Parallel Strongly Connected Components in Shared Memory Architectures

George M. Slota[2] Sivasankaran Rajamanickam[1]
Kamesh Madduri[2]

[1]Sandia National Laboratories
[2]Pennsylvania State University

21 February 2014

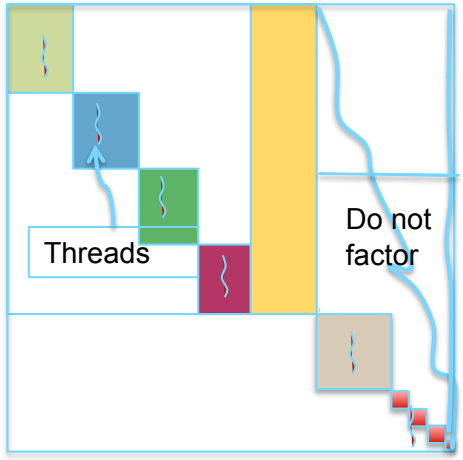
Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.
This work was supported by the DOE Office of Science through the FASTMath SciDAC Institute.

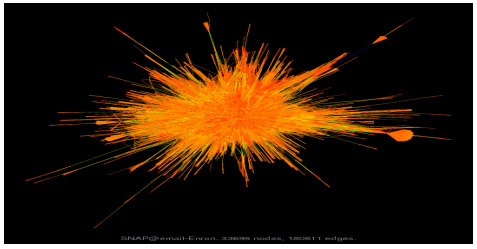
Overview

- 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 verificaton 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

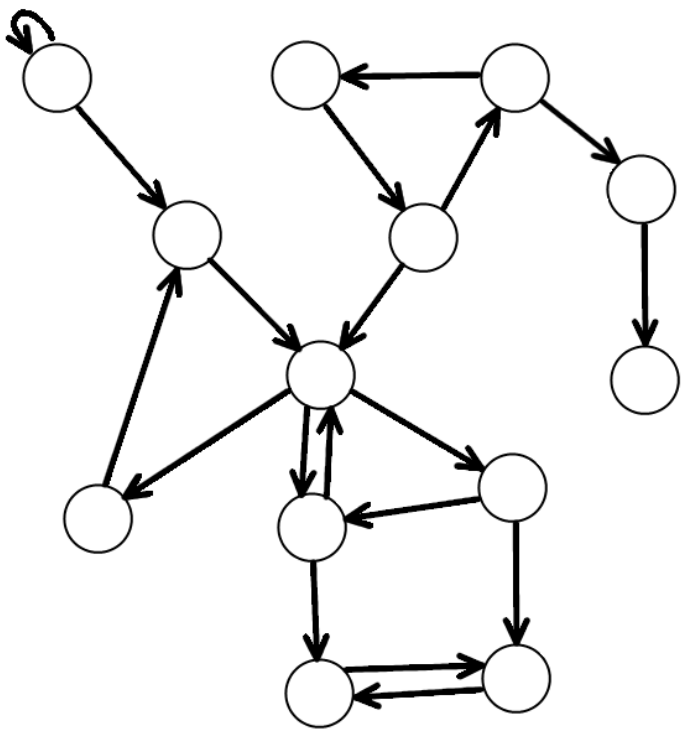
- Forward-Backward (FW-BW) (Hendrickson, Pinar, Plimpton, Fleischer, Mclendon)
- Coloring (Orzan)
- Task parallel, but own runtime with algorithmic improvements (Hong et al, SC 2013)
- Others (Barnat et al)

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.

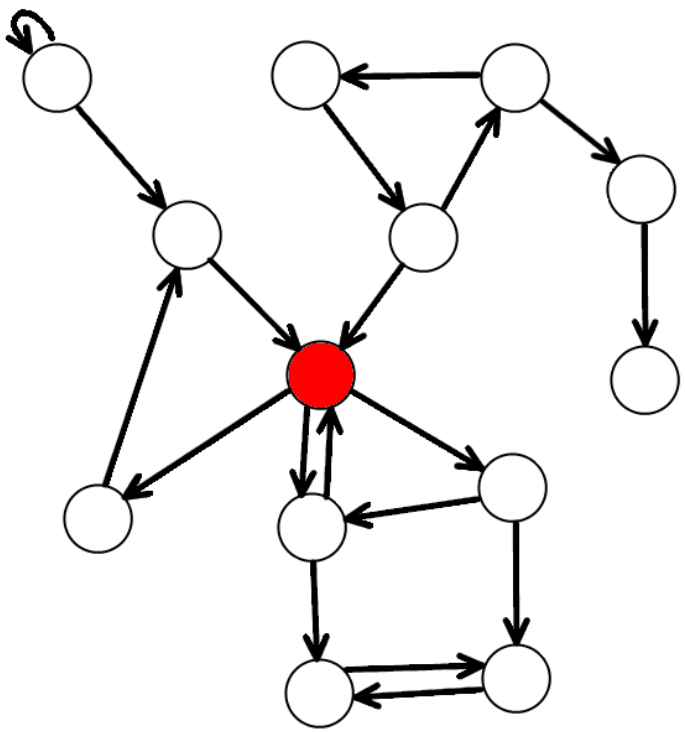
"BFS and Coloring-based Parallel Algorithms for Strongly Connected Components and Related Problems", G. Slota, S. Rajamanickam, K. Madduri (To appear: IPDPS 2014)

Previous Algorithms
Forward-Backward (FW-BW)



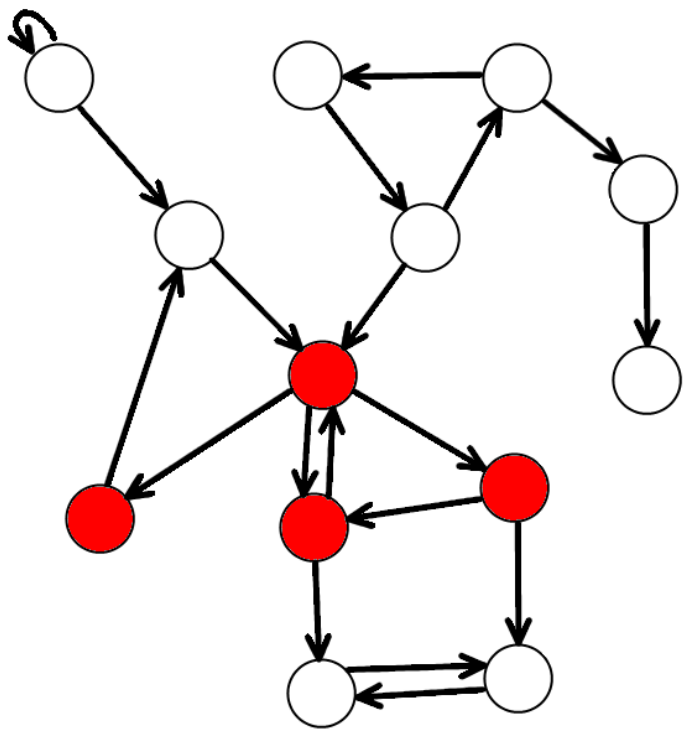
Previous Algorithms
Forward-Backward (FW-BW)

