SAND2013-9044C

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

George M. Slota*, Sivasankaran Rajamanickam†, and Kamesh Madduri*
*Computer Science and Engineering, The Pennsylvania State University. Email: gms5016@psu.edu, madduri@cse.psu.edu
†Scalable Algorithms Department, Sandia National Laboratories, Email: srajama@sandia.gov

***Abstract*—Finding the strongly connected components (SCCs) of a directed graph is a fundamental problem used in many fields. Tarjan's algorithm is an efficient serial algorithm to find the SCCs, but relies on a depth-first search, which is hard to parallelize. Several parallel algorithms have been proposed, but show poor load balance and slow performance on many real world graphs. We introduce the Multistep Method, which reduces total work when compared to the well known FW-BW algorithm and combines the advantages of several parallel SCC finding algorithms. It performs well on real world graphs irrespective of the properties of the graph, such as the size of the largest SCC or the overall number of SCCs. Our algorithm delivers up to 50× speedup over the serial Tarjan's algorithm, while being capable of fully decomposing a 1.5 billion edge graph in under two seconds. In comparison with other state-of-the- artparallel codes, our algorithm delivers up to 10× speedup. We generalize our approach to algorithms for finding connected and weakly connected components as well as introduce a novel algorithm for determining biconnected components in order to demonstrate how our approaches are applicable to a broader set of algorithms.**

***Keywords*-strongly connected components; BFS; coloring; multicore algorithms; performance analysis**

I. Introduction

The strongly connected components (SCCs) of a directed graph are the maximal subgraphs where every vertex within the subgraph can reach and can be reached by every other vertex within the subgraph. The decomposition of a directed graph into its strongly connected components is a useful analytic tool in many applications like social network analysis [1], compiler design, radiation transport solvers [2], and computing the block triangular form for linear solvers and preconditioners [3], [4].

The connected components of an undirected graph are the maximal subgraphs where every vertex in the subgraph has a path to every other vertex. Weakly connected components in a directed graph are the equivalent to connected components in undirected graphs if the direction of edges is ignored.

The biconnected components (BCCs) of a graph are the maximal biconnected subgraphs, which are subgraphs that will remain connected on the removal of any given vertex in the subgraph. Determining biconnectivity of a graph is especially important in networking and routing to determine redudancy.

II. Background

Determing the connected, strong connected, weakly connected, and biconnected components of graphs and networks has been extrensively studied. Linear time serial algorithms have been developed for each problem. However, for strongly and biconnected components, parallelization is not trivial. Several approaches have been proposed, but often the performance of these approaches on real world graphs is less than optimal.

A. Strongly Connected Components

There are several existing serial and parallel algorithms that have been used to determine the SCCs of a graph. This section will give a general overview of the most widely used ones. Popular serial methods include Tarjan's and Kosaraju's algorithms, while parallel methods are the Forward-Backward algorithm and Coloring.

1) Serial Algorithms: The primary serial algorithms used for determining strongly connected components within a graph are Tarjan's [5] and Kosaraju's [6] algorithms. Tarjan's algorithm uses a recursive depth first search (DFS) to form a search tree of explored vertices. The roots of the subtrees of the search tree form roots of strongly connected components. Kosaraju's algorithm performs two passes of the graph. It initially performs a DFS, placing each vertex onto a stack after it has been fully explored. After all vertices have been placed onto the stack, a vertex is popped from the stack and a DFS or BFS search is performed on the transpose of the graph. All vertices that can be reached by this vertex (that have not already been explored a second time) form an SCC. The runtime for both algorithms is in $O(n+m)$ time, where n is the number of vertices and m is the number of edges in a graph. However, since Tarjan's algorithm only requires a single search as apposed to Kosaraju's two, it is often faster in practice.

2) Forward-Backward: The Forward-Backward (FW-BW) algorithm [7] is given in Algorithm 1 and can be described as follows: Given a graph G, a single *pivot* vertex v is selected. This can be done either randomly or through simple heuristics. A BFS/DFS search is conducted starting from this vertex to determine all vertices which are reachable (forward sweep). These vertices form the *descendants* set (D). Another BFS/DFS search is performed from v but

on the transpose graph. This search (backward sweep) will find the set (P) of all vertices than can reach v called the *predecessors*.

Algorithm 1 Forward-Backward Algorithm

```
1:  procedure FWBW(G)
2:      if G = ∅ then
3:          return ∅
4:      select pivot v
5:      D ← DESC(G, v)
6:      P ← PRED(G, v)
7:      R ← (G \ (P ∪ D)
8:      S ← (D ∩ P)
9:      FWBW(D \ S)
10:     FWBW(P \ S)
11:     FWBW(R)
```

The intersection of these two sets form an SCC ($S = D \cap P$) that has the pivot v in it. If we remove all vertices in S from the graph, we can have up to three remaining distinct sets: $(D \setminus S), (P \setminus S)$, and *remainder* R, which is the set of vertices that we have not explored during either search from v. The FW-BW algorithm can then be recursively called on each of these three sets.

Possible parallelism exists for this algorithm on two levels. Primarily, as each of the three sets are distinct, they can each be explored in parallel. In addition, each of the forward and backward searches can be trivially parallelized using a standard parallel BFS or even DFS (since we don't care about ordering, only reachability).

Commonly, before FW-BW is called, *trimming* is performed. The trimming procedure was initially proposed as an extension to FW-BW [2], to remove all trivial strongly connected components. The procedure is quite simple: all vertices that have an in degree or out degree of zero (excluding self-loops) are removed. This can be performed iteratively or recursively as well, as removing a vertex will change the effective degrees of its neighbors. In this paper, we call performing a single iteration of trimming to be *simple* trimming and performing trimming iteratively as *complete* trimming. This procedure is very effective in improving the performance of the FW-BW algorithm, but can be beneficial to use before running other algorithms, as well.

