DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume 00, 0000

Parallel Implementation of Algorithms for Finding Connected Components in Graphs (Preprint)[†]

TSAN-SHENG HSU, VIJAYA RAMACHANDRAN,

AND NATHANIEL DEAN

June 10, 1996

ABSTRACT. In this paper, we describe our implementation of several parallel graph algorithms for finding connected components. Our implementation, with virtual processing, is on a 16,384-processor MasPar MP-1 using the language MPL. We present extensive test data on our code.

In our previous projects [**21, 22, 23**], we reported the implementation of an extensible parallel graph algorithms library. We developed general implementation and fine-tuning techniques without expending too much effort on optimizing each individual routine. We also handled the issue of implementing virtual processing.

In this paper, we describe several algorithms and fine-tuning techniques that we developed for the problem of finding connected components in parallel; many of the fine-tuning techniques are of general interest, and should be applicable to code for other problems. We present data on the execution time and memory usage of our various implementations.

1991 *Mathematics Subject Classification.* Primary 68-04; Secondary 05-04, 05C85, 68Q22.

Key words and phrases. parallel algorithms, graph algorithms, connected components, implementation, MasPar.

The first author was supported in part by NSC of Taiwan, ROC, Grants 84-2213-E-001-005 and 85-2213-E-001-003. The second author was supported in part by NSF Grant CCR-90-23059 and Texas Advanced Research Projects Grant 003658386.

This paper is in final form and no version of it will be submitted for publication elsewhere.

[†]This paper will appear in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume on the 3rd DIMACS Challenge, American Mathematical Societ.

1. Introduction

Over the past decade there has been a large amount of work in the theory of efficient, highly parallel graph algorithm design [**25, 27, 31, 46**]. Parallel algorithms that run in polylog time with linear or sub-linear number of processors have been developed for several fundamental problems on undirected graphs including connected components and spanning forest[§] [**2, 5, 7, 13, 16, 17, 24, 26, 42**], minimum spanning forest (MSF) [**2, 5, 6**], ear decomposition and 2-edge connectivity [**32, 37, 43**], open ear decomposition and biconnectivity [**32, 37, 43, 52**], triconnectivity [**12, 36**] and planarity [**44**]. All of these algorithms (with the exception of some algorithms for MSF) have the additional feature that they serialize into linear-time sequential algorithms. However, these algorithms are quite different from earlier linear time algorithms based on depth-first search [**51**] in that they are very modular in structure. For instance, the algorithm for ear decomposition calls subroutines for several basic problems such as connected components, spanning forest, the Euler tour technique on trees [**52**], least common ancestors in trees [**47, 52**] and range minima [**47, 52**]. More complex algorithms, such as those for triconnectivity and planarity call subroutines for open ear decomposition, in addition to subroutines for more basic problems. Thus an implementation of parallel algorithms for undirected graphs would have to proceed in a bottom-up fashion, starting with an implementation of basic primitives, and successively building up to more complex algorithms. Using this strategy, we have implemented efficient parallel algorithms for several combinatorial and graph problems [**21, 22, 23**].

Our implementations have been on the MasPar MP-1 in the parallel language MPL [**34, 35**], which is an extension of the C language [**28**]. In our previous papers [**21, 22, 23**], we reported on the implementation of an extensible parallel graph algorithms library using the approach outlined in the previous paragraph: We first built a *kernel* of basic parallel primitives and then implemented parallel graph algorithms in order of increasing complexity. In [**22**] we described general implementation and fine-tuning techniques used in the implementation of our parallel graph algorithms library; in this implementation we did not expend too much effort on optimizing each individual routine. Since the MasPar MP-1 does not support virtual processing in MPL, in [**23**] we handled the issue of implementing virtual processing. We then went into the basic routines we implemented in the kernel, and performed extensive fine-tuning; this is reported in [**21**]. In this paper, we present our work on fine-tuning the first parallel graph algorithm we implemented, that for finding connected components in an undirected graph. All of the non-trivial parallel graph algorithms that we have implemented, except the one for finding minimum spanning tree, call the routine for finding connected components before performing any further computation.

[§] In this paper, a spanning forest of a graph G is a maximal subgraph of G (w.r.t. the edges in G) that is a forest.

In this project we implemented several different parallel algorithms for the connected components problem, including one randomized algorithm, and tested our code with respect to various fine-tuning techniques.

Related work on implementing combinatorial algorithms on massively parallel machines can be found in [**1, 3, 4, 8, 9, 10, 11, 15, 16, 18, 19, 30, 38, 39, 41, 48**]. Also there has been work reported on implementing combinatorial algorithms on a vector super computer [**16, 45, 49**] and on a distributed memory machine [**29**].

The rest of the paper is organized as follows. Section 2 describes the algorithms implemented which includes an algorithm that we devised for this project. Section 3 gives general fine-tuning techniques for our code. Section 4 describes the testing scheme. Section 5 gives performance data. Finally Section 6 gives concluding remarks.

2. Algorithms

Given a list of vertices and edges in a graph, an algorithm for finding connected components assigns a unique component number $c(u)$ to each vertex u. Two vertices u and v are in the same component if and only if $c(u) = c(v)$. In addition to computing $c(u)$ for each vertex u, our implementation for finding connected components also returns the total number of connected components in the input graph. In this section, we describe four parallel algorithms that we have implemented.

