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 @ EuroSys’17

June 8, 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
majakabiljo@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 much more complex graph processing workflows than previously evaluated.

Social networks
Large-scale graph processing is ubiquitous

Social networks

Genome analysis
Powerful, heterogeneous machines

Terabytes of RAM on multiple sockets
Powerful, heterogeneous machines

Terabytes of sockets

Powerful many-core coprocessors
Powerful, heterogeneous machines

Terabytes of RAM on multiple sockets

Powerful many-core coprocessors

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

- Terabytes of sockets
- Powerful many-core coprocessors
- Fast, large-capacity Non-volatile Memory

Take advantage of heterogeneous machine to process tera-scale graphs
Table of contents

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, . . .)
- . . .
Example: Pagerank

- Calculate impact of each vertex

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

- Simple Algorithm:
  - In each iteration, distribute current impact along out-edges, weighted by degree
  - Sum up all incoming impacts \( \Rightarrow \) new impact for next iteration
  - Weight new impact with regularization factor \( \alpha = 0.85 \)
  - Repeat until no changes
1) Initialize
2) Propagate along outgoing edges
3) Sum up incoming contributions
4) Apply regularization: $x \ast 0.85 + 0.15$
5) Update outgoing edges for second iteration
6) Repeat until stabilized
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 ⇒ 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
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 of Mosaic architecture](attachment:image.png)
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*

```
1   2   3   4   5   6   7   8   9  10  11 12
Global adjacency
matrix
Source 
vertex
Target vertex
Partition
(S = 3)
P11 P12 P14 P13
P21 P22 P24 P23
P31 P32
P34 P33
P41 P42 P44 P43
```

⇒ Problem: No locality when switching column
Background: Row first

- Locality for read
- Multiple sequential writes
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:

Global adjacency matrix:

```
1  2   3   4   5   6   7   8   9  10  11  12
1  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12
2  | 6 | 8 | 2 | 1 | 7 | 5 | 6 | 3 | 4 |    |    
3  | 6 | 5 | 9 | 4 |    | 3 | 4 | 7 |    |    |    
4  | 3 | 5 | 6 | 4 | 7 | 3 | 4 | 7 | 6 |    |    
```

Partition (S = 3):

```
P_{11} P_{12} P_{14} P_{13}
P_{21} P_{22} P_{24} P_{23}
P_{31} P_{32} P_{34} P_{33}
P_{41} P_{42} P_{44} P_{43}
```
From global to local: Tiles

- Convert graph to set of tiles

2) Use first edge in tile $T_1$: 

![Diagram showing the conversion from global to local tiles]

- Source vertex (global)
- Target vertex (global)
- Global adjacency matrix
- Partition ($S = 3$)
- Local vertex id
- Local edge store order
- Meta information

Steffen Maass
Mosaic: Trillion Edges on a Single Machine
June 8, 2017 15 / 28
Convert graph to set of *tiles*

3) Use more edges in tile $T_1$:
From global to local: Tiles

- Convert graph to set of tiles

4) Use more edges in tile $T_1$:

![Diagram of a graph and its tile representation]
From global to local: Tiles

- Convert graph to set of tiles

5) Use more edges in tile \( T_1 \):

Steffen Maass
Mosaic: Trillion Edges on a Single Machine
June 8, 2017 15 / 28
From global to local: Tiles

- Convert graph to set of tiles

6) Next edges do not fit in $T_1$ anymore, construct $T_2$:
Locality with Hilbert-ordered tiles

- Overlapping sets of *sources* and *targets*

⇒ Better locality than row-first or column-first
1) Split original graphs into two subgraphs:
2) Internal data structure of $T_2$:

From global to local: Data structure
3) Compress edges: *Compressed sparse rows*

⇒ Efficient, local encoding, sequentially accessed
Better locality

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>
API: Pagerank example

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

**Edge-centric operation**

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

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

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

**Vertex-centric operation**

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

1) Start with $T_1$

```
Host

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>current state</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>next state</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>
```

**Xeon Phi**

```
T1

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
0.5 & 0.5 & 0.33 & 1 \\
\end{array}
\]

meta ($I_1$)

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
(1,1) & (3,5) & (2,2) & (4,4) \\
\end{array}
\]

Steffen Maass
Mosaic: Trillion Edges on a Single Machine
June 8, 2017
2) Execute *Pull* along all edges in $T_1$
3) Reduce all updates from $T_1$ onto next state

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

current state
```

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>0.50</td>
<td>0.00</td>
<td>1.00</td>
<td>0.83</td>
<td>0.00</td>
</tr>
</tbody>
</table>

next state
```

Xeon Phi

```
(1,1)(3,5)
(2,2)(4,4)
```

```
meta (I_1)
```

```
reduce(v)
```
4) Switch to $T_2$

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.00</td>
<td>0.50</td>
<td>0.00</td>
<td>1.00</td>
<td>0.83</td>
<td>0.00</td>
</tr>
</tbody>
</table>
```

```
T2
```

