Empowering Fast Incremental Computation Over Large Scale Dynamic Graphs

Charith Wickramaarachchi, Charalampos Chelmis and Viktor Prasanna

University of Southern California

Presented by: Shijie Zhou

University of Southern California
Large-Scale Graph Data

- Online social networks
  - Facebook
  - Twitter
  - Blogger
  - tsū

- Protein interactions

- Air traffic network

- Citation networks

- WWW

- Neural network
Large-Scale Graph Data are “Evolving”

- **Large volume**
  - > 2 B internet users\(^1\)
  - > 1 B active Facebook users\(^1\)
  - > 2.5 M daily active Twitter users\(^2\)

- **High velocity**
  - > 7.5 K Tweets/second\(^1\)
  - > 1.5 K Skype calls/second\(^1\)
  - > 2000k emails/second\(^1\)

---

\(^1\)http://www.internetlivestats.com

\(^2\)http://www.statista.com/
**Vertex-Centric Model (1)**

- Program written thinking as a vertex
- Computation performed at vertex level
- Communication using message passing between vertices
- Computation happens in iterations
  - Super-steps
- Bulk synchronous parallel model

**Example: Single source shortest path**

```java
Compute(Messages msgs) {
    int distance = IsSource(vertex_id()) ? 0 : INF;
    for each m in msgs {
        distance = (distance, m->value())
    }
    if (distance < getValue()) {
        setValue(distance)
        for each e in getOutEdges() {
            sendMessage(e.sink(), distance + e.value())
        }
        voteToHalt()
    }
}
```

Vertex-Centric Model(2)

Example: Single source shortest path
Vertex-Centric Model(3)

Existing Systems
- Google Pregel
- Apache Giraph
- Graph Lab

High latency
Incremental Computation On Large-Scale Graphs

• **Key idea:**
  – Minimize number of re-computations
Approach(1)

Memorization

• Assumptions
  – Deterministic graph algorithm
  – Vertex state at end of any super-step depends only on
    • State at end of previous super-step
    • Incoming messages
  • Memorize incoming messages and state at each super-step
  • Avoid re-computation on updated graph by comparing with the memorized state

1. Mark affected vertices after graph update

<table>
<thead>
<tr>
<th>Type of change</th>
<th>Affected vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex property change</td>
<td>Only the specific vertex</td>
</tr>
<tr>
<td>Vertex addition</td>
<td>The specific vertex and any vertices it points to</td>
</tr>
<tr>
<td>Vertex deletion</td>
<td>All neighbors of the vertex, connected with either incoming or outgoing edges</td>
</tr>
</tbody>
</table>
| Edge addition/deletion  | **Directed:** only the source vertex  
                          | **Undirected:** both ends of the edge                                              |
| Edge property change    | **Directed:** only the source vertex  
                          | **Undirected:** both ends of the edge                                              |

2. Re-execute the vertex if any of following are true
   – Any of the in-coming messages are different from memorized messages at that super-step
   – State is different from memorized state at that super-step
   – Affected vertex

• Advantages
   – Framework takes care of incremental execution

Challenges

• Vertex centric programming model
  – Little computation per vertex
  – Large number of global synchronization steps
    • High communication/computation ratio

• Vertex centric memorization
  – Per-vertex comparisons
    • High comparison cost/compute cost
Our Approach(1)

• **Approach**
  – Multilevel memorization to prune computation
    • Partition level: coarse grain pruning of re-computations
    • Vertex level: fine grain pruning of re-computations
  – Partition centric hierarchical BSP
    • Partition the graph
    • Local barrier synchronization within each partition
    • Global barrier synchronization across partitions
  – Resource allocation
    • 1 node → 1 partition
    • 1 core → subset of vertices
Our Approach (2)

