Finding Strongly Connected Components in Distributed Graphs

William McLendon III Bruce Hendrickson Steven J. Plimpton
Sandia National Laboratories
{wcmclen, bahendr, sjplimp}@sandia.gov

Lawrence Rauchwerger
Dept. of Computer Science, Texas A&M University
rwerger@cs.tamu.edu

Abstract

The traditional, serial, algorithm for finding the strongly connected components in a graph is based on depth first search and has complexity which is linear in the size of the graph. Depth first search is difficult to parallelize, which creates a need for a different parallel algorithm for this problem. We describe the implementation of a recently proposed parallel algorithm that finds strongly connected components in distributed graphs, and discuss how it is used in a radiation transport solver.

1 Introduction

A strongly connected component (SCC) of a directed graph is a maximal subset of vertices in which there is a directed path from any vertex to any other. A cycle in a directed graph is a path that is simple except the first and final vertices are the same. Although the number of cycles in a graph can be exponential in the number of vertices, the number of SCCs is at most linear in the number of vertices, since no vertex can be in more than one SCC. For our purposes we will only consider a subset of vertices to be an SCC if it has more than one vertex.

Tarjan's classic serial algorithm for detection of SCCs runs linearly with respect to the number of edges and uses depth-first search [1]. However, depth-first search is known to be difficult to parallelize – the special case of lexicographical depth first search is P-Complete [2, 3], which in practical terms means it is unlikely that a scalable parallel algorithm exists.

There are some parallel algorithms for detecting SCCs that do not rely on depth first search. Gazit and Miller have an NC algorithm which can be used for locating SCCs that uses matrix multiplication [4]. Vishkin and Cole [5] and Amato [6] have proposed optimizations or extensions of this algorithm, but they still require $O(n^{2.376})$ processors and $O(\log^2\ n)$ time where n is the number of vertices in the graph. A more complicated NC algorithm developed by Kao for planar graphs requires $O(\log^3\ n)$ time and $n/\log\ n$ processors [7]. Another parallel algorithm for planar graphs is due to Bader [8], but our applications are non-planar, arising from graphs associated with finite element mesh representations of 3-D domains.

In this paper we describe our modification of a recently proposed algorithm due to Fleischer, *et al.* [9] and our parallel implementation of it in MPI. The Fleischer, *et al.* algorithm, called DCSC for *divide–and–conquer strong components* is a recursive, divide–and–conquer approach that does not rely on depth first search. As shown in [9], its expected serial runtime is $O(m \log n)$, where m is the number of edges and n is the number of vertices in the graph. We describe the DCSC algorithm

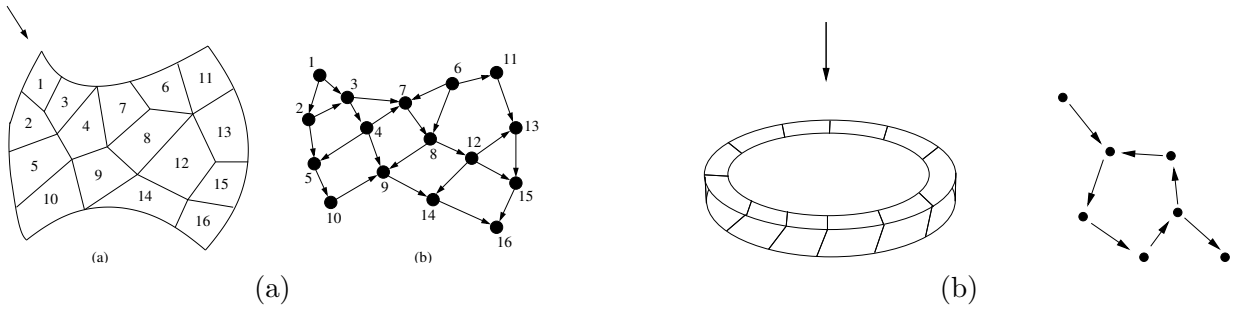


Figure 1: (a) An unstructured finite-element mesh (left) and its associated acyclic dependence graph for the angle shown (right). (b) A twisted ring of mesh elements that induces a cycle for the shown angle (left), and its dependence graph for the angle shown (right). A sweeping method will deadlock when it encounters a cycle such as this.

and our modifications to it in §2. We then present details of our parallel implementation in §3. In §4 we quantify the performance of our approach by presenting experimental results obtained on two different machines: CPlant, a 1700 processor DEC Alpha commodity cluster and ASCI Red, a 9280 processor Intel supercomputer. Both machines are located at Sandia National Laboratories.