- Select pivot



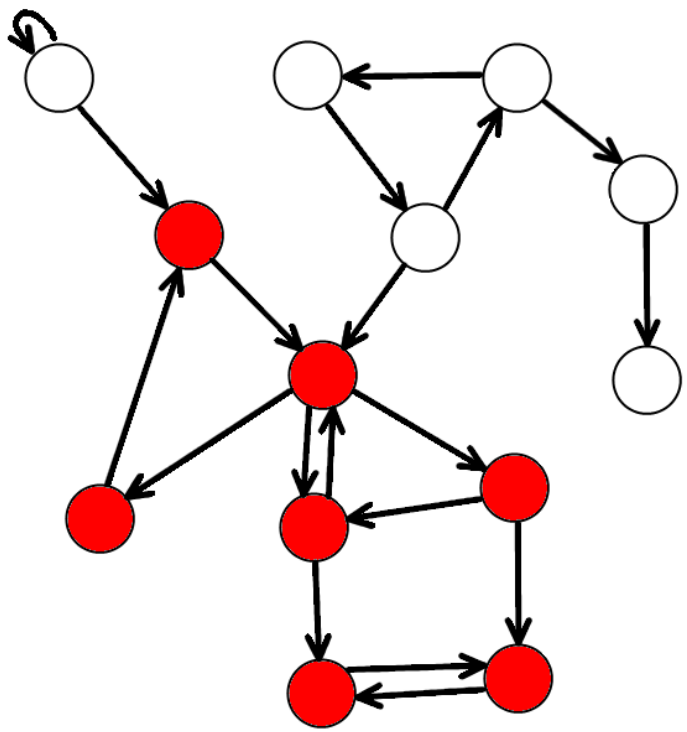
Previous Algorithms
Forward-Backward (FW-BW)

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



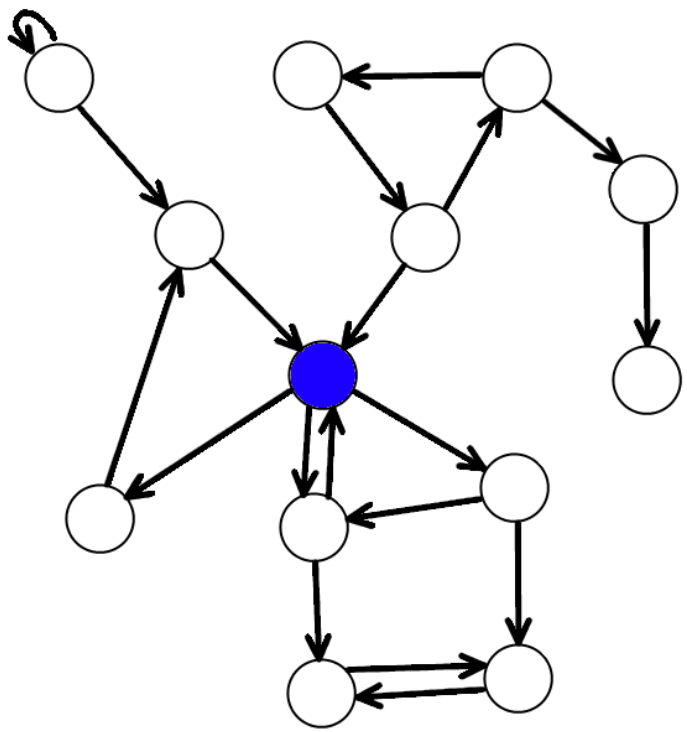
Previous Algorithms
Forward-Backward (FW-BW)

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



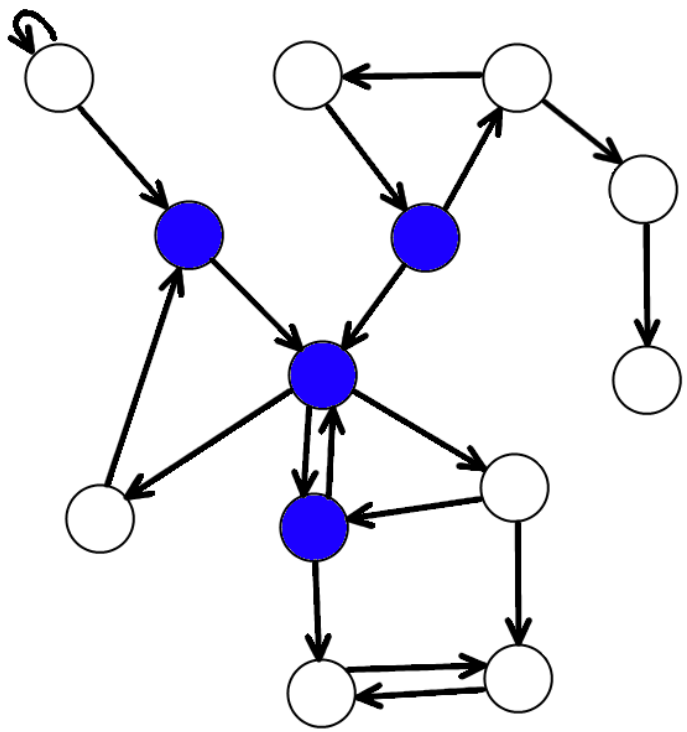
Previous Algorithms
Forward-Backward (FW-BW)

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



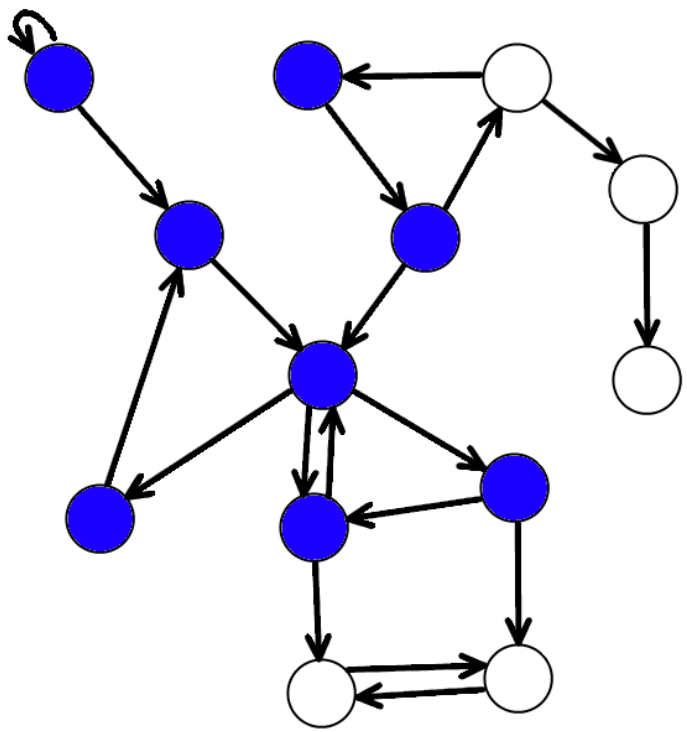
Previous Algorithms
Forward-Backward (FW-BW)

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



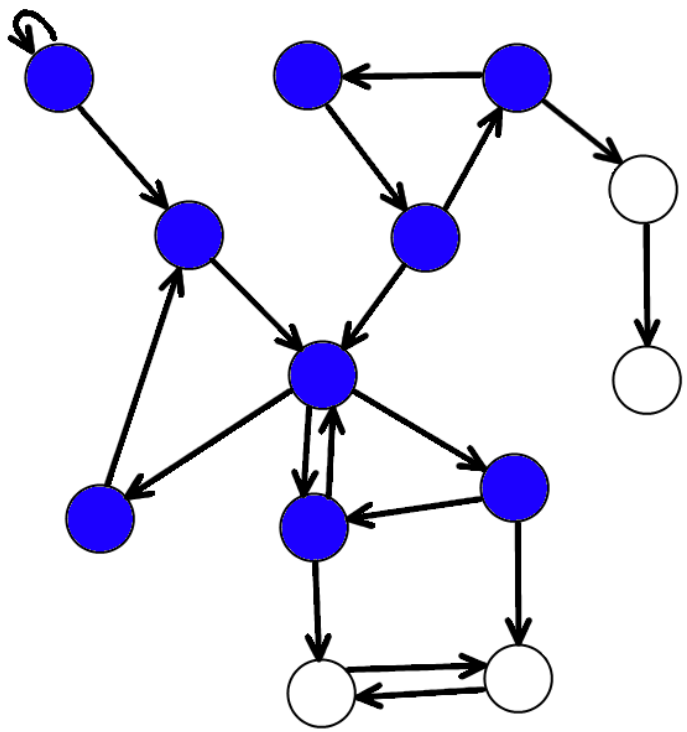
Previous Algorithms
Forward-Backward (FW-BW)

