Everything you always wanted to know about multicore graph processing but were afraid to ask

Jasmina Malicevic
EPFL, Switzerland

Baptiste Lepers
EPFL, Switzerland

Willy Zwaenepoel
EPFL, Switzerland

USENIX Annual Technical Conference, Jul 12th -14th 2017 , Santa Clara, CA ,USA
Graphs are everywhere

Social networks

Item recommendation

Search and website ranking
The maze of graph analytics platforms

<table>
<thead>
<tr>
<th>Single machine</th>
<th>In-memory</th>
<th>Out-of-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ligra</td>
<td></td>
<td>GraphChi</td>
</tr>
<tr>
<td>Polymer</td>
<td></td>
<td>X-Stream</td>
</tr>
<tr>
<td>Galois</td>
<td></td>
<td>GridGraph</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mosaic</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed</td>
<td>Pregel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Powergraph</td>
<td>Chaos</td>
</tr>
<tr>
<td></td>
<td>PowerLyra</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gemini</td>
<td></td>
</tr>
</tbody>
</table>
# The maze of graph analytics platforms

<table>
<thead>
<tr>
<th>Single machine</th>
<th>In-memory</th>
<th>Out-of-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ligra</td>
<td>Polymer</td>
<td>GraphChi</td>
</tr>
<tr>
<td>Polymer</td>
<td>Galois</td>
<td>X-Stream</td>
</tr>
<tr>
<td>Galois</td>
<td></td>
<td>GridGraph</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mosaic</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Distributed</th>
<th>In-memory</th>
<th>Out-of-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pregel</td>
<td>PowerLyra</td>
<td>Chaos</td>
</tr>
<tr>
<td>Powergraph</td>
<td>Gemini</td>
<td></td>
</tr>
<tr>
<td>PowerLyra</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gemini</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Everything you always wanted to know…

What techniques work and why?
Why is our work different?

- End-to-end evaluation

- Comparison of techniques, rather than systems
End-to-end evaluation

• Executing the algorithm is only one piece of the puzzle

End-to-end time = Pre-processing + Algorithm time
Motivation: Why end-to-end time?

- BFS on Twitter [Ligra]
Motivation: Why end-to-end time?

- BFS on Twitter [Ligra]

Need to understand the trade-off in end-to-end time!
Comparison of techniques not systems

• Implement techniques from different systems within one system

• Evaluation of techniques in isolation
  • Not constrained by system defined API

• Implementation is comparable/better than the original system
Questions we want to answer:

Pre-processing
• How to represent the graph?
• Cost of creating the representation?
• What data layout is best?

Algorithm
• Can we improve cache locality?
• Should we optimize for NUMA?
• Information flow: **push**, **pull** or a **both**?
The answers depend on:

- Algorithm – differ in # of active vertices per iteration
  - Only a subset active: BFS
  - Entire graph active: Pagerank, SpMV…

- Graph shape
  - Social networks (power law) graphs
  - Synthetic graph; 1B edges 64M vertices
  - Stored as edge array
Questions we want to answer:

Pre-processing

- How to represent the graph?
- Cost of creating the representation?
- What data layout is best?

Algorithm

- Can we improve cache locality?
- Should we optimize for NUMA?
- Information flow: push, pull or a both?
Graph representation

Edge array

- Layout is the same as input – no pre-processing
- To locate edges of a vertex, need to read all edges

Adjacency list: outgoing edges

- Pre-processing to group edges by vertex
- Easy to locate edges of a particular vertex
Questions we want to answer:

Pre-processing
- How to represent the graph?
- Cost of creating the representation?
- What data layout is best?

