ChunkGraph: Large Graph Processing with
Chunk-Based Graph Representation Model

Rui Wang, Weixu Zong, Shuibing He, Xinyu Chen, Zhenxin Li, Zheng Dang

Zhejiang University

USENIX ATC 2024
Explosive Growth in Graph Data Analytics

- Graph data

- Real-world graph datasets are continuously growing

- Graph Application
  - Social networks, webpage links, recommendation systems

#Facebook monthly active users between 2010 and 2023, Statista
Different Graph Systems Supporting Large Graph Processing

- **Objectives** for large graph processing systems:
  - Cost-effective, Scalable, Programming-friendly

Scalability vs. Price diagram:

- **Out-of-Core system**
  - X-Stream, FlashGraph, Graphene, Blaze

- **Memory-storage cache subsystems**
  - mmap-based graph system, TriCache

- **Distributed system**
  - PowerGraph, G-Miner, LiveGraph

- **In-memory system**
  - Ligra, Galois, GraphOne

**Our concerns**

Extra communication overhead

Limited capacity
SOTA Subgraph-based Out-of-core Graph Systems

- **Subgraph-based iterative model** divides the whole graph into **disjoint** intervals
  - Sequentially load each subgraph from disk during each iteration (e.g. access \(v_1, v_3\))

```
 subgraph \(g_0\)

```

```
 subgraph \(g_1\)

```

```
 subgraph \(g_2\)

