Graph Processing in NVRAM and Streaming Settings

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

Based on joint work with

Guy Blelloch and Julian Shun (PLDI’19)

Charles McGuffey, Hong Kang, Yan Gu, Guy Blelloch, Phil Gibbons, and Julian Shun (VLDB’20)
Graph Processing: algorithms and systems that enable us to analyze and understand graphs
Graph Processing: algorithms and systems that enable us to analyze and understand graphs
**Graph Processing**: algorithms and systems that enable us to analyze and understand graphs.

- **Input Graph**
- **Graph Processing**
  - Algorithms
    - Connectivity
    - Graph Clustering
    - Distance Computations
    - Dense Subgraphs

...
**Graph Processing:** algorithms and systems that enable us to analyze and understand graphs

- **Input Graph**
- **Graph Processing**
- **Output**

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

- Understanding
- Visualizations
- Graph-based features
- System-optimization
**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.”

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
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

What about graphs that are larger-than-DRAM?

A single shared-memory machine can already store the largest publicly available graph datasets, with plenty of room to spare.
NVRAM Graph Processing
Non-Volatile Memory (NVRAM)

Intel Optane DC Memory

- Cheaper than DRAM on a per-byte basis
- Order of magnitude more capacity
- Memory is persistent and byte-addressable

Can we design algorithms that effectively use NVRAM as a higher-capacity memory while achieving DRAM-competitive performance?
Our Machine

48 cores with 2-way hyper-threading
375GB DRAM and 3.024TB of NVRAM

DRAM: 6x32 GB per socket
NVRAM: 6x256GB per socket

❖ 8x more NVRAM than DRAM
❖ NVRAM read throughput ~3x lower than DRAM read
❖ NVRAM write throughput further 4x lower
Recent work on Asymmetry

Benchmarking

❖ Two recent studies by Izraelevitz et al. [0] and van Renen et al. [1] perform careful benchmarking of Optane memory, and report similar asymmetries

Algorithms and Systems for Asymmetric Settings

❖ Recent work explores how to minimize the number of NVRAM writes, e.g., [2 – 4], including many other papers

❖ Also significant work from systems, architecture, and database communities, e.g., [5 – 7], amongst many other papers

Sources:
Can we design practical and theoretically-sound techniques to overcome read/write asymmetry for graph problems on NVRAMs?
Real World Graphs are not Ultra-Sparse

Over 90% of graphs with > 1M vertices from SNAP and LAW datasets have \( m/n \geq 10 \)

We expect that ratio of NVRAM/DRAM in future systems will be similar (our ratio is 8x)

Sources:
https://snap.stanford.edu/data/
http://law.di.unimi.it/datasets.php
Our Approach

Semi-Asymmetric Approach

- Graph stored in NVRAM and accessed in a read-only mode
- Amount of DRAM is proportional to the number of vertices

Algorithm $O(n)$ space

read/write read-only
Our Approach

Semi-Asymmetric Approach

- Graph stored in NVRAM and accessed in a read-only mode
- Amount of DRAM is proportional to the number of vertices

Benefits

- Algorithms avoid costly NVRAM writes, and algorithm design is independent of this cost
- Algorithms do not contribute to NVRAM wear-out
Our Approach

Semi-Asymmetric Approach

❖ Graph stored in NVRAM and accessed in a read-only mode
❖ Amount of DRAM is proportional to the number of vertices

Benefits

❖ Algorithms avoid costly NVRAM writes, and algorithm design is independent of this cost
❖ Algorithms do not contribute to NVRAM wear-out

Our contribution:
This (restrictive) semi-asymmetric approach is effective for designing fast parallel graph algorithms
Parallel Semi-Asymmetric Model (PSAM)
Parallel Semi-Asymmetric Model (PSAM)

CPUs

DRAM

NVRAM

Read/Write: Unit Cost

12
Parallel Semi-Asymmetric Model (PSAM)
Parallel Semi-Asymmetric Model (PSAM)

CPUs

DRAM

Read/Write: Unit Cost

NVRAM

Unbounded Size

Read: Unit Cost

Write: Cost $\omega > 1$
Parallel Semi-Asymmetric Model (PSAM)

