BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems

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

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

20 May 2014
Overview

- Introduction
- Previous parallel strongly connected component (SCC) algorithms
- Multistep: Our shared memory parallel algorithm
- Extension to CC and WCC
- 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, Mclendon)
- 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.
- Easily extendable to computing connected and weakly connected components
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 \( P \))
Previous Algorithms
Forward-Backward (FW-BW)

- Select pivot
- Find all vertices that can be reached from the pivot (\textit{descendant} \((D)\))
- Find all vertices that can reach the pivot (\textit{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 = ∅ then
3:         return ∅
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*.
Consider vertex identifiers as *colors*

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

- Consider vertex identifiers as *colors*
- Highest colors are propagated **forward** through the network to create sets
Consider vertex identifiers as \textit{colors}.

- Highest colors are propagated \textbf{forward} through the network to create sets.

Remove found SCCs, reset colors, and repeat until no vertices remain.
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.
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 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.
1: procedure COLORSCC($G(V, E)$) 
2:    while $G \neq \emptyset$ do 
3:        for all $u \in V$ do Colors($u$) ← $u$
4:    while at least one vertex has changed colors do 
5:        for all $u \in V$ in parallel do 
6:            for all $(u, v) \in E$ do 
7:                if Colors($u$) > Colors($v$) then 
8:                    Colors($v$) ← Colors($u$)
9:    for all unique $c \in$ Colors in parallel do 
10:        $V_c$ ← $\{u \in V :$ Colors($u$) = $c\}$
11:        $SCCV_c$ ← BFS($G(V_c, E'(V_c))$, $u$)
12:    $V$ ← ($V \setminus SCCV_c$)
Barnat et al. (2011)
- Evaluated coloring, FW-BW, and several other variants 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
  - Using tasks for finding small SCCs has a lot of overhead, even for efficient tasking implementations
- 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 small SCCs
- Multistep: combine the best of these methods
Multistep Method

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

- 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
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 and coloring use thread local queues as well.
- “visited” array is not a bitmap, 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.
Implementation Details
Extending Multistep to CC and WCC

1: procedure MULTISTEP-(W)CC(G(V, E))
2: \( T \leftarrow \text{MS-SimpleTrim}(G) \)
3: \( V \leftarrow V \setminus T \)
4: Select \( v \in V \) for which \( d_{in}(v) \times d_{out}(v) \) is maximal
5: \( S \leftarrow \text{BFS}(G(V, E(V) \cup E'(V)), v) \)
6: \( V \leftarrow V \setminus S \)
7: while NumVerts(V) > \( n_{\text{cutoff}} \) do
8: \( C \leftarrow \text{MS-Coloring}(G(V, E(V) \cup E'(V))) \)
9: \( V \leftarrow V \setminus C \)
10: BFS-(W)CC(G(V, E(V) \cup E'(V)))

