1997

Connected Components in O(log3/2 n) Parallel Time for the CREW PRAM

Donald B. Johnson
Dartmouth College

P. Takis Metaxas
Wellesley College, pmetaxas@wellesley.edu

Connected Components in $O(\log^{3/2} n)$ Parallel Time for the CREW PRAM

Donald B. Johnson[1] Panagiotis Metaxas[2]
Dartmouth College[3]

[1]email address: djohnson@cardigan.dartmouth.edu
[2]Current address: Department of Computer Science, Wellesley College, Wellesley, MA 02181, email address: pmetaxas@lucy.wellesley.edu
[3]Department of Mathematics and Computer Science, 6188 Bradley Hall, Hanover, NH 03755

Running head: Connected Components in $o(\log^2 n)$ Time

Contact author: Panagiotis Metaxas
Department of Computer Science
Wellesley College
Wellesley, MA 02181

email: pmetaxas@lucy.wellesley.edu

Abstract

Finding the connected components of an undirected graph $G = (V, E)$ on $n = |V|$ vertices and $m = |E|$ edges is a fundamental computational problem. The best known parallel algorithm for the CREW PRAM model runs in $O(\log^2 n)$ time using $n^2/\log^2 n$ processors [6, 15]. For the CRCW PRAM model, in which concurrent writing is permitted, the best known algorithm runs in $O(\log n)$ time using slightly more than $(n+m)/\log n$ processors [26, 9, 5]. Simulating this algorithm on the weaker CREW model increases its running time to $O(\log^2 n)$ [10, 19, 29]. We present here a simple algorithm that runs in $O(\log^{3/2} n)$ time using $n+m$ CREW processors. Finding an $o(\log^2 n)$ parallel connectivity algorithm for this model was an open problem for many years.

1 Introduction

Let $G = (V, E)$ be an undirected graph on $n = |V|$ vertices and $m = |E|$ edges. A *path* p of length k is a sequence of edges $(e_1, \cdots, e_i, \cdots, e_k)$ such that $e_i \in E$ for $i = 1, \cdots, k$, and e_i and e_{i+1} have a common endpoint for $i = 1, \ldots, k-1$. At most one endpoint is common with any other. We say that two vertices belong to the same *connected component* if and only if there is a path containing them.

The problem of finding the connected components of a graph $G = (V, E)$ is to divide the vertex set V into equivalence classes, each one containing vertices that belong to the same connected component. These classes are sometimes represented by a set of pointers $p(x)$, where $x \in V$, such that vertices v and w are in the same class if and only if $p(v) = p(w)$ (Figure 1).

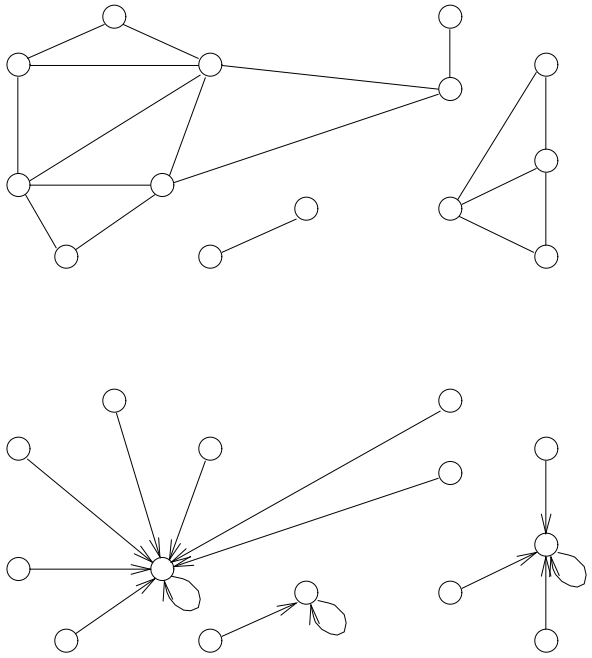

Figure 1: A graph G with three connected components (top) and the pointers p at the end of the computation (bottom).

It is well known that the connected components problem for both directed and undirected graphs has a linear-time sequential solution using depth-first search [27], but implementation of this method in parallel seems very difficult [25]. No polylogarithmic-time deterministic parallel algorithm is known for depth-first search,

and the best randomized algorithm that can be used to do depth first search [1] runs in $O(\log^7 n)$ using almost n^4 processors.

Prior to our work, the best known deterministic parallel algorithm for connectivity ran in $O(\log^2 n)$ time on the CREW PRAM using $n^2/\log^2 n$ processors. (Throughout the paper, when we wish to denote $\log_2 n$ explicitly, we use $\lg n$.) This result, due to [6], improved the processor complexity of [15]. In the CREW model of parallel computation, concurrent writing to any memory location by more than one processor is not allowed. For the CRCW PRAM model, in which concurrent writing is permitted, the best known algorithm runs in $O(\log n)$ time using $(n+m)\alpha(n,m)/\log n$ processors [26, 9, 5]. There is also a randomized algorithm [11] with the same time complexity. Simulating an algorithm designed for this model on the weaker CREW model increases its running time to $O(\log^2 n)$ [10, 19, 29].

