Parallel Strongly Connected Components in Shared Memory Architectures

George M. Slota\textsuperscript{2}  Sivasankaran Rajamanickam\textsuperscript{1}
Kamesh Madduri\textsuperscript{2}

\textsuperscript{1}Sandia National Laboratories
\textsuperscript{2}Pennsylvania State University

21 February 2014
Overview

- Introduction
- Previous parallel strongly connected component (SCC) algorithms
- Multistep: Our shared memory parallel algorithm
- Performance results
- Conclusions and Future work
Motivation

- Block Triangular Form (BTF): Useful in shared memory parallel direct and incomplete factorizations.
- Computing the strongly connected components (SCCs) of a matrix is key for computing the BTF.
- SCCs are also useful in formal verification and analyzing web-graphs.
- SCCs algorithms are also a good candidate to study task-parallel vs data-parallel algorithms in the existing architectures with the available runtime systems.
Introduction

- Computing strongly connected components (SCCs) refers to detection of all maximal strongly connected sub-graphs in a large directed graph.
- A strongly connected subgraph is a subgraph in which there is a path from every vertex to every other vertex.
- Standard sequential algorithm is Tarjan’s algorithm
  - DFS based recursive algorithm.
  - Not amenable to a scalable parallel algorithm.
Previous Parallel SCC Algorithms

- Forward-Backward (FW-BW) (Hendrickson, Pinar, Plimpton, Fleischer, McIendon)
- Coloring (Orzan)
- Task parallel, but own runtime with algorithmic improvements (Hong et al, SC 2013)
- Others (Barnat et al)
Our Contributions

- A Multistep method for SCC detection:
  - Data parallel SCC detection with the advantages of previous methods.
  - Uses minimal synchronization and fine-grained locking.
- Faster and scales better than the previous methods.
- Up to 9x faster than state-of-the-art Hong et al’s method.

Previous Algorithms
Forward-Backward (FW-BW)

Select pivot
Find all vertices that can be reached from the pivot (descendant $D$)
Find all vertices that can reach the pivot (predecessor $P$)
Intersection of those two sets is an SCC ($S = P \cap D$)
Now have three distinct sets leftover ($D \setminus S$, $P \setminus S$, and remainder $R$)
Previous Algorithms
Forward-Backward (FW-BW)

- Select pivot
Previous Algorithms
Forward-Backward (FW-BW)

- Select pivot
- Find all vertices that can be reached from the pivot (descendant \((D)\))
Previous Algorithms
Forward-Backward (FW-BW)

- Select pivot
- Find all vertices that can be reached from the pivot (descendant $D$)

Intersection of those two sets is an SCC ($S = P \cap D$)

Now have three distinct sets leftover ($D \setminus S$), ($P \setminus S$), and remainder ($R$)
Previous Algorithms
Forward-Backward (FW-BW)

- Select pivot
- Find all vertices that can be reached from the pivot (descendant \((D)\))
- Find all vertices that can reach the pivot (predecessor \((P)\))
Previous Algorithms
Forward-Backward (FW-BW)

- Select pivot
- Find all vertices that can be reached from the pivot (descendant \((D)\))
- Find all vertices that can reach the pivot (predecessor \((P)\))
Previous Algorithms
Forward-Backward (FW-BW)

- Select pivot
- Find all vertices that can be reached from the pivot (descendant ($D$))
- Find all vertices that can reach the pivot (predecessor ($P$))
Previous Algorithms
Forward-Backward (FW-BW)

- Select pivot
- Find all vertices that can be reached from the pivot (*descendant* \((D)\))
- Find all vertices that can reach the pivot (*predecessor* \((P)\))
Select pivot
Find all vertices that can be reached from the pivot (descendant \( D \))
Find all vertices that can reach the pivot (predecessor \( P \))
Intersection of those two sets is an SCC \( (S = P \cap D) \)
Previous Algorithms
Forward-Backward (FW-BW)

- Select pivot
- Find all vertices that can be reached from the pivot (descendant $(D)$)
- Find all vertices that can reach the pivot (predecessor $(P)$)
- Intersection of those two sets is an SCC $(S = P \cap D)$
- Now have three distinct sets leftover $(D \setminus S), (P \setminus S)$, and remainder $(R)$
Forward-Backward (FW-BW) Algorithm

1: procedure FW-BW(V)
2:   if $V = \emptyset$ then
3:     return $\emptyset$
4:   Select a pivot $u \in V$
5:   $D \leftarrow \text{BFS}(G(V, E(V)), u)$
6:   $P \leftarrow \text{BFS}(G(V, E'(V)), u)$
7:   $R \leftarrow (V \setminus (P \cup D))$
8:   $S \leftarrow (P \cap D)$
9:   new task do FW-BW($D \setminus S$)
10:  new task do FW-BW($P \setminus S$)
11:  new task do FW-BW($R$)
Previous Algorithms
Trimming

