A Comparison of Data-Parallel Algorithms for Connected Components

John Greiner

August 18, 1993

CMU-CS-93-191

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper presents a pragmatic comparison of three parallel algorithms for finding connected components, together with optimizations on these algorithms. Those being compared are two similar algorithms by Awerbuch and Shiloach [2] and by Shiloach and Vishkin [19] and a randomized contraction algorithm by Blelloch [7], based on algorithms by Reif [18] and Phillips [17]. Major improvements are given for the first two which significantly reduces the super-linear component of their work complexity. An improvement is also given for randomized algorithm, and this algorithm is shown to be the fastest of those tested. These comparisons are presented with NESL data-parallel code as executed on a Connection Machine 2.

This research was sponsored in part by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168, and in part by the ONR Graduate Fellowship Program.

Keywords: Computations on discrete structures, Data-parallel languages, Connected components algorithms

1. Introduction

The complexity of various PRAM algorithms has received much attention, but there has been relatively little work on the implementation and pragmatic efficiency of many of these algorithms. Moreover, much of this work has been for algorithms having regular communication patterns. More recently, attention has turned to the many common algorithms with irregular communication patterns, particularly graph algorithms having data-dependent communication.

One such problem is finding the connected components of a graph. Given a graph $G = (V, E)$, where V is a set of nodes (of size n) and E is a set of edges (of size m), the *connected components* of G are the sets of nodes such that all nodes in each set are mutually connected (reachable by some path), and no two nodes in different sets are connected. While this definition makes sense for both directed and undirected edges, the usual assumption for this problem is that edges are undirected.[1] This problem is most common in vision, to group pixels during image analysis, in physics, as part of the Swendsen-Wang algorithm for cluster identification [20], and VLSI design, for net extraction from circuit masks. For example, in vision, it is so important that some have even proposed specialized hardware for this algorithm, e.g., [23].

There has been much theoretical work on PRAM algorithms for finding the connected components of a graph, some of which are provably work-optimal. Much less work has pursued the pragmatic aspects of these algorithms. This paper compares implementations and provides optimizations of three algorithms, those of Shiloach and Vishkin [19], Awerbuch and Shiloach (A&S) [2], and a "random mate" (RM) algorithm of Blelloch [7]. The former two algorithms are quite similar and require $O(m \log n)$ work. The latter randomized algorithm uses the random mating of Reif [18], combined with the graph contraction of Phillips [17]. This algorithm is also $O(m \log n)$ work in the worst case, although for many classes of graphs, including planar graphs, it is $O(m)$ with high probability.

Obviously, there are many other algorithms that could be added to this comparison. These algorithms were chosen because of their simplicity and applicability to all classes of graphs. In contrast, the numerous algorithms in use in physics and vision typically only work on grids[2]. They also mesh stylistically with the NESL language in that they use concurrent reads and writes and are not specialized to a single communication architecture.

Two measures are used for making comparisons. Execution times on a Connection Machine 2 are given for the algorithms, using various sizes and classes of graphs. The random mate algorithm and the optimized A&S and S&V algorithms contract the graph and allow a machine-independent metric, the remaining number of edges.

The original presentation of the A&S algorithm is particularly inefficient because it doubles the size of the graph. After eliminating this inefficiency, the A&S variants generally outperform their S&V counterparts by a margin of approximately 5–10%. The remaining optimizations on these algorithms improve the algorithms by another factor of 2–3, depending on the structure of the graph. A modest optimization for random mate gives a speedup of about 5%. The random mate algorithms are theoretically superior to the S&V and A&S algorithms on some classes of graphs such as planar graphs. Furthermore, they are generally better in practice on most graphs, with the exceptions of small graphs and dense graphs (at least within the available memory size).

Code is given for the algorithms in the data-parallel style in the language NESL (Version 2.6). NESL syntax is similar to that of Standard ML, with data-parallel primitives corresponding to concurrent reads and writes.

The remainder of the introduction outlines the data-parallel paradigm and the NESL language, in particular. Section 2 describes the basic algorithms, while Section 3 describes modifications to these algorithms.

[1] Here, only the random mate-based algorithms require this assumption.

[2] The Swendsen-Wang algorithms are based on breadth-first search, which does work on all graphs.

Sections 4 and 5 describe the experiments and a summary of the results.

1.1. Data parallelism

The more commonly used models of parallelism feature multiple threads of control and are collectively known as control parallelism. Typically, a program can create an unbounded number of subprocesses communicating to each other in arbitrary patterns and each using different information such as separate control stacks, program counters, and local data. This flexibility can complicate programming beyond comprehension and lead to problems when debugging.

In contrast, data parallelism limits the programmer to a model of a single thread of control. The parallelism is constrained to replicating the thread of control over a collection of data. For example, two k-sequences of data would be stored, at least conceptually, so that the each of the corresponding elements of the two sequences are placed on one of k (virtual) processors. A function can then be mapped over the collection so that each processor performs the function on its local data. Applications are assumed to have collections of data large enough for the bulk of a program's work to be encapsulated in such parallel computations. Conventional uniprocessor programming idioms adapt easily to this restricted model, and many parallel algorithms are naturally written in this style.

1.2. NESL

NESL is a strongly typed, functional, data-parallel language developed under the direction of Blelloch. Its only parallelizable data collection type is the sequence, and it features efficient implementation of nested sequences. Syntactically, it resembles Standard ML, and it uses a similar polymorphic type inference system. Like many other functional languages, it has no primitive looping construct. Instead, recursion is used to implement loops, and uses of "tail recursion" are compiled into the equivalent iterative code using jumps, rather than procedure calls.

Any function may be mapped element-wise over a sequence, and it provides a fixed set of scan operations (also known as prefix sums) and arbitrary reorderings of sequences for communication. The primary communication constructs are

- `seq -> ind`: Returns the values of the sequence at the indicated indices. Any given index may occur more than once in the sequence of indices, corresponding to a concurrent read of the corresponding value.
- `seq <- ind_val`: Each element of sequence `ind_val` is an pair of an index and value. Returns the sequence that is like `seq` except that the given values are placed at the corresponding indices. Any given index may occur multiple times, corresponding to a concurrent write.
- {`exp : id`$_1$ `in seq`$_1$`; ... | cond`}: This syntax is based on standard set notation. In turn, bind the identifiers to each value in the corresponding sequences. Evaluate the expression for each set of bindings which satisfies the condition, and return a (packed) sequence of the results.

Implementations of NESL on hardware without concurrent reads and writes (CRCW) must simulate these features in software. For a more detailed description of the language, see [6].

