PhD Dissertation Defense:
Systems Abstractions for Big Data Processing on a Single Machine

Steffen Maass

Advisor: Taesoo Kim

April 3, 2019
Large-scale data processing is ubiquitous

One Trillion Edges: Graph Processing at Facebook-Scale

Avery Ching
Facebook
1 Hacker Lane
Menlo Park, California
aching@fb.com

Sergey Edunov
Facebook
1 Hacker Lane
Menlo Park, California
edunov@fb.com

Maja Kabiljo
Facebook
1 Hacker Lane
Menlo Park, California
majakabiljo@fb.com

Dionysis Logothetis
Facebook
1 Hacker Lane
Menlo Park, California
dionysios@fb.com

Sambavi Muthukrishnan
Facebook
1 Hacker Lane
Menlo Park, California
sambavim@fb.com

ABSTRACT
Analyzing large graphs provides valuable insights for social networking and web companies in content ranking and recommendations. While numerous graph processing systems have been developed and evaluated on available benchmark graphs of up to 60B edges, they often face significant difficulties in scaling to much larger graphs. Industry graphs can be two orders of magnitude larger - hundreds of billions or up to one trillion edges. In addition to scalability challenges, real-world applications often require much more complex graph processing workflows than previously avail-

Table 1: Popular benchmark graphs.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal [8]</td>
<td>4.8M</td>
<td>98M</td>
</tr>
<tr>
<td>Twitter 2010 [4]</td>
<td>422M</td>
<td>1.6B</td>
</tr>
<tr>
<td>UK web graph 2007 [10]</td>
<td>1.98B</td>
<td>3.7B</td>
</tr>
<tr>
<td>Yahoo web [9]</td>
<td>1.4B</td>
<td>0.6B</td>
</tr>
</tbody>
</table>

Social networks
Large-scale data processing is ubiquitous

**Social networks**

**Genome analysis**
Large-scale data processing is ubiquitous

Social networks

Graphs enable Machine Learning

Genome analysis

Learning with Minimal Supervision

Graph-powered Machine Learning at Google

Thursday, October 06, 2016

Posted by Sujith Ravi, Staff Research Scientist, Google Research

Recently, there have been significant advances in Machine Learning that enable computer systems to solve complex real-world problems. One of those advances is Google’s large scale, graph-based machine learning platform, built by the Expander team in Google Research. A technology that is behind many of the Google products and features you may use everyday, graph-based machine learning is a powerful tool that can be used to power useful features such as reminders in inbox and smart messaging in Allo, or used in conjunction with deep neural networks to power the latest image recognition system in Google Photos.
Gigabytes of RAM
Powerful single machines available

Intel Unveils Plans for Knights Mill, a Xeon Phi for Deep Learning

At the Intel Developer Forum (IDF) this week in San Francisco, Intel revealed it is working on a new Xeon Phi processor aimed at deep learning applications. Diane Bryant, executive VP and GM of Intel’s Data Center Group, unveiled the new chip, known as Knights Mill, during her IDF keynote address on Wednesday.

Gigabytes of RAM

Powerful many-core coprocessors
Powerful single machines available

Gigabytes of RAM

Powerful many-core coprocessors

Fast, large-capacity non-volatile Memory
Powerful single machines available

Gigabytes of RAM

Powerful many-core coprocessors

Fast, large-capacity non-volatile Memory

⇒ Enable tera-scale data processing on a single machine

However: design and systems-level mechanisms required!
Large-scale big data analytics is possible on a single machine using systems and design-level abstractions for commodity and heterogeneous single machines.
Large-scale big data analytics is possible on a single machine using systems and design-level abstractions for commodity and heterogeneous single machines.

