Large-Scale Graph Processing on Emerging Storage Devices

Nima Elyasi\textsuperscript{1}, Changho Choi\textsuperscript{2}, Anand Sivasubramaniam\textsuperscript{1}

\textsuperscript{1}Pennsylvania State University

\textsuperscript{2}Samsung Semiconductor Inc.
Graph Processing is Commonplace

- Search Engines
- Social Media
- Recommendations and Ads
- Map and Navigation

Logos for:
- Google
- Facebook
- Amazon
- Bing
- Twitter
- Netflix
- Map and Navigation Icon
Large-Scale Graph Processing Challenges

- Huge Datasets
- Irregular Accesses

High cost of DRAM

$\text{DRAM}$
Large-Scale Graph Processing Challenges

- Huge Datasets
- Irregular Accesses
- High cost of DRAM
- External Graph Processing is Desirable

DRAM

$$$$$

External Graph Processing is Desirable

NVMe SSD

$
Large-Scale Graph Processing Challenges

Huge Datasets

Irregular Accesses

High cost of DRAM

External Graph Processing is Desirable

$\text{DRAM}$

$\text{Irregular Accesses}$

$\text{Vertex List}$

$\text{NVMe SSD}$
Large-Scale Graph Processing Challenges

- Huge Datasets
- Irregular Accesses
- External Graph Processing is Desirable
- High cost of DRAM
- Fine-Grained and Random Accesses

DRAM

$\text{NVM} \text{e SSD}$
Fine-Grained Access in External Graph Processing

Irregular Accesses

SSD Page Size and Vertex Accesses Don’t Match!

SSD Page

Several KiloBytes
(4KB ~ 16KB)

Vertex Value

Several Bytes, e.g., 4Bytes

SSD Page 0

SSD Page 1

Vertex List
SSD Page Size and Vertex Accesses Don’t Match!

Irregular Accesses

SSD Page

Several KiloBytes (4KB ~ 16KB)

Vertex Value

Several Bytes, e.g., 4Bytes

Vertex updates are detrimental to:

Performance

Device Endurance
Providing Perfect Sequentiality as a Remedy

- If vertex data could be stored on DRAM
  - Fine-grained accesses was less of an issue

Instead, prior external graph processing framework maintains vertex data on SSD

GraFBoost, ISCA'18
Providing Perfect Sequentiality as a Remedy

- If vertex data could be stored on DRAM
  - Fine-grained accesses was less of an issue

Instead, prior external graph processing framework maintains vertex data on SSD

Achieves perfect sequentiality by coalescing fine-grained accesses
Vertex-centric Programming Model

- Iterative programming model
- Each vertex runs a user-defined program
- Sending updates to neighbors along outgoing edges
Prior External Graph Processing -- GraFBoost

Vertex Data

- Vertex 0: {Vertex ID, Value}
- Vertex 1: {Vertex ID, Value}
- Vertex 2: {Vertex ID, Value}
- Vertex 3: {Vertex ID, Value}
- Vertex 4: {Vertex ID, Value}

Index File

- Vertex 0 Offset
- Vertex 1 Offset
- Vertex 2 Offset
- Vertex 3 Offset
- Vertex 4 Offset

Edge File

- Vertex 0 Out Edge
- Vertex 0 Out Edge
- Vertex 0 Out Edge
- Vertex 1 Out Edge
- Vertex 1 Out Edge
- Vertex 2 Out Edge

Prior External Graph Processing -- GraFBoost

**Vertex Data**
- Vertex 0: {Vertex ID, Value}
- Vertex 1: {Vertex ID, Value}
- Vertex 2: {Vertex ID, Value}
- Vertex 3: {Vertex ID, Value}
- Vertex 4: {Vertex ID, Value}

**Index File**
- Vertex 0 Offset
- Vertex 1 Offset
- Vertex 2 Offset
- Vertex 3 Offset
- Vertex 4 Offset

**Edge File**
- Vertex 0 Out Edge
- Vertex 0 Out Edge
- Vertex 0 Out Edge
- Vertex 1 Out Edge
- Vertex 1 Out Edge
- Vertex 2 Out Edge