The motivation for this work is a computational code for solving radiation transport on 3D unstructured finite element grids. The model of radiation transport solver we have selected for this work solves the transport equations using a *sweep* method. Sweeping methods used in radiation transport discretize the radiation field by angle, and flux propagation is computed for a set of discrete directions or ordinates. The computation for each angle is performed by sweeping the flux across a grid, i.e., a finite element mesh. Radiation enters a mesh cell via faces whose outward normals point upwind, and exits through downwind faces. This implies an order of computation on the grid cells which, for a single ordinate direction, is represented as a directed dependence graph (DDG). Two example meshes and their associated dependence graphs for a particular angle are shown in Fig. 1.

Each of the (typically several hundred) ordinate directions induces an associated dependence graph. Sweeping methods will deadlock if any of the dependence graphs contains a cycle, such as the one in the dependence graph for the twisted grid shown in Fig. 1(b). Such situations are not uncommon in 3-D unstructured grids and in Lagrangian simulations where the underlying discretized object (the mesh) deforms over time. To avoid deadlock, cycles in the set of ordinate dependence graphs must be detected and broken before the sweep can be performed. Since the mesh elements (vertices of the dependence graph) are distributed across processors, a key step in parallelizing transport sweeps is a scalable parallel algorithm for cycle detection.

2 The Modified DCSC Algorithm

The main idea of the DCSC algorithm for strongly connected components is to recursively partition the directed graph $G = (V, E)$ in such a way that any SCCs will be entirely contained within a single partition. Each recursive step in DCSC begins with the selection of a random *pivot* vertex v. Next, the algorithm finds $Pred(G, v)$, the set of *predecessors* of v, which are all the vertices which can reach v by a directed path of edges. Similarly, it finds $Desc(G, v)$, the set of *descendants* of v, the vertices that can be reached from v by a directed path of edges. All vertices which are not predecessors or descendants are in the *remainder* set, $Rem(G, v)$. The partitioning is based on the

following Lemma [9].

Lemma 1 *The unique SCC containing v in G is $Pred(G, v) \cap Desc(G, v)$. Moreover, any other SCC of G is a subset of $Pred(G, v)$, a subset of $Desc(G, v)$, or a subset of $Rem(G, v)$.*

With this lemma, the graph is broken into three disjoint pieces, and the algorithm is applied recursively to each piece. The recursion stops when partitions contain zero or one vertex. The expected serial complexity of DCSC is shown in [9] to be $O(m \log n)$.

The DCSC algorithm is amenable to parallelism in two ways. First, each recursion generates a set of up to three independent problems which can be analyzed independently. Second, the principle computational step is the search for ancestors and descendants which is like a topological traversal of the graphs. This type of traversal has much more parallelism [10] than depth-first search used in Tarjan's algorithm. But this parallelism comes at the cost of an extra factor of $\log n$ in the run time. In addition, in the radiation transport applications which motivated our work, multiple directed graphs on the same nodal set need to be analyzed for cycles simultaneously. This provides yet more scope for parallelism as will be discussed further in §3.1

The radiation transport applications of interest to us generally have few SCCs. By efficiently eliminating portions of the graph without SCCs, we can reduce the size of a problem before invoking DCSC and so improve overall performance. Our algorithm to do this, which we call ModifiedDCSC, is outlined in Fig. 2. Steps (6)–(11) comprise the DCSC algorithm, but in our approach we perform a *trim* step at the beginning of each iteration which tries to reduce the size of the graph that must be processed by eliminating vertices which cannot be part of an SCC. The forward and backward trim steps involve *topological traversals* of the graph. The forward trim begins with all vertices with no ancestors and removes them and all their edges. After their removal, some other vertices may now have no ancestors and they are removed. The process continues until no more vertices can be removed. Vertices that are part of an SCC will not be eliminated during a trim due to the nature of the topological traversal. The reverse trim performs the same operation from the other end of the graph. All vertices with no descendants are removed recursively. These two trim operations can be performed in $O(m)$ serial time. If the graph has no SCCs then all the vertices will be removed in the forward trim. It is worth noting that the trim operation exactly mimics the steps in a transport sweep. But in our case instead of simulating radiation, we merely note whether or not a given cell can receive all the data it needs to do its computation.

We say that vertices that are reachable from a SCC are contained in the *shadow* of the SCC. The forward trim of G removes all vertices from V that are not contained in SCCs or in the forward shadow of some SCC. The reverse trim also produces a shadow in the reverse direction. The intersection of these two shadows, which we call the *dark shadow* is then partitioned via a single level of the DCSC algorithm.

Fig. 4(a) illustrates how the forward and reverse trim steps remove nodes which are neither a part of nor are dependent on a SCC. A more abstract view is shown in Fig. 3(a). If the reverse trim encounters another SCC within the shadow of the forward trim, a second shadow will be cast by this SCC into the previous shadow. The resulting dark shadow contains the vertices of G that must be further processed. The effectiveness of trimming the DDG is dependent on how close the SCCs are to the starting points of the forward and reverse topological traversals (vertices with in-degree or out-degree zero, respectively). If we encounter a SCC early in the traversal, the shadow will be large, thus reducing the effectiveness of the trim.

