Fast Connected Components Algorithms for the EREW PRAM

David R. Karger [*] Noam Nisan [†] Michal Parnas [‡]

July 1, 1997

Abstract

We present fast and efficient parallel algorithms for finding the connected components of an undirected graph. These algorithms run on the exclusive-read, exclusive-write (EREW) PRAM. On a graph with n vertices and m edges, our randomized algorithm runs in $O(\log n)$ time using $(m+n^{1+\epsilon})/\log n$ EREW processors (for any fixed $\epsilon > 0$). A variant uses $(m+n)/\log n$ processors and runs in $O(\log n \log\log n)$ time. A deterministic version of the algorithm runs in $O(\log^{1.5} n)$ time using $m+n$ EREW processors.

1 Introduction

Perhaps the most basic algorithmic problem involving an undirected graph is to find its connected components. In this problem, the input to the algorithm is an undirected graph $G = (V, E)$, with $|V| = n$ vertices and $|E| = m$ edges. The output is the connected components of the graph. There are various ways to represent the solution; the one we shall use is to label each vertex with the largest numbered vertex to which it is connected. Connected components can be found in linear sequential time by breadth-first search or depth-first search methods. However, these methods do not parallelize easily. Parallel algorithms for connected components have been known for quite some time ([HCS79], [CLC82], or see the survey in [KR90]). Until recently the best known algorithms required $O(\log n)$ time on CRCW PRAMs and $O(\log^2 n)$ time on CREW PRAMs (recall that CR (CW) PRAM allow multiple processors to concurrently read (write) to the same memory location, while ER (EW) PRAMs allow only one processor to read (write) at a time). The number of processors used by the best of those algorithms is nearly optimal in the deterministic case [SV82, CV91, AS87] and completely optimal in the randomized case [Gaz91].

In their survey, Karp and Ramachandran [KR90] raised the question of the existence of $o(\log^2 n)$-time algorithms for connected components on exclusive-write PRAMs. Recently, Johnson and Metaxas [JM91] developed a CREW algorithm that runs in $O(\log^{1.5} n)$ time and uses $m + n$ processors. An $O(\log^{1.5} n)$-time algorithm for the EREW PRAM is described in [NSW92], but this algorithm uses a large polynomial number of processors.

In this paper we present improved EREW algorithms for connected components. One contribution is the first (randomized) algorithm that runs in $O(\log n)$ time. It is based on the parallelization of random walk techniques studied in [AKL*79], where it is shown that a relatively short random walk will visit all the vertices in a graph.

[*]Department of Computer Science, Stanford University. Supported by a National Science Foundation Graduate Fellowship, by NSF Grant CCR-9010517, and grants from Mitsubishi and OTL.

[†]Institute of Computer Science, Hebrew University. Supported by the Wolfson Research Awards administered by the Israel Academy of Sciences and Humanities and by U.S.A.-Israel BSF 89-00126.

[‡]Institute of Computer Science, Hebrew University.

Theorem 1.1 *The connected components of an undirected graph can be computed on a randomized EREW PRAM in $O(\log n)$ time with high probability*[1] *using $(m + n^{1+\epsilon})/\log n$ processors for any fixed $\epsilon > 0$.*

The running time of this algorithm is optimal, as the results of [CDR86] and [DKR94] imply a lower bound of $\Omega(\log n)$ time even on a randomized CREW PRAM. For graphs that are not too sparse, *i.e.* with $\Omega(n^{1+\epsilon})$ edges, the processor costs are optimal as well, as the total work remains linear in the input size. For sparse graphs, using a linear number of processors slightly increases the running time:

Theorem 1.2 *The connected components of an undirected graph can be computed on a randomized EREW PRAM in $O(\log n \log\log n)$ time with high probability with $(m+n)/\log n$ processors.*

This is of course within an $O(\log\log n)$ factor of being work-optimal. An important related open problem is to design a *deterministic* $O(\log n)$ time algorithm. We have made some progress in this direction:

Theorem 1.3 *The connected components of an undirected graph can be computed on a deterministic EREW PRAM in $O(\log^{1.5} n)$ time using $m+n$ processors.*

The running time of this deterministic algorithm matches those of [JM91] and [NSW92]. It improves upon [JM91] by working in the more restricted EREW model instead of in the CREW model. It improves upon [NSW92] by requiring only a linear number of processors instead of a large polynomial number of processors. This last result was proved independently (using a different method) by [JM92].

After publication of the preliminary version of this paper [KNP92], several improvements were given. Chong and Lam [CL95] gave an $O(\log n \log\log n)$ deterministic algorithm that uses $m+n$ processors. Halperin and Zwick improved our methods to yield first [HZ94] an optimal randomized algorithm for connected components that runs in $O(\log n)$ time with a linear number of processors, and subsequently [HZ6] an optimal randomized algorithm for finding a spanning forest of the graph (note that our algorithm does not find spanning forests).

In the following sections, we present the connectivity algorithm. To simplify the exposition, we first present a randomized algorithm that uses $m+n$ rather than $(m+n)/\log n$ processors. We give a general overview of the algorithm and then fill in the details and provide proofs of correctness. We then discuss the changes needed to make the algorithm deterministic. The modifications needed to reduce the processor cost by an additional factor of $\log n$ are somewhat complex and are left to a later section.

2 Overview of The Algorithm

The algorithm is based on a simple and well known idea: repeatedly find groups of connected vertices in the graph and contract (*i.e.* merge) each group into a single vertex, finishing when each connected component is contracted to a single vertex. The question lies in how to find these connected groups.

Suppose we are fortunate, and the minimum degree of the graph is large. Let $N(i)$ be the set of vertices adjacent to i (including i). The following procedure can be applied, using a processor for each vertex and each edge:

[1]By "high probability" we mean that the probability of the event not happening can be made at most $n^{-\delta}$ for any fixed $\delta > 1$ without affecting the orders of run times.