Algorithm
- Can we improve cache locality?
- Should we optimize for hardware? (NUMA)
- Information flow: push, pull or both?
Creating adjacency lists using dynamic allocation

0 - 2

0 - 2 2 - 3 0 - 3 1 - 0 0 - 1

0 1 2 3

2 null null null
Creating adjacency lists using dynamic allocation

2 -3

0 - 2  2 -3  0 - 3  1 - 0  0 - 1

0  1  2  3

2
null
3
null
Creating adjacency lists using dynamic allocation

- Frequent reallocations
- Adjacency lists spread out in memory
Creating adjacency lists using sorting

- Load edge array into memory
- Sort by source or destination
- Vertices point to start of their adjacency list

✓ Avoid reallocations
✓ Adjacency lists contiguous in memory
Creating adjacency lists using sorting

- Load edge array into memory
- Sort by source or destination
- Vertices point to start of their adjacency list

Q: Which approach is better?

- Avoid reallocations
- Adjacency lists contiguous in memory
Which pre-processing method is better?

<table>
<thead>
<tr>
<th>Pre-processing technique</th>
<th>Time (sec)</th>
<th>LLC misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>15.0</td>
<td>69%</td>
</tr>
<tr>
<td>Count sort</td>
<td>13.5</td>
<td>71%</td>
</tr>
<tr>
<td>Radix sort</td>
<td>4.0</td>
<td>26%</td>
</tr>
</tbody>
</table>

Radix sort low LLC miss rate => 3.5X better
Questions we want to answer:

**Pre-processing**
- How to represent the graph?
- Cost of creating the representation?
- What data layout is best?

- Adjacency lists
- Edge arrays
- Radix sort wins for adjacency lists

**Algorithm**
- Can we improve cache locality?
- Should we optimize for NUMA?
- Information flow: **push, pull** or a **both**?
Which is data layout is better?
Which is better?

BFS

<table>
<thead>
<tr>
<th>Data layout</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjacency list</td>
<td>6</td>
</tr>
<tr>
<td>edge array</td>
<td>4</td>
</tr>
</tbody>
</table>

Pre-processing | Algorithm

Pagerank

<table>
<thead>
<tr>
<th>Data layout</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adjacency list</td>
<td>40</td>
</tr>
<tr>
<td>edge array</td>
<td>30</td>
</tr>
</tbody>
</table>

Pre-processing | Algorithm
Adjacency lists always wins?

- SpMV – one pass over the edge array
- Pre-processing not amortized
  - Cost is = pass over edge array
Questions we want to answer:

**Pre-processing**
- How to represent the graph?
- Cost of creating the representation?
- What data layout is best?

- Adjacency lists
- Edge arrays
- Radix sort wins for adjacency lists
- BFS: Adj. list
- PR: Adj. list
- SpMV: Edge array

**Algorithm**
- Can we improve cache locality?
- Should we optimize for NUMA?
- Information flow: **push**, **pull** or a **both**?
Memory accesses – edge arrays

Edge array: 0 - 3  2 -3  0 - 1  1 - 0  0 - 2

Fetch edge:

Fetch state of source:

Fetch state of destination:
Memory accesses – edge arrays

Edge array: 

0 - 3 2 -3 0 - 1 1 - 0 0 - 2

Vertex state array:

S(0) S(1) S(2) S(3)

Fetch edge: 

0 - 3

Fetch state of source: 

Fetch state of destination:

✓ Cache-friendly edge read
Memory accesses – edge arrays

Edge array:

0 - 3  2 - 3  0 - 1  1 - 0  0 - 2

Fetch edge:

0 - 3

Fetch state of source:

S(0)

Fetch state of destination:

✓ Cache-friendly edge read
x Potentially random access to source state
Memory accesses – edge arrays

Edge array: 0 - 3 2 - 3 0 - 1 1 - 0 0 - 2

Fetch edge: 0 - 3
Fetch state of source: S(0)
Fetch state of destination: S(3)

✓ Cache-friendly edge read
x Potentially random access to source state
x Random access to destination state
Memory accesses – adjacency lists

Adjacency list

Fetch edge:
Fetch state of source:
Fetch state of destination:

