GraphChi:
Disk-based Large-Scale Graph Computation on a Single Machine

Aapo Kyrölä
Ph.D. candidate @ CMU

http://www.cs.cmu.edu/~akyrola
Twitter: @kyrpov

Big Data – small machine
GraphChi can compute on the full Twitter follow-graph with just a standard laptop.

~ as fast as a very large Hadoop cluster!
(size of the graph Fall 2013, > 20B edges [Gupta et al 2013])
Disk-based Large-Scale Graph Computation on a Single Machine

Why Graphs?
BigData with **Structure**: BigGraph

- social graph
- social graph
- follow-graph
- consumer-products graph

- user-movie ratings graph
- DNA interaction graph
- WWW link graph
- Secret stuff
Why on a single machine?
Why use a cluster?

Two reasons:

1. One computer cannot handle my graph problem in a *reasonable* time.

2. I need to solve the problem very fast.
Why use a cluster?

Two reasons:

1. One computer cannot handle my graph problem in a reasonable time.

   Our work expands the space of feasible problems on one machine:
   - Our experiments use the same graphs, or bigger, than previous papers on distributed graph computation. (+ we can do Twitter graph on a laptop)

2. I need to solve the problem very fast.

   Our work raises the bar on required performance for a “complicated” system.
Benefits of single machine systems

Assuming it can handle your big problems…

1. Programmer productivity
   – Global state
   – Can use “real data” for development

2. Inexpensive to install, administer, less power.

Efficient Scaling

Distributed Graph System

6 machines

Task 1, Task 2, Task 3, Task 4, Task 5, Task 6

Time T

Single-computer system (capable of big tasks)

Task 1, Task 2, Task 3, Task 4, Task 5, Task 6

Time T

12 machines

(Significantly) less than 2x throughput with 2x machines

Task 1, Task 2, Task 3, Task 4, Task 5, Task 6

Time T

Exactly 2x throughput with 2x machines

Task 1, Task 2, Task 3, Task 4, Task 5, Task 6

Time T

Task 10, Task 11, Task 12

Press Release

Egham, UK, April 3, 2013

Gartner Says In-Memory Computing Is Racing Towards Mainstream Adoption

In-Memory Data Grid Market to Reach $1 Billion by 2016

Some new products introduced must gain market share; some will lose, they show where.

In this case, the product’s performance.

Violin Memory is an innovator in storage systems for computer servers. However, its success was uncommonly quick. Now, the company is going down-market, with data storage cards for individual computer servers. These data cards, we’re told...
Computing on Big Graphs

Disk-based Large-Scale Graph Computation on a Single Machine
Big Graphs != Big Data

Data size:

Facebook

140 billion connections

≈ 1 TB

Not a problem!

Computation:

Hard to scale
Research Goal

Compute on graphs with billions of edges, in a reasonable time, on a single PC.

– Reasonable = close to numbers previously reported for distributed systems in the literature.

Experiment PC: Mac Mini (2012)
Outline of the Talk

1. Background / Preliminaries
2. “Parallel Sliding Windows” -algorithm
3. Experimental evaluation
4. Evolving Graphs
5. Final remarks
Computational Model
Computational Model

• Graph G = (V, E)
  – directed edges: e = (source, destination)
  – each edge and vertex associated with a value (user-defined type)
  – vertex and edge values can be modified
    • (structure modification also supported)

Terms: e is an out-edge of A, and in-edge of B.
Vertex-centric Programming

- “Think like a vertex”
- Popularized by the Pregel and GraphLab projects
  - Historically, systolic computation and the Connection Machine

MyFunc(vertex)
{ // modify neighborhood }
The Main Challenge of Disk-based Graph Computation:

Random Access

~ 100K reads / sec (commodity)
~ 1M reads / sec (high-end arrays)

<< 5-10 M random edges / sec to achieve “reasonable performance”
Our Solution

Parallel Sliding Windows (PSW)
Parallel Sliding Windows: Phases

- PSW processes the graph one sub-graph at a time:

1. Load
2. Compute
3. Write

- In one iteration, the whole graph is processed.
  - And typically, next iteration is started.
PSW: Shards and Intervals

- Vertices are numbered from 1 to n
  - $P$ intervals, each associated with a shard on disk.
  - sub-graph = interval of vertices
PSW: Layout

Shard: in-edges for interval of vertices; sorted by source-id

- Vertices 1..100
- Vertices 101..700
- Vertices 701..1000
- Vertices 1001..10000

Shards small enough to fit in memory; balance size of shards
PSW: Loading Sub-graph