1. Each vertex looks at all vertices within distance two of itself, *i.e.* $N(N(i))$. If it finds a larger numbered vertex than itself, it makes this vertex its *parent*. Any vertex that fails to find a parent becomes a *leader*.

2. The selection of parents has created a group of trees with leaders at the roots (note that the tree edges need not be graph edges). Each tree is now contracted to a single vertex.

Assuming that the minimum degree is large, this process yields a much smaller graph, as the following lemma shows:

Lemma 2.1 (Neighborhoods) *If all neighborhoods $N(i)$ have size at least s, then at most n/s leaders can exist.*

Proof: The distance between two leaders must exceed 2. Thus the neighborhoods of two leaders are disjoint, and therefore at most n/s leaders remain. □
The contraction of the trees does not change the connected components, as can be seen from the following lemma:

Lemma 2.2 *In the contracted graph, two leaders are connected iff they were connected in the old graph.*

Proof: Note that two leaders are adjacent iff each had a descendant such that the descendants were adjacent. The lemma then follows from the fact that all vertices are necessarily connected to their leaders. □

The running time of the algorithm will depend on the number of rounds needed to contract every connected component into a single vertex. Since the reduction is based on neighborhood size, this number of rounds depends on the minimum degree s of the graph. The problem is that the minimum degree of the graph may be small, and therefore the procedure described above may fail to reduce the size of the graph significantly. We will show how to solve this problem by "imagining" additional edges in the graph in order to make the neighborhoods large. As long as the imaginary edges connect vertices that are connected in the original graph, the two lemmas given above continue to hold. Similar ideas are explored in [BR91] and [NSW92]. The remainder of this paper is dedicated primarily to the question of how to construct quickly and in parallel a large neighborhood for each vertex in the graph.

Our approach to this question began with the following observation. It is known that an EREW PRAM with a polynomial number of processors can simulate any logspace algorithm in $O(\log n)$ time (see [KR90]). This extends to the fact that a randomized EREW PRAM can simulate randomized-logspace algorithms.[2] Since a randomized-logspace algorithm for connectivity is known [AKL*79], a randomized EREW algorithm follows.

Unfortunately, the parallelization of the random walk algorithm of [AKL*79] requires $\Omega(mn^2)$ processors. The reason so many processors are needed is that a random walk of length $\Omega(mn)$ must be taken, to be sure of covering the entire graph. Thus the approach of [AKL*79] does not directly suggest an efficient algorithm. However, an important idea can be extracted from this approach, namely that a random walk visits a large number of vertices relatively quickly. We thus explore the use of *short* walks on the graph.

[2]In fact, the results in [Nis93] imply that any randomized-logspace algorithm with bounded two-sided error can be simulated with *zero-error* by a randomized EREW PRAM in $O(\log n)$ time using a polynomial number of processors.

Consider taking a walk of some length p from each vertex of the graph by traveling along edges of the graph. Using the vertices encountered along each walk as the neighborhood of the walk's starting vertex, we will apply the contraction procedure described above to reduce the size of the graph by a significant factor. This procedure will be called a *walk phase of length p*. In the randomized algorithm, the walk is a random walk; in the deterministic algorithm, the walk is based on a deterministic traversal sequence. A walk phase takes $O(\log n)$ time to simulate in parallel, but since the walk phases construct large neighborhoods, a very small number of them suffice to complete the algorithm. In more detail, consider the following procedure:

1. From each vertex $1 \leq i \leq n$, take a walk of length p. Let $W(i)$ be the *itinerary* of i, *i.e.* the set of vertices seen on the walk that starts at i.

2. Consider the edges defined by the walks, so that there is an edge $\{i, j\}$ if $j \in W(i)$ or $i \in W(j)$. These edges clearly connect vertices that are connected in G.

3. Using these "walk edges" to define the vertex neighborhoods, each vertex examines its neighborhoods, as was described at the start of this section, to find a parent or become a leader.

4. To contract the resulting collection of trees, each vertex finds the leader in its tree of parents and transfers all its edges to the leader (*i.e.*, replaces each edge $\{i, j\}$ in G by an edge from i's leader to j's leader).

When a walk phase is finished, we have a new graph G' whose vertices are the leaders in the old graph. Lemmas 2.1 and 2.2 tell us that G' is smaller than G and embodies the same connectivity information.

The connectivity algorithm itself will repeat the walk phases until the resulting graph has no edges. Each remaining vertex then represents the connected component containing that vertex. Every other vertex will have dropped out after selecting some vertex as its leader and giving that vertex its edges. The leader choices of the vertices form a forest—the root of each tree is one of the connected component representatives, and the vertices in each tree are a single connected component of the graph. Tree contraction can now be used to let each vertex identify its connected component representative.

To make the algorithm run quickly, we need to finish in a small number of walk phases. From this description, it can be seen that all we need in order to implement the algorithm is:

- A walk that visits a large number of vertices, and

- A way to simulate a walk phase quickly in parallel.

We now show in detail how to achieve these two goals.

3 Implementing a Walk Phase

In the course of the following discussion of the implementation of the algorithm, assume that G is totally connected. The results we wish to prove then follow by independently considering the action of the algorithm on each connected component of the graph.

The key question that must be solved is how to construct a walk that visits a large number of vertices. Using randomization, the solution is straightforward. Knowing that a random walk expects to cover all n vertices of a graph in time $n^{O(1)}$, we will deduce that a random walk of length

p visits $p^{\Omega(1)}$ vertices with high probability. This reduces the implementation of a walk phase to the problem of simulating a random walk from each vertex in parallel.