Vertex state array:
Memory accesses – adjacency lists

Adjacency list

0 ← 3 → 1 → 2

1 ← 0 → 2

2 ← 3 → 3

Vertex state array:

S(0)
S(1)
S(2)
S(3)

Fetch edge:

0

Fetch state of source:

Fetch state of destination:
Memory accesses – adjacency lists

Adjacency list

![Adjacency list diagram]

Fetch edge:

Fetch state of source:

Fetch state of destination:

✓ Cache-friendly edge read
Memory accesses – adjacency lists

Adjacency list

Fetch edge:

Fetch state of source:

Fetch state of destination:

✓ Cache-friendly edge read
✓ Cache-friendly source state read

Vertex state array:
Memory accesses – adjacency lists

Adjacency list

Fetch edge:
Fetch state of source:
Fetch state of destination:

✓ Cache-friendly edge read
✓ Cache-friendly source state read
✗ Random access to destination state

Vertex state array:
## LLC miss rate

<table>
<thead>
<tr>
<th>Data layout</th>
<th>BFS</th>
<th>PageRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge array</td>
<td>57%</td>
<td>83%</td>
</tr>
<tr>
<td>Adjacency list</td>
<td>63%</td>
<td>78%</td>
</tr>
</tbody>
</table>
### LLC miss rate

<table>
<thead>
<tr>
<th>Data layout</th>
<th>BFS</th>
<th>PageRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge array</td>
<td>57%</td>
<td>83%</td>
</tr>
<tr>
<td>Adjacency list</td>
<td>63%</td>
<td>78%</td>
</tr>
</tbody>
</table>

Q: Can the miss rate be improved? At what cost?
Improving cache locality

Idea: Constrain the number of vertices accessed

Solution: Use out-of core technique – 2D Grid [from GridGraph]
Improving cache locality

**Idea:** Constrain the number of vertices accessed

**Solution:** Use out-of-core technique – 2D Grid [from GridGraph]

- Vertices divided into ranges
- Edges placed in a cell:
  - Row of source vertex
  - Column of destination vertex
Improving cache locality

**Idea:** Constrain the number of vertices accessed

**Solution:** Use out-of-core technique – 2D Grid [from GridGraph]

- Vertices divided into ranges
- Edges placed in a cell:
  - Row of source vertex
  - Column of destination vertex
Improving cache locality

**Idea:** Constrain the number of vertices accessed

**Solution:** Use out-of-core technique – 2D Grid [from GridGraph]

- Vertices divided into ranges
- Edges placed in a cell:
  - Row of source vertex
  - Column of destination vertex
Improving cache locality

**Idea:** Constrain the number of vertices accessed

**Solution:** Use out-of core technique – 2D Grid [from GridGraph]

- Vertices divided into ranges
- Edges placed in a cell:
  - Row of source vertex
  - Column of destination vertex
Improving cache locality

**Idea:** Constrain the number of vertices accessed

**Solution:** Use out-of-core technique – 2D Grid [from GridGraph]

- Vertices divided into ranges
- Edges placed in a cell:
  - Row of source vertex
  - Column of destination vertex
- Compute over cells of row or column
Cache-miss rate: Grid

<table>
<thead>
<tr>
<th>Data layout</th>
<th>BFS</th>
<th>PageRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge array</td>
<td>57%</td>
<td>83%</td>
</tr>
<tr>
<td>Adjacency list</td>
<td>63%</td>
<td>78%</td>
</tr>
<tr>
<td>2D Grid</td>
<td>23%</td>
<td>35%</td>
</tr>
</tbody>
</table>
Evaluation: cache-optimization (BFS)

### BFS

<table>
<thead>
<tr>
<th>Data layout</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>adj list</td>
<td></td>
</tr>
<tr>
<td>edge array</td>
<td>10</td>
</tr>
<tr>
<td>grid</td>
<td>3</td>
</tr>
</tbody>
</table>

- **Algorithm**
Evaluation: cache-optimization (BFS)
Evaluation: cache-optimization (BFS)

Adjacency lists have the best performance on BFS.
Evaluation: cache-optimization (PageRank)
Evaluation: cache-optimization (PageRank)
Evaluation: cache-optimization (PageRank)

For Pagerank, the grid is the winning approach.
Questions we want to answer:

Pre-processing

✓ How to represent the graph?
✓ Cost of creating the representation?
✓ What data layout is best?