```

Load and compute subgraph \(g_0, g_1\) sequentially and iteratively

```
<table>
<thead>
<tr>
<th></th>
<th>(v_0)</th>
<th>(v_1)</th>
<th>(v_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th></th>
<th>(v_0)</th>
<th>(v_1)</th>
<th>(v_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th></th>
<th>(v_0)</th>
<th>(v_1)</th>
<th>(v_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>subgraph (g_0)</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>subgraph (g_1)</td>
<td>0</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>subgraph (g_2)</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Merge multiple random I/O ops into fewer sequential I/O ops
Limitations for Subgraph-based Iterative Model Graph Systems

**P1 Low I/O efficiency**
Access single vertex → load whole subgraph

**P2 Extra computing overhead**
Subgraph synchronization overhead

**P3 Expensive algorithm development costs**
Extra implementation for I/O management

Average I/O utilization is lower than 13% for BFS.

Blaze requires up to 154x more CPU instructions compared to in-memory system Ligra’s mmap variant.

<table>
<thead>
<tr>
<th>#line of codes</th>
<th>BFS</th>
<th>BC</th>
<th>PageRank</th>
<th>KCore</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ligra</td>
<td>34</td>
<td>123</td>
<td>77</td>
<td>86</td>
</tr>
<tr>
<td>Graphene</td>
<td>420</td>
<td>-</td>
<td>763</td>
<td>476</td>
</tr>
<tr>
<td>Blaze</td>
<td>75</td>
<td>197</td>
<td>159</td>
<td>133</td>
</tr>
</tbody>
</table>
Alternative: Memory-Storage Cache Subsystems

- Using **page cache based mechanism** to cache data from external storage

The diagram illustrates a graph with several subgraphs: $g_0$, $g_1$, and $g_2$. The vertex metadata (degree and pointer to adjacency list) is shown in the Memory (Cache) section. The adjacency list is also depicted on the Disk.

Load and cache the pages that are needed by graph computing.

Fine-grained for page access compared to subgraph access.
Limitations for Memory-Storage Cache Subsystems

**Problem 1:** Mismatch between page granularity and vertex access

- Real graph datasets behave power law degree distribution

Low-degree vertices \(\rightarrow\) Poor I/O efficiency

- 51.17% of non-sink vertices have only one or two in-neighbors

High-degree vertices \(\rightarrow\) Massive page table entries

- 0.09% of non-sink vertices account for 58.44% of total edges
- 7459 4KB-pages are needed for largest vertex’s neighbors

**Solution:** Use different storing strategy for vertices with different degrees
Limitations for Memory-Storage Cache Subsystems

- **Problem 2: Vertex cut**
  - A vertex smaller than one page is placed across two adjacent pages (e.g. $v_3, v_5$)

<table>
<thead>
<tr>
<th>Disk</th>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
<th>Page 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_0</td>
<td>1 2 3 4 2</td>
<td>0 6 4 6 7 5 6 8 4 7 8 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v_4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v_5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>adjacency list</td>
</tr>
<tr>
<td>v_6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v_7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v_8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  ($v_i$ indicate the adjacency list of vertex $i$)

  - CSR format: access 3 Pages
    - Access $v_1, v_3$
  - Best case: access 1 page

  | 1 3 4 2 0 6 4 6 7 5 6 6 2 4 6 7 |

  **Target:** Minimize the number of page accesses for each query
ChunkGraph: I/O efficient chunk-based graph representation model

T1. Classified and hierarchical vertex storage

T2. Chunk layout optimization

T3. Differentiated chunk access
Technique 1.1: Classified Hierarchical Vertex Storage

- All vertices are classified into **three categories** according to their degrees
  - **Mini vertex**: in-index storing without additional storage cost and indirect addressing
  - **Medium vertex**: chunk-based storing **without vertex cutting issues**
  - **Super vertex**: HugePage-based storing with lower page table and TLB overhead

---

**Mini vertex**

- \( d \in [1, 2] \)

<table>
<thead>
<tr>
<th>deg</th>
<th>nb_0</th>
<th>nb_1</th>
</tr>
</thead>
</table>

**Medium vertex**

- \( d \in [3, d_0] \)

<table>
<thead>
<tr>
<th>deg</th>
<th>cid</th>
<th>coff</th>
</tr>
</thead>
</table>

**Super vertex**

- \( d \in (d_0, \infty) \)

<table>
<thead>
<tr>
<th>deg</th>
<th>sv_off</th>
</tr>
</thead>
</table>

---

**Chunk Structure**

- \( c_i \): The \( i \)'th chunk of level \( L \)
- \( b_j \): The \( j \)'th vertex of level \( L \)

---

**Hierarchical Chunks**

- \( c_0, c_1, c_2, c_3, c_4, c_5, \ldots \)

---

**Super Vertices HugePage Region**

- \( b_0^{sv}, b_1^{sv}, \ldots \)
**Technique 1.2: Hierarchical Chunk Management**

- **Hierarchical** chunk size according to medium vertices’ degree
  - E.g. a four-layer hierarchical chunk implementation

  - The i’th chunk of level L
  - The j’th vertex of level L

  - L0 4KB chunks
  - L1 32KB chunks
  - L2 256KB chunks
  - L3 2MB chunks

  - Differentiated chunk buffer sizes according to each layer proportion

  \[ S_i = \frac{S_i \times M}{\sum_1^L S_j} \]
  - \( S_i \): chunk file size of layer i