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



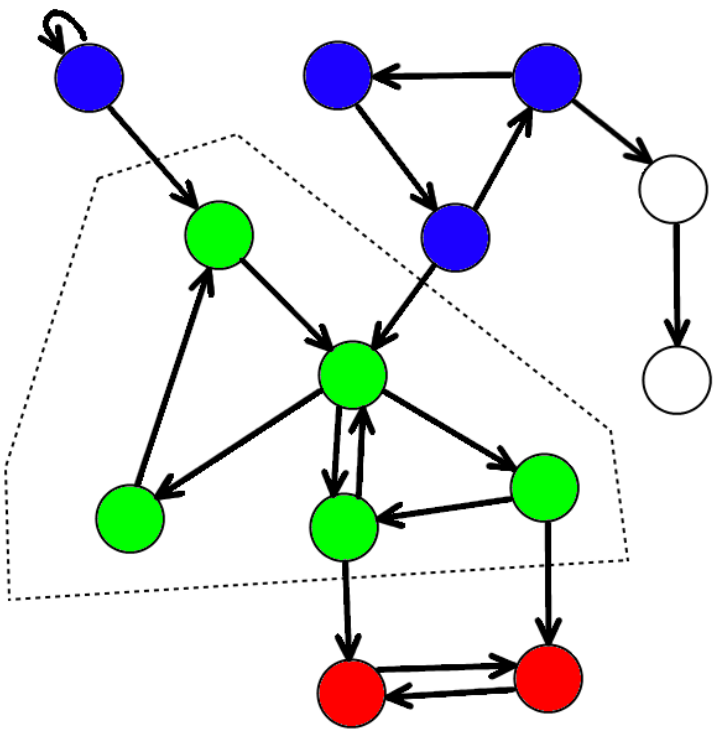
Previous Algorithms
Forward-Backward (FW-BW)

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



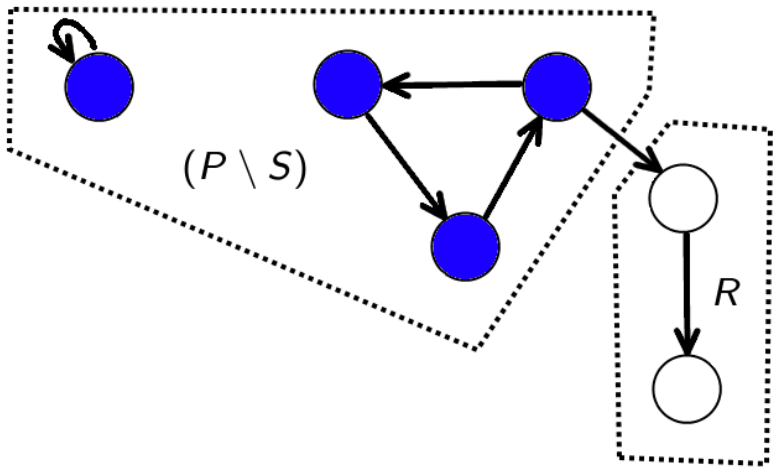
Previous Algorithms
Forward-Backward (FW-BW)

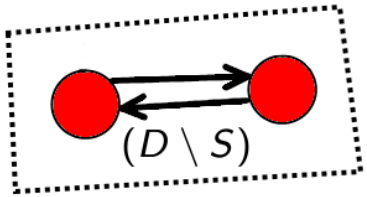
- Select pivot
- Find all vertices that can be reached from the pivot (**descendant** (D))
- Find all vertices that can reach the pivot (**predecessor** (P))
- Intersection of those two sets is an SCC ($S = P \cap D$)



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)



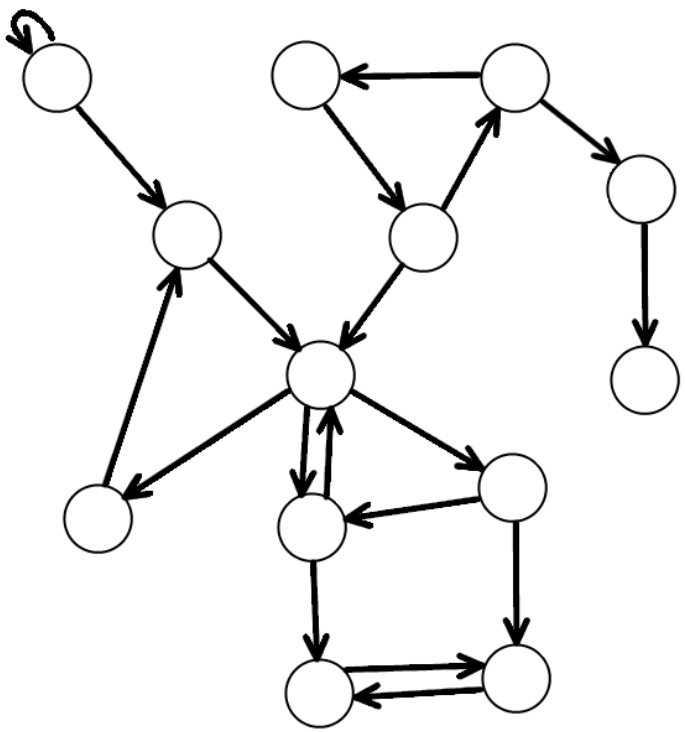


Forward-Backward (FW-BW) Algorithm

```
1:  procedure FW-BW(V)
2:      if V = ∅ then
3:          return ∅
4:      Select a pivot u ∈ V
5:      D ← BFS(G(V, E(V)), u)
6:      P ← BFS(G(V, E'(V)), u)
7:      R ← (V \ (P ∪ D)
8:      S ← (P ∩ D)
9:      new task do FW-BW(D \ S)
10:     new task do FW-BW(P \ S)
11:     new task do FW-BW(R)
```

