GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning

Xiaowei ZHU
Tsinghua University
Widely-Used Graph Processing
Existing Solutions

• Shared memory
  – Single-node & in-memory
  – Ligra, Galois, Polymer

• Distributed
  – Multi-node & in-memory
  – GraphLab, GraphX, PowerLyra

• Out-of-core
  – Single-node & disk-based
  – GraphChi, X-Stream, TurboGraph
Existing Solutions

• Shared memory
  – Single-node & in-memory
  – Ligra, Galois, Polymer

• Distributed
  – Multi-node & in-memory
  – GraphLab, GraphX, PowerLyra

• Out-of-core
  – Single-node & disk-based
  – GraphChi, X-Stream, TurboGraph

Large-scale
Limited capability to big graphs

Irregular structure
Imbalance of computation and communication

Inevitable random access
Expensive disk random access
Existing Solutions

• Shared memory
  – Single-node & in-memory
  – Ligra, Galois, Polymer

• Distributed
  – Multi-node & in-memory
  – GraphLab, GraphX, PowerLyra

• Out-of-core
  – Single-node & disk-based
  – GraphChi, X-Stream, TurboGraph

Most cost effective!

Large-scale
Limited capability to big graphs

Irregular structure
Balance of computation and communication

Inevitable random access
Expensive disk random access
Methodology

• How to handle graphs that is much larger than memory capacity?
  – Partition!

<table>
<thead>
<tr>
<th>System</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphChi</td>
<td>Shards</td>
</tr>
<tr>
<td>TurboGraph</td>
<td>Page</td>
</tr>
<tr>
<td>X-Stream</td>
<td>Streaming Partitions</td>
</tr>
<tr>
<td>PathGraph</td>
<td>Tree-Based Partitions</td>
</tr>
<tr>
<td>FlashGraph</td>
<td>Page</td>
</tr>
<tr>
<td>GridGraph</td>
<td>Chunks &amp; Blocks</td>
</tr>
</tbody>
</table>
State-of-the-Art Methodology

• X-Stream
  – Access edges *sequentially* from *disks*
  – Access vertices *randomly* inside *memory*
    • Guarantee locality of vertex accesses by partitioning
State-of-the-Art Methodology

• X-Stream
  – Access edges *sequentially* from *slow* memory
  – Access vertices *randomly* inside *fast* memory
    • Guarantee locality of vertex accesses by partitioning
**Edge-Centric Scatter-Gather**

1. **Edge Centric Scatter**
   - Edges (sequential read)
   - Scatter:
     - for each streaming partition
       - load **source** vertex chunk of edges into fast memory
       - stream **edges**
       - append to several **updates**

2. **Edge Centric Gather**
   - Updates (sequential write)
   - Gather:
     - for each streaming partition
       - load **destination** vertex chunk of updates into fast memory
       - stream **updates**
       - apply to vertices

---

X-Stream: Edge-centric Graph Processing using Streaming Partitions, A. Roy et al., SOSP 2013
Question: Is it possible to apply on-the-fly updates? (Thus bypass the writes and reads of updates.)

Can be as large as $O(E)$!
Basic Idea

Answer: Guarantee the locality of both source and destination vertices when streaming edges!

Streaming-Apply:
for each streaming edge block
load source and destination vertex chunk of edges into memory
stream edges
read from source vertices
write to destination vertices
Solution

• Grid representation
  – Dual sliding windows
  – Selective scheduling
• 2-level hierarchical partitioning
Grid Representation

- Vertices partitioned into $P$ equalized chunks
- Edges partitioned into $P \times P$ blocks
  - Row $\leftrightarrow$ source
  - Column $\leftrightarrow$ destination

```
  1  2  3  4

  1  | (1, 2) (2, 1) (1, 3) (2, 4) (3, 2) (4, 2) (4, 3)
  2  |  
  3  |  
  4  |  

Source Chunk 2

Destination Chunk 1

Edge Block(2, 1)
```

$P=2$
Streaming-Apply Processing Model

- Stream edges block by block
  - Each block corresponding to two vertex chunks
    - Source chunk + destination chunk
    - Fit into memory

- Difference with scatter-gather
  - 2 phases → 1 phase
  - Updates are applied on-the-fly
Dual Sliding Windows