2. Previous Parallel Algorithms

This section outlines the three algorithms from which refinements were made. For more detailed explanations, refer to the original papers as cited. The NESL implementations of these algorithms are given in Appendix A.

The first two algorithms are based on forming and combining trees of nodes, such that all nodes in a given tree belong to the same connected component. These algorithms combine trees to find the maximal such trees. The roots serve as representative elements of the trees, and the algorithms return the sequence of the roots corresponding to each node.

The trees are represented by a sequence of the parent of each node. There are two basic operations, *hooking* and *shortcutting* on trees, as diagrammed in Figure 1. Hooking combines pairs of trees to form larger trees if there is an edge between the two trees. Shortcutting flattens trees to improve the amortized efficiency of hooking. When neither operation can be applied, all trees are of depth one, *stars*, and the trees correspond to the maximal connected components. If shortcutting is performed often enough and hooking is done as to avoid cycles, the algorithms require $O(\log n)$ hooking steps, each of $O(m)$ work, so that the algorithms require $O(m \log n)$ work.

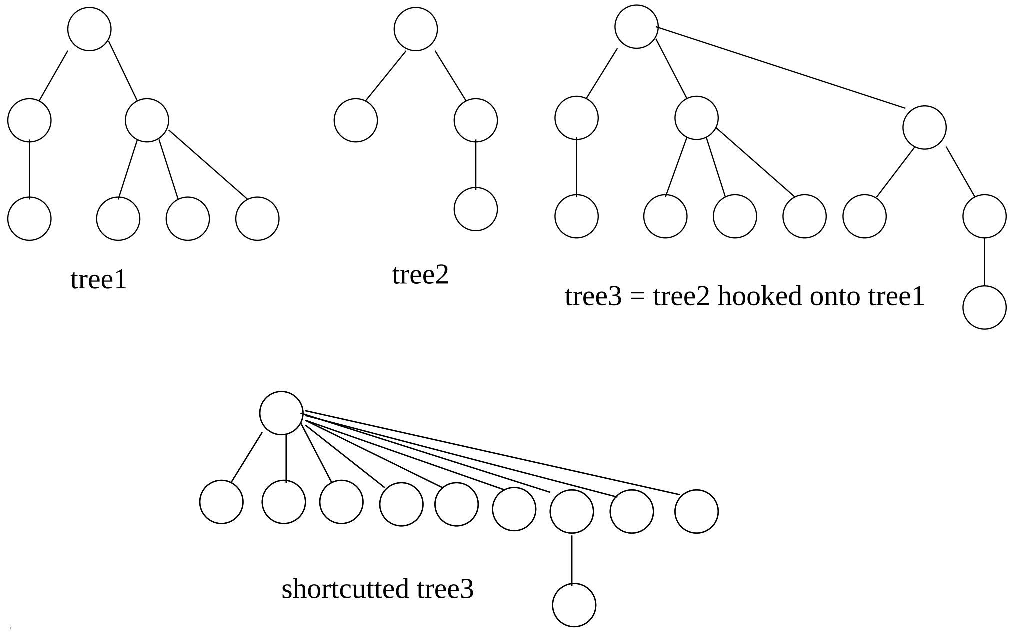

Figure 1: Hooking and shortcutting.

The third algorithm contracts the graph by combining nodes and edges such that the connected components of the new graph are the same as those of the original. The graph is contracted until no edges are left, so the remaining nodes correspond one-to-one to the connected components. Additional information is saved to compute to which connected component belongs each of the original nodes. It requires $O(\log m)$ iterations, each of $O(m_i)$ work, where m_i is the number of edges in the graph remaining on the i^{th} iteration. The total work complexity is $O(m)$ if the ratio of edges to nodes in within a certain range. As Reif and Gazit show, all other graphs can be transformed into the appropriate class in $O(m)$ work.

2.1. Shiloach and Vishkin

The algorithm of Shiloach and Vishkin [19] uses several data structures to represent the trees of nodes: the parent relation of the tree, the parent relation from the previous iteration of the algorithm, and a sequence indicating which iteration each node was last named a parent of some other node. (The cryptic name `qs` for this last sequence is taken directly from S&V.) The n nodes are named by the integers $0 \ldots n-1$. The parent relation is then a sequence of integers, where the i^{th} element of the sequence is the parent of node i.

The first step of each iteration shortcuts the trees and initializes the data structure `qs` for the iteration.

Next, two different hooking steps are used. *Conditional* hooking combines two trees so that the larger numbered root is below the smaller. *Unconditional* hooking only hooks *stagnant* trees onto other trees. A tree is stagnant if it has not been involved in shortcutting or conditional hooking on this iteration. The latter kind of hooking is necessary to avoid a worst case of $n-1$ iterations, as fully described by S&V. It is this test for a root being stagnant which uses the third piece of information encoding the tree.

A second shortcutting step simplifies the complexity analysis given by S&V. While it improves performance, its use is not necessary to result in $O(m \log n)$ work.

The algorithm terminates if no node changes were made to the trees, as indicated by the `qs` sequence. For the sake of clarity, the given code calculates the termination condition slightly differently than in S&V.

2.2. Awerbuch and Shiloach

The algorithm of Awerbuch and Shiloach [2] is a simplification of that of Shiloach and Vishkin. In particular, unconditional hooking is simplified so that instead of hooking stagnant trees onto other trees, only stars can be hooked onto trees. The advantage is that testing for membership in a star can be done without calculating the extra data structure `qs` of S&V. Instead, the test uses only properties of the parent relation. On the other hand, the new star membership test is relatively expensive because of communication costs. So, the rooted tree is represented by a single parent relation.

However, as argued by A&S, for the algorithm's invariants to hold on the first iteration, an extra n "dummy" nodes and n edges are added to the graph. These edges connect the i^{th} original node with the i^{th} dummy node.

Also, the optional shortcut is eliminated (presumably for simplicity). The `AS_starcheck` routine is also used for termination of the algorithm: it halts when all nodes are members of stars. At that point, the parent of each node is the root of its connected component. Thus, the resulting control structure loops over the two forms of hooking, shortcutting, and testing for the termination condition.

2.3. Random Mate

The random mate algorithm was originally an adaptation by Reif [18] of the S&V algorithm, replacing both kinds of hooking with a single randomized version, called mating. In this step, each node is randomly assigned one of two labels, plus or minus, with equal probability. Edges from positive to negative nodes are selected, with the restriction that only one edge may be selected pointing from any given node. This restriction is implemented via an implicit concurrent write which arbitrarily picks a single target for the node.