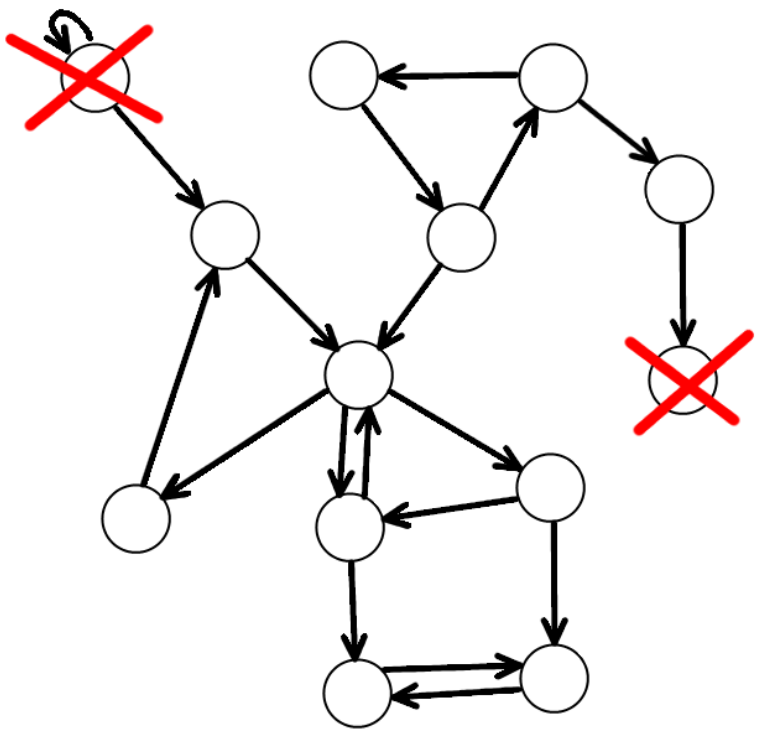
Previous Algorithms
Trimming

- Used to find trivial SCCs



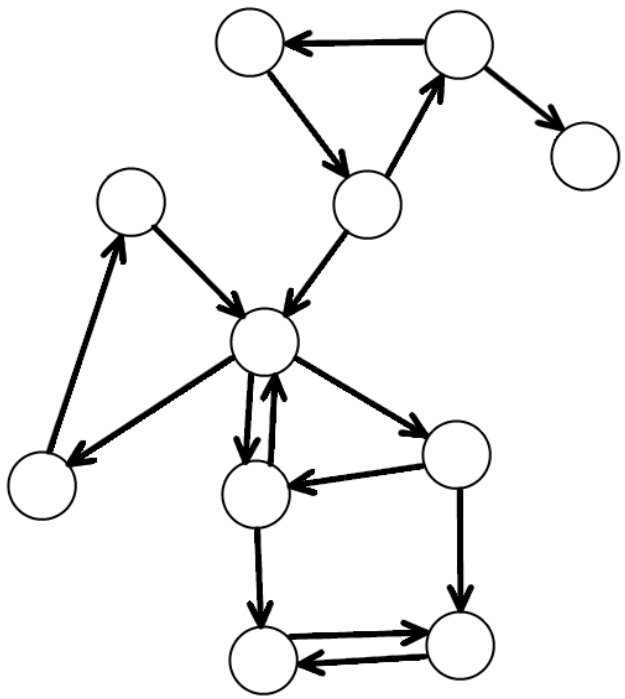
Previous Algorithms
Trimming

- 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)



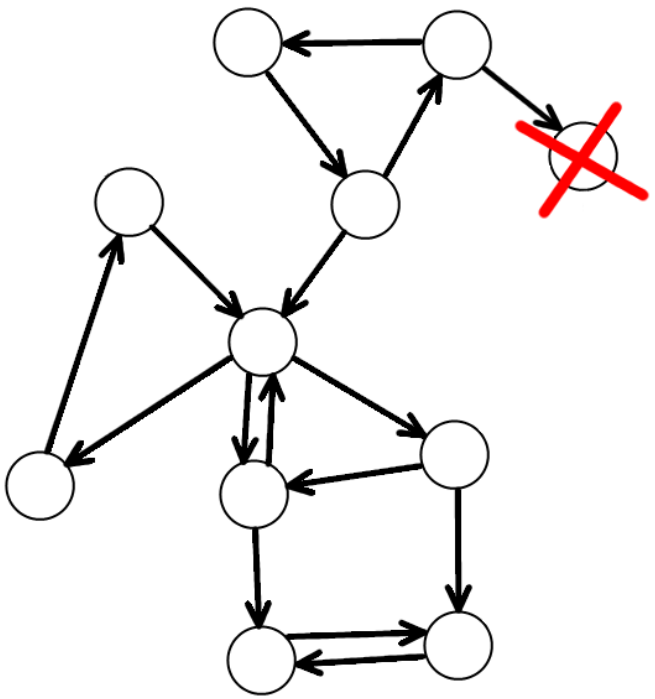
Previous Algorithms
Trimming

- 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)



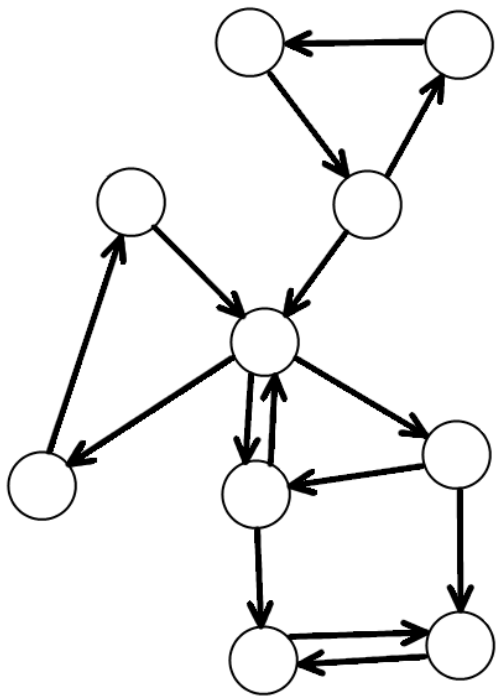
Previous Algorithms
Trimming

- 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)



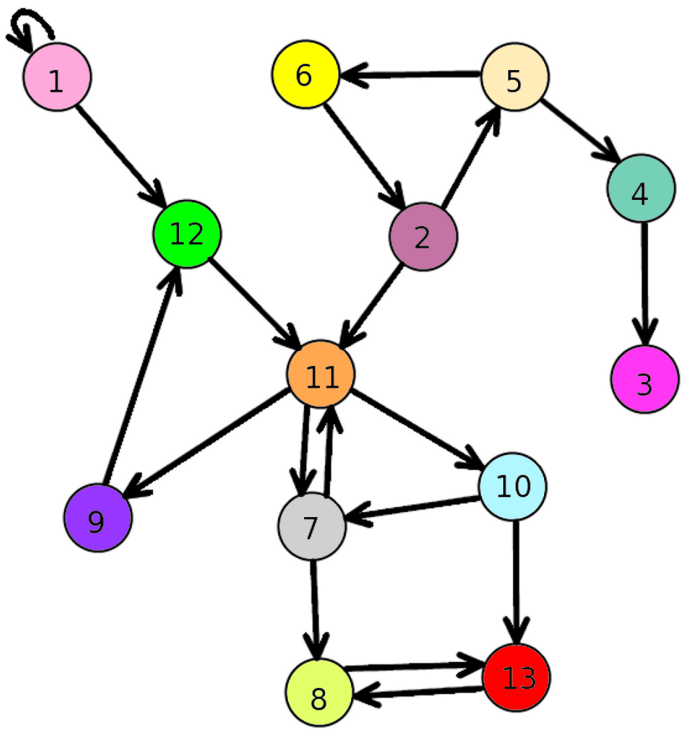
Previous Algorithms
Trimming

- 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)



