X-Stream: Edge-centric Graph Processing using Streaming Partitions

Amitabha Roy, Ivo Mihailovic, Willy Zwaenepoel
Motivation

- Graph processing is widely used
  - Especially on huge graphs (billions of vertices, trillions of edges)
- Existing systems such as Pregel and PowerGraph have a good programming model
  - Implementation results in many random memory accesses
  - Sequential accesses to edges could result in better performance
Scatter–Gather Programming Model

- Maintain state in vertices
- **Scatter** - propagate vertex state to neighbours
- **Gather** - cumulate updates and recompute state
- Approach is **vertex-centric**

```plaintext
vertex_scatter(vertex v)
    send updates over outgoing edges of v

vertex_gather(vertex v)
    apply updates from inbound edges of v

while not done
    for all vertices v that need to scatter updates
        vertex_scatter(v)
    for all vertices v that have updates
        vertex_gather(v)
```
# Vertex-centric vs Edge-centric

<table>
<thead>
<tr>
<th>Vertex-centric</th>
<th>Edge-centric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterates over vertices</td>
<td>Iterates over edges</td>
</tr>
<tr>
<td>Random accesses of edges</td>
<td>Sequential accesses of edges</td>
</tr>
<tr>
<td>Requires pre-sorting to generate indexed edge list</td>
<td>Requires no pre-sorting</td>
</tr>
<tr>
<td>Edges only used when needed</td>
<td>All edges used per iteration</td>
</tr>
</tbody>
</table>
Edge-centric Model

- Approach is **edge-centric**
- Still maintain state in vertices
- Can stream edges and updates from storage - *no longer random access to edges*
- We now have random access to vertices

```plaintext
edge_scatter(edge e)
    send update over e

edge_gather(update u)
    apply update u to u.destination

while not done
    for all edges e
        edge_scatter(e)
    for all updates u
        update_gather(u)
```
Edge-centric Model

1. Edge Centric Scatter
   Edges (sequential read)
   Vertices (random read/write)
   Updates (sequential write)

2. Edge Centric Gather
   Updates (sequential read)
   Vertices (random read/write)
Streaming Partitions

- Comprised of 3 elements
  - *Vertex set* - a subset of vertices from the graph
  - *Edge list* - all edges where the *source* vertex is in the partition’s vertex set
  - *Update list* - all updates where the *destination* vertex is in the partition’s vertex set
- Vertex set exists in a *cache*
  - Mitigates the random access issue
  - Size must be balanced though
Streaming Partitions

- Must now introduce a new shuffle phase
  - Destination of update may not reside in the same streaming partition
  - Shuffle Phase reorders the updates so this is the case
  - Have $U_{out}$ and $U_{in}$ stream

```
scatter phase:
  for each streaming_partition p
    read in vertex set of p
    for each edge e in edge list of p
      edge_scatter(e): append update to $U_{out}$

shuffle phase:
  for each update u in $U_{out}$
    let p = partition containing target of u
    append u to $U_{in}(p)$
  destroy $U_{out}$

gather phase:
  for each streaming_partition p
    read in vertex set of p
    for each update u in $U_{in}(p)$
      edge_gather(u)
    destroy $U_{in}(p)$
```
## Two scenarios for X-Stream

<table>
<thead>
<tr>
<th>In-memory</th>
<th>Out-of-core</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Fast Storage</em> = CPU cache</td>
<td><em>Fast Storage</em> = Main Memory</td>
</tr>
<tr>
<td><em>Slow Storage</em> = Main Memory</td>
<td><em>Slow Storage</em> = SSD or Hard Disk</td>
</tr>
<tr>
<td>Very limited by size of storage</td>
<td>Huge capacity of storage</td>
</tr>
</tbody>
</table>
In-Memory Specifics

- Cache is much smaller than main memory
  - Results in more streaming partitions needed
- Need to go parallel to reach peak streaming bandwidth
  - Each partition can operate independently
    - Can result in workload imbalance
    - Fix by stealing jobs
Out-of-Core Specifics

- Merge scatter and shuffle phase
  - Only shuffle when output buffer is full

- Use a stream buffer to store updates
  - Makes shuffling more efficient - $O(n)$
  - Requires two stream buffers
    - Input to shuffle
    - Output of shuffle
  - Also used for partitioning
Evaluation