- **CPUs**
  - Read/Write: Unit Cost

- **DRAM**
  - Regular model: $O(n)$
  - Relaxed model: $O(n + m/log n)$

- **NVRAM**
  - Unbounded Size
  - Read: Unit Cost
  - Write: Cost $\omega > 1$
Overview of Semi-Asymmetric Algorithms

- Start with work-efficient shared-memory algorithms from the Graph Based Benchmark Suite (GBBS)
- Implement interface primitives used by GBBS algorithms (edgeMap and filtering) efficiently in the PSAM
Overview of Semi-Asymmetric Algorithms

- Start with work-efficient shared-memory algorithms from the Graph Based Benchmark Suite (GBBS)
- Implement interface primitives used by GBBS algorithms (edgeMap and filtering) efficiently in the PSAM

**edgeMap**

<table>
<thead>
<tr>
<th>Problem</th>
<th>GBBS Work</th>
<th>Sage Work</th>
<th>Sage Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-First Search</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(d_{\text{max}} \log n)$</td>
</tr>
<tr>
<td>Weighted BFS</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(d_{\text{max}} \log n)$</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(d_{\text{max}} \log n)$</td>
</tr>
<tr>
<td>Single-Source Widest Path</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(d_{\text{max}} \log n)$</td>
</tr>
<tr>
<td>Single-Source Betweenness</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(d_{\text{max}} \log n)$</td>
</tr>
<tr>
<td>$O(\delta)$-Spanner</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(k \log n)$</td>
</tr>
<tr>
<td>LDD</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(k \log n)$</td>
</tr>
<tr>
<td>Connectivity</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(k \log n)$</td>
</tr>
<tr>
<td>Spanning Forest</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(k \log n)$</td>
</tr>
<tr>
<td>Graph Coloring</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(k \log n)$</td>
</tr>
<tr>
<td>Maximal Independent Set</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(k \log n)$</td>
</tr>
</tbody>
</table>

GBBS work indicates the work of naively converting existing shared-memory algorithms from GBBS to NVRAM algorithms

**GBBS Interface**

- **VertexSubset**: represent subsets of vertices
- **Bucketing**: dynamic mapping from IDs to set of ordered buckets
- **Vertex**: primitives on incident edges, e.g., map, reduce, filter, intersect, ...
- **Graph**: graph parallel operators, e.g., edgeMap, graph contraction, ...

**Graph Formats**

- Low-level access to CSR graph formats (uncompressed and compressed graph representations)

**Parallel Primitives and Runtime**

**Filtering (relaxed model)**

<table>
<thead>
<tr>
<th>Problem</th>
<th>GBBS Work</th>
<th>Sage Work</th>
<th>Sage Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biconnectivity</td>
<td>$O(mn)$</td>
<td>$O(mn)$</td>
<td>$O(d_{\text{max}} \log n)$</td>
</tr>
<tr>
<td>Apx. Set Cover</td>
<td>$O(mn)$</td>
<td>$O(mn)$</td>
<td>$O(d_{\text{max}} \log n)$</td>
</tr>
<tr>
<td>Triangle Counting</td>
<td>$O(mn)$</td>
<td>$O(mn)$</td>
<td>$O(d_{\text{max}} \log n)$</td>
</tr>
<tr>
<td>Maximal Matching</td>
<td>$O(mn)$</td>
<td>$O(mn)$</td>
<td>$O(d_{\text{max}} \log n)$</td>
</tr>
</tbody>
</table>

**Other Techniques**

<table>
<thead>
<tr>
<th>Problem</th>
<th>GBBS Work</th>
<th>Sage Work</th>
<th>Sage Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>PageRank Iteration</td>
<td>$O(mn)$</td>
<td>$O(mn)$</td>
<td>$O(d_{\text{max}} \log n)$</td>
</tr>
<tr>
<td>PageRank</td>
<td>$O(mn)$</td>
<td>$O(mn)$</td>
<td>$O(d_{\text{max}} \log n)$</td>
</tr>
<tr>
<td>k-core</td>
<td>$O(mn)$</td>
<td>$O(mn)$</td>
<td>$O(d_{\text{max}} \log n)$</td>
</tr>
<tr>
<td>Apx. Densest Subgraph</td>
<td>$O(mn)$</td>
<td>$O(mn)$</td>
<td>$O(d_{\text{max}} \log n)$</td>
</tr>
</tbody>
</table>
Semi-Asymmetric Filtering