All of the parallel algorithms we implemented use the well-known 'hooking-and-pointer-jumping' technique for finding connected components. Since our code takes care of isolated vertices at the start of the computation, we will assume here for convenience that there is no isolated vertex in the input graph. The execution of this type of computation proceeds in iterations. In each iteration, the following hooking and pointer jumping operations are performed. Initially, each vertex is assigned to a different set by itself. During execution, if two sets of vertices are found to belong to the same connected component, then these two sets are merged. We repeat the merging process until all vertices in each connected component are in the same set. The data structure for a set of vertices during the execution is a *tree loop*, where each vertex in the set has an outgoing pointer that points to another vertex in the set with the constraint that exactly one vertex has a pointer that points to itself. (Note that this is sometimes called a 'zero-tree-loop' in the literature [**20**].) Let the height of a tree loop T be the number of vertices in a longest simple directed path in T. A tree loop whose height is 2 is a *rooted star*. The vertex with self loop in a tree loop is the *root*. Two tree loops are merged by changing the pointer of the root of a tree loop to a vertex in the other tree loop. During the execution, it is desirable to reduce the height of a tree loop by performing a pointer jumping on vertices in the tree loop. When the algorithm terminates, all tree loops become rooted stars. The

```
each vertex is in a tree loop by itself;
assign a unique number α(u) to each vertex u;
let p(u) be the current pointer of the vertex u;
repeat
1.  /* Conditional star hooking.  */
     for all edges (u,v) execute in parallel
          if u is in a rooted star and α(p(u)) > α(p(v)) then
1.1            p(p(u)) ← p(v);
2.  /* Unconditional star hooking.  */
     for all edges (u,v) execute in parallel
          if u is in a rooted star and p(u) ≠ p(v) then
2.1            p(p(u)) ← p(v);
3.  /* Pointer jumping.  */
     perform a pointer jumping operation on all vertices
until there is no change in the current set of tree loops;
```

ALGORITHM 1. An algorithm for finding connected components by Awerbuch and Shiloach [**2**].

component number can thus be assigned on the roots and the component number of each vertex is the component number of its root. The number of connected component is equal to the number of tree loops.

Based on the method used to determine the set of tree loops to be merged, we can have many different algorithms. We implemented the following four algorithms, all of which run in $O(\log n)$ time (with high probability for the randomized algorithm) using a linear number of processors on a CRCW PRAM. Although techniques are known to reduce to the number of processors used in the algorithms [**7**], we chose not to implement them because the associated algorithms are quite complicated and the overhead is likely to be too large. The use of CRCW PRAM algorithms involved dealing with concurrent memory accesses; we discuss our implementation of concurrent memory accesses in Sections 3.4 and 3.7.

2.1. Awerbuch and Shiloach. The algorithm by Awerbuch and Shiloach [**2**] is shown in Algorithm 1. In each hooking-and-pointer-jumping iteration of this algorithm, two hooks are performed. The first hook, which is called *conditional star hooking*, makes the root of a rooted star point to a tree loop. In order to prevent two rooted stars from hooking to each other, the algorithm requires the root of the hooking star to have a smaller vertex number than the vertex number of the vertex to which it points. The second hook, which is called *unconditional star hooking*, makes the root of a rooted star point to a tree loop that is not itself.

Note that steps 1.1 and 2.1 are concurrent write operations. Note also that this algorithm needs to check which vertices are in rooted stars. This can be implemented using 2 concurrent read operations and one current write operation as follows. Using one concurrent read, each vertex finds its grandparent (i.e., the

```
each vertex is in a tree loop by itself;
assign a unique number α(u) to each vertex u;
let p(u) be the current pointer of the vertex u;
repeat
1.  /* Conditional hooking. */
     for all edges (u,v) execute in parallel
          if (u is the root or a child of a root) and α(p(u)) > α(p(v)) then
1.1            p(p(u)) ← p(v);
2.  /* Unconditional star hooking. */
     for all edges (u,v) execute in parallel
          if u is in a rooted star and p(u) ≠ p(v) then
2.1            p(p(u)) ← p(v);
3.  /* Pointer jumping. */
     perform a pointer jumping operation on all vertices
until there is no change in the current set of tree loops;
```

ALGORITHM 2. An algorithm for finding connected components by Shiloach and Vishkin [**50**] as described in Chapter 5.1.3 of JáJá [**25**].

parent of its parent). For each vertex whose grandparent is different from its parent, we mark (concurrent write) a flag f for its grandparent. Any vertex that is marked is not in a rooted star. Every unmarked vertex reads the flag f from its grandparent. Unmarked vertices whose grandparents are marked are also not in rooted stars.

2.2. Shiloach and Vishkin. The next algorithm we implemented (Algorithm 2) is by Shiloach and Vishkin [**50**] and also appears in Chapter 5.1.3 of JáJá [**25**]. In each of the hooking-and-pointer-jumping iteration of this algorithm, two hooks are performed as in Algorithm 1. The first hook, which is called *conditional hooking*, is similar to conditional star hooking as described in Algorithm 1, except that the root of a tree loop (rather than a rooted star) is made to point to another tree loop. The second hook is the same with the *unconditional star hooking* as described in Algorithm 1. This algorithm needs to check which vertices are in rooted stars only once during each iteration, instead of twice as in Algorithm 1.

2.3. A Revised Deterministic Algorithm. Algorithm 3 is a revised deterministic algorithm that we developed for this implementation project. In each hooking-and-pointer-jumping iteration of this algorithm, only one hook is performed. This is a conditional star hooking similar to the one in Algorithm 1 except that the tie-breaking rule when two rooted stars try to hook to each other alternates between the following two rules: In even-numbered iterations, the algorithm favors the rooted star with a larger vertex number while in odd-numbered iterations, the algorithm favors the rooted star with a smaller vertex number. This guarantees termination in a logarithmic number of iterations. Note that this algorithm tries to balance the amount of work performed and

```
each vertex is in a tree loop by itself;
assign a unique number α(u) to each vertex u;
let p(u) be the current pointer of the vertex u;
repeat
1.  /* Conditional star hooking.  */
     for all edges (u,v) execute in parallel
          if the number of iterations executed so far is even then
               if u is in a rooted star and α(p(u)) > α(p(v)) then
1.1                 p(p(u)) ← p(v);
          else
               if u is in a rooted star and α(p(u)) < α(p(v)) then
1.2                 p(p(u)) ← p(v);
2.  /* Pointer jumping.  */
     perform a pointer jumping operation on all vertices
until there is no change in the current set of tree loops;
```

ALGORITHM 3. A revised deterministic algorithm for finding connected components.

the number of tree loops reduced in each iteration. There is only one hook per iteration.