This algorithm by Blelloch combines mating with the graph contraction of Phillips [17], so that each successive iteration works with a smaller graph. The edges are contracted with the selected, or *active* nodes, producing supernodes. The edges are contracted by renaming with the new supernodes and removing self-edges, although because of conflicts, not necessarily all of the active edges are used for contraction. Thus, these edges correspond to the parent relation of the previous algorithm. After the graph has been fully contracted, the remaining nodes represent the connected components of the original graph, and correspond to the roots of the trees formed in the previous algorithm. Figure 2 represents one of these iterations.

Next, the graph must be re-expanded, using the active edges, to propagate the name of these final supernodes to the nodes of the original graph. For this purpose, the active edges of each iteration are placed on the run-time recursion stack.

The implementation of the algorithm is given in Appendix A. The nodes of the graph are represented by the endpoints of the edges. As mentioned, the algorithm is recursive, so that the active edges are placed on the stack for use during expansion. The graph is expanded as the recursion stack unwinds, and the

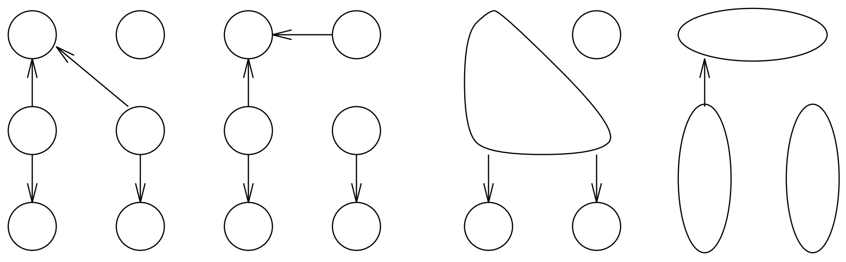

Figure 2: One iteration of contraction.

supernode relation returned from recursive calls and the active edges are used to propagate the name of each root to all nodes in its component.

An unconventional feature of this version of partitioning (`RM_partition`) is that the mating is not truly random. The "randomness" is generated by using on the i^{th} iteration the $(i \bmod \log_2 n)^{\text{th}}$ bit of the (arbitrary) node numbers. A true pseudo-random alternative (RM3) is given in Appendix B, but experiments indicate the given code to be better in practice because partitioning with this method requires much less time-consuming communication. Furthermore, it produces partitions with similar numbers of active edges, except that the randomized version typically finds larger partitions on very sparse graphs.

3. Modifications

All of the new algorithms are modifications of the previous three. Major changes are made to the A&S and S&V algorithms, drastically reducing the constant on the $c(m \log n)$ term of the $O(m \log n)$ complexity. A modest improvement is also given for random mate.

3.1. Shiloach and Vishkin-based

The following changes are made to the original algorithm (SV1) and are further described in this section.

- Shortcutting more aggressively. (SV2)
- Using unconditional hooking less often. (SV3)
- Contracting the edges of the graph, as in random mate. (SV4)

For simplicity, each algorithm includes all previous optimizations, so that, for example, SV4 uses all of these modifications.

To further reduce the depth of the trees, extra shortcutting may be performed each iteration. Flatter trees allow the termination condition to be detected earlier. For a given (finite) tree, only a finite amount of shortcutting is useful, until a fixed point is found. The given heuristic closely estimates the number of shortcuts needed to reach this point.

An alternative is to guarantee that the maximal amount of shortcutting is performed. That can be done by repeatedly shortcutting until the operation does not further change the graph, as in `shortcut_max`. In practice, however, the improvement resulting from the graph contracting more quickly is more than offset by the higher cost incurred by testing whether the shortcut operation modified the graph.[3]

Unconditional hooking is only necessary in a small percentage of cases. Empirical evidence suggests that a relatively small number of edges are ever used by the step. Only executing the step occasionally (here, every third iteration) improves performance, while still avoiding the need for a linear number of iterations. Also, since the number of live edges is by far the greatest during early iterations, it is best to avoid using the step then.

The next modification is an adaptation from the random mate algorithm. On each iteration, the live edges are replaced by renaming the endpoint with the parents of the endpoints, and then eliminating self-edges. In this case, aggressive shortcutting is especially beneficial since flatter trees result in more edges being contracted.

Since a node's parent is in the same connected component as the node, if there were a path between two nodes using the old edges, there is still a path between the nodes using the new live edges and the parent relation. Thus, all information necessary for finding the connected components remains. Even though the number of live edges monotonically decreases, the complexity of each iteration is still bounded by the number of nodes, because of the shortcutting operations.

However, this modification is only an improvement for some classes of graphs. In particular, it is not beneficial if the number of edges in the graph is much larger than the number of nodes (e.g., $m \approx n^2$). Since $O(n)$ edges and nodes are eliminated per iteration, in this case a proportionally small fraction of the edges is being removed, and the cost of the operation overshadows the benefits.

Additionally, if there are no live edges left, it is clear that further iterations of the algorithm perform only shortcutting, so a special case is made of this to avoid overhead on the last iterations.

For brevity, these changes are grouped together in the presentation, as shown in the following code for the main loop. However, each is independently useful.

```
function SV_alg4(ps,qs,es,iter) =
if zerop(#es) then shortcut_max(ps)
else let (ps1,qs1) = SV_init(ps,qs,iter);
         (ps2,qs2) = SV_cond_hook(ps1,ps,qs1,es,iter);
         ps3       = if uncond_hookp(iter) then SV_uncond_hook(ps2,qs2,es,iter) else ps2;
     in if not(any({q == iter : q in qs2})) then ps3
        else let ps4 = shortcut_n(ps3,shortcut_heuristic(#es));
             in SV_alg4(ps4,qs2,shrink_edges(ps4,es),1+iter) $
```

3.2. Awerbuch and Shiloach-based

The following changes are made to the original algorithm (AS1) and are described further in this section.

- Modifying the first iteration, so that dummy nodes and edges are unnecessary. (AS2)
- Optimizing detection of the termination condition. (This optimization is later made redundant by the final modification.) (AS3)
- Shortcutting more aggressively. (AS4)
- Using unconditional hooking less often. (AS5)

[3] On the other hand, by guaranteeing that all trees are stars, further optimizations could be made. One precondition of conditional hooking is trivially satisfied, and unconditional hooking is entirely unnecessary. This is further pursued in [10].

- Contracting the edges of the graph, as in random mate. (AS6)

The most glaring efficiency problem with the original presentation is the addition of dummy nodes and edges, effectively doubling the size of the graph. These nodes and edges are used only on the first iteration to establish the tree structure expected by the hooking steps. After the first iteration, they will always be at the bottom of the trees and be irrelevant. In order to eliminate these dummy nodes and edges, one can use specialized versions of the hooking steps (The functions `AS_lone_cond_hook` and `AS_lone_uncond_hook` in Appendix B.) on the first iteration.