- Used to find trivial SCCs
Previous Algorithms
Trimming

- Used to find trivial SCCs
- Detect and prune all vertices that have an in/out degree of 0 or an in/out degree of 1 with a self loop (simple trimming)
Previous Algorithms
Trimming

- Used to find trivial SCCs
- Detect and prune all vertices that have an in/out degree of 0 or an in/out degree of 1 with a self loop (simple trimming)
Previous Algorithms

Trimming

- Used to find trivial SCCs
- Detect and prune all vertices that have an in/out degree of 0 or an in/out degree of 1 with a self loop (simple trimming)
- Repeat iteratively until no more vertices can be removed (complete trimming)
Previous Algorithms
Trimming

- Used to find trivial SCCs
- Detect and prune all vertices that have an in/out degree of 0 or an in/out degree of 1 with a self loop (simple trimming)
- Repeat iteratively until no more vertices can be removed (complete trimming)
Consider vertex identifiers as *colors*. 

- Highest colors are propagated forward through the network to create sets. 
- Consider the original vertex of each color to be the root of a new SCC. 
- Each SCC is all vertices (of the same color as the root) reachable backward from each root. 
- Remove found SCCs, reset colors, and repeat until no vertices remain.
Previous Algorithms
Coloring

- Consider vertex identifiers as *colors*
- Highest colors are propagated **forward** through the network to create sets
Consider vertex identifiers as *colors*

- Highest colors are propagated **forward** through the network to create sets
Consider vertex identifiers as *colors*

- Highest colors are propagated **forward** through the network to create sets

Consider the original vertex of each color to be the root of a new SCC. Each SCC is all vertices (of the same color as the root) reachable backward from each root.

Remove found SCCs, reset colors, and repeat until no vertices remain.
Previous Algorithms

Coloring

- Consider vertex identifiers as *colors*
- Highest colors are propagated **forward** through the network to create sets
- Consider the original vertex of each color to be the *root* of a new SCC
Previous Algorithms

Coloring

- Consider vertex identifiers as *colors*
- Highest colors are propagated **forward** through the network to create sets
- Consider the original vertex of each color to be the *root* of a new SCC
- Each SCC is all vertices (of the same color as the root) reachable **backward** from each root.
Previous Algorithms
Coloring

- Consider vertex identifiers as *colors*
- Highest colors are propagated **forward** through the network to create sets
- Consider the original vertex of each color to be the *root* of a new SCC
- Each SCC is all vertices (of the same color as the root) reachable **backward** from each root.
Previous Algorithms

Coloring

- Consider vertex identifiers as *colors*
- Highest colors are propagated ***forward*** through the network to create sets
- Consider the original vertex of each color to be the *root* of a new SCC
- Each SCC is all vertices (of the same color as the root) reachable ***backward*** from each root.
Previous Algorithms

Coloring

- Consider vertex identifiers as *colors*
- Highest colors are propagated **forward** through the network to create sets
- Consider the original vertex of each color to be the *root* of a new SCC
- Each SCC is all vertices (of the same color as the root) reachable **backward** from each root.
Previous Algorithms
Coloring

- Consider vertex identifiers as *colors*
- Highest colors are propagated **forward** through the network to create sets
- Consider the original vertex of each color to be the *root* of a new SCC
- Each SCC is all vertices (of the same color as the root) reachable **backward** from each root.
- Remove found SCCs, reset colors, and repeat until no vertices remain
Previous Algorithms
Coloring

- Consider vertex identifiers as \textit{colors}
- Highest colors are propagated \textbf{forward} through the network to create sets
- Consider the original vertex of each color to be the \textit{root} of a new SCC
- Each SCC is all vertices (of the same color as the root) reachable \textbf{backward} from each root.
- Remove found SCCs, reset colors, and repeat until no vertices remain
1: procedure \textsc{ColorSCC}(G(V, E))
2: \hspace{1em} while $G \neq \emptyset$ do
3: \hspace{2em} for all $u \in V$ do $\text{Colors}(u) \leftarrow u$
4: \hspace{1em} while at least one vertex has changed colors do
5: \hspace{2em} for all $u \in V$ in parallel do
6: \hspace{3em} for all $(u, v) \in E$ do
7: \hspace{4em} if $\text{Colors}(u) > \text{Colors}(v)$ then
8: \hspace{5em} $\text{Colors}(v) \leftarrow \text{Colors}(u)$
9: \hspace{2em} for all unique $c \in \text{Colors}$ in parallel do
10: \hspace{3em} $V_c \leftarrow \{u \in V : \text{Colors}(u) = c\}$
11: \hspace{3em} $SCV_c \leftarrow \text{BFS}(G(V_c, E'(V_c)), u)$
12: $V \leftarrow (V \setminus SCV_c)$
Barnat et al. (2011)
- Evaluated coloring, FW-BW, and several other algorithms running in parallel on CPU and Nvidia CUDA platform