• Programming model
  – Vertex centric
    • Very similar to Pregel model
  – Two levels of iterations
    • Sub-super-steps: intra partition
    • Super-step: inter partition
  – Messaging
    • Intra partition messages sent within sub-super-steps
    • Inter partition messages are aggregated and send at start of each new super-step
  – Reduce operation to limit communication
    • Enable users to minimize communication between partitions
Our Approach(3)

• Advantages
  – Framework takes care of incremental execution
  – Less global barrier synchronization overhead
  – Partition/Sub-graph level pruning of re-computations
    • Less comparison cost / computation cost
Algorithm 1 Max Vertex Using HBSP

1: procedure COMPUTE(Vertex v, Iterator<Messages> msgs)
2:     if super-step == 0 and sub-super-step == 0 then
3:         BROADCASTGREATESTNEIGHBOR(v) ▷ Find the greatest vertex id \( m \) from the neighborhood set (including self), set \( m \) as the current value, and sent it to all neighbors
4:         return
5:     end if
6:     changed ← false
7:     maxId ← v.value
8:     while msgs.hasNext do
9:         m = msgs.next
10:        if maxId < m.value then
11:            maxId ← m.value
12:            change ← true
13:        end if
14:     end while
15:     if changed then
16:         v.value ← maxId
17:         BROADCASTUPDATE(v) ▷ Send the vertex value to all neighbors of \( v \)
18:     end if
19: end procedure
Example(2)

• Reduce Operation
  – Max vertex id

\[
\text{reduce}(\text{I neighborId}, \ \text{List}<\text{Messages}> \ \text{msgs}) \ \{ \\
\begin{align*}
\text{int} \ max &= 0 \\
\text{for each} \ \text{msg in msgs} \ { \\
\text{max} &= \text{max}(\text{max}, \ \text{msg}.\text{value}) \\
\} \\
\text{sendMessage}(\text{neighbour}, \ \text{new Message(\text{max})});
\}
\]

Per each remote vertex with outgoing messages
Implementation

- Implemented vertex centric and hierarchical BSP model on Apache Giraph (1.1.0)
- Local barriers using in-memory data structures. (Semaphores)
- Communication within partition each using in-memory data structures.
- Number of threads = number of cores
- Work stealing to reduce imbalance computation within each worker.
- Vertex to partition mapping is provided by user
- Memorized state was stored at partition level in local memory
Experimental Setup(1)

- **Cluster of 15 nodes**
  - 8-Core Intel Xeon CPU
  - 16GB RAM

- **Workers**
  - Number of workers: 12
  - Memory per worker: 14GB

- **Datasets**

<table>
<thead>
<tr>
<th>Dataset</th>
<th># Vertices</th>
<th># Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>SlashDot (SD)</td>
<td>82,168</td>
<td>948,464</td>
</tr>
<tr>
<td>Road Network - CA (RN)</td>
<td>1,965,206</td>
<td>2,766,607</td>
</tr>
</tbody>
</table>

https://snap.stanford.edu/data/
Experimental Setup (2)

- Algorithms
  - Connected component (CC)
  - Single source shortest path (SSSP)
- Generating random graphs for each dataset
  - 100 new edges added randomly
  - 30 random edges deleted
- Partitioning algorithms
  - Random
  - Metis
- Fraction of computations saved
  - Logged number vertices executed
    - Without memorization (r_e)
    - With memorization (m_e)
  - \((r_e - m_e) / r_e\)
Fraction of Computations Saved

RN: Road network
SD: SlashDot
Reduction in Super-Steps(1)

• SlashDot dataset

On static graphs

On updated graphs using memorization
Reduction in Super-Steps(2)

- Road network dataset

On static graphs

On updated graphs using memorization
Conclusion

• Hierarchical BSP Model
  – Huge reduction on number of super-steps for sparse graphs
  – Simple programming abstraction

• Memorization with HBSP
  – No/Minimal impact on number of saved computations
  – Takes the burden of developing incremental graph algorithms

• Future Work
  – Reduce memorization overhead
    • Memory
    • Computation (Due to comparisons)
  – Combine with existing distributed time series graph processing models