Irregular Graph Algorithms on Modern Multicore, Manycore, and Distributed Processing Systems

Comprehensive Examination

George M. Slota

Scalable Computing Laboratory
Department of Computer Science and Engineering
The Pennsylvania State University

11 Feb. 2015

Presentation Overview

- Motivation
- Color-coding – FASCIA & FASTPATH
- Graph Connectivity – The Multistep Method
- Distributed Graph Layout – PULP and DGL
- Conclusions

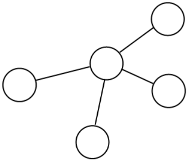
Motivation

- Graph analysis is key for the study of biological, chemical, social, and other networks
- Real-world graphs are big, irregular, complex
  - Graph analytics is one of DARPA's 23 toughest mathematical challenges
  - Facebook graph: 800M people, 100B friendships
  - Web graph: 3.5B sites, 129B links
  - Brain graph: 100B neurons, 1,000T synaptic connections
- Modern computational systems are also big and complex
  - Multiple levels of parallelism, memory hierarchy, configurations
  - Heterogenous – host, GPU, coprocessors (Xeon Phi MIC)
  - Optimization – account for socket-level, node-level, and distributed

- How do we design parallel graph algorithms for computational efficiency under all of the aforementioned difficulties?
- What algorithmic traits are common to various irregular graph algorithms that we can optimize for?
- How do we store/organize graphs efficiently in shared and distributed memory?

Color-coding and FASCIA, FASTPATH

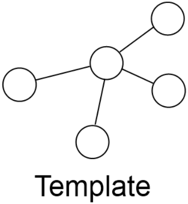
Part 1: **Color-coding** – subgraph counting and enumeration, minimum-weight path finding

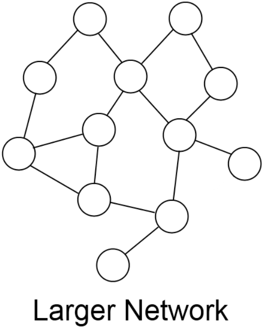


Template

Color-coding and FASCIA, FASTPATH

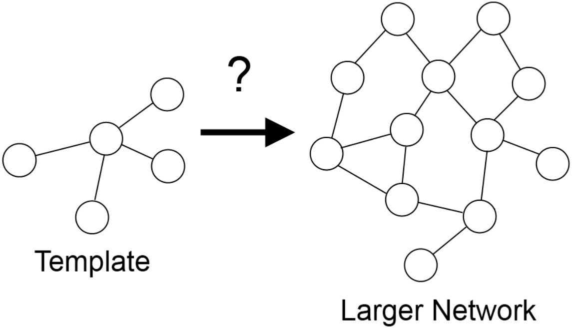
Subgraph Counting

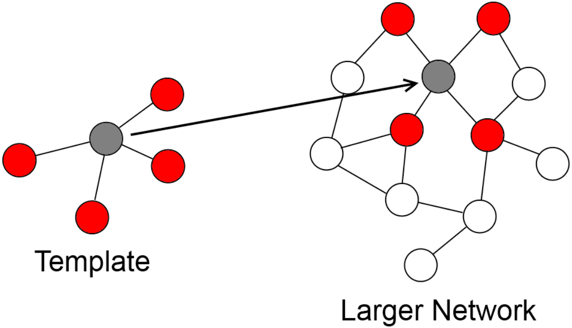




Color-coding and FASCIA, FASTPATH

Subgraph Counting

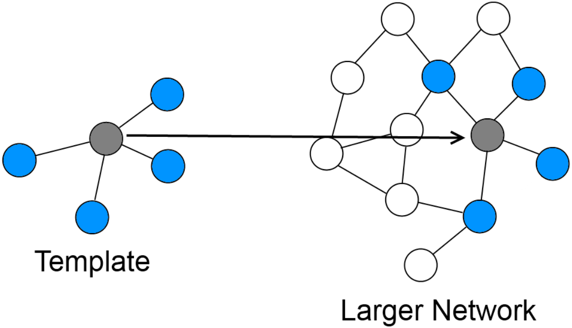




Template
Larger Network

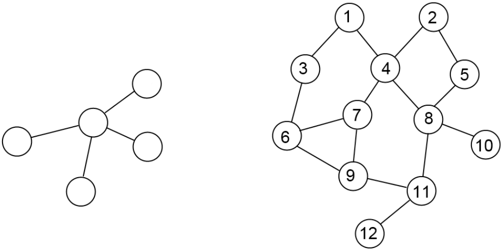
Color-coding and FASCIA, FASTPATH

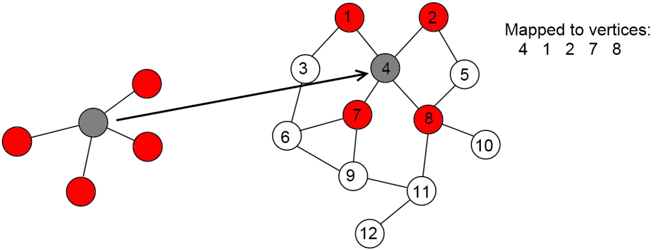
Subgraph Counting



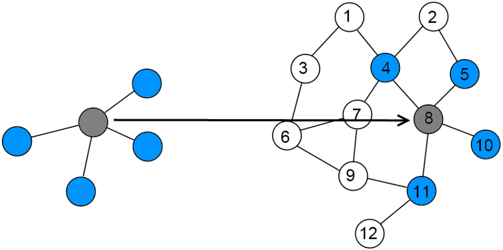
Color-coding and FASCIA, FASTPATH

Subgraph Enumeration




1
2
3
4
5
6
7
8
9
10
11
12
Mapped to vertices:
4 1 2 7 8

Color-coding and FASCIA, FASTPATH
Subgraph Enumeration



Mapped to vertices:

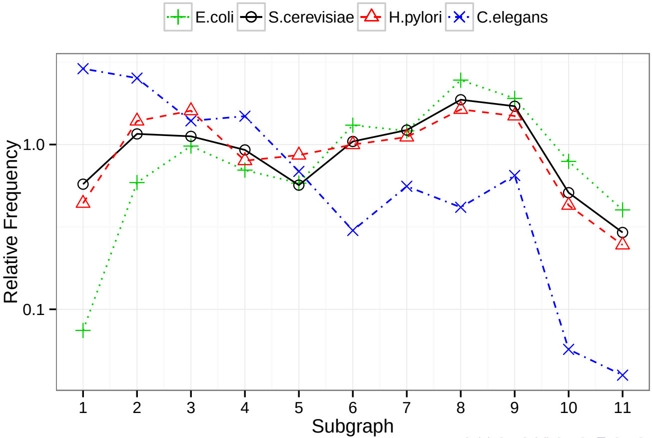
4	1	2	7	8
8	4	5	11	10

Color-coding and FASCIA, FASTPATH

Why do we want fast algorithms for subgraph counting, min-weight path finding?

- Important to bioinformatics, chemoinformatics, social network analysis, communication network analysis, etc.
- Forms basis of more complex analysis
  - Motif finding
  - Graphlet frequency distance (GFD)
  - Graphlet degree distributions (GDD)
  - Graphlet degree signatures (GDS)
- Counting and enumeration on large networks is very tough, $O(n^k)$ complexity for naïve algorithm
- Finding minimum-weight paths – NP-hard problem

- Motif finding: Look for all subgraphs of a certain size (and structure)



Color-coding and FASCIA, FASTPATH

Motif finding, GFD, and GDD, min-weight paths

- Motif finding: Look for all subgraphs of a certain size (and structure)
- GFD: Numerically compare occurrence frequency to other networks

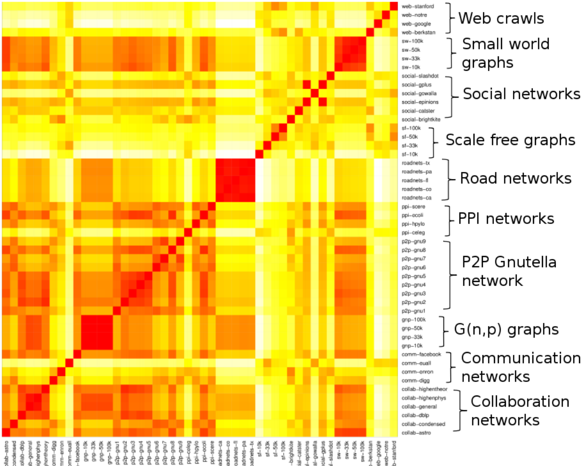
$$S_i(G) = -\log(\frac{C_i(G)}{\sum_{i=1}^{\mathrm{n}} C_i(G)})$$

$$D(G,H) = \sum_{i=1}^{\mathrm{n}} |S_i(G) - S_i(H)|$$

Color-coding and FASCIA, FASTPATH

Motif finding, GFD, and GDD, min-weight paths

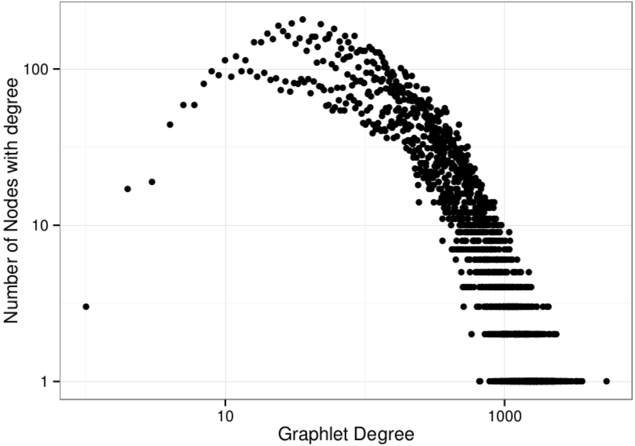
- Motif finding: Look for all subgraphs of a certain size (and structure)
- GFD: Numerically compare occurrence frequency to other networks



Color-coding and FASCIA, FASTPATH

Motif finding, GFD, and GDD, min-weight paths

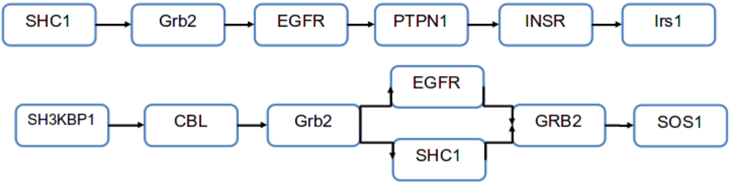
- Motif finding: Look for all subgraphs of a certain size (and structure)
- GFD: Numerically compare occurrence frequency to other networks
- GDD, GDS: Numerically compare embeddings/vertex distribution



Color-coding and FASCIA, FASTPATH

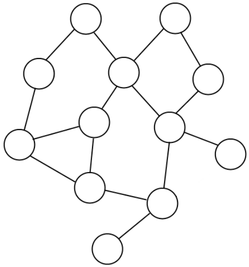
Motif finding, GFD, and GDD, min-weight paths

- Motif finding: Look for all subgraphs of a certain size (and structure)
- GFD: Numerically compare occurrence frequency to other networks
- GDD, GDS: Numerically compare embeddings/vertex distribution
- Min-weight paths: often biological significance in PPI nets