Algorithm

✓ Can we improve cache locality?
  • Should we optimize for NUMA?
  • Information flow: push, pull or both?

○ Adjacency lists    ○ Edge arrays
  ○ Radix sort wins for adjacency lists
  ○ BFS: Adj. list   ○ PR: Grid   ○ SpMV: Edge array

○ Yes. By laying out the edges in a grid format
  ○ BFS: Adj. list   ○ PR: Grid   ○ SpMV: Edge array
NUMA-Aware optimizations

• NUMA-Aware data placement
  • Additional partitioning step in the pre-processing phase

• NUMA-Aware computation
  • Threads compute on local data

• Evaluation environment
  • Machine A: 2 NUMA nodes, 128GB DRAM, 16 Cores
  • Machine B: 4 NUMA nodes, 256GB DRAM, 32 Cores
NUMA-Aware data placement

- Vertices spread across NUMA nodes
- Edges collocated with their destination vertex
NUMA-Aware data placement

- Vertices spread across NUMA nodes
- Edges collocated with their destination vertex
NUMA-Aware data placement

- Vertices spread across NUMA nodes
- Edges collocated with their destination vertex
PageRank

MA: 2 NUMA nodes

MB: 4 NUMA nodes
PageRank

MA: 2 NUMA nodes
MB: 4 NUMA nodes

27% - 50% Improved compute time
PageRank

MA: 2 NUMA nodes

MB: 4 NUMA nodes
PageRank

Pre-processing amortized only on Machine B

MA: 2 NUMA nodes

NUMA: 4 NUMA nodes
BFS

<table>
<thead>
<tr>
<th>Time (sec)</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>interleaved</td>
<td></td>
</tr>
<tr>
<td>NUMA</td>
<td></td>
</tr>
<tr>
<td>interleaved</td>
<td></td>
</tr>
<tr>
<td>NUMA</td>
<td></td>
</tr>
</tbody>
</table>

MA: 2 NUMA nodes

MB: 4 NUMA nodes
BFS

No gain in algorithm time, contention on memory bus
Questions we want to answer:

**Pre-processing**
- How to represent the graph?
  - Adjacency lists
  - Edge arrays
- Cost of creating the representation?
  - Radix sort wins for adjacency lists
- What data layout is best?
  - BFS: Adj. list
  - PR: Grid
  - SpMV: Edge array

**Algorithm**
- Can we improve cache locality?
  - Yes. By laying out the edges in a grid format
  - BFS: Adj. list
  - PR: Grid
  - SpMV: Edge array
- Should we optimize for NUMA?
  - Can pay off only on big machines
  - BFS & SpMV: No gain
  - PR: NUMA-optimize

- Information flow: push, pull or a both?
Information flow

• Push
  • You **push** information to your neighbors
  • You need **outgoing edges**

• Pull
  • You **pull** information from your neighbors
  • You need **incoming edges**
Which one is better?

• Push
  • You push information to your neighbors - write to state of others

• Pull
  • You pull information from your neighbors – write to own state
Which one is better?