**Diagrams**
- $V_0 \rightarrow V_x$
- $V_0 \rightarrow V_y$
- $V_0 \rightarrow V_z$

Prior External Graph Processing -- GraFBoost

Vertex Data
- Vertex 0: {Vertex ID, Value}
- Vertex 1: {Vertex ID, Value}
- Vertex 2: {Vertex ID, Value}
- Vertex 3: {Vertex ID, Value}
- Vertex 4: {Vertex ID, Value}

Index File
- Vertex 0 Offset
- Vertex 1 Offset
- Vertex 2 Offset
- Vertex 3 Offset
- Vertex 4 Offset

Edge File
- Vertex 0 Out Edge
- Vertex 0 Out Edge
- Vertex 0 Out Edge
- Vertex 1 Out Edge
- Vertex 1 Out Edge
- Vertex 2 Out Edge

Vertex Data

Keys: \{V_x, V_y, V_z\}, Value: \{V_0 \text{ value}\}

\langle V_x, V_0 \text{ value}\rangle, \langle V_y, V_0 \text{ value}\rangle, \langle V_z, V_0 \text{ value}\rangle
Prior External Graph Processing -- GraFBoost

**Vertex Data**
- Vertex 0
  - `{Vertex ID, Value}`
  - `{Vertex ID, Value}`
- Vertex 1
  - `{Vertex ID, Value}`
- Vertex 2
  - `{Vertex ID, Value}`
- Vertex 3
  - `{Vertex ID, Value}`
- Vertex 4
  - `{Vertex ID, Value}`

**Index File**
- Vertex 0 Offset
- Vertex 1 Offset
- Vertex 2 Offset
- Vertex 3 Offset
- Vertex 4 Offset

**Edge File**
- Vertex 0 Out Edge
- Vertex 0 Out Edge
- Vertex 0 Out Edge
- Vertex 1 Out Edge
- Vertex 1 Out Edge
- Vertex 2 Out Edge

**Vertex Data**
- $V_0 \rightarrow V_x$
- $V_0 \rightarrow V_y$
- $V_0 \rightarrow V_z$

**Keys:** {$V_x$, $V_y$, $V_z$}, **Value:** {$V_0$ value}

**<V_x, V_0 value>, <V_y, V_0 value>, <V_z, V_0 value>**

GraFBoost sorts key-value pairs in memory, logs them in SSD, merges them, and updates vertex list in SSD.

Computation Overhead of Sort!

- Up to 60% sort overhead (web graph)

- Higher sort overhead for PageRank
  - Processes all vertices in each iteration and generates more updates
Scalability Issue

Current External Graph Processing:

- **Read from SSD**: Linear Time $O(|E|)$
- **Sort in Memory**: $|E|\times\log(|E|)$
- **Write to SSD**: Linear Time $O(|E|)$
Current External Graph Processing:

- **Read from SSD**: Linear Time $O(|E|)$
- **Sort in Memory**: $|E|\times\log(|E|)$
- **Write to SSD**: Linear Time $O(|E|)$

Assuming DRAM “$k$” times faster than SSD (e.g., $k=30$):

- When $k < \log(|E|) \Rightarrow$ Sorting can become bottleneck
Current External Graph Processing:

- Read from SSD: Linear Time $O(|E|)$
- Sort in Memory: $|E| \times \log(|E|)$
- Write to SSD: Linear Time $O(|E|)$

Assuming DRAM "k" times faster than SSD (e.g., k=30):

When $k \times \log(|E|) \rightarrow$ Sorting can become bottleneck
Scalability Issue

Current External Graph Processing:

- Read from SSD: Linear Time $O(|E|)$
- Sort in Memory: $|E| \cdot \log(|E|)$
- Write to SSD: Linear Time $O(|E|)$

Assuming DRAM "k" times faster than SSD (e.g., k=30):

When $k < \log(|E|)$ → Sorting can become bottleneck

Instead, we propose a vertex partitioning to eliminate the sorting
Extensive Prior Efforts on Partitioning Graph Data:

- Not well suited for fully external graph processing

Require all vertices be present in main memory

Do not decouple vertices and edges

Need each partition be completely present in cache or memory

Dramatically increasing number of partitions and incurring high cross-partition communication

FlashGraph, FAST'15
GraphChi, OSDI'12,
Mosaic, EuroSys'17

PowerGraph, OSDI'12
GridGraph, USENIX ATC'15
GraphP, HPCA'18
Instead, We Propose a Partitioning for Vertex Data