The partitioning of the dark shadow into disjoint regions is illustrated in Fig. 3(b). An example of this partitioning on an actual dependence graph is shown in Fig. 4(b), where the graph is

Algorithm: ModifiedDCSC(G)

```
1. IF G has no more than 1 vertex THEN return
2. TRIM G in forward direction
3. IF G is not empty THEN
4.    TRIM G in backward direction
5.    Select pivot v from the dark shadow of G
6.    MARK Pred(G,v) and Desc(G,v) in G
7.    SCC(G,v) = Pred(G,v) ∩ Desc(G,v)
8.    DO in parallel:
9.       ModifiedDCSC( Pred(G,v) \ SCC(G,v) )
10.      ModifiedDCSC( Desc(G,v) \ SCC(G,v) )
11.      ModifiedDCSC( Rem(G,v) )
12.ENDIF
```

Figure 2: ModifiedDCSC Algorithm.

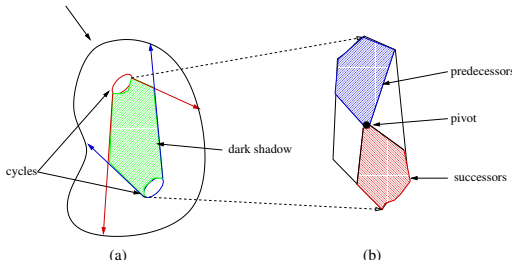
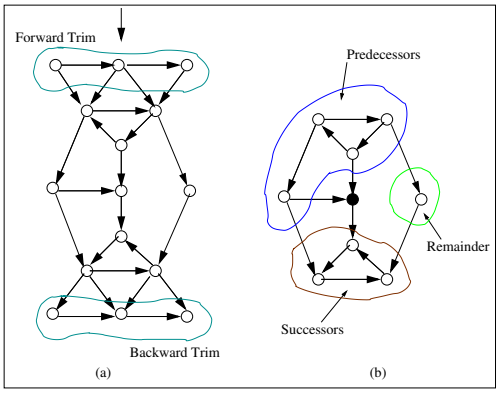


Figure 3: (a) An abstract 2D mesh containing two SCCs (the circular rings). The TRIM steps remove the white regions, leaving the dark shadow (shaded). (b) The dark shadow is partitioned around the pivot into *Pred*, *Desc*, and *Rem* sets by the MARK step.



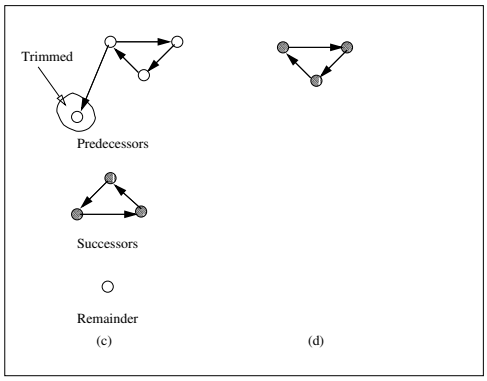


Figure 4: Example graph being trimmed in (a) by forward and backward trims. Part (b) shows the predecessor, descendant, and remainder markings of the resulting dark shadow for the selected pivot (the solid vertex). Parts (c) and (d) show the subgraphs as they are reduced to strongly connected components.

partitioned into three subgraphs that do not share any SCCs. Fig. 4(c, d) show two more iterations of ModifiedDCSC where the graphs are reduced to SCCs.

The dark shadow is then partitioned by a *marking* step. The marking algorithm proceeds as topological traversals of G, originating from a single vertex in G we call the pivot, v. There are two traversals, one which follows forward edges from v marking all vertices that are *reachable from* v, and one that follows edges backwards to mark all vertices *from which* v *is reachable*. The topological nature of this traversal allows for parallelism in the same manner as in trimming.

After marking is complete, the nodes that are both predecessors and descendants comprise the SCC containing v. The pivot v, and any SCC containing it, are extracted from the graph, partitioning it into three disjoint subgraphs containing $Pred(G, v)$, $Desc(G, v)$, and $Rem(G, v)$. The key observation of [9] is that any SCCs remaining in the graph will be wholly contained within one of these regions. Thus we can call ModifiedDCSC recursively on each of the 3 new graphs. The recursion stops when all subgraphs contain one or fewer vertices.

