iSpan: Parallel Identification of Strongly Connected Components with Spanning Trees

Yuede Ji  Hang Liu  H. Howie Huang

The George Washington University
University of Massachusetts Lowell
Graph is Everywhere

```c
void foo()
{
    int x = source();
    if (x < MAX)
    {
        int y = 2 * x;
        sink(y);
    }
}
```

United States transmission grid
Source: FEMA
Strongly Connected Component (SCC)

• In a directed graph, an SCC is a maximal subset of the vertices that every vertex has a directed path to all the others
• SCC detection will find all the SCCs in the directed graph
SCC Applications

• Topological sort
  • Rank the vertices in a partial order
  • Vertices in one cycle are ranked equally
  • SCC is used to find all the cycles

• Reachability query
  • Whether a vertex can reach another one through a directed path
  • Convert a general directed graph into a directed acyclic graph (DAG) by contracting an SCC as a vertex
Outline

• Background and Observations
• iSpan
• Experiment
• Conclusion
Background: SCC Property

- Power-law property
  - A single large SCC takes the majority of the vertices
  - The rest are a large number of small SCCs, especially size-1 SCC

Flickr graph,
- $|V| = 2,302,926$
- $|SCC| = 485,572$

- 426,926 size-1 SCCs (88% of SCCs)
- 58,635 (12%) other small SCCs, size < 150
- One large SCC, 70% vertices
Prior Work

- Large SCC
  - Forward-Backward (FW-BW) algorithm
- Small SCC
  - Applies Trim-1/2 to quickly remove the size-1/2 SCCs
  - Applies FW-BW algorithm again for the rest

BFS FW-BW [SC’13]; Multistep [IPDPS’14]
FW-BW Algorithm

Forward-Backward (FW-BW) algorithm takes ~80% of total runtime
Observation

- The FW-BW can be done by any spanning tree construction algorithm.
- BFS produces the correct levels, which is not only unnecessary for SCC, but also requires synchronization and thus limits the performance.

Forward spanning tree

Backward spanning tree

Detected SCC
iSpan Techniques

- Relaxed synchronization (Rsync)
- Fast spanning tree construction method
- Scale to multiple machines
Relaxed Synchronization (Rsync)

Current Sync BFS (bottom-up)
1 foreach unvisited vertex \( v \) in parallel do
2    foreach vertex \( w \) in InNeighbor(\( v \)) do
3        if visit[\( w \)] == level then
4            visit[\( v \)] = level + 1;
5        break;
6    barrier();
...

Rsync
1 foreach unvisited vertex \( v \) in parallel do
2    foreach vertex \( w \) in InNeighbor(\( v \)) do
3        if visit[\( w \)] is visited then
4            visit[\( v \)] = visited;
5        break;
6    barrier();
...

![Sync BFS Tree](image1)
![Rsync Spanning Tree](image2)
**Rsync Benefits**

- The vertex that should be visited at later level can be visited earlier
- Benefit #1: Reduce the number of synchronizations required
  - 18% for forward traversal
  - 25% for backward traversal

---

**The number of synchronizations of Sync over Rsync**

<table>
<thead>
<tr>
<th></th>
<th>Forward</th>
<th>Backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>BD</td>
<td>1.2</td>
<td>1</td>
</tr>
<tr>
<td>DB</td>
<td>1.3</td>
<td>1.2</td>
</tr>
<tr>
<td>FB</td>
<td>1.4</td>
<td>1.3</td>
</tr>
<tr>
<td>FL</td>
<td>1.5</td>
<td>1.4</td>
</tr>
<tr>
<td>HD</td>
<td>1.6</td>
<td>1.5</td>
</tr>
<tr>
<td>LJ</td>
<td>1.7</td>
<td>1.6</td>
</tr>
<tr>
<td>PK</td>
<td>1.8</td>
<td>1.7</td>
</tr>
<tr>
<td>TW</td>
<td>1.9</td>
<td>1.8</td>
</tr>
<tr>
<td>WC</td>
<td>2</td>
<td>1.9</td>
</tr>
<tr>
<td>WE</td>
<td>2.1</td>
<td>2</td>
</tr>
<tr>
<td>WL</td>
<td>2.2</td>
<td>2.1</td>
</tr>
<tr>
<td>WT</td>
<td>2.3</td>
<td>2.2</td>
</tr>
<tr>
<td>RD</td>
<td>2.4</td>
<td>2.3</td>
</tr>
<tr>
<td>RM</td>
<td>2.5</td>
<td>2.4</td>
</tr>
<tr>
<td>AVG</td>
<td>1.5</td>
<td>1.5</td>
</tr>
</tbody>
</table>
Rsync Benefits