Motivation

❖ Some algorithms remove, or batch-delete edges over the course of their operation for work-efficiency

❖ Modifying the graph directly requires writing to NVRAM
Semi-Asymmetric Filtering

Motivation

❖ Some algorithms remove, or batch-delete edges over the course of their operation for work-efficiency

❖ Modifying the graph directly requires writing to NVRAM

Semi-Asymmetric Filtering

❖ Work in the relaxed model

❖ Use one bit per edge and mirror the CSR structure (in NVRAM) using a blocked approach in DRAM
Semi-Asymmetric Filtering

Graph

- logically deleted
- present in graph
Semi-Asymmetric Filtering

High-level Approach

(i) Set a filter block size, and logically chunk the CSR structure into chunks of this size

Graph in CSR format, stored in NVRAM \((\mathcal{F}_B = 2)\)
Semi-Asymmetric Filtering

High-level Approach

(ii) Create a “mirrored” filter structure in DRAM, storing 1 bit per edge in NVRAM

GraphFilter in CSR format, stored in DRAM ($\mathcal{F}_B = 2$)
Semi-Asymmetric Filtering

Structure Overview

Note: Blocks with no "1" bits remaining are deleted
Relationship to Other Models

Semi-External Memory (SE) Model

- SE model performs block-transfers, with a focus on I/O cost [0, 1]
- Both PSAM and SE models provide the same amount of DRAM, but SE does not account for DRAM reads and writes

Sources:
Relationship to Other Models

Semi-External Memory (SE) Model

- SE model performs block-transfers, with a focus on I/O cost $[0, 1]$.
- Both PSAM and SE models provide the same amount of DRAM, but SE does not account for DRAM reads and writes.

Asymmetric RAM and Asymmetric Nested Parallel Models

- Both ARAM [2] and ANP [3] models capture asymmetry of writing to NVRAM.
- Unlike ARAM/ANP models, the PSAM includes a fast memory, and is specialized for graph problems.

Sources:
Semi-Asymmetric Graph Engine (Sage) Approach

App Direct Mode enables a direct implementation of PSAM algorithms
NUMA Optimization in Sage

Consider an algorithm that maps over all vertices, and for each vertex performs a reduction over the neighbors of the vertex.

Three experiments based on (threads, storage)
NUMA Optimization in Sage

Cross-socket NVM reads should be avoided

First run:
- Socket 0: 7 s
- Socket 1: > 4x slower

Subsequent runs:
- ~7 s
- Socket 0: 26 s
- Socket 1: ~7 s subsequently
NUMA Optimization in Sage

4.3 s for microbenchmark
Both graphs stored in compressed CSR format
Existing Approaches: DRAM as a Cache

- Applications do not distinguish between DRAM and NVRAM
- Existing shared-memory software does not require modification
- Workloads that are larger than DRAM can involve costly NVRAM writes
Applications do not distinguish between DRAM and NVRAM

Existing shared-memory software does not require modification

Workloads that are larger than DRAM can involve costly NVRAM writes

Galois (Gill et al.)

Gill et al. study the performance of the Galois engine using MemMode

They show promising results for scaling to larger than DRAM sizes
Applications do not distinguish between DRAM and NVRAM

Existing shared-memory software does not require modification

Workloads that are larger than DRAM can involve costly NVRAM writes

Galois (Gill et al.)

Gill et al. study the performance of the Galois engine using MemMode

They show promising results for scaling to larger than DRAM sizes

How does our approach compare?
WebDataCommons Graph

- Largest publicly available graph today
- 3.5B vertices connected by 128B edges (225B symmetrized)
Results for Larger-than-DRAM Graphs

WebDataCommons Graph

- Largest publicly available graph today
- 3.5B vertices connected by 128B edges (225B symmetrized)

Experiment

- Compare Sage results with
  - GBBS using MemMode (existing shared-memory codes)
  - Galois using MemMode (using numbers reported by authors on the same machine)
Results for Larger-than-DRAM Graphs

Run on a 48-core machine with 2-way hyper-threading, 375 GB of DRAM and 3 TB of NVRAM
Results for Larger-than-DRAM Graphs

Run on a 48-core machine with 2-way hyper-threading, 375 GB of DRAM and 3 TB of NVRAM

