GraphChi: Large-Scale Graph Computation on Just a PC

Aapo Kyrölä (CMU)  
Guy Blelloch (CMU)  
Carlos Guestrin (UW)

In co-operation with the GraphLab team.
BigData with **Structure**: BigGraph

- social graph (Facebook)
- social graph (LinkedIn)
- follow-graph (Twitter)
- consumer-products graph (Amazon)
- user-movie ratings graph (Netflix)
- DNA interaction graph (ENCODE)
- WWW link graph (Google)

*etc.*
Big Graphs != Big Data

Data size:
Facebook

140 billion connections
≈ 1 TB

Not a problem!

Computation:

Hard to scale
Could we compute Big Graphs on a single machine?

Disk-based Graph Computation
Distributed State is Hard to Program

Writing distributed applications remains cumbersome.
Efficient Scaling

- Businesses need to compute hundreds of distinct tasks on the same graph
  - Example: personalized recommendations.

Parallelize each task

Parallelize across tasks
Other Benefits

• Costs
  – Easier management, simpler hardware.

• Energy Consumption
  – Full utilization of a single computer.

• Embedded systems, mobile devices
  – A basic flash-drive can fit a huge graph.
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)
Computational Model
Computational Model

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

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

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

Random Access
Random Access Problem

- Symmetrized adjacency file with values,

<table>
<thead>
<tr>
<th>vertex</th>
<th>in-neighbors</th>
<th>out-neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3:2.3, 19:1.3, 49:0.65</td>
<td>781:2.3, 881:4.2...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>19</td>
<td>3:1.2,</td>
<td>781:2.3, 881:4.2...</td>
</tr>
</tbody>
</table>

For sufficient performance, millions of random accesses / second would be needed. Even for SSD, this is too much.

- ... or with

<table>
<thead>
<tr>
<th>vertex</th>
<th>in-neighbor-ptr</th>
<th>out-neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3:881, 19:10092, 49:20763,...</td>
<td>781:2.3, 881:4.2...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>19</td>
<td>3:882, 9:2872, ...</td>
<td>5:1.3, 28:2.2,...</td>
</tr>
</tbody>
</table>
Possible Solutions

1. Use SSD as a memory-extension?
   [SSDAlloc, NSDI’11]
   
   Too many small objects, need millions / sec.

2. Compress the graph structure to fit into RAM?
   [→ WebGraph framework]
   
   Associated values do not compress well, and are mutated.

3. Cluster the graph and handle each cluster separately in RAM?

   Expensive; The number of inter-cluster edges is big.

4. Caching of hot nodes?
   
   Unpredictable performance.
Our Solution

Parallel Sliding Windows (PSW)
Parallel Sliding Windows: Phases

• PSW processes the graph one sub-graph 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

1. Load
2. Compute
3. Write

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

1. Load
2. Compute
3. Write
PSW: Loading Sub-graph
Load subgraph for vertices 101..700

1. Load
2. Compute
3. Write
PSW Load-Phase

Only $P$ large reads for each interval.

$P^2$ reads on one full pass.

<table>
<thead>
<tr>
<th>1. Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Compute</td>
</tr>
<tr>
<td>3. Write</td>
</tr>
</tbody>
</table>

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.
PSW: Commit to Disk

- In write phase, the blocks are written back to disk
  - Next load-phase sees the preceding writes \(\rightarrow\) asynchronous.

In total:

- \(P^2\) reads and writes / full pass on the graph.
  - \(\rightarrow\) 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

**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, 1TB 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

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.

On a Mac Mini:
- GraphChi can solve as big problems as existing large-scale systems.
- Comparable performance.

See the paper for more comparisons.
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).

---

**Paper**: scalability of other applications.
**Bottlenecks / Multicore**

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

---

![GraphChi Benchmark Results](image_url)

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

Graphs whose structure changes over time
Evolving Graphs: Introduction

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

- Desired functionality:
  - Ability to add and remove edges in streaming fashion;
  - ... while continuing computation.

- Related work:
  - *Kineograph* (EuroSys ’12), distributed system for computation on a changing graph.
PSW and Evolving Graphs

- Adding edges
  - Each (shard, interval) has an associated edge-buffer.
- Removing edges: Edge flagged as “removed”.

![Diagram showing edge buffer for each shard and interval](CarnegieMellon)
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.

![Diagram showing shard recreation and splitting process](image-url)
EVALUATION: EVOLVING GRAPHS

Streaming Graph experiment
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.
When graph grows, shard recreations become more expensive.
Streaming: Computational Throughput

Throughput varies strongly due to shard rewrites and asymmetric computation.
Conclusion
Future Directions

- This work: small amount of memory.
- What if have hundreds of GBs of RAM?

Come to the poster on Monday to discuss!
Conclusion

- 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.

Source code and examples: [http://graphchi.org](http://graphchi.org)
License: Apache 2.0
Extra Slides
PSW is Asynchronous

- If $V > U$, and there is edge $(U, V, &x) = (V, U, &x)$, $\text{update}(V)$ will observe change to $x$ done by $\text{update}(U)$:
  - Memory-shard for interval $(j+1)$ will contain writes to shard($j$) done on previous intervals.
  - Previous slide: If $U, V$ in the same interval.

- PSW implements the Gauss-Seidel (asynchronous) model of computation
  - Shown to allow, in many cases, clearly faster convergence of computation than Bulk-Synchronous Parallel (BSP).
  - Each edge stored only once.

$$V_i^t \leftarrow F(V_0^t, V_1^t, \ldots, V_{i-1}^t, V_i^{t-1}, V_{i+1}^{t-1}, \ldots)$$
Number of Shards

- If $P$ is in the “dozens”, there is not much effect on performance.
I/O Complexity

• See the paper for theoretical analysis in the Aggarwal-Vitter’s I/O model.
  – Worst-case only 2x best-case.

• Intuition:
Multiple hard-drives (RAIDish)

- GraphChi supports *striping* shards to multiple disks $\rightarrow$ Parallel I/O.

Experiment on a 16-core AMD server (from year 2007).
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.
Computational Setting

• Constraints:
  A. Not enough memory to store the whole graph in memory, nor all the vertex values.
  B. Enough memory to store one vertex and its edges w/ associated values.

• Largest example graph used in the experiments:
  – Yahoo-web graph: 6.7 B edges, 1.4 B vertices

Recently GraphChi has been used on a MacBook Pro to compute with the most recent Twitter follow-graph (last year: 15 B edges)