It is well known that CRCW can be simulated on an EREW machine with an $O(\log n)$ slowdown in running time and no increase in processor cost ([KR90], pp. 894-895). Since we wish the walk phase to have running time $O(\log n)$, we feel free to say that "processors concurrently read or write," so long as this occurs only a constant number of times.

We now discuss the details of implementing a walk phase of length p using $m + pn$ processors.

3.1 Data Structures and Processor Allocation

Each vertex i has a list L_i of edges leaving i. The edge between i and j appears as (i, j) in L_i and as (j, i) in L_j. The edge lists are stored contiguously in one array L of length $2m$, sorted by order of lists $L_1, L_2, ..., L_n$.

Each vertex i also has two variables, first_i and last_i, which indicate the beginning and end of the list L_i in L. It is easy to determine the number of neighbors of i by computing $\mathit{last}_i - \mathit{first}_i + 1$.

The algorithm uses an $n \times p$ array WALK to simulate random walks of length p. Two more arrays, MAX of dimensions $n \times p$, and PARENT of length n, are used to find the leaders of the graph.

We use $O(\log n)$ time to redistribute the processors at the beginning of each walk phase:

- There are p processors assigned to each vertex. Let $P_{i,1}, P_{i,2} ... P_{i,p}$ be the processors assigned to vertex i, for $i = 1, ..., n$.
- One processor is assigned to every edge. Denote these processors by $P_1, P_2, ..., P_m$. These processors will be called *edge processors.*

Therefore the total number of processors is $m + pn$. Notice that after each walk phase the number of vertices of the graph reduces, and therefore in each walk phase we can allocate more processors per vertex. This will allow us to increase the length p of a walk phase, and thus to contract the connected components even faster.

We now turn to the details involved in executing the four steps outlined in the overview of the algorithm.

3.2 Step 1: Simulating the Random Walk

We wish to simulate the process of taking a random walk of length p simultaneously from all vertices of G. For each vertex $1 \leq i \leq n$, and each $1 \leq t \leq p$, processor $P_{i,t}$ chooses a neighbor of vertex i uniformly at random, and writes it into $\mathit{WALK}[i, t]$. Each processor does so using three concurrent reads of first_i, last_i and L.

Consider the random variables u_t^i defined by:

$$\begin{aligned} u_1^i &= i \\ u_{t+1}^i &= \mathit{WALK}[u_t^i, t] \quad \text{for} \quad t = 1, ..., p. \end{aligned}$$

By the choice of the $\mathit{WALK}[i, t]$ values, the random variable u_t^i is a random walk starting at vertex i, for each $1 \leq i \leq n$. The random walks with different sources are not independent, but this will not affect the analysis. As mentioned in the overview, let the itinerary $W(i)$ be the set of vertices encountered on the random walk from i. Figure 1 shows a filled WALK array in which, for example, $W(8) = \{7, 8, 9\}$ (in the algorithm, each walk step moves from one vertex to a different vertex, but in our picture, for the sake of clarity, we have drawn some horizontal edges that imply walk steps that stand still). We will show later that all the $W(i)$ are large, i.e., of size exceeding $p^{\Omega(1)}$.

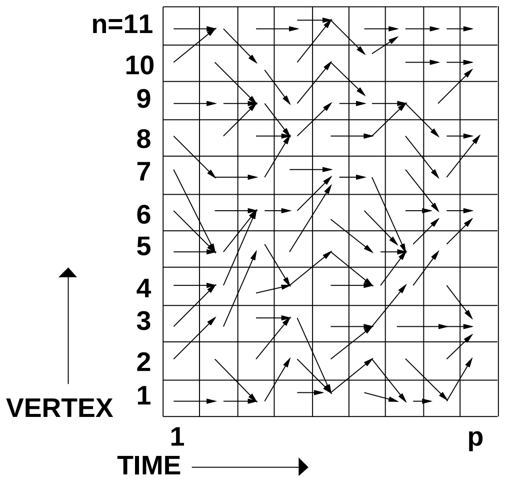

Figure 1: The WALK array.

3.3 Step 2: Finding Neighborhoods

As stated in Step 2 of the outline, consider the *walk edges* defined by including (i, j) if $i \in W(j)$ or $j \in W(i)$. In Step 3 each vertex looks for a parent among vertices up to two walk edges away. These edges are not actually constructed; instead, each vertex deduces the information it needs directly from the *WALK* array. Furthermore, the edges considered are actually a superset of the walk edges, which will define larger neighborhoods $N(i) \supset W(i)$. But it will still be true that i is connected to all vertices in $N(i)$. Since larger neighborhoods cause a greater reduction in the size of the graph, this use of more edges can only help.

The values placed in the walk array in Step 1 can be seen to define a collection of trees (the values provide parent pointers). We let the neighborhood of a vertex be the vertices to which it is connected by one of these trees. More formally, for each $i = 1, ..., n$, let $T_{i,t}$ be the set of array entries $[j, t']$, which are in the tree containing the entry $[i, t]$, and let $T_i = \bigcup_t T_{i,t}$. Define $N(i) = \{j \mid (\exists t)[j, t] \in T_i\}$ to be the neighborhood of vertex i. In other words, imagine an edge from i to j whenever i and j share a tree. Note that if the random walk from i encounters j then $[j, t] \in T_i$ for some t, so $W(i) \subset N(i)$. Also, since each tree edge corresponds to a step in a random walk, and thus to an edge in G, all members of $N(i)$ are necessarily connected to i.

Let H denote the graph with vertex set V but with edges defined by the neighborhoods $N(i)$. This is the graph that will be used to find connected sets of vertices to contract.