  - M: total available memory size
Technique 2.1: Reordering based Chunk Layout Optimization

- Observation: A vertex is likely to be accessed after its neighbors or sibling vertices accessed
  - E.g. Run BFS on root 3. (Vertex access order: 3, 1, 5, 7, 4, 6, 0, 2)

(a) Current vertex-id based chunk layout

<table>
<thead>
<tr>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
<th>Page 4</th>
<th>Page 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_0)</td>
<td>(v_1)</td>
<td>(v_3)</td>
<td>(v_4)</td>
<td>(v_5)</td>
<td>(v_7)</td>
</tr>
</tbody>
</table>

(b) Reordering based chunk layout

<table>
<thead>
<tr>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 2</th>
<th>Page 3</th>
<th>Page 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_3)</td>
<td>(v_1)</td>
<td>(v_5)</td>
<td>(v_7)</td>
<td>(v_6)</td>
</tr>
</tbody>
</table>

Better temporal locality, buffer cache hit ratio, providing opportunities for sequential I/O

Problem: reordered optimization still suffers from inter-fragmentation within chunk
**Technique 2.2: Vertex-combination Chunk Layout Optimization**

- **Solution:** combine the vertices with complementary degree into one chunk

(a) chunk layout only with reordering based optimization

(b) Combination based chunk layout

Less *intra-chunk fragmentation* and better *spatial locality*, minimizing chunk access

#Fragmentation chunk is reduced from 95.24% to 52.77% on YahooWeb dataset.
**Technique 3: Differentiated Chunk Access Optimization**

- Graph algorithms usually involves different graph access pattern
  - **Top-down**: sparse access, only activated vertices
  - **Bottom-up**: dense access, scan the whole graph in vertex ID order

For bottom-up algorithm, we should traverse all vertices according to **edge data order**
**Technique 3: Differentiated Chunk Access Optimization**

- **Differentiated chunk access** pattern for bottom-up access
  - Store **key-value pair** <reordered_id, vid> to support chunk order access

<table>
<thead>
<tr>
<th>vertex metadata</th>
<th>(v_0)</th>
<th>(v_1)</th>
<th>(v_2)</th>
<th>(v_3)</th>
<th>(v_4)</th>
<th>(v_5)</th>
<th>(v_6)</th>
<th>(v_7)</th>
</tr>
</thead>
</table>

**Key-value pair**

- \(<0, 3>, <1, 5>, <2, 1>, <3, 7>\)
- \(<4, 2>, <5, 6>, <6, 0>, <7, 4>\)

- **Access pattern for bottom-up**

<table>
<thead>
<tr>
<th>edge data</th>
<th>(v_3)</th>
<th>(v_5)</th>
<th>(v_1)</th>
<th>(v_7)</th>
<th>(v_2)</th>
<th>(v_6)</th>
<th>(v_0)</th>
<th>(v_4)</th>
</tr>
</thead>
</table>

**Avoid random access** for vertices due to reordering optimization
Prototype System and Implementations

- **ChunkGraph** is implemented based on Ligra’s graph interface
Evaluation settings

- **Testbed**
  - A server with 2 sockets, each with 24 physical cores
  - \(8 \times 16\text{GB} = 128\text{GB} \text{ DRAM} + 2 \times 3.84\text{TB} \text{ SSD}\)

- **Graph datasets**

  | Dataset          | \(|V|\)  | \(|E|\) | CSR Size | Chunk Size |
  |------------------|--------|--------|----------|------------|
  | Twitter (TT)     | 61.6M  | 1.5B   | 11.9GB   | 13.5GB     |
  | Friendster (FS)  | 68.3M  | 2.6B   | 20.3GB   | 21.2GB     |
  | UKdomain (UK)    | 101.7M | 3.1B   | 26.2GB   | 27.5GB     |
  | YahooWeb (YW)    | 1.4B   | 6.6B   | 70.5GB   | 77.8GB     |
  | Kron29 (K29)     | 512M   | 8B     | 72GB     | 78.2GB     |
  | Kron30 (K30)     | 1B     | 16B    | 144GB    | 156.3GB    |

Real World Graph

Synthetic Graph
Evaluation settings

- **Comparison systems**
  - **Blaze**
    - The SOTA out-of-core graph system optimized for modern fast SSDs
  - **Ligra-mmap**
    - Ligra’s variant using mmap to map the graph data files into the virtual memory space

- **Evaluation metrics**
  - Graph query performance
    - BFS, SSSP, BC, Kcores, Radii, PageRank
  - I/O overhead
  - Computation overhead
Evaluation 1: Graph query performance

ChunkGraph achieves $1.62x-23.09x$ speedup upon Blaze, and $1.08x-2.94x$ compared to Ligra-mmap on sparsely accessed algorithms BFS, SSSP, BC.
**Evaluation 2: I/O overhead**

- Disk read amount of different algorithms on Yahoo and Kron30

![Bar charts comparing disk read amount for different algorithms on Yahoo and Kron30](image)

ChunkGraph reduces disk read amount by **4.68×** and **1.98×** on average, compared to Blaze and Ligra-mmap respectively.
Evaluation 3: Computation overhead

- CPU instructions executed during different algorithms’ execution

![Bar charts comparing CPU instructions across different algorithms and graph systems for Yahoo and Kron30 datasets.]

ChunkGraph reduces the number of CPU instructions by 185.01 times compared to the external memory graph system Blaze.
Conclusion

- **ChunkGraph**: an I/O efficient external graph system for processing large-scale graphs
  - **Classified and hierarchical** vertex storage strategy
  - Chunk layout optimization based on **vertex reordering and combination**
  - **Differentiated chunk access** optimization
  - Encompass both out-of-core systems and memory-storage cache subsystems

- More evaluation results and analysis are in the paper
- The source code is at [https://github.com/ZoRax-A5/ChunkGraph](https://github.com/ZoRax-A5/ChunkGraph)
Thanks for your attention!

Weixu Zong @ ZJU
zorax@zju.edu.cn