Efficient, Reachability-based, Parallel Algorithms for Finding Strongly Connected Components*

[Technical Report]

Daniel Tomkins, Timmie Smith, Nancy M. Amato, Lawrence Rauchwerger
Department of Computer Science and Engineering
Texas A&M University
College Station, TX 77845-3112

ABSTRACT

Large, complex graphs are increasingly used to represent unstructured data in scientific applications. Applications such as discrete ordinates methods for radiation transport require these graphs to be directed with all cycles eliminated, where cycles are identified by finding the graph.s strongly connected components (SCCs). Deterministic parallel algorithms identify SCCs in polylogarithmic time, but the work of the algorithms on sparse graphs exceeds the linear sequential complexity bound. Randomized algorithms based on reachability — the ability to get from one vertex to another along a directed path — are generally favorable for sparse graphs, but either do not perform well on all graphs or introduce substantial overheads. This paper introduces two new algorithms, SCCMulti and SCCMulti-PriorityAware, which perform well on all graphs without significant overhead. We also provide a characterization that compares reachability-based SCC algorithms. Experimental results demonstrate scalability for both algorithms to 393,216 of cores on several types of graphs.

Categories and Subject Descriptors: Theory of Computation [Design and Analysis of Algorithms]: Parallel Algorithms

Keywords: Strongly Connected Components, Randomized Algorithms, Parallel Graph Algorithms, Parallel Complexity Theory

1. INTRODUCTION

Graphs are used in many fields, including particle transport studies [1], social network analysis [7], and motion planning [13]. We can efficiently analyze graph properties using sequential techniques, but many graphs, especially those used in scientific and social network applications, exceed the memory available to a single processor.

Unfortunately, directed graphs are generally difficult to analyze in parallel; many sequential techniques are inherently non-parallelizable [15]. Indeed, parallel graph algorithms have been studied for several decades, but for many problems we are still unable to reach the theoretical bounds provided by sequential algorithms.

This paper considers one such problem: finding strongly connected components (SCCs) in a directed graph. An SCC is a maximal subgraph of a directed graph such that there is a path from every SCC node to every other SCC node. SCCs are used in compiler analysis [2], data mining, and scheduling [6]. They can be used to locate cycles and collapse cyclic graphs to their component directed, acyclic graphs [16]. Significantly, particle transport applications perform a traversal of the spatial domain in each direction of particle travel; in order for the traversals to terminate, cycled in the graph representing the spatial domain must be identified and removed [1].

The optimal sequential algorithm for finding SCCs is Tarjan's linear-time algorithm [19], which depends on depth-first search (DFS). Unfortunately DFS is P-complete [15]. To overcome this, parallel SCC algorithm research initially focused on algorithms based on the transitive closure of the graph [8, 9, 3]. However, computing transitive closure for sparse graphs is costly in the total amount of work and memory. Other deterministic algorithms improved the total work, but only for certain subsets of graphs [12, 4].

Due to the cost of the deterministic algorithms, randomized algorithms are of interest [17, 6, 14, 16]. Currently, the best parallel SCC algorithms share a basic technique and structure; we call this family of algorithms *reachability-based SCC* (RB-SCC) *algorithms*, because they use directed paths from a given source node to separate the SCCs. However, each of these RB-SCC algorithms has some drawback that limits its functionality. For instance, the Divide-and-Conquer Strong Components (DCSC) algorithm [6, 14] has a large set of worst case inputs; it performs poorly on graphs that have low average reachability. The MultiPivot algorithm [16] probabilistically guarantees its asymptotic complexity for all inputs, but uses extra computation to choose source nodes for the reachability queries. A third algorithm, NSCC [17], offers DCSC's average case work and MultiPivot's input-independence, but contains a substantial memory overhead and certain stateful requirements on

*This research supported in part by NSF awards CNS-0551685, CCF 0702765, CCF-0833199, CCF-1439145, CCF-1423111, CCF-0830753, IIS-0916053, IIS-0917266, EFRI-1240483, RI-1217991, by NIH NCI R25 CA090301-11, by DOE awards DE-AC02-06CH11357, DE-NA0002376, B575363, by Samsung, IBM, Intel, and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST).

its reachability queries.

In this work, we present two new randomized RB-SCC algorithms, SCCMULTI and SCCMULTI-PRIORITYAWARE (SCCMULTI-PA), that overcome several of the drawbacks of previous RB-SCC algorithms. Like MULTIPIVOT, our algorithms select multiple nodes and compute reachability information about them. Like DCSC, they use the information obtained to remove many edges of the graph, fracturing it into many pieces. As we will show, both SCCMULTI and SCCMULTI-PA obtain the fast speed of DCSC and NSCC, the input independence of MULTIPIVOT and NSCC, and the low memory overhead of DCSC and MULTIPIVOT.

The main contributions of this paper include

- a survey of reachability-based SCC algorithms and a framework to compare them;
- SCCMULTI and SCCMULTI-PA, two new randomized, reachability-based SCC algorithms;
- probabilistic guarantees that SCCMULTI and SCCMULTI-PA are expected to use the resources of $O(\log n)$ reachability queries; and
- experimental results that validate our algorithms' theoretical properties up to 393,216 cores.

2. PRELIMINARIES

Before discussing our new algorithms, we review some graph terminology and related algorithms. We will also survey previous randomized RB-SCC algorithms in Section 3, which will set the stage for our own algorithms, SCCMULTI and SCCMULTI-PA.

2.1 Graph Terminology

DEFINITION 2.1 (DIGRAPH). *A directed graph (or digraph) G is a pair (V, E), where V is a non-empty set of nodes (equivalently, vertices) and $E \subseteq \{(u,t) : u, t \in V\}$ is a set of directed edges, or ordered pairs of elements of V.*

In the context of this paper, we frequently call a digraph a *graph* and a directed edge an *edge*. We also say that vertex $v \in G$ if $v \in V$ and that edge $(u,t) \in G$ if $(u,t) \in E$. As is common in graph theory, we denote $|V|$ by n and $|E|$ by m.

DEFINITION 2.2 (TRANSPOSE GRAPH). *For a graph G, the transpose or reverse graph of $G = (V, E)$ is a graph $G' = (V, E')$ where $(u,t) \in E \Leftrightarrow (t,u) \in E'$. That is, G' is G with all edges reversed.*

DEFINITION 2.3 (PATH). *For a graph G and nodes $u, t \in G$, a directed path (or path) is a sequence of edges from one vertex to another; i.e.,*

$$path(u,t) = ((v_1 = u, v_2), (v_2, v_3), \ldots, (v_{l-1}, v_l = t)),$$

where $(v_i, v_{i+1}) \in G$ for all $i \in [1, l-1]$. Furthermore, $\exists path(u,u) \in G$ for any $u \in G$; this is a "trivial path" of length 0.

Reachability expresses the ability to get from one vertex to another vertex through some path. That is, the reachability relation is the transitive closure of the edge set; a node u reaches a node t iff there is a path from u to t.

DEFINITION 2.4 (PREDECESSORS). *For a graph $G = (V, E)$ and vertex $t \in G$, the predecessors of t is the set*

$$PRED(t) = \{u : \exists path(u,t) \in G\}.$$

DEFINITION 2.5 (SUCCESSORS). *For a graph $G = (V, E)$ and vertex $u \in G$, the successors of u is the set*

$$SUCC(u) = \{t : \exists path(u,t) \in G\}.$$

We say that the predecessors of t *reach* t, and the successors of u are *reached by* u.

DEFINITION 2.6 (SCC). *A strongly connected component (SCC) of a directed graph G is a maximal subgraph $S \subseteq G$ such that*

$$\exists path(u,t)\ \forall u, t \in S.$$

That is, it is a maximal subgraph such that every node has a path to every other node.

The following property of SCCs, presented as Lemma 1 in [6], is important throughout this work:

LEMMA 2.7. *Denote the SCC containing $v \in V$ as SCC_v. Then $SCC_v = SUCC(v) \bigcap PRED(v)$.*

DEFINITION 2.8 (TRANSITIVE CLOSURE). *For a graph G, the transitive closure of $G = (V, E)$ is a graph $TC(G) = (V, E')$ such that $\forall u, t \in G$,*

$$t \in SUCC(u) \Leftrightarrow (u,t) \in E'.$$

That is, the transitive closure has an edge between any two nodes which can reach each other.