3) Coloring: The coloring algorithm for SCC decomposition is given in Algorithm 2. This algorithm is similar to FW-BW in that it proceeds in forward and backwards passes. However, the approach is also quite different, as it generates multiple pivots, or more commonly termed, *roots*, during the forward phase and only looks at a subset of G for each pivot on the backwards phase.

Given a graph G, the algorithm starts by assigning a set of unique numeric *colors* to all the vertices, most easily as the vertex identifiers v_{id} for all $v \in G$. These colors are then propagated outwardly from each vertex in the graph. If a vertex v has any neighbors $u \in N(v)$, where N is the neighbor list, with a color lower than it's own, the neighbors' colors are updated to that of the vertex. This process continues until no more vertices change their color.

Algorithm 2 Coloring Algorithm

```
1:  while G ≠ ∅ do
2:      initialize colors(v_id) = v_id
3:      while at least one vertex has changed colors do
4:          for all v ∈ G do
5:              for all u ∈ N(v) do
6:                  if colors(v) > colors(u) then
7:                      colors(u) ← colors(v)
8:      for all unique c ∈ colors do
9:          SCC(c_v) ← PRED(G(c_v), c)
10:         G ← (G \ SCC(c_v))
```

We have now effectively partitioned the graph into distinct sets with separate colors c. As we started with v_{id} as our colors, for each distinct c, there is a unique vertex c_v with that identifier. We consider c_v as the root of a new SCC, $SCC(c_v)$. The SCC is then all vertices that can be reached backward from c_v that are also colored with the same color c. We remove $SCC(c_v)$ from G, find the rest of the SCCs for all c in the current iteration, and then proceed to the next iteration to continue until G is empty.

Parallelizing this algorithm is trivial, as both the forward coloring step across all v and the backward SCC step across all c_v can be parallelized quite easilly.

4) Other Parallel SCC Approaches: There has been other more recent work aimed at further improving upon the preceding algorithms, as well as developing newer algorithms to further decompose the graph and/or improve performance and scalability.

One example is the OBF algorithm of Barnat et al. [8], which, like coloring, aims at every iteration to further decompose the graph into multiple distinct partitions each containing a single SCCs. The OBF decomposition step can be performed much quicker than coloring, however, it does not necessary result in as many partitions. Barnat et al. [9] further implemented the OBF algorithm as well as FW-BW and coloring on the Nvidia CUDA platform, demonstrating considerable speedup over equivalent CPU implementations.

More recently, Hong et al. [10], [11] demonstrated several improvements to the FW-BW algorithm and trimming procedure through expanding trimming to find both 1-vertex and 2-vertex SCCs, further decomposing the graph after the first SCC is found by partitioning based on weakly connected components, and implementing a dual-level task-based queue for the recursive step of FW-BW to improve times and reduce overhead for the task-based parallelism.

B. Connected and Weakly Connected Components

The approaches towards determining connected components and weakly connected components in graphs are similar. There are two primary parallel methods, using techniques similar to those described in the preceeding sections. Firstly, a parallel BFS can be used for connected components. Any vertices reachable by the BFS traversal will be in the same component. We continue selecting new unvisited vertices as BFS roots until all vertices have been visited and all connected components identified. The procedure is the same for weakly connected components, but it is obviously required to examine both in and out edges.

We can use a coloring approach. Each vertex is initialized with a unique color, and the maximal colors are propogated throughout the network. Once the colors reach a stable point, all vertices contained in each discrete component will have the same color.

C. Biconnected Components

The optimal linear time sequential algorithm for determining the biconnected components of a graph is based on DFS and was originally proposed by Hopcroft and Tarjan [12]. A parallel algorithm was later developed by Tarjan and Vishkin [13] which is based on computing a spanning tree followed by a Eulerian tour, determining low and high values for each vertex based on preorder numbering, which can then be used to create an auxilliary graph. The connected components of the auxilliary graph form the BCCs of the original graph. Several improvements have since been made to the original Tarjan and Vishkin parallel algorithm to reduce work and improve parallelism [14], [15].

III. Multistep Method For SCC Decomposition

We will now introduce our algorithm for graph SCC decomposition, the Multistep method. The reason for this name comes from the fact that it is a combination of some of the previously described parallel algorithms stepped through in a certain order. This section will give our justifications for developing the algorithm in this way, as well as provide detail into our algorithm's specifics. Additionally, we will discribe how some of the techniques created during development of the Multistep method can also be applied towards determining the connected, weakly conencted, and biconnected components of a graph.

A. Observations

The FW-BW algorithm can be quite efficient if a graph has relatively small number of large and equally-sized SCCs, as the leftover partitions in each step would on average result in similar amounts of parallel work. The FW and BW searches could also be efficiently parallelized in this instance.

However, the structure of most real world graphs is very different. From observations, most real-world graphs have one giant SCC containing a large fraction of the total vertices, and many many small SCCs and often disconnected SCCs that remain once the large SCC is removed [1]. Running a naïve implementation of FW-BW would result in a large work imbalance after the initial SCC is removed, where the partitioning of the leftover partitions would be heavily dominated by the remainder set. Additionally, using a naïve task-based parallelism model would add considerable overhead as each new task might only be finding an SCC of a few vertices in size before completing. As we will show, even implementations that use a smarter tasking model [10], [11] can still suffer when the graph gets large enough. In general, the size of the recursive tree and therefore the overall runtime of the FW-BW algorithm is dominated by the total number of SCCs that are in the initial graph.

Conversely, the coloring algorithm is quite efficient when the graph contains a large number of small and disconnected components, as the runtime of each step is dominated by the diameter of the largest connected component in the graph, or the number of steps needed to do the full coloring. This also leads to very poor performance on real-world graphs, as the time for each iteration can be very large while the largest SCCs remain, and there is no guarantee that these SCCs will be removed in any of the first few iterations.