2.4. A Simple Randomized Algorithm. Algorithm 4 is a simple randomized algorithm (see, e.g., Chapter 4.3 in [**46**]) that avoids the checking of rooted stars by making sure that each tree loop is a rooted star at the beginning of each hooking-and-pointer-jumping iteration. In each iteration, two hooks between rooted stars are performed. By using a random bit in each vertex, the root of a rooted star with the random bit 1 is made to point to the root of a rooted star with the random bit 0. By enforcing the tie-braking rule using random bits, the height of each resulting tree loop is less than four. After a pointer jumping, all height-three tree loops become rooted stars. This simple randomized algorithm differs from the previous three algorithms by saving the efforts of checking for rooted stars in each iteration. Note that in this algorithm, the height of each tree loop is at most 2 after step 2. Thus all vertices are in rooted stars (or isolated vertices) after step 3. By using random bits to break ties in hooking, this algorithm avoids the construction of a tree loop with height larger than 2. Thus it is possible that it would take a larger number of iterations for the algorithm to terminate.

During the implementation of this algorithm, we found that when the number of vertices is small relative to the number of physical processors, the system pseudo random bits that we generated do not have good random behavior. Thus it usually took a very large number of iterations and a very long time for the algorithm to terminate. (The same problem is also reported in our implementation of a randomized list ranking algorithm [**21**].)

To avoid the above problem, we revised our algorithm as follows. We execute our randomized algorithm until the number of live edges left is less than half of the number of physical processors. Then we switch to the deterministic code

```
each vertex is in a tree loop by itself;
initially all edges are live;
let p(u) be the current pointer of the vertex u;
while there is a live edge do
1.  /* Coin tossing on vertices.  */
     assign a random bit β(u) to each vertex u;
2.  /* Hooking */
     for all live edges (u,v) execute in parallel
          if p(u) = p(v) then (u,v) is dead;
          else if β(p(u)) = 1 and β(p(v)) = 0 then
2.1              p(p(u)) ← p(v);
          else if β(p(u)) = 0 and β(p(v)) = 1 then
2.2              p(p(v)) ← p(u);
3.  /* Pointer jumping.  */
     perform a pointer jumping operation on all vertices
end ;
```

ALGORITHM 4. A simple randomized algorithm for finding connected components (see, e.g., Chapter 4.3 in [**46**]).

(without virtual processing) as described in [**22**]. Since it is possible for a very dense graph to have a small number of vertices, but a very large number of edges, the problem with the system pseudo random bits remained for dense graphs even after this modification. To handle this problem, we also revised the way our randomized algorithm picks (pseudo) random bits. Let n be the number of vertices. The randomized algorithm first picks a random bit for each vertex. Then we count the number of random bits that are 1. If the number of 1 bits is less than $\frac{n}{8}$ or is greater than $\frac{7n}{8}$, then we use the following algorithm to re-pick pseudo random bits. We generate a positive 32-bit pseudo random number for each physical processor that contains some vertices. We assign the number 0 to each physical processor that does not contain vertices. Then we compute the ranks of the random numbers generated. Let b_i be the parity of the rank associated with the ith physical processor. We assign the bit $(b_i + j) \bmod 2$ to the jth vertex in the ith physical processor. Since we need to compute ranks, the second method takes more time than the first method. Thus we use the first method to generate pseudo random bits and switch to the second method only if the first method fails.

We found that when the number of vertices is more than a quarter of the number of physical processors, we never use the second method in our testing. When the second method is used, the number of vertices in each physical processor is at most 1. As a result, the number of 1 bits and the number of 0 bits differ by at most 1.

3. General Fine-Tuning Techniques

In this section, we describe several fine-tuning techniques.

3.1. Compressed Data Structure for Edges. Three of our algorithms (Algorithms 1–3) require that two copies of an undirected edge to be stored for processing. To save memory usage, we store only one copy. Whenever we need to perform operations based on the set of edges, our code performs the same operations twice, assuming that we have two copies of the same edge available. As indicated in [**23**], the amount of extra computation time used is negligible. By doing this, we are able to handle input sizes up to twice as large as we could without the compressed data structure.

3.2. Special Routine for the First Iteration of Hooking. During the first iteration of the first three algorithms, isolated vertices instead of the roots of rooted stars hook into other tree loops. Checking for isolated vertices requires only one concurrent write operation and is much faster than checking for rooted stars. Thus we can use special routines to compute the first iteration of Algorithms 1–3.

3.3. Check for Live Edges. Algorithm 4 introduces a techniques to get rid of edges connecting two vertices that are inside the same tree loop (i.e., that are already known to be in the same connected component). The check works as follows: An edge (u, v) is known to be in the same connected component if u and v have the same parent pointer. Let edge (u, v) be *dead* if the current parent pointers of u and v are the same. An edge is *live* if it is not dead. At the start of the each iteration, the algorithm checks the parent pointers for the endpoints of edges that are currently live. Dead edges do not participate in further computation.

Algorithm 4 uses this check of live edges not only to remove dead edges, but also to detect the termination of computation. In addition, by getting rid of dead edges, the total number of operations performed during each iteration is reduced. This technique can be applied to the other three algorithms as well.

3.4. Implementation of Concurrent Write Operations. In our algorithms, we need to implement arbitrary concurrent write operations. The system-provided routine `rsend` in MPL language can be used to directly implement arbitrary write operations. The execution time of an arbitrary concurrent write operation increases when the maximum number of concurrent write requests per physical processor increases. This is the 'Queue-Write' model that is addressed in [**14**].

Another way of implementing concurrent write in MPL is to use the routine `sendwith`. This routine executes the standard simulation of a concurrent write step on an exclusive write PRAM [**27**], and requires the use of sorting and additional working memory space. The execution time of a concurrent write operation when `sendwith` is used depends only on the total number of write requests, and not on the maximum number of concurrent write requests per physical processor. Further, with this routine, one can implement priority write with the same delay as an arbitrary write. This delay is considerably less than the delay caused by the routine `rsend` when the maximum concurrency at a memory location is large, but the reverse is true when the concurrency at every memory location is small.