We present an efficient and simple algorithm that runs in $O(\log^{3/2} n)$ time using $n+m$ CREW processors. This is a somewhat surprising result because, as Karp and Ramachandran have noted [19], "graph problems seem to need at least $\log^2 n$ time on a CREW and EREW PRAM and $\log n$ on a CRCW PRAM". This observation stems from the fact that many parallel graph algorithms employ a connectivity procedure as a subroutine. Indeed, our result improves the running times of algorithms for several graph problems, including ear decomposition [22], biconnectivity [28], strong orientation [30] and Euler tours [3].

Our algorithm is the first parallel connectivity algorithm with running time $o(\log^2 n)$ for this model. In the course of devising the algorithm, algorithmic techniques were invented which may have other applications, since they address problems arising often in parallel graph algorithms.

While there are three major innovations on which our new time bound depends, the most subtle of these is the scheduling of the rate of growth of connected components. We have found a suitable rate for components to grow so as to control the overhead attendant on redundant edge removal and, since components may actually grow faster than this ideal rate, we have devised an algorithm which, when necessary, recognizes episodically when a component is growing too fast and therefore can be ignored.

A PRAM (Parallel Random Access Machine) employs p processors, each one able to perform the usual computation of a sequential machine using some finite amount of local memory. The processors communicate through a shared global memory to which all are connected. Depending on the way the access of the processors to the global memory is handled, PRAMs are classified as EREW, CREW and CRCW. (In the model names, E stands for "exclusive" and C for "concurrent".) In the EREW PRAM, no two processors are allowed to read concurrently from or write concurrently to the same cell in the shared memory. In the CREW PRAM, concurrent reads are allowed, and in the CRCW PRAM, both concurrent reads and writes are allowed.

(See [19] for more details on the PRAM model.)

The paper is organized as follows: Section 2 gives a general overview of the major difficulties that arise in the process of discovering a fast parallel connectivity algorithm for a model that does not allow write conflicts. Then, Sections 3, 4 and 5 address these difficulties independently. Section 6 describes how the solutions proposed are incorporated into the algorithm. It also gives an overview of the algorithm. Section 7 presents the algorithm in detail and Section 8 contains the correctness and complexity proofs. Finally, Section 9 presents conclusions.

We present notation as the need for it arises. In the case of edges, we use the notation $e = (i, j)$, for vertices i and j in E, to represent both undirected edges and directed arcs and pointers, relying on context to make clear to the reader what is meant.

2 The General Idea and Implementation Difficulties

We introduce first the general idea behind the algorithm. Then, we discuss the problems encountered in implementing this general idea and the solutions we propose.

Let $G = (V, E)$ be the input graph with $n = |V|$ vertices and $m = |E|$ edges. We assume that there is one processor, *Proc(i)*, assigned to each vertex $i \in V$ and one processor, *Proc(i,j)*, assigned to each undirected edge $(i, j) \in E$ which, for implementation reasons, is represented as two directed edges. At later stages of the algorithm, this same processor is assigned to the directed arcs and pointers the algorithm constructs between these two vertices. Thus m processors suffice for all the edges the algorithm uses.

The algorithm deals with *components*, which are sets of vertices already known to belong to the same connected component of G. Each component is equipped with an *edge-list*, a linked list of the edges that connect it to other components. Initially there are n components — each vertex is a separate component. The algorithm proceeds as follows (See Figure 2):

repeat until there are no edges left:

1. Each component picks, if possible, the first edge from its edge-list leading to a neighboring component (called its *mate*), and *hooks* by pointing to it. The hooking process creates clusters of components called pseudotrees (directed graphs with exactly one directed cycle). If a component has an empty edge list, it hooks to itself.

2. Each pseudotree is identified as a new component, the vertices of which are the components referred to in step 1. above. One of these vertices is designated to