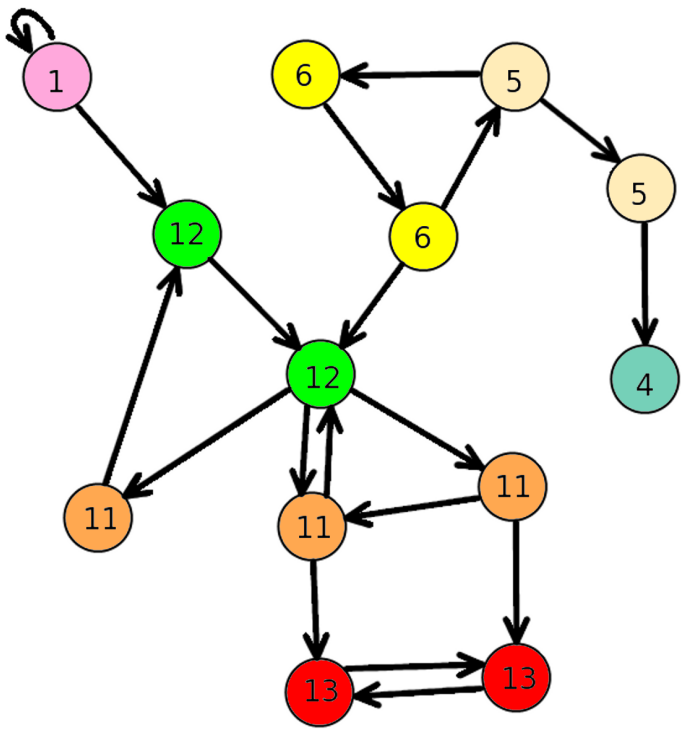
Previous Algorithms
Coloring

- Consider vertex identifiers as *colors*



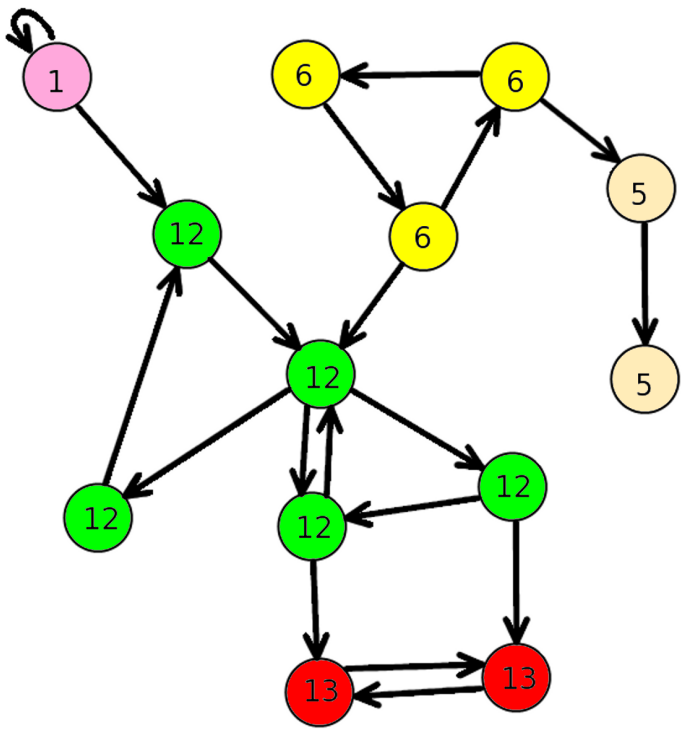
Previous Algorithms
Coloring

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



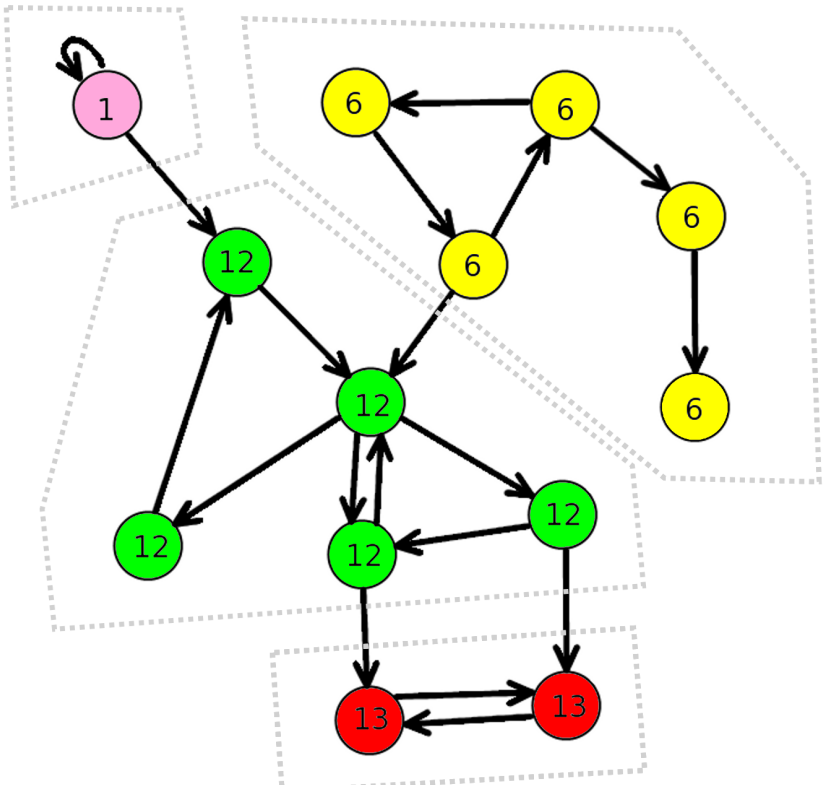
Previous Algorithms
Coloring

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



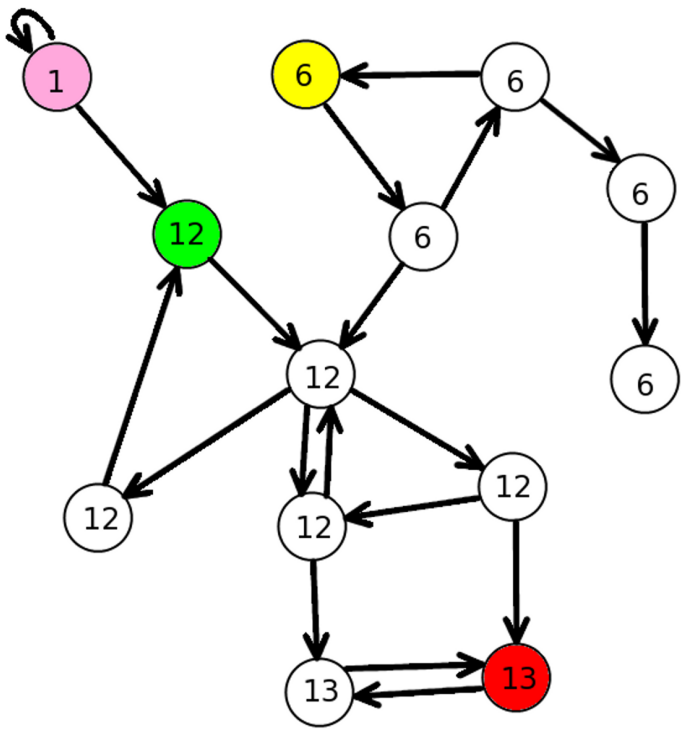
Previous Algorithms
Coloring

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



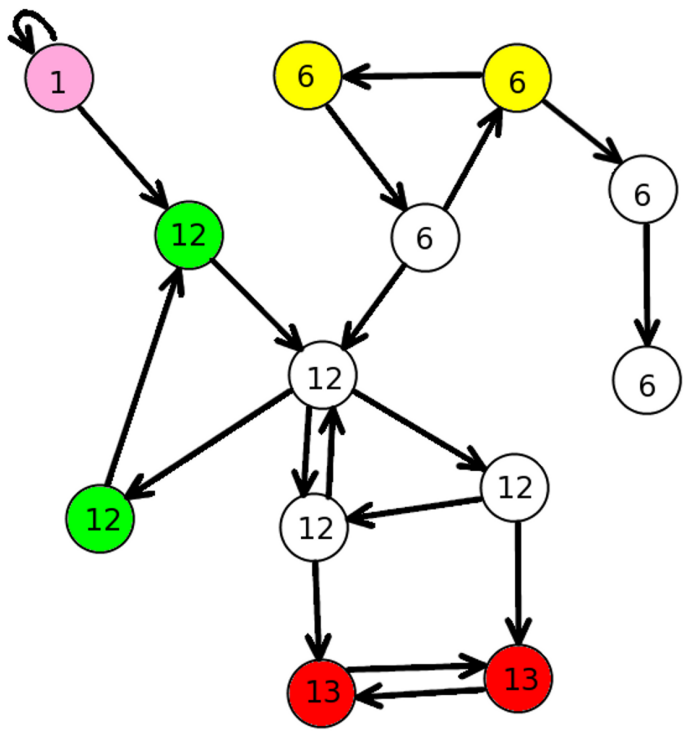
Previous Algorithms
Coloring

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



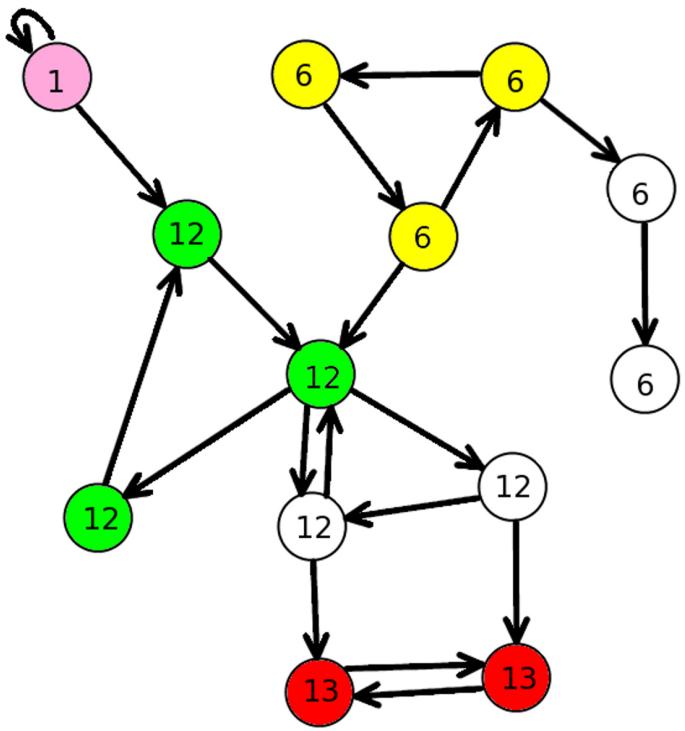
Previous Algorithms
Coloring

- 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.



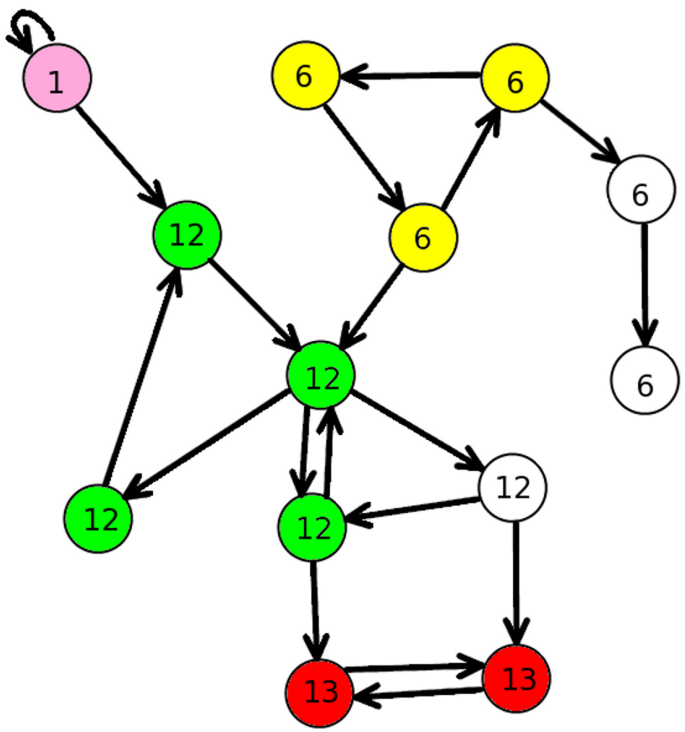
Previous Algorithms
Coloring

- 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.



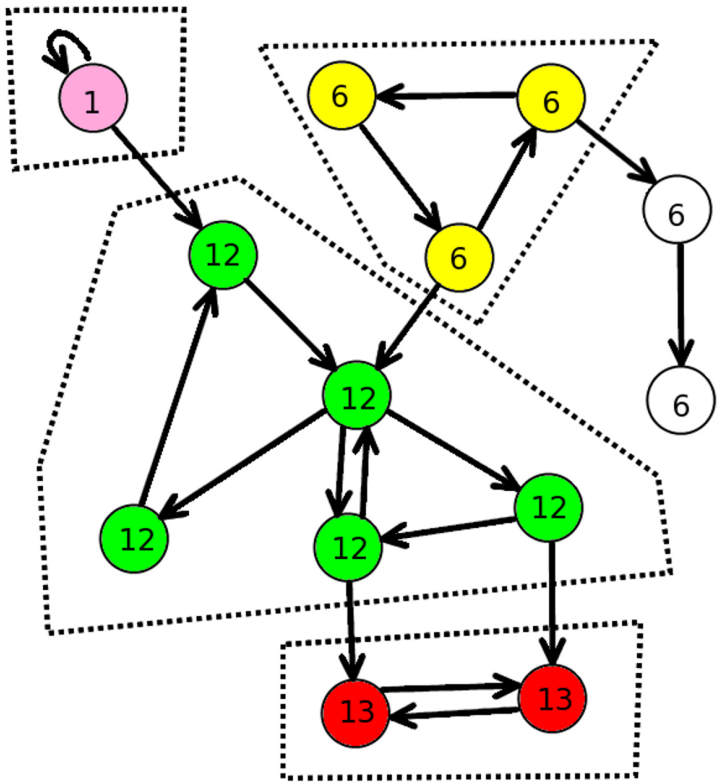
Previous Algorithms
Coloring

- 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.

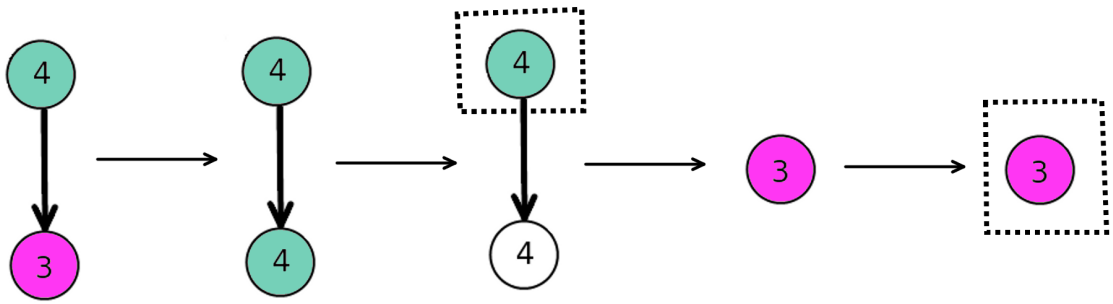


Previous Algorithms
Coloring

- 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 *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