- Sequential Access is better than Random Access on every medium

<table>
<thead>
<tr>
<th>Medium</th>
<th>Read (MB/s)</th>
<th>Write (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Random</td>
<td>Sequential</td>
</tr>
<tr>
<td>RAM (1 core)</td>
<td>567</td>
<td>2605</td>
</tr>
<tr>
<td>RAM (16 cores)</td>
<td>14198</td>
<td>25658</td>
</tr>
<tr>
<td>SSD</td>
<td>22.5</td>
<td>667.69</td>
</tr>
<tr>
<td>Magnetic Disk</td>
<td>0.6</td>
<td>328</td>
</tr>
</tbody>
</table>

Figure 11: Sequential Access vs. Random Access
Evaluation

- Initial results were poor for some datasets
  - The DIMACS and Yahoo web-graph datasets have a wide diameter
  - Results in more scatter-gather iterations as information passes from one end to the other
  - Each iteration requires entire edge list to be streamed; not much useful work each time
Evaluation

- Demonstrates strong scaling with regard to number of threads and I/O Parallelism

Figure 14: Strong Scaling

Figure 15: I/O Parallelism
Evaluation – In Memory

- Compared to Ligra
  - BFS and Pagerank were used to test

- For BFS without preprocessing times
  - Ligra performs better

- Due to sequential access, IPC was much higher

```
<table>
<thead>
<tr>
<th>Threads</th>
<th>Ligra (s)</th>
<th>X-Stream (s)</th>
<th>Ligra-pre (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11.10</td>
<td>168.50</td>
<td>1250.00</td>
</tr>
<tr>
<td>2</td>
<td>5.59</td>
<td>86.97</td>
<td>647.00</td>
</tr>
<tr>
<td>4</td>
<td>2.83</td>
<td>45.12</td>
<td>352.00</td>
</tr>
<tr>
<td>8</td>
<td>1.48</td>
<td>26.68</td>
<td>209.40</td>
</tr>
<tr>
<td>16</td>
<td>0.85</td>
<td>18.48</td>
<td>157.20</td>
</tr>
</tbody>
</table>

| Pagerank |           |              |               |
| 1        | 990.20    | 455.06       | 1264.00       |
| 2        | 510.60    | 241.56       | 654.00        |
| 4        | 269.60    | 129.72       | 355.00        |
| 8        | 145.40    | 83.42        | 211.40        |
| 16       | 79.24     | 50.06        | 160.20        |
```

Figure 20: Ligra [48] on Twitter (99% CI under 5%)

```
<table>
<thead>
<tr>
<th>IPC</th>
<th>BFS [33]</th>
<th>X-Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem refs.</td>
<td>982 million</td>
<td>620 million</td>
</tr>
<tr>
<td>Ligra,BFS [48]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IPC</td>
<td>Ligra-BFS</td>
<td>X-Stream</td>
</tr>
<tr>
<td>Mem refs.</td>
<td>1.3 billion</td>
<td>1.5 billion</td>
</tr>
</tbody>
</table>
```

Figure 21: Instructions per Cycle and Total Number of Memory References for BFS
Evaluation - Out-of-Core

- Compared to GraphChi

- GraphChi needs to resort edges by destination vertex before applying updates (reported as re-sort)
  - Accounts for significant proportion of execution

- In most cases X-Stream completed before GraphChi Presorted

- Disk bandwidth of X-Stream is much greater
Strengths and Weaknesses

**Strengths**

- Utilises sequential access to effectively maximise streaming bandwidth
- Performs quicker in general than most state of the art systems on some algorithms

**Weaknesses**

- If no pre-processing is required for the vertex-centric approach, performance can be weaker
- Performance is poor on a graph with a high diameter
- “Wasted Edges” also negatively impacts performance
Criticism

- Little focus on running X-Stream on multiple machines
  - See next slide
- Assumes graph has many more edges than vertices
- No discussion on the limitations of the programming model
Impact

- Nearly 450 citations
- Authors extended X-Stream with Chaos
  - Chaos essentially is the distributed version of X-Stream
Questions