2.2 Parallel Graph Algorithms

This section reviews useful classes of parallel graph algorithms. We first discuss algorithms for finding the reachability of a vertex, which is closely related to the SCC property. Then we discuss deterministic, parallel SCC algorithms, which perform prohibitive total work on sparse graphs and lead to the investigation of randomized algorithms.

2.2.1 Parallel Reachability Algorithms

Any algorithm that can take $v \in V$ and return $SUCC(v)$ is called a *reachability algorithm*, and a single call to one of these algorithms is called a *reachability query*. Existing parallel algorithms for this problem include Spencer's parallel breadth-first-search (PBFS) [18, 17] and the algorithm of Ullman and Yannakakis [21].

In this work, we assume that RB-SCC algorithms are provided with a black-box, parallel reachability algorithm, *RQ-fw*(G, v, mk), that takes a graph G, a vertex $v \in G$, and a mark mk and modifies G, marking all nodes in $SUCC(v)$ with mk. That is, $SUCC(v)$ is the set of nodes marked with an mk after a call to *RQ-fw*(G, v, mk). All marks are stored on the vertices, not in any auxiliary structure. To be a black-box algorithm, *RQ-fw*$(\cdot)$ requires that the vertex provide an interface for being marked; this typically involves setting a flag on the vertex, although the marks could be a more complex data structure. We also assume that *RQ-bw*(G, v, mk), which marks all nodes in $PRED(v)$, is provided; this is just *RQ-fw*$(\cdot)$ on the transpose graph.

We assume that the algorithm used for *RQ-fw*(G, v, mk) is a state-of-the-art, parallel reachability algorithm, such as PBFS or Ullman and Yannakakis's algorithm.

2.2.2 Deterministic Parallel SCC Algorithms

The SCC problem is readily seen to be in the class NC. Assume that we are given an ordering on the nodes and the transitive closure of the graph. This allows the development of a naïve $O(\log n)$ time, $O(n^2)$ processor algorithm for finding SCCs: For every $u \in G$, perform a parallel reduction on the transitive closure to find the lowest $t \in G$ which reaches and is reached by u, using this t as the SCC marking for u. This uses $O(\log n)$ time and $O(n^2)$ processors.

Algorithm	Year	Reach. Queries	Memory per Node	Note
NSCC [18]	1991	$\widehat{O}(\log n)$	$O(f(G))$	Must use PBFS [18]; $f(\cdot)$ is sublinear
DCSC [6]	2000	Input Dependent	$O(1)$	$O(n)$ depth on some inputs
MultiPivot [16]	2007	$\widehat{O}(\log^2 n)$	$O(1)$	–
SCCMulti (This Paper)	2015	$\widehat{O}(\log n)$	$\widehat{O}(1)$	–
SCCMulti-PA (This Paper)	2015	$\widehat{O}(\log n)$	$O(1)$	Uses specialized reachability query

Table 1. Summary of Reachability-Based SCC Algorithms; $\widehat{O}(\cdot)$ is defined in Section 3

Most deterministic, parallel SCC algorithms use matrix multiplication to find the transitive closure of the graph [8, 9, 3]. These algorithms solve the problem in $O(\log^k n)$ time with $O(n^{2+\epsilon})$ processors (for specific $k \geq 2$ and $0 < \epsilon < 1$ pairs). Other algorithms, such as Ullman and Yannakakis's reachability algorithm, also calculate the transitive closure, but this takes $O(n^2)$ space to store. While this is a reasonable strategy for dense graphs (m is $\Theta(n^2)$), for sparse graphs this total work far exceeds the linear bound of $O(m)$.

There exist efficient algorithms for planar graphs [12], even attaining $O(n)$ work for large n [4], yet no work optimal deterministic algorithm is known to find SCCs on arbitrary sparse graphs.

Because of the cost associated with these deterministic algorithms, attention turned to randomized algorithms that offer an expectation of solving the problem quickly. These algorithms are described in the next section.

3. REACHABILITY-BASED SCC ALGORITHMS

Many of the best-performing parallel SCC algorithms (Table 1) are quite similar. This paper introduces the *Reachability-Based SCC* (RB-SCC) *Algorithm* framework and shows that these randomized SCC algorithms fit this model.

3.1 RB-SCC Algorithm Framework

The RB-SCC algorithms surveyed here share many features: they select a random pivot or set of pivots, test how these pivots reach the graph, and then use this information to divide the problem into smaller subproblems by dividing the graph into subgraphs. The algorithms differ only in the methods used to select pivots and how the reachability information is used to divide the graph.

Algorithm 1 RB-SCC Algorithm Framework

Input: a digraph G
Output: G with SCCs marked
1: **if** G is empty **then return**
2: select a set of pivot nodes $S \subseteq G$
3: **for all** $v \in S$ **parallel do**
4: //parallel; modifies $w.marks\text{-}fw$ for $w \in SUCC(v)$
5: $RQ\text{-}fw(G, v, mk_v)$
6: //parallel; modifies $w.marks\text{-}bw$ for $w \in PRED(v)$
7: $RQ\text{-}bw(G, v, mk_v)$
8: using marks, partition G into SCCs and other subgraphs
9: in_parallel(repeat operations on partitions of G)

The general framework of an RB-SCC algorithm can be seen in Algorithm 1, and an example of its execution is shown in Figure 1. In line 2, the algorithm selects a set of pivot nodes; for our example, the pivot is labeled v. Then, in the loop beginning in line 3, the algorithm finds the reachability of each pivot (where *marks-fw* is represented by horizontal hatching and *marks-bw* by vertical hatching). Finally, in line 8 the algorithm divides the graph according to the reachability information.

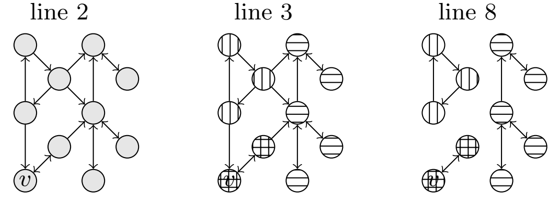


Figure 1. Example execution of an RB-SCC algorithm. $SUCC(v)$ has horizontal hatching, $PRED(v)$ has vertical hatching, and SCC_v has grid hatching.

Each of the algorithms presented in this section implement the general framework differently, primarily in how they select pivots (line 2) and how they use the reachability information to divide G (line 8). These differences lead to differences in performance. For instance, poorly selected pivots can cause imbalanced divisions and inefficient processing of subpartitions, whereas investing too much computation in choosing a good pivot can lead to wasted work.

3.1.1 *Parallel Complexity*

The strength of RB-SCC algorithms derives from the fact that most of their operations — selecting pivots and removing edges — are easy to parallelize. Since these operations are more efficient than a reachability query, we are able to bound the resources used by RB-SCC algorithms in terms of the number of reachability queries they use.

All of these algorithms are Las Vegas algorithms. This means that, while they always produce the correct result, the resource bounds are not deterministic. For clarity, we express probabilistic bounds using the $\widehat{O}(\cdot)$ notation. Bounding a resource by $\widehat{O}(f(n))$ indicates that there exists a positive constant α such that the expected value of the resource is no more than $\alpha f(n)$.

Serial Reachability Queries. Assume that we are given a set of vertices and asked to find the reachability of each one. There are two main approaches to solving this problem in parallel: we can run reachability queries *serially* (waiting for the results of the first before beginning the second), or we can start them in parallel.

Using either of these schemes, the total work will be the same. However, the latter is a better use of parallelism; if the reachability of each vertex is small, then much of the system will be idle while performing any given reachability query. This presents the major challenge for RB-SCC algorithms; waiting for small reachability queries is temporally expensive, whereas starting too many big reachability queries before dividing the graph increases the total work.