• Access edge blocks in column-oriented order
  – From left to right
    – Destination window slides as column moves
  • From top to bottom
    – Source window slides as row moves
  – Optimize write amount
    • 1 pass over the destination vertices
Dual Sliding Windows

\[ \text{PageRank}_i = (1 - d) + d \times \sum_{j \in N_{\text{in}}(i)} \frac{\text{PageRank}_j}{\text{OutDegree}_j} \]
Dual Sliding Windows

Stream Block (1, 1)

Cache source vertex chunk 1 in memory (miss);
Cache destination vertex chunk 1 in memory (miss);
Read edges (1, 2), (2, 1) from disk;

<table>
<thead>
<tr>
<th>Object</th>
<th>I/O Amt.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edges</td>
<td>0 → 2</td>
</tr>
<tr>
<td>Src. vertex</td>
<td>0 → 2</td>
</tr>
<tr>
<td>Dest. vertex</td>
<td>0 → 2</td>
</tr>
</tbody>
</table>

Graph:

- Vertices: 1, 2, 3, 4
- Edges: (1, 2), (2, 1), (2, 3), (3, 4), (4, 1)
- Edges weights: 1, 2, 0.5
- Time blocks:
  - Block 1: 1, 2, 0.5
  - Block 2: 1, 1, 0
- Bandwidth is calculated based on the number of edges coming into each vertex.
Dual Sliding Windows

Cache source vertex chunk 2 in memory (miss);
Cache destination vertex chunk 1 in memory (hit);
Read edges (3, 2), (4, 2) from disk;

Stream Block (2, 1)
Dual Sliding Windows

![Graph](image)

**Stream Block (1, 2)**

<table>
<thead>
<tr>
<th>PR</th>
<th>Deg</th>
<th>NewPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>(1, 2)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>(2, 1)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(3, 2)</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>(4, 2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object</th>
<th>I/O Amt.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edges</td>
<td>4 → 6</td>
</tr>
<tr>
<td>Src. vertex</td>
<td>4 → 6</td>
</tr>
<tr>
<td>Dest. vertex</td>
<td>2 → 6</td>
</tr>
</tbody>
</table>

Cache source vertex chunk 1 in **memory (miss)**;
Cache destination vertex chunk 2 in **memory (miss)**;
Write back destination vertex chunk 1 to **disk**;
Read edges (1, 3), (2, 4) from **disk**;
Dual Sliding Windows

Cache source vertex chunk 2 in memory (miss);
Cache destination vertex chunk 2 in memory (hit);
Read edges (4, 3) from disk;
Dual Sliding Windows

![Graph Diagram]

**P=2**

<table>
<thead>
<tr>
<th>PR</th>
<th>Deg</th>
<th>NewPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>(1, 2)</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>(2, 1)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>(3, 2)</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>(4, 2)</td>
</tr>
</tbody>
</table>

Iteration 1 finishes

Write back destination vertex chunk 2 to **disk**;

**Table**

<table>
<thead>
<tr>
<th>Object</th>
<th>I/O Amt.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edges</td>
<td>7</td>
</tr>
<tr>
<td>Src. vertex</td>
<td>8</td>
</tr>
<tr>
<td>Dest. vertex</td>
<td>6 → 8</td>
</tr>
</tbody>
</table>
I/O Access Amount

• For 1 iteration

\[ E + (2 + P) \times V \]

1 pass over the edges (read)

P pass over the source vertices (read)

1 pass over the destination vertices (read+write)

Implication: P should be the minimum value that enables needed vertex data to be fit into memory.
I/O Access Amount

• For 1 iteration

\[ E + (2 + P) \times V \]

1 pass over the edges (read)

P pass over the source vertices (read)

1 pass over the destination vertices (read+write)

Implication: P should be the minimum value that enables needed vertex data to be fit into memory.
Memory Access Amount

• For 1 iteration

\[ E + (2 + P) \times V \]

1 pass over the edges (read)

P pass over the source vertices (read)

1 pass over the destination vertices (read+write)

Implication: P should be the minimum value that enables needed vertex data to be fit into memory.
Selective Scheduling

• Skip blocks with no active edges
  – Very simple but important optimization
  – Effective for lots of algorithms
    • BFS, WCC, ...
Selective Scheduling

BFS from 1 with P = 2

Before

<table>
<thead>
<tr>
<th>Active</th>
<th>False</th>
<th>True</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Access 4 edges