- Simple to extend Multistep idea to CC, WCC
- Trim zero degree verts
- Run single BFS including both in and out edges for WCC
- Perform Coloring with both in and out edges
- Run standard serial BFS algorithm for (W)CC with remainder
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.
- **Multistep-(W)CC**: Multistep for CC and WCC
- **Ligra**: Ligra CC coloring implementation (Shun and Blelloch PPoPP13)
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>$deg$ avg</th>
<th>$deg$ max</th>
<th>$\tilde{D}$</th>
<th>(S)CCs count</th>
<th>(S)CCs max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Twitter</td>
<td>53M</td>
<td>2000M</td>
<td>37</td>
<td>780K</td>
<td>19</td>
<td>12M</td>
<td>41M</td>
</tr>
<tr>
<td>ItWeb</td>
<td>41M</td>
<td>1200M</td>
<td>28</td>
<td>10K</td>
<td>830</td>
<td>30M</td>
<td>6.8M</td>
</tr>
<tr>
<td>WikiLinks</td>
<td>26M</td>
<td>600M</td>
<td>23</td>
<td>39K</td>
<td>170</td>
<td>6.6M</td>
<td>19M</td>
</tr>
<tr>
<td>LiveJournal</td>
<td>4.8M</td>
<td>69M</td>
<td>14</td>
<td>20K</td>
<td>18</td>
<td>970K</td>
<td>3.8M</td>
</tr>
<tr>
<td>XyceTest</td>
<td>1.9M</td>
<td>8.3M</td>
<td>4.2</td>
<td>246</td>
<td>93</td>
<td>400K</td>
<td>1.5M</td>
</tr>
<tr>
<td>RDF_Data</td>
<td>1.9M</td>
<td>130M</td>
<td>70</td>
<td>10K</td>
<td>7</td>
<td>1.9M</td>
<td>1</td>
</tr>
<tr>
<td>RDF_linkedct</td>
<td>15M</td>
<td>34M</td>
<td>2.3</td>
<td>72K</td>
<td>13</td>
<td>15M</td>
<td>1</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>
<td>210K</td>
<td>360K</td>
</tr>
<tr>
<td>R-MAT.22</td>
<td>2.1M</td>
<td>34M</td>
<td>16</td>
<td>60K</td>
<td>9</td>
<td>790K</td>
<td>1.3M</td>
</tr>
<tr>
<td>R-MAT.24</td>
<td>7.7M</td>
<td>130M</td>
<td>17</td>
<td>150K</td>
<td>9</td>
<td>3.0M</td>
<td>4.7M</td>
</tr>
<tr>
<td>GNP.1</td>
<td>10M</td>
<td>200M</td>
<td>20</td>
<td>49</td>
<td>7</td>
<td>1</td>
<td>10M</td>
</tr>
<tr>
<td>GNP.10</td>
<td>10M</td>
<td>200M</td>
<td>20</td>
<td>49</td>
<td>7</td>
<td>10</td>
<td>5.0M</td>
</tr>
<tr>
<td>Friendster</td>
<td>66M</td>
<td>1800M</td>
<td>53</td>
<td>5.2K</td>
<td>34</td>
<td>70</td>
<td>66M</td>
</tr>
<tr>
<td>Orkut</td>
<td>3.1M</td>
<td>117M</td>
<td>76</td>
<td>33K</td>
<td>11</td>
<td>1</td>
<td>3.1M</td>
</tr>
<tr>
<td>Cube</td>
<td>2.1M</td>
<td>62M</td>
<td>56</td>
<td>69</td>
<td>157</td>
<td>47K</td>
<td>2.1M</td>
</tr>
<tr>
<td>Kron.21</td>
<td>1.5M</td>
<td>91M</td>
<td>118</td>
<td>213K</td>
<td>8</td>
<td>94</td>
<td>1.5M</td>
</tr>
</tbody>
</table>
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.
The graph structure determines the runtime of different stages

- Large number of non-trivial 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 al.

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>MS</th>
<th>Hong</th>
<th>FW-BW</th>
<th>Color</th>
<th>MS Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Serial</td>
</tr>
<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>
</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>
</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>
</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>
</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>
</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>
</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>
</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>
</tr>
</tbody>
</table>
Performance Results - Connected Components

Strong Scaling

- Multistep for CC compared to MS-Coloring and Ligra CC color-based approach
- Scaling shown against baseline serial BFS approach
Performance Results - Weakly Connected Components

Strong Scaling

- Multistep for WCC compared to MS-Coloring
Conclusions and Future work

- New Multistep shared-memory algorithm for computing CCs, SCCs, and WCCs in large graphs
- Faster than three different algorithms on a variety of graphs
- Current state of task parallellism in OpenMP/TBB is not fine-grained enough for these algorithms
- Future work: investigate performance on many-core architectures (Xeon Phi) and scaling for larger graphs
Conclusions and Future work

- New Multistep shared-memory algorithm for computing CCs, SCCs, and WCCs in large graphs
- Faster than three different algorithms on a variety of graphs
- Current state of task parallellism in OpenMP/TBB is not fine-grained enough for these algorithms
- Future work: investigate performance on many-core architectures (Xeon Phi) and scaling for larger graphs

Questions?