3.4 Step 3: Choosing Leaders

We now implement the process of choosing a maximum parent of distance at most two in H, as described in the overview of Section 2. This is achieved by calling twice the following procedure *Max-Neighbor*. The first call to this procedure chooses for each vertex i the maximum vertex in $N(i)$ (which corresponds to finding the maximum vertex of distance one from each vertex on the graph H). The second call finds the maximum vertex chosen by any vertex in $N(i)$ (which corresponds to finding the maximum vertex of distance two from each vertex on the graph H). Initialize the array $PARENT$ to $PARENT[i] = i$, and then call the following procedure twice:

Procedure *Max-Neighbor:*

1. For each $i = 1, ..., n$ and $t = 1, ..., p$, set $MAX[i,t] = \max_{[j,t'] \in T_{i,t}} PARENT[j]$.
2. For each $i = 1, ..., n$ set $PARENT[i] = \max_t MAX[i,t]$.

The first step iteration of this process labels each vertex with its largest "neighbor," so the second iteration labels vertices with their largest neighbor at distance two. In other words, at the end of this process $PARENT[i]$ contains the parent of vertex i. Vertex i is a leader if $PARENT[i] = i$.

Implementing *Max-Neighbor* is straightforward. Step 2 is trivial. Step 1, maximizing over a tree, can be implemented, using Euler tour techniques on the array $WALK$, in time $O(\log n)$ using np processors (see [KR90], pp. 879-883). The only nonstandard detail is that our filling up of the walk array has created trees with unidirectional edges, while the Euler tour method requires bidirectional edges. To build these edges, proceed as follows. Copy the $WALK$ array, and then sort the edges $([i,t],[j,t+1])$ in the $WALK$ array according to their second endpoints. We can do this in $O(\log np)$ time using np processors, either by applying Cole's sorting algorithm [Col88], or via a simple bucket sort using $(np)^2$ space. After the sort, all the edges that point to a particular position in the $WALK$ array are grouped together for application of the Euler tour technique.

3.5 Step 4: Create the New Graph G'

In this final step we must construct the new graph G' whose vertices are the leaders in the graph G. The selection of parents in procedure *Max-Neighbor* created a group of trees (not to be confused with the trees in the $WALK$ array) with leaders at the roots. Each vertex now finds the leader at the root of its tree by using Euler tours as before [KR90]. We can now create the new smaller graph $G' = (V', E')$. The set of vertices V' is the set of leaders, *i.e.* $V' = \{leader(i) \mid i \in V\}$. The set of edges of G' is obtained by transforming each edge (i,j) of E to an edge $(leader(i), leader(j))$, *i.e.* $E' = \{(leader(i), leader(j)) \mid (i,j) \in E\}$.

To construct the set E', each edge processor P_k for $k = 1, ..., m$ concurrently reads the leaders of each of its endpoints and renames its edge appropriately. If P_k is handling edge (i,j), then P_k checks if $leader(i) = leader(j)$. If so, this edge has been contracted and is now useless, so P_k writes 0 at $L[k]$. If not, it writes the edge $(leader(i), leader(j))$ at $L[k]$.

Next sort L lexicographically by left and right endpoints in $O(\log n)$ time using Cole's sorting algorithm [Col88] or a bucket sort. The renaming may yield multiple copies of some edges. These must be removed because otherwise the random walk becomes "biased" towards visiting vertices that are connected by many edges; our analysis requires that the random walk is not biased. To remove these multiple edges, each edge processor P_k looks to its left at L. If $L[k] = L[k-1]$, then P_i writes a 0 at $L[k]$. Now compact the array L using standard parallel compaction ([KR90], pp. 875-876).

Next it is neccessary to update the $first_i$ and $last_i$ variables. To do so, first set $first_i = -1$ (using the processors $P_{i,1}$). Then each edge processor P_k looks to its left (right) at L, and if it is at the beginning (end) of the edge list of some processor i, it updates $first_i$ ($last_i$). Afterwards, any vertex that still has $first_i = -1$ must have no incident edges. Such isolated vertices are marked as the representatives of connected components and removed. It should be noted that the new graph G' still has at most m edges.

Clearly, all the operations described above can be done in $O(\log n)$ time. Thus, we have proved:

Lemma 3.1 *A walk phase of length p can be implemented in $O(\log n + \log p)$ time using $m + pn$ EREW processors.*

4 Iterating the Walk Phase

Now that we have shown the required time and processor bounds, it remains to show that the new graph G' has significantly fewer vertices, and that as a consequence the algorithm terminates in a small number of walk phases. We require the following corollary to the known results regarding the cover time of random walks on graphs. This lemma was first observed by Linial ([Lin]). For completeness we sketch the proof.

Lemma 4.1 *Let G be an undirected graph. Let v be any vertex in G that is contained in a connected component of at least t vertices. Then the expected time needed for a random walk starting from v to see t vertices is $O(t^4)$.*

Proof: Define the random variable X_t to be the time it takes a random walk that starts at v to see t vertices. Assume that by time X_t we saw the set of vertices C_t. Let $w \notin C_t$ be a vertex that is adjacent to some vertex in C_t. Then the expected time to cover the graph $C_t \bigcup w$ (and thus see a new vertex) is $O(t^3)$ if we don't leave $C_t \bigcup w$. If we do leave it, then we shall see a new vertex even sooner. Hence $E(X_{t+1}) = E(X_t) + O(t^3) = O(t^4)$. □

Recently it was shown by Barnes and Feige ([BF93]) that $O(t^3)$ expected time is sufficient to see t vertices. We can now obtain:

Lemma 4.2 *After a walk phase of length p, for every vertex $i = 1, ..., n$, the itineraries satisfy $|W(i)| = \Omega((\frac{p}{\log n})^\alpha)$ with high probability, where $\alpha = 1/4$.*

Proof: Consider the walk to be a composition of $\Omega(\log n)$ "subwalks" of equal length $\Omega(\frac{p}{\log n})$. Call each subwalk *good* if it visits $\Omega((\frac{p}{\log n})^\alpha)$ vertices. By the Markov inequality and Lemma 4.1, each subwalk has a constant probability of being good. This is true even if we condition on the outcomes of previous subwalks. Thus all the subwalks fail to be good with polynomially small probability. □

The result of [BF93] allows us to take $\alpha = 1/3$, thus improving the constant factors in the following analysis.