Reorganizing graph data so that vertices associated with each partition can fit in main memory
Instead, We Propose a Partitioning for Vertex Data

Reorganizing graph data so that vertices associated with each partition can fit in main memory

Source Vertex Data

<table>
<thead>
<tr>
<th>Partition 0</th>
<th>Partition 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex 0</td>
<td>Vertex 3</td>
</tr>
<tr>
<td>Vertex 1</td>
<td>Vertex 4</td>
</tr>
<tr>
<td>Vertex 2</td>
<td>Vertex 5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Destination Vertex Data

<table>
<thead>
<tr>
<th>Source Vertex Data</th>
<th>Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex A</td>
<td>Offset A</td>
</tr>
<tr>
<td>Vertex B</td>
<td>Offset B</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Vertex C</td>
<td>Offset C</td>
</tr>
</tbody>
</table>

Edge Data

<table>
<thead>
<tr>
<th>Out-edge</th>
<th>Out-edge</th>
<th>Out-edge</th>
<th>Out-edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex A</td>
<td>Vertex A</td>
<td>Vertex B</td>
<td>Vertex C</td>
</tr>
</tbody>
</table>
Execution Flow

In each iteration:

- Memory
  - Vertex Data
  - Destination Vertex for a partition

SSD

Partition 0
- Vertex 0
- Vertex 1
- Vertex 2
...
In each iteration:

Memory
Vertex Data

Source Vertex Data
Vertex ID & Value
Offset A
Offset B
Offset C

Partition 0
Vertex 0
Vertex 1
Vertex 2

SSD
A Chunk of Source Vertex (32MB)

Update Destination Vertices
Reading Neighboring Information

Destination Vertex for a partition

Execution Flow
In each iteration:

**Execution Flow**

- **Source Vertex Data**
  - Vertex ID & Value
  - Index
  - Vertex A: Offset A
  - Vertex B: Offset B
  - Vertex C: Offset C

- **Partition 0**
  - Vertex 0
  - Vertex 1
  - Vertex 2
  - ...
In each iteration:

Execution Flow

Memory

Vertex Data

Update Destination Vertices

A Chunk of Source Vertex (32MB)

Reading Neighboring Information

Generate Mirror Updates for other partitions

SSD

Write all updated vertex data on SSD

Update Destination Vertices

How to Update Vertex List in Main Memory?

Partition 0

Vertex 0 | Vertex 1 | Vertex 2

Vertex ID & Value | Index

Vertex A | Offset A
Vertex B | Offset B
Vertex C | Offset C
Multiple threads are updating elements of the same vertex list
- High synchronization cost

Vertex List
Multiple threads are updating elements of the same vertex list
- High synchronization cost

Vertex List

Buffer, e.g., 1MB
Required Meta-Data for Mirror Updates

**Metadata**

For each Vertex
- Vertex 0 Part ID(s)
- Vertex 1 Part ID(s)
- Vertex 2 Part ID(s)
-...

For each partition
- Partition 0 Start Index End Index
- Partition i Start Index End Index
-...

**Source Vertex Table**

Partition $i$
- Start Index
- Mirrors for Partition 0
- End Index

$O(|V|)$ running time for updating mirrors
Experimental Setup

- **Processor**: Intel Xeon -- 48 Cores

- **Memory**: DRAM – 256 GB

- **SSD**: Two Samsung NVMe SSDs
  - 3.2 TB capacity in total, and 6.4 GB/s Sequential Read Speed

- **Graph Algorithms**:
  - PageRank and Breadth-First-Search (BFS)

- **Input Graphs**:
  - Web, Twitter, Synthetic (Kron)
Performance Evaluation

- More than 2X Improvement Compared to GrafSoft
- Providing Higher Benefits for larger graphs (Web, Kron32)
- Incurring around 10% space overhead for partitioning
Execution Time Breakdown

- Mirror updates account for 8-12% of execution time

- I/O does not remain the main contributor to the total execution time
Concluding Remarks

• Large-scale graph processing suffers from random updates to vertices

• State-of-the-art provides perfect sequentiality by sorting all updates
  - High computation overhead

• A partitioning for vertex data is proposed to eliminate the need for perfect sequentiality

• In Future: Addressing timely evolving graphs

• Thanks to GraFboost authors (Sang-Woo Jun)!
Thanks!