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.33</td>
<td>0.33</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>1.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1,4)</td>
<td>(3,5)</td>
<td>(2,6)</td>
<td>(4,3)</td>
</tr>
</tbody>
</table>
```

Steffen Maass  Mosaic: Trillion Edges on a Single Machine  June 8, 2017  20 / 28
5) *Pull all updates*

![Diagram ofPagerank example]

**Host**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>current state</strong></td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td><strong>next state</strong></td>
<td>0.00</td>
<td>0.50</td>
<td>0.00</td>
<td>1.00</td>
<td>0.83</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**Xeon Phi**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>current state</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>next state</strong></td>
<td>0.00</td>
<td>0.83</td>
<td>0.00</td>
<td>1.83</td>
</tr>
</tbody>
</table>

**Pull(e)**

```
(1,4) (3,5) (2,6) (4,3)
```

**Current state**

```
1.00 1.00 1.00 1.00 1.00 1.00
```

**Next state**

```
0.00 0.50 0.00 1.00 0.83 0.00
```

**meta (I_2)**

```
1 2 3 4
```

Steffen Maass

Mosaic: Trillion Edges on a Single Machine

June 8, 2017

20 / 28
6) *Reduce* updates from $T_2$

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>current state</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>next state</td>
<td>0.00</td>
<td>0.50</td>
<td>1.83</td>
<td>1.00</td>
<td>0.83</td>
<td>0.83</td>
</tr>
</tbody>
</table>
```

```
T2

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.33</td>
<td>0.33</td>
<td>0.5</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
(1,4) (3,5) (2,6) (4,3)
```

```
Xeon Phi
```

```
(   ,4)
(   ,6)
(   ,5)
(   ,3)
```

```
Reduce(v)
```

```
meta (I2)
```

```
Host
```

```
1 2 3 4 5 6
```

```
1.00 1.00 1.00 1.00 1.00 1.00
```

```
0.00 0.50 1.83 1.00 0.83 0.83
```

```
0.00 0.83 0.00 1.83
```

```
1 2 3 4
```

```
Xeon Phi
Steffen Maass
```

---

Steffen Maass
Mosaic: Trillion Edges on a Single Machine
June 8, 2017

Page 20 of 28
7) All tiles processed, *Apply* processed updates

![Pagerank example diagram]

**Host**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

**Apply(v)**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.15</td>
<td>0.58</td>
<td>1.71</td>
<td>1.00</td>
<td>0.86</td>
<td>0.86</td>
</tr>
</tbody>
</table>

**T2**

- **current state**
  - 1: 0.33
  - 2: 0.33
  - 3: 0.5
  - 4: 1
  - 5: 0.5

**next state**

- 1: 0.00
- 2: 0.83
- 3: 0.00
- 4: 1.83

**meta (I2)**

- (1,4), (3,5)
- (2,6), (4,3)
8) Switch *current* and *next* state, clear *next* state for second iteration

Host

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>cur</td>
<td>0.15</td>
<td>0.58</td>
<td>1.71</td>
<td>1.00</td>
<td>0.86</td>
<td>0.86</td>
</tr>
<tr>
<td>nex</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

T1

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.08</td>
<td>0.08</td>
<td>0.33</td>
<td>0.58</td>
<td>0.15</td>
<td>0.58</td>
</tr>
<tr>
<td>cur</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Xeon Phi

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta(I1)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>(1,1)</td>
<td>(3,5)</td>
<td>(2,2)</td>
<td>(4,4)</td>
<td></td>
</tr>
</tbody>
</table>
Questions:
- Preprocessing Cost
- Performance (in comparison)
- Impact of Design Decisions
- Scalability
Evaluation - Setup

- **Single Server:**
  - 2 sockets, 12 cores each
  - 768GB RAM
  - 4 Xeon Phi (KNC, 61 cores)
  - 6 NVMe (1.2TB each)

- 7 Algorithms

- 6 Datasets (3 real world, 3 synthetic)
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
**Performance comparison**

- **Comparison to other single machine engines with Pagerank:**

![Comparison to other single machine engines with Pagerank](image-url)

- **Mosaic**
- **GridGraph**
- **X-Stream**
- **GraphChi**
Mosaic outperforms other system by $2.7 \times$ to $58.6 \times$
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
Evaluation - Scalability

- Dimensions
  - Add Xeon Phis/NVMes
  - Add threads on each Xeon Phi

⇒ Mosaic scales well when adding threads/Xeon Phis
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

Code is open-source: https://github.com/sslab-gatech/mosaic
Thank you!
Evaluation - Selective scheduling

- Skip subgraphs without active vertices
- Especially useful for traversal algorithms: BFS, Connected Components, ...

\[ \Rightarrow 2.2 \times \text{speedup for BFS on twitter graph} \]
Evaluation - Dynamic load balancing

- Effect of choosing the wrong load balancing scheme
- Mosaic has light-weight balancing scheme

⇒ Up to $5.8 \times$ improvement in running time by choosing correct load balancing scheme
Mosaic architecture

- Host and Xeon Phi component
- Streaming-based design