Our experimental data indicates that if we modify our connected components algorithms and use priority write operations in places where arbitrary concurrent write operations are needed, we tend to have fewer number of iterations. For example, if there are several candidates which the root r of a tree loop can hook to, using an arbitrary write operation causes r to hook to an arbitrary tree loop. Instead, if we use a priority write operation and cause r to hook to a vertex with the largest vertex number, the maximum height of the resulting tree loops tends to be smaller than the maximum height of the tree loops formed using an arbitrary write operation. In our implementation we used both arbitrary concurrent writes using the `rsend` routine and priority concurrent writes using the `sendwith` routine and compared the performance of the resulting codes. (The performance data will be shown and discussed in Section 5.1.)

In addition to using just `rsend` or just `sendwith` to implement concurrent write operations, we can also use the following hybrid implementation. We note that our algorithms usually have a large number of hooks (and thus a high probability of having large number of concurrent write requests per physical processor) during the first iteration. The number of hooks performed tends to be smaller after the first iteration. Thus we can use priority concurrent write (with the `sendwith` routine) in the first iteration and arbitrary concurrent write (with the `rsend` routine) (i.e., queue-write [**14**]) in the remaining iterations. We will show in Section 5 that this hybrid implementation improves the performance of some of our algorithms.

3.5. Edge Condensation. During the execution of the algorithm, vertices in a tree loop can be viewed as a super vertex and can be collapsed. When collapsing vertices, multiple edges can be removed using sorting. We implement the collapsing of vertices by the following method. During each iteration, whenever we need to retrieve the pointer of an end point of an edge, we replace the end point with its pointer. We refer to this operation as *edge condensation.* If a tree loop is a rooted star, and an edge incident on it is examined, then all of the vertices in the tree loop are collapsed into a super vertex. After each iteration, we sort (lexicographically) the set of edges and remove multiple copies of the same edge.

3.6. Deferred Pointer Jumping. Given a tree loop, the depth of a vertex u is the number of vertices on the path from u to the root of its tree loop. The pointer of a vertex with depth less than or equal to 2 does not change when pointer jumping is performed. To reduce the number of pointer chasing operations performed during pointer jumping, we can use the following *deferred pointer jumping* scheme.

Note that it takes one concurrent write operation to check whether a non-root vertex is a leaf in a tree loop by marking a flag for the parent of each vertex. After the marking, i.e., concurrent write, the vertices remain unmarked are leaves. Initially, all vertices are active. During the beginning of each iteration, we mark the leaves in tree loops as 'inactive.' Inactive vertices keep their pointers, but do not participate further computation, and edge condensation is performed on edges incident on them to move these edges further up the tree loop. Given a set of tree loops, let the set of *active tree loops* be the induced subgraph on active vertices. Note that a rooted star that marks its leaves inactive becomes an isolated vertex. Thus in our new scheme, we hook active tree loops that are isolated vertices instead of rooted stars. We find the set of isolated vertices using one concurrent write operation as follows. We mark a flag for the parent of each vertex using concurrent write. After the marking, the vertices that remain unmarked and whose parent pointers are null or point to themselves are isolated vertices. Finally, after the original algorithm terminates, we mark all vertices active and perform pointer jumping until all tree loops are rooted stars.

Recall that the original star-checking algorithm as described in Section 2.1 uses two concurrent read operations and one concurrent write operation. Under the new scheme, it takes 1 concurrent write to find inactive vertices. After marking inactive vertices, rooted stars becomes isolated vertices. It takes only one concurrent write to find isolated vertices. Our experimental data indicates that this new method runs faster.

Note also that after removing inactive vertices, the height of each tree loop decreases (unless all tree loops are isolated vertices). Without using the new scheme, all depth-1 vertices (i.e., children of the root) in a tree loop perform a pointer jump operation even though their pointers do not change after the jump. This is a waste of communication bandwidth. Using our new scheme, leaves that are originally depth-1 do not participate in pointer jumping. Thus our new scheme not only reduces the time needed to check which tree loop to hook to, but also reduces the heights of the tree loops we work with, and the number of pointer chasing operations we perform during each iteration.

3.7. Implementation of Concurrent Read. In implementing pointer jumping, we need to use concurrent read operations. The MPL language provides system routine `rfetch` to directly implement it. The execution time of an concurrent read operation increases when the maximum number of concurrent read requests per physical processor increases [**21, 40**]. This is the 'Queue-Read' model that is addressed in [**14**]. As indicated in the experiments performed in [**21**], an exclusive read implementation of concurrent read operations using sorting outperforms the `rfetch` concurrent read implementation when the maximum number of concurrent read requests on one processor is larger than 256. We found that when performing pointer jumping on dense graphs with less vertices than the number of physical processors in the system, the maximum number of concurrent read requests on one physical processor becomes quite large at the very last few iterations. Thus it is desirable to switch to an exclusive read implementation during this stage. Hence in our implementations we use `rfetch` for deferred pointer jumping in the initial iterations, but when the number of vertices becomes less than or equal to the number of physical processors we use an exclusive read implementation to perform deferred pointer jumping operations. We will show in Section 5 this hybrid implementation of concurrent read operations improves the performance of some of our algorithms.

4. Testing Scheme

4.1. Computing Platform. All of our parallel implementations are on a MasPar MP-1 with 16,384 processors. The MasPar computer [**33**] is a fine-grained massively parallel single-instruction-multiple-data (SIMD) computer. All of its parallel processors synchronously execute the same instruction at the same time. A description of the hardware architecture and the software environment of MasPar MP-1 can be found in [**22**].

We used 4 kilo-bytes of memory per physical processor to test our various implementations for connected components. We show the performance of our code running on the largest data that we could fit into the system.