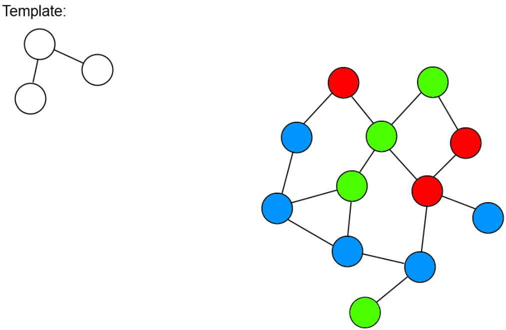
Color-coding and FASCIA, FASTPATH
Color-coding for Approximate Subgraph Counting

- Color-coding [Alon et al., 1995]: approximate count of *tree-like non-induced subgraph*



Color-coding and FASCIA, FASTPATH
Color-coding for Approximate Subgraph Counting

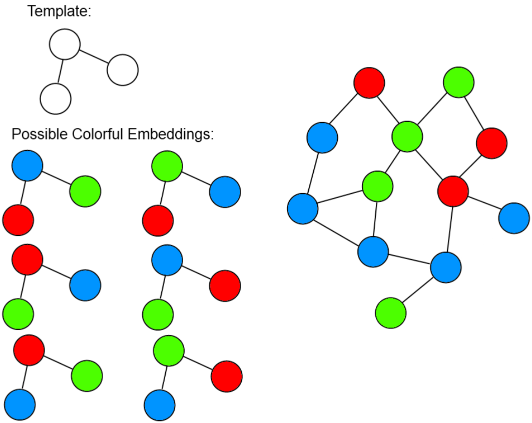
- Color-coding [Alon et al., 1995]: approximate count of *tree-like non-induced subgraph*



Color-coding and FASCIA, FASTPATH

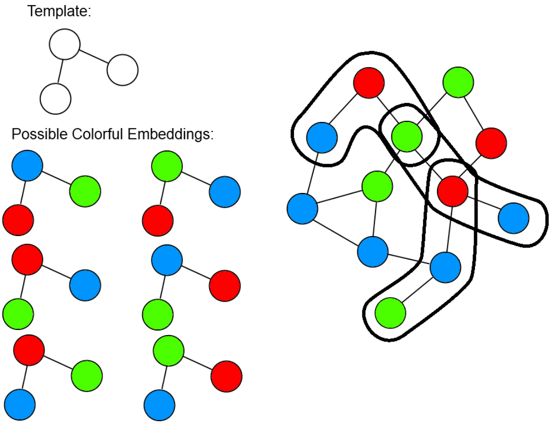
Color-coding for Approximate Subgraph Counting

- Color-coding [Alon et al., 1995]: approximate count of *tree-like non-induced subgraph*



Color-coding and FASCIA, FASTPATH
Color-coding for Approximate Subgraph Counting

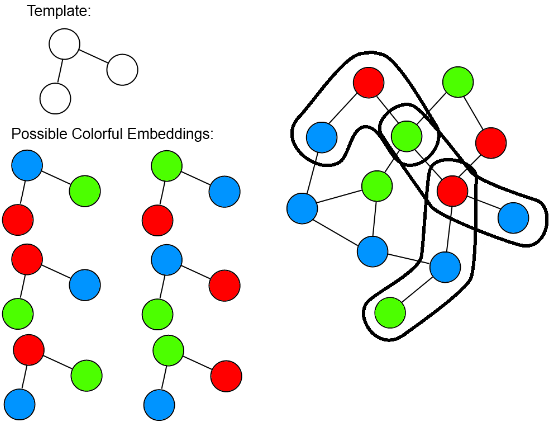
- Color-coding [Alon et al., 1995]: approximate count of *tree-like non-induced subgraph*



Color-coding and FASCIA, FASTPATH

Color-coding for Approximate Subgraph Counting

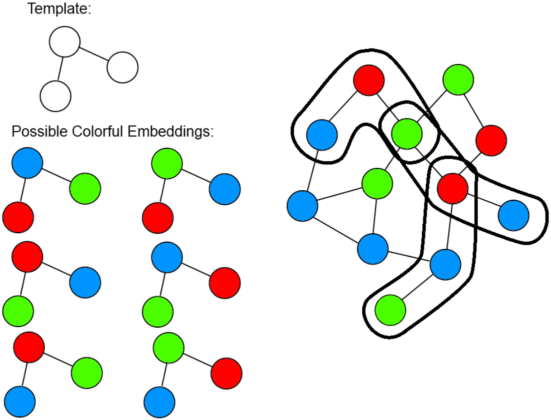
- Color-coding [Alon et al., 1995]: approximate count of *tree-like non-induced subgraph*
- $cnt_{colorful} = 3$, $C_{total} = 3^3$, $C_{colorful} = 3!$, $P = \frac{3!}{3^3}$
- $cnt_{estimate} = \frac{cnt_{colorful}}{P} = \mathbf{13.5}$



Color-coding and FASCIA, FASTPATH

Color-coding for Approximate Subgraph Counting

- Color-coding [Alon et al., 1995]: approximate count of *tree-like non-induced subgraph*
- $cnt_{colorful} = 3$, $C_{total} = 3^3$, $C_{colorful} = 3!$, $P = \frac{3!}{3^3}$
- $cnt_{estimate} = \frac{cnt_{colorful}}{P} = \mathbf{13.5}$
- Each iteration can run in $O(m \cdot 2^k \cdot e^k)$



Color-coding and FASCIA, FASTPATH

Related work for color-coding and subgraph counting

- Alon et al.'s Implementation [Alon et al., 2008]
  - Motif finding on PPI networks
- MODA [Omidi et al., 2009]
  - Uses approximation or exact scheme
  - Motif finding on small networks
- PARSE [Zhao et al., 2010a]
  - Distributed color-coding algorithm using MPI
  - Handles large graphs through partitioning
- SAHAD [Zhao et al., 2012b]
  - Distributed color-coding algorithm using Hadoop (MapReduce)
  - Handles vertex-labeled graphs, computes graphlet degree distributions

Color-coding and Fascia, FastPath
Fascia

Fascia: Fast Approximate Subgraph Counting and Enumeration

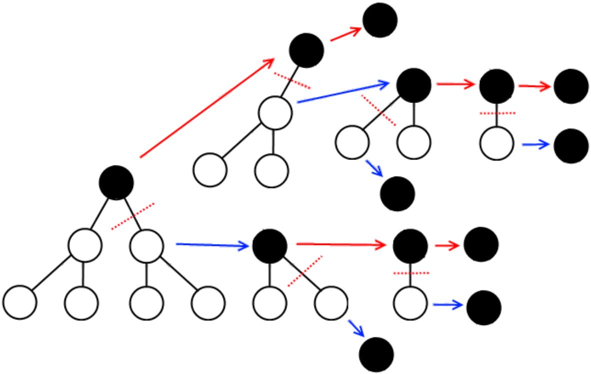
- Count and enumerate subgraphs, supports node labels
- Perform motif finding, calculate GDD
- Algorithmic Optimizations:
    - Combinatorial indexing scheme for color mappings
    - Shared and distributed memory parallelization strategies
    - Memory reduction via array design, hashing schemes
    - Speedup via template partitioning and work avoidance

Color-coding and FASCIA, FASTPATH

Algorithmic Overview Description

Partition input template T (k vertices) into subtemplates S_i using *single edge cuts*.
Select *Niter* to be performed
for $i = 1$ to *Niter* **do**
    Randomly assign to each v in G a color between 0 and $k - 1$.
    for all $v \in G$ **do**
        Use a **dynamic programming scheme** to count *colorful* non-induced occurrences of T rooted at v.
Take average of all *Niter* counts to be final count.

Color-coding and FASCIA, FASTPATH
Template partitioning



Color-coding and FASCIA, FASTPATH

FASCIA Dynamic Programming Step

for all sub-templates S_i created from partitioning T, in reverse order they were created during partitioning **do**
    for all vertices $v \in G$ **do**
        if S_i consists of a single node **then**
            Set **table**$[S_i][v][\mathit{color\ of\ v}] := 1$
        else
            S_i consists of active child a_i and passive child p_i
            for all colorsets C of unique values mapped to S **do**
                Set $\mathit{count} := 0$
                for all $u \in N(v)$, $N(v)$ is the neighborhood of v **do**
                    for all possible combinations C_a and C_p created by splitting C and mapping onto a_i and p_i **do**
                        $\mathit{count} \mathrel{+}= \mathrm{table}[a_i][v][C_a] \cdot \mathrm{table}[p_i][u][C_p]$
                Set $\mathrm{table}[S_i][v][C] := \mathit{count}$
$\mathit{templateCount} := \sum_v \sum_C \mathrm{table}[T][v][C]$

Color-coding and FASCIA, FASTPATH

FASTPATH Dynamic Programming Step

Initialize all weights$[1][v \in V][1 \cdots c_1] \leftarrow \infty$

for $i = 2$ to $L + 1$ **do**

  for all vertices $v \in G$ **do**

    S_i consists of active child a_i and passive child p_i

    $|a_i| = 1, \ |p_i| = i - 1$

    for all colorsets C of unique values mapped to S **do**

      Set $min_w := 0$

      for all $u \in N(v)$, $N(v)$ is the neighborhood of v **do**

        for all possible combinations C_a and C_p created by splitting C and mapping onto a_i and p_i **do**

          Set $w_a :=$ EdgeWeight(u, v)

          Set $w_p :=$ Weights$[i-1][u][C_p]$

          if $w_a + w_p \leq min_w$ **then**

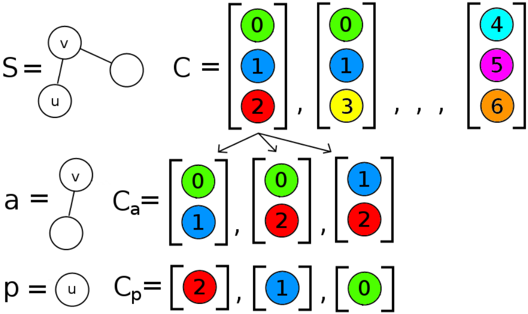
            Set $min_w := w_a + w_p$

      Set Weights$[S_i][v][C] := min_w$

Return Min(Weights$[S_{L+1}][\cdots][\cdots]$)

Color-coding and FASCIA, FASTPATH

Colorset and count calculation for FASCIA

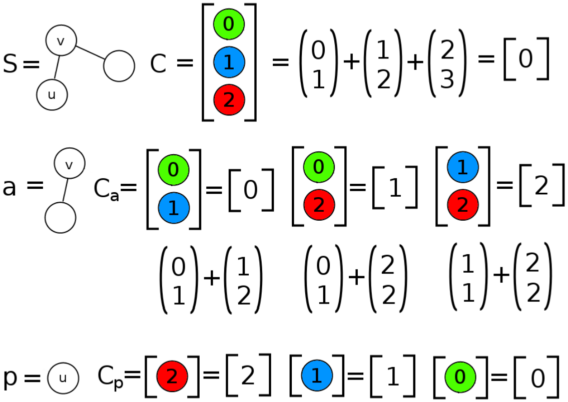


$$table[S][v][C] = \sum_{C_a C_p} \sum_{u} table[a][v][C] * table[p][u][C]$$

Color-coding and FASCIA, FASTPATH

Combinatorial indexing scheme

- Combinatorial number system to represent colorsets: $C = \binom{c_1}{1} + \binom{c_2}{2} + \cdots + \binom{c_k}{k}$, where $c_1 < c_2 < \cdots < c_k$
- Precompute indexes and ordering in advance, store in table (<2MB for k = 12)
- This avoid avoids explicit handling/passing of colors, or computation of colorset indexes during runtime



