GraphChi: Large-Scale Computation on Just a PC

6.886
Joana M. F. da Trindade
Apr 5th 2019
Motivation

Large graphs are everywhere: social networks, web graphs, protein interaction, ....

Cannot be naturally decomposed into smaller parts for parallel processing, so just using MapReduce is inefficient.

At the time (2012) some distributed systems tried to address this:
- Specialized graph processing: Pregel, GraphLab
- General: Piccolo and Spark

Born out of frustration with distributed computing: if graph fits into disk, can we perform advanced graph computations on just a personal computer?
Why is it so hard to efficiently use local storage?

This class so far has covered some of the reasons :-)  
- Specialized external versions of algorithms needed to use I/O efficiently

Large graphs (multiple TBs) may be sparse and irregular  
Irregular -> a vertex can be connected to any other vertex; little locality  
Sparse -> graphs with power-law degree distributions have a long tail of nodes with small amount of in/out edges  
Sparsity and irregularity -> random access fetching small amounts of data
Can’t we just throw more machines at the problem?

Costly
- More engineers required to manage cluster infrastructure
- More machines -> higher total energy consumption

Utilization
- W/o efficient external memory algorithms, no guarantee of better utilization (e.g., load imbalance, idle machines waiting for stragglers)

Slower
- Total time = local algo + time for message exchange and cluster coordination
Authors: let’s instead better utilize our single node

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: vertex-centric

- 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)
Computational model: vertex-centric

MyFunc(vertex)
{ // modify neighborhood }

Algorithm 1: Typical vertex update-function
1 Update(vertex) begin
2 x[] ← read values of in- and out-edges of vertex ;
3 vertex.value ← f(x[]);
4 foreach edge of vertex do
5    edge.value ← g(vertex.value, edge.value);
6 end
7 end
Problem: still have random access in vertex-centric

Assumptions:

- Graph is large enough to fit disk, but \textbf{not} in memory
- Data for all in/out edges of \textbf{any single vertex} fit in memory

Requires random reads/writes across vertex partitions to process in/out edges
Options to avoid random access for vertex-centric

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.

[GraphChi OSDI 2012 slides]
This paper: Parallel Sliding Windows (PSW)

Reduce random access by collocating vertex data with edge data:

Process one subgraph at a time in 3 stages: (1) load subgraph from disk, (2) update vertices and edges, and (3) write updated values to disk
GraphChi main execution flow

1. **Load degree data for vertices in interval \([a,b]\)**
   - Preallocate edge arrays and vertex objects.

2. **LOAD FROM DISK...**
   - In-edges and internal out-edges from **memory shard**
   - Out-edges from \(P-1\) **sliding shards**
     - **vertex values**

3. **Execute vertex update-functions**

4. **WRITE TO DISK...**
   - **memory shard** edge data.
   - Edge values of the \(P-1\) **sliding windows**.
   - Updated **vertex values**.

5. **Next interval**
Load stage: example PSW with 4 intervals

Interval 1

- Shard 1: Vertices 1..100
- Shard 2: Vertices 101..700
- Shard 3: Vertices 701..1000
- Shard 4: Vertices 1001..10000

Out-edges arranged in sequence in other shards

Load all in-edges in memory

In-edges for vertices 1..100 sorted by source id
Load stage: example PSW with 4 intervals

Interval 1

Shard 1
- Vertices 1..100

Shard 2
- Vertices 101..700

Shard 3
- Vertices 701..1000

Shard 4
- Vertices 1001..10000

Out-edge blocks
Load stage: example PSW with 4 intervals

Interval 2

Shard 1: Vertices 1..100
Shard 2: Vertices 101..700
Shard 3: Vertices 701..1000
Shard 4: Vertices 1001..10000

Next shard is loaded into memory
Sliding shards move forward to match it
Load stage: example PSW with 4 intervals

Interval 3

Shard 1
Vertices 1..100

Shard 2
Vertices 101..700

Shard 3
Vertices 701..1000

Shard 4
Vertices 1001..10000

Next shard is loaded into memory
Sliding shards move forward to match it
Load stage: example PSW with 4 intervals

Interval 4

Next shard is loaded into memory
Sliding shards move forward to match it

Shard 1
Vertices 1..100

Shard 2
Vertices 101..700

Shard 3
Vertices 701..1000

Shard 4
Vertices 1001..10000
Update stage

Update-function is executed on vertices in the interval

Edges point to in-memory data blocks
Commit stage

Blocks written back to disk

Next load-phase sees preceding writes

[GraphChi OSDI 2012 slides]
Optimization: dynamic selective scheduling

When active set is sparse (e.g., traversal algos), it is inefficient to process all edges

