Accelerating PageRank in Shared-Memory for Efficient Social Network Graph Analytics

Baofu Huang, Zhidan Liu*, Kaishun Wu
Shenzhen University, China
Graph Analytics

Social Network

Graph is Ubiquitous

Road Network

Web Network

Biological Network
Graph Computing

• Graph applications execute in two conceptual phases: message exchange (ME) and local update (LU)

![Graph Computing in GAS model](image)

**Figure.** Graph Computing in GAS model

PageRank

- Important benchmark for evaluating graph analytic frameworks
- Fundamental node ranking algorithm
  - Iteratively compute weighted sum of neighbor's PR[\(v_i\)]

\[
PR_i + 1(u) = \frac{1 - d}{|V|} + d \sum_{v \in N_i(u)} \frac{PR(v)}{|N_o(v)|}
\]

- where \(d\) is the damp factor, \(N_i(u)\) and \(N_o(v)\) represent \(u\)'s in-neighbors and vertex \(v\)'s out-neighbors, respectively\(^1\)

Efficient PageRank Computing

• As the magnitude of graph data grows rapidly, how to compute PageRank efficiently?
  - Serial computing or parallel computing
  - Single-machine computing or distributed computing

As the magnitude of graph data grows rapidly, how to compute PageRank efficiently?

- Serial computing or parallel computing
- Single-machine computing or distributed computing

Efficient PageRank Computing

• As the magnitude of graph data grows rapidly, how to compute PageRank efficiently?
  - Serial computing or parallel computing
  - Single-machine computing or distributed computing

Many distributed systems can not defeat graph computing in single thread because of their expensive communication cost.

Efficient PageRank Computing

• As the magnitude of graph data grows rapidly, how to compute PageRank efficiently?
  - Serial computing or parallel computing
  - Single-machine computing or distributed computing

Many distributed systems can not defeat graph computing in single thread because of their expensive communication cost

Limitation 1: Push Direction PageRank

**Algorithm.** *Push-based Parallelled PageRank Computing*

```c
parallel_for (int vSrc = 0; vSrc < numVertices; ++vSrc) {
    if (!frontier.contains(vSrc)) continue;
    for (int d = 0; d < vertex[vSrc].outdegree; ++d) {
        const int vDst = vertex[vSrc].outneighbor[d];
        if (converged.contains(vDst)) continue;
        atomicCAS(vertex[vDst].value, compute(vertex[vSrc].value, vertex[vDst].value));
    }
}
```

**Fig 1.** Push Model

**Fig 2.** Multi-thread computing

**Fig 3.** CAS for conflict
**Limitation 1: Push Direction PageRank**

**Algorithm.** *Push-based Parallelled PageRank Computing*

```cpp
parallel_for (int vSrc = 0; vSrc < numVertices; ++vSrc) {
    if (!frontier.contains(vSrc)) continue;
    for (int d = 0; d < vertex[vSrc].outdegree; ++d) {
        const int vDst = vertex[vSrc].outneighbor[d];
        if (converged.contains(vDst)) continue;
        atomicCAS(vertex[vDst].value, compute(vertex[vSrc].value, vertex[vDst].value));
    }
}
```

**Fig 1.** Push Model

**Fig 2.** Multi-thread computing

**Fig 3.** CAS for conflict
Limitation 2: Computing Redundancy

**Limitation 2: Computing Redundancy**

**Pseudocode for PageRank**

1. initData(v): v.rank = 0.15; Δ = -0.85;
2. initMsg: Activate(u), u ∈ V
3. ......

**Fig. example graph**

**1st iteration:**

PR₁[3] = compute{PR₀[1],PR₀[2]} → unconverged
PR₁[1] = compute{PR₀[4]} → unconverged
PR₁[2] = compute{}; → converged
PR₁[4] = compute{}; → converged

**2nd iteration:**

PR₂[3] = compute{PR₁[1]} → unconverged
PR₂[1] = compute{}; → converged

**3rd iteration:**

PR₃[3] = compute{} → converged
**Limitation 2: Computing Redundancy**

**Pseudocode for PageRank**

1. initData(v): v.rank = 0.15; Δ = -0.85;
2. initMsg: Activate($u$), $u \in V$
3. .......

![Example graph](image)

**Fig. example graph**

- **1st iteration:**
  - $PR_1[3] = \text{compute}\{PR_0[1], PR_0[2]\}$ --&gt; unconverged
  - $PR_1[1] = \text{compute}\{PR_0[4]\}$ --&gt; unconverged
  - $PR_1[2] = \text{compute}\{}$; --&gt; converged
  - $PR_1[4] = \text{compute}\{}$; --&gt; converged

- **2nd iteration:**
  - $PR_2[3] = \text{compute}\{PR_1[1]\}$ --&gt; unconverged
  - $PR_2[1] = \text{compute}\{}$; --&gt; converged

- **3rd iteration:**
  - $PR_3[3] = \text{compute}\{}$ --&gt; converged

- A vertex will not converge until all its in-neighbors have become converged.
- Not all vertices need to start computing from the beginning, e.g. vertex 3.
Limitation 3: Communicating Redundancy

\begin{align*}
\text{Gather} \ (v, n) &: \quad \text{return } n.\text{rank}/\#\text{outNbrs}(v) \\
\text{Acc} \ (a, b) &: \quad \text{return } a + b \\
\text{Apply} \ (v, \text{sum}) &: \quad v.\text{rank} = 0.15 + 0.85 \times \text{sum} \\
\text{Scatter} \ (v, n) &: \quad \text{if} \ (\!\text{converged}(v)) \quad \text{activate}(n)
\end{align*}