Color-coding and FASCIA, FASTPATH

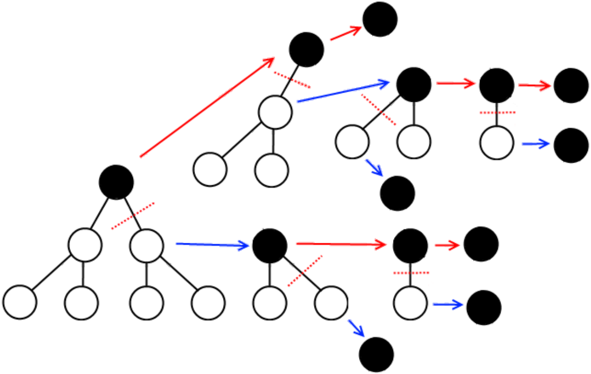
Memory optimizations

- We implement both a three-level array and hash table
  - Initialize storage in table on per-vertex basis
  - Hash table exploits random coloring to uniformly distribute and calculate keys
  - Generally: **array method more memory efficient for dense skewed graphs, hash table more efficient for sparse graphs**
- CSR representation in distributed setting
  - For each subtemplate we have a rectangular table $(v \times C)$
  - Convert to CSR (compressed sparse row format)
  - **Observe up to 75% reduction in distributed communication, even for dense graphs**

Color-coding and FASCIA, FASTPATH

Template partitioning and work avoidance

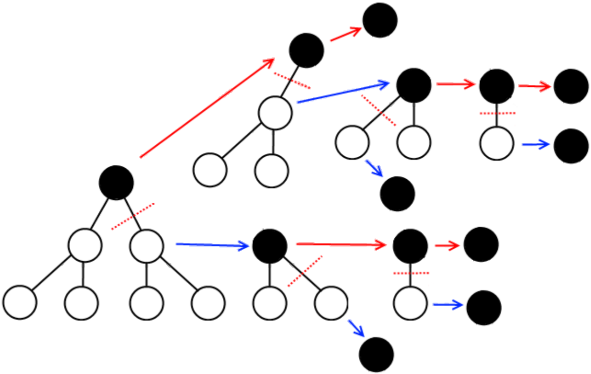
- Basic partitioning: try to evenly partition template



Color-coding and FASCIA, FASTPATH

Template partitioning and work avoidance

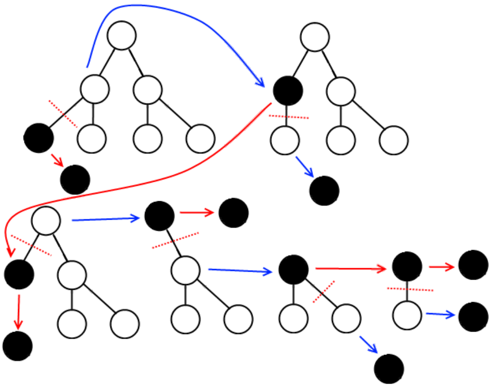
- Basic partitioning: try to evenly partition template
- Observation: algorithm runtime is proportional to $\sum_i \binom{k}{S_i} \cdot \binom{S_i}{a_i}$, i.e. $\sum_i |C_i| \cdot |C_{a_i}|$



Color-coding and FASCIA, FASTPATH

Template partitioning and work avoidance

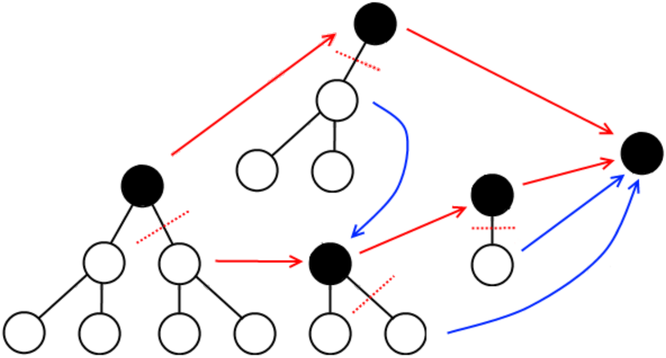
- Basic partitioning: try to evenly partition template
- Observation: algorithm runtime is proportional to $\sum_i \binom{k}{S_i} \cdot \binom{S_i}{a_i}$, i.e. $\sum_i |C_i| \cdot |C_{a_i}|$
- This sum can be minimized by a one-at-a-time partitioning approach



Color-coding and FASCIA, FASTPATH

Template partitioning and work avoidance

- Basic partitioning: try to evenly partition template
- Observation: algorithm runtime is proportional to $\sum_i \binom{k}{S_i} \cdot \binom{S_i}{a_i}$, i.e. $\sum_i |C_i| \cdot |C_{a_i}|$
- This sum can be minimized by a one-at-a-time partitioning approach
- On certain templates, this sum can be minimized by exploiting latent symmetry, HOWEVER ...



- Inner loop parallelization: **forall** $v \in G$
- Outer loop parallelization: **for** $i = 1$ to *Niter*
- Outer loop requires individual dynamic tables for each separate iteration, storage increases linearly with thread count
- Possible to do arbitrary combinations, e.g. a 12 thread machine with 2 outer loop threads each with 6 inner loop threads

```
Partition input template T
Select Niter to be performed
for i = 1 to Niter in parallel do
    Randomly color G
    for all S_i created during partitioning, a_i and p_i children do
        for all v ∈ G in parallel do
            ....
Take average of all Niter counts to be final count.
```

Color-coding and FASCIA, FASTPATH

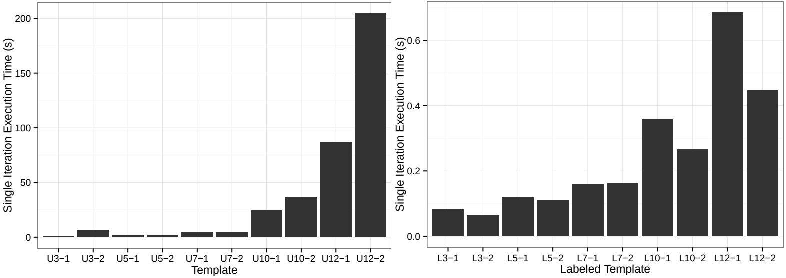
Distributed memory parallelization - partitioned counting

Partition input template T
Select *Niter* to be performed
for $i = 1$ to *Niter* **in parallel do**
    Randomly color G
    for all S_i created during partitioning, a_i and p_i children **do**
        Init table for V_r (vertex partition on task r)
        for all $v \in V_r$ **in parallel do**
            ....
        Set $\langle N, I, B \rangle :=$ Compress$(Table_{i,r})$
        All-to-all exchange of $\langle N, I, B \rangle$
        Update $Table_{i,r}$ based on information received
    Set $Count_r := Count_r + \sum_{v}^{V_r} \sum_{c}^{C_T} Count_{T,c,v}$
$Count \leftarrow$ Reduce$(Count_r)$
Scale *Count* based on Niter and colorful embed prob.

Color-coding and Fascia, FastPath

Fascia large network runtimes – shared memory

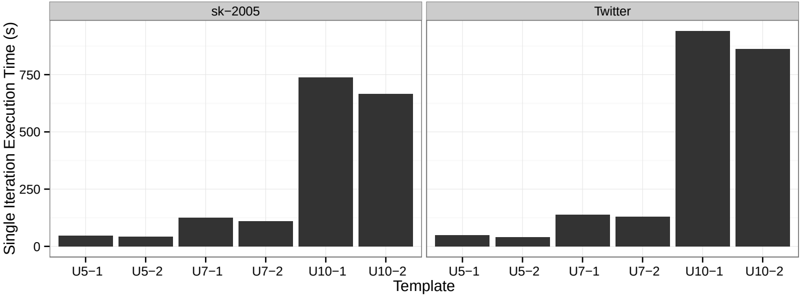
- Parallel (16 cores) results for all unlabeled (left) and labeled (right) templates on Portland network (n=1.6M, m=31M)
- 8 possible demographic labels (M/F and kid/youth/adult/senior)
- ~200 seconds for up to 12 vertex unlabeled template, less than 1 second for all labeled templates



Color-coding and FASCIA, FASTPATH

FASCIA large network runtimes – distributed memory

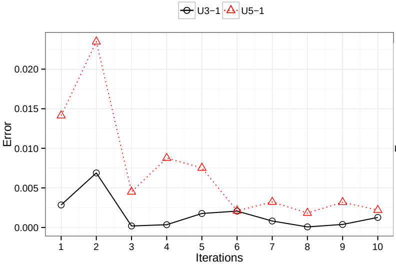
- Parallel (MPI+OpenMP, 16 tasks, 256 total cores) results for sk web crawl (n=44M, m=1.6B) and Twitter (n=44M, m=2.0B)
- Less than 15 minutes required to count 10 vertex templates

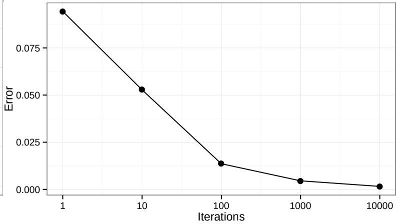


Color-coding and FASCIA, FASTPATH

FASCIA approximation Error in template counts

- Enron (left) with U3-1 and U5-1 and *H. Pylori* (right) across all 11 unique templates of size 7
- Error increases with template size and inversely to network size

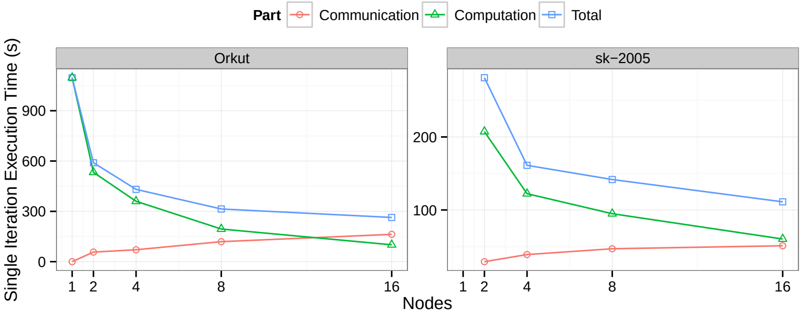




Color-coding and FASCIA, FASTPATH

FASCIA parallel Scaling – distributed memory

- Partitioned counting scaling on Orkut (n=3.1M, m=117M) and sk (n=44M, m=1.6B)
- Total speedups of about 4× for Orkut and 2.5× for sk for 16 tasks
- Communication costs increase proportionally to inter-task edges