Another bottleneck is the star membership test, which is relatively expensive. As shown in the code below, its use as a test for termination of the main loop can be specialized to `AS_starcheck_all`, which eliminates most of the communication costs of `AS_starcheck`.

```
% Equivalent to all(AS_starcheck(ps)), but faster. %
function AS_starcheck_all(ps) = all({p == gp : p in ps; gp in shortcut(ps)}) $
```

The remaining modifications are the same as made in Section 3.1 for the similar S&V algorithm. The main loop of the resulting algorithm is shown below.

```
function AS_alg6(ps,es,iter) =
if zerop(#es) then shortcut_max(ps)
else let ps1 = AS_cond_hook(ps,es);
         ps2 = if uncond_hookp(iter) then AS_uncond_hook(ps1,es) else ps1;
         ps3 = shortcut_n(ps2,shortcut_heuristic(#es));
         es1 = shrink_edges(ps3,es);
     in AS_alg6(ps3,es1,1+iter) $
```

3.3. Random Mate-based

The one optimization of random mate is to ensure that each iteration has a non-zero number of active edges so that the algorithm does not loop through the entire `RM_reduce_graph` routine without the graph changing, as in the following function.

```
function RM_active_edges2(es,bits,step) =
let aes     = {e : e in es; active in RM_partition(es,step) | active};
    newstep = rem(step+1,bits);
in if zerop(#aes) then RM_active_edges2(es,bits,newstep)
   else (flip_edges(aes,{nthbit(from,step) : from in edges_froms(aes)}),newstep) $
```

A more general test would require that a "significant" number of active edges be selected in order to use the partition. But then the algorithm sometimes discards many partitions until one is used, and in practice, this did not improve the algorithm.

4. Testing Method

To test the performance and the algorithms, four different classes of graphs were used. Test runs used subsets of these classes of graphs generated by randomly choosing a uniformly distributed fraction of each graph's edges.

- Subsets of two-dimensional toroidal grids: Each vertex has a subset of the four neighbors of such a grid.
- Subsets of three-dimensional toroidal grids: Each vertex has a subset of the six neighbors of such a grid.
- "Tertiary" graphs: Each vertex has three neighbors picked uniformly at random.
- Subsets of complete graphs: Each vertex is connected to a subset of all other vertices. To some degree, these represent the general case.

Grid-based graphs are commonly used in both vision and physics. Subsets of complete graphs ("random graphs") represent the most general, and frequently worst, case. Tertiary graphs are a representative intermediate case.

For the grid-based graphs, two different fractions of edges were used, resulting in graphs which are or are not highly connected. Graphs having more (less) than two edges per vertex are (not) highly connected, since for the graph to be fully connected, each vertex must have at least two edges. So, for 2D grids, using a random subset of more than half of the edges will result in a relatively highly connected graph. The testing here uses subsets of 30% and 60% of the edges. Similarly, for 3D grids, we choose fractions less and greater than one third: 20% and 40%. For complete graphs, fixed fractional subsets are again used. However, since the number of edges increases quadratically, larger graphs are increasingly connected.

We now define some standard terms of graph theory. These properties of graphs will effect the performance of the algorithms and allow us to explain our results.

The *degree* of vertices in the graph is the number of incident edges at each vertex and is a measure of the connectivity of the graph. Vertices in two-dimensional grids have a degree of four; three-dimensional grids, six; tertiary graphs, at most six; and random graphs, up to n.

An *edge separator* of a graph is a set of edges which, if removed, will separate the graph into independent subgraphs of approximately the same size. The size of the separators of a graph is another measure of connectivity. The divide-and-conquer strategy of random mate tends to perform well on graphs with small separators. Two-dimensional grids have separators of size $O(\sqrt{n})$; three-dimensional grids, $O(\sqrt[3/2]{n})$; tertiary graphs, $O(n)$; and random graphs, $O(n)$.

The *diameter* of a graph is the length of the longest of the shortest paths between all vertices in the graph. A large diameter indicates that the trees of the algorithms will be deep, so that the effects of shortcutting will be more significant. Two-dimensional grids have diameters of size $O(\sqrt{n})$; three-dimensional grids, $O(\sqrt[3]{n})$. Tertiary and random graphs typically have much smaller diameters, e.g., the expected size for tertiary graphs is $O(\log n)$.

Recall that the A&S and S&V algorithms assume that each edge is listed twice, pointed in each direction, whereas the random mate algorithms need only one copy of each edge. So, the former algorithms must use twice as many edges to represent the same graph.

The NESL code was executed[4] on one quarter of a 32K processor Connection Machine 2, i.e., 8K processors each with 32KB of local memory per processor. Preliminary timings obtained on a Cray Y-MP have entirely similar relative results.

[4] NESL is currently compiled to VCODE which is then interpreted.

5. Experimental Results

The following plots compare the performance of the algorithms on such graphs. Most plots display average running times of several algorithms for graphs, ranging in size upto as bounded by the available memory. Execution times are taken as the average over ten trials each, whereas edge and node counts are taken from single trials.

Figures 3 and 4 show the percentage of the original edges that remain after each iteration of the optimized A&S and RM algorithms. Naturally, this uses the version of A&S which does contract the edges. These plots use the largest graphs allowed in the available memory, although smaller graphs produced similar results.

