15-min Review of:

GraphChi

Large-Scale Graph Computation on Just a PC

Presented by Niko Stahl for R212
Context

● **Familiar problem:**
How to process a graph that does not fit into main memory?

● **Previous solutions:**
Distribute: Split data across machines. These approaches have to deal with fault tolerance and can be difficult to debug.
GraphChi’s solution

Efficiently use disks (HDD or SDD) of a single machine.
The key problem with using disks:

Random Access is **slow** for disk-based computation. And naive implementations of graph algorithms require frequent random access.
The Model - Overview

Parallel Sliding Window (PSW) algorithm:

For each vertex \( v \), optimize processing of the subgraph containing \( v \) and its neighbours (predecessor and successors) by maximizing sequential memory access.
The Model - Primitives

- Each vertex is mapped to an interval.
- Each interval stored on a shard, which can be loaded completely into memory (~ order of $10^2$ MB).
- Edges mapped to intervals based on their target vertex.
The Model - An Example

- Vertices are accessed one shard at a time.
- For each shard $s$, the edges containing vertices on $s$ must be accessed to construct the subgraph.
  - in-edges: already in interval $i$
  - out-edges: ordered by source on sliding shards. Therefore, they can be accessed sequentially.
The Model - An Example

Crux: Within each sliding shard access is sequential
Let $P$ be the number of intervals. PSW requires $P$ sequential disk accesses to process each interval. $O(P^2)$ reads/writes at each iteration ($P<1,000$).
PSW Properties

Pros

+ Asynchronous model (not BSP). More efficient because vertices can be processed in any order (this can be useful for some algorithms, e.g. Dijkstra’s shortest path).
+ Extends well to evolving graphs
+ Easier to debug because computation runs on a single machine.

Cons

- Standard vertex queries are not efficient (building neighborhood of a vertex require scan of a memory shard)
Evaluation

Mac Mini
- 8 GB RAM
- 256 GB SSD
- 1 TB HDD
- 2.5 GHz

<table>
<thead>
<tr>
<th>Graph name</th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>live-journal [3]</td>
<td>4.8M</td>
<td>69M</td>
</tr>
<tr>
<td>netflix [6]</td>
<td>0.5M</td>
<td>99M</td>
</tr>
<tr>
<td>domain [47]</td>
<td>26M</td>
<td>0.37B</td>
</tr>
<tr>
<td>twitter-2010 [28]</td>
<td>42M</td>
<td>1.5B</td>
</tr>
<tr>
<td>uk-union [12]</td>
<td>133M</td>
<td>5.4B</td>
</tr>
<tr>
<td>yahoo-web [47]</td>
<td>1.4B</td>
<td>6.6B</td>
</tr>
</tbody>
</table>
Evaluation

**PageRank**
- **Twitter-2010 (1.5B edges)**
  - GraphChi (Mac Mini)
  - Spark (50 machines)

**WebGraph Belief Propagation (U Kang et al.)**
- **Yahoo-web (6.7B edges)**
  - GraphChi (Mac Mini)
  - Pegasus / Hadoop (100 machines)

**Matrix Factorization (Alt. Least Sqr.)**
- **Netflix (99B edges)**
  - GraphChi (Mac Mini)
  - GraphLab v1 (8 cores)

**Triangle Counting**
- **twitter-2010 (1.5B edges)**
  - GraphChi (Mac Mini)
  - Hadoop (1636 machines)
Comparing with GraphLab 2

GraphLab with 512 CPUs (64 machines).

Computation on Twitter Graph:

- **PageRank**: 40x faster than GraphChi
- **Triangle Counting**: 30x faster than GraphChi
- **Note**: In terms of CPU, GraphLab requires 256x the resources
Conclusion & Personal View

- Good for prototyping. Is support for evolving graphs necessary?
- The evaluation does not include setup time.
- The model is intuitive. But it is not clear how cumbersome the programming interface is.
- GraphChi introduces an alternative perspective on large scale graph computation. It relies on algorithms and data structures instead of greater resources.