Color-coding and FASCIA, FASTPATH

FASCIA comparison to previous work

- FASCIA and [Alon et al., 2008], both running on 16 cores

Network	Motifs	Subgraphs	Alon et al.	FASCIA	Improv.
S. cerevisiae	7	11	120s	7.5s	**16**×

- FASCIA (16 cores) and [Zhao et al., 2010b] (160 cores)

Network	Template	Naïve	PARSE	FASCIA	Speedup
GNP50k	U6-1	86s	∼11s	0.24s	**46**×

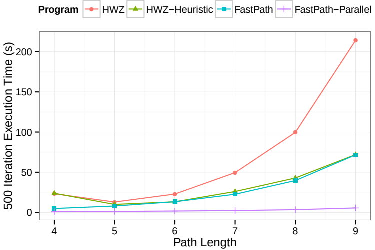
- FASCIA (16 cores) and [Zhao et al., 2012a] (1344 cores)

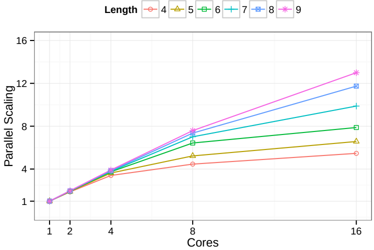
Network	Template	Naïve	SAHAD	FASCIA	Speedup
GNP100k	U7-3	5420s	∼360s	0.3s	**1,400**×

Color-coding and FASCIA, FASTPATH

FASTPATH execution time and scaling

- Observe close or faster execution time to state-of-the-art Hüffner et al. code [Hüffner et al., 2008]
- Speedup proportional to path length, due to increasing ratio of parallel to serial work





Color-coding and FASCIA, FASTPATH

Future Work

- Complex template structures, directed edges, edge labels
- Color-coding can theoretically be used to count all bounded tree-width subgraphs, we only consider tree-width=1 for FASCIA
- Color-coding can also find simple cycles
- FASCIA algorithm is computationally intensive, utilize GPU accelerators
- Implement known optimizations for FASTPATH [Hüffner et al., 2008, Gabr et al., 2012]

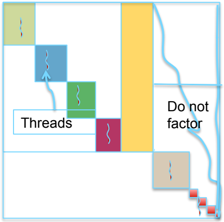
Connectivity Algorithms for Multicore Platforms

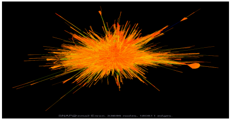
Part 2: **Multistep** – approaches for connected, weakly connected, and strongly connected components

Connectivity Algorithms for Multicore Platforms

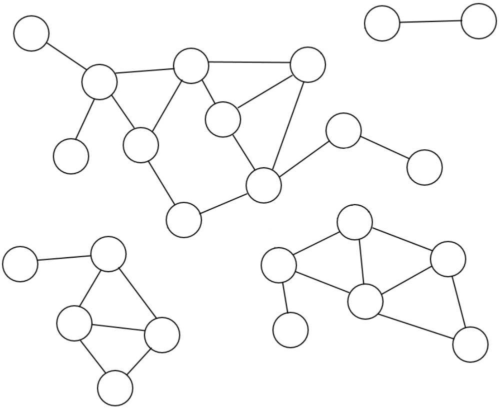
Motivation for parallel connectivity algorithms

- 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.
- Connectivity algorithms are also useful in general network analysis

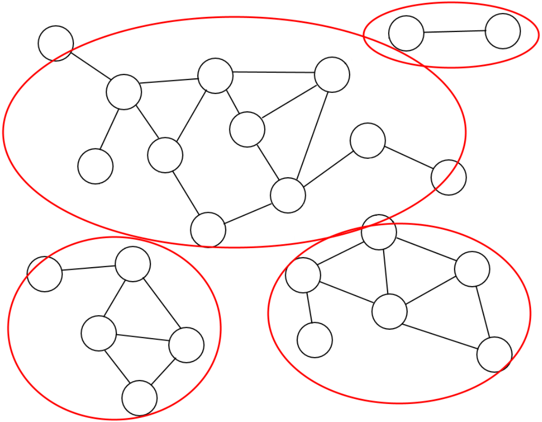




- CC: sets of vertices linked by undirected paths



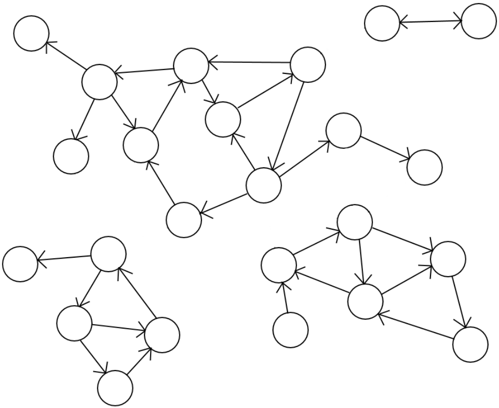
- CC: sets of vertices linked by undirected paths



Connectivity Algorithms for Multicore Platforms

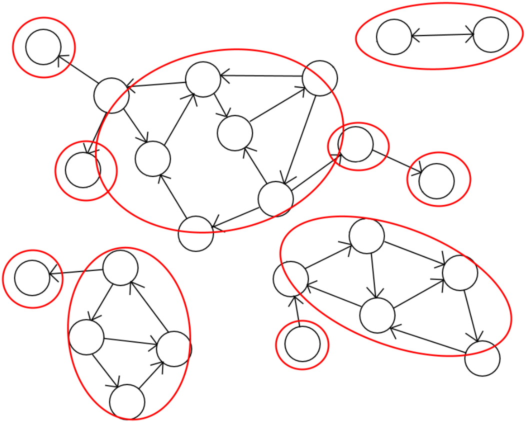
SCC, CC, WCC definitions

- CC: sets of vertices linked by undirected paths
- WCC: CC for directed graphs, when considering all edges undirected



Connectivity Algorithms for Multicore Platforms
SCC, CC, WCC definitions

- CC: sets of vertices linked by undirected paths
- WCC: CC for directed graphs, when considering all edges undirected
- SCC: maximal strongly connected subgraphs, path from every vertex to every other vertex



- Forward-Backward (FW-BW) and Trimming [Fleischer et al., 2000, W. McLendon III et al., 2005]
- Coloring [Orzan, 2004]
- State-of-the-art – FW-BW with low overhead task-parallel runtime environment and several optimizations [Hong et al., 2013]
- Others [Barnat and Moravec, 2006]
- Standard sequential algorithm is Tarjan's algorithm [Tarjan, 1972]
  - DFS-based recursive algorithm.
  - Not amenable to a scalable parallel algorithm.

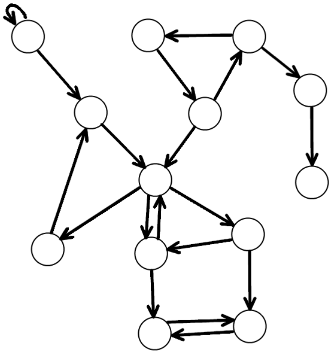
Connectivity Algorithms for Multicore Platforms

Multistep Method

procedure MULTISTEP($G(V, E)$)
  $T \leftarrow$ MS-SimpleTrim(G)
  $V \leftarrow V \setminus T$
  Select $v \in V$ for which $d_{in}(v) * d_{out}(v)$ is maximal
  $D \leftarrow$ BFS($G(V, E(V)), v$)
  $S \leftarrow D \cap$ BFS($G(D, E'(D)), v$)
  $V \leftarrow V \setminus S$
  while NumVerts(V) $> n_{\text{cutoff}}$ **do**
    $C \leftarrow$ MS-Coloring($G(V, E(V))$)
    $V \leftarrow V \setminus C$
  Tarjan($G(V, E(V))$)

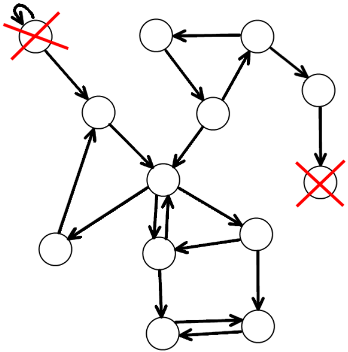
- Do simple trimming
- Perform single iteration of FW-BW to remove giant SCC
- Do coloring until some threshold of remaining vertices is reached
- Finish with serial algorithm
- Easily extendable to CC, WCC

- Used to find trivial SCCs



Connectivity Algorithms for Multicore Platforms
Trimming algorithm

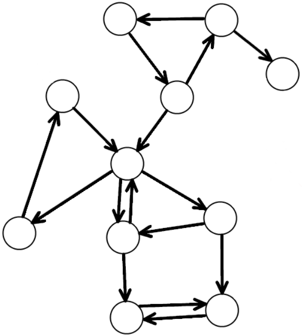
- Used to find trivial SCCs
- Detect and prune all vertices that have an in/out degree of 0 or an in/out degree of 1 with a self loop (simple trimming)



Connectivity Algorithms for Multicore Platforms

Trimming algorithm

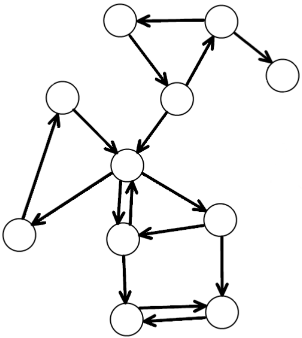
- Used to find trivial SCCs
- Detect and prune all vertices that have an in/out degree of 0 or an in/out degree of 1 with a self loop (simple trimming)



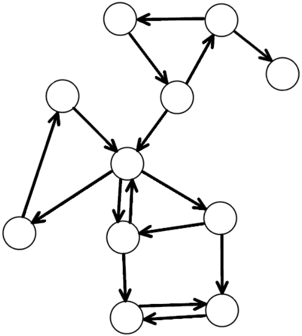
Connectivity Algorithms for Multicore Platforms

Trimming algorithm

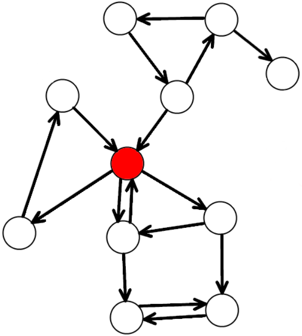
- Used to find trivial SCCs
- Detect and prune all vertices that have an in/out degree of 0 or an in/out degree of 1 with a self loop (simple trimming)
- Repeat iteratively until no more vertices can be removed (complete trimming)