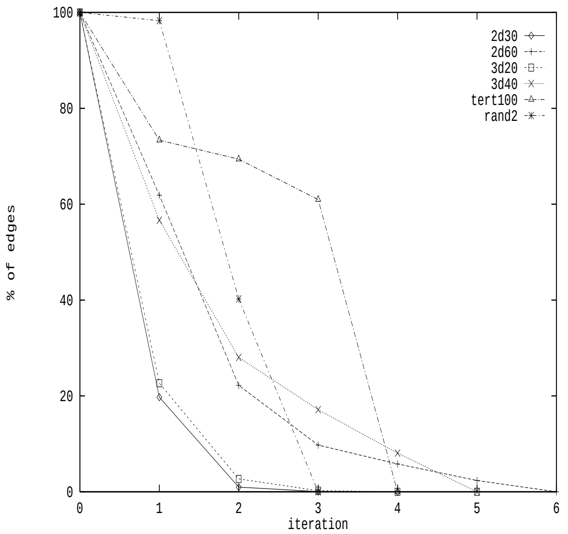

Figure 3: Percent of original edges remaining after each iteration of AS6

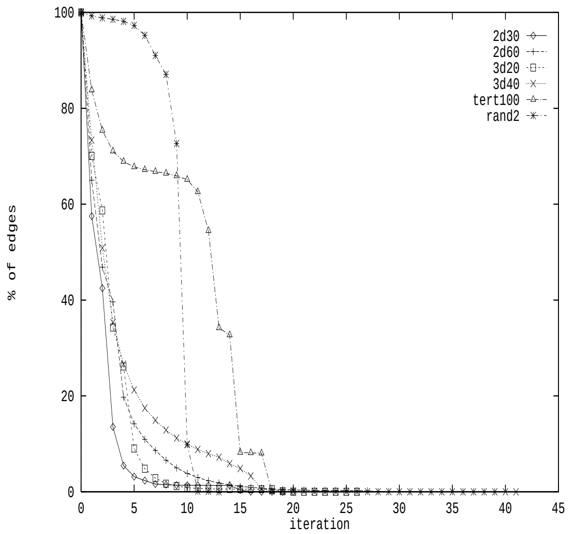

Figure 4: Percent of original edges remaining after each iteration of RM2

For tertiary, and especially random graphs, the random mate algorithm uses relatively few iterations to terminate, but initially contracts the graph very little. Thus, these few iterations are relatively expensive. For the grid-based graphs, the early contraction is very quick, but many iterations are needed to eliminate the remaining edges, particularly for the more highly connected graphs.

On average, half of the remaining edges are active on each iteration of random mate. As a result, between a quarter and a half of the remaining non-singleton nodes are removed each iteration, depending on the class of graph. And as shown by [17], planar graphs have at most a constant multiple more edges than nodes. And since random mate contracts planar graphs into planar graphs, the number of edges decreases at a similar rate to that of the nodes. This plot empirically confirms that fact, and indicates that the same likely holds for three-dimensional grids.

For random graphs, again about the same number of edges as nodes are contracted during the early iterations. But, this is only a small fraction of the number of edges, which is initially proportional to the *square* of the initial number of nodes. Thus during contraction, the graph becomes increasingly dense until it is almost fully connected.[5] But, the the number of remaining edges is bounded by the square of the number of remaining nodes. This upper bound now becomes relevant, and the the edges quickly contract. For tertiary graphs, a similar phenomenon is seen, except that since the initial number of edges is only a constant multiple of the initial number of nodes, the early iterations contract a greater fraction of the edges.

[5] A similar mating algorithm is used by Gazit [8] to transform sparse graphs into dense graphs.

The space complexity of random mate is dominated by the space needed for storing the active edges on the stack.[6] With high probability, this is proportional to the sum over all iterations of the number of remaining graph edges. For grid-based graphs, the geometric decrease in the number of edges indicates that space complexity is a constant multiple of the number of edges. In general, it is at least bounded by $O(m \log m)$, the size of the edges multiplied by the number of iterations, although a tighter bound might be provable. Compare this to the lower space complexity $O(m)$ of the tree-based algorithms. The total number of active edges stored could be bounded by m by only saving those active edges used for contraction.

The plot for the optimized A&S algorithm is very similar. However, note that it uses a much smaller number of iterations, partly because each iterations performs several shortcut operations.

Figures 5 and 6 compare the optimized algorithms to each other on the toroidal grids. The former compares the optimized S&V, A&S, and RM algorithms on two dimensional grids, using 30% of the edges; it also compares the same A&S and RM algorithms using 60% of the edges. The latter compares these algorithms on the three dimensional grids using 20% and 40% of the graph edges.

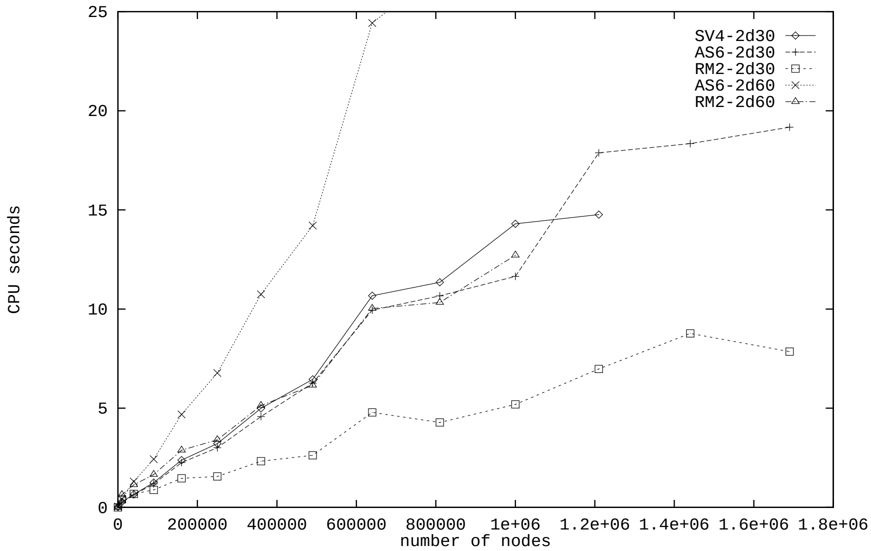

Figure 5: Optimized algorithms on 2D grids, 30% and 60%

Not surprisingly, the similar S&V and A&S algorithms result in very similar running times, although the latter is up to 23% faster on the graphs tested here. Random mate outperforms both of the other algorithms on all but the smallest of grid-based graphs. Within the range of sizes shown here, RM is up to 288% faster than A&S. Since random mate has a better expected work complexity for these graphs, this comparative advantage grows with graph size.

Figure 7 again compares the optimized S&V and A&S algorithms, as well as *all* of the RM algorithms on "random" graphs. Here, 2% of the edges of the complete graphs are used. Recall that RM3 uses the pseudo-random partitioning, which is clearly very costly on these graphs. In fact, this holds for all graphs tested. While random mate is still faster than both A&S and S&V, its advantage is slimmer than with the grids. Random mate is consistenly about 50% faster than A&S.

Similarly, Figure 8 uses tertiary graphs to compare all of the A&S algorithms described. Each of the first five algorithms consistently outperforms the previous algorithms. While not plotted here, this also holds for the other classes of graphs, so that each of the corresponding modifications is indeed an optimization. However, the final modification, that of contracting the edges of the graph, is obviously not beneficial is this

[6] For simplicity, we are here assuming that $n < m$. In general, n should be added to each of these space complexities.