- A vertex can be early terminated by newly visited vertices
- Benefit #2: Reduce the number of traversed edges
  - 2.7x for forward traversal
  - 3.3x for backward traversal

The traversed edge number of Sync over Rsync

- **Sync / Rsync**
- **Forward**
- **Backward**

<table>
<thead>
<tr>
<th>RM</th>
<th>FL</th>
<th>HD</th>
<th>WL</th>
<th>TW</th>
<th>RD</th>
<th>PK</th>
<th>BD</th>
<th>WC</th>
<th>FB</th>
<th>DB</th>
<th>WT</th>
<th>LJ</th>
<th>WE</th>
<th>AVG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fast Spanning Tree Construction

• Direction-optimizing BFS [SC’12]
  • Top-down - visited vertex find unvisited
  • Bottom-up - unvisited vertex find visited
• Switch based on the workload
  • Starts from top-down
  • Switches to bottom-up in the middle (majority of the time)
  • Switches back to top-down
Async Top-down

• Asynchronous traversal (Async)
  • Each thread takes equal partition at the beginning
  • Each thread works on its own partition till the end
    • does not incur any synchronization
• Long tail frontier distribution
  • Frontier is the number of visited vertices in one level
• Comparison (switch at 35-th level)
  • Sync top-down requires 1,326 levels (thus synchronizations) vs Async top-down 1 level

Long tail in the Wikipedia graph
Benefits of Fast Spanning Tree Construction

- Large SCC (FW-BW)
  - Speedup over our optimized direction-optimizing BFS
  - 1.1x with Async top-down,
  - 1.6x with Rsync bottom-up
  - 1.8x with Async + Rsync

Benefits of Different Techniques on the Large SCC
Distribute iSpan to Multiple Machines

- **Challenge:**
  - High communication cost
  - Vertex-centric 1-D partition

- **Large SCC**
  - Bitwise status compression
    - Visited, unvisited, newly visited
    - 2 bits for one status, thus $|V|/8$ bytes
  - Frontier queue (FQ)
    - Only the newly visited vertices
  - Hybrid queue - $\min(|V|/8, |FQ|)$

- **Small SCCs**
  - Compact the graph for the remaining vertices and edges
    - Average 2.1% vertices and 0.5% edges left
  - Replicate this graph on every machine
    - No communication is required
Outline

• Background and Observations
• iSpan
• Experiment
• Conclusion
Experiments

- Hardware
  - A single server with two Intel Xeon E5-2683 (2.00 GHz) CPUs with 56 hardware threads
  - 32 nodes on university clusters
- Software
  - GCC 4.8.5
  - OpenMP 3.1
  - OpenMPI 1.8
- Report average of 10 runs
- Use the same configuration for comparison
Graph Benchmarks

- 17 graphs
- 14 real-world graphs
  - Konect, University of Koblenz-Landau
  - SNAP, Stanford
- 3 synthetic graphs
  - R-MAT
  - Kron
- 17 graphs
- 14 real-world graphs
  - Konect, University of Koblenz-Landau
  - SNAP, Stanford
- 3 synthetic graphs
  - R-MAT
  - Kron

