Low-Latency Graph Streaming Using Compressed Purely-Functional Trees

Laxman Dhulipala, Guy Blelloch, and Julian Shun
Streaming Graph Processing

Update Stream

Query Stream

G : graph

Goals: Serializability for updates/queries, achieve low latency and high throughput
Example: Fraud Detection

- Bank maintains a transaction graph
- Transactions occur at a high rate (1k-10k/sec)
- Goal: quickly detect anomalies in evolving transaction graph
Relaxing Serializability

- Could detect a cycle that never existed!
Existing Work

• Single Version Systems
  • Maintain a **single** version of the graph
  • Common approach in graph streaming (e.g., STINGER, cuSTINGER, and KickStarter)
  • Need to separate queries from updates for serializability

• Multi-Version Systems
  • Support multiple graph snapshots (e.g., LLAMA, Kineograph, GraphOne, and some graph databases)
  • Snapshots are not space-efficient and lead to high latency

• **Our framework Aspen** uses lightweight snapshots to enable low-latency concurrent queries and updates
Terminology: Streaming vs. Dynamic

• **Streaming graph processing**: Goal is to run algorithms on a graph that is changing in real-time while obtaining serializable results
  - Need to process updates concurrently with algorithm execution

• **Dynamic graph algorithms**: Goal is to update the result of an algorithm based on updates to the graph itself
  - Should be more efficient than recomputing answer from scratch
  - Allows for barriers between algorithm execution and processing updates

• This talk is about streaming graph processing
Graphs Using Purely Functional Trees

• Purely functional trees can be updated efficiently (in logarithmic time/space) while retaining old copy of tree
• Aspen uses tree of vertices, where each vertex stores a tree of its incident edges

= vertex

= edge tree
Updates via Path Copying

- Easy to generate new versions via path copying
Updates via Path Copying

- Easy to generate new versions via path copying
Updates via Path Copying

- Easy to generate new versions via path copying

![Diagram showing path copying](image)
Updates via Path Copying

- Easy to generate new versions via path copying

- We can obtain immutability versions of the tree
Immutability Enables Concurrency

Latest version

#refs=1
Immutability Enables Concurrency
Immutability Enables Concurrency

Latest version

#refs=2
Immutability Enables Concurrency
Immutability Enables Concurrency
Immutability Enables Concurrency

Latest version

#refs=1

#refs=1
Immutability Enables Concurrency
Immutability Enables Concurrency

Garbage collect all tree nodes whose reference count is decremented to 0
Disadvantages of representing graphs using trees

- Poor Cache Usage
  - One tree node per vertex and edge
  - One cache miss per edge access in the worst case

- Space Inefficiency
  - Need to store children pointers and metadata on tree nodes
  - Lose ability to perform integer compression

Requires close to 7TB of memory to store the symmetrized Hyperlink 2012 graph (225B edges)!
Space Overhead of Graphs using Trees

Ligra+: state-of-the-art static compressed graph representation supporting efficient parallel operations.
Space Overhead of Graphs using Trees

![Graph showing space overhead for different datasets using trees and Ligra+](image)

- **Graphs using trees**
- **Ligra+**

- **LiveJournal**
- **Orkut**
- **Twitter**
- **ClueWeb** (HL2012, HL2014)

- **Space used (Gb)**: $10^{-2}$ to $10^{4}$
- **Number of edges**: $10^{7}$ to $10^{12}$

- **6.8Tb**
- **351Gb**
- **19.6x**

- **Note**: The graph illustrates the space overhead of graphs using trees and Ligra+ as the number of edges increases.
C-tree

- Purely functional **compressed** tree data structure
- 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

- Supports parallel bulk insertions and deletions efficiently
Space Usage of Graphs using C-trees
Space Usage of Graphs using C-trees

The diagram illustrates the space usage (in Gb) of different graph datasets as a function of the number of edges. The x-axis represents the number of edges, ranging from $10^7$ to $10^{12}$, while the y-axis represents the space used, ranging from $10^{-2}$ to $10^4$ Gb.

- **Graphs using trees** are shown with blue dots.
- **Ligra+** is represented by yellow dots.
- **Aspen** is indicated with green dots.

Key points:
- HL2012: 701 Gb
- HL2014: 351 Gb
- ClueWeb: 6.8 Tb
- Twitter: 9x better

The graph compares the space usage of various graph datasets and demonstrates the efficiency of C-trees in managing large graphs.
Aspen Framework

- Extension of Ligra with primitives for **updating graphs**
- Supports single-writer multi-reader concurrency

```
versioned graph
acquire()
release(G_s)
multi_insert(E_B), multi_delete(E_B)
...```

Aspen

- Updating Interface

Ligra

- Bucketing
- Vertex Subsets
- Graphs
Concurrent Queries and Updates

• 72-core hyper-threaded machine with 1TB RAM
• 1 hyper-thread updating graph while remaining hyper-threads running parallel BFS

Less than 3% impact on queries in concurrent setting
Aspen processes the Hyperlink 2012 graph at over 100M edge updates per second

About 1.4x faster than GraphOne (developed concurrently and independently) based on a rough comparison
Conclusion

- Aspen: a framework for streaming graph processing using purely functional trees
  - Code online: https://github.com/ldhulipala/aspen/

- Current bounds for C-tree are randomized
  - Ongoing work on designing a deterministic version

- Aspen for external memory or other settings

- Lots of papers on individual dynamic graph algorithms (mostly sequential, a few parallel)
  - Ongoing work: parallel dynamic graph algorithms
  - Open question: design a high-level parallel programming framework
  - Bigger open question: design a framework for dynamic graph algorithms in the streaming setting