4.2. Test Inputs. In our code, an undirected graph is represented by a list of edges in it. We tested our programs using random graphs of three different edge densities*:

- dense graphs where $m = \frac{n^2}{4}$;
- intermediate-density graphs where $m = n^{1.5}$;
- sparse graphs where $m = \frac{3n}{2}$.

To generate a random graph with n vertices and m edges, we first generated an empty graph with n vertices. Then we added one edge at a time with each edge being chosen with uniform probability until exactly m edges were generated. For each size and sparsity, we generated four different test graphs. We ran each program on each test graph for 10 iterations and recorded the average of the 40 trials.

We also used the following special classes of graphs that are reported in the experiments conducted in [**16**].

- Two-dimensional wrap-around grids that are squares with 30% and 60% of their edges (chosen randomly). Thus given a graph in this class, $m = 0.6 \cdot n$ when the edge density is 30% and $m = 1.2 \cdot n$ when the edge density is 60%.
- Three-dimensional wrap-around grids that have the same size in each dimension with 20% and 40% of their edges (chosen randomly). Thus

*In this paper, n and m always represent the number of vertices and edges in the input graph, respectively.

given a graph in this class, $m = 0.6 \cdot n$ when the edge density is 30% and $m = 1.2 \cdot n$ when the edge density is 60%.

- Tertiary graphs in which each vertex randomly selects three neighbors. Note that tertiary graphs are multi-graphs and $m = 3 \cdot n$.
- Random graphs with $m = 0.02 \cdot \frac{n \cdot (n-1)}{2}$.

5. Performance Data

We implemented the Algorithms 1 through 4 described in Section 2. All of our code used the compressed data structure that represents each undirected edge only once, a special routine for the first iteration, and performed the check for live edges. We also did not use deferred pointer jumping in Algorithms 1 and 2, since we found that there is no performance improvement by adding this feature.

5.1. Deterministic Algorithms. We first tested the following 5 different deterministic code using 16,384 PE's with 4 kilo-bytes of memory per PE.

- Code A: Algorithm 2.
- Code B: Code A with edge condensation, but does not remove duplicated edges.
- Code C: Algorithm 1.
- Code D: Code C with edge condensation, but does not remove duplicated edges.
- Code E: Algorithm 3 with edge condensation and deferred pointer jumping, but does not remove duplicated edges.

For each code, we tested two different methods to implement concurrent write. In the first version, we used arbitrary concurrent write operations (using the `rsend` routine), and in the second version we used priority concurrent write operations (using the `sendwith` routine). The performance data is shown in Table 1. We observe that for dense graphs, most of the runs terminate in a few iterations, and Algorithm 2 (i.e., Code A and Code B) outperforms the rest. Algorithm 1 (i.e., Code C) outperforms others on priority write implementation. Algorithm 3 (i.e., Code E) performs better when the graph is very sparse. We also observe that the priority write version decreases the number of iterations needed for code to terminate. However, the total execution time does not necessary decrease because of the overhead involved in implementing priority concurrent write operations using the `sendwith` routine.

Removing Duplicated Edges and Implementing Hybrid Concurrent Read/Write. We re-ran the above code with the modification of removing duplicated edges in each iteration. We found that the performance for Code A through D does not improve. We also used hybrid concurrent read and write in our code. We found that the performance of Code A through D did not improve too much. However, the performance of Code E greatly improves and

		$m = n^2/4$		$m = n^{3/2}$		$m = 3n/2$	
		seconds	iterations	seconds	iterations	seconds	iterations
Code A	Arbitrary CW	3.6	5.0	2.9	6.0	50.3	8.5
	Priority CW	5.2	3.5	4.2	4.2	52.8	7.0
Code B	Arbitrary CW	3.6	5.0	2.9	6.0	46.1	7.2
	Priority CW	5.2	3.5	4.2	4.2	47.3	6.2
Code C	Arbitrary CW	5.0	6.0	6.7	8.0	103.3	11.5
	Priority CW	2.2	3.0	4.0	5.0	72.5	7.0
Code D	Arbitrary CW	13.7	6.0	21.5	7.8	105.3	10.8
	Priority CW	12.0	3.0	17.8	5.0	84.0	7.0
Code E	Arbitrary CW	33.4	14.5	33.7	17.0	17.4	19.5
	Priority CW	8.9	5.0	18.8	9.0	22.0	19.5

TABLE 1. Performance for 5 different deterministic code when $m = 262,142$. We show both the execution time (in seconds) and the number of iterations the algorithm needs to terminate.

outperforms all other code in all classes of graphs. The performance data is shown in Table 2.

5.2. Randomized Algorithm. We incorporated the features of removing duplicated edges and deferred pointer jumping in our randomized algorithm. The performance of our code is shown in Table 3. We observe that the performance of our randomized code is slower than our best deterministic version on the set of graphs that we have tested.

5.3. Further Testing. We tested our code on several other classes of graphs which are shown in Section 4.2. The performance data for Code E with the features of removing duplicated edges and using hybrid concurrent write operations is shown in Table 4. The performance data for our randomized code is shown in Tables 5 and 6. We note that our randomized code outperforms our best deterministic code on grids. However, our deterministic code still outperforms our randomized code on random graphs and tertiary graphs.

5.4. Memory Usage. Using 16,384 physical processors and 4 kilo-bytes of memory per physical processor (which is $\frac{1}{16}$ of the total available memory), we were able to run our randomized algorithm for any graph with upto 0.52 million vertices and 0.52 million edges. That is, we are able to store 32 vertices and 32 edges in a physical processor. Our deterministic algorithm runs faster, though it requires more memory. We show the performance of our deterministic code for graphs with upto 0.26 million vertices and 0.26 million edges. For graph of this size, we store 16 vertices and 16 edges in a physical processor.