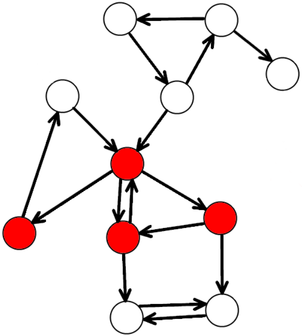
- Select pivot



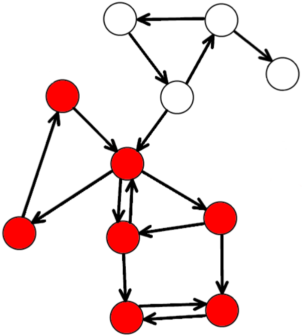
- Select pivot



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



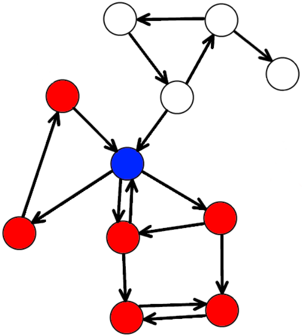
- Select pivot
- Find all vertices that can be reached from the pivot (**descendant** (D))



Connectivity Algorithms for Multicore Platforms

Forward-Backward (FW-BW) algorithm

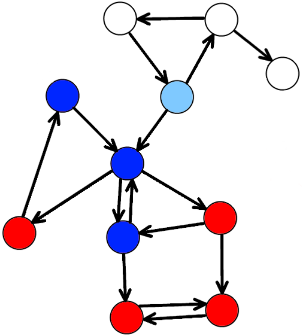
- Select pivot
- Find all vertices that can be reached from the pivot (**descendant** (D))
- Find all vertices that can reach the pivot (**predecessor** (P))



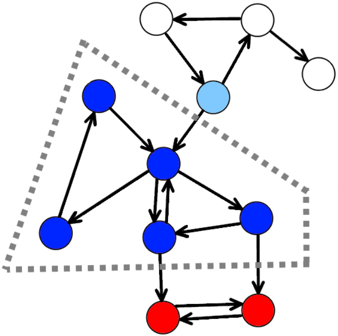
Connectivity Algorithms for Multicore Platforms

Forward-Backward (FW-BW) algorithm

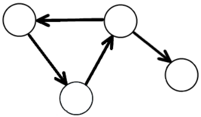
- Select pivot
- Find all vertices that can be reached from the pivot (**descendant** (D))
- Find all vertices that can reach the pivot (**predecessor** (P))



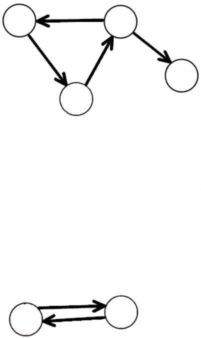
- Select pivot
- Find all vertices that can be reached from the pivot (**descendant** (D))
- Find all vertices that can reach the pivot (**predecessor** (P))
- Intersection of those two sets is an SCC ($S = P \cap D$)



- 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 ($S = P \cap D$)

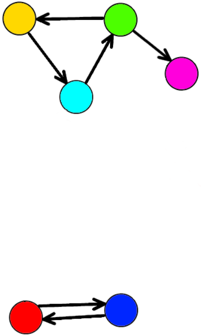


- Consider vertex identifiers as *colors*



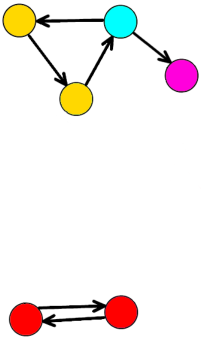
Connectivity Algorithms for Multicore Platforms
Coloring algorithm

- Consider vertex identifiers as *colors*



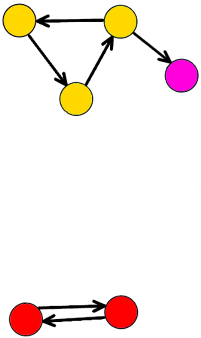
Connectivity Algorithms for Multicore Platforms
Coloring algorithm

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



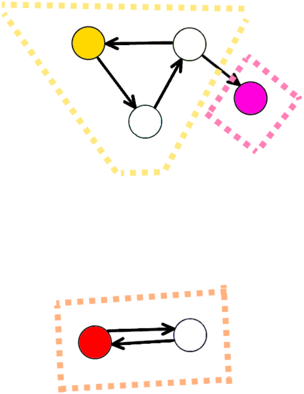
Connectivity Algorithms for Multicore Platforms
Coloring algorithm

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



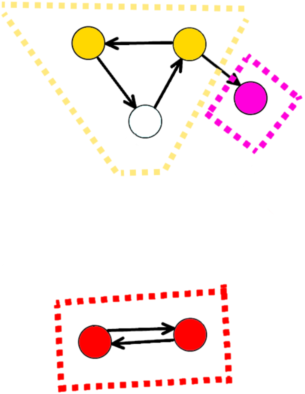
Connectivity Algorithms for Multicore Platforms
Coloring algorithm

- 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



Connectivity Algorithms for Multicore Platforms
Coloring algorithm

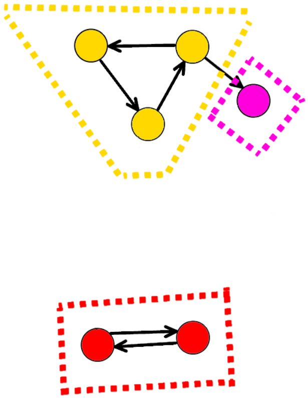
- Consider vertex identifiers as *colors*
- Highest colors are propagated **forward** through the network to create sets
- Consider the original vertex of each color to be the *root* of a new SCC
- Each SCC is all vertices (of the same color as the root) reachable **backward** from each root.



Connectivity Algorithms for Multicore Platforms

Coloring algorithm

- 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



Connectivity Algorithms for Multicore Platforms

Multistep parallelization and optimization for multicore

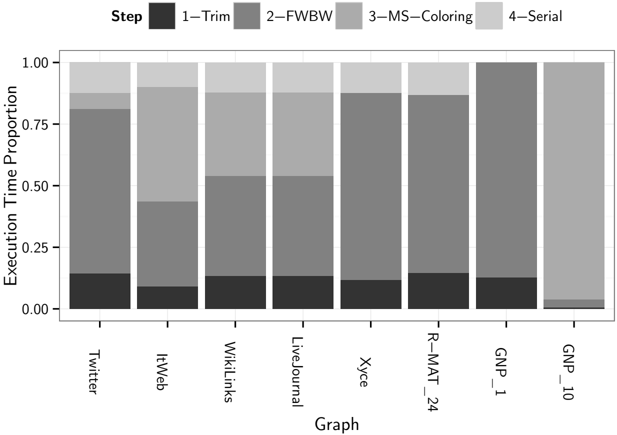
Multistep – primary subroutines are BFS and color propagation

- Thread-owned queues, combine asynchronously
- Avoid all explicit locking when possible for shared data
- Boolean vs. bitmap for status marking
- Per-socket graph partitioning for multi-socket systems
- Direction-optimizing BFS [Beamer et al., 2012]

Connectivity Algorithms for Multicore Platforms

Multistep timing breakdown

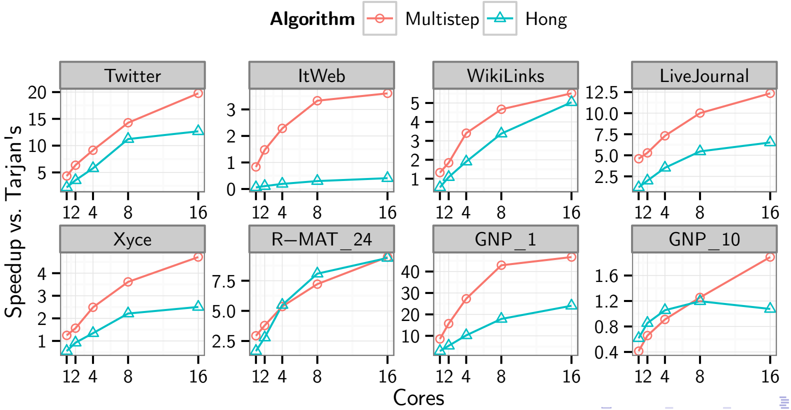
- The graph structure determines the runtime of different stages
- Large number of non-trivial SCCs affects FW-BW (tasking overhead)
- Large diameter or a large SCC affects coloring



Connectivity Algorithms for Multicore Platforms

Multistep strong scaling

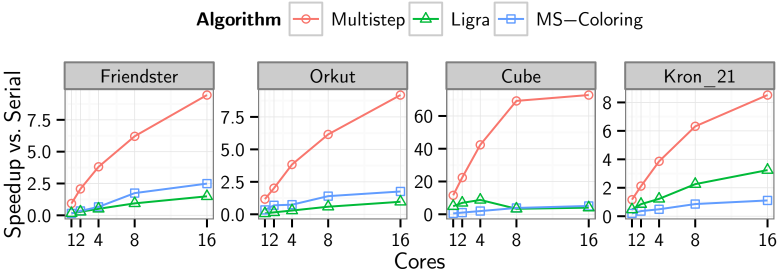
- Both Multistep and Hong et al scale well in most graphs.
- Lots of small non-trivial SCCs in ItWeb affects the performance of Hong et all.
- Relative to Tarzan's Algorithm, Multistep results in better speedups.



Connectivity Algorithms for Multicore Platforms

Strong scaling for CC

- Multistep for CC compared to MS-Coloring and Ligra CC color-based approach
- Scaling shown against baseline serial BFS approach



- Further explore effects of BFS/color propagation optimizations on various multicore system configurations
- Distributed memory implementation of Multistep
- Biconnected components, triconnected components, etc.

Distributed Graph Layout

Part 3: **Distributed Graph Layout** – PULP Partitioning & DGL vertex ordering

Distributed Graph Layout

Partitioning

- **Graph Partitioning**: Given a graph $G(V, E)$ and p processes or tasks, assign each task a p-way disjoint subset of vertices and their incident edges from G
  - Balance constraints – (weighted) vertices per part, (weighted) edges per part
  - Quality metrics – edge cut, communication volume, maximal per-part edge cut
- We consider:
  - Balancing edges **and** vertices per part
  - Minimizing edge cut (EC) **and** maximal per-part edge cut (EC_{max})

- Lots of graph algorithms follow a certain iterative model
  - BFS, SSSP, FASCIA subgraph counting
  - Computation, synchronization, communication, synchronization, computation, etc.
- Computational load: proportional to vertices and edges per-part
- Communication load: proportional to total edge cut and max per-part cut
- We want to minimize the maximal time among tasks for each comp/comm stage

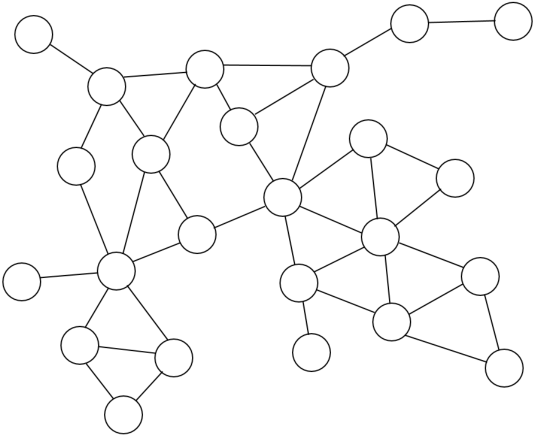
Distributed Graph Layout

Partitioning - HPC Approaches

- (Par)METIS [Karypis and Kumar], PT-SCOTCH [Chevalier and Pellegrini, 2008], Chaco [Hendrickson and Leland, 1995], etc.
- Multilevel methods:
  - *Coarsen* the input graph in several iterative steps
  - At coarsest level, partition graph via local methods following balance constraints and quality objectives
  - Iteratively *uncoarsen* graph, refine partitioning
- **Problem 1**: Designed for traditional HPC scientific problems (e.g. meshes) – limited balance constraints and quality objectives
- **Problem 2**: Multilevel approach – high memory requirements, can run slowly and lack scalability

