Computing Strongly Connected Components in Modern Architectures
Shared-Memory Implementation and Testing

George M. Slota$^{1,2}$  Sivasankaran Rajamanickam$^1$
Erik G. Boman$^1$

$^1$Sandia National Laboratories
$^2$Pennsylvania State University

8 July 2013
Overview

- Previous parallel strongly connected component (SCC) algorithms
- Current implementation and improvements
- Performance results
- Conclusions and Future work
Previous Related Algorithms

- Forward-Backward (FW-BW) [1]
- Trimming [2]
- Coloring [3]
- Others [4, 5]
Previous Algorithms
Forward-Backward (FW-BW)

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

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

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

- Select pivot
- Find all **descendant** vertices that can be reached by pivot ($D$)
Previous Algorithms
Forward-Backward (FW-BW)

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

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

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

- Select pivot
- Find all \textbf{descendant} vertices that can be reached by pivot ($D$)
- Find all \textbf{predecessor} vertices that can reach pivot ($P$)
Previous Algorithms
Forward-Backward (FW-BW)

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

- Select pivot
- Find all **descendant** vertices that can be reached by pivot ($D$)
- Find all **predecessor** vertices that can reach pivot ($P$)
- 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

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)
- 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

- Consider the original vertex of each color to be the root of a new SCC
- Each SCC is all vertices reachable **backward** from each root with the same color

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

![Diagram showing vertex coloring process](image-url)
Consider vertex identifiers as *colors*

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

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
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 reachable backward from each root with the same color
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 reachable *backward* from each root with the same color
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 reachable \textbf{backward} from each root with the same color
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 reachable **backward** from each root with the same color.
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 reachable *backward* from each root with the same color
- 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
- Each SCC is all vertices reachable **backward** from each root with the same color
- Remove found SCCs, reset colors, and repeat until no vertices remain
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

- Most real-world graphs have one giant SCC and many many small SCCs
- 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
- Coloring is very inefficient at finding a large SCC, but is efficient at finding many small ones
- Tarjan’s [6] serial algorithm runs extremely quick for a small number of vertices, scales poorly for a larger number of vertices
- Obvious solution: combine these methods
Current Implementation
Multistep Method

- Do (no/simple/complete) 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 \((S = P \cap D)\)
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 \((D)\)
Current Implementation
Multistep Method: FWBW-SCC

- 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 \((D)\)
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 \((D)\)
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.
We can also combine the forward and backward search
We can also combine the forward and backward search. As we search forward, if we find any vertex already marked as in the SCC, we know the current vertex must be as well.
We can also combine the forward and backward search. As we search forward, if we find any vertex already marked as in the SCC, we know the current vertex must be as well.
Current Implementation
Multistep Method: Hybrid Search

- We can also combine the forward and backward search
- As we search forward, if we find any vertex already marked as in the SCC, we know the current vertex must be as well
Current Implementation
Multistep Method: Hybrid Search

- We can also combine the forward and backward search.
- As we search forward, if we find any vertex already marked as in the SCC, we know the current vertex must be as well.
- We mark the current vertex and all of its already visited predecessors as in the SCC.
- We can also combine the forward and backward search.
- As we search forward, if we find any vertex already marked as in the SCC, we know the current vertex must be as well.
- We mark the current vertex and all of its already visited predecessors as in the SCC.
Current Implementation
Multistep Method: Hybrid Search

- We can also combine the forward and backward search
- As we search forward, if we find any vertex already marked as in the SCC, we know the current vertex must be as well
- We mark the current vertex and all of its already visited predecessors as in the SCC
Current Implementation
Multistep Method: Hybrid Search

- We can also combine the forward and backward search
- As we search forward, if we find any vertex already marked as in the SCC, we know the current vertex must be as well
- We mark the current vertex and all of its already visited predecessors as in the SCC
We can also combine the forward and backward search.