3.1.2 Trim *Subroutine*

Since every SCC discovered requires a reachability query, processing graphs made up of many small SCCs can be very expensive. McLendon et. al. [14] observed that determining that a node is a source (i.e., a node with no predecessors)

or sink (i.e., a node with no successors) can be done locally, and that any source or sink node is an SCC of size one. Using this idea, they add a TRIM preprocessing subroutine which uses a single scan; this scan begins from all source or sink nodes, removes them, and then checks their neighbors for the source or sink condition.

Since TRIM can be done with one scan of the graph that uses no more time or work than a reachability query on the entire graph, it affects neither the time nor the work of any of these RB-SCC algorithms — they already require at least one reachability query per step. However, TRIM can provide a significant advantage on graphs containing a large percentage of size one SCCs, as is shown in [14].

3.2 DCSC

The Divide-and-Conquer Strong Components (DCSC) algorithm of Fleischer et al. [6, 14] fits very simply within the general RB-SCC framework: DCSC randomly selects one node as the pivot, groups the nodes into SCC_v, $SUCC(v)$, $PRED(v)$, or $OTHER$ and recurses on all groups but SCC_v.

Algorithm 2 DCSC[6, 14]

1: **if** G is empty **then return**
2: $v \leftarrow random_node(G)$
3: //parallel; $w.marks\text{-}fw \leftarrow mk$ for $w \in SUCC(v)$
4: $RQ\text{-}fw(G, v, mk)$
5: //parallel; $w.marks\text{-}bw \leftarrow mk$ for $w \in PRED(v)$
6: $RQ\text{-}bw(G, v, mk)$
7: **for all** $w \in G$ **parallel do**
8:   **if** $mk = w.marks\text{-}fw$ **and** $mk = w.marks\text{-}bw$ **then**
9:     $SCC_v \leftarrow SCC_v \cup \{w\}$ //partition SCC
10:   **else if** $mk = w.marks\text{-}fw$ **then**
11:     $SUC \leftarrow SUC \cup \{w\}$ //partition subgraph
12:   **else if** $mk = w.marks\text{-}bw$ **then**
13:     $PRE \leftarrow PRE \cup \{w\}$ //partition subgraph
14:   **else**
15:     $OTH \leftarrow OTH \cup \{w\}$ //partition subgraph
16: in_parallel(DCSC(SUC), DCSC(PRE), DCSC(OTH))

If the three recursed groups are approximately the same size, the depth of recursion for DCSC is $\widehat{O}(\log n)$. However, when any one of the groups dominates in size, the performance is significantly affected. As mentioned in [16], DCSC has recursion depth $\Theta(n)$ in the case that SCCs are not connected to each other (or, in fact, whenever the average reachability of any vertex is small). It is worth noting that on meshes, DCSC's depth is $\widehat{O}(\log n)$; this has been its application in scientific experiments [6].

DCSC's poor performance on some graphs is an example of the problem of waiting on small reachability queries. The algorithm always performs one reachability query for each SCC; however, when the reachability queries are performed serially and on a small portion of the graph, there is little opportunity for the algorithm to exploit parallelism.

Efficient Implementation for Small-World Graphs. Small-world graphs tend to have one large SCC containing over 90% of the graph. If this large SCC is removed, the remaining graph is comprised of small SCCs with low average reachability. DCSC is very likely to find the large SCC on the first level of recursion, and the remaining graph presents DCSC's worst case input, so it is not efficient for these graphs.

Hong et al. [11] proposed a version of DCSC for small-world graphs, and their shared-memory implementation showed the first consistent speedup over Tarjan's algorithm for this type of graph. They attained this speedup by dividing the graph into small pieces through two additional mechanisms: they remove all SCCs of size 2 using a modified TRIM step and then divide the graph based on its weakly connected components (WCCs). Since each divided piece has been identified, the modified DCSC algorithm is able to recurse on each piece independently.

Finding the WCCs is extremely useful for DCSC, since on most inputs this avoids the worst case input. However, graphs can be constructed which have low average reachability and still are not quickly broken into separate WCCs. (For an example, see Appendix.) These graphs provide a new worst case for the modified DCSC, so it is still not probabilistically guaranteed to be efficient.

3.3 MultiPivot

Schudy [16] describes MULTIPIVOT, shown in Algorithm 3, which chooses pivots more intelligently than DCSC. To select a pivot, MULTIPIVOT randomly permutes the graph nodes. It then uses a binary-search to find the minimal k such that the first k vertices reach at least half the graph. The pivot is then set as the kth vertex, and MULTIPIVOT partitions the vertices into five sets based on how they are reached by v_k and the first k vertices.

Algorithm 3 MULTIPIVOT [16]

1: **if** G is empty **then return**
2: randomly permute $G.V$
3: find max(k) such that MPCOUNTER(G, $k-1$) $< \frac{|G|}{2}$
4: **for all** $w \in G$ **parallel do**
5:   **if** $mk = w.marks\text{-}fw$ **then**
6:     $A \leftarrow A \cup \{w\}$ //partition subgraph
7: //parallel; $w.marks\text{-}fw \leftarrow mk_k$ for $w \in SUCC(v_k)$
8: $RQ\text{-}fw(G, v_k, mk_k)$
9: **for all** $w \in G$ **parallel do**
10:   **if** $mk_k = w.marks\text{-}fw$ **then**
11:     $B \leftarrow B \cup \{w\}$ //partition subgraph
12: //parallel; $w.marks\text{-}bw \leftarrow mk_k$ for $w \in (PRED(v_k) \cap B)$
13: $RQ\text{-}bw(B, v_k, mk_k)$
14: **for all** $w \in B$ **parallel do**
15:   **if** $mk_k = w.marks\text{-}bw$ **then**
16:     $SCC_{v_k} \leftarrow SCC_{v_k} \cup \{w\}$ //partition SCC
17: in_parallel(MULTIPIVOT($G \setminus A \setminus B$), MULTIPIVOT($A \setminus B$), MULTIPIVOT($B \setminus A \setminus SCC_{v_k}$), MULTIPIVOT($A \cap B \setminus SCC_{v_k}$))

MPCOUNTER(G, k)

1: **for all** $v \in G$ **parallel do**
2:   $v.marks\text{-}fw \leftarrow 0$
3: **for all** $v \in G$ **parallel do**
4:   //parallel; $w.marks\text{-}fw \leftarrow mk$ for $w \in SUCC(v_i), i \le k$
5:   **if** $v.index \le k$ **then** $RQ\text{-}fw(G, v, mk)$
6: **return** the number of nodes with mark mk

Schudy [16] demonstrates that, with high probability, MULTIPIVOT's subgraphs are all smaller than $\frac{n}{2}$ nodes; therefore, the probabilistically guaranteed depth of recursion, regardless of the input graph, is $\widehat{O}(\log n)$. Furthermore, because the algorithm recurses on balanced partitions, the size of the divided subgraphs is inversely proportional to the number of recursive tasks, so the algorithm avoids waiting on queries that do not reach much of the graph.

The primary work at each level is in finding the minimal k that meets the requirements, which takes $O(\log n)$ steps with binary search. Each step of this binary search uses one multi-source reachability query; therefore, MULTIPIVOT performs $O(\log n)$ reachability queries at each level. It has $\widehat{O}(\log n)$ levels, so it uses $\widehat{O}(\log^2 n)$ total reachability queries. While this is worse than DCSC's best case, it is

significantly better than DCSC's worst case; that is, MultiPivot gives up some efficiency for overall consistency of expected resources used.

3.4 NSCC

Spencer's NSCC algorithm [17] is very similar to DCSC, but it includes logic to avoid waiting on small reachability queries. NSCC, shown in Algorithm 4, serially fractures the graph into many pieces using only the work and time of one reachability query. Instead of choosing only one pivot on each level of recursion, NSCC uses a loop to iteratively choose multiple pivots. On each iteration of this loop, NSCC randomly chooses a pivot v and finds $SUCC(v)$, recursing on $SUCC(v) \setminus SCC_v$.