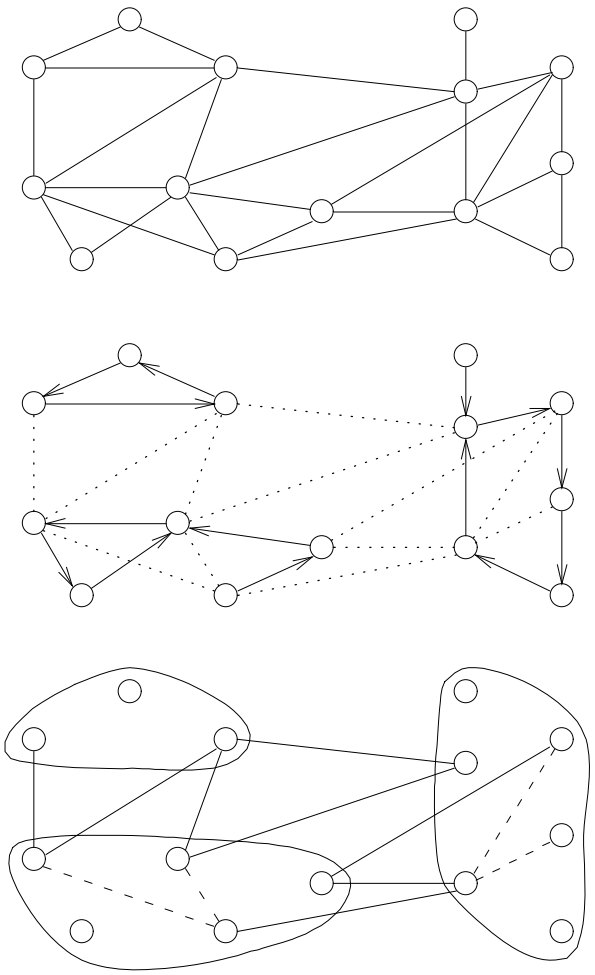

Figure 2: (Top) The input graph G. (Middle) Vertices have picked their mates. Dotted are those edges that were not picked by any vertex. An arc points from a vertex to its mate. Three pseudotrees are shown in this figure. Note that each pseudotree contains a cycle. (Bottom) The new components have been identified. Dashed edges are internal edges that will not help the component grow. Note that there are multiple edges between components.

be its *representative*. Each representative receives into its edge-list all the edges contained in the edge-lists of the vertices in its pseudotree.

3. Edges internal to components and multiple between components are removed.

There are three problems we have to deal with in order for the algorithm to run fast without concurrent writing.

Eliminating cycles. The parallel hooking in the first step of the algorithm above creates pseudotrees which need to be contracted. The usual pointer-doubling technique does not work on cycles when exclusive writing is required. Previous algorithms deal with this problem in different ways. The algorithm of [15] spends $O(\log n)$ time to create trivial pseudotrees, while the algorithm of [26] uses the power of concurrent writing to avoid pseudotree creation.

We solve this problem with a cycle-reducing shortcutting technique we introduce in Section 3. This technique, when applied to a pseudotree, contracts it to a rooted tree in time logarithmic in the length of its cycle, and when applied to a rooted tree, contracts it to a rooted star in time logarithmic in the length of its longest path.

Constructing the edge-list of a new component. Computing the set of the edges of all the components in a pseudotree without concurrent writing may be time consuming: There is possibly a large number of components that hook together in the first step and therefore a large number of components that are ready to give their edge-lists simultaneously to the new component's edge-list. We note that [26] uses the power of concurrent writing to overcome this problem, while [15] uses an adjacency matrix and $O(n^2)$ processors to solve it in $O(\log n)$ time.

The edge-plugging scheme we introduce in Section 4 produces a new edge list in constant time without concurrent writing, whether or not the component is yet contracted to a rooted star.

Finding a mate component. Having a component pick a mate may also be time consuming: There may be a large number of edges internal to the component, and this number grows every time components hook. None of these internal edges can be used to find a mate. Therefore, a component may fail many times to find a mate if it picks internal edges. On the other hand, removing all the internal edges before picking an edge may also take a long time.

This problem is solved by the growth-control schedule we introduce in Section 5. Components are scheduled to grow in size in a uniform way that controls their

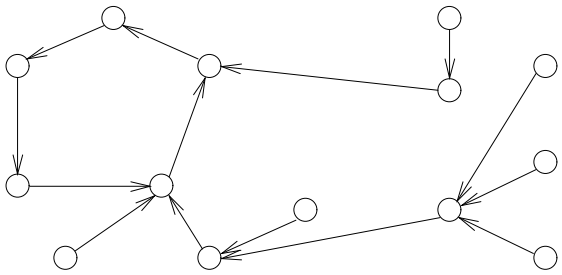

Figure 3: A pseudotree.

minimum sizes as long as continued growth is possible. At the same time internal edges are identified and removed periodically to make hooking more efficient.

We should note that, even though both the cycle-reducing technique and the edge-plugging scheme provide valuable tools for the algorithm, it is the growth-control schedule that achieves the $o(\log^2 n)$ running time. We have found a growth rate for components which is fast enough for our time bound, and at which the overhead attendant on redundant edge removal can be controlled, and for which components that grow faster can be ignored. Our algorithm recognizes such faster growing components at points where it is necessary to do so, and it guarantees that components grow at least at the minimum rate.

The techniques we present may have application in other parallel graph algorithms, since the problems they address arise often in the design of parallel algorithms. In the following sections we present these techniques independently of the main result. Lastly we show how they combine to create a fast connectivity algorithm.