Coloring Algorithms

```
1:  procedure ColorSCC(G(V, E))
2:      while G ≠ ∅ do
3:          for all u ∈ V do Colors(u) ← u
4:          while at least one vertex has changed colors do
5:              for all u ∈ V in parallel do
6:                  for all ⟨u, v⟩ ∈ E do
7:                      if Colors(u) > Colors(v) then
8:                          Colors(v) ← Colors(u)
9:          for all unique c ∈ Colors in parallel do
10:             V_c ← {u ∈ V : Colors(u) = c}
11:             SCV_c ← BFS(G(V_c, E'(V_c)), u)
12:             V ← (V \ SCV_c)
```

Previous Algorithms
Other Previous Work

- 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

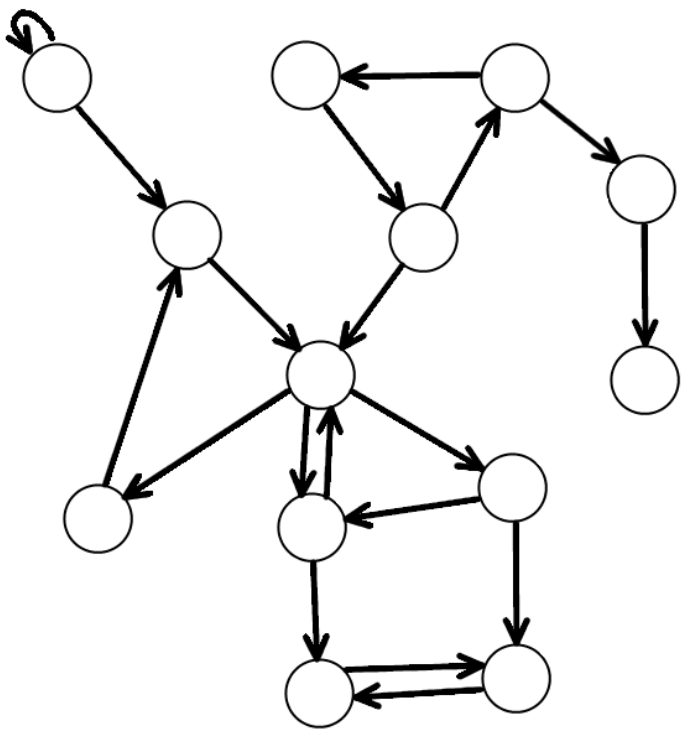
Multistep Method

1: **procedure** MULTISTEP($G(V, E)$)
2: $T \leftarrow$ MS-SimpleTrim(G)
3: $V \leftarrow V \setminus T$
4: Select $v \in V$ for which $d_{in}(v) * d_{out}(v)$ 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** NumVerts(V) $> n_{\text{cutoff}}$ **do**
9: $C \leftarrow$ MS-Coloring($G(V, E(V))$)
10: $V \leftarrow V \setminus C$
11: 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