Algorithm 4 NSCC [17]

```
1: ρ ← f(|G|) (where f(·) is sublinear)
2: while |G| > ρ do
3:    for all v ∈ {u : |SUCC(u)| < ρ} do
4:       G ← G \ SUCC(v)
5:    v ← random_node(G)
6:    SCC_v ← SUCC(v) ∩ PRED(v)
7:    in_parallel(NSCC(SUCC(v) \ SCC_v))
8:    G ← G \ SUCC(v)
9: NSCC(G)
```

Spencer showed that, with constant probability, each recursive call operates on no more than a constant fraction of the graph. Therefore, the total depth of recursion is $\widehat{O}(\log n)$. While the total number of reachability queries at each level is not constant, their total time and work is no more than that of one reachability query on the entire graph. Therefore, with high probability, the algorithm uses no more time or work than $\widehat{O}(\log n)$ reachability queries.

While the total work of NSCC is probabilistically guaranteed to be better than either DCSC or MultiPivot, it trades this performance for a significant memory overhead. To avoid waiting on small reachability queries, NSCC requires some assurance that each $SUCC(v)$ is not "too small." To meet this requirement, NSCC starts each iteration by removing any nodes v for which $SUCC(v)$ is small (line 4). However, for this operation to be efficient, NSCC must be able to tell in constant time whether $|SUCC(v)| > \rho$ for every $v \in G$, where ρ is a function of the number of processors and size of the graph. Therefore, NSCC cannot use the black-box $RQ\text{-}fw(G, v, mk)$; when NSCC finds reachability or partitions the subgraphs, it must maintain a representative set of reachable nodes for every $v \in G$. This is a significant overhead, but without this step NSCC is functionally equivalent to DCSC and has the same weaknesses.

4. SCCMulti ALGORITHMS

The main contribution of this paper is the pair of SCCMulti algorithms that use multiple, overlapping reachability queries on each step to divide the graph quickly. Both algorithms only use the time and work of $\widehat{O}(\log n)$ reachability queries: SCCMulti gambles on the amount of work it does per iteration (guaranteeing $O(\log n)$ iterations), while SCCMulti-PA gambles on the number of iterations but guarantees $O(m)$ work per iteration. As a result of their structures, SCCMulti does well when reachability queries are inexpensive, whereas SCCMulti-PA does well when the average degree of the graph is small.

4.1 SCCMulti

SCCMulti, shown in Algorithm 5, selects multiple unique pivots from which to traverse. It then performs reachability queries for all pivots at one time. After eliminating the SCC of each pivot, it removes edges between nodes with different sets of visiting pivots.

To accommodate multiple pivots visiting the same node, *marks-fw* and *marks-bw* are *unordered sets* (hash sets). Each call to $RQ\text{-}fw(G, v, mk)$ operates as before, except that the mark is added to the set and does not overwrite previous marks.

Algorithm 5 SCCMulti

```
1: for k from 0 to log n do
2:    for all v ∈ G parallel do
3:       v.marks-fw ← ∅, v.marks-bw ← ∅, v.min ← ∞
4:    for all i ∈ [1, 2^k] parallel do
5:       v^i_rnd ← random_node(G)
6:       //parallel; adds i to w.marks-fw for w ∈ SUCC(v^i_rnd)
7:       RQ-fw(G, v^i_rnd, i)
8:       //parallel; adds i to w.marks-bw for w ∈ PRED(v^i_rnd)
9:       RQ-bw(G, v^i_rnd, i)
10:   //partition SCCs
11:   for all v ∈ G parallel do
12:      for all mk ∈ v.marks-bw do
13:         if v.marks-fw.contains(mk) and mk < v.min then
14:            v.min ← mk
15:      if v.min ≠ ∞ then
16:         SCC_v.min ← SCC_v.min ∪ {v}
17:         G ← G \ {v}
18:   //partition other subgraphs
19:   for all (u, t) ∈ E parallel do
20:      if u.marks-fw ≠ t.marks-fw
            or u.marks-bw ≠ t.marks-bw then
21:         E ← E \ {(u, t)}
```

4.1.1 Correctness

If nodes $u, t \in G$ are in the same SCC, they will reach and be reached by the same set of nodes. It follows that if $u.marks\text{-}fw \neq t.marks\text{-}fw$, the reachability, and thereby the SCCs, of u and t must be different. Therefore, any edges removed by SCCMulti cannot be internal to an SCC. From Lemma 2.7, any SCCs identified as the intersection of $SUCC(v)$ and $PRED(v)$ are correct. Since the algorithm correctly identifies the SCC of each pivot, it is complete. Furthermore, at least one pivot is selected and removed on each iteration; therefore, the algorithm terminates in a finite amount of time. Since SCCMulti correctly identifies all SCCs and terminates, it must be correct.

4.1.2 Analysis

First, observe that SCCMulti loops exactly $\log n$ times (line 1); therefore, as long as each iteration does $\widehat{O}(1)$ reachability queries per iteration, SCCMulti will have complexity of $\widehat{O}(\log n)$ reachability queries.

We analyze the complexity of SCCMulti in two steps. First, we prove that the total expected time and work of any iteration's reachability queries is not more than that of a single reachability query that reaches the entire graph; that is, each iteration marks $\widehat{O}(n)$ nodes. Then, we show that, like other RB-SCC algorithms, the work done in each iteration is dominated by the reachability queries.

Work on Each Iteration. We wish to prove that the resources used by each iteration are bounded by those required for a single reachability query. Since SCCMulti performs all reachability queries in parallel (as opposed to serially,

like DCSC or NSCC), we are not concerned with waiting on small reachability queries; we only need to show that $\widehat{O}(n)$ nodes will be tested for reachability on each iteration.

The proof proceeds as a series of lemmas. First, we show that the reachability of a graph G and its transitive closure $TC(G)$ are the same and that the transitive closure operation and graph division, an operation we define below, are associative (i.e., the order in which they are applied is not important). Next, we show that the reachability of a graph is proportional to the number of edges in the graph's transitive closure. Finally, we show that the expected number of edges in any graph is inversely proportional to the number of times the graph has been divided, concluding that the expected reachability of any graph is inversely proportional to the number of pivots that have acted upon it.

We first present a few definitions useful in this proof.

DEFINITION 4.1 (VERTEX REACH). *The reach of vertex $v \in G$ is*

$$reach(v) = |PRED(v) \cup SUCC(v)|,$$

i.e., the number of nodes reached by or reaching v.

DEFINITION 4.2 (GRAPH REACH). *The reach of a graph G is*

$$reach(G) = \sum_{v \in G} reach(v);$$

i.e., $reach(G)$ is the total reach of the graph, and the average reach of each node is $\frac{reach(G)}{n}$.

DEFINITION 4.3 (DIVIDED GRAPH).

1. *For a graph G and vertex $v \in G$, denote by G/v the graph resulting when G is divided by v; that is, when the following edges are removed:*
   (a) (u, t) is removed if $u \in PRED(v)$ and $t \notin PRED(v)$.
   (b) (u, t) is removed if $u \notin SUCC(v)$ and $t \in SUCC(v)$.
   (c) (u, t) is removed if $u \in SCC(v)$ or $t \in SCC(v)$.
   This operation is illustrated in Figure 2.
2. *For a graph G and $q \in \mathbb{Z}_+$, denote by $[G/v_{rand}]^q$ the graph resulting when G is serially divided by q random vertices.*

This definition comes from the behavior of a pivot in SCCMULTI. While SCCMULTI removes the SCC nodes from the graph, here we only remove the edges to and from them.

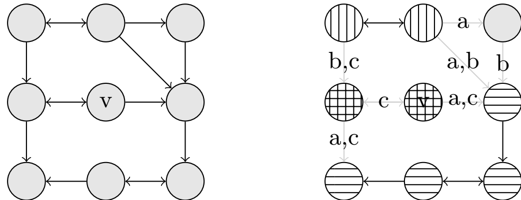


Figure 2. Example of a graph division. Nodes in $SUCC(v)$ have horizontal hatching; nodes in $PRED(v)$ have vertical hatching. Edges are labeled with the rules from Definition 4.3 part 1 that removed them.