Iteration 1

<table>
<thead>
<tr>
<th>(1, 2)</th>
<th>(1, 3)</th>
<th>(2, 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2, 1)</td>
<td>(2, 4)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(3, 2)</th>
<th>(4, 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4, 2)</td>
<td></td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>Active</th>
<th>False</th>
<th>True</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Selective Scheduling

BFS from 1 with P = 2

<table>
<thead>
<tr>
<th>Active</th>
<th>False</th>
<th>True</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Before

Iteration 2

Access 7 edges

<table>
<thead>
<tr>
<th>Active</th>
<th>False</th>
<th>False</th>
<th>False</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

After
Selective Scheduling

BFS from 1 with P = 2

**Before**

<table>
<thead>
<tr>
<th>Parent</th>
<th>Active</th>
<th>(1, 2)</th>
<th>(1, 3)</th>
<th>(2, 1)</th>
<th>(2, 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>4</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

**Access 3 edges**

<table>
<thead>
<tr>
<th>Parent</th>
<th>(1, 2)</th>
<th>(1, 3)</th>
<th>(2, 1)</th>
<th>(2, 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1, 2)</td>
<td>(1, 3)</td>
<td>(2, 1)</td>
<td>(2, 4)</td>
</tr>
<tr>
<td>2</td>
<td>(1, 2)</td>
<td>(1, 3)</td>
<td>(2, 1)</td>
<td>(2, 4)</td>
</tr>
<tr>
<td>3</td>
<td>(1, 2)</td>
<td>(1, 3)</td>
<td>(2, 1)</td>
<td>(2, 4)</td>
</tr>
<tr>
<td>4</td>
<td>(1, 2)</td>
<td>(1, 3)</td>
<td>(2, 1)</td>
<td>(2, 4)</td>
</tr>
</tbody>
</table>

**Iteration 3**

**After**

<table>
<thead>
<tr>
<th>Parent</th>
<th>(1, 2)</th>
<th>(1, 3)</th>
<th>(2, 1)</th>
<th>(2, 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1, 2)</td>
<td>(1, 3)</td>
<td>(2, 1)</td>
<td>(2, 4)</td>
</tr>
<tr>
<td>2</td>
<td>(1, 2)</td>
<td>(1, 3)</td>
<td>(2, 1)</td>
<td>(2, 4)</td>
</tr>
<tr>
<td>3</td>
<td>(1, 2)</td>
<td>(1, 3)</td>
<td>(2, 1)</td>
<td>(2, 4)</td>
</tr>
<tr>
<td>4</td>
<td>(1, 2)</td>
<td>(1, 3)</td>
<td>(2, 1)</td>
<td>(2, 4)</td>
</tr>
</tbody>
</table>
Selective Scheduling

BFS from 1 with P = 2

1

2

3

4

BFS finishes

Parent | 1 | 1 | 1 | 2
---|---|---|---|---
| (1, 2) | (1, 3) |
| (2, 1) | (2, 4) |
| (3, 2) | (4, 3) |

4+7+3=14
Access 14 edges in all
Impact of P on Selective Scheduling

Effect becomes better with more fine-grained partitioning.

Implication: A larger value of P is preferred.

<table>
<thead>
<tr>
<th>P</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge accesses</td>
<td>21 (=7+7+7)</td>
<td>14 = (4+7+3)</td>
<td>7 = (2+3+2)</td>
</tr>
</tbody>
</table>
Dilemma on Selection of $P$

- **Coarse-grained**
  - Fewer accesses on vertices
  - Poorer locality
  - Less selective scheduling

- **Fine-grained**
  - Better locality
  - More selective scheduling
  - More accesses on vertices
Dilemma from Memory Hierarchy

• Different selections of P
  – Disk – Memory hierarchy
    • Fit hot vertex data into memory
  – Memory – Cache hierarchy
    • Fit hot vertex data into cache
  – Disk – Memory – Cache
    • ?
2-Level Hierarchical Partitioning

• Apply a $Q \times Q$ partitioning over the $P \times P$ grid
  • $Q \geq V / M$
  • $P \geq V / C$
  • $C << M$
  • $P >> Q$

− Group the small blocks into larger ones

$P = \text{number of partitions, } M = \text{memory capacity, } C = \text{LLC capacity, } V = \text{size of vertices}$
Programming Interface