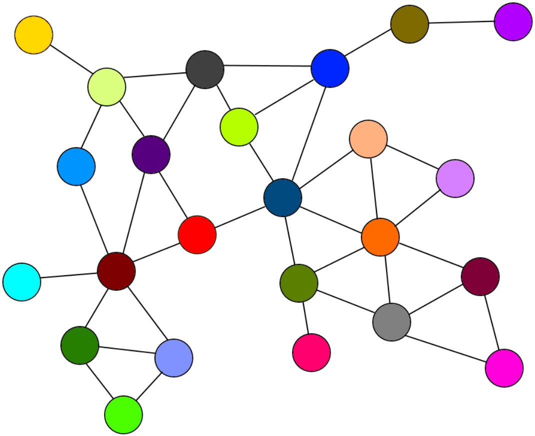
- **Label propagation**: initialize a graph with n labels, iteratively assign to each vertex the maximal per-label count over all neighbors to generate clusters [Raghavan et al., 2007]
  - Clustering algorithm - dense clusters hold same label
  - Fast - each iteration in $O(n + m)$
  - Naïvelyparallel - only per-vertex label updates
  - *Observation*: Possible applications for large-scale small-world graph partitioning

Distributed Graph Layout
Label Propagation



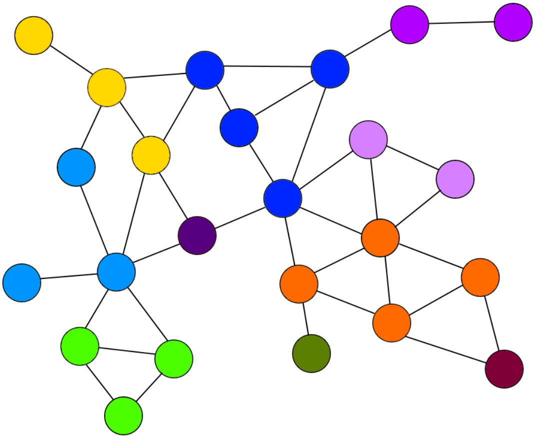
Distributed Graph Layout

Label Propagation



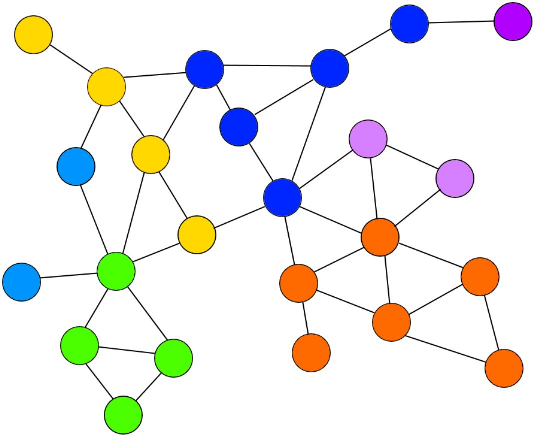
Distributed Graph Layout

Label Propagation



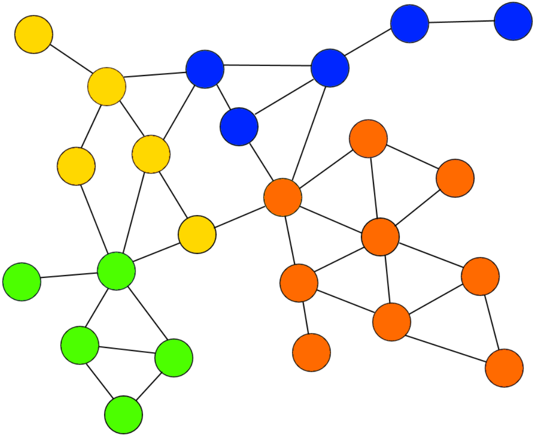
Distributed Graph Layout

Label Propagation



Distributed Graph Layout

Label Propagation



Distributed Graph Layout

Partitioning - "Big Data" Approaches

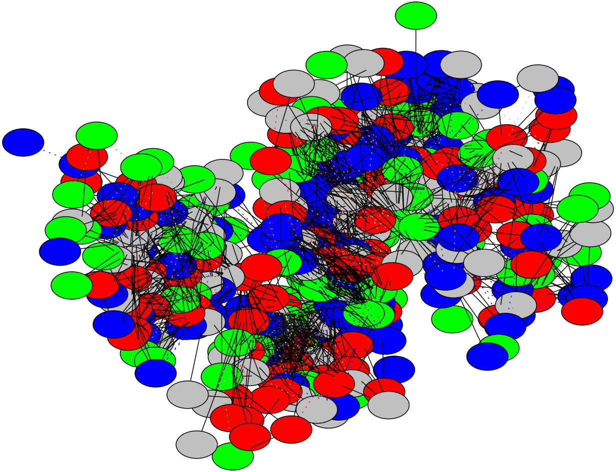
- Methods designed for small-world graphs (e.g. social networks and web graphs)
- Exploit label propagation/clustering for partitioning:
    - Multilevel methods - use label propagation to coarsen graph [Wang et al., 2014, Meyerhenke et al., 2014]
    - Single level methods - use label propagation to directly create partitioning [Ugander and Backstrom, 2013, Vaquero et al., 2013]
- **Problem 1**: Multilevel methods still can lack scalability, might also require running traditional partitioner at coarsest level
- **Problem 2**: Single level methods can produce sub-optimal partition quality

PuLP : **P**artitioning **U**sing **L**abel **P**ropagation

- Utilize label propagation for:
  - Vertex balanced partitions, minimize edge cut (PuLP)
  - Vertex and edge balanced partitions, minimize edge cut (PuLP-M)
  - Vertex and edge balanced partitions, minimize edge cut and maximal per-part edge cut (PuLP-MM)
  - Any combination of the above - multi objective, multi constraint

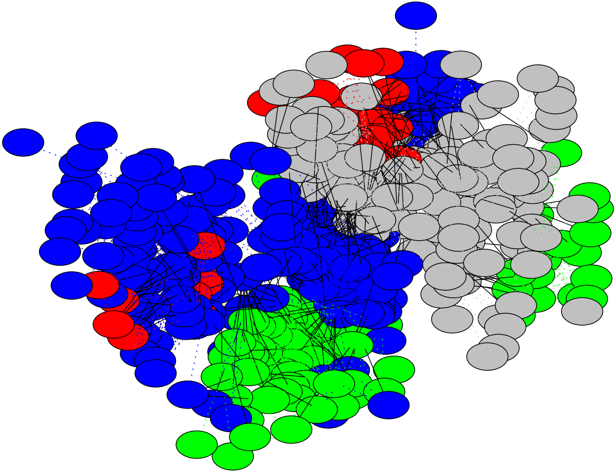
Distributed Graph Layout
PuLP algorithm

- Randomly initialize partition labels



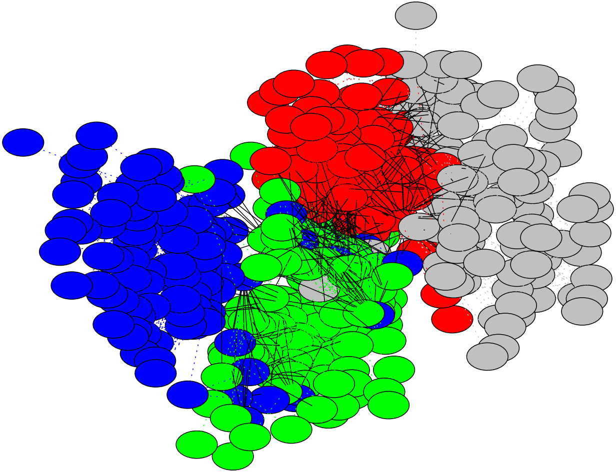
Distributed Graph Layout
PuLP algorithm

- Randomly initialize partition labels
- Run label propagation to create initial parts



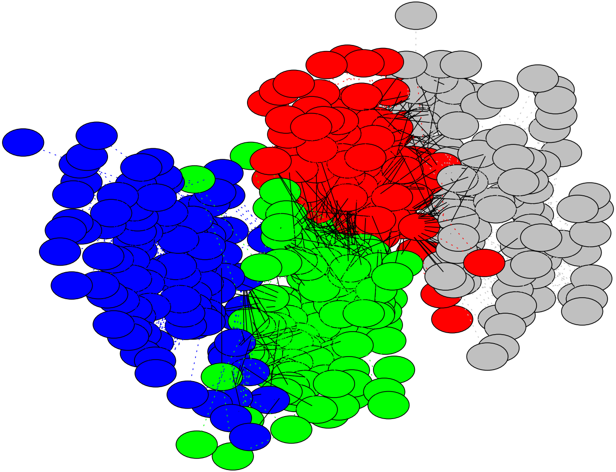
Distributed Graph Layout
PuLP algorithm

- Randomly initialize partition labels
- Run label propagation to create initial parts
- Iteratively balance for vertices, minimize edge cut



Distributed Graph Layout
PuLP algorithm

- Randomly initialize partition labels
- Run label propagation to create initial parts
- Iteratively balance for vertices, minimize edge cut
- Balance for edges, minimize per-part edge cut



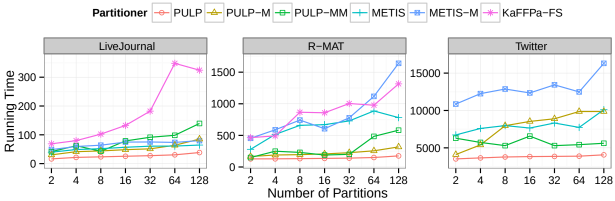
Distributed Graph Layout
Vertex Ordering

- We consider layout as both partitioning-vertex ordering
- Per-part vertex ordering – increase locality of memory references
- RCM commonly used in sparse matrix and graph applications [Cuthill and McKee, 1969]
- DGL ordering – RCM approximation that is both faster to calculate and can improve computation time for various algorithms

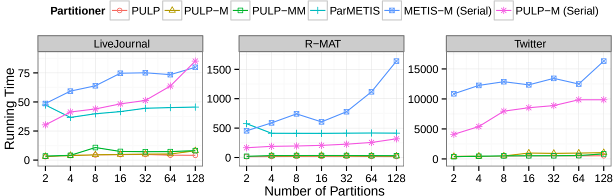
Distributed Graph Layout

PuLP Running Times - Serial (top), Parallel (bottom)

- In serial, PuLP-MM runs 1.7× faster (geometric mean) than next fastest of METIS and KaFFPa



- In parallel, PuLP-MM runs 14.5× faster (geometric mean) than next fastest (ParMETIS times are fastest of 1 to 256 cores)