DEFINITION 4.4 (TOPOLOGICAL-WEAK ORDERING). *For any graph G, an ordering of G's vertices $(v_1, v_2, \ldots, v_n)$ is called topological-weak if for all $1 \leq i < j \leq n$,*

$$v_j \in PRED(v_i) \Rightarrow v_j \in SCC(v_i).$$

That is, the only vertices in $PRED(v_i)$ which can come after v_i in the ordering must be in $SCC(v_i)$.

A topological-weak ordering differs from the standard topological ordering, which is only valid for an acyclic graph, in that it allows arbitrary ordering of nodes internal to an SCC.

Now we begin the proof.

LEMMA 4.5. *For any graph G,*

$$reach(G) = reach(TC(G)).$$

PROOF. Edges added by the transitive closure operation do not change the reach of any vertex. □

LEMMA 4.6. *For any graph G and $v \in G$,*

$$TC\,(G/v) = TC(G)/v.$$

PROOF. In this proof, we denote by $G.V$ the vertex set and by $G.E$ the edge set of graph G.

First, observe that $G.V = TC(G).V$ since no vertices are removed when dividing the graph. Hence, we only need to show that $TC(G/v).E \subseteq (TC(G)/v).E$ and that $TC(G/v).E \supseteq (TC(G)/v).E$.

$\subseteq$ Assume, for the sake of contradiction, that there exists some edge $(u, t) \in TC(G/v)$ that is not in $TC(G)/v$. That is, $\exists \text{path}(u, t) \in G/v$, but dividing $TC(G)$ by v removes the edge (u, t) from $TC(G)$. Since (u, t) was removed from $TC(G)$, then either $u \in PRED(v), t \notin PRED(v)$ or $u \notin SUCC(v), t \in SUCC(v)$. WLOG due to symmetry, assume $u \in PRED(v), t \notin PRED(v)$.

We know that $\exists \text{path}(u, t) \in G/v$; since $u \in PRED(v)$, $t \notin PRED(v)$, there must be some edge (w, x) along this path such that $w \in PRED(v), x \notin PRED(v)$ (that is, the state must change along this path). This edge should have been removed when v divided G, so this is a contradiction.

$\supseteq$ Assume, for the sake of contradiction, that there exists some edge $(u, t) \in TC(G)/v$ such that $(u, t) \notin TC(G/v)$. That is, dividing by v does not remove (u, t) from $TC(G)$, but it does break any paths from u to t in G. To break a path, v must remove some edge $(w, x) \in \text{path}(u, t)$, either because $w \in PRED(v), x \notin PRED(v)$ or because $w \notin SUCC(v), x \in SUCC(v)$. WLOG due to symmetry, assume $w \in PRED(v), x \notin PRED(v)$.

Because $(w, x) \in \text{path}(u, t)$, $u \in PRED(w)$, and since we assume $w \in PRED(v)$, then $u \in PRED(v)$. Because (u, t) was not removed from $TC(G)$, u and t were reached the same way by v, so $t \in PRED(v)$. Since $(w, x) \in \text{path}(u, t)$ and $x \in PRED(t)$, then $x \in PRED(v)$. This is a contradiction.

□

LEMMA 4.7. *For any graph G such that $TC(G) = (V, E')$,*

$$reach(G) \leq 2 \cdot |E'| + n.$$

PROOF. All of the nodes reached by a vertex v are its neighbors in the transitive closure. Every edge has two endpoints, so the total number of neighbors is not more than $2 \cdot |E'|$. Each node can also reach itself, so the total reachability is not more than $2 \cdot |E'| + n$. □

LEMMA 4.8. *When a graph G is divided by a random vertex, the expected number of edges removed from G is at least $\frac{m^2}{2n^2} = \widehat{O}(\frac{m^2}{n^2})$.*

PROOF. First consider a single vertex $v \in G$, which has f forward edges, and count how many of these edges we expect to be removed from v.

Let $(u_1, u_2, \ldots, u_f)$ be a topological-weak ordering of v's forward neighbors. For any u_i, $v \in \mathit{PRED}(u_i)$; if u_i is selected to divide the graph, the edge (v, u_j) will be removed unless $u_j \in \mathit{PRED}(u_i)$ **and** $u_j \notin \mathit{SCC}(u_i)$. Since v's forward neighbors are topological-weakly ordered, only $i-1$ such u_j can exist.

Each u_i will be selected to divide the graph with probability $\frac{1}{n}$. Therefore, we expect at least

$$\sum_{i=1}^{f} \frac{f-(i-1)}{n} = \frac{f^2+f}{2n} \geq \frac{f^2}{2n}$$

forward edges of v to be removed by a random vertex.

v is only one vertex; over all vertices, we expect at least $\sum_{i=1}^{n} \frac{f_i^2}{2n}$ edges to be removed, where f_i is the number of forward edges on the ith vertex. Since $\sum f_i = m$, $\sum f_i^2$ is minimized when $f_i = \frac{m}{n}$. Therefore, we expect at least

$$\sum_{i=1}^{n} \frac{\left(\frac{m}{n}\right)^2}{2n} = \frac{m^2}{2n^2}$$

edges to be removed. □

Lemma 4.9. *For any graph G and $q \in \mathbb{Z}_+$, the expected number of edges in $[G/v_{rand}]^q$ is no more than $\frac{2n^2}{q+2} = \widehat{O}(\frac{n^2}{q})$.*

Proof. For the sake of contradiction, assume that there exists some graph G and $q \in \mathbb{Z}_+$ such that m_q, the expected number of edges in $[G/v_{rand}]^q$, exceeds $\frac{2n^2}{q+2}$. Furthermore, assume that q is the smallest such integer. No graph may have more than n^2 edges, so $q \neq 0$.

Since q is the smallest such integer, m_{q-1} is no more than $\frac{2n^2}{q+1}$. Also, $m_q \leq m_{q-1}$, so $m_{q-1} > \frac{2n^2}{q+2}$. To restate,

$$\frac{2n^2}{q+2} < m_{q-1} \leq \frac{2n^2}{q+1}.$$

By Lemma 4.8, the expected number of edges remaining after the qth division is

$$\begin{aligned} m_q &\leq m_{q-1} - \frac{m_{q-1}^2}{2n^2} \\ &= m_{q-1} \cdot \left(\frac{2n^2 - m_{q-1}}{2n^2}\right) \\ &\leq \frac{2n^2}{q+1} \cdot \left(\frac{2n^2 - \frac{2n^2}{q+2}}{2n^2}\right) \\ &= \frac{2n^2}{q+2}. \end{aligned}$$

This violates the assumption. □

Lemma 4.10. *For graph G and $q \in \mathbb{Z}_+$, the expected reachability of any single vertex in $[G/v_{rand}]^q$ is $\widehat{O}(\frac{n}{q})$.*

Proof. By Lemmas 4.6 and 4.9, the expected number of edges in $TC([G/v_{rand}]^q) = [TC(G)/v_{rand}]^q$ is $\widehat{O}(\frac{n^2}{q})$. By this and Lemma 4.7, the expected reach of $[G/v_{rand}]^q$ is $\widehat{O}(\frac{n^2}{q})$, so the expected reachability of any single vertex is $\widehat{O}(\frac{n}{q})$. □

Note that $[G/v_{rand}]^q$ is expected to have at least as many edges as a graph which has been operated on by q pivots in SCCMulti, since $[G/v_{rand}]^q$ is divided by each pivot one after the other, whereas divisions in SCCMulti may overlap (thereby removing at least as many edges). On iteration k, SCCMulti divides by $O(2^k)$ vertices; since $O(2^k)$ vertices divide the graph before iteration k, we will reach $\widehat{O}(2^k \frac{n}{2^k}) = \widehat{O}(n)$ vertices (and $\widehat{O}(m)$ edges) on this iteration. Furthermore, only $\widehat{O}(n)$ marks will be created, so the memory per node is $\widehat{O}(1)$.