\textbf{Figure.} The sample code of PageRank on various systems.\footnote{J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: distributed graph parallel computation on natural graphs. InUSENIX OSDI, 2012.}

- Unconverged Vertices have to communicate with their neighbors twice per iteration.
Overview

- Components of APPR
  - Graph Partitioner
  - Degree-aware Scheduler
  - Message Controller
Opt 1: Destination-Centric Graph Partitioning

- Partitioning is done by grouping edges based on destination
- It works well in most cases
Opt 2: Degree-Aware Computation Scheduler

- Low in-degree (L) vertices compute ahead of High in-degree (H) vertices
- This lazy strategy does not affect the correctness of computing results

**Figure.** The number of converged vertices for graph orkut in each iteration.
Opt 3: Message Controller

• How does it work?
  1. The active vertices all push their value to their neighbors
  2. A vertex will push both status and new update to its neighbors at the same time if it's still not converged, or it will take no actions

• In one iteration, a vertex needs to communicate with its neighbors at most once

Figure. Message Controller of APPR
Experimental Setup

• Baseline:
  ✓ pullPR: PullPR implements PageRank in the pull direction\(^1\)
  ✓ pushPR: PushPR implements PageRank in the push direction
  ✓ PCPM: PCPM is the state-of-the-art method that optimizes the parallel PageRank computation based on a partition centric processing methodology

• Platform:
  ✓ Intel(R) Xeon(R) E5-2630 v4 processors @2.20GHz
  ✓ Dual-socket --- 10 cores per socket with 192 GB memory

• Dataset

<table>
<thead>
<tr>
<th>Graph</th>
<th>Description</th>
<th>#vertices(M)</th>
<th>#edges(M)</th>
<th>(d)</th>
<th>Disk size(G)</th>
</tr>
</thead>
<tbody>
<tr>
<td>livej</td>
<td>Social network</td>
<td>7.5</td>
<td>112.3</td>
<td>15</td>
<td>1.6</td>
</tr>
<tr>
<td>twitter</td>
<td>Social network</td>
<td>21.3</td>
<td>265.0</td>
<td>12</td>
<td>5.2</td>
</tr>
<tr>
<td>orkut</td>
<td>Social network</td>
<td>3.0</td>
<td>106.3</td>
<td>35</td>
<td>1.6</td>
</tr>
<tr>
<td>pld</td>
<td>Web Pages</td>
<td>42.9</td>
<td>623.1</td>
<td>15</td>
<td>10.9</td>
</tr>
<tr>
<td>sd</td>
<td>Web Pages</td>
<td>94.9</td>
<td>1937.5</td>
<td>20</td>
<td>34.4</td>
</tr>
</tbody>
</table>

Experimental Results

- Overall performance

<table>
<thead>
<tr>
<th>Graph</th>
<th>PullPR</th>
<th>PushPR</th>
<th>PCPM</th>
<th>APPR</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>livej</td>
<td>1.4</td>
<td>2.5</td>
<td>4.0</td>
<td>1.0</td>
<td>1.4 ~ 4.0</td>
</tr>
<tr>
<td>twitter</td>
<td>5.6</td>
<td>14.6</td>
<td>7.5</td>
<td>3.7</td>
<td>1.5 ~ 3.9</td>
</tr>
<tr>
<td>orkut</td>
<td>1.9</td>
<td>3.0</td>
<td>1.7</td>
<td>0.5</td>
<td>3.4 ~ 6.0</td>
</tr>
<tr>
<td>pld</td>
<td>29.5</td>
<td>59.5</td>
<td>13.6</td>
<td>11.6</td>
<td>1.2 ~ 5.1</td>
</tr>
<tr>
<td>sd</td>
<td>94.9</td>
<td>99.8</td>
<td>35.1</td>
<td>29.5</td>
<td>1.2 ~ 3.4</td>
</tr>
</tbody>
</table>

- Up to 4.0x speedup over PCPM
- Up to 6.0x speedup over PushPR
- Up to 3.8x speedup over PullPR
Experimental Results

• Pre-processing time

<table>
<thead>
<tr>
<th>Method</th>
<th>livej</th>
<th>twitter</th>
<th>orkut</th>
<th>pld</th>
<th>sd</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCPM</td>
<td>0.04</td>
<td>0.34</td>
<td>0.08</td>
<td>0.20</td>
<td>0.54</td>
</tr>
<tr>
<td>APPR</td>
<td>0.11</td>
<td>0.50</td>
<td>0.18</td>
<td>0.55</td>
<td>1.39</td>
</tr>
</tbody>
</table>

- APPR spends slightly more time to pre-process a graph than PCPM
- The pre-processing time of both methods is proportional to the graph size, i.e., a larger graph needs more pre-processing time
Experimental Results

• Evaluation of APPR Design
  - APPR-S <-> APPR without degree-aware scheduler
  - APPR-M <-> APPR without message controller

✓ Degree-aware scheduler brings 14% ~ 24% improvements on execution time
✓ On average, the message controller module accelerates PageRank computation by 104%
See paper for more results ...
Conclusion

• We present APPR to accelerate parallel PageRank computation in the shared-memory platforms for large scale graphs
  - Destination-centric graph partitioner to avoid synchronization issues
  - Degree-aware computation scheduler to reduce unnecessary operations
  - Message controller to improve the efficiency of memory accesses

• APPR outperforms state-of-the-art methods with on average 2.4x speedup in execution time and 16.4x reduction in communication messages for social network graphs
Thank You