Distributed Graph Layout

PuLP memory utilization for 128 partitions

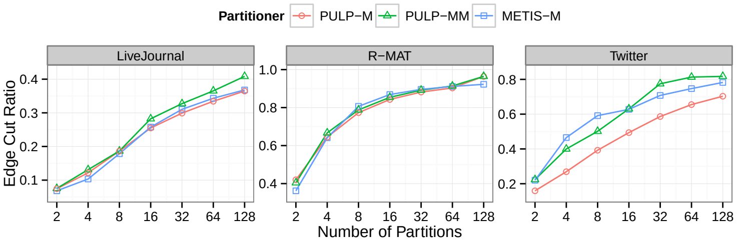
- PuLP utilizes minimal memory, $O(n)$, 8-39$\times$ less than other partitioners
- Savings are mostly from avoiding a multilevel approach

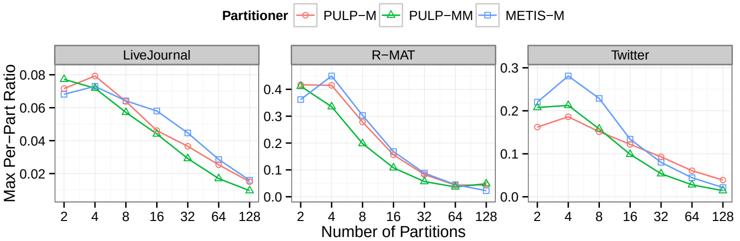
Network	Memory Utilization				**Improv.**
	METIS-M	KaFFPa	PuLP-MM	Graph Size	
LiveJournal	7.2 GB	5.0 GB	0.44 GB	0.33 GB	21$\times$
Orkut	21 GB	13 GB	0.99 GB	0.88 GB	23$\times$
R-MAT	42 GB	-	1.2 GB	1.02 GB	35$\times$
DBpedia	46 GB	-	2.8 GB	1.6 GB	28$\times$
WikiLinks	103 GB	42 GB	5.3 GB	4.1 GB	25$\times$
sk-2005	121 GB	-	16 GB	13.7 GB	8$\times$
Twitter	487 GB	-	14 GB	12.2 GB	39$\times$

Distributed Graph Layout

PuLP quality - Edge Cut and Edge Cut Max

- PuLP-M produces better edge cut than METIS-M over most graphs
- PuLP-MM produces better max edge cut than METIS-M over most graphs

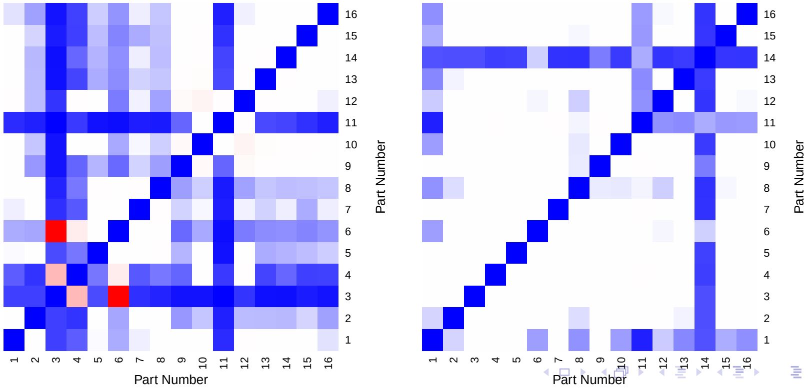




Distributed Graph Layout

PuLP– balanced communication

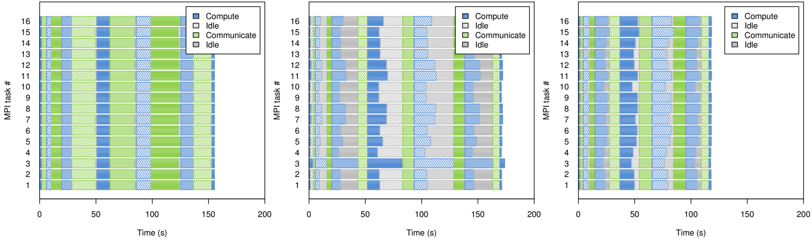
- uk-2005 graph from LAW, METIS-M (left) vs. PuLP-MM (right)
- Blue: low comm; White: avg comm; Red: High comm
- PuLP reduces max inter-part communication requirements and balances total communication load through all tasks



Distributed Graph Layout

PuLP balancing computation and communication

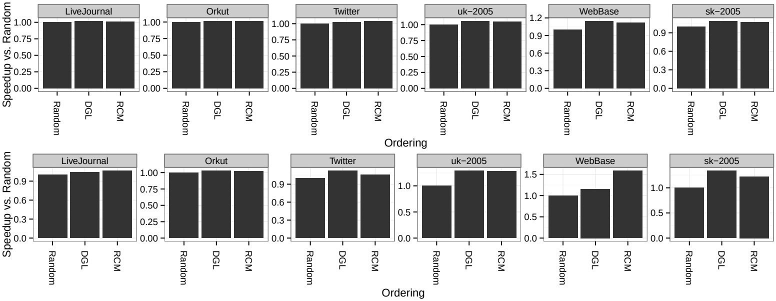
- 16 tasks for Fascia with LiveJournal graph with random (left), METIS (middle), and PuLP (right) partitionings
- Note tradeoff between work balance and communication load, need to account for both in many irregular graph applications



Distributed Graph Layout

DGL ordering – computational speedups

- Speedup with DGL ordering vs. random and RCM
- With 16 parts for FASCIA (top) and 64 parts for SSSP (bottom)
- Better speedups on larger graphs, cache performance more important



Future Work

- Explore techniques for avoiding local minima, such as simulated annealing, etc.
- Further parallelization in distributed environment for massive-scale graphs
- Explore tradeoff and interactions in various parameters and iteration counts

Conclusions

Improvements with implemented algorithms

- Algorithmic improvements give FASCIA and FASTPATH orders-of-magnitude speedup over prior art
- Memory improvements allow FASCIA to perform counts of non-trivial subgraphs on graphs an order-of-magnitude larger than has ever previously been attempted
- Multistep demonstrates speedups compared to previous and current state-of-the-art component decomposition algorithms
- Partitioning with PULP gives considerable reduction in computation and memory requirements relative to the current state of the art with minimal to no reduction in cut quality.

Conclusions
Overall lessons learned

- Parallel algorithm design
  - Minimizing synchronization costs
  - Keeping memory accesses local
  - Even work distribution among threads and tasks
- Identifying algorithmic traits across graph algorithms
  - Many graph algorithms follow an iterative nested-loop structure
  - Many graph algorithms use common subroutines such as BFS, etc.
- Storing and organizing graphs efficiently in memory
  - Optimizing layout for specific graph types and applications
  - Balance cost tradeoffs for both communication and computation
  - Need for parameter tuning and experimental evaluation

Backup Slides

Color-coding and FASCIA, FASTPATH

Template partitioning, work reduction

- If $|a_i| = 1$ or $|p_i| = 1$, we can do $\sim\frac{1}{k}$ of the original work, $C_a = color(v)$ or $C_p = color(u)$ is fixed
- If table$[a][v][\,] =$ NULL, we avoid all work for v
- Can do the same for all $u \in N(v)$
- Order the way in which we process all S_i to minimize memory usage

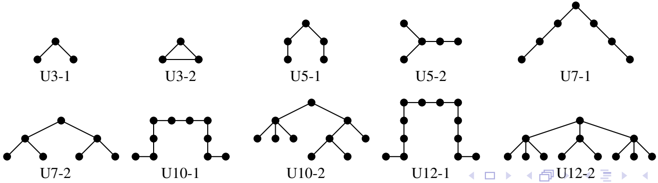
for all S_i created during partitioning, a_i and p_i children **do**
    for all $v \in G$, where table$[a_i][v]$!= NULL **do**
        for all C possibly mapped to S_i **do**
            for all C_p, C_a from C, where C_a, C_p is fixed if $|a_i| = 1$ *or* $|p_i| = 1$ **do**
                for all $u \in N(v)$, where table$[p_i][u]$!= NULL **do**
                    count += table$[a_i][v][C_a]$·table$[p_i][u][C_p]$
            Set table$[S_i][v][C]$:= *count*

Color-coding and FASCIA, FASTPATH

Networks and templates analyzed

- Gordon at SDSC, 2× Xeon E5 @ 2.6GHz (Sandy Bridge), 64 GB DDR3
- Database of Interacting Proteins, SNAP, and Virginia Tech NDSSL

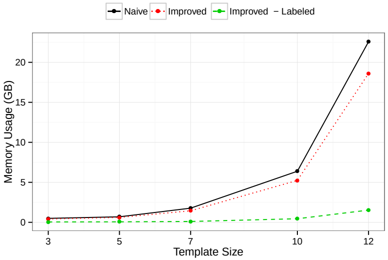
Network	n	m	d_{avg}	d_{max}
Portland	1,588,212	31,204,286	39.3	275
PA Road Net	1,090,917	1,541,898	2.8	9
Slashdot	82,168	438,643	10.7	2510
Enron	33,696	180,811	10.7	1383
E. coli	2,546	11,520	9.0	178
S. cerevisiae	5,021	22,119	8.8	289
H. pylori	687	1,352	3.9	54
C. elegans	2,391	3,831	3.2	187



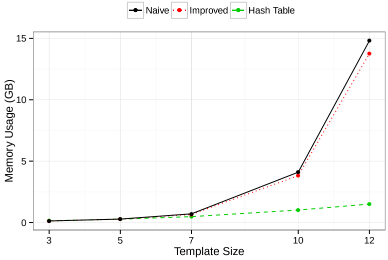
Color-coding and FASCIA, FASTPATH

FASCIA memory savings – array and table methods

- Memory requirements on Portland and PA Road network for improved array (left) and hash table (right)
- Using all UX-1 chain templates
- Up to 20%-90% savings versus naïve method



Portland Network with Improved Array

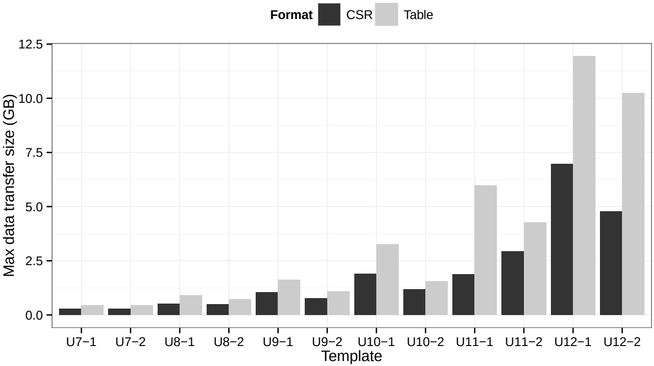


PA Road Network with Hash Table

Color-coding and FASCIA, FASTPATH

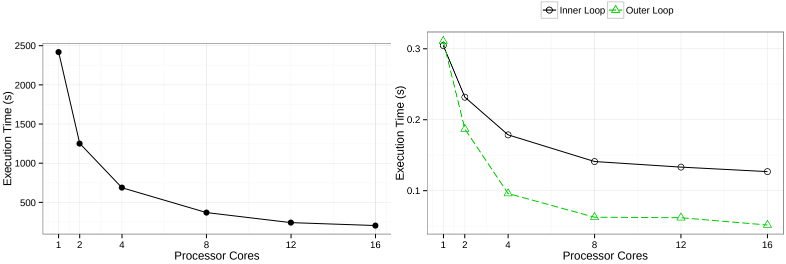
FASCIA communication savings – CSR representation