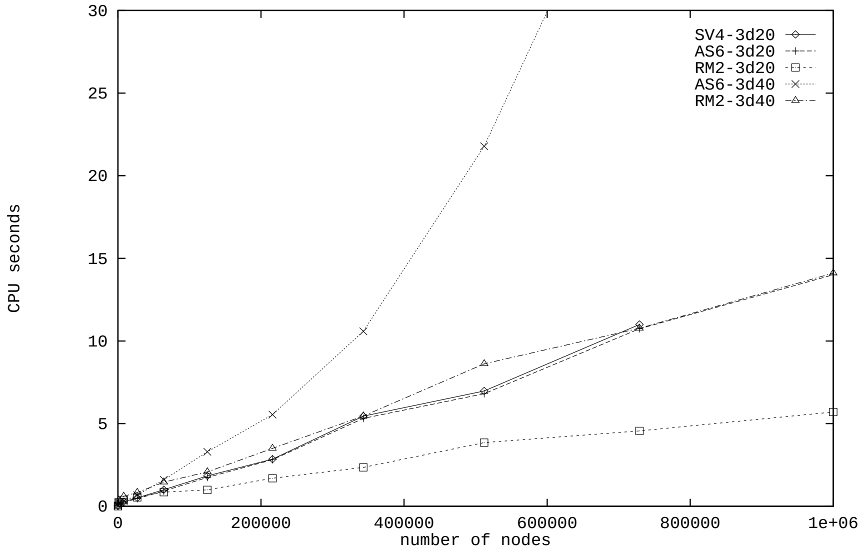

Figure 6: Optimized algorithms on 3D grids, 20% and 40%

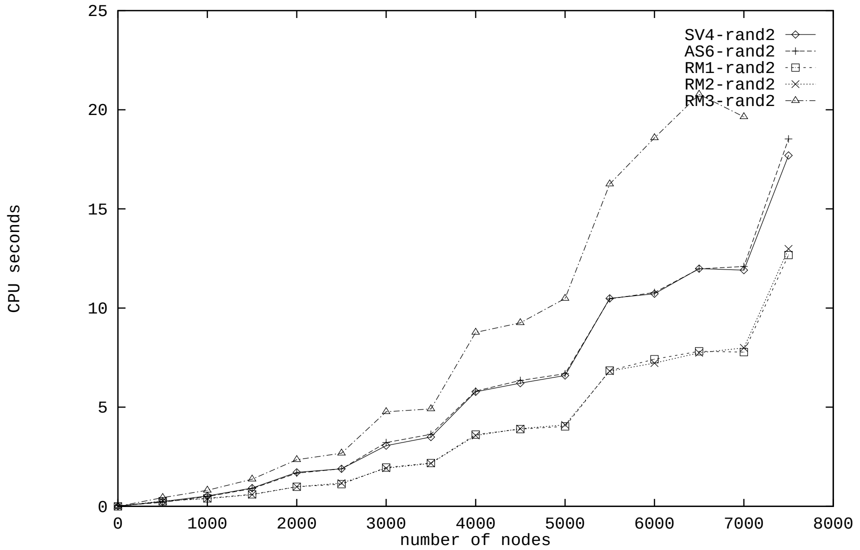

Figure 7: Optimized algorithms and all RM algorithms on random graphs, 2%

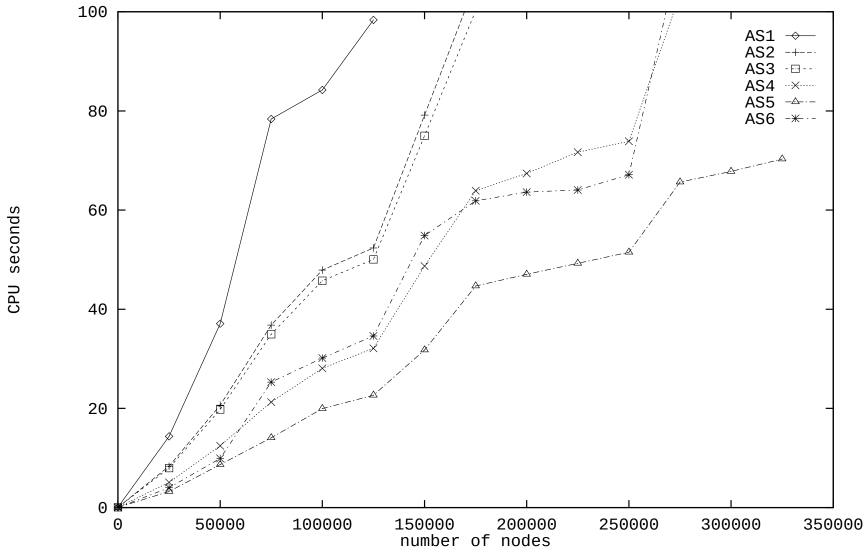

Figure 8: All A&S algorithms on tertiary graphs

case. As previously discussed, contracting the edges is not cost-effective for the relatively dense tertiary and random graphs, while it is an improvement for the grid-based graphs.

6. Conclusions and Future Work

Previous work on parallel algorithms for connected components has concentrated on theory and largely ignore pragmatics. This paper has investigated implementations of algorithms by Awerbuch and Shiloach, Shiloach and Vishkin, and Blelloch. We have shown that the published versions of the former two algorithms are inefficient, as compared to the latter.

But, several modifications have been presented to significantly improve both the A&S and S&V algorithms by constants factors, with a overall speedup factor of approximately five for A&S. Two different optimized A&S algorithms are given, such that one (AS5) is better for the dense tertiary and random graphs, and the other (AS6) is better for the grid-based graphs. Nevertheless, the random mate algorithm is faster than all of the other algorithms tested here, for all but the smallest of graphs.

For a more detailed analysis, accurate cost models of the algorithms should be developed. In particular, this would allow a theoretical basis for improving the several heuristics used.

While the edge-contracting modification to the S&V and A&S algorithms is adapted from random mate, further combining of the algorithms might be useful. For example, the more expensive pseudo-random partitioning could be used only on the final iterations of random mate, when its higher cost may be offset by the better partitions it generates then. Or, iterations of random mate and A&S could be interleaved to combine strengths. Bounding the maximum number of A&S iterations would retain the $O(m)$ work complexity of random mate. Gazit [8] uses one such combination, by using a mating algorithm to preprocess sparse graphs, before using an algorithm based on S&V.

One such hybrid algorithm has been implemented which incorporates both shortcutting and graph contraction. Results indicate that it consistently outperforms all algorithms tested here [10].

Another possible modification for random mate, suggested by Dafna Talmor, addresses the worst case of random mate of many active edges pointing *to* a single node. On each iteration, the active edges would be selected, and the edges contracted as presently done, which would only use one edge in this worst case. Next, those unused active edges would be flipped and serve as the active edges for a second contraction.

Acknowledgements

Thanks go to Guy Blelloch for lots of help with the algorithms and NESL, and Jay Sippelstein for help with NESL.