Hong et al. (2013)
- Parallel FW-BW with 1 and 2 sized SCC trimming, set partitioning after finding largest SCC based on WCCs, in-house task queue for load balancing
Current Implementation

Observations

- FW-BW can be efficient at finding large SCCs, but when there are many small disconnected ones, the remainder set will dominate, creating a large work imbalance.
  - Current implementation of tasks has a huge overhead. Finding SCC of size one is terribly inefficient with a new task.
- Coloring is very inefficient at finding a large SCC, but is efficient at finding many small ones.
  - Data parallel, but colors reassigned multiple times in a large SCC.
- Tarjan’s [6] serial algorithm runs extremely quick for a small number of vertices. (100K)
- Most real-world graphs have one giant SCC and many many small SCCs.

Multistep: combine the best of these methods.
Multistep Method

1: procedure \textsc{Multistep}(G(V, E))
2: \hspace{1em} T ← MS-SimpleTrim(G)
3: \hspace{1em} V ← V \setminus T
4: \hspace{1em} Select \( v \in V \) for which \( d_{in}(v) \cdot d_{out}(v) \) is maximal
5: \hspace{1em} D ← BFS(G(V, E(V)), v)
6: \hspace{1em} S ← D \cap BFS(G(D, E'(D)), v)
7: \hspace{1em} V ← V \setminus S
8: \hspace{1em} \textbf{while} NumVerts(V) > n_{cutoff} \hspace{1em} \textbf{do}
9: \hspace{2em} C ← MS-Coloring(G(V, E(V)))
10: \hspace{2em} V ← V \setminus C
11: \hspace{1em} \text{Tarjan}(G(V, E(V)))

- Do simple trimming
- Perform single iteration of FW-BW to remove giant SCC
- Do coloring until some threshold of remaining vertices is reached
- Finish with serial algorithm
Multistep Method

- Since we don’t care about \((D \setminus S), (P \setminus S), R\) sets, we only need to look for \((S = P \cap D)\)
Multistep Method

- Since we don’t care about $(D \setminus S), (P \setminus S), R$ sets, we only need to look for $(S = P \cap D)$
- Begin as before, select pivot and find all of $(D)$
Multistep Method

- Since we don’t care about \((D \setminus S), (P \setminus S), R\) sets, we only need to look for \((S = P \cap D)\).
- Begin as before, select pivot and find all of \((D)\).
- For backward search, only consider vertices already marked in \((D)\).
Multistep Method

- Since we don’t care about \((D \setminus S), (P \setminus S), R\) sets, we only need to look for \((S = P \cap D)\)
- Begin as before, select pivot and find all of \((D)\)
- For backward search, only consider vertices already marked in \((D)\)
Multistep Method

- Since we don’t care about \((D \setminus S), (P \setminus S), R\) sets, we only need to look for \((S = P \cap D)\)
- Begin as before, select pivot and find all of \((D)\)
- For backward search, only consider vertices already marked in \((D)\)
- For certain graphs, this can dramatically decrease the search space
Implementation Details

- Simple trimming can be implemented using flip of a boolean array.
- Complete trimming also needs a current and future queues for parallel performance. (Thread private queues combined at the end of an iteration).
- BFS uses thread local queues as well.
- “visited” array is not a bit map, but a boolean.
  - more accesses to “visited” than BFS
  - less arithmetic to find the index
  - guaranteed atomic read/writes at byte level (Intel IA-32, Intel 64)
- Per socket graph partitioning did not help performance
- “Direction-optimizing” BFS (Beamer et al) is used as well.
Performance Results

Test Algorithms

- **Multistep**: Simple trimming, parallel BFS, coloring until less than 100k vertices remain, serial Tarjan
- **FW-BW**: Complete trimming, FW-BW algorithm until completion
- **Coloring**: Coloring.
- **Serial**: Serial Tarjan
- **Hong et al**: FW-BW, custom task queue.
Performance Results
Test Environment and Graphs

- Compton (Intel): Xeon E5-2670 (Sandybridge), dual socket, 16 cores.