Dominated Work. In addition to the reachability queries, SCCMulti performs two operations on the marking sets. First, for every vertex v, it must find the smallest mark which is in both *v.marks-fw* and *v.marks-bw* (lines 12 to 14). Since the expected number of marks per node is $\widehat{O}(1)$, scanning through *v.marks-bw* (line 12) takes $\widehat{O}(1)$ work. *marks-fw* is an unordered hash set, so *marks-fw.contains*$(\cdot)$ is an $O(1)$ work operation. Therefore, this operation takes $\widehat{O}(1)$ work per vertex.

SCCMulti must also compare the marking sets across every edge; this can be done in $O(1)$ work per edge. Consider the edge (u, t): any mark that reaches u will reach t along (u, t). Consequently, *u.marks-fw* $\subseteq$ *t.marks-fw*, so *u.marks-fw* $\neq$ *t.marks-fw* only if $|$*u.marks-fw*$| > |$*t.marks-fw*$|$.

Therefore, the resources used by this algorithm are dominated by the reachability queries.

4.2 SCCMulti-PriorityAware

Assume that there exists a blocking, *priority-aware* reachability query; that is, every pivot is assigned a priority, and the query marks the set of nodes that are reached by the current pivot but not by any higher priority pivot. When this algorithm is called with more than one pivot, the highest priority pivot will mark every node it would have marked in the black-box, non priority-aware *RQ-fw*(G, v, m), while lower priority pivots will only mark a subset of nodes that they would otherwise mark.

If this scheme is implemented such that processors first operate using the highest priority pivot of which they are aware, then it will take $O(1)$ times longer than if the highest priority pivot was operating alone; any time used by a processor before it becomes aware of the highest priority pivot would have otherwise been spent in an idle state.

When the cost of each reachability query is high, it may be advantageous to use these *priority-aware* reachability queries. Since this type of reachability query can stop when it visits a node already visited by a higher priority query, this limits the total work to $O(1)$ times that of one reachability query that reaches the entire graph.

SCCMulti-PA, shown in Algorithm 6, uses this strategy. Instead of maintaining a set of visiting pivots, each node only stores its highest priority visiting pivot. Any node that has the same highest priority pivot both forward and backward is added to that pivot's SCC. Then, any edges between nodes marked differently are removed, and the algorithm repeats.

This algorithm attempts n reachability queries in each iteration; using a black-box reachability algorithm, the total work in each iteration would be much higher than that of a single reachability query. Therefore, we require that the reachability query be priority-aware; it stops traversing the graph when it has reached nodes already visited by higher priority queries, and each compute node always operates first on the highest priority pivot of which it is aware.

4.2.1 Correctness

If nodes $u, t \in G$ are in the same SCC, they will reach and be reached by the same highest-priority pivot, so SCCMulti-PA will not remove any edges internal to an SCC. At the same time, every iteration must have some highest-priority

Algorithm 6 SCCMulti-PA

```
1: while |G| > 0 do
2:   for all v ∈ G parallel do
3:     v.marks-fw ← ∅, v.marks-bw ← ∅
4:     v.priority ← rand(0, n) * n + v.index
5:   for all v ∈ G parallel do
6:     RQ-fw-PA(G, v, v.priority)      //see discussion
7:     RQ-bw-PA(G, v, v.priority)      //see discussion
8:   //partition SCCs
9:   for all v ∈ G parallel do
10:    if v.marks-fw = v.marks-bw then
11:      SCC_{v.marks-fw} ← SCC_{v.marks-fw} ∪ {v}
12:      G = G \ {v}
13:  //partition subgraphs
14:  for all (u, t) ∈ E parallel do
15:    if u.succ ≠ t.succ or u.pred ≠ t.pred then
16:      E ← E \ {(u, t)}
```

pivot, so the algorithm must terminate. Therefore, the algorithm correctly labels each SCC and terminates.

4.2.2 Analysis

First, we show that SCCMulti-PA expects $\widehat{O}(\log n)$ iterations. Then, we again show that each iteration's slowest step is the reachability queries.

Number of Iterations. By Lemma 4.10, vertices in a graph G that has been divided by q pivots have reach of $\widehat{O}(\frac{n}{q})$; therefore, $\widehat{O}(n/\frac{n}{q}) = \widehat{O}(q)$ pivots will reach their own SCCs without being overwritten by a higher priority pivot. By this, the number of non-overwritten pivots doubles on each iteration, so SCCMulti-PA will take $\widehat{O}(\log n)$ iterations.

Since SCCMulti-PA tests reachability on $O(n)$ vertices on each iteration and uses $\widehat{O}(\log n)$ iterations, it will also use the time and work of $\widehat{O}(\log n)$ reachability queries on the entire graph.

Dominated Work. At the end of every iteration of SCCMulti-PA, every node will have been labeled by some pivot. Since the reachability queries are priority-aware, the time required for all of them will be asymptotically equal to that of a single reachability query over the entire graph.

5. EXPERIMENTAL RESULTS

Our experimental evaluation of the RB-SCC algorithms compares the absolute execution time and scalability of each algorithm on different types of graphs. The use of multiple graphs from different domains demonstrates that the performance of our algorithms is not restricted to a specific problem domain. That is, the algorithmic characteristics noted in the preceding discussion are evident when the algorithms are used to identify the SCCs of graphs.

5.1 Experimental Setup

We implemented DCSC, Hong's modified DCSC (HONG), MultiPivot, SCCMulti, and SCCMulti-PA using stapl [5]. *RQ-fw*$(\cdot)$ and *RQ-bw*$(\cdot)$, used by all algorithms, were implemented as flood fills using stapl's KLA paradigm [10], which enables simple development of efficient parallel algorithms for which the level of asynchrony can be varied parametrically. The flood fill algorithm for SCCMulti-PA uses a priority scheduler to determine which task from the set of ready tasks is run next. Vertex operations are guaranteed to be atomic by stapl's task scheduler and runtime system. We did not implement NSCC due to its high memory requirement. We include an evaluation of our reachability query (RQ), which measures the time required to perform $\lceil \log n \rceil$ queries on the input graph, to benchmark the execution times of the algorithms. All implementations use the Trim subroutine (Section 3.1.2) as a preprocessing phase at each level.

Each iteration of MultiPivot takes approximately the same time as one complete call to SCCMulti or SCCMulti-PA; because of this, we always stopped MultiPivot after four iterations. Our results show that on all graphs the SCCMulti algorithms outperform this abbreviated MultiPivot; running MultiPivot to completion would consume our limited computing resources without changing the relative outcome.

Our experiments were run on a Cray XE6m-200 (XE6M) available in our department. This machine has twenty-four compute nodes, twelve of which have a single AMD Opteron 6272 'Interlagos' 16-core processor running at 2.1GHz and a total of 32GB of memory (2GB per core). The remaining twelve nodes have two 16-core 'Interlagos' processors running at 2.1GHz and a total of 64GB of memory (maintaining 2GB per core). Experiments run on 32 to 256 cores were restricted to run on nodes with 32 cores in order to minimize variability in the experimental setup.

We evaluated the scalability of all five algorithms on a 393,216 core IBM BG/Q system (BG/Q) at Lawrence Livermore National Laboratory. The BG/Q system has 24,576 nodes, each with 16GB of memory and a single 16 core IBM PowerPC A2 processor running at 1.6 GHz. We ran each experiment 10 times and for all results present the mean along with a 95% confidence interval using the t-distribution. Additional results can be found in a technical report [20] available on our laboratory website, http://parasol.tamu.edu/publications.

5.2 Input Graphs

In order to show that the performance of our algorithms is not limited to a particular class of graphs, the algorithms were evaluated using five different types of graphs:

Separable Cycles (SC): A graph with a set of unconnected cycles.

Chained Cycles (CC): A graph consisting of a chain of cycles.

Permuted Mesh (PM): A graph representing a 3D mesh with a specified percentage of edges randomly reversed.