3 Implementation

In serial, the ModifiedDCSC algorithm outlined in Fig. 2 is straightforward to implement. The principle computational steps are the trim operations (a topological traversal of the graph) and the mark operations (determination of ancestors and descendants). Each of these graph operations can be performed efficiently using a task queue. For the forward trim, begin by placing all the vertices with no ancestors in a queue of tasks. Now remove a vertex from the queue, delete it from the graph and decrement the ancestor count of all the vertices it points to. If any of these vertices now have no ancestors, add them to the queue. When the queue is empty, the trim step is finished. The backward trim is closely analogous but with descendants and ancestors flipped.

A similar approach works for the descendant (and ancestor) determinations in the mark phase. Initially, the pivot vertex is marked and placed in a task queue. Now vertices are removed from the task queue one at a time, and all of their unmarked children (or parents) are added to the task queue. The process continues until the task queue becomes empty. In this way, all the trim and mark operations in a single level of recursion can be performed in $O(m)$ time.

We have implemented a parallel version of the ModifiedDCSC algorithm in C with MPI. Our code takes as input a distributed finite element grid and a list of ordinates directions (angles). From this, we generate an independent directed dependence graph (DDG) for each ordinate. This is done by letting each vertex correspond to a finite element, and each edge correspond to a face shared between two adjacent finite elements. The edge is directed according to the direction in which the ordinate angle passes through the face. This manner of construction results in a set of DDGs that have the same set of vertices as well. A particular vertex or edge is owned by the same processor in all graphs–the distribution of all graphs comes from the distribution of the mesh across processors. The edges are directed *differently* for each DDG resulting in different graphs and consequently different SCCs. Each DDG is fully distributed across all the processors. Since all graphs are fully distributed, we gain additional parallelism by finding the SCCs in each DDG simultaneously.

Vertices on processor boundaries have access to ghost nodes, which store information about the vertex on the neighboring processor. Such information includes the processor ID of the owning processor, location of the ghost node in that processor's data structure, as well as marking and trimming status of the ghost node.

We should note that conceptually the forward and backward trims are separated as in Fig. 2 but they can be performed simultaneously. The implementation of ModifiedDCSC performs its trimming in this manner, starting from both ends and working towards the middle of each DDG.

Parallelization of the ModifiedDCSC algorithm for distributed graphs in which SCCs may span multiple processors raises a number of algorithmic and software challenges as discussed in the following subsections.

3.1 Simultaneous work on multiple problem instances

The divide-and-conquer nature of ModifiedDCSC allows us to exploit additional parallelism on multiple problem instances. There are basically two cases of this which we can make note of: (1) multiple subgraphs from each recursion, and (2) many angles which are processed in a radiation transport simulation.

First, each recursive call to ModifiedDCSC will divide every graph into subgraphs based on the results from vertex marking. Since these graphs cannot share any SCCs (Lemma 1) they are treated as independent problems. Each graph containing SCCs generates up to three recursive subproblems

as indicated in steps (9)–(11) of Fig. 2. In our parallel implementation all subproblems for all the different graphs are placed into the list of graphs for subsequent recursion.

Second, recall that in the radiation transport application, for which our code was developed, we need to search for SCCs in a set of directed graphs corresponding to different ordinates. In serial, there is no reason not to work on each graph in succession. But in parallel, solving each graph in succession is not the best method because it would add unnecessary overhead from termination detection, etc. Also, each of these graphs will partition differently due to randomization in pivot selection as well as differences in graph structures, so as recursion continues we gain parallelism by having a better overall distribution of work across processors. Searching all graphs simultaneously also reduces idle time.

In parallel, the trim and mark steps for a particular ordinate will generally enable simultaneous activity by only a subset of processors. By working on all the graphs simultaneously, we can keep more processors busy and so get improved overall performance. Our implementation continues to follow the approach sketched in Fig. 2, but now multiple graphs are being worked on at the same time. All the graphs are subjected to trimming simultaneously. Then all the searches for ancestors and descendants are performed concurrently. This complicates the code since interprocessor messages and elements in the task queue must include an indication of which graph they are associated with.

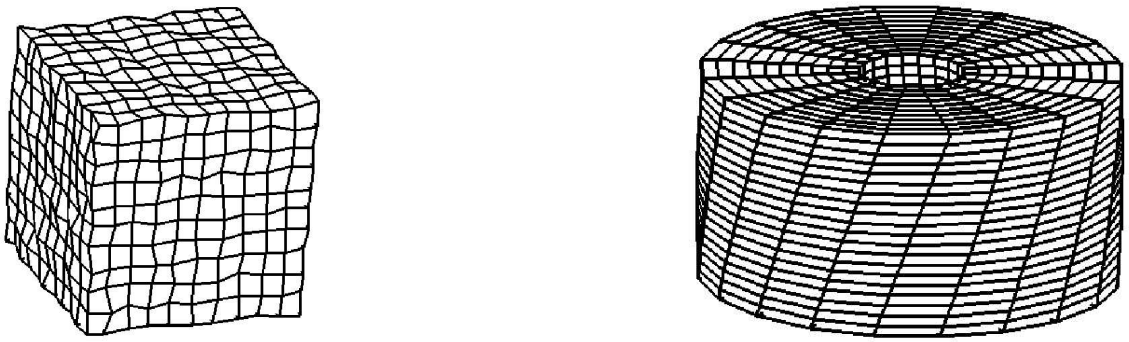
One complication of doing this is that at a given level of recursion, there can be many subgraphs in the system from each angle being treated as independent graphs. We label every graph with the two *graph id* tags. Ths first tag is used to identify the graph in its original context. The second tag is unique for every subgraph produced via recursive partitioning.