It is also important to note that below a certain threshold of the number of vertices in a graph, the general parallel overhead inherent in any implementation results in worse performance when compared to simply using Tarjan's or Kosaraju's serial algorithm.

B. Description of Method

Based on the above observations, we have developed the Multistep method. This method aims at maximizing the advantages and minimizing the drawbacks of trimming, FW-BW, coloring, and a serial algorithm by applying them in sequence to decompose large real-world graphs into their strongly-connected components.

An overview of the algorithm is given in Algorithm 3. here are four primary phases of the algorithm. The first phase is trimming. We choose to do only a single iteration of trimming, as experiments have shown that the vertices trimmed in the second or subsequent iterations can be more efficiently handled in the coloring or serial phases.

Algorithm 3 Multistep Algorithm

```
1: procedure MULTISTEP(G)
2:     Trim_simple(G)
3:     v = max(v_dg), where v_dg = d_in(v) * d_out(v)
4:     FWBW_SCC(G, v)
5:     while NumVerts(G) > V_cutoff do
6:         ColoringStep(G)
7:     Tarjan(G)
```

In the second phase, we select an initial pivot vertex as the vertex in the graph that has the largest product of its

in degree and out degree. This is an attempt to increase the chances that our initial pivot is contained within the largest SCC. Although there is no guarantee to ensure that this will be the case, in practice with real world graphs it acts as a very good heuristic.

With the chosen pivot we do one iteration of a modified FW-BW algorithm. Our changes to the FW-BW iteration avoid considerable work during this phase by not computing the three sets D, P and R and leaving all vertices that are not part of the one SCC we computed to the next step. Since we do not care about explicitly partitioning the sets, we can also avoid fully exploring the graph on the backward search. During the backward search, if we encounter a vertex that was not encountered on the forward search we can safely ignore it and avoid adding it to our next level queue.

For a simple proof as to why this will work, assume by contradiction that a predecessor vertex p_i we find during the backward phase that was not marked during the forward phase has a predecessor p_j that was marked during the forward phase. This cannot happen as if the predecessor was previously marked, then the original predecessor p_i, being a descendant of p_j, would therefore have been marked as well. Since we know that, in order to be in the SCC, a vertex must be marked as both a descendant and predecessor, then we can safely ignore all of these p_i. For certain graphs, this can result in a considerably shorted search during the backward phase.

At the end of the FW-BW step, we simply take the remaining vertices and pass them all off to coloring. We run coloring until the number of vertices remaining crosses below a certain threshold determined experimentally, and then pass off the still remaining vertices to the final step, which is just Tarjan's serial algorithm.

IV. Implementation Details

This section will provide a bit more detail into some of the implementation specifics. All code was written in C++ using OpenMP for shared-memory parallelization. We achieve most of our performance by avoiding atomic or locking operations whenever possible through thread-owned queues and mitigation of race conditions and by utilizing various techniques to avoid work.

A. Trim Step

We consider two different approaches for trimming. For simple trimming, we only need to look at the degree values as initially set when the graph was created. Therefore, we just need to do a single pass through all vertices, retrieve their in/out degrees, and flip their valid boolean if either is zero.

Complete trimming is a bit more complex. To greatly speed up processing during complete trimming, we create current and future queues and an additional boolean array of values to signify for each vertex if they are currently placed in the future queue. We place all vertices in the current queue to begin with.We then determine the effective in and out degrees for all vertices contained in the current queue. If a vertex has an effective in or out degree of zero, then it is marked as no longer valid. Additionally, any valid vertices that the removed vertex was pointing to or had pointing at it are then placed in the future queue and marked as such. After the current queue is empty, the queues are swapped with the marks reset.

This process is repeated for as many iterations as necessary. The queues are used to avoid having to look through all vertices at each iteration, as it has been observed that the long *tendrils* [16] of vertices in lots of real world graphs tend to result in long tails of iterations where only a few vertices are removed at a time. The marking is done to prevent a vertex from being placed in the future queue multiple times. To avoid the synchronization overhead that would be required with a parallel queue, we maintain separate queues for each thread and combine them into the next level queue at the end of each step.

Although complete trimming is easily paralellizable and can be quite fast, with the queues and marking being done very similarly to how we will soon describe our BFS and coloring, we find simple trimming to generally perform better overall. It is observed that a single iteration will remove the vast majority of vertices than *can* be removed, it does not require the explicit calculation or tracking of changing degrees at each iteration, and does not need queues or other such structures to maintain. Simply passing off the vertices not trimmed in the first iteration to be handled by coloring or the serial algorithm gave us superior performance for all tested graphs.

B. Breadth-First Search

An overview of the BFS used in our Multistep approach is given in Algorithm 4. We will now describe some of the optimizations and design choices we implemented.

A typical BFS optimization is to use a bitmap of length n to signify whether or not a vertex has already been visited, and to avoid further exploring the vertex if it has been. A bitmap is able to fit completely in the last level cache of modern server-grade CPUs for graphs of up to tens of millions of vertices. It is assumed that by staying in cache, a quick boolean check is possible and accesses to main memory are minimized.

