Mosaic: Processing a Trillion-Edge Graph on a Single Machine

Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, Taesoo Kim

Georgia Institute of Technology

Best Student Paper

April 26, 2017
Large-scale graph 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
mjakabiljo@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 6.6B 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 more complex graph processing workflows than previously eval-

Social networks

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>0.9M</td>
</tr>
<tr>
<td>Twitter 2010 [4]</td>
<td>42M</td>
<td>1.6B</td>
</tr>
<tr>
<td>UK web graph 2007 [10]</td>
<td>1.0B</td>
<td>3.7B</td>
</tr>
<tr>
<td>Yahoo web [9]</td>
<td>1.4B</td>
<td>0.6B</td>
</tr>
</tbody>
</table>

a project to run Facebook-scale graph applications in the summer of 2012 and is still the case today.
Large-scale graph processing is ubiquitous

Social networks

Genome analysis
Large-scale graph processing is ubiquitous

Social networks

Genome analysis

Graphs enable Machine Learning
Powerful, heterogeneous machines

More sockets. More memory. More SAP HANA.

by Cori Pasinetti on July 29, 2015

SGI UV 300H 20-Socket Appliance Certified by SAP to Run SAP HANA® Under Controlled Availability
Announcing the first 20-socket SAP HANA-certified in-memory server!

SGI announced today that the SGI® UV™ 300H is now SAP®-certified to run the SAP HANA® platform in controlled availability at 20-sockets—delivering up to 15 terabytes (TB) of in-memory computing capacity in a single node. Asserting the value of key enhancements in support package stack 10 (SPS10) for SAP HANA and SAP’s close collaboration with system providers, SGI UV 300H delivers outstanding single-node performance and simplicity for enterprises moving to SAP HANA to gain business breakthroughs.

SGI UV 300H is a specialized offering in the SGI® UV™ server line for in-memory computing that enables enterprises to further unlock value from information in real-time, boost innovation, and lower IT costs with SAP HANA. Featuring a highly differentiated single-node architecture, the system delivers significant performance advantages for businesses running SAP® Business Suite 4 SAP HANA (SAP S/4HANA) and complex analytics at extreme scale. The single-node simplicity also helps enterprises eliminate overhead associated with clustered environments, streamline high availability, and scale-up seamlessly as data volumes grow with near-linear performance.

Integrated with the recently announced SAP HANA SPS10, SGI UV 300H capitalizes on deep collaboration between SAP, Intel and SGI to optimize SAP HANA-based workloads on multicore NUMA (non-uniform memory access) systems. This enables enterprises to leverage single-node systems with very large memory capacity and

Terabytes of RAM on multiple sockets
Powerful, heterogeneous machines

Terabytes of sockets

Powerful many-core coprocessors
Powerful, heterogeneous machines

Terabytes of sockets

Powerful many-core coprocessors

Fast, large-capacity Non-volatile Memory
Powerful, heterogeneous machines

Take advantage of heterogeneous machine to process tera-scale graphs

Terabytes of sockets

Powerful many-core coprocessors

Fast, large-capacity Non-volatile Memory
1. Graph Processing: Sample Application

2. Design
   - Mosaic Architecture
   - Graph Encoding
   - API

3. Evaluation
Graph Processing: Applications

- Community Detection
- Find Common Friends
- Find Shortest Paths
- Estimate Impact of Vertices (webpages, users, ...)
- ...

Steffen Maass
Mosaic: Trillion Edges on a Single Machine
April 26, 2017
Graph Processing has many faces:

- Single Machine
  - Out-of-core
  - In memory
- Cluster
  - Out-of-core
  - In memory
Graph Processing has many faces:

- **Single Machine**
  - Out-of-core $\Rightarrow$ Cheap, but potentially slow
  - In memory $\Rightarrow$ Fast, but limited graph size

- **Cluster**
  - Out-of-core $\Rightarrow$ Large graphs, but expensive & slow
  - In memory $\Rightarrow$ Large graphs & fast, but very expensive
Graph Processing has many faces:

- **Single Machine**
  - Out-of-core ⇒ Cheap, but potentially slow
  - In memory ⇒ Fast, but limited graph size

- **Cluster**
  - Out-of-core ⇒ Large graphs, but expensive & slow
  - In memory ⇒ Large graphs & fast, but very expensive

⇒ Single machine, out-of-core is most cost-effective
⇒ Goal: Good performance and large graphs!
Mosaic: Design goals

**Goal**

Run *algorithms* on *very large graphs* on *a single machine* using *coprocessors*

Enabled by:

- Common, familiar API (vertex/edge-centric)
- Encoding: Lossless compression
- Cache locality
- Processing on isolated subgraphs
Architecture of Mosaic

- Usage of Xeon Phi & NVMe
- Involvement of Host

![Diagram showing architecture of Mosaic, with components like Global vertex state, <current state>, <next state>, Host Processors (Xeon), Meta transfer, Tile transfer, edge processing,fetch, receive, and a connection to NVMe and Xeon Phi.]
Graph encoding: Idea

**Compression**
Split graph into subgraphs, use local (short) identifiers

**Cache locality**
- Inside subgraphs: Sort by access order
- Between subgraphs: Overlap vertex sets
Background: Column first