1.94x speedup on average over Galois (state-of-the-art existing approach to NVRAM graph processing), and 1.87x speedup over simply running GBBS codes using MemMode
Results for Graphs Stored in Main Memory

**ClueWeb Graph**

- Large web crawl with ~1B vertices connected by 42B edges (74B symmetrized)
- Graph fits entirely in the main memory of our machine

libvmmalloc: see [https://pmem.io/pmdk/libvmmalloc/](https://pmem.io/pmdk/libvmmalloc/)
Results for Graphs Stored in Main Memory

ClueWeb Graph

- Large web crawl with ~1B vertices connected by 42B edges (74B symmetrized)
- Graph fits entirely in the main memory of our machine

Experiment

- Compare Sage (graph stored on NVRAM) with
  - Sage (graph stored in DRAM)
  - GBBS (graph stored in DRAM)
  - GBBS with libvmmalloc (graph stored on NVRAM)

libvmmalloc: see https://pmem.io/pmdk/libvmmalloc/
Results for Graphs Stored in Main Memory

Run on a 48-core machine with 2-way hyper-threading, 375 GB of DRAM and 3 TB of NVRAM
Results for Graphs Stored in Main Memory

Run on a 48-core machine with 2-way hyper-threading, 375 GB of DRAM and 3 TB of NVRAM

Sage provides DRAM-competitive performance even when reading graph from NVRAM (only 5% slower on average)
Lessons and Directions for Future Work
Lessons and Directions for Future Work

Avoid Cross-Socket NVRAM Traffic

- NUMA optimization which reads from the copy of the read-only graph from the same socket achieves 6x speedup over cross-socket approach
Lessons and Directions for Future Work

Avoid Cross-Socket NVRAM Traffic

❖ NUMA optimization which reads from the copy of the read-only graph from the same socket achieves 6x speedup over cross-socket approach

Utilize App-Direct Mode

❖ Nearly 2x improvement for App-Direct based PSAM algorithms over two fast Memory Mode approaches
Lessons and Directions for Future Work

Avoid Cross-Socket NVRAM Traffic

❖ NUMA optimization which reads from the copy of the read-only graph from the same socket achieves 6x speedup over cross-socket approach

Utilize App-Direct Mode

❖ Nearly 2x improvement for App-Direct based PSAM algorithms over two fast Memory Mode approaches

Avoid NVRAM Writes

❖ PSAM implementations which only read from NVRAM are over 6x faster than our algorithms which write to NVRAM (using libvmmalloc)
Streaming Graph Processing
Dynamic Graph Processing

Measuring the spread of infections

Source: Infection transmission in a dynamic network
Dynamic Graph Processing

Measuring the spread of infections

Source: Infection transmission in a dynamic network
Dynamic Graph Processing

Measuring the spread of infections

Preventing money laundering and fraud

Other Applications
- Recommendation Systems
- Geospatial Systems
Dynamic Graph Processing

Measuring the spread of infections

Preventing money laundering and fraud

Other Applications
- Recommendation Systems
- Geospatial Systems

Many important applications must maintain information about evolving graphs!

Source: Infection transmission in a dynamic network
Dynamic Graph Processing: Example
Dynamic Graph Processing: Example
Dynamic Graph Processing: Example
Dynamic Graph Processing: Example

Update Stream
Dynamic Graph Processing: Example

Update Stream

The Washington Post

REUTERS

Query Stream

Fetch similar vertices

Clustering Coefficients

Fetch vertex u’s neighbors

Centrality Ranking
Dynamic Graph Processing: Example

Update Stream

The Washington Post

REUTERS

Query Stream

Fetch similar vertices
Fetch vertex u's neighbors

Clustering Coefficients
Centrality Ranking

Streaming Graph Processing
Update the graph (in parallel);
Execute arbitrary queries on snapshots.
Dynamic Graph Processing: Example

Update Stream
The Washington Post
REUTERS

Query Stream
Fetch similar vertices
Clustering Coefficients
Fetch vertex u's neighbors
Centrality Ranking

Streaming Graph Processing
Update the graph (in parallel);
Execute arbitrary queries on snapshots.