3.2 Termination detection

ModifiedDCSC is designed to run on distributed memory computers and we do not know beforehand how many vertices TRIM or MARK will visit. This complicates the trim and mark operations in several ways. First, we don't maintain a global task queue, but rather a local queue on each processor. A vertex that needs to be added to the work queue (or have its ancestor count decremented) may reside on another processor. We handle this by sending a message to the relevant processor who then marks the vertex and adds it its own local queue. This precisely mimics the parallelization of transport sweeps described by Plimpton, *et al.* [11]. The more subtle challenge is determining when a trim or mark step is completed. Just because a processor has an empty task queue doesn't mean it has no work left to do. It may yet receive a message from another processor telling it to add tasks to its queue or to decrement an ancestor count. The trim or mark operation isn't complete until all processors have empty task queues, and all messages have been received. We cannot know a-priori how many nodes will be visited during trimming or marking due to the effects of the SCCs. Because of this, we must determine when there is no more work left using a *termination detection* algorithm.

In our first implementation we used a token ring method, but this method is not scalable and has since been replaced. The current implementation of ModifiedDCSC uses a binary tree topology similar to the approach of Baker *et. al*, *et al.* [12] and requires only $O(\log P)$ time where P is the number of processors.

This binary tree implementation sets up each processor as a node in a binary tree topology. Termination only occurs when $totalsends - totalreceives = 0$ and no new work has been performed since the last check. First messages proceed up the tree, sending the ongoing count of sends and receives as well as a count of the total work for each subtree.



(a) (b)

Figure 5: Two grids used for testing the strongly connected component detection. (a) a rectangular mesh where the corner point of each cell is displaced by a random amount. (b) a hollow cylinder with the grid twisted along the vertical axis.

When the root node (processor 0) receives a message from both of its children, it compares the counts against the termination condition and it checks that the work count is unchanged. If these conditions are not met, the root node saves the work count and sends a DOWN message to each of its children.

Upon receipt of a DOWN message, each node forwards it on to their children until the DOWN messages reach the leaves. A leaf will change state from DOWN to UP and will send a message to its parent when it has no more work to do locally. This process detects termination after two passes through the tree. Since it is a binary tree, the scaling should be $O(\log P)$.

4 Experimental Results

We conducted a series of experiments to illustrate the performance characteristics of ModifiedDCSC on two different parallel architectures. The first system used is the Intel TeraFLOPS (ASCI Red) supercomputer at Sandia National Laboratories. It is a massively parallel distributed memory computer consisting of 4640 nodes with 2 Intel Pentium Pro 333 MHz processors per node, or 9280 processors in total. Each processor has 32 KB L1 and a 512 KB L2 cache and 256 MB per node. ASCI Red uses proprietary message passing hardware with 310 Mbytes/sec bandwidth and 15μsec latency.

The second system we used for gathering experimental data is the CPlant cluster at Sandia National Laboratories. The CPlant machine is a commodity cluster built with 500 MHz DEC-Alpha processors. Each processor has 256 MB RAM and uses Myrinet interconnect (100 MBytes/sec, 60μsec latency).

The input graphs are generated from finite element meshes. These meshes are made up of hexahedral cells defined by 8 corner nodes. Each cell represents a vertex in our graphs for SCC searching. An edge is inserted between two vertices if their corresponding cells share a face.

We used two geometries for our experiments, which are illustrated in Fig. 5. The first mesh is a rectangular grid which is deformed by randomly perturbing the location of the corner nodes of each cell. The magnitude of the random displacement is bounded by a specified percentage of the original inter-node distance. As the magnitude of corner displacement is increased we expect that more SCCs will be produced, and should be evenly distributed throughout the graph.