However, our experimentation has demonstrated that a boolean array actually outperforms a bitmap for our test environment. The likely reason for this is two-fold. Firstly, in order to calculate the address for a specific bit, at least an integer division, remainder operation, and bit shift is required. Additionally, since the CPU on our testbed only guarantees atomic read/writes starting at the byte level [17, s. 8.1.1], either explicit atomic operations are needed or a more complex bit read/write function is required [18]. By using a

Algorithm 4 Multistep BFS

```
1:  queue ← p
2:  do_hybrid ← false
3:  while queue ≠ ∅ do
4:      if hybrid = false then
5:          for all v ∈ queue do in parallel
6:              for n ∈ N_out(v) do
7:                  if visited(n) = false then
8:                      visited(n) ← true
9:                      thread_queue ← n
10:         end for
11:     else
12:         for all v ∈ G do in parallel
13:             if visited(v) = false then
14:                 for all n ∈ N_in(v) do
15:                     if visited(n) = true then
16:                         visited(v) ← true
17:                         thread_queue ← v
18:                         break
19:         end for
20:     Synchronize
21:     Single thread does
22:         hybrid ← EvaluateHybridSwitch()
23:         if hybrid = false then
24:             for all thread_queue do
25:                 for all v ∈ thread_queue do
26:                     queue ← v
27:     end single thread
28:     Synchronize
```

byte-addressed boolean array and avoiding explicit locks, we see considerably faster runtimes. Although avoiding explicit locks might result in extra work (two threads see a vertex as unexplored, set the same boolean to the same value, and put the vertex in the next level queue twice), experimentation has shown this to be of minimal concern.

As mentioned previously, we avoid additional locks and atomic operations by giving each threads its own next level queue. At the end of each level, the vertices from each thread queue are collected and placed in the frontier for the next level by a single thread. Although this can be performed in parallel by computing offsets based on the length of each thread's queue, experimentation has shown it to be very fast in serial, taking on the order of milliseconds even for millions of nodes. This step can also be skipped when we are going to run the hybrid bottom-up BFS, which will be discussed next. Overall this technique greatly outperformed a shared queue with locks.

A hybrid bottom-up approach to the BFS was recently introduced by Beamer et al. [19]. They noted that at certain levels of a BFS in real world graphs (small world, scale free), it is actually more efficient to simply look in the reverse direction. Instead of all vertices currently on the frontier looking at all their children, all unvisited vertices simply attempt to find their parent on the frontier. It is not even necessary to check if the parent is explicitly on the frontier, but only if the parent has already been marked as visited (the child would have already been discovered if the parent's level is one or more previous to that of the current frontier).

Beamer et al. found that this approach will vastly decrease the total number of edge examinations needed during the BFS, improving search times by over $3\times$. Upon implementing their approach as described with parameters $(\alpha = 15, \beta = 25)$, we noticed considerable speedup as well. A different design choice we had to implement was to maintain the thread queues while we are currently running the bottom-up hybrid as opposed to explicitly rebuilding the queue from scratch when we switch the hybrid off. This is due to the fact that we do not maintain any explicit BFS tree as we only require the visited array to determine the SCC, so we have no ability to track BFS level on a per-vertex basis.

A final optimization we investigated but did not include in our final version was a per-socket graph partitioning and exploration scheme similar to the ones described in Agarwal et al. [20] and Chhugani et al. [18]. Although these partitioning approaches improved parallel scaling, it was only in a limited number of instances that actual runtimes improved due to the additional overhead. A wide variety of parameters and optimizations were explored, including reverting to actual bitmaps, partitioning based on number of vertices and number of edges, and even permuting the input graphs to improve work balance, among others. However, no considerable and definitive improvements were ever noted over simply using our BFS with the bottom-up hybrid, so these approaches was abandoned.

C. Coloring

The vertex coloring step of our coloring algorithm is implemented quite similarly to our BFS and complete trimming algorithm, and is given in Algorithm 5. Initially, all still valid vertices are assigned a color as their vertex id v_{id}, or index, in our graph structure. All valid vertices are then placed into the queue. For all vertices in parallel that are contained in the queue, we check to see if they have a higher color than their children. If they do, the color is passed to the child, and both the parent and child are placed in the thread's next level queue, and globally marked as such for all threads to see.

We also place the parent in the queue to, once again, avoid explicit locks. It is possible and very likely that two parents will have higher colors than a shared child, creating a race condition. Both parents will once again examine their children on the next iteration to make sure that either the color that was given by them or a higher one has been placed. Additionally, since only a higher color can be assigned, we can ignore the race condition created if a parent has their own color overwritten before they assign their previous one to the child. It was also tried to avoid locks instead by going bottom-up and having children look at their parents' and own color and take the largest, avoiding

Algorithm 5 Coloring

```
1: for all v ∈ G do
2:     color(v) ← v_id
3:     queue ← v
4:     in_next_queue(v) ← false
5: while queue ≠ ∅ do
6:     for all v ∈ queue do in parallel
7:         for n ∈ N_out(v) do
8:             if color(v) > color(n) then
9:                 color(n) ← color(v)
10:                if in_next_queue(n) = false then
11:                    in_next_queue(n) = true
12:                    thread_queue ← n
13:        if any n changed color then
14:            if in_next_queue(v) = false then
15:                in_next_queue(v) = true
16:                thread_queue ← v
17:    end for
18:    Synchronize
19:    Single thread does
20:        for all thread_queue do
21:            for all v ∈ thread_queue do
22:                queue ← v
23:                in_next_queue(v) ← false
24:    end single thread
25:    Synchronize
```

the race condition entirely. However, this is much slower in practice, because either all vertices need to be examined at each iteration, or the out vertices of the child need to be examined to create the queue, effectively doubling the amount of memory transfer needed for each iteration.

Our parallel SCC finding on the reverse step is fairly standard, as it is a trivial algorithm to parallelize. We simply determine the root vertices by finding all unique colors in the graph, and then run a serial DFS on the transverse graph from each root, only looking at vertices with the same color as the root. We use a DFS here, since there isn't further room for parallelism, and experimentation has shown our serial DFS to be faster than our serial BFS.