Approaches:

- **Systems-level** analysis and mechanism for improving virtual memory performance with LATR
- **Design-level** abstractions for trillion-scale graph analytics with Mosaic
- **Design-level** abstractions and data structures for billion-scale evolving graphs with Cytom
High-level structure

Operating System (LATR)
- Static (MOSAIC)
- Evolving (CYTOM)

Graph Processing
- Static (MOSAIC)
- Evolving (CYTOM)

Operating System (LATR)

Raw Data

Communities

Ranking

Anomaly

Raw Data

Operating System (LATR)

Graph Processing

Communities

Ranking

Anomaly
Focus of this talk: Processing evolving graphs with **CYTOM**
Cyton builds on lessons learned with Mosaic

However: Different challenges for setting of evolving graphs
Roadmap

1 Thesis Statement and Approaches

2 Cytom: Processing Billion-Scale Evolving Graphs
   - Applications & Challenges
   - Cytom’s Cell-based Graph Representation
   - Handling Deletions
   - Programming Interface

3 Conclusion
New Tweets per second record, and how!

By @rafi

Friday, 16 August 2013  

Recently, something remarkable happened on Twitter: On Saturday, August 3 in Japan, people watched an airing of *Castle in the Sky*, and at one moment they took to Twitter so much that we hit a one-second peak of 143,199 Tweets per second. (August 2 at 7:21:50 PDT; August 3 at 11:21:50 JST)

To give you some context of how that compares to typical numbers, we normally take in more than 500 million Tweets a day which means about 5,700 Tweets a second, on average. This particular spike was around 25 times greater than our steady state.

Social networks
Importance of evolving graphs

New Tweets per second record, and how!

Social network

Live traffic & route calculation
Importance of evolving graphs

New Tweets per second record, and how!

Social network

Live traffic & route

Analysis of next-gen Cellular Data, Connected Cars, ...
Importance of evolving graphs

New Tweets per second record, and how!

Applications enable by processing evolving graphs

- Online route calculation
- Online Anomaly detection
- Online Ranking
- Online Community detection

⇒ Applications enable by processing evolving graphs

Live traffic & route calculation

Analysis of next-gen Cellular Data, Connected Cars, ...
Challenges of billion-scale evolving graphs

- Constant stream of updates
- Optimizing graph storage format not possible on every update
  - Graph data structures supporting updates
- Co-running graph updates with algorithm
- Special challenges for single machines (compared to distributed systems):
  - Limited memory & compute
  - Fault tolerance
Background: Graph data structures for evolving graphs

- Goal: Efficient update support and computational efficiency
- Four options:
  - Adjacency matrix
  - Adjacency lists
  - Compressed sparse rows (CSR)
  - Edge lists
Background: Graph data structures for evolving graphs

- Goal: Efficient update support and computational efficiency
- Four options:
  - Adjacency matrix
  - Adjacency lists
  - Compressed sparse rows (CSR)
  - Edge lists

Sample graph:
Background: Adjacency matrix

- Minimal storage per edge (1 bit)
- Great for dense graphs
- Limitation: Most real-world graphs are sparse

Example:

![Adjacency matrix example](image)
For best case, storage cost per edge: 8 bytes
Relatively simple construction
Limitation: Overhead on traversal and allocation of new edges
Used by Stinger
For best case, storage cost per edge: 8 bytes
Efficient access and lower storage overhead
Limitation: Changes require rebuilding or complicated overflow areas
Used by Llama
Background: Compressed sparse rows (CSR)

- For best case, storage cost per edge: 8 bytes
- Efficient access and lower storage overhead
- Limitation: Changes require rebuilding or complicated overflow areas
- Used by Llama

Example: Add edge 4
Background: Compressed sparse rows (CSR)

- For best case, storage cost per edge: 8 bytes
- Efficient access and lower storage overhead
- Limitation: Changes require rebuilding or complicated overflow areas
- Used by Llama

Problem: Array has to be shifted
Background: Edge lists

- Simple construction
- Adding new edges possible
- Limitation: Overhead for storage cost, 16 bytes per edge
- Used by Graphln, GraphOne

```
  4 ➔ 1 ➔ 5
  2 ➔ 3 ➔ 6
```

1. (4, 5)
2. (4, 6)
3. (5, 6)
## Data structures summary

### Comparison of presented data structures

<table>
<thead>
<tr>
<th>Representation</th>
<th>Sparse graphs</th>
<th>Updates</th>
<th>Traversals</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adj. matrix</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>1 bit</td>
</tr>
<tr>
<td>Adj. lists</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>8 bytes</td>
</tr>
<tr>
<td>CSR</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Edge list</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>16 bytes</td>
</tr>
</tbody>
</table>
Comparison of presented data structures