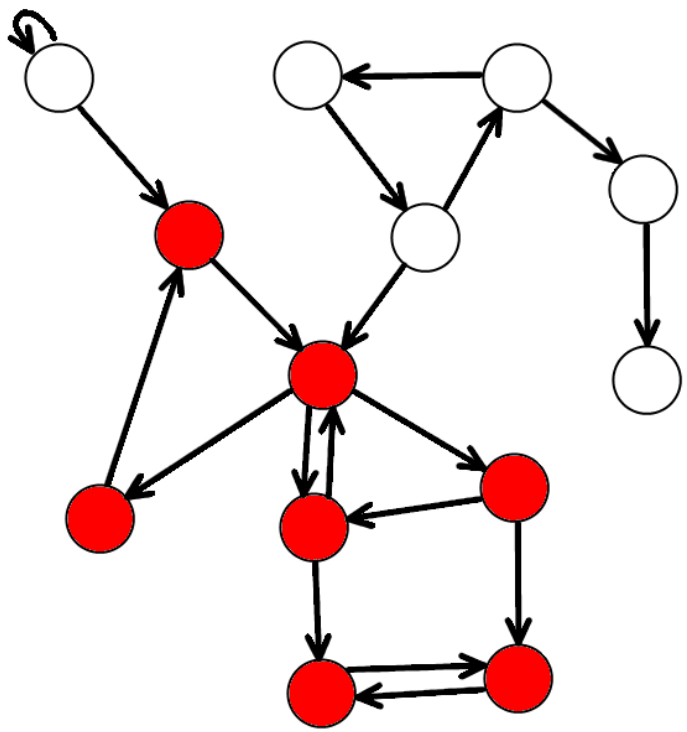
Multistep Method

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



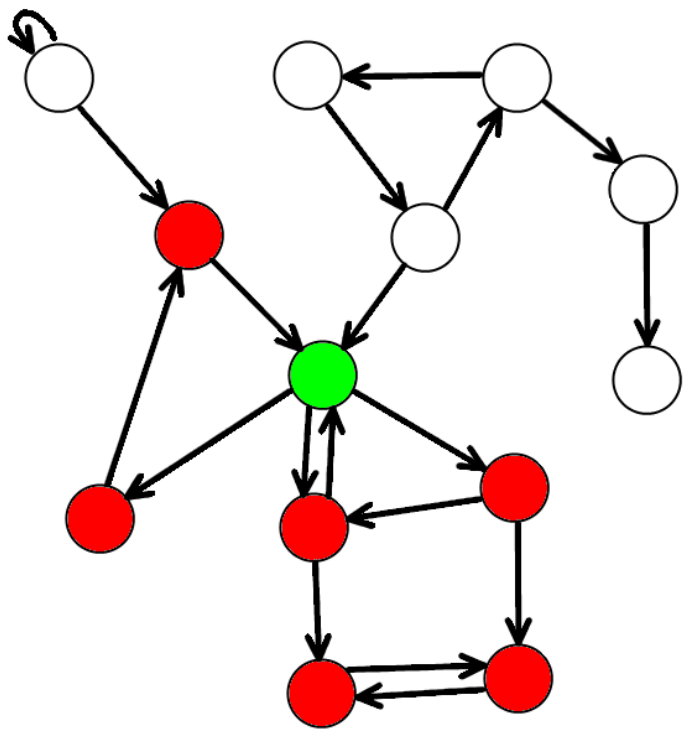
Multistep Method

- 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)



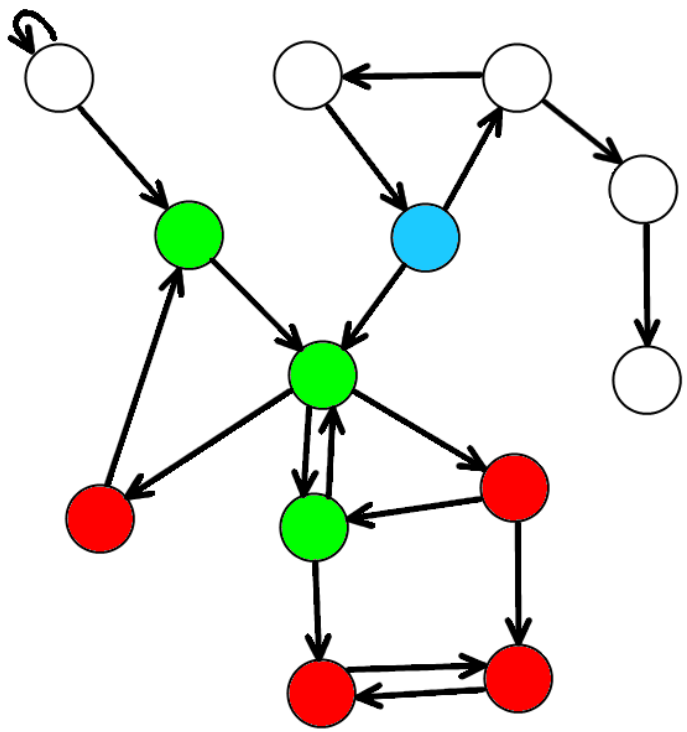
Multistep Method

- 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)