D. Serial Step

We implement a simple and efficient recursive Tarjan's for our serial algorithm. We chose Tarjan's as our serial algorithm over Kosaraju's, based mainly on superior experimental runtimes. Although Kosaraju's can benefit from parallelization during the backwards step, the benefit when the graph has a small number of (usually) disconnected SCCs, is quite small if not negative.

We experimentally determined that a cutoff of about 100,000 remaining vertices is a relatively good heuristic for switching to the serial algorithm, although this is hardware specific. Some graphs benefit from running coloring all the way to completion, while some others would benefit more from switching to serial sooner. However, determining this cutoff without prior knowledge of the graph is quite difficult (possibly some calculation based on the number of steps needed to fully color the graph), and the difference is usually close to negligible.

E. Connected Components and Weakly Connected Components

Our Multistep method can be easily implemented for determining connected and weakly connected components, as well. For these procedure, Multistep simply uses the middle two steps by initially determining the massive (weakly) connected component through parallel BFS and then subsequently performing coloring on the vertices remaining after the massive component has been removed.

F. Biconnected Components

We now introduce a novel algorithm for identifying *articulation vertices* for biconnected component decomposition that can utilize the previously described BFS techniques. This algorithm relies on an initial BFS traversal which creates a BFS tree. An articulation vertex can be identified in the BFS tree by the fact that it has at least a single child vertex which does not have a path to any other vertex on the same BFS level as the articulation vertex that does not pass through the articulation vertex. This indicates that the child vertex is in a separate biconnected component. A simple proof is as follows: if there was some path from the child vertex to another vertex on the same level as its parent, this other vertex and the parent would have to share one common ancestor at a higher level up the BFS tree, which would by definition imply that all edges connecting these vertices are in the same biconnected component.

Algorithm 6 BCC-BFS

```
1: parents, levels ← BFS(G)
2: for all v ∈ G do
3:     for all u ∈ N(v) where parents[u] = v do
4:         max_level ← BFS(u, (G \ v))
5:         if max_level = levels[u] then
6:             v ← is_articulation
7:             break
```

The new algorithm, given by 6, relies purely on multiple BFS searches. Initially we create our BFS tree, represented as $parents$ and $levels$ arrays, to track the parent and level of each vertex. Our goal is to examine every vertex $v \in G$ to check if it is an articulation vertex. We take every child that v has as is indicated by the BFS tree, and run a new BFS from it on $G \setminus v$. If we are not able to identify any vertex during that BFS search which is on the same level as v, then we can mark v as an articulation vertex. This algorithm is efficiently parallelized across all $v \in G$. Although the vast number of BFS searches on the inner loops may seem like a lot of work, it is minimized by the fact that only a minority of vertices actually have any children in the BFS tree for real world graphs. Additionally, ruling out a vertex which

does is quite fast in a high density graph, since a vertex with a higher level is typically encountered after only one or two BFS iterations.

To explicitly check whether or not a root of the BFS tree is an articulation vertex, we need to examine whether or not one of its children can reach all of the others. This can be a costly procedure. However, it is also easily mitigated. For almost all real world graphs there are vertices with a degree of one. A vertex who is the sole neighbor one of these vertices is then easily identified as an articulation point. All we then need to do is begin our BFS traversal from a known articulation point so we are not required to explicitly have to check it. Another option is simply to rerun a new BFS from scratch using new roots and only check our original roots for being articulation points. Because the initial BFS search is the fastest part of the procedure, this can be a valid option as well.

If we wish to label the edges for each BCC, we would want to start checking for articulation vertices at the lowest levels of the BFS tree and work from the bottom up. We track all unlabeled edges encountered when doing an inner BFS, and label these edges as composing a BCC when we find an articulation point (note that we can't use the hybrid-BFS technique during this inner search since it won't observe all edges). Since we are working from the bottom up, there is no risk of encountering an unlabeled BCC and marking two discrete BCCs as the same one. This also avoids having to do additional work to determine if a BFS root is an articulation vertex, since all remaining unlabeled edges encountered on a BFS from any of its children will just form the remaining BCC(s).

V. Experimental Setup

Experiments were performed on a 2 socket machine with 64GB RAM and Intel(R) Xeon(R) E5-2670 CPUs at 2.60GHz (Sandy Bridge) each having 20MB last level cache (*Compton*). Compton was running RHEL 6.1 and compilation was done with `icc` version 13.1.2. The `-O3` optimization parameter was used with the `-fopenmp` flag. Environmental variable `KMP_AFFINITY` was used to control thread locality when needed.

For comparison to other work, we also run SCC code provided by Hong et al. [10], [11] and CC code (Ligra) released by Shun and Blelloch [21]. We used the same compilation procedures and runtime environment when possible, with the exception of using Cilk++ parallelization support with Ligra instead of OpenMP. This was observed to be faster in practice.

Several large real world graphs were used in the course of this work. They are listed in Table I. These graphs were retrieved from a number of sources, namely the SNAP Database [22], the Koblenz Network Collection [23], the DIMACS 10th implementation challenge [24], and the University of Florida Sparse Matrix Collection [25]. The R-MAT [26] and $G(n, p)$ networks were generated with the GTGraph [27] suite using the default parameters.

Network	n	m	d_{avg}	d_{max}	Dia.	# (S)CCs	max (S)CC
Friendster	66M	1.8B	53	5.2K	34	70	66M
Orkut	3.1M	117M	76	33K	11	1	3.1M
Kron_21	1.5M	91M	118	213K	8	94	1.5M
Cube	2.1M	62M	56	69	157	47K	2.1M
Twitter	53MM	2.0B	37	780k	19	12M	41M
Italy Web	41M	1.2B	28	10K	830	30M	6.8M
WikiLinks	26M	0.6B	23	39K	170	6.6M	19M
GNP_1	10M	200M	20	49	7	1	10M
GNP_10	10M	200M	20	49	7	10	5.0M
LiveJournal	4.8M	69M	14	20K	18	970K	3.8M
RDF_data	1.9M	130M	70	10K	7	1.9M	1
RDF_linkedct	15M	34M	2.3	72K	13	15M	1
XyceTest	1.9M	8.3M	4.2	246	93	400K	1.5M
R-MAT_20	560K	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

Table I: Network sizes and parameters for all networks.