3 Pseudotree Contraction Rules

A *pseudotree* $P = (C, D)$ is a connected directed subgraph with $|C| = n$ vertices and $|D| = n$ arcs for some n, for which each vertex has outdegree one (Figure 3). An immediate consequence of the outdegree constraint is that every pseudotree has exactly one simple directed cycle (which may be a loop). We call the number of arcs in the cycle of a pseudotree P its *circumference*, $circ(P)$.

A *rooted tree* is a pseudotree whose cycle is a loop on some vertex r called the *root*. So, it has circumference one. A *rooted star*, (C, D), is a rooted tree with root r, where, for all vertices $x \in C$, $(x, r) \in D$.

A *pseudoforest* $F = (V, A)$ is a collection of pseudotrees with vertices V, and arcs A on these vertices.

We define the pseudotree contraction problem as follows:

Problem 1 Pseudotree Contraction. *Given a pseudotree $P = (C, D)$, create a rooted star $R = (C, D')$ having as root some vertex $r \in C$ such that for each $v \in C$, $(v, r) \in D'$.*

We show how to solve the pseudotree contraction problem in $O(\log |C|)$ parallel time using $|C|$ CREW PRAM processors.

Pseudoforests are especially interesting in parallel computation. Many parallel graph algorithms in addition to ours create pseudotrees using an operation called *hooking*, in which each vertex simultaneously chooses a neighbor to point to. Connectivity, minimum spanning tree, maximal independent set and tree-coloring, [23, 26, 15, 5, 12, 21, 17] are among the problems with algorithms that deal with creation and manipulation of pseudotrees.

However, the presence of the cycle complicates the parallel contraction of the pseudotrees. The reason that the well known pointer-doubling technique [33] does not work on a cycle is that it does not terminate (Figure 4). Even if one modifies pointer doubling to recognize a cycle by keeping track of when a pointer again reaches a vertex it pointed to earlier, pointer-doubling performs poorly as it may run in time linear in the circumference of the cycle.

We introduce here a set of pointer-jumping rules called *cycle-reducing (CR) shortcutting rules.* These rules are used to reduce a pseudotree to a rooted star (see Figure 5), without concurrent writing by the processors involved, in time $\lceil \log_{3/2} h \rceil$ where h is the longest simple directed path of the pseudotree.

Let $G = (V, E)$ be a directed graph. We assume that each vertex $v \in V$ has a unique identifier, for example, the number of the processor responsible for the vertex. A comparison between two vertices corresponds to a comparison between their two identifiers. Let $F = (V, p)$ with $F \subseteq G$ be a given pseudoforest defined on the vertices of G. We would like to contract each of F's pseudotrees to a rooted star. We do so using the CR shortcutting rules (Figure 6). These rules assign the vertex r having the smallest number among all the vertices in the cycle of the pseudotree to be the root of the future rooted tree. The idea behind the CR rules is that they do not let any of the vertices of the pseudotree shortcut over any vertex that could be the future root.

To use the CR rules, we designate the tail vertex of certain pointers (and thereby the pointer) as *bold.* Bold pointers belong only to possible roots of a pseudotree. Each vertex v of the pseudotree first executes the *rule-enabling* statement:

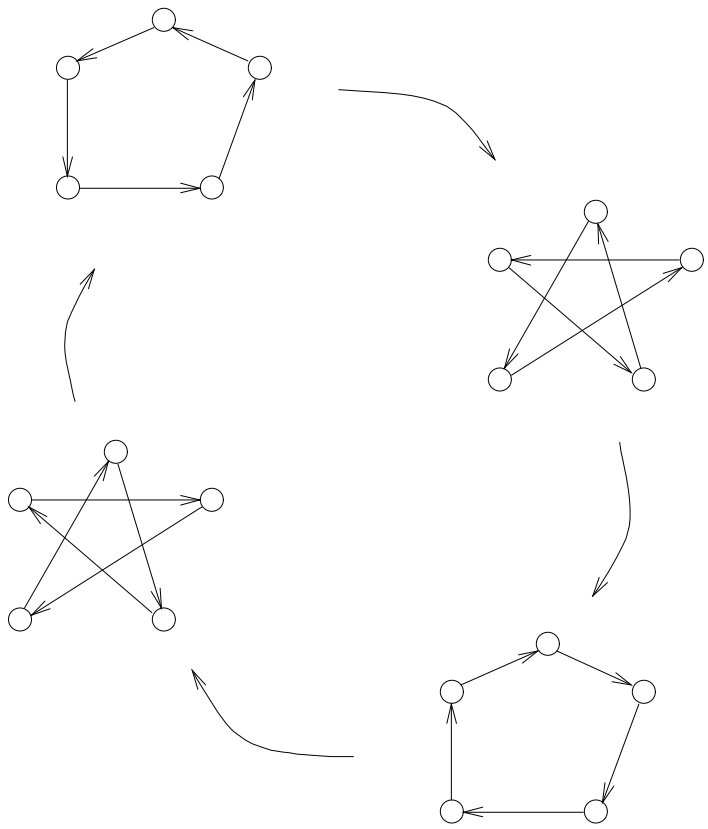

Figure 4: The usual pointer jumping technique cannot deal with cycles.

for each vertex $v \in C$ **in parallel do**
$bold(v) \leftarrow id(v) < id(p(v))$

To contract a pseudotree, its vertices repeatedly execute the *CR procedure* given below. The rules of this procedure are also graphically described in Figure 6. It is convenient in the discussion to refer to an ordinary pointer as *light*, in constrast to *bold*.

