Prlter: A Distributed Framework for Prioritized Iterative Computations

Yanfeng Zhang, Qixin Gao, Lixin Gao, Cuirong Wang

Northeastern University, China
UMass Amherst
Iterative Computation is Everywhere

Youtube video suggestion

PageRank

BFS

Pattern Recognition

Clustering
Large-Scale Iterative Computation

+ Involve massive data sets
++ Process massive data iteratively
+++ Converge fast

Our Goal: a framework supporting iterative computation running in the cloud
Outline

• Prioritized Iteration
• PrIter’s Design & Implementation
• Evaluation
• Conclusions
Parallel Single Source Shortest Path (SSSP): 1
SSSP: 1+3
Parallel SSSP: 1+3+5

Parallel execution
Parallel SSSP: 1+3+5+4
Parallel SSSP: 13 updates
Prioritized SSSP: 1
Prioritized SSSP: 1+1
Prioritized SSSP: 1+1+2
Prioritized SSSP: 1+1+2+1
Prioritized SSSP: 1+1+2+1+1
Prioritized SSSP: 7 updates
PageRank

\[ R^{(k)} = dW R^{(k-1)} + (1 - d)E \]

- \( R \): size-\( n \) ranking vector (\( n \))
- \( W \): \( n \times n \) matrix, \( W_{ij} = 1/\text{deg}(j) \)
- \( d \): damping factor (0\(<d<1\), usually 0.8)
- \( E \): size-\( n \) vector (1/\( n \), 1/\( n \)\(,...,1/\( n \)\))
- \( k \): iteration number
Prioritized PageRank

• Priority is not that obvious as in SSSP
Incremental Update Function

\[
R^{(k)} = dW R^{(k-1)} + (1 - d)E
\]

\[
R^{(k)} = (1 - d)E \sum_{l=0}^{k} d^l W^l + d^k W^k R^0
\]

\[
R^{(i)} = R^{(i-1)} + \Delta R^{(i)}, \quad \Delta R^{(i)} = dW \Delta R^{(i-1)}
\]

\[
R^{(0)} = 0 \quad \Delta R^{(0)} = (1 - d)E
\]
Prioritized Incremental PageRank

\[ R^{(i)} = R^{(i-1)} + \Delta R^{(i)}, \quad \Delta R^{(i)} = dW \Delta R^{(i-1)} \]

Measure convergence speed by \( \| R^{(i)} \|_1 \), monotonously increasing to \( \| R^{(\infty)} \|_1 = 1 \)

\[ \| R^{(i)} \|_1 = \| R^{(i-1)} \|_1 + \| \Delta R^{(i)} \|_1 \]

Suppose in each iteration, update is only applied to the diagonal elements of \( \Delta R^{(i-1)} \), that is all the other entries are temporarily set as 0.

\[ \| \Delta R^{(i-1)} \|_1 = \Delta R^{(i-1)}(j) \]

\[ \| \Delta R^{(i)} \|_1 = \| dW \Delta R^{(i-1)} \|_1 = d \Delta R^{(i-1)}(j) \]

Priority is given to large entries of matrix \( w \)

\[ \| w_j \|_1 = 1 \]

\( W \) is column normalized matrix
Theoretical Results

• Incremental PageRank converges to the same vector as PageRank
• Prioritized Incremental PageRank
  – Converges to the same vector as PageRank
  – Converges faster than Incremental PageRank
• Same can be proved for many other iterative computation
  – Personalized PageRank
  – Katz Measure Computation
  – Label Propagation such as Adsorption
  – Single Source Shortest Path
Existing Frameworks

• MapReduce, Dryad
• Support Iterative Computation in the cloud
  – HaLoop (VLDB ‘10)
  – Spark (HotCloud ‘10)
  – Piccolo (OSDI ‘10)
  – Pregel (SIGMOD ‘10)
  – Twistter (MapReduce ‘10)
  – CIEL (NSDI ‘11)
  – iMapReduce (DataCloud ‘11)
• GraphLab
  – Multicore environment
Outline

• Prioritized Iteration
• PrIter’s Design & Implementation
• Evaluation
• Conclusions
Iterative Processing

Graph Partition (1) → Map 1

Graph Partition (2) → Map 2

Graph Partition (n) → Map n

Shuffle

KV → Reduce 1

KV → Reduce 2

KV → Reduce n
Update StateTable

Input $<K, V>$

1. Map
2. $<K, V>$
3. Update State
4. $<K, state>$
5. Decide Priority
6. $<K, priority>$
7. StateTable
8. StateTable

Diagram:
- Map
- Priority Queue
- UpdateState
- cState
- iState
- Decide Priority
- StateTable
- node
- priority
- iState
- cState
- ...
Extract PriorityQueue