The first four listed graphs are undirected while the rest are directed. Friendster, LiveJournal, Orkut, and Twitter are social networks [28]–[30]. Italy Web is a web crawl of the .it domain [31]. WikiLinks is the cross-link network between articles on Wikipedia [32]. XyceTest is a Sandia National Labs electrical simulation network and Cube is 3D coupled consolidation problem of a cube discretized with tetrahedral finite elements. R-MAT_20/22/24 are R-MAT graphs of scale 20, 22, and 24, respectively. The Kron_21 graph is a scale 21 graph created from the Kronecker generator of the Graph500 benchmark. Finally, $G(n, p)$ is an Erdős-Rényi random graph.

These graphs were selected to represent a wide mix of graph size and structure. The number of SCCs/CCs and max SCC/CC both play a large role in the general performance of decomposition algorithms, while the average degree and graph diameter can have a large effect on the BFS runtimes that is necessarily used for these algorithms.

VI. Experimental Results

In this section, we are going to compare our Multistep SCC algorithm runtimes and scaling to our implementations of baseline FW-BW and Coloring algorithms, as well as Hong et al.'s FW-BW algorithm. Further, we will compare our Multistep connected components algorithm to baseline coloring and the coloring that was implemented in the Ligra graph processing framework. We will then finally compare our weakly connected components algorithm to our coloring approach and our biconnected algorithm to the optimal serial algorithm.Additionally, we are going to give justification for the algorithmic choices we have made and their influences on performance for different graph structures.

A. Strongly Connected Component Decomposition

Figure 1 gives a comparison for absolute runtimes of 16 cores of our Compton machine for baseline Coloring and FW-BW with complete trimming, as well as Multistep with simple trimming and Hong et al.ś *Method 2* on several directed graphs. Both Multistep and Hong et al. show

considerable improvements over the baseline approaches. The performance of the baseline approaches is also most dependent on graph structure. The algorithms with a large proportion of the their vertices in the massive SCC, such as the $G(n, p)$, R-MAT, and Xyce graphs, show very poor performance with coloring due to the long time needed to fully propagate the colors. Further, networks with a large absolute number of SCCs show poor performance with FW-BW, due to the recursive and tasking overhead.

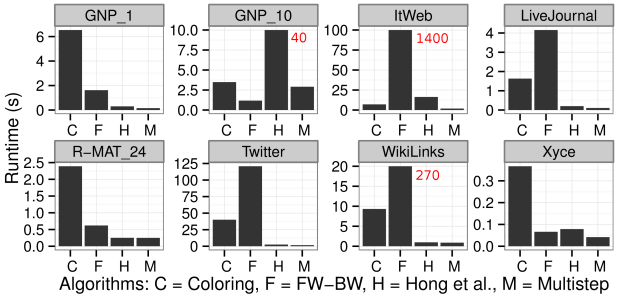


Figure 1: Comparison of Multistep with naïve FW-BW implementation, Coloring, and Hong et al.

Although Hong et al. attempts to minimize the impact of the recursive and tasking overhead with their partitioning step based on WCCs and smart tasking queue, on graphs with a very high number of small but non-trivial SCCs such as ItWeb, the overhead inherent in the FW-BW algorithm can still dominate the runtime. It can also be noted that our coloring step will at each iteration partition the graph into *at least* as many discrete partitions that a WCC decomposition will.

Figure 2 gives parallel scaling of both Multistep and Hong et al. For 1, 2, 4, 8, and 16 cores relative to a serial Tarjan's implementation. Both Multistep and Hong et al. demonstrates good scaling on most test instances. As mentioned before, the Hong et al. runtime on ItWeb is greatly affected by the number of SCCs. Additionally, on ItWeb, there are long strings of trivial and non-trivial SCCs, which results in a relatively long time spent in the multiple trimming steps that are in Hong et al.'s approach, as well as a long time in the WCC decomposition step.

Figure **??** gives the breakdown for each stage of multistep as the proportion of total runtime. As can be observed, the runtime proportion for the FW-BW and Coloring steps is mostly dependent on graph structure, with coloring taking a larger proportion when the graph is large with a high diameter.

Figure **??** also gives further justification for our choice of doing simple trimming versus complete trimming. In general, the extra time spent doing iterative trimming does not decrease the FW-BW or Coloring steps enough to make fully trimming a graph time-effective. As is shown

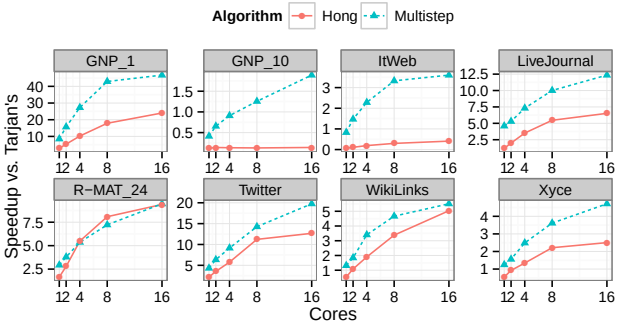


Figure 2: Parallel scaling of Multistep and Hong et al. relative to Tarjan's serial algorithm.

on LiveJournal, but was noted on other graphs as well, doing no trimming at all can end up being faster than fully trimming the graph with our Multistep approach. While running Multistep across a wide variety of graphs, fully trimming the graph never improved runtimes versus only doing a single iteration.