<table>
<thead>
<tr>
<th>Representation</th>
<th>Sparse graphs</th>
<th>Updates</th>
<th>Traversals</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adj. matrix</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>1 bit</td>
</tr>
<tr>
<td>Adj. lists</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>8 bytes</td>
</tr>
<tr>
<td>CSR</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Edge list</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>16 bytes</td>
</tr>
<tr>
<td>Edge list&lt;sub&gt;CYTOTM&lt;/sub&gt;</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

- Our choice for evolving graphs: *edge lists*
- However: Mitigation for storage overhead needed
High-level architecture

From raw input graph to algorithmic output:

- **Input Graph**
  - **Tissue Manager 1**
    - **Cell Manager**
      - $\text{Cell Manager}_{11}$
      - $\text{Cell Manager}_{22}$
      - $\text{Cell Manager}_{12}$
      - $\text{Cell Manager}_{21}$
  - **Tissue Manager 2**
- **Cell Distributor**
  - **Persistence Manager**
  - **Algorithm Executor**
  - **Global vertex state**
    - <current state>
    - <next state>
  - **Checkpoints**
High-level architecture

Step 1: Update to input graph
High-level architecture

Step 2: Update Cytom’s internal structure
Step 3: Distribute update
Step 4: Algorithmic processing of update
First focus: Cytom’s cells
Construction of *Cytom’s* cells

- Lightweight partitioning

Step 1: Input graph:
Construction of Cytom’s cells

- Lightweight partitioning

Step 2: Adjacency Matrix
Construction of Cytom’s cells

- Lightweight partitioning

Step 3: Construction of cell $C_{11}$
Construction of CYTOM’s cells

- Lightweight partitioning

Step 4: Construction of all other cells
Construction of Cytom’s cells

- Lightweight partitioning

Step 5: Overlay current cell size
Construction of CYTOM’s cells: Adding new edges

- Adding new edges lightweight
- Division to obtain partition
Construction of Cytom’s cells: Adding new edges

- Adding new edges lightweight
- Division to obtain partition

![Diagram of Cytom's cells with added edges and global adjacency matrix]

Target vertex

Source vertex

Global adjacency matrix

Cell

Steffen Maass
Abstractions for Big Data Processing
April 3, 2019 19 / 46
Cytom’s cells

- Subgraph-centric
- Difference to *tiles* (MOSAIC) or *partitions* (GridGraph)?
  - Both designed for static execution
  - Immutable structures
  - Changes only possible coupled with preprocessing step
  - Cytom’s cells can grow (and shrink) as required without preprocessing
- Enables short identifiers (16 bits)
  - $2 \times$ to $4 \times$ less memory (32 or 64 bit identifiers)
  - Compared to MOSAIC: No metadata needed
Benefit of Cytom’s cells

- Enables short identifiers (16 bits)
  - 32 bits (4 bytes) per edge
  - $2 \times$ to $4 \times$ less memory (32 or 64 bit identifiers)
  - No metadata needed
- Usage of edge lists
  - Insert: Append at end of vector

Base case with global edge lists

Source vertex

Target vertex

Edge list

$\begin{align*}
\text{Source vertex} & \quad \text{Target vertex} \\
\begin{array}{ccc}
4 & 5 & 6 \\
4 & 3 & 5 \\
5 & 6 \\
6 \\
\end{array} & \quad \begin{array}{c}
(4, 5) \\
(5, 6) \\
64b \quad 64b \\
16B \\
\end{array}
\end{align*}$
Benefit of *Cytom*’s cells

- Enables short identifiers (16 bits)
  - 32 bits (4 bytes) per edge
  - $2 \times$ to $4 \times$ less memory (32 or 64 bit identifiers)
  - No metadata needed
- Usage of edge lists
  - Insert: Append at end of vector

Idea: Translate global IDs into local ones, store in *Cytom*’s edge list
Benefit of Cytom’s cells

- Enables short identifiers (16 bits)
  - 32 bits (4 bytes) per edge
  - $2 \times$ to $4 \times$ less memory (32 or 64 bit identifiers)
  - No metadata needed
- Usage of edge lists
  - Insert: Append at end of vector

Idea: Translate global IDs into local ones, store in Cytom’s edge list

⇒ Impact of shortened identifiers?