Their solution: coarse-grained selection:

- Further splits shards into sub-indices
- When neighbors are activated, sets a bit mask
- Loops through bit-mask to figure out which sub-indices to split

Other optimizations in the paper, e.g., buffered updates for evolving graphs
Advantages of PSW

Less random access when compared to other solutions:

- Most reads are performed over sequential chunks

P shards requires only $P^2$ random reads (across sliding shards)

Each edge in a single graph full scan:

- Only read up to 2 times
- Only written up to 2 times
Drawbacks of PSW

Costly initial pre-processing to ensure sorting order within shards

- E.g., takes 10 min to load twitter 2010 graph ($|V| = 42M$, $|E| = 1.5B$)

Storage overhead: vertex value is duplicated

Results with selective scheduling not included in the paper

Does not do well when compared to competing system at the time

- PowerGraph (distributed version of Graphlab), which uses Gather-Apply-Scatter (GAS) for vertex-partitioning better suited for power-law
Evaluation

Comparison against inconsistent system configurations

Versus Hadoop-based Pegasus

- Single node GraphChi: 27min; Pegasus w/ 100 machines: 22 min

Versus in-memory systems

- 2x slower than single-node GraphLab
- 2x slower than 50-node Spark
## Evaluation: datasets

<table>
<thead>
<tr>
<th>Graph name</th>
<th>Vertices</th>
<th>Edges</th>
<th>P</th>
<th>Preproc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>live-journal [3]</td>
<td>4.8M</td>
<td>69M</td>
<td>3</td>
<td>0.5 min</td>
</tr>
<tr>
<td>netflix [6]</td>
<td>0.5M</td>
<td>99M</td>
<td>20</td>
<td>1 min</td>
</tr>
<tr>
<td>domain [44]</td>
<td>26M</td>
<td>0.37B</td>
<td>20</td>
<td>2 min</td>
</tr>
<tr>
<td>twitter-2010 [26]</td>
<td>42M</td>
<td>1.5B</td>
<td>20</td>
<td>10 min</td>
</tr>
<tr>
<td>uk-union [11]</td>
<td>133M</td>
<td>5.4B</td>
<td>50</td>
<td>33 min</td>
</tr>
<tr>
<td>yahoo-web [44]</td>
<td>1.4B</td>
<td>6.6B</td>
<td>50</td>
<td>37 min</td>
</tr>
</tbody>
</table>
Evaluation: vs other systems (inconsistent configs)

<table>
<thead>
<tr>
<th>Iter.</th>
<th>Comparative result</th>
<th>GraphChi (Mac Mini)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>GraphLab[30] on AMD server (8 CPUs) 87 s</td>
<td>132 s</td>
</tr>
<tr>
<td>5</td>
<td>Spark [45] with 50 nodes (100 CPUs): 486.6 s</td>
<td>790 s</td>
</tr>
<tr>
<td>100</td>
<td>Stanford GPS, 30 EC2 nodes (60 virt. cores), 144 min</td>
<td>approx. 581 min</td>
</tr>
<tr>
<td>1</td>
<td>Piccolo, 100 EC2 instances (200 cores) 70 s</td>
<td>approx. 26 min</td>
</tr>
<tr>
<td>1</td>
<td>Pegasus (Hadoop) on 100 machines: 22 min</td>
<td>27 min</td>
</tr>
<tr>
<td>10</td>
<td>GraphLab on AMD server: 4.7 min</td>
<td>9.8 min (in-mem)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>40 min (edge-repl.)</td>
</tr>
<tr>
<td></td>
<td>Hadoop, 1636 nodes: 423 min</td>
<td>60 min</td>
</tr>
<tr>
<td>1</td>
<td>PowerGraph, 64 x 8 cores: 3.6 s</td>
<td>158 s</td>
</tr>
<tr>
<td></td>
<td>PowerGraph, 64 x 8 cores: 1.5 min</td>
<td>60 min</td>
</tr>
</tbody>
</table>
Evaluation: micro-benchmarks

- Performance scales linearly as function of disks
- Little benefits from multithreading when comp. complexity is low (saturates I/O)
- Large benefits when using larger blocks, as less I/O is required
Conclusion

Processing graphs that fit in disk but do not fit in memory is challenging.

Authors propose Parallel Sliding Windows (PSW) as a way to reduce random access for graph computation.

Core contribution: on-disk graph partitioning targeted for vertex-centric model that minimizes random reads / writes.

Results were encouraging: beats distributed systems with 10s of nodes.

However, distributed systems with alternative partitioning can still do better (e.g., PowerGraph using GAS for vertex partitioning).