Figure 4 gives approximate weak scaling for three R-MAT test graphs (R-MAT_20/22/24). The test graphs' number of vertices, edges, number of SCCs, and size of largest SCC all increase by approximately a factor of 4 (n=560K/2.1M/7.7M, m=8.4M/34M/130M, numSCCs=210K/790K/3.0M, maxSCC=360K/1.3M/4.7M). From Figure 4, we can see that Multistep scales better than simply FW-BW or coloring.

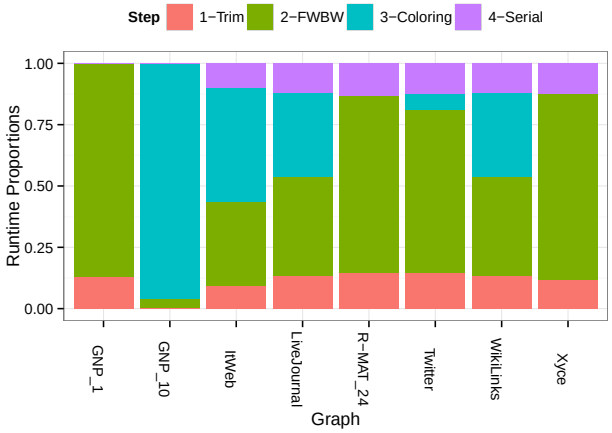
B. Connected and Weakly Connected Component Decomposition

We next compare our approach to Ligra for the problem of determining connected components. Ligra implements a parallel coloring-based algorithm. We show scaling relative to a serial DFS approach. From Figure 5, we observe that Multistep greatly outperforms the other approaches on all tested graphs. On the larger graphs of Friendster and Orkut, our coloring approach outperforms Ligra as well.

Figure 6 gives the speedup of the Multistep method and our coloring approach for determining the weakly connected components of several graphs. We give speedup relative to the serial DFS approach. Once again we observe good scaling of Multistep relative to the serial code.

C. Biconnected Component Decomposition

Figure 7 gives the parallel scaling of our new BFS-based BCC decomposition algorithm compared to the serial Hopcroft and Tarjan DFS-based algorithm. On the three largest non-fully biconnected graphs, we achieve up to $8\times$ speedup good scaling across 16 cores. However, our approach does not scale well with the fully-biconnected Cube graph. This is likely due to its regular structure, which



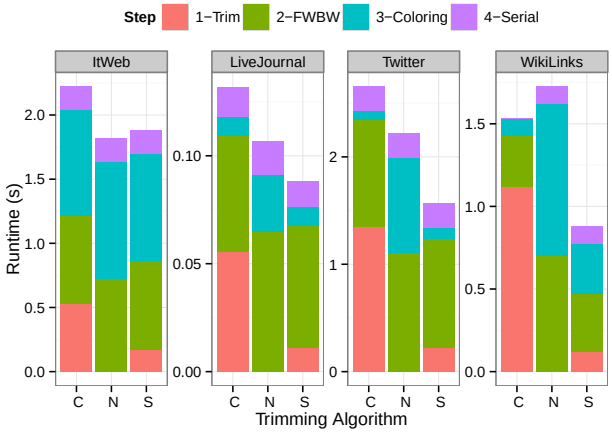


Figure 3: Proportion of time spent in each state of the Multistep algorithm as well as a comparison between trimming procedures for WikiLinks and LiveJournal.

minimized the inner-loop BFS time to a single iteration on most instances. Because of the lack of overall work necessary for this graph, it outperforms the serial algorithm algorithm on a single core while overhead dominates its parallel runtime.

D. Breadth-First Search

We finally analyze our breadth-first search implementation across all tested graphs. Using the standard billions of edges traversed per second (GTEPS) metric of graph traversal speed, we note in Figure **??** that our code achieves several GTEPS across a number of graphs. We also note the dependence of graph traversal speed relative to the graphs average degree.

VII. Conclusion

This paper introduced the Multistep Method, a combination of previous parallel algorithm for the task of strongly

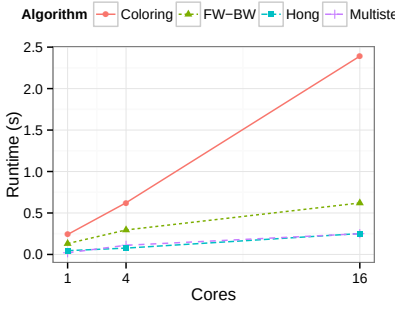


Figure 4: Approximate weak scaling of Multistep compared to Coloring and naïve FW-BW.

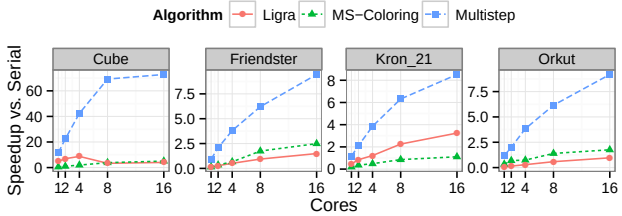


Figure 5: Parallel scaling of CC Multistep, Coloring, and Ligra relative to the serial DFS approach.

connected component decomposition of large graphs. We demonstrate speedup over the current state of the art, while showing how our techniques can be applied to a broad class of graph-based algorithms.

Acknowledgment

We thank Erik Boman, Karen Devine and Bruce Hendrickson for encouraging us to focus on this problem. Sandia National Laboratories is a multi-program laboratory

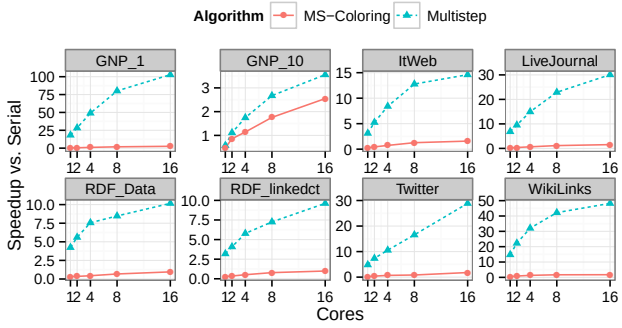


Figure 6: Comparison of WCC Multistep with Coloring relative to the serial DFS approach.