<table>
<thead>
<tr>
<th>Network</th>
<th>n</th>
<th>m</th>
<th>( \text{deg} )</th>
<th>( \tilde{D} )</th>
<th>(S)CCs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>avg</td>
<td>max</td>
<td></td>
<td>count</td>
<td>max</td>
</tr>
<tr>
<td>Twitter</td>
<td>53M</td>
<td>2000M</td>
<td>37</td>
<td>780K</td>
<td>19</td>
</tr>
<tr>
<td>ItWeb</td>
<td>41M</td>
<td>1200M</td>
<td>28</td>
<td>10K</td>
<td>830</td>
</tr>
<tr>
<td>WikiLinks</td>
<td>26M</td>
<td>600M</td>
<td>23</td>
<td>39K</td>
<td>170</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>4.8M</td>
<td>69M</td>
<td>14</td>
<td>20K</td>
<td>18</td>
</tr>
<tr>
<td>XyceTest</td>
<td>1.9M</td>
<td>8.3M</td>
<td>4.2</td>
<td>246</td>
<td>93</td>
</tr>
<tr>
<td>RDF_Data</td>
<td>1.9M</td>
<td>130M</td>
<td>70</td>
<td>10K</td>
<td>7</td>
</tr>
<tr>
<td>RDF_linkedct</td>
<td>15M</td>
<td>34M</td>
<td>2.3</td>
<td>72K</td>
<td>13</td>
</tr>
<tr>
<td>R-MAT_20</td>
<td>0.56M</td>
<td>8.4M</td>
<td>15</td>
<td>24K</td>
<td>9</td>
</tr>
<tr>
<td>R-MAT_22</td>
<td>2.1M</td>
<td>34M</td>
<td>16</td>
<td>60K</td>
<td>9</td>
</tr>
<tr>
<td>R-MAT_24</td>
<td>7.7M</td>
<td>130M</td>
<td>17</td>
<td>150K</td>
<td>9</td>
</tr>
<tr>
<td>GNP_1</td>
<td>10M</td>
<td>200M</td>
<td>20</td>
<td>49</td>
<td>7</td>
</tr>
<tr>
<td>GNP_10</td>
<td>10M</td>
<td>200M</td>
<td>20</td>
<td>49</td>
<td>7</td>
</tr>
</tbody>
</table>
Performance Results
Trimming Options

- Doing complete trimming isn’t always the best choice for multistep; sometimes even no trimming is fastest; extra trimming work is handled better by coloring or serial algorithm.
- Complete is almost always the best choice when doing FW-BW.
Performance Results

Timing Breakdown

- The graph structure determines the runtime of different stages
- Large number of SCCs affects FW-BW (tasking overhead)
- Large diameter or a large SCC affects coloring
Both Multistep and Hong et al scale well in most graphs.

Lots of small non-trivial SCCs in ItWeb affects the performance of Hong et all.

Relative to Tarzan’s Algorithm Multistep results in better speedups.
## Performance Results
### Runtime and Speedups

<table>
<thead>
<tr>
<th>Network</th>
<th>Serial</th>
<th>Execution time (s)</th>
<th>MS</th>
<th>Hong</th>
<th>FW-BW</th>
<th>Color</th>
<th>MS Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter</td>
<td>33.0</td>
<td>1.60</td>
<td>2.6</td>
<td>120.00</td>
<td>40.0</td>
<td>20.0×</td>
<td>1.6×</td>
</tr>
<tr>
<td>ItWeb</td>
<td>6.7</td>
<td>1.80</td>
<td>16.0</td>
<td>1400.00</td>
<td>7.1</td>
<td>3.6×</td>
<td>3.6×</td>
</tr>
<tr>
<td>WikiLinks</td>
<td>4.9</td>
<td>0.90</td>
<td>0.98</td>
<td>270.00</td>
<td>9.3</td>
<td>5.5×</td>
<td>1.1×</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>1.3</td>
<td>0.11</td>
<td>0.20</td>
<td>4.10</td>
<td>1.6</td>
<td>12.0×</td>
<td>1.9×</td>
</tr>
<tr>
<td>XyceTest</td>
<td>0.2</td>
<td>0.04</td>
<td>0.08</td>
<td>0.07</td>
<td>0.37</td>
<td>4.7×</td>
<td>1.9×</td>
</tr>
<tr>
<td>R-MAT_24</td>
<td>2.4</td>
<td>0.25</td>
<td>0.25</td>
<td>0.62</td>
<td>2.4</td>
<td>9.5×</td>
<td>1.0×</td>
</tr>
<tr>
<td>GNP_1</td>
<td>7.2</td>
<td>0.15</td>
<td>0.30</td>
<td>1.60</td>
<td>6.5</td>
<td>47.0×</td>
<td>1.9×</td>
</tr>
<tr>
<td>GNP_10</td>
<td>5.5</td>
<td>2.90</td>
<td>5.10</td>
<td>1.20</td>
<td>3.5</td>
<td>1.9×</td>
<td>0.6×</td>
</tr>
</tbody>
</table>
Conclusions and Future work

- New Multistep algorithm for computing the SCCs.
- Faster than three different algorithms on a variety of graphs.
- Current state of task parallelism in OpenMP/TBB is not fine-grained enough for these algorithms.
- Testing this out in Intel MICs and compare performance.
Bibliography