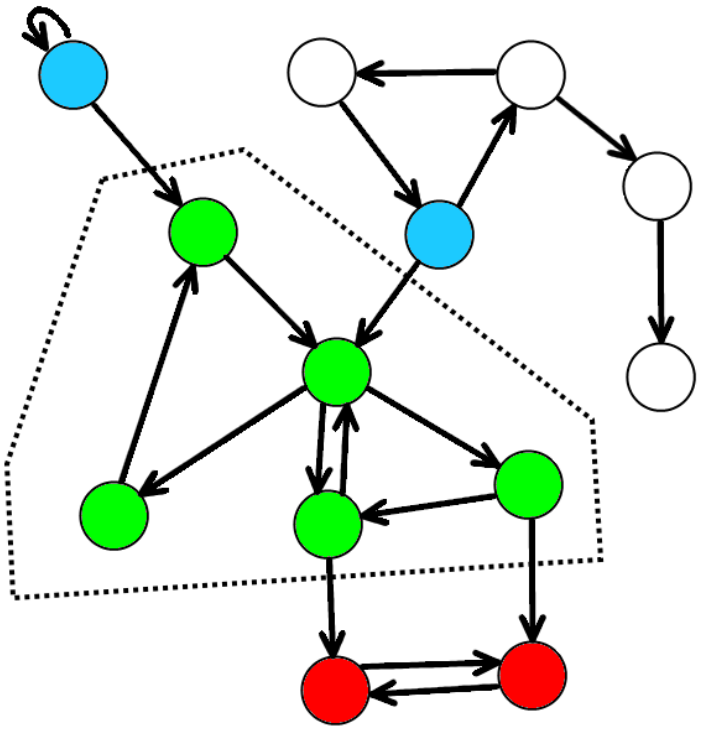
Multistep Method

- 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

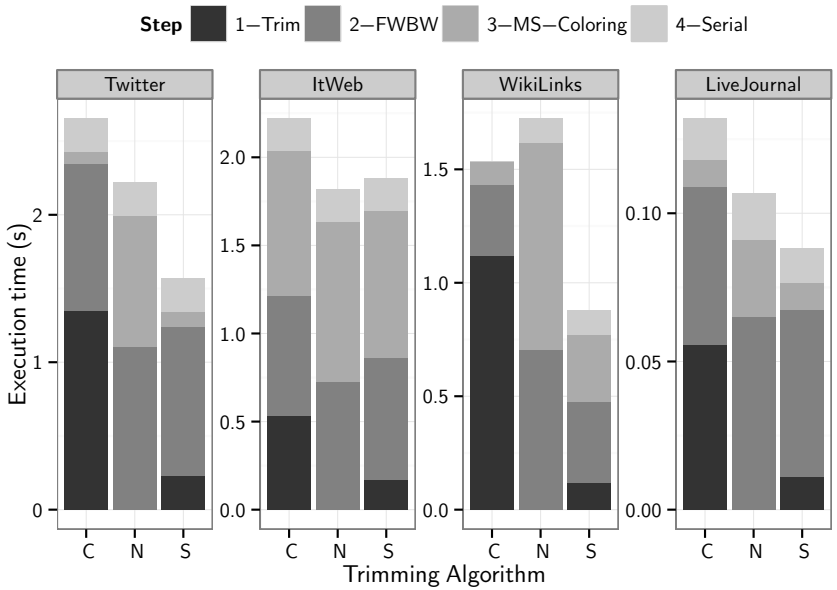
Test Environment and Graphs

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

Network	n	m	deg		$\widetilde{D}$	(S)CCs	
			avg	max		count	max
Twitter	53M	2000M	37	780K	19	12M	41M
ItWeb	41M	1200M	28	10K	830	30M	6.8M
WikiLinks	26M	600M	23	39K	170	6.6M	19M
LiveJournal	4.8M	69M	14	20K	18	970K	3.8M
XyceTest	1.9M	8.3M	4.2	246	93	400K	1.5M
RDF_Data	1.9M	130M	70	10K	7	1.9M	1
RDF_linkedct	15M	34M	2.3	72K	13	15M	1
R-MAT_20	0.56M	8.4M	15	24K	9	210K	360K
R-MAT_22	2.1M	34M	16	60K	9	790K	1.3M
R-MAT_24	7.7M	130M	17	150K	9	3.0M	4.7M
GNP_1	10M	200M	20	49	7	1	10M
GNP_10	10M	200M	20	49	7	10	5.0M

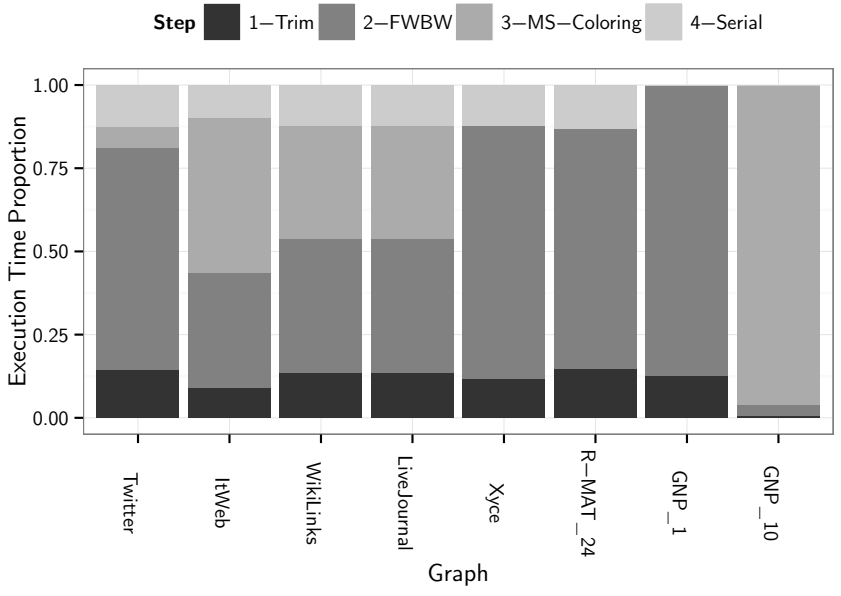
Performance Results
Trimming Options

- 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

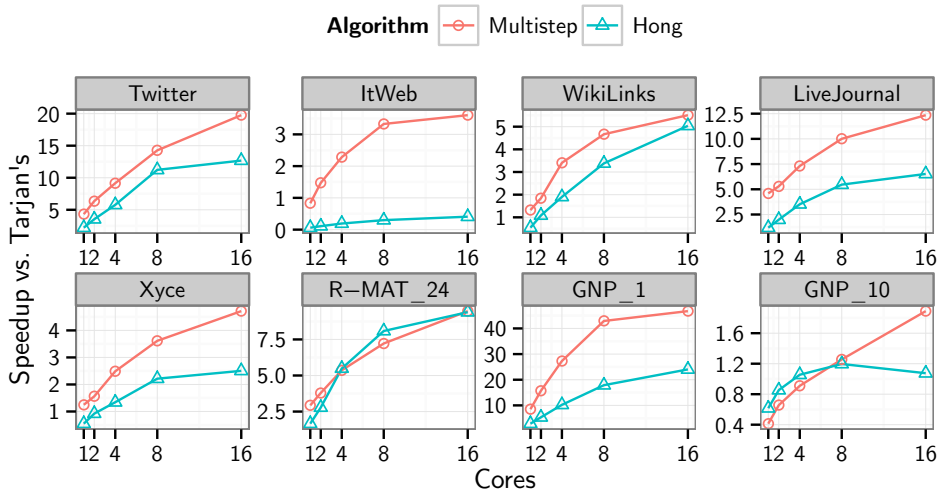
- The graph structure determines the runtime of different stages
- Large number of SCCs affects FW-BW (tasking overhead)
- Large diameter or a large SCC affects coloring



Performance Results - Strong Scaling

Strong Scaling

- 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

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

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 parallellism in OpenMP/TBB is not fine-grained enough for these algorithms.
- Testing this out in Intel MICs and compare performance.

Bibliography

[1] W. M. III, B. Hendrickson, and S. J. Plimpton, "Finding strongly connected components in distributed graphs," *Lecture Notes in Computer Science*, vol. 1800, pp. 505–512, 2000.

[2] L. K. Fleischer, B. Hendrickson, and A. Pinar, "On identifying strongly connected components in parallel," *Parallel and Distributed Processing*, vol. 65, pp. 901–910, 2005.

[3] S. Orzan, "On distributed verication and veried distribution," Ph.D. dissertation, Free University of Amsterdam, 2004.

[4] J. Barnat, P. Bauch, L. Brim, and M. Cevska, "Computing strongly connected components in parallel on cuda," in *Parallel and Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 544–555.

[5] S. Hong, N. C. Rodia, and K. Olukotun, "Technical report: On fast parallel detection of strongly connected components (scc) in small-world graphs," Stanford University, Tech. Rep., 2013.

[6] R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM Journal of Computing*, vol. 1, pp. 146–160, 1972.

[7] J. Leskovec, "SNAP: Stanford network analysis project," http://snap.stanford.edu/index.html, last accessed 3 July 2013.

[8] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *ICDM*, 2012.

[9] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *KDD*, 2006.

[10] J. Leskovec, D. Huttenlocher, and J. Kleinberg, "Signed networks in social media," in *CHI*, 2010.

[11] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[12] DIMACS, "9th dimacs implementation challenge - shortest paths."

[13] . Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2005.