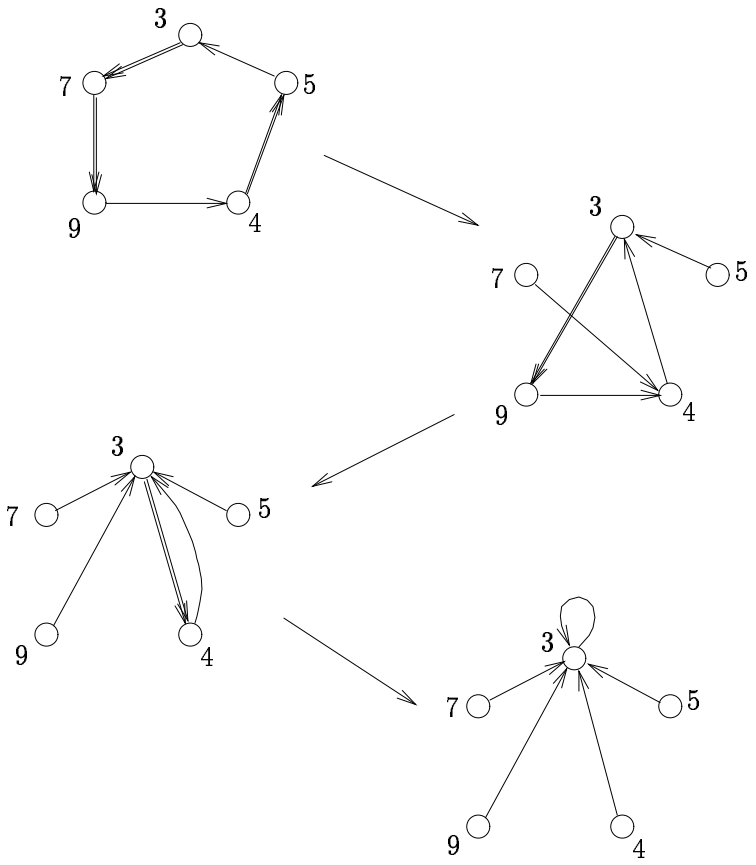

Figure 5: Using the CR shortcutting rules, the cycle can be reduced to a rooted tree in logarithmic number of steps.

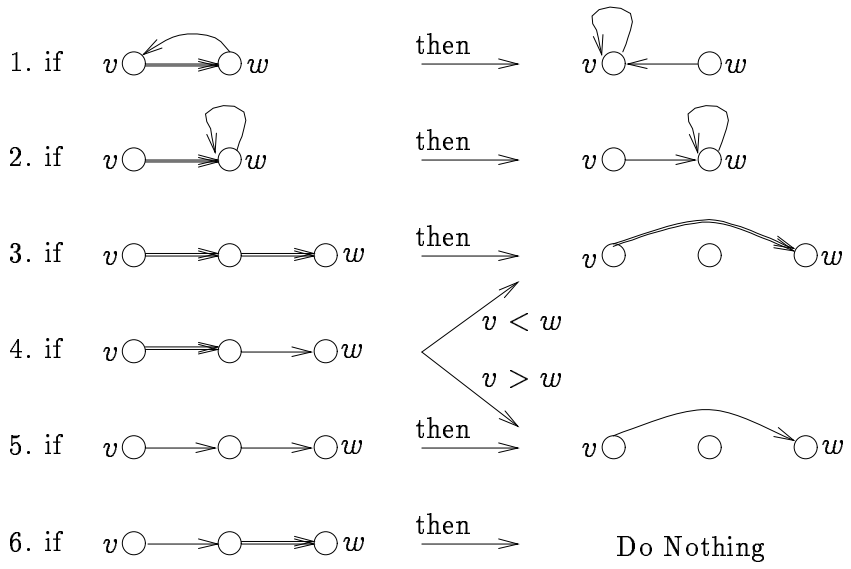

Figure 6: Cycle-Reducing rules. (1) and (2) Terminating rules, (3) Bold-bold rule, (4) Bold-light rule, (5) Light-light rule, (6) Light-Bold rule. Comparison between vertices means comparison between their corresponding numbers.