- Locality for *write*
- Multiple sequential *reads*
Background: Row first

- Locality for *read*
- Multiple sequential *writes*

![Global adjacency matrix with source and target vertices, partition (S = 3)](image)

\[ \begin{align*}
P_{11} & & P_{12} & & P_{13} & & P_{14} \\
P_{21} & & P_{22} & & P_{23} & & P_{24} \\
P_{31} & & P_{32} & & P_{33} & & P_{34} \\
P_{41} & & P_{42} & & P_{43} & & P_{44}
\end{align*} \]

\[ \Rightarrow \text{Problem: No locality when switching row} \]
Background: Hilbert order

- Space-filling curve
- Provides locality between adjacent data points
From global to local: Tiles

- Convert graph to set of *tiles*

1) Start with adjacency Matrix:

![Graph](image-url)

![Matrix](image-url)
From global to local: Tiles

- Convert graph to set of *tiles*

2) Use first edge in tile $T_1$:
From global to local: Tiles

- Convert graph to set of tiles

3) Consume as many edges as possible:

![Diagram showing the conversion of a graph to tiles and the consumption of edges](image-url)
From global to local: Tiles

- Convert graph to set of tiles

4) Next edges do not fit in \( T_1 \), construct \( T_2 \):

- Source vertex (global)
- Target vertex (global)
- Global adjacency matrix
- Partition \((S = 3)\)
- \( T_1 \) (local)
  - \( (1,1) \), \( (3,5) \), \( (2,2) \), \( (4,4) \)
- \( T_2 \) (local)
  - \( (1,4) \), \( (3,5) \), \( (2,6) \), \( (4,3) \)

- \( 1 \): local vertex id
- \( 1 \): local → global id
- \( 1 \): local edge store order
Locality with Hilbert-ordered tiles

- Overlapping sets of sources and targets

⇒ Better locality than row-first or column-first
API: Pagerank example

- **Pull**: Gather per edge information
- **Reduce**: Combine results from multiple subgraphs
- **Apply**: Calculate non-associative regularization

**Edge-centric operation**

```
// On edge processor (co-processor)
// Edge e = (Vertex src, Vertex tgt)
def Pull(Vertex src, Vertex tgt):
    return src.val / src.out_degree
```

```
// On edge processor/global reducers (both)
def Reduce(Vertex v1, Vertex v2):
    return v1.val + v2.val
```

```
// On global reducers (host)
def Apply(Vertex v):
    v.val = (1 - α) + α × v.val
```

**Vertex-centric operation**

Formula: \( \text{Pagerank}_v = \alpha \times \left( \sum_{u \in \text{Neighborhood}(v)} \frac{\text{Pagerank}_u}{\text{degree}_u} \right) + (1 - \alpha) \)
Evaluation: Preprocessing

- Mosaic needs explicit preprocessing step
- 2-4 min for small datasets, 51 minutes for webgraph, 31 hours for trillion edges
- But: Can be amortized during execution:
  - GridGraph: Mosaic faster after
    - twitter: 20 iterations
    - uk2007: 8 iterations
  - X-Stream: Mosaic faster after
    - twitter: 8 iterations
    - uk2007: 5 iterations
Hilbert-ordered tiles allow efficient encoding of local graphs

Effect: up to 68% reduction in data size

<table>
<thead>
<tr>
<th>Graph</th>
<th>#vertices</th>
<th>#edges</th>
<th>Raw data</th>
<th>Mosaic size (red.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>*rmat24</td>
<td>16.8 M</td>
<td>0.3 B</td>
<td>2.0 GB</td>
<td>1.1 GB (−45.0%)</td>
</tr>
<tr>
<td>twitter</td>
<td>41.6 M</td>
<td>1.5 B</td>
<td>10.9 GB</td>
<td>7.7 GB (−29.4%)</td>
</tr>
<tr>
<td>*rmat27</td>
<td>134.2 M</td>
<td>2.1 B</td>
<td>16.0 GB</td>
<td>11.1 GB (−30.6%)</td>
</tr>
<tr>
<td>uk2007-05</td>
<td>105.8 M</td>
<td>3.7 B</td>
<td>27.9 GB</td>
<td>8.7 GB (−68.8%)</td>
</tr>
<tr>
<td>hyperlink14</td>
<td>1,724.6 M</td>
<td>64.4 B</td>
<td>480.0 GB</td>
<td>152.4 GB (−68.3%)</td>
</tr>
<tr>
<td>*rmat-trillion</td>
<td>4,294.9 M</td>
<td>1,000.0 B</td>
<td>8,000.0 GB</td>
<td>4,816.7 GB (−39.8%)</td>
</tr>
</tbody>
</table>
Hilbert-ordered tiles: Cache locality

- Cache misses and execution times for three different strategies

⇒ Hilbert-ordered tiles have up to 45% better cache locality, up to 43% reduction in runtime
Comparison to other single machine engines with PageRank:
Comparison to other single machine engines with Pagerank:

- Mosaic outperforms other system by $2.7 \times$ to $58.6 \times$
Conclusion

- Mosaic, a graph processing engine for trillion edge graphs on a single machine
- Hilbert-ordered tiles allow:
  - Enable localized processing on coprocessors
  - Optimizes cache locality
  - Enables compression
Thank you!