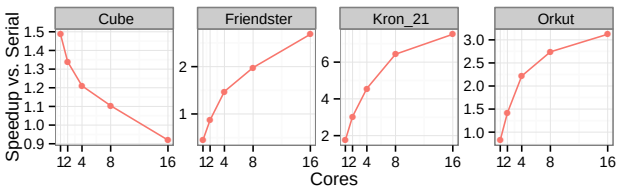


Figure 7: Parallel scaling of BCC-BFS relative to the serial Hopcroft and Tarjan algorithm.

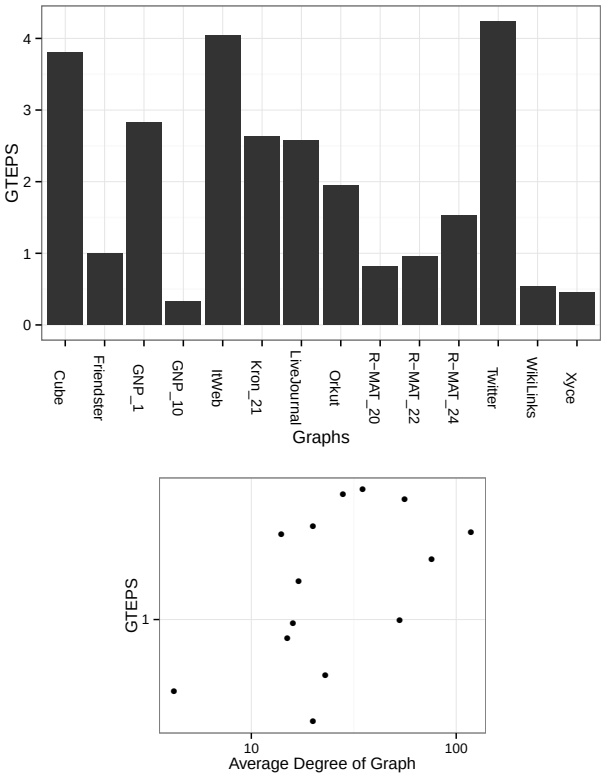


Figure 8: Runtime GTEPS for each tested graph as well as the relationship between GTEPS and average graph degree.

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 partially supported by the DOE Office of Science through the FASTMath SciDAC Institute.

References

[1] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 2007, pp. 29–42.

[2] W. McLendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger, "Finding strongly connected components in distributed graphs," *Journal of Parallel and Distributed Computing*, vol. 65, no. 8, pp. 901–910, 2005.

[3] A. Pothen and C.-J. Fan, "Computing the block triangular form of a sparse matrix," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 4, pp. 303–324, 1990.

[4] H. K. Thornquist, E. R. Keiter, R. J. Hoekstra, D. M. Day, and E. G. Boman, "A parallel preconditioning strategy for efficient transistor-level circuit simulation," in *Computer-Aided Design-Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*. IEEE, 2009, pp. 410–417.

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

[6] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. Addison-Wesley, 1983.

[7] L. K. Fleischer, B. Hendrickson, and A. Pinar, "On identifying strongly connected components in parallel," *Lecture Notes in Computer Science*, vol. 1800, pp. 505–512, 2000.

[8] J. Barnat and P. Moravec, "Parallel algorithms for finding sccs in implicitly given graphs," *Formal Methods: Applications and Technology*, vol. 4346, pp. 316–330, 2006.

[9] 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.

[10] 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.

[11] ——, "On fast parallel detection of strongly connected components (scc) in small-world graphs," in *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013, p. to appear.

[12] J. Hopcroft and R. Tarjan, "Efficient algorithms for graph manipulation," *Communcations of the ACM*, vol. 16, no. 6, pp. 374–378, 1973.

[13] R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM Journal on Computing*, vol. 14, no. 4, pp. 862–874, 1985.

[14] G. Cong and D. A. Bader, "An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (smps)," in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE, 2005, pp. 45b–45b.

[15] J. A. Edwards and U. Vishkin, "Better speedups using simpler parallel programming for graph connectivity and biconnectivity," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 2012, pp. 103–114.

[16] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," *Computer networks*, vol. 33, no. 1, pp. 309–320, 2000.

[17] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual, System Programming Guide, Part 1*. Intel Press, 2011, vol. 3A.

[18] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency," in *Supercomputing*, 2012.

[19] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Supercomputing*, 2012.

[20] V. Argarwal, F. P. D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Supercomputing*, 2010.

[21] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

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

[23] J. Kunegis, "KONECT - the koblenz network collection," konect.uni-koblenz.de, last accessed 31 July 2013.

[24] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "Graph partitioning and graph clustering, 10th dimacs implementation challenge workshop," *Contemporary Mathematics*, vol. 588, 2013.

[25] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.

[26] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SDM*, 2004.

[27] K. Madduri and D. A. Bader, "GTgraph: A suite of synthetic graph generators," http://www.cse.psu.edu/~madduri/software/GTgraph/, last accessed 31 July 2013.

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

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

[30] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW '10: Proceedings of the 19th international conference on World wide web*. New York, NY, USA: ACM, 2010, pp. 591–600.

[31] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[32] "Wikipedia links, english network dataset – KONECT," Oct. 2013. [Online]. Available: http://konect.uni-koblenz.de/networks/wikipedia_link_en