if $bold(v)$ **and** $p(p(v)) = v$ **then** $bold(v) \leftarrow false$
$p(v) \leftarrow v$
else if $bold(v)$ **and** $p(p(v)) = p(v)$ **then** $bold(v) \leftarrow false$
else if $bold(v)$ **and** $bold(p(v))$ **then** $p(v) \leftarrow p(p(v))$
else if $bold(v)$ **and** $not(bold(p(v)))$
then if $id(v) > id(p(v))$ **then** $bold(v) \leftarrow false$
$p(v) \leftarrow p(p(v))$
endif
else if $not(bold(v))$ **and** $not(bold(p(v)))$ **then** $p(v) \leftarrow p(p(v))$
{ **else if** $not(bold(v))$ **and** $bold(p(v))$ **then** do nothing **endif** }
endif
endif
endif
endif

We refer to the first two rules as *terminating rules*, and to the remaining four

rules as *bold-bold*, *bold-light*, *light-light* and *light-bold*, respectively. The names are given according to the v and $p(v)$ pointers. Shortcutting occurs in all but the light-bold rule, in which case the vertex attempting the jump might shortcut over the smallest numbered vertex of a cycle.

We first show that the CR rules do, in fact, contract a pseudotree to a rooted tree. Then we prove that this contraction takes time logarithmic in the number of vertices in the pseudotree.

Lemma 1 *Let P be a pseudotree with cycle c. When the CR rules apply on P for a long enough period of time, they contract it to a rooted tree. The root of this tree is the smallest numbered vertex, r, that appears on c.*

Proof. We say that vertex v has *reached* vertex u if $p(v) = u$. According to the rule-enabling statement and the CR rules, vertex r has a bold pointer for as long as there is a nontrivial cycle in the pseudotree. At the same time, any vertex of the cycle c that reaches r must do so with a light pointer, called a *last* pointer. (Note that there is only one last pointer on each pseudotree, because there is only one cycle.) So, no vertex in c can shortcut over r because the light-bold rule applies. Moreover, r continually shortcuts over the other pointers in the cycle since the bold-bold and bold-light rules permit it. Eventually, the first terminating rule applies and r reaches itself, effectively becoming the root of a rooted tree. □

Let P be an n-vertex pseudotree with vertex set C and let r be its root, if P is a rooted tree, or its *future root* (the smallest numbered vertex on P's cycle). We define the *distance*, d_v, of a vertex $v \in C$ to be the number of pointers (pseudotree arcs) on the shortest directed path from v to r that uses a last pointer. The inclusion of the last pointer condition in the definition of d_v is needed to account for vertices not on C that point to r with a bold pointer. Since these vertices jump over r during the cycle contraction, we need to make their distances greater than d_r. Also, we define d_v^k of vertex v to denote d_v after k applications of the CR rules on P. We can write d_v as d_v^0. We show that each application of the CR rules on P decreases the distances d_v of the vertices $v \in C$ by roughly a factor of two-thirds.

Lemma 2

$$d_v^k \leq \lceil \frac{2d_v^{k-1}}{3} \rceil$$

Proof: The proof is by induction on the distance d_v^k. The base case holds trivially, since for all v and for all k such that $d_v^{k-1} \leq 2$ the lemma is true.

For a given vertex v we assume that for all the vertices having distance smaller than d_v^k and for all the $k-1$ previous applications of the CR rules, the hypothesis holds.

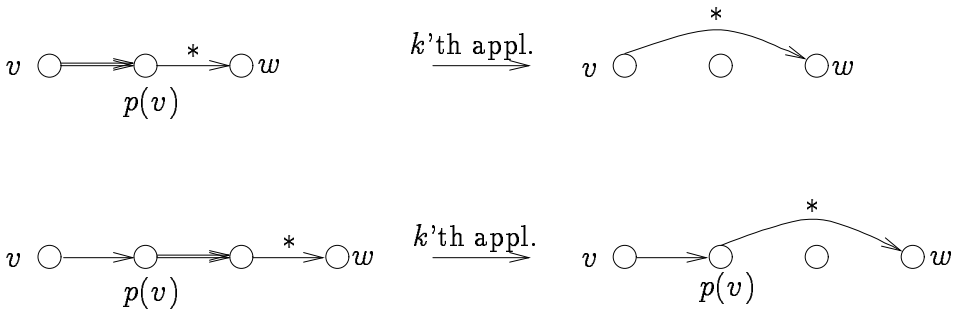

Figure 7: (Top) Case 1 of the Lemma. The light-light case is similar. (Bottom) Case 2 of the Lemma. An asterisk (*) over a pointer means that this pointer could be either bold or light.