- Extract <K,V>s that have higher priority
  - Sampling method to retrieve a threshold
  - A single pass of StateTable
  - Priority larger than threshold, enqueue
Optimal Queue Size

\[ q^* = \sqrt{\frac{\beta \cdot N \cdot T_{ovhd}}{\alpha \cdot T_{proc}}} \]

- Use the default value
- Set a value based on N
Outline

• Prioritized Iteration
• PrIter’s Design & Implementation
• Evaluation
• Conclusions
Experiment Cluster

• 4-node local cluster
  – E8200 dual-core 2.66GHz CPU, 3GB of RAM, and 160GB storage

• Amazon EC2 cloud
  – 100 medium instances
# Data Sets

## Table 1: Data Sets Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSSP</td>
<td>Facebook</td>
<td>1,204,004</td>
<td>20,492,148</td>
</tr>
<tr>
<td></td>
<td>LiveJournal</td>
<td>4,847,571</td>
<td>68,993,773</td>
</tr>
<tr>
<td></td>
<td>roadCA</td>
<td>1,965,206</td>
<td>5,533,214</td>
</tr>
<tr>
<td>PageRank</td>
<td>Berk-Stan</td>
<td>685,231</td>
<td>7,600,595</td>
</tr>
<tr>
<td></td>
<td>Google</td>
<td>875,713</td>
<td>5,105,039</td>
</tr>
<tr>
<td></td>
<td>Notre dame</td>
<td>325,729</td>
<td>1,497,134</td>
</tr>
<tr>
<td></td>
<td>Synth_WEB</td>
<td>100,000,000</td>
<td>1,216,907,427</td>
</tr>
<tr>
<td>Adsorption</td>
<td>Facebook</td>
<td>1,204,004</td>
<td>20,492,148</td>
</tr>
<tr>
<td></td>
<td>Youtube</td>
<td>311,805</td>
<td>1,761,812</td>
</tr>
<tr>
<td></td>
<td>Amazon</td>
<td>403,394</td>
<td>3,387,388</td>
</tr>
<tr>
<td>ConnComp</td>
<td>Amazon</td>
<td>403,394</td>
<td>3,387,388</td>
</tr>
<tr>
<td></td>
<td>Wiki-talk</td>
<td>2,394,385</td>
<td>5,021,410</td>
</tr>
<tr>
<td></td>
<td>LiveJournal</td>
<td>4,847,571</td>
<td>68,993,773</td>
</tr>
</tbody>
</table>
Convergence Speed (Local Cluster)

SSSP (Facebook)

PageRank (BerkStan)
Convergence Speed (Local Cluster)

- Adsorption (Facebook)
- Connected Components (Amazon)
Convergence Time (EC2)

Pagerank

- Hadoop: 100%
- iMapReduce: 58.5%
- Priter w/o pri: 39.1%
- Priter: 11.0%
Conclusions

• Proven effectiveness of prioritized iterative computation
  – Converge faster
  – Derive the same converged result

• PrIter
  – Distributed framework that supports prioritized iteration
  – Reduces expensive data access by performing more effective update first
  – Achieve up to 50x speedup comparing to Hadoop
  – MapReduce like API

• Our source code is available at
  http://code.google.com/p/priter/
Optimal Queue Size

- The optimal case: update the most prioritized node
  - Need a pass of StateTable (overhead)
  - NOT practical
- The other extreme case: update all nodes in StateTable
  - NOT prioritized iteration
- Less updates (less workload) need more frequent queue extractions
- An optimal queue size that balances these two
Optimal Queue Size

\[
\min_q \left\{ f(q) \cdot T_{proc} + \frac{f(q)}{q} \cdot N \cdot T_{ovhd} \right\}
\]

- \(q\): queue size
- \(f(q)\): total updates needed
- \(f(q)/q\): number of queue extractions
- \(N\): StateTable size
- \(T_{proc}\): processing time for a node
- \(T_{ovhd}\): scanning time for table entry
Optimal Queue Size

- Online estimation
- Set a default value based on N

\[ q^* = \sqrt{\frac{\beta \cdot N \cdot T_{ovhd}}{\alpha \cdot T_{proc}}} \]