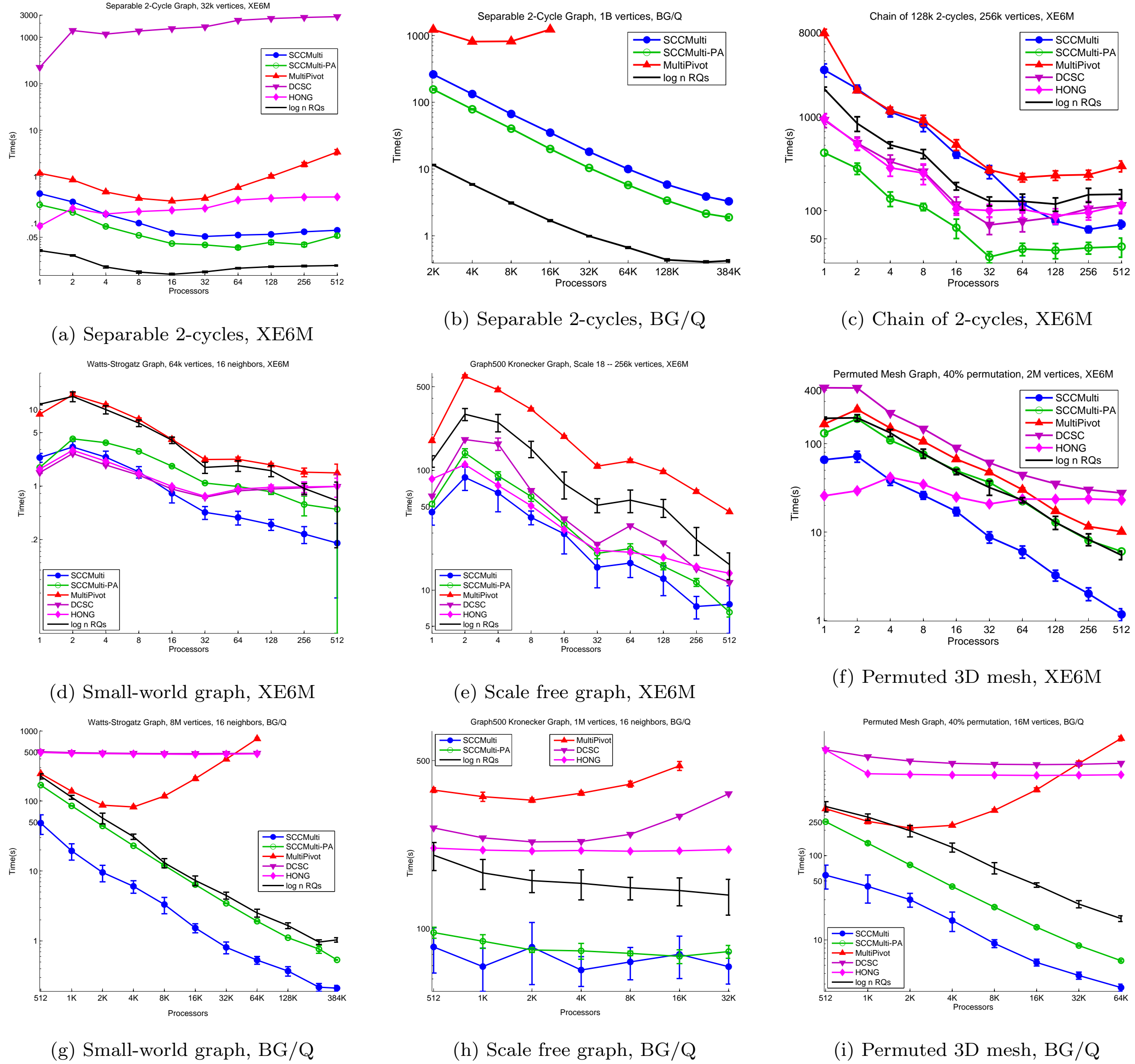
Watts-Strogatz (WS): A graph with small-world properties created by forming a ring of vertices and, for each vertex, randomly rewiring each of its edges to one of its k neighbors.

Graph 500 (G500): A graph with scale-free, small-world properties created using the modified Kronecker generator from the Graph 500 benchmark.

SC was chosen to represent the class of problems where the graph has low average reachability; this is the classically difficult input for DCSC. CC is an extreme case that stresses the underlying RQ. PM represents graphs found, for example, in scientific applications where the spatial domain is represented by a graph and local mutations cause the connectivity between vertices to change. WS and G500 have many of the small-world properties found in graphs from social networks, road maps, and other physical phenomena.

5.3 Comparison of RB-SCC Algorithms

We first compare the RB-SCC algorithms on the XE6M using a 32,768 vertex SC graph with cycles of size 2, shown



(a) Separable 2-cycles, XE6M

(b) Separable 2-cycles, BG/Q

(c) Chain of 2-cycles, XE6M

(d) Small-world graph, XE6M

(e) Scale free graph, XE6M

(f) Permuted 3D mesh, XE6M

(g) Small-world graph, BG/Q

(h) Scale free graph, BG/Q

(i) Permuted 3D mesh, BG/Q

Figure 3. Timing results for different graphs on XE6M and BG/Q platforms

in Figure 3(a). For this input DCSC does not scale and requires two to three orders of magnitude longer than the other algorithms; as mentioned in Section 3.2, unconnected SCCs do not allow DCSC to exploit any parallelism. HONG has significantly reduced run time compared to DCSC, but does not change improve in scalability. For SC, the four iterations of MultiPivot take approximately four times as long as SCCMulti and SCCMulti-PA. This is expected, as MultiPivot does as much work at each level as SCCMulti does in total. Both SCCMulti and SCCMulti-PA scale better than any of the other algorithms, in addition to having lower execution times; the scalability of both SCCMulti and SCCMulti-PA is comparable to that of the RQ. (Again, note that RQ shows the time required for $\lceil \log n \rceil$ serial reachability queries.) We expect all algorithms to scale with the RQ, but the communication cost of maintaining subgraph information penalizes DCSC, HONG, and MultiPivot.

The comparison of the algorithms on the CC input is shown in Figure 3(c). DCSC and SCCMulti-PA have comparable execution time and scalability. MultiPivot is still the slowest of the algorithms, despite its execution being halted after four iterations. Since this is a graph with exceptionally low degree, SCCMulti-PA outperforms all the other algorithms, and SCCMulti scales similarly. The decrease in scalability from four to eight cores is due to limited memory bandwidth as all eight cores on one die of the 16-core processor are utilized. The lack of scalability at higher core counts is due to the small graph size, used to allow MultiPivot to complete its abbreviated execution in a reasonable amount of time.

Figure 3(d) presents the comparison of the algorithms on a WS input. MultiPivot remains the most computationally expensive, but after an increase in execution time be-

tween 1 and 2 cores it scales similarly to the others. DCSC, HONG, SCCMulti, and SCCMulti-PA have similar execution times on lower core counts, but as the number of cores increases SCCMulti and SCCMulti-PA outscale DCSC and HONG. In all experiments the scalability of SCCMulti and SCCMulti-PA is similar to that of the $\lceil \log n \rceil$ RQs.

The execution times of the RB-SCC algorithms on a G500 input are shown in Figure 3(e). Similar to the results obtained on the WS graph, MultiPivot is the most computationally expensive algorithm. SCCMulti, SCCMulti-PA, DCSC and HONG scale similarly and have comparable execution times. All algorithms show a decrease in scaling from 32 to 64 cores when two nodes of the system are employed instead of a single node. Scalability beyond 32 cores is lower due to the large number of messages required to complete a single RQ and the fact that most of these messages are between different nodes of the system.

The performance of all of the RB-SCC algorithms on a PM with 40% edge reversal is shown in Figure 3(f). All of the algorithms, with the exception of HONG, scale similarly, with SCCMulti performing the best on all core counts (this is a higher degree graph with low diameter). Hong's modified DCSC does not scale well due to the increased communication required by the computation of the WCCs.

5.4 Scalability of RB-SCC Algorithms

To evaluate the scalability of the algorithms on higher core counts we use the SC, PM, WS, and G500. The SC scalability up to 384K cores is shown in Figure 3(b), where DCSC and HONG are excluded because of their already poor scalability in Figure 3(a). Because of the overhead involved in finding subgraphs, MultiPivot does not scale on these core counts, and it was not run beyond 16,384 cores. SCCMulti and SCCMulti-PA both scale well across all core counts evaluated. The execution of $\lceil \log n \rceil$ reachability queries(RQ), on which all of the algorithms are based, is also included in the figure to show that the underlying query is itself scalable to high core counts.