A Code of original algorithms

The following are common routines used by the algorithms.

```
function edges_froms(es) = {from : (from,to) in es} $
function edges_tos(es) = {to : (from,to) in es} $


function parents_edges(ps,es) =
{(pfrom,pto) : pfrom in ps -> edges_froms(es); pto in ps -> edges_tos(es)} $


function shrink_edges(ps,es) =
{(pfrom,pto) : (pfrom,pto) in parents_edges(ps,es) | pfrom /= pto} $


% Convert edges from undirected to directed %
function direct_edges(es) = es ++ flip_edges(es,{t : es}) $


function flip_edges(es,flips) =
{(select(flip,to,from),select(flip,from,to)) : (from,to) in es; flip in flips} $


function shortcut(ps) = ps -> ps $
```

The following is the original S&V algorithm.

```
function SV_init(ps,qs,iter) =
let gps = shortcut(ps) in (gps,qs <- {(gp,iter) : gp in gps; p in ps | gp /= p}) $

function SV_cond_hook(newps,ps,qs,es,iter) =
let newp_es1 = parents_edges(newps,es);
    newp_es2 = {(newpfrom,newpto) : (newpfrom,newpto) in newp_es1;
                                    pfrom in ps -> edges_froms(es)
                                  | (newpfrom == pfrom) and
                                    (newpto < newpfrom)};
in (newps <- newp_es2,qs <- {(newpto,iter) : newpto in edges_tos(newp_es2)}) $


function SV_stagnantp(p,gp,qp,iter) = (p == gp) and (qp < iter) $

function SV_uncond_hook(ps,qs,es,iter) =
```

```
let pes = parents_edges(ps,es);
in ps <- {(pfrom,pto) : (pfrom,pto) in pes;
                        gpfrom in ps -> edges_froms(pes);
                        qpfrom in qs -> edges_froms(pes)
                       | SV_stagnantp(pfrom,gpfrom,qpfrom,iter) and (pfrom /= pto)} $


function SV_alg1(ps,qs,es,iter) =
let (ps1,qs1) = SV_init(ps,qs,iter);
    (ps2,qs2) = SV_cond_hook(ps1,ps,qs1,es,iter);
    ps3       = SV_uncond_hook(ps2,qs2,es,iter);
in if not(any({q == iter : q in qs2})) then ps3
   else SV_alg1(shortcut(ps3),qs2,es,1+iter) $


% find connected components of graph using S&V's alg. %
function cc_SV1(es,num_ns) =
SV_alg1(index(num_ns),dist(0,num_ns),direct_edges(es),0) $
```

The following is the original A&S algorithm. Included in the comments of the provided code are Awerbuch and Shiloach's own descriptions.[7]

```
% If G(i) = D(i) and D(i) > D(j) then D(D(i)) := D(j) %
function AS_cond_hook(ps,es) =
let pes = parents_edges(ps,es);
in ps <- {(pfrom,pto) : (pfrom,pto) in pes; gpfrom in ps -> edges_froms(pes)
                       | (gpfrom == pfrom) and (pfrom > pto)} $

% ST(i) := TRUE;  If D(i) \= G(i) then ST(i),ST(G(i)) := FALSE; ST(i) := ST(G(i)) %
function AS_starcheck(ps) =
let gps = shortcut(ps);
    sts = {p == gp : p in ps; gp in gps} <- {(gp,f) : p in ps; gp in gps | p /= gp};
in sts -> gps $

% If i belongs to a star and D(i) /= D(j) then D(D(i)) := D(j) %
function AS_uncond_hook(ps,es) =
ps <- {(pfrom,pto) : (pfrom,pto) in parents_edges(ps,es);
                     instarp in AS_starcheck(ps) -> edges_froms(es)
                   | instarp and (pfrom /= pto)} $

function AS_alg1(ps,es,iter) =
let ps1 = AS_cond_hook(ps,es);
    ps2 = AS_uncond_hook(ps1,es);
in if all(AS_starcheck(ps2)) then ps2 else AS_alg1(shortcut(ps2),es,1+iter) $


% For all nodes i, add node i' (= i + num_ns) and add edge (i,i') %
function add_dummy_nodes(es,num_ns) =
(es ++ {(n,n + num_ns) : n in index(num_ns)},num_ns + num_ns) $


function remove_dummy_nodes(ps) = take(ps,#ps / 2) $
```

[7] They use the naming scheme of `D(i)` as the parent of the source node of the unnamed edge, and `G(j)` as the grandparent of the edge's target node.

```
function cc_AS1(es,num_ns) =
let (newedges,newnum_ns) = add_dummy_nodes(es,num_ns);
in remove_dummy_nodes(AS_alg1(index(newnum_ns),direct_edges(newedges),0)) $
```

And the following is the code for the original random mate algorithm.

```
function RM_reduce_graph1(ns,es,bits,step) =
if zerop(#es) then ns
else let % contraction %
         aes       = RM_active_edges1(es,step);
         new_ns    = ns <- aes;
         newedges  = shrink_edges(new_ns,es);
         old_roots = RM_reduce_graph1(new_ns,newedges,bits,rem(step+1,bits));
     in  % Compute new roots -- expansion %
        old_roots <- {(afrom,v) : afrom in edges_froms(aes);
                                  v in old_roots -> edges_tos(aes)} $


function RM_partition(es,step) =
{nthbit(from,step) xor nthbit(to,step) : (from,to) in es} $

function RM_active_edges1(es,step) =
let aes = {e : e in es; active in RM_partition(es,step) | active};
in flip_edges(aes,{nthbit(from,step) : from in edges_froms(aes)}) $


function nthbit(n,bit) = zerop(lshift(1,bit) and n) $


% Find the connected components by reduce_graph %
function cc_RM1(es,num_ns) =
if plusp(num_ns)
then RM_reduce_graph1(index(num_ns),es,trunc(log(float(num_ns),2.0)) + 1,0)
else [] int $
```

B Supplementary Modifications Code

The following is supplementary NESL code for the modifications to the algorithms.

```
function shortcut_n(ps,n) =
if n <= 0 then ps else shortcut_n(shortcut(ps),n -1) $

function shortcut_max(ps) =
let gps = shortcut_n(ps,4);
in if all({p == gp : p in ps; gp in gps}) then gps else shortcut_max(gps) $

% Heuristically estimate number of shortcuts until only stars left %
function shortcut_heuristic(numedges) =
if zerop(numedges) then 1 else min(1,trunc(log(float(numedges),10.0)) - 1)  $


% Test if should do uncond_hook this iteration
  uncond_hook expensive on early iterations %
function uncond_hookp(iter) = zerop(rem(1+iter,3)) $
```