Clustering
Triangle Counting
Connected Components
Dynamic Graph Processing: Example

**Update Stream**
- The Washington Post
- Reuters

**Query Stream**
- Fetch similar vertices
- Clustering Coefficients
- Fetch vertex u’s neighbors
- Centrality Ranking

**Streaming Graph Processing**
- Update the graph (in parallel);
- Execute arbitrary queries on snapshots.

**Batch-Dynamic Graph Processing**
- Pre-determined queries;
- Process updates faster than recomputation.

**Clustering**
- Triangle Counting
- Connected Components
Streaming Graph Processing

- Edge queries
- Local algorithms
- Global algorithms

Graph Updates → Graph-Streaming System → Responses

Graph Queries

Graph-Streaming System
Graph Query Processing

Goal: low-latency for both updates and queries arriving concurrently to the system
Streaming Graph Processing

Graph Updates

Graph-Streaming System

Graph Queries

Responses

Edge queries

Local algorithms

Global algorithms

Goal: low-latency for both updates and queries arriving concurrently to the system

Single-version

STINGER [EMRB’12]
cuSTINGER [GB’16]
Kickstarter [VGX’17]
Streaming Graph Processing

**Graph Updates**

**Graph Queries**

**Graph-Streaming System**

**Responses**

**Goal:** low-latency for both updates and queries arriving concurrently to the system

---

### Single-version

- **STINGER** [EMRB’12]
- **cuSTINGER** [GB’16]
- **Kickstarter** [VGX’17]

### Multi-version (snapshot-based)

- **Kineograph** [CHKMW+’12]
- **LLAMA** [MMMS’15]
Low-Latency Graph Streaming using Compressed Purely-Functional Trees [DBS’19]

Can we design a system that can compactly represent and concurrently update and query the largest real-world graphs?
Aspen: A Low-Latency Graph Streaming System

Snapshots implement the GBBS interface, making it possible to run parallel graph algorithms from GBBS on snapshots in Aspen.

- Breadth-First Search
- Maximal Independent Set
- Parallel Connectivity
- And many others

Purely-Functional Graph Representation

Compressed Purely-Functional Trees
Aspen: A Low-Latency Graph Streaming System

Graph Algorithm Interface

Snapshots implement the GBBS interface, making it possible to run parallel graph algorithms from GBBS on snapshots in Aspen.

Breadth-First Search

Maximal Independent Set

Parallel Connectivity

And many others

Aspen: A Low-Latency Graph Streaming System

Query Interface

Acquire

Release

Acquires or releases a snapshot of the graph.

Update Interface

InsertBatch

DeleteBatch

Update the graph with the changes in the sequence of edge insertions or deletions

Purely-Functional Graph Representation

Compressed Purely-Functional Trees

InsertBatch

Update Interface

DeleteBatch

Query Interface

Acquire

Release

Acquires or releases a snapshot of the graph.
Graph Algorithm Interface

Snapshots implement the GBBS interface, making it possible to run parallel graph algorithms from GBBS on snapshots in Aspen.

Breadth-First Search
Maximal Independent Set
Parallel Connectivity
And many others

Query Interface

Acquire
Release

Acquires or releases a snapshot of the graph.

Update Interface

InsertBatch
DeleteBatch

Update the graph with the changes in the sequence of edge insertions or deletions.

Aspen: A Low-Latency Graph Streaming System

Purely-Functional Graph Representation

Compressed Purely-Functional Trees
Main contribution: designing a scalable, space-efficient, and efficiently-updatable graph representation using compressed purely-functional trees
Purely-Functional Trees
Purely-Functional Trees

Red-black, AVL, or weight-balanced trees
Representing Graphs using Trees

G
Representing Graphs using Trees

![Graph G](image)

![Tree representation of G](image)
Representing Graphs using Trees

Graph $G$

Vertex 0's Edge Tree

Tree $(G)$
Representing Graphs using Trees

(Vertex 0’s Edge Tree)

(Vertex 3’s Edge Tree)
Trees enable Simple Snapshots
Trees enable Simple Snapshots

A snapshot is just a tree root
Trees enable Simple Snapshots

A snapshot is just a tree root
Trees enable Simple Snapshots

A snapshot is just a tree root

Insert(12)
Trees enable Simple Snapshots

A snapshot is just a tree root

Insert(12)
Trees enable Simple Snapshots

A snapshot is just a tree root

Algorithms generalize to handle batches of updates in low work/depth [BFS’16]
Purely-Functional Trees are Safe for Concurrency
Purely-Functional Trees are Safe for Concurrency
Purely-Functional Trees are Safe for Concurrency
Purely-Functional Trees are Safe for Concurrency
Queries are serialized once they acquire a tree root.
Challenges
Challenges