Source vertex
Target vertex

Edge list$_{Cytom}$

$\begin{array}{l}
\text{(1, 2)} \\
\text{(1, 3)} \\
\text{(2, 3)}
\end{array}$

$16b$ $16b$

$4R$
Impact of short identifiers - lower memory footprint

⇒ Up to 24% higher overall performance, 47% fewer cache references
How to operate on active cells?
How to load balance cells?

- Problem: Many more cells than threads
  - Millions of cells, dozens of threads
- Skewed cell size distribution
How to load balance cells?

- **Problem:** Many more cells than threads
  - Millions of cells, dozens of threads
  - Skewed cell size distribution

Four choices implemented:
- Static partitioning
- Work pool
- Cell distributor
- Hierarchical cell distributor

![Graphs showing cell distribution](image)

- Count Edges in Cell
  - Sorted Cell ID

- Orkut
  - Count Edges in Cell vs Sorted Cell ID
  - 1 × 10^7
  - 1 × 10^6
  - 100000
  - 10000
  - 1000
  - 100
  - 10
  - 1

- Livejournal
  - Count Edges in Cell vs Sorted Cell ID
  - 1 × 10^7
  - 1 × 10^6
  - 100000
  - 10000
  - 1000
  - 100
  - 10
  - 1
How to load balance cells?

- Problem: Many more cells than threads
  - Millions of cells, dozens of threads
- Skewed cell size distribution
- Four choices implemented:
  - Static partitioning

![Graph showing throughput vs. static]

Throughput (M edges/sec)

0.0M, 2.0M, 4.0M, 6.0M, 8.0M, 10.0M, 12.0M
How to load balance cells?

- Problem: Many more cells than threads
  - Millions of cells, dozens of threads
- Skewed cell size distribution
- Four choices implemented:
  - Static partitioning
  - Work pool

![Graph showing throughput comparison between static and work-pool methods]
How to load balance cells?

- Problem: Many more cells than threads
  - Millions of cells, dozens of threads
- Skewed cell size distribution
- Four choices implemented:
  - Static partitioning
  - Work pool
  - Cell distributor
How to load balance cells?

- Problem: Many more cells than threads
  - Millions of cells, dozens of threads
- Skewed cell size distribution
- Four choices implemented:
  - Static partitioning
  - Work pool
  - Cell distributor
  - Hierarchical cell distributor
How to load balance cells?

- Problem: Many more cells than threads
  - Millions of cells, dozens of threads
- Skewed cell size distribution
- Four choices implemented:
  - Static partitioning
  - Work pool
  - Cell distributor
  - Hierarchical cell distributor

⇒ Hierarchical cell distributor improves throughput by 2.1×
Further optimizations enabled by 2D structure?
2D-partitioning: Opportunity for different traversals

- Multiple traversal strategies are possible with CYTOM

- Evaluated options:
  - Column first
    - Read optimized

![Diagram of global adjacency matrix and partitioning]

Source vertex | Global adjacency matrix | Target vertex
--- | --- | ---
P11, P12, P14, P13 | | P14
P21, P22, P23, P24 | | P24
P31, P32, P34, P33 | | P33
P41, P42, P44, P43 | | P44

Partition (S = 3)
Multiple traversal strategies are possible with **Cytom**

- **Evaluated options:**
  - Column first
    - **Read** optimized
  - Row first
    - **Write** optimized
Multiple traversal strategies are possible with CYTOM

Evaluated options:
- Column first
  - Read optimized
- Row first
  - Write optimized
- Hilbert order
  - Mix between read and write optimization
Previous results

- With **Mosaic**: Hilbert order best for locality and performance
  - Optimize for **reads** and **writes**
- **Mosaic**'s scenario:
  - Batch-processing of large amounts of edges
  - Entire graph being processed at a time
- However: **Cytom**'s scenario is different:
  - Small sets of changes, e.g., one million edges (maximum: 12 MB of vertex and edge data)
  - Only parts of the graph being processed
- For **Cytom**: Optimizing **writes** beneficial!
Traversal strategies with **CYTOM**