- Comparison between maximal communication in GB required for partitioned counting on Portland network
- Geometric mean of 35% reduction, maximal of 77% reduction



FASCIA parallel Scaling – shared memory

- Inner loop for large graphs (**forall** $v \in G$)
- Outer loop for small graphs (**for** $i = 1$ to $Niter$)
- U12-2 Template on Portland (left) and Enron (right), 16 threads
- About 12× speedup for inner loop on Portland
- About 6× speedup for outer loop on Enron, 3.5× for inner loop



Our Contributions

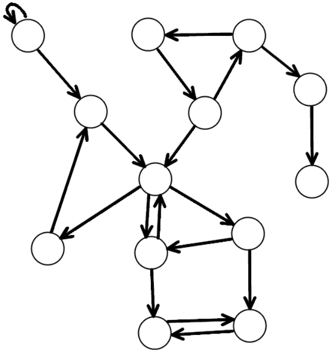
- A Multistep method for SCC detection:
  - Data parallel SCC detection with the advantages of previous methods.
  - Uses minimal synchronization and fine-grained locking.
- Faster and scales better than the previous methods.
- Up to 9x faster than state-of-the-art Hong et al's method.
- Easily extendable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

Connectivity Algorithms for Multicore Platforms
Observations on previous algorithms

- 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 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**

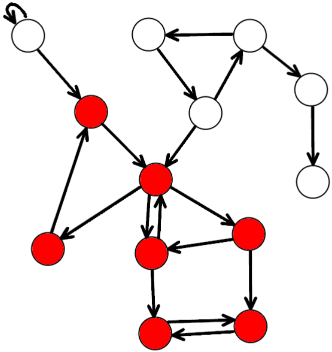
Multistep Method

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



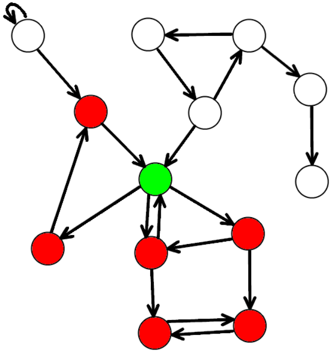
Multistep Method

- 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)



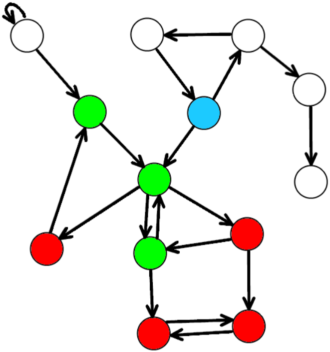
Multistep Method

- 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)



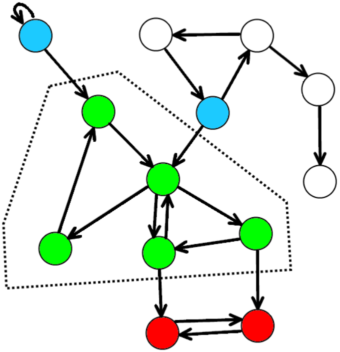
Multistep Method

- 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

Extending Multistep to CC and WCC

procedure MULTISTEP-(W)CC($G(V, E)$)
  $T \leftarrow$ MS-SimpleTrim(G)
  $V \leftarrow V \setminus T$
  Select $v \in V$ for which $d_{in}(v) * d_{out}(v)$ is maximal
  $S \leftarrow$ BFS($G(V, E(V) \cup E'(V)), v$)
  $V \leftarrow V \setminus S$
  while NumVerts(V) $> n_{\text{cutoff}}$ **do**
    $C \leftarrow$ MS-Coloring($G(V, E(V) \cup E'(V))$)
    $V \leftarrow V \setminus C$
  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

Implementation Details
Multistep on GPU

GPU description template algorithm, bfs and coloring, etc.

Implementation Details
Multistep on GPU

implementation details, 3 approaches, what to optimize for

Implementation Details
Multistep on GPU

implementation details, delayed

Implementation Details
Multistep on GPU

implementation details, manhattan local

Implementation Details
Multistep on GPU

implementation details, manhattan global

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)

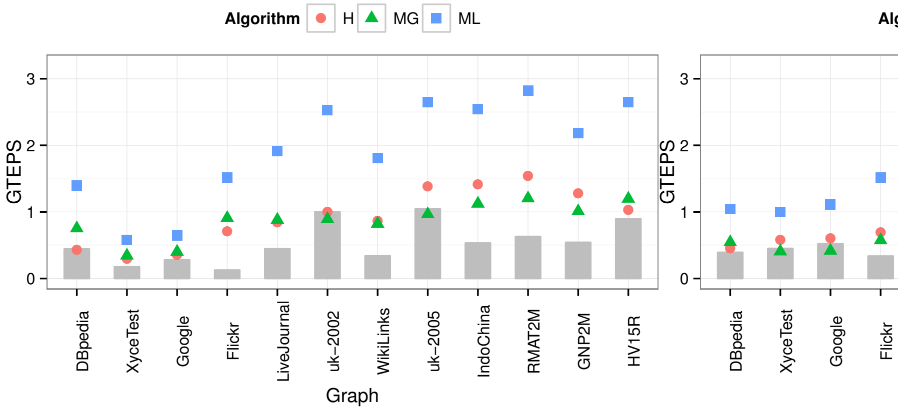
Performance Results
Test Environment and Graphs

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

Network	n	m	deg avg	deg max	$\widetilde{D}$	(S)CCs count	(S)CCs 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
Friendster	66M	1800M	53	5.2K	34	70	66M
Orkut	3.1M	117M	76	33K	11	1	3.1M
Cube	2.1M	62M	56	69	157	47K	2.1M
Kron_21	1.5M	91M	118	213K	8	94	1.5M

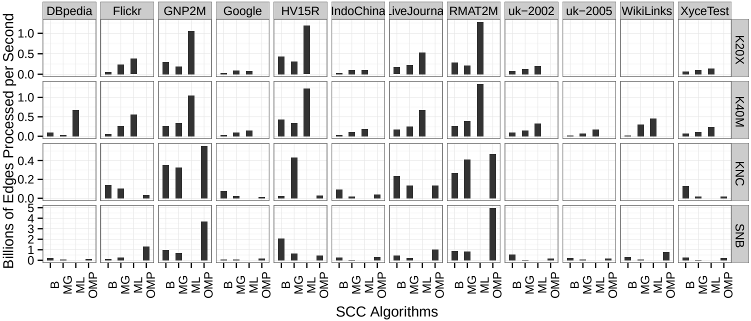
Performance Results - GPU

BFS and colorings



Performance Results - GPU

SCC results



Results

Test Environment and Graphs

- Test system: *Compton*
  - Intel Xeon E5-2670 (Sandy Bridge), dual-socket, 16 cores, 64 GB memory.
- Test graphs:
  - LAW graphs from UF Sparse Matrix, SNAP, MPI, Koblenz
  - Real (one R-MAT), small-world, 60 K–70 M vertices, 275 K–2 B edges
- Test Algorithms:
  - **METIS** - single constraint single objective
  - **METIS-M** - multi constraint single objective
  - **ParMETIS** - METIS-M running in parallel
  - **KaFFPa** - single constraint single objective
  - **PuLP** - single constraint single objective
  - **PuLP-M** - multi constraint single objective
  - **PuLP-MM** - multi constraint multi objective
- Metrics: 2–128 partitions, serial and parallel running times, memory utilization, edge cut, max per-partition edge cut

Bibliography I

N. Alon, R. Yuster, and U. Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S.C. Sahinalp. Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13):i241–i249, 2008.

J. Barnat and P. Moravec. Parallel algorithms for finding SCCs in implicitly given graphs. *Formal Methods: Applications and Technology*, 4346:316–330, 2006.

S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proc. Supercomputing (SC)*, 2012.

Cédric Chevalier and François Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6):318–331, 2008.

E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. 1969 24th Nat'l. Conf.*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

L.K. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In *Parallel and Distributed Processing*, volume 1800 of *LNCS*, pages 505–511. Springer Berlin Heidelberg, 2000.

Haitham Gabr, Alin Dobra, and Tamer Kahveci. From uncertain protein interaction networks to signaling pathways through intensive color coding. In *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, pages 111–122. World Scientific, 2012.

Bruce Hendrickson and Robert W Leland. A multi-level algorithm for partitioning graphs. In *Supercomputng*, 1995.

S. Hong, N. C. Rodia, and K. Olukotun. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *Proc. Supercomputing*, 2013.

Falk Hüffner, Sebastian Wernicke, and Thomas Zichner. Algorithm engineering for color-coding with applications to signaling pathway detection. *Algorithmica*, 52(2):114–132, 2008.

Bibliography II

G. Karypis and V. Kumar. MeTis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. version 5.1.0. `http://glaros.dtc.umn.edu/gkhome/metis/metis/download`, last accessed July 2014.

Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning complex networks via size-constrained clustering. *CoRR*, abs/1402.3281, 2014.

S. Omidi, F. Schreiber, and A. Masoudi-Nejad. MODA: an efficient algorithm for network motif discovery in biological networks. *Genes Genet Syst*, 84(5):385–395, 2009.

S. Orzan. *On Distributed Verication and Veried Distribution*. PhD thesis, Free University of Amsterdam, 2004.

U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.

J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *Proc. Web Search and Data Mining (WSDM)*, 2013.

Luis Vaquero, Félix Cuadrado, Dionysios Logothetis, and Claudio Martella. xDGP: A dynamic graph processing system with adaptive partitioning. *CoRR*, abs/1309.1049, 2013.

W. McLendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

Lu Wang, Yanghua Xiao, Bin Shao, and Haixun Wang. How to partition a billion-node graph. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 568–579. IEEE, 2014.

Z. Zhao, M. Khan, V. S. A. Kumar, and M. V. Marathe. Subgraph enumeration in large social contact networks using parallel color coding and streaming. In *Proc. 39th Int'l. Conf. on Parallel Processing (ICPP)*, pages 594–603, 2010a.

Bibliography III

Z Zhao, M Khan, VSA Kumar, and M Marathe. Subgraph enumeration in large social contact networks using parallel color coding and streaming. In *39th International Conference on Parallel Processing*, 2010b.

Z Zhao, G Wang, A R Butt, M Khan, V S Kumar, and M V Marathe. SAHad: Subgraph analysis in massive networks using hadoop. In *IEEE 26th International Parallel and Distributed Processing Symposium*, 2012a.

Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. S. A. Kumar, and M. V. Marathe. SAHAD: Subgraph analysis in massive networks using Hadoop. In *Proc. 26th Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, pages 390–401, 2012b.