We observe that we do not occupy all 4 kilo-bytes of space by storing 16 vertices and 16 edges in a physical processor, and we should have enough space to pack 24 vertices and 24 edges. However, the sorting program that we used [**41**] requires that the number of data allocated in each physical processor to be power of 2. We use sorting to implement priority concurrent write operations (using the `sendwith` routine) and to remove duplicated edges. Thus we are unable to run our algorithms on larger graphs even though we have space left. For

			$m = 32,766$		$m = 65,534$		$m = 131,070$		$m = 262,142$	
			secs	itrs	secs	itrs	secs	itrs	secs	itrs
$m = n^2/4$	code from **[23]**		0.6	NA	1.6	NA	2.4	NA	5.3	NA
	Code C	orig.	0.3	3.0	0.7	3.0	1.1	3.0	2.2	3.0
	Pri. write	rev.	0.4	3.0	0.8	3.0	1.3	3.0	2.6	3.0
	Code E, Pri. write	orig.	1.1	5.0	2.0	5.0	4.3	5.0	8.9	5.0
	Code E	rev.	0.2	3.5	0.5	4.0	0.8	4.0	1.6	4.0
$m = n^{3/2}$	code from **[23]**		0.9	NA	1.9	NA	3.4	NA	7.2	NA
	Code A	orig.	0.6	6.0	0.9	5.8	1.7	6.0	2.9	6.0
	Arb. write	rev.	0.8	5.8	1.2	5.5	2.5	6.0	4.5	6.0
	Code E, Pri. write	orig.	2.3	9.0	4.6	9.0	9.1	9.0	18.8	9.0
	Code E	rev.	0.3	6.0	0.6	7.0	1.1	7.0	2.2	7.0
$m = 3n/2$	code from **[23]**		7.4	NA	15.6	NA	33.4	NA	49.1	NA
	Code B	orig.	5.7	6.0	11.2	6.0	21.7	6.0	47.3	6.2
	Pri. write	rev.	5.0	6.0	11.7	6.5	25.2	6.8	54.6	7.0
	Code E, Arb. write	orig.	1.4	15.5	3.4	18.0	8.2	20.0	17.4	19.5
	Code E	rev.	0.8	14.0	1.5	14.5	3.2	16.0	6.3	15.0

TABLE 2. The performance of our deterministic code after adding the feature of removing duplicated edges and also the features of hybrid concurrent read and write. For each class of graphs, we show the performance of the program which has the best performance among Code A through D and the fastest version of Code E. We apply the feature of removing duplicated edges in all of our tested code and apply the feature of hybrid concurrent write on Code E only. For comparison, we also list the performance of the code in [**23**] (based on Algorithm 1) which is not fine-tuned.

	$m = 32,766$		$m = 65,534$		$m = 131,070$		$m = 262,142$		$m = 524,286$	
	secs	itrs	secs	itrs	secs	itrs	secs	itrs	secs	itrs
$m = n^2/4$	1.2	3.0	2.0	3.1	3.5	4.1	6.6	4.0	13.4	4.5
$m = n^{3/2}$	1.2	3.3	1.9	4.0	3.7	4.9	6.4	4.7	13.0	5.2
$m = 3n/2$	1.9	6.0	3.2	7.6	5.6	7.7	11.6	8.5	25.4	9.8

TABLE 3. The performance of our randomized algorithm (with removing duplicated edges) is shown. We show both the total execution time (in seconds) and the number of iterations needed for the algorithm to reduce its number of edge to 8,192, in which case we switch to a deterministic algorithm.

	2-D grids		3-D grids		Tertiary graphs	Random graphs
	30%	60%	20%	40%		
n	261,121	218,089	250,047	216,000	87,380	5,119
m	157,285	262,142	157,285	262,142	262,140	262,142
seconds	4.5	6.7	5.5	5.9	5.8	2.3
iterations	12.5	20.0	16.5	17.0	11.5	7.0

TABLE 4. The performance of our best deterministic code on several special classes of graphs.

	2-D grids				3-D grids			
	30%		60%		20%		40%	
n	261,121	524,176	218,089	435,600	250,047	512,000	216,000	421,875
m	157,285	314,571	262,142	524,286	157,285	314,571	262,142	524,286
secs	4.0	8.8	5.7	12.6	4.1	9.3	7.6	16.4
itrs	7.0	8.0	9.0	11.0	7.1	8.1	9.6	11.0

TABLE 5. The performance of our randomized code on grid graphs.

	Tertiary graphs		Random graphs	
n	87,380	174,762	5,119	7,240
m	262,140	524,286	262,142	524,286
secs	9.6	21.3	7.0	15.2
itrs	6.7	7.9	4.8	5.5

TABLE 6. The performance of our randomized code on two special classes of graphs.

comparison, the code that we had for our previous implementation [**23**] (which is not fine-tuned) used 16 kilo-bytes of space to process upto 64 vertices and 64 edges per physical processor. Our randomized code used only half of the amount of memory because of the simplicity of the underlying algorithm. As a result of our fine-tuning, our current deterministic code uses 25% less memory than our earlier version in [**23**].

5.5. Comparison to Related Work. In [**16**], Greiner reported the implementation of several parallel algorithms for finding connected components on a massively parallel computer CM-2 using a quarter of the processors (8,192 processors) and all 32 kilo-bytes of memory in each processor. He also reported an implementation on a vector super computer Cray C-90 using one processor. Greiner implemented the algorithms of Shiloach and Vishkin [**50**] and Awerbuch and Shiloach [**2**], and the simple randomized algorithm that we implemented. Greiner did not use the system pseudo number generator for generating random bits in his randomized code. Instead, the ith "random" bit for a vertex is the $(i \bmod \log_2 n)$th bit of the vertex number. He implemented fine-tuning techniques which include routines similar to our first iteration of hooking, check of live edges, and edge condensation. He also implemented hybrid algorithms that combine features in the above three algorithms and an algorithm in Hirchberg, Chandra, and Sarwate [**20**]. His hybrid algorithm has the best performance for the classes of graphs tested. Greiner did not implement the compressed data structure for edges, various implementation of concurrent write operations, or deferred pointer jumping, all of which have been implemented in our work. Our revised deterministic algorithm is also different from any of the algorithms that he used.