Throughput (Million edges/second)

- **Pagerank**
  - Column First
  - Hilbert
  - Row First

$$\Rightarrow$$ Row-first, write-optimized strategy performs up to 80% better
Optimization: Selective scheduling

- Important in evolving graphs:
  - Not all vertices active all the time
- Effect: Many inactive edges
- Common in many algorithms
- Idea: Only operate on cells with active vertices
Optimization: Selective scheduling

- Important in evolving graphs:
  - Not all vertices active all the time
- Effect: Many inactive edges
- Common in many algorithms
- Idea: Only operate on cells with active vertices

![Global adjacency matrix diagram]

**Source vertex**

**Target vertex**
Optimization: Selective scheduling

- Important in evolving graphs:
  - Not all vertices active all the time
- Effect: Many inactive edges
- Common in many algorithms
- Idea: Only operate on cells with active vertices

Target vertex

Source vertex  Global adjacency matrix

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}
\]
Impact of selective scheduling

⇒ Up to $5.6 \times$ improvement due to reduced number cells
What is Cytom’s algorithmic interface?
Example connected components

- Iterative algorithm
- Minimize connected component over incoming edges

Step 1: Initialize with local vertex ID
Example connected components

- Iterative algorithm
- Minimize connected component over incoming edges

Step 2: Minimize over all incoming edges with *Pull*

CC: 1  Iteration: 1

CC: 2

CC: 3

Pull

Pull
Example connected components

- Iterative algorithm
- Minimize connected component over incoming edges

Step 3: Update local connected component value

 Iteration: 1

CC: 1

CC: 1

CC: 2

CC: 2

CC: 1

CC: 1

CC: 1

CC: 2
Example connected components

- Iterative algorithm
- Minimize connected component over incoming edges

Step 4: Run *apply* and mark active vertices
Example connected components

- Iterative algorithm
- Minimize connected component over incoming edges

Step 5: New iteration, minimize with *Pull*

![Diagram showing connected components and iterative algorithm](image)
Example connected components

- Iterative algorithm
- Minimize connected component over incoming edges

Step 6: Run Apply on all active vertices
Example connected components

- Iterative algorithm
- Minimize connected component over incoming edges

Step 7: Convergence reached after one more iteration
Handling updates

- Add new vertex and edge
- Algorithm has to update connected component
Handling updates

- Add new vertex and edge
- Algorithm has to update connected component

Step 1: Add edge $\textcircled{4}$ and vertex $\textcircled{4}$, init connected component
Handling updates

- Add new vertex and edge
- Algorithm has to update connected component

Step 2: *Pull* incoming edge for ④
Handling updates

- Add new vertex and edge
- Algorithm has to update connected component

Step 3: Convergence after next iteration
Dealing with deletions: Critical edges

- Potentially devastating impact on algorithmic result
- Example with connected components:
Dealing with deletions: Critical edges

- Potentially devastating impact on algorithmic result
- Example with connected components:
Dealing with deletions: Critical edges

- Potentially devastating impact on algorithmic result
- Example with connected components:

![Connected Components Diagram]

CC: 1

1

CC: 1

1

CC: 1

1

CC: 1

1

CC: 1

1

CC: 1

1

CC: 1

1

CC: 1

1

CC: 1

1

CC: 1

1
Dealing with deletions: Critical edges

- Potentially devastating impact on algorithmic result
- Example with connected components:

![Diagram showing connected components after deletion]
Dealing with deletions: Critical edges

- Potentially devastating impact on algorithmic result
- Example with connected components:

![Diagram showing connected components](image)

CC: 1

CC: 1

CC: 1
Dealing with deletions: Critical edges

- "Safe" solution: Re-execute algorithm
- Cytom’s approach: Mark *critical* edges carrying results
- If critical edge deleted or updated: Re-execute

![Diagram showing critical edges and their implications](image-url)
Delete non-critical edge ②:
Delete *critical* edge ①:
Re-execute algorithm for correct result:
Programming interface