That is, we assume that for all vertices u and for all k satisfying $1 \leq d_u^{k-1} < d_v^{k-1}$ the following holds:

$$d_u^k \leq \left\lceil \frac{2d_u^{k-1}}{3} \right\rceil$$

With this hypothesis we now prove:

$$d_v^k \leq \left\lceil \frac{2d_v^{k-1}}{3} \right\rceil$$

We consider two cases, one accounting for application of the bold-bold, bold-light and light-light rules on v (that is, when v's pointer is shortcutting), and one for the light-bold rule (when v's pointer is stuck, but its parent's pointer is shortcutting).

Case 1: Let us assume that v's pointer is bold or $p(v)$'s pointer is light. Since in all cases v's pointer shortcuts (Figure 7, top), the analysis is identical.

Let $w = p(p(v))$. We have that $d_v^{k-1} - 2 = d_w^{k-1}$, and we assume that $d_w^k \leq \lceil 2d_w^{k-1}/3 \rceil$. We want to prove that $d_v^k \leq \lceil 2d_v^{k-1}/3 \rceil$.

At the k'th application of the CR rules v reaches w, so we have:

$$\begin{aligned} d_v^k &\leq 1 + d_w^k \\ &\leq 1 + \left\lceil \frac{2d_w^{k-1}}{3} \right\rceil \\ &= \left\lceil 1 + \frac{2(d_v^{k-1} - 2)}{3} \right\rceil \\ &= \left\lceil \frac{2d_v^{k-1}}{3} - \frac{1}{3} \right\rceil \end{aligned}$$

$$\leq \lceil \frac{2d_v^{k-1}}{3} \rceil .$$

CASE 2: This is the case where v's pointer is light (Figure 7, bottom) and $p(v)$'s pointer is bold. The pointer of $p(p(v))$ is either light or bold, but in any case $p(v)$ shortcuts. So, let $w = p(p(p(v)))$. We have that $d_v^{k-1} - 3 = d_w^{k-1}$, and we assume that: $d_w^k \leq \lceil 2d_w^{k-1}/3 \rceil$. We want to prove that: $d_v^k \leq \lceil 2d_v^{k-1}/3 \rceil$.

At the k'th application of the CR rules $p(v)$ reaches w, therefore we have:

$$\begin{aligned} d_v^k &\leq 2 + d_w^k \\ &\leq 2 + \lceil \frac{2d_w^{k-1}}{3} \rceil \\ &= \lceil 2 + \frac{2(d_v^{k-1} - 3)}{3} \rceil \\ &= \lceil 2 + \frac{2d_v^{k-1}}{3} - \frac{6}{3} \rceil \\ &= \lceil \frac{2d_v^{k-1}}{3} \rceil . \end{aligned}$$

□

The following lemmas are useful in proving the connectivity algorithm correct. Let $\alpha = 1.71 > 1/(\lg \frac{3}{2})$. (Recall that we use $\lg n$ to denote $\log_2 n$.)

Lemma 3 *When the cycle-reducing rules are applied $\lceil \alpha t \rceil$ times to a rooted tree, any vertex within distance 2^t from the root reaches the root of the tree.*

Proof. From Lemma 2 and the two terminating rules we derive that when the cycle-reducing rules are applied t times to a rooted tree, any vertex within distance $(\frac{3}{2})^t - 2$ from the root reaches the root of the tree. Vertices with $d^{i-1} = 2, 1 \leq i \leq t$ (i.e., within distance 2 from the root), do so in the next step of the CR rules. The Lemma follows by observing that $\lceil \alpha t \rceil > t/(\lg \frac{3}{2})$, therefore $(\frac{3}{2})^{\lceil \alpha t \rceil} > 2^t$. □

Lemma 4 *When the cycle-reducing rules are applied $\lceil \alpha t \rceil$ times to a pseudotree P whose cycle has circumference no larger than 2^t, they contract it to a rooted tree with root the vertex r having the smallest number among the vertices in the cycle. Moreover, any vertex within distance 2^t from r in the original pseudotree has reached r.*

Proof. We observe that $d_r = circ(P)$. If $circ(P) < 2^t$, then r's pointer reaches itself in $\lceil \alpha t \rceil$ steps (Lemma 3) and r becomes the root of a rooted tree. On the other hand, any vertex at distance 2^t from r reaches r after $\lceil \alpha t \rceil$ applications of the CR rules. □

So we have proved the following theorem for the CR rules:

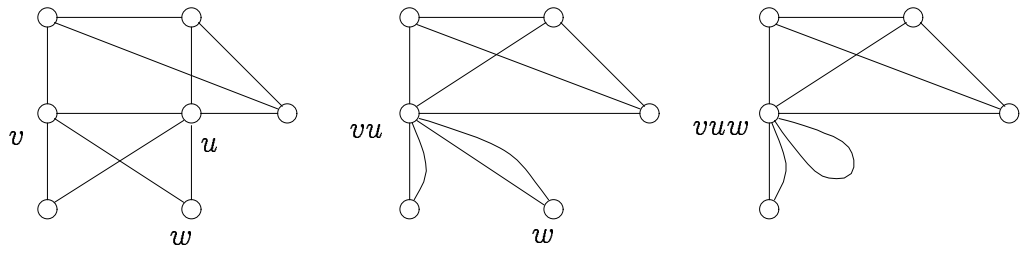

Figure 8: Contraction of the vertices v, u, w.