Corollary 4.3 *A walk phase of length p reduces the number of vertices in a graph by a factor of $\Omega((\frac{p}{\log n})^\alpha)$ with high probability.*

Proof: Consider the above two lemmas, and the fact that $W(i) \subset N(i)$. Now apply the Neighborhoods Lemma (2.1). □

We can now analyze the running time.

Lemma 4.4 *Using $m + pn$ processors, with high probability we can fully identify the connected components of the graph with $O(\log(\log n / \log p))$ walk phases.*

Proof: Assume for now that $p > \log^2 n$. The hypothesis gives us at least p processors per vertex. Running a walk phase of length p yields a graph of $O(n(\frac{\log^\alpha n}{p^\alpha}))$ vertices. On this graph redistribute the processors to get

$$\frac{np}{n \log^\alpha n/p^\alpha} = \frac{p^{1+\alpha}}{\log^\alpha n} > p^{9/8}$$

processors per vertex. Thus the number of processors per vertex after t walk phases is described with high probability by the recurrence

$$p_{t+1} > p_t^{9/8}$$

with solution

$$p_t > p^{(9/8)^t}.$$

Thus p_t exceeds pn within $O(\log(\log n/\log p))$ steps. Since this implies that all the processors are assigned to one vertex, the algorithm must be finished at this point. Therefore this is the maximum expected number of walk phases needed.

There remains the detail of what to do if initially $p < \log^2 n$. To handle this case, note that even if $p = 1$, so that the random walks are in fact just inspections of a single neighbor, the neighborhoods still have size two. Thus the size of the graph is still reduced by a factor of two in each walk phase. Therefore, $O(\log(\log^2 n/p)) = O(\log(\log n/\log p))$ walk phases suffice to raise p to $\log^2 n$ and thus reduce to the previous case. □

Since each walk phase is simulated in $O(\log n)$ time, the overall running time of the algorithm is $O(\log n \log(\log n/\log p))$. Theorems 1.1 and 1.2 follow immediately, up to a factor of $\log n$ in the processor count that is removed in Section 7.

5 Using Fewer Random Bits

Randomness is used in our algorithm only to construct random walks. We show how to restrict this use of randomness to $O(n^\epsilon)$ bits, for any $\epsilon > 0$. Note first that once we have n^ϵ processors per vertex, and can simulate random walks of length n^ϵ, there is no need to reassign processors to vertices, since an additional $O(1/\epsilon)$ walk phases of length n^ϵ will finish the problem. Therefore assume that random walks never exceed length n^ϵ. Now observe that a walk phase of length p needs only $p \log n$ random bits. Two entries in the *WALK* array need be independent only if it possible for a walk defined in the array to encounter both of them. Therefore entries $WALK[i,t]$ and $WALK[i',t]$, $i \neq i'$, can use the same random seed in selecting edges, since a particular walk is only at one place at any particular time.

Corollary 5.1 *Connected components can be found in time* $O(\log n \log(\log n/\log p) + (\log n)/\epsilon)$ *using* $m + pn$ *processors and* $O(n^\epsilon)$ *random bits.*

6 The Deterministic Version

Our techniques can also be used to obtain a deterministic algorithm for the EREW PRAM that runs in $O(\log^{1.5} n)$ time using $m + n$ processors. This improves on the deterministic $O(\log^{1.5} n)$ time algorithm of [NSW92], and matches an independent result of [JM92]. As in [NSW92], we use a universal sequence instead of a random walk. It will be convenient to consider a generalization of the universal sequences of [AKL*79] to allow walks on non-regular graphs.

Definition 6.1 *A graph G with at most r vertices will be called* r-labeled *if the edges adjacent to each vertex are labeled with unique numbers from $\{1, 2, ..., r\}$.*

Definition 6.2 *Given a string $\sigma \in \{1, 2, ..., r\}^*$ and an r-labeled graph G, a* walk according to σ *starting from a given vertex will follow edge labeled i at step j if $\sigma_j = i$. If $\sigma_j = i$ and none of the edges leaving the current vertex are labeled i, the walk will remain in that vertex.*

Definition 6.3 *A string $\sigma \in \{1, 2, ..., r\}^*$ is called an* r-universal sequence *if for every graph G with at most r vertices and any r-labeling of G, a walk according to σ visits all the vertices of G, regardless of the starting vertex.*

By following the proofs of [AKL*79], [BNS92] and [Nis92], it is not difficult to see that the construction of [Nis92] yields an r-universal sequence of length $r^{O(\log r)}$ in our general sense. We need only the following two properties:

Theorem 6.4 ([Nis92]) *An r-universal sequence of length $l = r^{O(\log r)}$ can be generated by an EREW PRAM in $O(\log l)$ time using $O(l \log l)$ processors.*

Lemma 6.5 *For any undirected connected graph G with at least r vertices, and for any vertex v in G, a walk along an r-universal sequence σ, starting from v visits at least r vertices of G.*

Proof: Label G so that each vertex of degree d is labeled with the numbers $\{1, 2, ..., d\}$. Assume the claim is false and σ visits less than r vertices.