- \textbf{StreamVertices}(F_v, F)

- \textbf{StreamEdges}(F_e, F)

---

\textbf{Algorithm 3} PageRank

\begin{algorithm}
\begin{algorithmic}
\Function{CONTRIBUTE}{e}
\State Accum(\&NewPR[e.dest], \ \frac{PR[e.source]}{Deg[e.source]})
\EndFunction
\Function{COMPUTE}{v}
\State NewPR[v] = 1 - d + d \times NewPR[v]
\State \Return |NewPR[v] - PR[v]|
\EndFunction
\State d = 0.85
\State PR = \{1, \ldots, 1\}
\State Converged = 0
\While{\neg Converged}
\State NewPR = \{0, \ldots, 0\}
\State StreamEdges(Contribute)
\State \underline{Diff} = StreamVertices(Compute)
\State Swap(PR, NewPR)
\State Converged = \frac{\text{Diff}}{\sqrt{V}} \leq \text{Threshold}
\EndWhile
\end{algorithmic}
\end{algorithm}
Evaluation

• Test environment
  – AWS EC2 i2.xlarge
    • 4 hyperthread cores
    • 30.5GB memory
    • 1 × 800GB SSD
  – AWS EC2 d2.xlarge
    • 4 hyperthread cores
    • 30.5GB memory
    • 3 × 2TB HDD

• Applications
  – BFS, WCC, SpMV, PageRank

<table>
<thead>
<tr>
<th>Dataset</th>
<th>V</th>
<th>E</th>
<th>Data size</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJournal</td>
<td>4.85M</td>
<td>69.0M</td>
<td>527MB</td>
<td>4</td>
</tr>
<tr>
<td>Twitter</td>
<td>61.6M</td>
<td>1.47B</td>
<td>11GB</td>
<td>32</td>
</tr>
<tr>
<td>UK</td>
<td>106M</td>
<td>3.74B</td>
<td>28GB</td>
<td>64</td>
</tr>
<tr>
<td>Yahoo</td>
<td>1.41B</td>
<td>6.64B</td>
<td>50GB</td>
<td>512</td>
</tr>
</tbody>
</table>
LiveJournal

Twitter

UK

Yahoo

<table>
<thead>
<tr>
<th>Runtime(S)</th>
<th>BFS</th>
<th>WCC</th>
<th>SpMV</th>
<th>PageRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>GraphChi</td>
<td>-</td>
<td>114162</td>
<td>2676</td>
<td>13076</td>
</tr>
<tr>
<td>X-Stream</td>
<td>-</td>
<td>-</td>
<td>1076</td>
<td>9957</td>
</tr>
<tr>
<td>GridGraph</td>
<td>16815</td>
<td>3602</td>
<td>263.1</td>
<td>4719</td>
</tr>
</tbody>
</table>

“-” indicates failing to finish in 48 hours

i2.xlarge, memory limited to 8GB
Disk Bandwidth Usage

I/O throughput of a 10-minute interval running PageRank on Yahoo graph
Effect of Dual Sliding Windows

PageRank on Yahoo

[Bar chart showing I/O Amount for Reads and Writes for GraphChi, X-Stream, and GridGraph]
Effect of Selective Scheduling

![Graph showing the effect of selective scheduling on X-Stream and GridGraph]

WCC on Twitter
Impact of P on In-Memory Performance

PageRank on Twitter
Edges cached in 30.5GB memory
Optimal around P=32 where needed vertex data can be fit into L3 cache.
Impact of Q on Out-of-Core Performance

SpMV on Yahoo
Memory limited to 8GB
Optimal at Q=2 where needed vertex data can be fit into memory.
Comparison with Distributed Systems

- PowerGraph, GraphX *
  - 16 × m2.4xlarge, $15.98/h

- GridGraph
  - i2.4xlarge, $3.41/h
    - 4 SSDs
    - 1.8GB/s disk bandwidth

* GraphX: Graph Processing in a Distributed Dataflow Framework, JE Gonzalez et al., OSDI 2014
Conclusion

• GridGraph
  – Dual sliding windows
    • Reduce I/O amount, especially writes
  – Selective scheduling
    • Reduce unnecessary I/O
  – 2-level hierarchical grid partitioning
    • Applicable to 3-level (cache-memory-disk) hierarchy
Thanks!