Figure 3(g) illustrates the scalability of all five algorithms on a WS graph of 8 million vertices out to 393,216 cores. The scalability of the algorithms on PM out to 65,536 cores is shown in Figure 3(i). The trends noted in earlier experiments continue to hold: SCCMulti and SCCMulti-PA are consistently the fastest algorithms and demonstrate superior scalability comparable to the baseline of $\lceil \log n \rceil$ RQs. Again, because of the overhead involved in finding pivots, MultiPivot does not scale on the large number of cores being utilized, and it was not run beyond 64K cores. The bookkeeping of DCSC and HONG prevent them from scaling, and the algorithms require more execution time than all other algorithms (except on the highest core counts of each experiment, when MultiPivot's execution time exceeds them). In Figures 3(b) and (g) the three highest core counts used are 128K, 256K, and 384K.

Figure 3(h) shows that none of the algorithms scale well on the scale-free graph from the Graph 500 benchmark. This is due to the lack of scalability of the reachability query on which all of the algorithms are based. The reachability query does not scale due to the large amount of inter-node communication required in order to traverse the graph.

The results show that, across a variety of graphs, SCCMulti and SCCMulti-PA have low execution times and exhibit scalable performance out to 384k cores, making them a preferable alternative to MultiPivot, DCSC, and HONG.

6. CONCLUSION

Computing SCCs is an important problem, yet solutions to it are difficult to parallelize. Many of the best performing parallel SCC algorithms for sparse graphs work by choosing random pivots, computing the reachability of these pivots, and using this information to divide the graph. This paper surveys these reachability-based SCC (RB-SCC) algorithms and shows that they subscribe to a single framework.

The main contribution of this paper is two new parallel SCC algorithms, SCCMulti and SCCMulti-PA, that use multiple pivots in the same iteration to divide the graph into many pieces. SCCMulti guarantees $O(\log n)$ iterations and probabilistically guarantees the time and work of $\widehat{O}(1)$ reachability queries per iteration, while SCCMulti-PA guarantees the time and work of $O(1)$ reachability queries per iteration, but only probabilistically guarantees $\widehat{O}(\log n)$ iterations. Unlike previous RB-SCC algorithms (DCSC, MultiPivot, and NSCC), these algorithms offer this performance on any graph without introducing significant computation or memory overhead. Our experimental results show that SCCMulti and SCCMulti-PA scale up to hundreds of thousands of cores; unlike DCSC, they attain consistent performance regardless of the input graph, and they are much faster than MultiPivot. Therefore, they are consistently fast, scalable parallel SCC algorithms.

APPENDIX

This section describes a graph that will take DCSC $\Theta(n)$ recursive levels, even for the modified DCSC algorithm of Hong et al. [11]. Consider a 3D Cartesian space. Place a node at every integral $(x, y, 0)$ and $(x, y, 1)$, creating two horizontal planes of nodes. For every node in the $z = 0$ plane, create an edge to the five vertices closest to it in the $z = 1$ plane; that is, $(x, y, 0)$ will have edges to $(x-1, y, 1)$, $(x+1, y, 1)$, $(x, y, 1)$, $(x, y-1, 1)$, and $(x, y+1, 1)$. A portion of this graph is shown in Figure 4. The nodes in the $z = 0$ plane lie along dashed lines, while the nodes in the $z = 1$ plane lie along solid lines. The five directed lines show the outgoing edges for an example node in the $z = 0$ plane.

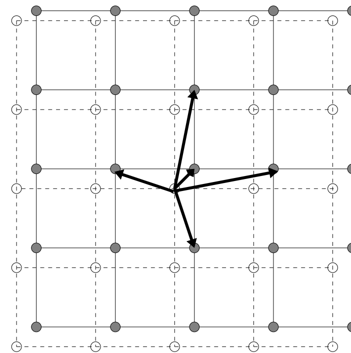


Figure 4. A bad case graph for the modified DCSC

This is a directed graph with low average reachability; however, no pivot will ever divide it into more than seven weakly connected components: the SCC of the pivot (which is the pivot itself), the five vertices to which the pivot has edges, and the rest of the graph.

One could argue that the SCCs of this graph would be found by the modified Trim subroutine. This is indeed true; however, replacing every individual vertex with a 3-cycle would make Trim useless, but would not significantly change

the properties of the graph. Therefore, this graph presents a worst case graph for DCSC that is not assisted by the adaptation of Hong et al.[11].

References

[1] M. L. Adams and E. W. Larsen. Fast iterative methods for discrete-ordinates particle transport calculations. *Progress in Nuclear Energy*, 40(1):3–159, 2002.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] N. Amato. Improved processor bounds for parallel algorithms for weighted directed graphs. *Information Processing Letters*, 45(3):147–152, Mar. 1993.

[4] D. Bader. A practical parallel algorithm for cycle detection in partitioned digraphs. Technical Report AHPCC-TR-99-013, Electrical & Computer Eng. Dept., Univ. New Mexico, Albuquerque, NM, 1999.

[5] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. STAPL: Standard template adaptive parallel library. In *Proc. Annual Haifa Experimental Systems Conference (SYSTOR)*, pages 1–10, New York, NY, USA, 2010. ACM.

[6] L. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In J. D. P. Rolim, editor, *IPDPS Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pages 505–511. Springer, 2000.

[7] A. A. Gakh, E. G. Gakh, B. G. Sumpter, and D. W. Noid. Neural network-graph theory approach to the prediction of the physical properties of organic compounds. *J. of Chemical Information and Computer Sciences*, 34(4):832–839, 1994.

[8] H. Gazit and G. L. Miller. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Information Processing Letters*, 28(2):61–65, June 1988.

[9] M. Goodrich, Y. Matias, and U. Vishkin. Optimal parallel approximation for prefix sums and integer sorting. In *Proc. of the 5th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 241–250, 1994.

[10] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. KLA: A new algorithmic paradigm for parallel graph computations. In *Proc. Intern. Conf. Parallel Architecture and Compilation Techniques (PACT)*, PACT '14, pages 27–38, New York, NY, USA, 2014. ACM. Conference Best Paper Award.

[11] S. Hong, N. C. Rodia, and K. Olukotun. On fast parallel detection of strongly connected components (scc) in small-world graphs. In *Proc. of the Intern. Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 92:1–92:11, New York, NY, USA, 2013. ACM.

[12] M.-Y. Kao, S.-H. Teng, and K. Toyama. An optimal parallel algorithm for planar cycle separators. *Algorithmica*, pages 398–408, 1995.

[13] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *Robotics and Automation, IEEE Trans. on*, 12(4):566 –580, aug 1996.

[14] W. McLendon, III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger. Finding strongly connected components in distributed graphs. *J. Parallel Distrib. Comput.*, 65:901–910, 2005.

[15] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[16] W. Schudy. Finding strongly connected components in parallel using o(log2n) reachability queries. In *Proc. of the 20th Annual Symp. on Parallelism in Algorithms and Architectures*, SPAA '08, pages 146–151, New York, NY, USA, 2008. ACM.

[17] T. H. Spencer. More time-work tradeoffs for parallel graph algorithms. In *Proc. of the 3rd Annual ACM Symp. on Parallel Algorithms and Architectures*, SPAA '91, pages 81–93, New York, NY, USA, 1991. ACM.

[18] T. H. Spencer. Time-work tradeoffs for parallel graph algorithms. In *Proc. of the 2nd Annual ACM-SIAM Symp. on Discrete Algorithms*, SODA '91, pages 425–432, Philadelphia, PA, USA, 1991. SIAM.

[19] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[20] D. Tomkins, T. Smith, N. M. Amato, and L. Rauchwerger. Efficient, reachability-based, parallel algorithms for finding strongly connected components. Technical Report TR15-002, Dept. of Computer Science and Engineering, Texas A&M University, January 2015.

[21] J. D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM J. Comput.*, 20(1):100–125, 1991.