Let C_r be the graph induced by all the vertices σ visits. Let $w \notin C_r$ be a vertex adjacent to some vertex $v' \in C_r$, such that the edge (v', w) is labeled with a number less than r (this is possible since v' has at most $r - 2$ neighbors in C_r). Then the graph $C_r \cup w$ is an r-labeled graph with at most r vertices, and thus a walk according to σ should cover it and thus visit w; a contradiction. $\square$

The deterministic algorithm proceeds as follows: instead of taking a random walk from each vertex, generate an r-universal sequence and then walk along this sequence. The parameter l is chosen such that the length of the resulting universal sequence is p (where p is the number of processors allotted to each vertex in the graph); thus $r = 2^{O(\sqrt{\log p})}$. We are thus assured by the Neighborhoods Lemma (2.1) that the number of vertices in the graph at the next round shrinks by a factor of at least r.

Letting p_i be the number of processors allotted to each vertex at iteration i, we argue as in the random walk case. We have the following recursion:

$$\begin{aligned} p_1 &= 2 \\ p_{i+1} &= p_i \cdot 2^{\sqrt{\log p_i}} . \end{aligned}$$

Lemma 6.6 *Let p_1=2 and $p_{i+1} = p_i \cdot 2^{\sqrt{\log p_i}}$. Then $p_j = n$ for some $j = O(\sqrt{\log n})$.*

Proof: Let $q_i = \log p_i$. Therefore, $q_1 = 1$ and $q_{i+1} = q_i + \sqrt{q_i}$. Then for every i, $q_{(i+\sqrt{q_i})} \geq 2q_i$. Thus, the time to reach $q_j = \log n$ is at most $\sqrt{1} + \sqrt{2} + \sqrt{4} + \sqrt{8} + \ldots + \sqrt{\log n} = O(\sqrt{\log n})$. $\Box$

As a result of this lemma, we can conclude that after $O(\sqrt{\log n})$ walk phases the graph is contracted to a single vertex.

Theorem 1.3 follows immediately. Observe that starting with (polynomially many) more processors does not decrease the running time in this case.

Corollary 6.7 *If an n-universal sequence of polynomial length can be generated deterministically in $O(\log n)$ time, then connected components can be found in $O(\log n)$ time deterministically using $m + n^{1+\epsilon}$ processors for any fixed ϵ.*

7 Approaching Optimal Work

The algorithms described above perform work that exceeds the optimal by a factor of $O(\log n \log \log_p n)$. Here we reduce this factor to $O(\log \log_p n)$, and how in fact the $O(\log n \log \log_p n)$ running time can be achieved with $(m + pn)/\log n$ processors (this will be optimal for $p = n^\epsilon$). We begin with the assumption that $p > \log^2 n$, and later show how this assumption can be removed. Assume without loss of generality that $m \geq n/2$, since an initial step of the algorithm can use $n/\log n$ processors to remove any vertices with no edges.

7.1 Assuming $p > \log^2 n$

Observe first that the difference between using pn and $pn/\log n$ processors can be ignored, since for $p > \log^2 n$, $\log \frac{\log n}{\log p} = \Theta(\log \frac{\log n}{\log(p/\log n)})$. Thus the only need is to perform the m-processor steps with $m/\log n$ processors.

To do so, note that m processors are used for only one purpose: to update the edge list after leaders have been identified. There are three phases in this update process:

1. Replace the edge (i, j) by the edge $(\mathit{leader}(i), \mathit{leader}(j))$.

2. Detect and remove *dead* edges, namely those that now have the form (i, i) because both endpoints chose the same leader.

3. Sort the remaining edges to remove duplicates and create edge lists for the contracted graph.

The real sticking point in this process is Step 3. Since potentially nearly m edges may remain in the contracted graph, and since sorting them requires $\Omega(m \log m)$ work, it is unclear how Step 3 can be performed.[3] Getting around this problem is the main topic of this section.

We begin by showing that Steps 1 and 2 are easy to perform with $m/\log n$ processors. We allocate the processors according to the following scheme. Break the list of edges into sequential blocks of size $\log n$, and assign one processor to each block. Recall that the edge list is sorted by the first vertex in each edge. Therefore, the i^{th} block contains first some of the edges of some first vertex f_i, then all the edges of some set of vertices V_i, and finally some of the edges of a last vertex l_i.

The advantage of this assignment is that it allows us to simulate, in $O(\log n)$ time, a single concurrent read by each edge (i, j) of some information from vertex i, and similarly, in $O(\log n)$

[3]Possibly some form of bucket sort could be used to circumvent the sorting lower bound.

time, concurrent writes (with any conflict resolution scheme desired) by each edge (i,j) to vertex i. To simulate the read, proceed as follows. First use the standard concurrent read simulation to let processor i read from f_i and l_i into its local memory; these two reads take $O(\log n)$ time. Then each processor updates the $\log n$ edges it is responsible for—these updates now require only exclusive reads from its local copies of f_i and l_i or from the global values in the vertices V_i. The concurrent write simulation is similar.

Because of this simulation, we will freely use instructions of the form "each edge (i,j) reads from or writes to its vertex i," with the understanding that each such step actually takes $O(\log n)$ time.

One other small change is that it is necessary for each edge (i,j) to have a pointer to its *twin* edge (j,i) that is maintained as edges are moved around.

It is now easy to perform Step 1 in $O(\log n)$ time—each edge (i,j) concurrently reads $\mathit{leader}(i)$, and replaces i by $\mathit{leader}(i)$ in (i,j) and in its twin (j,i). Step 2 can be performed easily by $m/\log n$ processors in $O(\log n)$ time with a standard array compaction algorithm.