Poor Cache Usage
Challenges

Poor Cache Usage

Edge Tree

2

0

5

1
Challenges

Poor Cache Usage

Space Inefficiency
Challenges

Poor Cache Usage

Space Inefficiency

- Significant space overheads for tree nodes
- Lose ability to compress adjacency lists
Challenges

Poor Cache Usage

Space Inefficiency

❖ Significant space overheads for tree nodes
❖ Lose ability to compress adjacency lists
Challenges

Poor Cache Usage

- Significant space overheads for tree nodes
- Lose ability to compress adjacency lists

Space Inefficiency

- Significant space overheads for tree nodes
- Lose ability to compress adjacency lists

Aspen using naive approach requires 7 TB of memory for WDC2012 graph
Challenges

Poor Cache Usage
❖ Significant space overheads for tree nodes
❖ Lose ability to compress adjacency lists

Space Inefficiency
❖ Significant space overheads for tree nodes
❖ Lose ability to compress adjacency lists

To overcome these challenges we designed C-trees: compressed purely-functional trees

Aspen using naive approach requires 7 TB of memory for WDC2012 graph
C-trees

❖ Chunking parameter $B$. Fix a hash function, $h$

❖ Select elements as *heads* with probability $1/B$ using $h$

![C-tree diagram]

= heads
C-trees

- Chunking parameter $B$. Fix a hash function, $h$
- Select elements as *heads* with probability $1/B$ using $h$
C-trees

- Chunking parameter $B$. Fix a hash function, $h$
- Select elements as *heads* with probability $1/B$ using $h$
C-trees

- Chunking parameter $B$. Fix a hash function, $h$
- Select elements as *heads* with probability $1/B$ using $h$
C-trees

- Chunking parameter $B$. Fix a hash function, $h$
- Select elements as *heads* with probability $1/B$ using $h$

Further improve space usage for integer C-trees by difference encoding chunks
Space Improvement in Aspen using C-trees
Space Improvement in Aspen using C-trees

![Graph showing space usage compared to number of edges for various datasets: Orkut, Twitter, ClueWeb, LiveJournal, WDC2012, and WDC2014. The graph compares Purely-functional tree, parallel-byte, and C-tree with 9x smaller space improvement.](image-url)
Space Improvement in Aspen using C-trees

Fully-dynamic representation of the WebDataCommons hyperlink graph using 700GB of memory
Operations on C-trees
Build(Seq $S$)
Build(Seq $S$)

Map(Ctree $C, f$)
Build(Seq $S$)

Map(Ctree $C,f$)

MultiInsert(Ctree $C$, Seq $S$)
Build(Seq $S$)

Map(Ctree $C, f$)

MultiInsert(Ctree $C$, Seq $S$)

$C_S = \text{Build(Seq } S)\]
\text{Output } = \text{Union}(C, C_S)$
Batch Updates on Trees

\[ \text{union}(t_1, t_2) \]

[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA’16)
Batch Updates on Trees

union(t₁, t₂)

[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA’16)
Batch Updates on Trees

union($t_1$, $t_2$)

expose($t_1$)

split($t_2$, $k_1$)

[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA’16)
Batch Updates on Trees

union(t₁, t₂)

[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA’16)
Batch Updates on Trees

union($t_1, t_2$)

$\text{expose}(t_1)$

$\text{split}(t_2, k_1)$

$\text{join}(L, k_1, R)$

$\text{union}(l_1, l_2)$

$\text{union}(r_1, r_2)$

Similar algorithms for difference and intersection

[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA’16)
The Join Function

$\text{join}(L, k, R)$

[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA’16)
Join enables balance-agnostic expression of all other primitives[1]

[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA’16)
Batch Updates on Trees

union($t_1, t_2$) runs in

$$O\left( m \log \left( \frac{n}{m} + 1 \right) \right)$$

work and

$$O\left( \log n \log m \right)$$

depth
Batch Updates on Trees

union($t_1, t_2$) runs in

$$O\left( m \log \left( \frac{n}{m} + 1 \right) \right)$$ work and

$$O \left( \log n \log m \right)$$ depth

Proof idea from [1]:

Splitting a tree costs $O \left( \log |T| \right)$ work and depth

Overall cost = work done over all splits

[1] Just Join for Parallel Ordered Sets, Blelloch et al. (SPAA’16)
Batch Updates on Trees

\[ \text{union}(C_1=(T_1, P_1)), C_2=(T_2, P_2)) \]
Batch Updates on Trees

\[ \text{union}(C_1=(T_1, P_1)), C_2=(T_2, P_2)) \]

Expose one of the trees
Batch Updates on Trees

union(C1=(T1, P1)), C2=(T2, P2))

Split the other C-tree with k2
Batch Updates on Trees

union(C1=(T1, P1)), C2=(T2, P2))

Split the other C-tree with k2

Part of v2 may belong in BT2, similarly with BP2
Batch Updates on C-trees

union(C_1=(T_1, P_1)), C_2=(T_2, P_2))

- Expose(T_2)
- v_2: tail
- L_2
- R_2

- (B_1, (BT_2, BP_2)) = Split(C_1, k_2)
- B_2 = (BT_2, BP_2)
- < k_2
- > k_2

- SplitChunk(v_2, Smallest(BT_2))
- SplitChunk(BP_2, Smallest(R_2))

Split v_2 based on BT_2, BP_2 based on R_2
Batch Updates on Trees

union(C₁=(T₁, P₁)), C₂=(T₂, P₂))

Recursive union of two C-trees

Join done on the underlying purely-functional tree
Batch Updates on Trees

union(C₁=(T₁, P₁)), C₂=(T₂, P₂))

union(C₁, C₂) runs in

\[ O\left(B^2m \log \left(\frac{n}{m} + 1\right)\right) \] expected work

\[ O\left(B \log n \log m\right) \] depth whp
Experiments
Our Machine

Dell PowerEdge R930

- 72-cores, 2-way hyper-threaded*
- 1TB of main memory
- Cost: about 20k USD

* (4 x 2.4GHz 18-core E7-8867 v4 Xeon processors)
Our Machine

Dell PowerEdge R930

- 72-cores, 2-way hyper-threaded*
- 1TB of main memory
- Cost: about 20k USD

* (4 x 2.4GHz 18-core E7-8867 v4 Xeon processors)
Streaming Experiment
Streaming Experiment

Sampled insertions + deletions from G
Streaming Experiment

Stream of Parallel BFS queries from random vertices

Sampled insertions + deletions from G

BFS trees
Streaming Experiment

Stream of Parallel BFS queries from random vertices

BFS trees

Sampled insertions + deletions from G

What’s the impact on the concurrent execution on latency?
Less than 3% impact on queries in the concurrent setting
Batch Update Experiment
Batch Update Experiment

Edge insertions drawn from RMAT
Batch Update Experiment

Edge insertions drawn from RMAT

Represent $G$ using Aspen and STINGER
Batch Update Experiment

Edge insertions drawn from RMAT

Represent $G$ using Aspen and STINGER

How does the throughput scale as a function of batch size?
Batch Update Performance

![Graph showing batch update performance](image)

Throughput on 72 cores

Batch Size

- Aspen Batch Updates
- STINGER Batch Updates
Batch Update Performance

Throughput on 72 cores

32x

Batch Update Performance

Aspen Batch Updates
STINGER Batch Updates
Batch Update Performance

![Graph showing throughput on 72 cores vs. batch size for Aspen and STINGER batch updates. The graph indicates a significant performance improvement.]
Building on Aspen and C-trees
Batch-Dynamic Graph Processing

Dynamic Algorithm

$G_{\text{Prev}}$  $G_{\text{Next}}$

Updates

Aspen

E.g.: Connected components, clustering coefficients, graph clusterings, etc
Batch-Dynamic Algorithms

Interested in practical and memory-efficient dynamic graph algorithms

Thank you!

Aspen
Scalable graph data structures and interfaces for processing streaming graphs

[github.com/ldhulipala/aspen](https://github.com/ldhulipala/aspen)

- has strong theoretical bounds
- provides memory-efficient graph representations
- enables lightweight snapshots
- runs on commodity hardware
- can process the largest publicly-available graphs