Load subgraph for vertices 1..100

Vertices 1..100
Shard 1

Vertices 101..700
Shard 2

Vertices 701..1000
Shard 3

Vertices 1001..10000
Shard 4

Load all in-edges in memory

What about out-edges?
Arranged in sequence in other shards
PSW: Loading Sub-graph

Load subgraph for vertices 101..700

Vertices 1..100
- Shard 1

Vertices 101..700
- Shard 2

Vertices 701..1000
- Shard 3

Vertices 1001..10000
- Shard 4

Load all in-edges in memory

In-edges for vertices 1..100 sorted by source_id

Out-edge blocks in memory

1. Load
2. Compute
3. Write

PSW Load-Phase

Only P large reads for each interval.

$P^2$ reads on one full pass.

Interval 1

Shard 1  Shard 2  Shard 3  Shard 4
PSW: Execute updates

- Update-function is executed on interval’s vertices
- Edges have pointers to the loaded data blocks
  - Changes take effect immediately → asynchronous.

2. Compute

3. Write

Deterministic scheduling prevents races between neighboring vertices.
PSW: Commit to Disk

- In write phase, the blocks are written back to disk
  - Next load-phase sees the preceding writes → asynchronous.

In total:

$P^2$ reads and writes / full pass on the graph.

→ Performs well on both SSD and hard drive.
GraphChi: Implementation

Evaluation & Experiments
GraphChi

- C++ implementation: 8,000 lines of code
  - Java-implementation also available (~ 2-3x slower), with a Scala API.
- Several optimizations to PSW (see paper).

Source code and examples: http://graphchi.org
EVALUATION:
APPLICABILITY
Evaluation: Is PSW expressive enough?

**Graph Mining**
- Connected components
- Approx. shortest paths
- Triangle counting
- Community Detection

**SpMV**
- PageRank
- Generic

**Recommendations**
- Random walks

**Collaborative Filtering**
(by Danny Bickson)
- ALS
- SGD
- Sparse-ALS
- SVD, SVD++
- Item-CF
  + many more

**Probabilistic Graphical Models**
- Belief Propagation

Algorithms implemented for GraphChi (Oct 2012)
Comparisons to existing systems

IS GRAPHCHI FAST ENOUGH?
Experiment Setting

- Mac Mini (Apple Inc.)
  - 8 GB RAM
  - 256 GB SSD, 1 TB hard drive
  - Intel Core i5, 2.5 GHz

- Experiment graphs:

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>P (shards)</th>
<th>Preprocessing</th>
</tr>
</thead>
<tbody>
<tr>
<td>live-journal</td>
<td>4.8M</td>
<td>69M</td>
<td>3</td>
<td>0.5 min</td>
</tr>
<tr>
<td>netflix</td>
<td>0.5M</td>
<td>99M</td>
<td>20</td>
<td>1 min</td>
</tr>
<tr>
<td>twitter-2010</td>
<td>42M</td>
<td>1.5B</td>
<td>20</td>
<td>2 min</td>
</tr>
<tr>
<td>uk-2007-05</td>
<td>106M</td>
<td>3.7B</td>
<td>40</td>
<td>31 min</td>
</tr>
<tr>
<td>uk-union</td>
<td>133M</td>
<td>5.4B</td>
<td>50</td>
<td>33 min</td>
</tr>
<tr>
<td>yahoo-web</td>
<td>1.4B</td>
<td>6.6B</td>
<td>50</td>
<td>37 min</td>
</tr>
</tbody>
</table>
Comparison to Existing Systems

**PageRank**

**WebGraph Belief Propagation (U Kang et al.)**

- Twitter-2010 (1.5B edges)
- Yahoo-web (6.7B edges)

- ✔ GraphChi can solve as big problems as existing large-scale systems.
- ✔ Comparable performance.

- GraphLab v1 (8 cores)

- Hadoop (1636 machines)

**Notes:** comparison results do not include time to transfer the data to cluster, preprocessing, or the time to load the graph from disk. GraphChi computes asynchronously, while all but GraphLab synchronously.
PowerGraph Comparison

• **PowerGraph / GraphLab 2** outperforms previous systems by a wide margin on natural graphs.

• With 64 more machines, 512 more CPUs:
  – **Pagerank**: 40x faster than GraphChi
  – **Triangle counting**: 30x faster than GraphChi.