• Push
  • You **push** information to your neighbors - write to state of others
    ✓ Good when few vertices are active
    ✗ Needs locks

• Pull
  • You **pull** information from your neighbors – write to own state
    ✓ Good when many vertices are active
    ✓ Locks can be avoided
PUSH vs. PULL – BFS & PR

BFS

Information flow

Time (s)

0
1
2
3
4
5
6
7
8

adj. push
adj. pull

Pre-processing

Pagerank

Information flow

Pre-processing
Algorithm

grid (push)
grid (pull)
Questions we want to answer:

**Pre-processing**
- How to represent the graph?
  - Adjacency lists
  - Edge arrays
- Cost of creating the representation?
  - Radix sort wins for adjacency lists
- What data layout is best?
  - BFS: Adj. list
  - PR: Grid
  - SpMV: Edge array

**Algorithm**
- Can we improve cache locality?
  - Yes. By laying out the edges in a grid format
  - BFS: Adj. list
  - PR: Grid
  - SpMV: Edge array
  - Can pay off only on big machines
- Should we optimize for NUMA?
  - BFS & SpMV: No gain
  - PR: NUMA-optimize
  - Less synchronization not always a win
  - BFS: Push (locks)
  - PR: Pull (no locks)

- Information flow: **push, pull** or a **both**?
Push & Pull both win in different situations

• Combine them
  • Use push when it is efficient
  • Use pull when it is efficient
  • Cost: You need both, incoming and outgoing edges
Benefit of Push/Pull

BFS

Time (s)

Information flow

adj. push

adj. push/pull
Benefit of Push/Pull

BFS

<table>
<thead>
<tr>
<th>Information flow</th>
<th>adj. push</th>
<th>adj. push/pull</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

Pre-processing: 1, Algorithm: 8

BFS
Questions we want to answer:

- How to represent the graph?
- Cost of creating the representation?
- What data layout is best?
- Can we improve cache locality?
- Should we optimize for NUMA?
- Information flow: push, pull or both?

Pre-processing:
- Adjacency lists
- Edge arrays
- Radix sort wins for adjacency lists
- BFS: Adj. list  PR: Grid  SpMV: Edge array

Algorithms:
- Yes. By laying out the edges in a grid format
- BFS: Adj. list  PR: Grid  SpMV: Edge array
- Can pay off only on big machines
- BFS & SpMV: No gain  PR: NUMA-optimize
- Less synchronization not always a win
- BFS: Push (locks)  PR: Pull (no locks)
- Push/Pull no win in end-to-end (directed graphs)
Additional results in the paper

• Scalability of pre-processing approaches

• Relation between pre-processing and loading from HDD and SSD

• Results on other algorithms

• Results for different graph types
## Systems that motivated the paper

<table>
<thead>
<tr>
<th>System</th>
<th>Data Layout</th>
<th>Iteration Model</th>
<th>Push or Pull</th>
<th>NUMA-Aware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ligra [PPoPP ‘13]</td>
<td>Adj. List</td>
<td>Vertex-centric</td>
<td>Push &amp; Pull</td>
<td>-</td>
</tr>
<tr>
<td>X-Stream [SOSP’13]</td>
<td>Edge Array</td>
<td>Edge-centric</td>
<td>Push</td>
<td>-</td>
</tr>
<tr>
<td>GridGraph [ATC ‘15]</td>
<td>Grid</td>
<td>Grid-cell</td>
<td>Push</td>
<td>-</td>
</tr>
</tbody>
</table>
Summary

**Pre-processing**
- Edge arrays
- Adjacency lists
- Sorting techniques

**Algorithm time**
- Cache-optimizations
- Push vs. Pull
- Synchronization
- NUMA-aware computation
Conclusion

- Improvement in computation is not free
- **Trade-off** between added pre-processing time and algorithm time

Whether optimization cost in pre-processing is amortized, depends on algorithm:

- SpMV: Short algorithm and does not benefit from additional optimizations
- BFS: Building adjacency lists
- Pagerank: Optimizing for cache locality (grid) & NUMA-Awareness

Fork us on GitHub: [https://github.com/epfl-labos/EverythingGraph.git](https://github.com/epfl-labos/EverythingGraph.git)