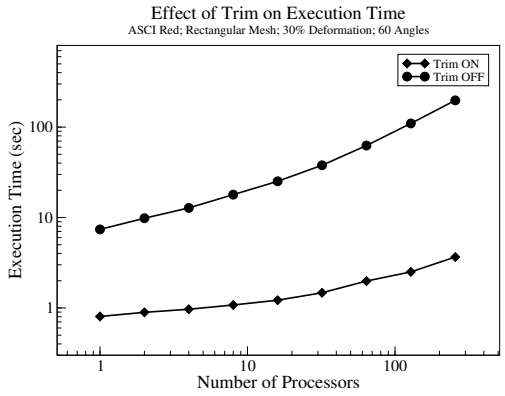


Figure 6: Effects of TRIM on execution time of ModifiedDCSC.

The second geometry is a cylindrical mesh consisting of concentric stacked rings which are then twisted to produce SCCs. Note however that the SCCs will only be generated for ordinate directions nearly parallel to the cylindrical axis. For both geometries, we maintained a scaling of 1000 vertices per graph per processor throughout all experiments. For the warped rectangle, as we doubled the number of processors we doubled the number of grid points in the x then y z direction, keeping the aspect ratio at most 2. For the cylinder, we just halved the angular separation, doubling the number of elements in each annulus.

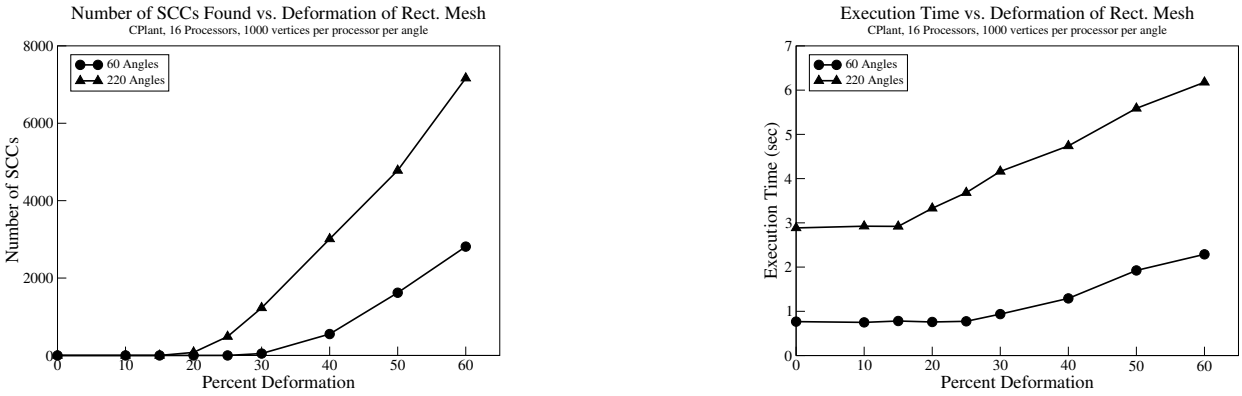
These two geometries are very different in terms of the SCCs they produce and their effects on the behavior of ModifiedDCSC. In the rectangular mesh, we found that the average SCC is small, consisting of fewer than 10 vertices. For the twisted cylinder, an SCC typically consists of thousands of vertices, usually all the vertices in a plane of the cylinder.

Both problems are scaled in terms of grid size, but the scaling properties of the number of SCCs varies. Fo the cylinder the number of SCCs remains fixed, but the size of the SCCs grows with problem size. For the rectangular problems under a fixed deformation, the size of each SCC remains the same, but the number of them grows with problem size. These two problems were selected to provide insight into how the performance of ModifiedDCSC is influenced by the input graphs. For parallel execution, all the grids were partitioned using the multilevel KL algorithm in the Chaco tool [13].

4.1 Graph Trimming

The purpose of this experiment is to show the impact of the TRIM addition to the original DCSC algorithm. With the TRIM step turned off, we essentially have the DCSC algorithm. For this experiment, we measured the execution time taken by DCSC and ModifiedDCSC to detect the SCCs for the rectangular mesh at 30% deformation.

Figure 6 shows a comparison of execution time on a rectangular mesh with trimming enabled and disabled. We can see that the addition of vertex trimming to the DCSC algorithm results in nearly an order of magnitude reduction in execution time for this mesh type. This confirms that trimming offers a significant performance improvement over the untrimmed version (DCSC) for meshes with sparse SCCs.



(a) (b)

Figure 7: Effects of mesh deformation on SCC count (a), and execution time, (b).

4.2 SCC Count

In this experiment, we studied the effect of increasing the deformation in a rectangular mesh. The number of processors was held constant at 16 for this experiment, varying only the magnitude of corner-node displacement in our meshes as a percentage of the distance to the nearest node. As illustrated in Fig. 7, the number of SCCs grows rapidly with the amount of deformation. The rightmost graph in Fig. 7, shows that the execution time also grows with the number of SCCs, albeit less dramatically than the growth in the number of SCCs. The reasons for this is that ModifiedDCSC is dependent on the *number of SCCs* since only one SCC per graph is found and removed per recursive iteration. Increasing the number of SCCs will naturally increase the number of iterations required to find them all. There is a synchronization during each recursive step of our implementation, so it follows that our execution times will increase with the number of iterations.