<table>
<thead>
<tr>
<th>Graph (Abbr.)</th>
<th># of SCC</th>
<th>Large SCC Size</th>
<th># of SCC</th>
<th>Large SCC Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baidu (BD)</td>
<td></td>
<td></td>
<td>1,503,004</td>
<td>609,905</td>
</tr>
<tr>
<td>Dbpedia (DB)</td>
<td></td>
<td></td>
<td>3,636,316</td>
<td>178,593</td>
</tr>
<tr>
<td>Flickr (FL)</td>
<td></td>
<td></td>
<td>485,572</td>
<td>1,605,184</td>
</tr>
<tr>
<td>Hudong (HD)</td>
<td></td>
<td></td>
<td>2,189,120</td>
<td>185,668</td>
</tr>
<tr>
<td>Livejournal (LJ)</td>
<td></td>
<td></td>
<td>971,233</td>
<td>3,828,682</td>
</tr>
<tr>
<td>Pokec (PK)</td>
<td></td>
<td></td>
<td>325,893</td>
<td>1,304,537</td>
</tr>
<tr>
<td>Wiki_comm (WC)</td>
<td></td>
<td></td>
<td>2,281,880</td>
<td>111,881</td>
</tr>
<tr>
<td>Wikipedia_en (WE)</td>
<td></td>
<td></td>
<td>14,459,547</td>
<td>3,796,073</td>
</tr>
<tr>
<td>Wikipedia_link (WL)</td>
<td></td>
<td></td>
<td>4,266,559</td>
<td>6,916,926</td>
</tr>
<tr>
<td>Wiki_talk_en (WT)</td>
<td></td>
<td></td>
<td>2,736,716</td>
<td>249,610</td>
</tr>
<tr>
<td>Twitter_www (TW)</td>
<td></td>
<td></td>
<td>8,044,729</td>
<td>33,479,734</td>
</tr>
<tr>
<td>Facebook (FB)</td>
<td></td>
<td></td>
<td>93,892,292</td>
<td>2,186,877</td>
</tr>
<tr>
<td>Twitter_mpi (TM)</td>
<td></td>
<td></td>
<td>12,407,622</td>
<td>40,012,384</td>
</tr>
<tr>
<td>Friendster (FR)</td>
<td></td>
<td></td>
<td>48,928,140</td>
<td></td>
</tr>
<tr>
<td>Random (RD)</td>
<td></td>
<td></td>
<td>2</td>
<td>4,000,000</td>
</tr>
<tr>
<td>R-MAT (RM)</td>
<td></td>
<td></td>
<td>2,105,950</td>
<td>1,894,035</td>
</tr>
<tr>
<td>Kron_30 (KR)</td>
<td></td>
<td></td>
<td>364,818,982</td>
<td></td>
</tr>
</tbody>
</table>
Speedup of iSpan over Related Work

- DFS style work: **67x** over Tarjan [SICOMP’72]; **21x** over UFSCC [PPoPP’16];
- BFS style work: **4.1x** over BFS FW-BW [SC’13]; **3.6x** over Multistep [IPDPS’14]
• iSpan achieves average 11x speedup over baseline, Multistep 9.6x, BFS FW-BW 5.5x

• For billion vertex graph, iSpan gets 19x, Multistep 10x
Scale to Multiple Machines

- Use four representative graphs, LJ, FL (social graph), WE (web graph), RD (synthetic graph)
- Compare with HPCGraph [IPDPS’16]
  - Better scalability for WE, FL, and RD
  - Worse scalability for LJ - faster single node runtime, 8x
Conclusion

• New insight: spanning tree is sufficient for SCC
• New relaxed synchronization approach
• New spanning tree construction method
• Results:
  • Outperform current state-of-the-art DFS and BFS-based methods by average 18x and 4x
  • Scale to billion vertex graph on tens of machines
Thank You

Check out our graph code repository at [github.com/iHeartGraph/](https://github.com/iHeartGraph/)