GraphChi has state-of-the-art performance / CPU.
Sneak peek

SYSTEM EVALUATION

Consult the paper for a comprehensive evaluation:
- HD vs. SSD
- Striping data across multiple hard drives
- Comparison to an in-memory version
- Bottlenecks analysis
- Effect of the number of shards
- Block size and performance.
Scalability / Input Size [SSD]

- Throughput: number of edges processed / second.

**Conclusion:** the throughput remains roughly constant when graph size is increased.

GraphChi with hard-drive is ~ 2x slower than SSD (if computational cost low).

**PageRank -- throughput (Mac Mini, SSD)**

- Performance vs. Graph size
- Billsions on x-axis
- Throughput in number of edges processed per second
- Datasets: domain, twitter-2010, uk-2007-05, uk-union, yahoo-web

Paper: scalability of other applications.
Bottlenecks

- Cost of constructing the sub-graph in memory is almost as large as the I/O cost on an SSD
  - Graph construction requires a lot of random access in RAM → memory bandwidth becomes a bottleneck.
Bottlenecks / Multicore

- Computationally intensive applications benefit substantially from parallel execution.
- GraphChi saturates SSD I/O with 2 threads.

**Connected Components**

![Connected Components Diagram](image)

**Matrix Factorization (ALS)**

![Matrix Factorization Diagram](image)

Experiment on MacBook Pro with 4 cores / SSD.
EVOLVING GRAPHS
Evolving Graphs

• Most interesting networks grow continuously:
  – New connections made, some ‘unfriended’.

• Desired functionality:
  – Ability to add and remove edges in streaming fashion;
  – ... while continuing computation.
PSW and Evolving Graphs

• Adding edges
  – Each (shard, interval) has an associated edge-buffer.

• Removing edges: Edge flagged as “removed”.

(interval(1) → edge-buffer(j, 1)
  (for example)
  Twitter “firehose”
  (for example)
  Twitter “firehose”
  (for example)
  Twitter “firehose”

(interval(2) → edge-buffer(j, 2)
  (for example)
  Twitter “firehose”
  (for example)
  Twitter “firehose”
  (for example)
  Twitter “firehose”

(interval(P) → edge-buffer(j, P)
  (for example)
  Twitter “firehose”
  (for example)
  Twitter “firehose”
  (for example)
  Twitter “firehose”

shard(j) → New edges
  (for example)
  Twitter “firehose”
  (for example)
  Twitter “firehose”
  (for example)
  Twitter “firehose”

Recreating Shards on Disk

- When buffers fill up, shards are recreated on disk:
  - Too big shards are split.
- During recreation, deleted edges are permanently removed.

\[
\text{interval}(1) \rightarrow \text{interval}(2) \cdots \rightarrow \text{interval}(P) \rightarrow \text{shard}(j) \\
\text{interval}(1) \rightarrow \text{interval}(2) \cdots \rightarrow \text{interval}(P+1) \rightarrow \text{shard}(j+1)
\]
Streaming Graph Experiment

- On the Mac Mini:
  - Streamed edges in random order from the twitter-2010 graph (1.5 B edges)
    - With maximum rate of 100K or 200K edges/sec. (very high rate)
  - Simultaneously run PageRank.
  - Data layout:
    - Edges were streamed from hard drive
    - Shards were stored on SSD.
Ingest Rate

When graph grows, shard recreations become more expensive.
Summary

• Parallel Sliding Windows algorithm enables processing of large graphs with very few non-sequential disk accesses.

• For the system researchers, GraphChi is a solid baseline for system evaluation
  – It can solve as big problems as distributed systems.

• Takeaway: Appropriate data structures as an alternative to scaling up.
FINAL REMARKS
Single Machine vs. Cluster

• Most “Big Data” computations are **I/O-bound**
  – **Single machine**: disk bandwidth + seek latency
  – **Distributed memory**: network bandwidth + network latency

• Complexity / challenges:
  – **Single machine**: algorithms and data structures that reduce random access
  – **Distributed**: admin, coordination, consistency, fault tolerance

• Total cost
  – Programmer productivity
  – Specialized vs. Generalized frameworks
Recent developments

• Recently two disk-based graph computation systems published:
  – TurboGraph (KDD’13)
  – X-Stream (SOSP’13 in October)

• Significantly better performance than GraphChi on many problems
  – Avoid preprocessing (“sharding”)
  – But GraphChi can do some computation that X-Stream cannot (triangle counting and related);
    TurboGraph requires SSD

  – Hot research area!
Thank you!

Aapo Kyrölä
Ph.D. candidate @ CMU – soon to graduate! (Currently visiting U.W)

http://www.cs.cmu.edu/~akyrola
Twitter: @kyrprov