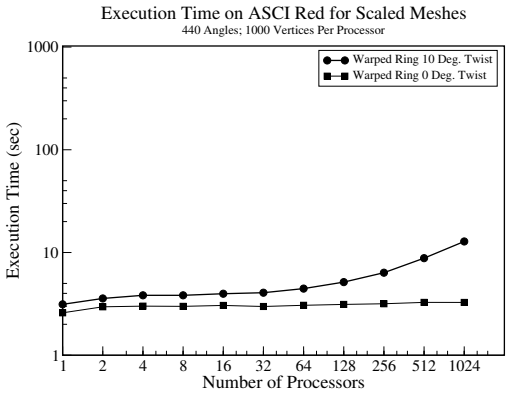
Because ModifiedDCSC partitions each graph into as many as 3 independent subgraphs with each iteration, and DCSC can only detect one SCC per graph per iteration, the number of SCCs that can be found grows exponentially with each additional iteration. Therefore, if the number of SCCs grows exponentially, we should only observe a linear growth in execution time due to additional overhead of each additional iteration.

There is another factor to consider in this example as well. Because this experiment holds the graph size constant and increases the number of SCCs present, we can say that the *density* of SCCs is increasing. By increasing the density of SCCs in G, we also reduce the effectiveness of trimming G.

4.3 Scaled Graphs

In this experiment, we investigated the behavior of ModifiedDCSC on scaled size problems on both CPlant and ASCI Red. For all tests, we set the graph size to 1000 vertices per processor. The graphs in Figures 8 and 9 illustrate the execution times we will discuss in this section.

The first graph we look at is the twisted cylinder. In this test, we scale the problem size with the number of processors, but the total number of SCCs remains constant. The two cylinders tested with 0 and 10 degrees of twist produced 0 and 40 SCCs, respectively, for all tests. As we can see in the graphs of their execution times in Fig. 8, the cylinder with zero cycles scaled very well to 1024 processors. For this problem, the code need only perform a single TRIM for each angle. When



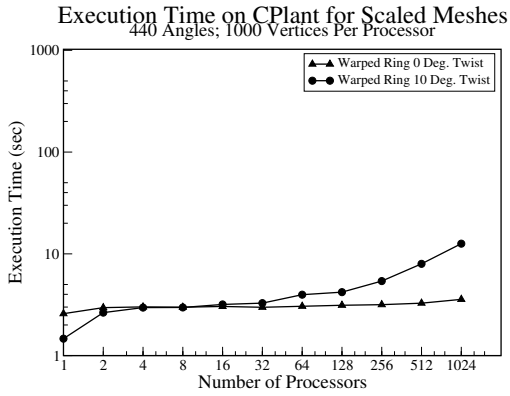
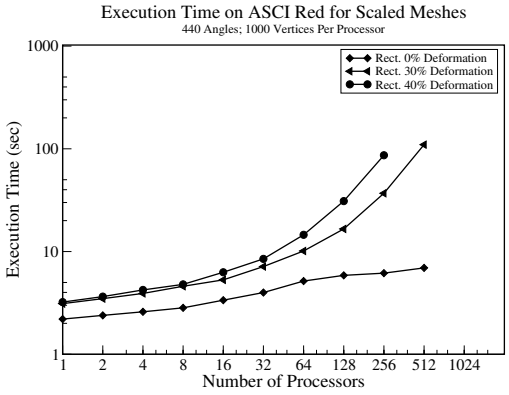


Figure 8: Execution times for the twisted cylinder meshes on ASCI Red and CPlant. Vertices scaled with processors 1000 vertices / processor. The number of SCCs is constant.



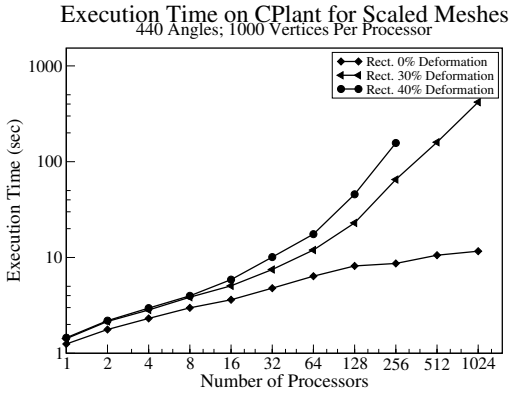


Figure 9: Execution times for the rectangular grids measured on ASCI Red and CPlant. Vertices scaled with 1000 vertices / processor. The density of SCCs is constant for a given % deformation.

we add some cycles, the execution time begins increasing noticeably around 64 processors. This is due primarily to the increase in parallel overhead. The effects of synchronization and termination detection, though minimized, are magnified by the cumulative latency of so many processors, and ModifiedDCSC is a communication-intensive application.

