact retrieval requires exhaustive scan in the worst case; settle for approximation instead.
- Measure recall@k: the fraction of output candidates in true top k neighbors
Clustering Problem: Building Bucketing-Based Indexes

❖ **Build:**
  ❖ Assign points to one (or more) buckets
  ❖ Nearby points likely to be in the same buckets

❖ **Query:**
  ❖ Probe a subset of buckets for the queried point
  ❖ Compare with all points in these buckets and report top-k
Main ideas:

- Build graphs with polylog(n) degree
- Satisfy the “relative neighbor” property (RNGs):
- Points $p, q$ connected by an edge if there does not exist a third point $r$ that is closer to both $p, q$ than they are to each other
k-Means clustering partitions the data into $k$ convex clusters.

Idea: run k-means with reasonably large $k$ (e.g., on an $n = 1e^9$ point dataset, we might use $k = 1e^6$).
Such a large value of $k$ creates an interesting routing problem—given a query $q$, which buckets (clusters) should we probe?

Idea: just build another ANN index over the centroids. In this case, a graph index (e.g., HNSW or DiskANN)

In practice, we will figure out the $k'$ closest centroids to the query and probe the clusters for these centroids.
k-Means for Product Quantization

- Vectors in modern applications are large
  - Recent OpenAI text embeddings have ~1600 dimensions. Used to be 8 times larger until recently
  - More dimensions useful in applications, but costly to store and search

- PQ: main idea
  - $D \rightarrow D^*$ dimensions
  - Reduce range of each dimension
    - i.e., use uint8 instead of float
k-Means for Product Quantization

- PQ: main idea
  - $D \rightarrow D^*$ dimensions
  - Reduce range of each dimension
    - i.e., use uint8 instead of float

- Range reduction works by using the id of a centroid (say one of $2^8 = 256$ centroids)
- Original point can be approximated by remembering the position of the centroid
k-Means for Product Quantization

❖ PQ: main idea
❖ $D \rightarrow D^*$ dimensions
❖ Reduce range of each dimension
❖ i.e., use uint8 instead of float
❖ Range reduction works by using the id of a centroid (say one of $2^8 = 256$ centroids)
❖ Original point can be approximated by remembering the position of the centroid

All of these applications require fast k-means clustering

Can we build fast implementations with good accuracy (ideally with some theoretical guarantees) and good scalability?
Our plan: implement a variety of k-means baselines

- k-means objective: partition input points into $k$ clusters $C_1, \ldots, C_k$ minimizing:

$$
\sum_{i=1}^{k} \sum_{x \in C_i} \|x - \mu_i\|^2
$$

$$
\mu_i = \text{mean}(C_i) = \frac{1}{|C_i|} \sum_{x \in C_i} x
$$

- Related to the idea of minimizing the variance of a cluster (also called “Sum-of-Squared Deviations”)
Lloyd’s Algorithm

- Lloyd’s algorithm (baseline)
- Consists of two steps. Suppose some initial centers $c_1, \ldots, c_k$ given:
  1. **Assignment:**
     - assign each $p \in P$ to the cluster corresponding to its nearest center
  2. **Update:**
     - recompute $c_i$ based on the set of points assigned to $C_i$
Lloyd’s Algorithm

- Lloyd’s algorithm (baseline)
- Consists of two steps. Suppose some initial centers $c_1, \ldots, c_k$ given:

  1. **Assignment:**
     - assign each $p \in P$ to the cluster corresponding to its nearest center

  2. **Update:**
     - recompute $c_i$ based on the set of points assigned to $C_i$
Lloyd’s Algorithm

- Lloyd’s algorithm (baseline)
- Consists of two steps. Suppose some initial centers $c_1, \ldots, c_k$ given:

  (1) Assignment:
  - assign each $p \in P$ to the cluster corresponding to its nearest center.

  (2) Update:
  - recompute $c_i$ based on the set of points assigned to $C_i$.

What is the cost of one Lloyd’s iteration in terms of $n, k, d$?

What potential for optimizations are there?
Better initialization: k-means++

❖ Instead of picking $k$ random centers initially:
  ❖ Pick one center uniformly at random
  ❖ For each point $p$ not yet elected as a center, compute $D(x)$, the distance between $p$ and its nearest center
  ❖ Sample an unchosen point to be chosen as the center where points are sampled with probability proportional to $D(x)^2$

❖ Amazingly, can show that the centers that result from this procedure are an $O(\log n)$ approximation of OPT (in expectation)
Scalable initialization: $k$-means||

- A slightly more complex scheme, but admits more parallelism:
  - Sample $O(k)$ points in each round
  - Repeat for approximately $O(\log n)$ rounds
  - Yields $O(k \log n)$ points that are then reclustered into $k$ initial centers

- Theory: initial $O(k \log n)$ centers give a constant factor approximation of $\text{OPT}$
Avoiding distance comparisons

- Costly part of Lloyd iteration is comparing each point $p$ with all $k$ centers (costs $O(nkd)$)
- Idea: use triangle inequality to avoid distance computations for points

\[
d(p, C_1(p)) \rightarrow d(p, C_2(p))
\]

\[
d(p, C_1(p)) + \delta(C_1(p))
\]

\[
d(p, C_2(p)) - \delta_{\text{max}}
\]
Project plan:

❖ Build a highly optimized shared-memory library of k-means implementations
❖ Evaluate existing algorithms for large $n, k, d$:
  ❖ $n = 1B$ points
  ❖ $k = 1M$ centers
  ❖ $d \in [100,1600]$

Evaluate performance on real-world embedding datasets from ANN search applications

(Hopefully) design new algorithms and heuristics to obtain scalability improvements at billion-scale!
Thank you!

Massively Parallel Computation

Adaptive Massively Parallel Computation

Low-Outdegree Orientation

Dynamic Connectivity

Euler Tour Trees

RC-Trees

Massively Parallel Computation

Graph-Based HAC

Connectivity and Related Problems

ConnectIt

Theoretically-Efficient Parallel Graph Algorithms

Static Graph Processing

Static Graph Processing

Graph-Based HAC

Connectivity and Related Problems

ConnectIt

Theoretically-Efficient Parallel Graph Algorithms

GBBS

Sage

Semi-Asymmetric Model

Graph Compression

Aspen

Julienne

laxman@umd.edu