The following functions are for the A&S algorithms.

```
function cc_AS6(es,num_ns) =
let ps  = index(num_ns);
    es  = direct_edges(es);
    ps1 = AS_lone_cond_hook(ps,es);
    ps3 = shortcut_n(ps1,shortcut_heuristic(#es));
    es1 = shrink_edges(ps3,es);
in AS_alg6(ps3,es1,0) $

% Starchecking for 1st iter. IF no dummy nodes %
function AS_lonecheck(ps) =
let ns = index(#ps);
in {p == n : p in ps; n in ns} <- {(p,f) : p in ps; n in ns | p == n} $

% If G(i) = D(i) and D(i) > D(j) then D(D(i)) := D(j)
  Use 1st iter. IF no dummy nodes, when G(i) = D(i). %
function AS_lone_cond_hook(ps,es) =
ps <- {(pfrom,pto) : (pfrom,pto) in parents_edges(ps,es) | pfrom > pto} $

% If i belongs to a star and D(i) /= D(j) then D(D(i)) := D(j)
  Use 1st iter. IF no dummy nodes %
function AS_lone_uncond_hook(ps,es) =
ps <- {(pfrom,pto) : (pfrom,pto) in parents_edges(ps,es);
                     in_starp in AS_lonecheck(ps) -> edges_froms(es)
                   | in_starp and (pfrom /= pto)} $
```

The following is a truly pseudo-random version of partitioning. Note the large amount of communication necessary. The extra argument `flip_nodeps` is a sequence of the length of the number of nodes in the original graph, which is allocated once at the beginning of the algorithm.

```
% Randomly partition edge end_points into two halves
  Needs #flip_nodeps == max nodenum for efficiency %
function RM_partition3(es,flip_nodeps) =
let flip_nodeps = flip_nodeps <- {(from,zerop(rand(r))) : r in dist(2,#es);
                                                          from in edges_froms(es)};
    flip_nodeps = flip_nodeps <- {(to,zerop(rand(r))) : r in dist(2,#es);
                                                        to in edges_tos(es)};
in ({flipfrom xor flipto : (flipfrom,flipto) in parents_edges(flip_nodeps,es)},
    flip_nodeps) $

function RM_active_edges3(es,flip_nodeps) =
let (actives,flip_nodeps) = RM_partition3(es,flip_nodeps);
    aes                   = {e : e in es; active in actives | active};
in if zerop(#aes) then RM_active_edges3(es,flip_nodeps)
   else flip_edges(aes,flip_nodeps -> edges_froms(aes)) $
```

References

[1] Agrawal, Ajit; Nekludova, Lena; and Lim, Willie. *A parallel $O(\log n)$ algorithm for finding connected components in planar images.* Technical report TMC-122. Thinking Machines Corp. February 1987.

[2] Awerbuch, B. and Shiloach, Y. *New Connectivity and MSF Algorithms for Ultracomputer and PRAM.* In *Proceedings of the International Conference on Parallel Processing*, pages 175-179. 1983.

[3] Das, S. K.; Deo, N.; and Prasad, S. *Parallel graph algorithms for hypercube computers.* In *Parallel Computing*, vol. 13, no. 2, pages 143–158. February 1990.

[4] Blelloch, Guy E. *Vector Models for Data-Parallel Computing.* MIT Press, Cambridge, MA, 1990.

[5] Kao, Ming-Ying and Shannon, Gregory E. *Linear-processor NC algorithms for planar directed graphs.* Technical report 306. Indiana University, Bloomington, Computer Science Dept. 1990.

[6] Blelloch, Guy E. *NESL: A Nested Data-Parallel Language.* Technical Report CMU–CS–92–103, Carnegie Mellon University, January 1992.

[7] Blelloch, Guy E. Unpublished CVL code. 1990.

[8] Gazit, Hillel. *An Optimal Randomized Parallel Algorithm for Finding Connected Components in a Graph.* In *SIAM Journal of Computing*, Vol. 20, No. 6, December 1991.

[9] Gopalakrishnan, P. S.; Ramakrishnan, I. V.; and Kanal, Laveen N. *An efficient connected components algorithm on a mesh-connected computer.* Technical report TR-1467, University of Maryland. 1987

[10] Greiner, John and Blelloch, Guy. *Data-Parallel Connected Components Algorithms.* To appear in *High Performance Computing*, ed. Gary Sabot.

[11] Hagerup, T. *Optimal parallel algorithms on planar graphs.* In *Information and Computation*, vol. 84, no. 1, pages 71–96. January 1990.

[12] Hambrusch, Susanne and TeWinkel, Lynn. *A study of connected component labeling algorithms on the MPP.* In *Proceedings of the Third International Conference on Supercomputing*, vol. 1, pages 477–483. May 1988.

[13] Han, Y. and Wagner, R. A. *An efficient and fast parallel-connected component algorithm.* In *Journal of the Association for Computing Machinery*, vol. 37, no. 3, pages 626–642. July 1990.

[14] Lim, Willie; Agrawal, Ajit; and Nekludova, Lena. *A fast parallel algorithm for labeling connected components in image arrays.* Technical Report TMC-124, Thinking Machines Corp. April, 1987.

[15] Lim, Willie. *Fast algorithms for labeling connected components in 2-D arrays.* Technical report TMC-125, Thinking Machines Corp. November, 1987.

[16] Pardalos, Panos M. and Rentala, Chandra S. *Computational aspects of a parallel algorithm to find the connected components of a graph.* Technical report CS-89-01, Pennsylvania State University Department of Computer Science. 1989.

[17] Phillips, Cynthia A. *Parallel Graph Contraction.* In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 148–157. June 1989.

[18] Reif, John H. *Optimal Parallel Algorithms for Integer Sorting and Graph Connectivity.* Technical Report TR-08-85, Harvard University, March 1985.

[19] Shiloach, Yossi and Vishkin, Uzi. *An $O(\log n)$ Parallel Connectivity Algorithm.* In *Journal of Algorithms*, pages 57–67, 1982.

[20] Swendsen, Robert H. and Wang, Jian-Sheng *Nonuniversal Critical Dynamics in Monte Carlo Simulations.* In *Physical Review Letters*, vol. 58, no. 2, pages 86–88. January 1987.

[21] Woo, Jinwoon and Sahni, Sartaj. *Hypercube computing: Connected components.* Technical report TR-88-50, University of Minnesota Computer Science Department. July 1988.

[22] Yang, Xue Dong. *An improved algorithm for labeling connected components in a binary image.* Technical report 89-981. Cornell University Department of Computer Science. March 1989.

[23] Yang, Xue Dong. *Design of fast connected components hardware.* In *Proceedings of the Computer Society Conference on Computer Vision and Pattern Recognition*, pages 937–944. 1988.