Our code on the MasPar MP-1 is about half as fast as Greiner's code on the CM-2 for random graphs and tertiary graphs. On grids, his code is more than twice as fast. We observe that for grid graphs, his randomized code, though not the fastest overall, has about the same performance as his best code. His code on the Cray C-90 is about 15 times faster than his code on the CM-2. It should be noted that the CM-2 is a more expensive machine than the MasPar MP-1, and the Cray C-90 is a much more expensive machine than the MP-1.

The issue of memory usage is not addressed in [**16**]. Using four times the amount of total memory that we used, Greiner shows CM-2 performance data on random graphs with about 0.52 millions edges. He shows CM-2 performance data on grids and tertiary graphs with twice as many edges. As indicated in Section 5.4, our randomized code can run graphs with 0.52 millions edges. We also show performance data for our deterministic code on graphs with 0.26 millions edges. Since we use only quarter the amount of memory as Greiner's implementation on the CM-2, it appears that our code uses less space than his.

In [**29**], a distributed memory implementation of the algorithm by Shiloach and Vishkin [**50**] is reported. After fine-tuning, they obtain a speedup of 20 using a 32 processor CM-5 on grid graphs and obtain virtually no speedup on sparse random graphs. The performance of our massively parallel implementation seems to be more adaptable to different classes of graphs.

In [**30**], a mesh implementation of hooking-and-pointer-jumping type algorithms is reported on a MasPar MP-1 using 8,192 processors. By using the underlying mesh architecture and the fact that the mesh communication is more than 100 times faster than the global router communication, their implementation is generally faster than ours on dense graphs. Our implementation runs in about the same speed than theirs on very sparse graphs. In [**30**], the input graph is represented by an adjacency linked list on sparse graphs and is represented by an adjacency matrix on dense graphs while we use an arbitrary edge list to represent any input graph. It appears that our input data format is more flexible. Note that it takes non-trivial time to prepare an adjacency linked list or an adjacency matrix data structure for a graph. Because of the different data structures used for the input graph, our implementation also has a more efficient usage of the memory if the difference between the minimum vertex degree and the maximum vertex degree is large. We also have a more balanced usage of memory in each processor if the graph is very dense.

6. Concluding Remarks

In this paper, we have described our project on the implementation and fine-tuning of parallel code for the important problem of finding connected components in an undirected graph. Our fine-tuned code is more than 7 times faster than our original code on very sparse graph and also uses less memory. A randomized version of our code requires less memory, although it runs slower. This version can be used when memory is at a premium. We also note that our Code E with various revisions outperforms all other programs on all classes of graphs except grids. On grids, our randomized code has the best performance.

References

1. R. Anderson and J. Setubal, *On the parallel implementation of Goldberg's maximum flow algorithm*, Proc. 4th ACM Symp. on Parallel Algorithms and Architectures, 1992, pp. 168–177.
2. B. Awerbuch and Y. Shiloach, *New connectivity and MSF algorithms for shuffle-exchange network and PRAM*, IEEE Tran. on Computers (1987), 1258–1263.
3. G. E. Blelloch, *Scan primitives and parallel vector models*, Ph.D. thesis, M.I.T., October 1989.
4. G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, *A comparison of sorting algorithms for the Connection Machine CM-2*, Proc. 3th ACM Symp. on Parallel Algorithms and Architectures, 1991, pp. 3–16.
5. K. W. Chong and T. W. Lam, *Finding connected components in $O(\log n \log\log n)$ time on the EREW PRAM*, Proc. 4th Annual ACM-SIAM Symp. on Discrete Algorithms, 1993, pp. 11–20.
6. R. Cole, P. N. Klein, and R. E. Tarjan, *A linear-work parallel algorithm for finding minimum spanning trees*, Proc. 6th ACM Symp. on Parallel Algorithms and Architectures, 1994, pp. 11–15.
7. R. Cole and U. Vishkin, *Approximate parallel scheduling. Part II: Applications to logarithmic-time optimal graph algorithms*, Information and Computation **92** (1991), 1–47.
8. E. Dekel, D. Nassimi, and S. Sahni, *Parallel matrix and graph algorithms*, SIAM J. Comput. **10** (1981), 657–675.
9. B. Dixon and A. K. Lenstra, *Factoring integers using SIMD sieves*, Manuscript, 1992.
10. ———, *Massively parallel elliptic curve factoring*, Manuscript, 1992.
11. T. Feder, A. G. Greenberg, V. Ramachandran, M. Rauch, and L.-C. Wang, *Circuit switched link simulation: Algorithms, complexity and implementation*, Draft manuscript, 1992.
12. D. Fussel, V. Ramachandran, and R. Thurimella, *Finding triconnected components by local replacements*, SIAM J. Comput. **22** (1993), no. 3, 587–616.
13. H. Gazit, *An optimal randomized parallel algorithm for finding connected components in a graph*, SIAM J. Comput. **20** (1991), no. 6, 1046–1067.
14. P. B. Gibbons, Y. Matias, and V. Ramachandran, *The QRQW PRAM: Accounting for contention in parallel algorithms*, Proc. 5th ACM-SIAM Symp. on Discrete Algorithms, 1994, pp. 638–648, SIAM J. Comput., to appear.
15. A. G. Greenberg, B. D. Lubachevsky, and L.-C. Wang, *Experience in massively parallel discrete event simulation*, Proc. 5th ACM Symp. on Parallel Algorithms and Architectures, 1993, pp. 193–202.
16. J. Greiner, *A comparison of data-parallel algorithms for connected components*, Proc. 6th ACM Symp. on Parallel Algorithms and Architectures, 1994, pp. 16–25.
17. S. Halperin and U. Zwick, *An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM*, Proc. 6th ACM Symp. on Parallel Algorithms and Architectures, 1994, pp. 1–10.
18. W. Hightower, J. Prins, and J. Reif, *Implementations of randomized sorting on large parallel machines*, Proc. 4th ACM Symp. on Parallel Algorithms and Architectures, 1992, pp. 158–167.
19. W. D. Hillis and G. L. Steele Jr., *Data parallel algorithms*, Communications of the ACM **29** (1986), 1170–1183.
20. D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, *Computing connected components on parallel computers*, Communications of the ACM **22** (1979), no. 8, 461–464.
21. T.-s. Hsu and V. Ramachandran, *Efficient massively parallel implementation of some combinatorial algorithms*, Theoretical Computer Science (1996, to appear).
22. T.-s. Hsu, V. Ramachandran, and N. Dean, *Implementation of parallel graph algorithms on the MasPar*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 15, American Mathematical Society, 1994, pp. 165–198.
23. ———, *Implementation of parallel graph algorithms on a massively parallel SIMD computer with virtual processing*, Proc. 9th International Parallel Processing Symp., 1995, pp. 106–112.