It remains to deal with the difficulty of Step 3. The approach we take is to ensure that the number of edges remaining after Step 2 (counting duplications) is small, so that few processors are needed to perform Step 3. We use the following lemma:

Lemma 7.1 *If each edge of a graph is selected independently with probability q, and connected components induced by the selected edges are contracted in the original graph, then with high probability the number of edges of the contracted graph is $O(n \ln n/q)$.*

In [KKT95], the number of remaining edges is shown to be $O(n/q)$ with high probability; this does not improve our application.

Proof: The number of edges in the contracted graph is just the number of edges crossing between the different connected components induced by the selected edges. The number of different arrangements of connected components is certainly no more than the number of ways to partition the set of n vertices into at most n groups, namely n^n. For any given partition that cuts k edges, the probability that no crossing edge is chosen is $(1-q)^k \approx e^{-kq}$. The probability that k edges are cut in the partition resulting from the connected component construction is just the probability that for some partition with at least k crossing edges, no one of these k edges is chosen. This is at most $n^n e^{-kq} = e^{n \ln n - kq}$, which is negligible when $kq = \Omega(n \ln n)$. □

We therefore use the following approach: given an m-edge graph, choose $m/\log n$ edges at random, and using $(m+pn)/\log n$ processors and the basic algorithm, compute connected components in this sampled graph in $O(\log n \log\log_p n)$ time. This labels all vertices in a given connected component of the sampled graph with a single vertex. If the label of a vertex is considered to be its choice of a leader, then we can contract the original graph as if a walk phase has been performed. We use $m/\log n$ processors to relabel the edges and remove dead edges as was described at the beginning of this section. Lemma 7.1 shows that at this point $O(n\log^2 n)$ edges remain, so $O(n \log^2 n)$ processors suffice to perform Step 3, sorting the edges and removing duplicates. We can finish the calculation by finding connected components in the resulting contracted graph; since the number of edges in the graph is $O(n\log^2 n)$, and since by assumption the number of processors is $pn > n\log^2 n$, this can be done in $O(\log n \log\log_p n)$ time with the available processors.

This shows that when $p > \log^2 n$ connected components can be found in $O(\log n \log\log_p n)$ time using $O((m+pn)/\log n)$ processors.

7.2 Removing the Assumption

We now handle the case $p < \log^2 n$. With such a value of p, the running time that we must achieve is $O(\log n \log\log n)$. Assume that in fact $p = 1$, since this merely restricts us further.

It suffices to find a procedure that, in $O(\log n)$ time and using $(m+n)/\log n$ processors, reduces the number of vertices by a constant factor. After $O(\log\log n)$ phases of this procedure, the graph will have $n' = O(n/\log^3 n)$ vertices. The algorithm of the previous section can be applied to solve this graph in $O(\log n \log\log n)$ time using $O(m/\log n + n' \log^2 n) = O((m+n)/\log n)$ processors.

We use the same allocation of processors to blocks of $\log n$ edges that was used previously, allowing the same simulation of concurrent reads and writes. The following procedure reduces the number of vertices in the graph by at least half in $O(\log n)$ time using $(m+n)/\log n$ processors:

1. For each vertex, compute the identity of its largest and smallest neighbors. To find the maximum, each edge (i, j) concurrently writes j to vertex i, letting multiple writes yield the maximum value written. Then do the same for the minimum value.

2. It is necessarily the case that either half the vertices have a larger neighbor or that half the vertices have a smaller neighbor. Which case we are in can be determined in $O(\log n)$ time using $n/\log n$ processors and the information from the previous step. Assume the first case; the other can be handled the same way.

3. Vertices with no larger neighbors are *leaders,* as before. The largest vertex j that is a neighbor of a non-leader vertex i becomes the *parent* of i as before. We mark the edge (i, j) in i's edge list, as well as its twin edge (j, i) in j's edge list.

4. Each vertex i can now identify its children: it examines its edges (i, j) and checks which ones were marked in the previous step.

5. Since each vertex can identify both parent and children, Euler tour techniques can be used to transform each tree of parent pointers into a list of the vertices in the tree, and to let each vertex identify its leader.

6. Having identified leaders, relabel the edge list precisely as was done in Section 7.1

7. Since there is a linked list of vertices for each leader, and since each vertex has a contiguous list of its edges, we have an implicit linked list of all the edges that will be incident to a given leader after the graph is contracted. This allows us to use optimal list ranking to count the number of edges belonging to each leader in $O(\log n)$ time with $n/\log n$ processors, and rank them.

8. Take a length n array, and write into the k^{th} position the number of edges belonging to k if k is a leader, and 0 otherwise. Then, using array compaction, each leader can find the number of edges belonging to the leaders that precede it in the contracted graph.

9. The list ranking information just described suffices to determine the position each edge should be copied to in the edge array of the contracted graph, so the copying can be done in $O(\log n)$ time.

Note that duplicate edges do not affect this procedure, so they can be ignored until we have reduced to the n-processor scenario. Lemma 7.1, used to reduce to $m/\log n$ processors, holds even when duplicate edges exist. It will be necessary to remove duplicates from the set of $m/\log n$ edges that we sample for the first application of the random walk algorithm, but the $m/\log n$ processors that we have are sufficient to do this by sorting.

8 Conclusion

Since the publication of the preliminary version of this paper [KNP92], Halperin and Zwick [HZ94] have used it to derive a work and processor optimal randomized EREW connected components algorithm. The obvious remaining open problem is to find a deterministic $O(\log n)$ time EREW algorithm for connected components. One way to work towards this goal is to improve on the bounded space universal traversal sequence construction which is used in our deterministic algorithm, since any improvement in the space needed immediately yields a faster algorithm. There also remains an intriguing gap in the CRCW model, where the best known algorithm has running time $O(\log n)$ but the best known lower bound is $\Omega(\log n / \log\log n)$.

9 Acknowledgments

Thanks to Daphne Koller, who made our collaboration possible. Thanks also to Rajeev Motwani and Serge Plotkin for helpful discussions.