The second graph we performed scaled testing on is the rectangular grid mesh. We applied scalability testing to three different rectangular meshes; one with 0% deformation (no SCCs), one with 30% deformation (moderate SCC), and one with 40% deformation (more SCCs). Figure 9 shows the execution times for ModifiedDCSC to solve these graphs measured on ASCI Red and CPlant. First note that unlike the twisted cylinder timings, even the zero cycle instance shows significant runtime growth with the number of processors. For the cylinder, the different ordinates enter the geometry in different places, and so many processors can begin working immediately. For the rectangular grid, all the angles enter at one of the eight corners, and so as the number of processors grows the percentage of initially idle processors grows as well.

Second, we see that it takes much more time to find all of the SCCs for these graphs than it did for the graphs based on cylindrical meshes. This is not unexpected because the SCC *density* is constant for a particular graph, therefore the *number of SCCs* also scales with problem size. So some growth in runtime is expected from the observations in §4.2. However, the runtime here is

also effected by the parallel overhead. The combination of these two factors, along with the initial latency associated with a fixed number of initially active processors, leads to the fairly substantial runtime growth on large numbers of processors.

However it is worth noting that the radiation transport calculations that motivated our work will require many hundreds or thousands of seconds for large computations on many processors. So even this worst-case performance results in runtimes that are dominated by the physical simulation (see, for example, [14]).

5 Conclusions

We described the implementation of a new parallel algorithm, ModifiedDCSC, that finds strongly connected components in direct graphs on distributed memory computers. The traditional, serial, algorithm for finding the strongly connected components in a graph, G(V,E), is based on depth first search and has $O(|E| + |V|)$ complexity. Depth first search is difficult to parallelize, causing the need an algorithm with more available parallelism.

Special consideration was taken during development for our specific application in sweep calculations for radiation transport, though this algorithm is not limited to these graphs only.

The performance of ModifiedDCSC is greatly effected by the geometry and the number of SCCs in input graphs. Since this algorithm is dominated by communication, scalability can be limited depending on the nature of the graph that is being searched. We have shown the results from experiments on thousands of processors with reasonable scalability.

For radiation transport applications, the number of SCCs generated on any given time step is expected to be low. While in principle many SCCs can be generated cumulatively over many time steps, in practice remeshing is employed to improve the mesh geometry before the SCC count gets very large. The execution time for ModifiedDCSC has been shown to be much less than that of the numerical computation it precedes [14]

Consequently, we consider our work to be the first practical parallel implementation of an algorithm to detect strongly connected components for general graphs.

6 Acknowledgements

This work was performed at Sandia National Laboratories, a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the U.S. DOE under contract number DE-AC-94AL85000. The work has been sponsored by DOE's ASCI program.

References

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

[2] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science, Algorithms and Complexity*, pages 869–941. Jan van Leeuwen, ed., Elsevier Science Publishers B. V., 1990.

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

[4] 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, 1988.

[5] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81:334–352, 1989.

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

[7] M. Y. Kao and G. E. Shannon. Linear-processor NC algorithms for planar directed graphs II: Directed spanning trees. Technical Report DUKE–TR–1990–02, Duke University, Durham, NC, 1990.

[8] D. Bader. A practical parallel algorithm for cycle detection in partitioned digraphs. Technical Report AHPCC-TR-99-013, University of New Mexico, Albuquerque, NM, 1999.

[9] L. Fleischer, B. A. Hendrickson, and A. Pinar. *On Identifying Strongly Connected Components in Parallel* in solving irregularly structured problems in parallel. volume 1800 of *Lecture Notes in Computer Science*, pages 505–512. Springer-Verlag, 2000.

[10] Michael J. Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Computing Surveys (CSUR)*, 16(3):319–348, 1984.

[11] S. J. Plimpton, B. A. Hendrickson, S. P. Burns, and W. C. McLendon III. Parallel algorithms for radiation transport on unstructured grids. In *Proc. of SuperComputing 2000 (SC2000), Dallas, TX*, November 2000.

[12] A. H. Baker, S. Crivelli, and E. R. Jessup. An efficient parallel termination detection algorithm. Technical Report CU-CS-915-01, University of Colorado, 2001.

[13] B. Hendrickson and R. Leland. The Chaco user's guide, version 2.0. Technical Report SAND94–2692, Sandia National Laboratories, Albuquerque, NM, October 1994.

[14] S. J. Plimpton, B. A. Hendrickson, S. P. Burns, W. C. McLendon III, and L. Rauchwerger. Parallel algorithms for Sn transport on unstructured grids. *submitted to Nuclear Science and Engineering*, 2002.