24. K. Iwama and Y. Kambayashi, *A simpler parallel algorithm for graph connectivity*, Journal of Algorithms **16** (1994), 190–217.
25. J. JáJá, *An introduction to parallel algorithms*, Addison-Wesley, 1992.
26. D. R. Karger, N. Nisan, and M. Parnas, *Fast connected components algorithms for the EREW PRAM*, Proc. 4th ACM Symp. on Parallel Algorithms and Architectures, 1992, pp. 373–381.
27. R. M. Karp and V. Ramachandran, *Parallel algorithms for shared-memory machines*, Handbook of Theoretical Computer Science (J. van Leeuwen, ed.), North Holland, 1990, pp. 869–941.
28. B. W. Kernighan and D. M. Ritchie, *The C programming language*, Prentice Hall, Englewood Cliffs, NJ, 1988, Second Edition.
29. A. Krishnamurthy, S. Lumetta, D. E. Culler, and K. Yelick, *Connected components on distributed memory machines*, Presented at the 3rd DIMACS Implementation Challenge Workshop, October, 1994.
30. S. Kumar, S. M. Goddard, and J. F. Prins, *Connected-components algorithms for mesh-connected parallel computers*, Presented at the 3rd DIMACS Implementation Challenge Workshop, October, 1994.
31. F. T. Leighton, *Introduction to parallel algorithms and architectures: Arrays, trees, hypercubes*, Morgan Kaufmann, 1992.
32. Y. Maon, B. Schieber, and U. Vishkin, *Parallel ear decomposition search (EDS) and st-numbering in graphs*, Theoret. Comput. Sci. (1986), 277–298.
33. MasPar Computer Co., *MasPar system overview*, version 2.0 ed., March 1991.
34. MasPar Computer Co., *MasPar parallel application language (MPL) reference manual*, version 3.0, rev. a3 ed., July 1992.
35. MasPar Computer Co., *MasPar parallel application language (MPL) user guide*, version 3.1, rev. a3 ed., November 1992.
36. G. L. Miller and V. Ramachandran, *A new triconnectivity algorithm and its applications*, Combinatorica **12** (1992), 53–76.
37. ———, *Efficient parallel ear decomposition with applications*, Manuscript, MSRI, Berkeley, CA, January 1986.
38. B. Narendran and P. Tiwari, *Polynomial root-finding: Analysis and computational investigation of a parallel algorithm*, Proc. 4th ACM Symp. on Parallel Algorithms and Architectures, 1992, pp. 178–187.
39. P. M. Pardalos, M. G.C. Resende, and K.G. Ramakrishnan (eds.), *Parallel processing of discrete optimization problems*, DIMACS series in discrete mathematics and theoretical computer science, vol. 22, American Mathematical Society, 1995.
40. L. Prechelt, *Measurements of MasPar MP-1216A communication operations*, Tech. Report 01/93, Institute für Programmstrukturen und Datenorganisation, Fakultät für Informatik, Universität Karlsruhe, Germany, January 1993.
41. J. F. Prins and J. A. Smith, *Parallel sorting of large arrays on the MasPar MP-1*, Proc. 3rd Symp. on the Frontiers of Massively Parallel Computation, 1990, pp. 59–64.
42. T. Radzik, *Computing connected components on EREW PRAM*, Tech. report, King's College, London, 1994, Tech. Rep. 94/02.
43. V. Ramachandran, *Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity*, Synthesis of Parallel Algorithms (J. H. Reif, ed.), Morgan-Kaufmann, 1993, pp. 275–340.
44. V. Ramachandran and J. Reif, *Planarity testing in parallel*, Jour. Comput. and Sys. Sci. **49** (1994), no. 3, 517–561, Special Issue for *FOCS '89*.
45. M. Reid-Miller, *List ranking and list scan on the CRAY C-90*, Proc. 6th ACM Symp. on Parallel Algorithms and Architectures, 1994, pp. 104–113.
46. J. H. Reif (ed.), *Synthesis of parallel algorithms*, Morgan-Kaufmann, 1993.
47. B. Schieber and U. Vishkin, *On finding lowest common ancestors: Simplification and parallelization*, SIAM J. Comput. **17** (1988), no. 6, 1253–1262.
48. J. T. Schwartz, *Ultracomputers*, ACM Trans. on Programming Languages and Systems **2** (1980), 484–521.

49. T. J. Sheffler, *Implementing the multiprefix operation on parallel and vector computers*, Proc. 5th ACM Symp. on Parallel Algorithms and Architectures, 1993, pp. 377–386.
50. Y. Shiloach and U. Vishkin, *An* $o(\log n)$ *parallel connectivity algorithm*, Journal of Algorithms (1982), 57–67.
51. R. E. Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Comput. **1** (1972), 146–160.
52. R. E. Tarjan and U. Vishkin, *An efficient parallel biconnectivity algorithm*, SIAM J. Comput. **14** (1985), 862–874.

Inst. of Information Science, Academia Sinica, Nankang 115, Taipei, Taiwan, ROC
E-mail address: tshsu@iis.sinica.edu.tw

Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712, USA
E-mail address: vlr@cs.utexas.edu

S/W Production Research, AT&T Bell Labs., Murray Hill, NJ 07960, USA
E-mail address: nate@research.att.com