- Scatter-Gather-Apply (GAS) variant
- Algorithm developer not exposed to subgraph-centric aspects
- Special addition for evolving graphs: `edgeChanged` callback
  - React to edge changes
  - Trigger algorithm re-execution if necessary (e.g., deletions)
Example: Connected components

Edge processing callbacks:

*Edge-centric operation*

```python
// Inside a Cell (local)
// Edge e = (Vertex src, Vertex tgt)
def Pull(Vertex src, Vertex tgt):
    if src.value < tgt.value:
        markCritical(src, tgt)
    return src.value
```

*Vertex-centric operation*

```
// Collecting Cell Results (local & global)
def Reduce(Vertex v1, Vertex v2):
    return min(v1.value, v2.value)

// On Global Vertices
def Apply(Vertex v):
    if current(v.value) != next(v.value):
        activate(v)
```

Local graph processing on Cell

Global graph processing
Example: Connected components

Special for evolving graphs: `edgeChanged` callback

- Filter updates without implied changes
- Handle deletions

```python
1 // On Every Edge Update
2 def EdgeChanged(Vertex src, Vertex tgt, Event e):
3     switch(e):
4         case Inserted:
5             // Update if New Path Discovered
6             if src.value < tgt.value:
7                 tgt.value = src.value
8                 activate(tgt)
9                 markCritical(src, tgt)
10         case Deleted:
11             if isCritical(src, tgt):
12                 reExecuteAlgorithm()
```

Edge-centric operation
Impact of `edgeChanged` callback for insertions

- Benefit: Filter out uninteresting updates

⇒ Up to $2.8 \times$ improvement in overall throughput due to reduced work for algorithm
Impact of `edgeChanged` callback for deletions

Re-execution vs. *critical* edges approach with CC:

⇒ Up to $2.2 \times$ improvement, less when more deletions
Comparison to other systems

- How does Cytom perform compared to other systems?
- Metrics:
  - Throughput of graph updates
  - Overall throughput with updates and algorithm processing
Features offered by each system

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>LLAMA</th>
<th>GraphIn</th>
<th>GraphOne</th>
<th>Stinger</th>
<th>Cytom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremental proc.</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Synchronous proc.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Asynchronous proc.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Approximation</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Disk persistence</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Deletions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>Snapshot view</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>
Throughput for graph updates

- Compared to GraphOne, Cytom offers $2.8 \times$ the throughput
  - Uses CSR for static snapshots, edge lists for new data
  - Log-structured approach
- Compared to Differential Dataflow, Cytom offers $4.8 \times$ the throughput
  - Generalized data flow system
  - Graph support with edge lists
Comparison to GraphIn

- GraphIn uses edge lists
- Only scales to \( \approx 100M \) edges

\[\Rightarrow \text{Up to } 5.5 \times \text{ improvement in throughput}\]
Comparison to **Stinger**

- **Stinger**'s focus: Use atomics to insert into adjacency lists
- Earliest systems of all evaluated ones

⇒ At least $60 \times$ speedup due to more efficient graph updates and APIs
Cytom, a billion-scale engine for evolving graphs

- Cell-based design enables high update rate
- Selective scheduling enabled by cell-based design
- `edgeChanged` callback to quickly react to graph updates
Conclusion

- **Cytom**, a billion-scale engine for evolving graphs
  - Cell-based design enables high update rate
  - Selective scheduling enabled by cell-based design
  - `edgeChanged` callback to quickly react to graph updates

- Additional components of dissertation:
  - **Mosaic**, a trillion-scale engine for static graphs
  - **Latr**, an OS-level approach to reduce overheads of synchronous TLB shootdowns

- Outcome: We can do large-scale data analytics on a single machine
Conclusion

- **Cytom**, a billion-scale engine for evolving graphs
  - Cell-based design enables high update rate
  - Selective scheduling enabled by cell-based design
  - `edgeChanged` callback to quickly react to graph updates

- Additional components of dissertation:
  - **Mosaic**, a trillion-scale engine for static graphs
  - **Latr**, an OS-level approach to reduce overheads of synchronous TLB shootdowns

- Outcome: We can do large-scale data analytics on a single machine

Thanks!