- As we search forward, if we find any vertex already marked as in the SCC, we know the current vertex must be as well.
- We mark the current vertex and all of its already visited predecessors as in the SCC.
Performance Results
Test Algorithms

- **Multistep**: Simple trimming, parallel DFS FWBW-SCC, coloring until less than 100k vertices remain, serial Tarjan
- **Hybrid**: Simple trimming, parallel DFS hybrid search, coloring until less than 100k vertices remain, serial Tarjan
- **FW-BW**: Complete trimming, FW-BW algorithm until completion
- **Coloring**: Complete trimming, Coloring until 100K vertices, serial Tarjan
- **Serial**: Serial Tarjan
Performance Results
Test Environment and Networks

- **Vesper (AMD):** 64 Magnycour cores, 8x8 configuration.
- **Compton (Intel):** Xeon E5-2670 (Sandybridge), dual socket, 16 cores.

<table>
<thead>
<tr>
<th>Network</th>
<th>n</th>
<th>m</th>
<th>$d_{avg}$</th>
<th>$d_{max}$</th>
<th>Dia.</th>
<th># SCCs</th>
<th>max SCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Friendster</td>
<td>66M</td>
<td>1.8B</td>
<td>53</td>
<td>5.2K</td>
<td>25</td>
<td>3M</td>
<td>63M</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>Wiki-Talk</td>
<td>2.4M</td>
<td>5.0M</td>
<td>2.1</td>
<td>100K</td>
<td>9</td>
<td>2.2M</td>
<td>500K</td>
</tr>
<tr>
<td>USA Road Net</td>
<td>24M</td>
<td>29M</td>
<td>1.2</td>
<td>9</td>
<td>~8K</td>
<td>1</td>
<td>24M</td>
</tr>
<tr>
<td>Patents</td>
<td>3.8M</td>
<td>17M</td>
<td>4.4</td>
<td>770</td>
<td>22</td>
<td>3.8M</td>
<td>1</td>
</tr>
<tr>
<td>Google Web</td>
<td>876K</td>
<td>5.1M</td>
<td>5.8</td>
<td>460</td>
<td>22</td>
<td>410K</td>
<td>430K</td>
</tr>
<tr>
<td>XyceTest</td>
<td>1.9M</td>
<td>8.3M</td>
<td>4.2</td>
<td>246</td>
<td>~91</td>
<td>400K</td>
<td>1.5M</td>
</tr>
<tr>
<td>R-MAT 20</td>
<td>560K</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>
</tbody>
</table>

Graphs used in experiments retrieved from [7, 8, 9].
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 (no trimming not run with FW-BW due to excessive processing times).
Performance Results
LiveJournal on AMD

- Multistep and hybrid show good (5×) parallel scaling
- FW-BW algorithm suffers from previously mentioned load imbalance
- Relatively high diameter (18) results in poor coloring performance
Performance Results
Friendster on AMD

- About (4×) parallel scaling for multistep
- FW-BW and multistep/hybrid algorithms similar due to small number of SCCs remaining after initial trim (∼70)
- Very poor coloring performance from high diameter and large size
Performance Results
USA Road Net on Intel

- Coloring not shown in results due to very poor performance (over 1000s)
- Similar performance between multistep, hybrid, and FW-BW due to graph being fully connected; all three algorithms explore the entire graph
- Although not much speedup, all three are still faster than Tarjan’s
Performance Results

XyceTest on Intel

- Good speedup for all tested algorithms (×7 max)
- Hybrid algorithm fastest over multistep by ~10%
- Both hybrid and multistep are faster than Tarjan’s with a single thread
70% of the time multistep is the best approach
- Coloring by itself is the worst algorithm
- FW-BW is within $2 \times$ of best algorithm in 70% of the problems
R-MAT graphs (20,22,24): n, m, number of SCCs, and size of max SCC all increase by $\sim 4\times$ with each graph

- Approximate weak scaling observed with FW-BW and multistep/hybrid
- Scaling much worse for Tarjan’s and much much worse for coloring


Conclusions and Future work

- Current individual algorithms have drawbacks, using multistep/hybrid approach exploits strengths and minimizes weaknesses.
- Parallel multistep shows good speedup compared to serial Tarjan, sometimes is faster than Tarjan with a single thread.
- For most large real world graphs, a majority of the time is spent in initial SCC search, best way to decrease computation times is with a better hardware optimized BFS/DFS search.
- Further optimizations to the FWBW-SCC and FWBW-hybrid search algorithms may be possible.