References

[AKL*79] R. Aleliunas, R. M. Karp, R. J. Lipton, L. Lovasz, and C. Rackoff. Random Walks, Universal Traversal Sequences and the Complexity of Maze Problems. In *Proceedings of the 20th Annual Symposium on the Foundations of Computer Science*, pages 218-223. IEEE Computer Society Press, October 1979.

[AS87] B. Awerbuch and Y. Shiloach. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. In *IEEE Transactions on Computers*, C-36(10), pages1258-1263, October 1987.

[BF93] G. Barnes and U. Feige. Short Random Walks on Graphs. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 728-737, ACM Press, May 1993

[BNS92] L. Babai, N. Nisan and M. Szegedy. Multiparty Protocols, Pseudorandom Generators for Logspace, and Time-Space Trade-offs. In *Journal of Computer and System Sciences,* 45(2), pages 204-232, 1992.

[BR91] G. Barnes, and W. L. Ruzzo. Deterministic Algorithms for Undirected $s-t$ Connectivity using Polynomial Time and Sublinear Space. In *Proceedings of the 23rd ACM Symposium on Theory of Computing*, pages 48-53, ACM Press, May 1991.

[CDR86] S. Cook, C. Dwork and R. Reischuk. Upper and Lower Bounds for Parallel Random Access Machines without Simultaneous Writes. In *SIAM Journal of Computing*, 15(1), pages 87-97, February 1986.

[CL95] K. Chong and T. Lam. Finding Connected Components in $O(\log n \log\log n)$ Time on the EREW PRAM. In *Journal of Algorithms*, 18(3), pages 378-402, May 1995.

[CLC82] F. Y. Chin, J. Lam and I. N. Chen. Efficient Parallel Algorithms for some Graph Problems In *Communications of ACM,* 25(9), pages 659-665, September 1982.

[Col88] R. Cole. Parallel Merge-Sort. In *SIAM Journal on Computing*, 17(4), pages 770-785, August 1988.

[CV91] R. Cole and U. Vishkin. Approximate Parallel Scheduling. II. Applications to Logarithmic-Time Optimal Parallel Graph Algorithms. In *Information and Computation (formerly Information and Control),* 92(1), pages 1-47, May 1991.

[DKR94] M. Dietzfelbinger, M. Kutylowski, R. Reischuk. Exact Lower Time Bounds for Computing Boolean Functions on CREW PRAMs. In *Journal of Computer and System Sciences,* 48(2), pages 231-254, April 1994. A preliminary version appeared in SPAA 1992.

[Gaz91] H. Gazit. An Optimal Randomized Parallel Algorithm for Finding the Connected Components of a Graph. In *SIAM Journal on Computing,* 20(6), pages 1046-1067, 1991. A preliminary version appeared in FOCS 1986.

[HCS79] D. S. Hirschberg, A. K. Chandra and D. V. Sarwate. Computing Connected Components on Parallel Computers. In *Communications of ACM,* 22(8),pages 461-464, August 1979.

[HZ94] S. Halperin and U. Zwick. An Optimal Randomized Logarithmic Time Connectivity Algorithm for the EREW PRAM. In *Proceedings of the* 6^{th} *Annual ACM-SIAM Symposium on Parallel Algorithms and Architectures,* pages 1-10, ACM 1994.

[HZ6] S. Halperin and U. Zwick. Optimal Randomized EREW PRAM Algorithms for Finding Spanning Forests and other Basic Graph Connectivity Problems. In *Proceedings of the* 7^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms,* pages 438-447, ACM-SIAM, January 1996.

[JM91] D.B. Johnson and P. Metaxas. Connected Components in $O(\log^{3/2}|V|)$ Parallel Time for the CREW PRAM. In *Proceedings of the* 32^{nd} *Annual Symposium on Foundations of Computer Science*,pages 688-697, IEEE Computer Society Press, October 1991.

[JM92] D.B. Johnson and P. Metaxas. A Parallel Algorithm for Computing Minimum Spanning Trees. In *Proceedings of the* 4^{th} *Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 363-372, 1992.

[KKT95] D. R. Karger, P. N. Klein and R. E. Tarjan. A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees. In *Journal of the ACM,* 42(2), pages 321-328, 1995.

[KNP92] D. R. Karger, N. Nisan and M. Parnas. Fast Connected Components Algorithms for the EREW PRAM. In *Proceedings of the* 4^{th} *Annual ACM-SIAM Symposium on Parallel Algorithms and Architectures,* pages 562-572, 1992.

[KR90] R. M. Karp and V. Ramachandran. Parallel Algorithms for Shared-Memory Machines. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 869-932, MIT Press, Cambridge, MA, 1990.

[Lin] N. Linial. *Personal Communication.*

[Nis92] N. Nisan. Pseudorandom Generators for Space-Bounded Computation. In *Combinatorica,* 12(4), pages 449-461, 1992. A preliminary version appeared in STOC 1990.

[Nis93] N. Nisan. On Read-Once vs. Multiple Access to Randomness in Logspace. In *Theoretical Computer Science,* 107, pages 135-144, 1993. A preliminary version appeared in *Proceedings of the* 5^{th} *IEEE Structure in Complexity Theory Conference*, 1990.

[NSW92] N. Nisan, E. Szemeredi and A. Wigderson. Undirected Connectivity in $O(\log^{1.5} n)$ Space. In *Proceedings of the* 33^{rd} *Annual Symposium on Foundations of Computer Science*, pages 24-29, IEEE Computer Society Press, October 1992.

[SV82] Y. Shiloach and U. Vishkin. An $O(\log n)$ Parallel Connectivity Algorithm. In *Journal of Algorithms,* 3: pages 57-67, 1982.