Theorem 1 *A pseudotree P with $h = \max_{v \in P}\{d_v\}$ is contracted to a rooted star R after $\lceil \lg_{3/2} h \rceil$ applications of the CR rules. The root of R is the smallest numbered vertex on P's cycle.*

□

4 Edge-Plugging Scheme

A common representation of a graph $G = (V, E)$ is the *adjacency list*: The graph is represented as an array of $|V|$ vertices, and each vertex v in the array is equipped with a pointer to its *edge-list* $L(v)$, a linked list of all the edges that connect v to other vertices of the graph. The pointer $next(e)$ points to the edge appearing after edge e in the edge-list where e is contained.

Contraction is one of the basic operations defined on graphs [32]. Under this operation, two vertices v and w connected with an edge (v, w) are identified as a new vertex vw (Figure 8). We can generalize slightly this operation to be performed on a subset of tree-connected vertices of the graph, rather than just on the vertices of one edge. Again, this subset is identified with a new vertex (which some authors call a *supervertex*). In practice, one of the vertices in the subset, called the *representative*, plays the role of the new vertex. To keep the representation of the graph consistent, one needs to put all the edges formerly belonging to the edge-lists of each vertex in the set into the edge-list of the newly formed vertex.

As discussed in the previous section, pseudotrees are vertex subsets that appear naturally in parallel computation. Without loss of generality, we can think of the representative r as being the vertex assigned as the root by the CR rules. So, the following problem naturally arises:

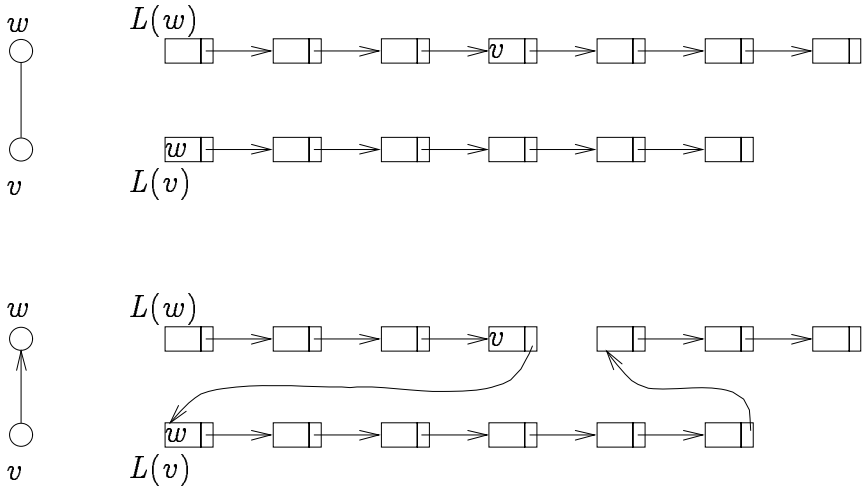

Figure 9: Edge lists of nodes v and w before (top) and after (bottom) the edge plugging step.

Problem 2 Edge-list Augmentation. *Given a pseudotree $P = (C, D)$ of a graph $G = (V, E)$, i.e., $C \subseteq V$ and $(u, w) \in D \Rightarrow (u, w) \in E$, augment the edge-list of one of C's vertices, say the representative r, with the edges that are included in the edge-lists of all the vertices $v \in C$.*

We show how to solve this problem in constant time without memory access conflicts when the representative of the pseudotree is known.

Let the *first* and *last* functions defined on $L(v)$ give the first and last edges, respectively, appearing in $L(v)$. For implementation reasons it is convenient to assume that there is a fake edge at the end of each edge-list. All these functions are easily implemented with pointers in the straightforward way.

We represent each undirected edge (v, w) by two *twin* copies (v, w) and (w, v). The former is included in $L(v)$ and the latter in $L(w)$. The two copies are interconnected via a function $twin(e)$ which gives the address of the twin copy of edge e. We can assume that both (v, w) and (w, v) are being simulated by the same processor. Therefore, calculating the *twin* function in constant time is straightforward. If this is not the case, then the *twin* function is calculated in two steps using array $M[1..n, 1..n]$ and $2m$ processors: First, $Proc(v, w)$ writes in $M[v, w]$ the address of edge (v, w). Next, the processor reads in $twin((v, w))$ the address of edge (w, v) from $M[w, v]$.

The edge-list augmentation problem can be solved with the *edge-plugging* scheme we present here. Let $v \in C$, $v \neq r$, be a vertex in the pseudotree and $(v, w), (v, w) \in$

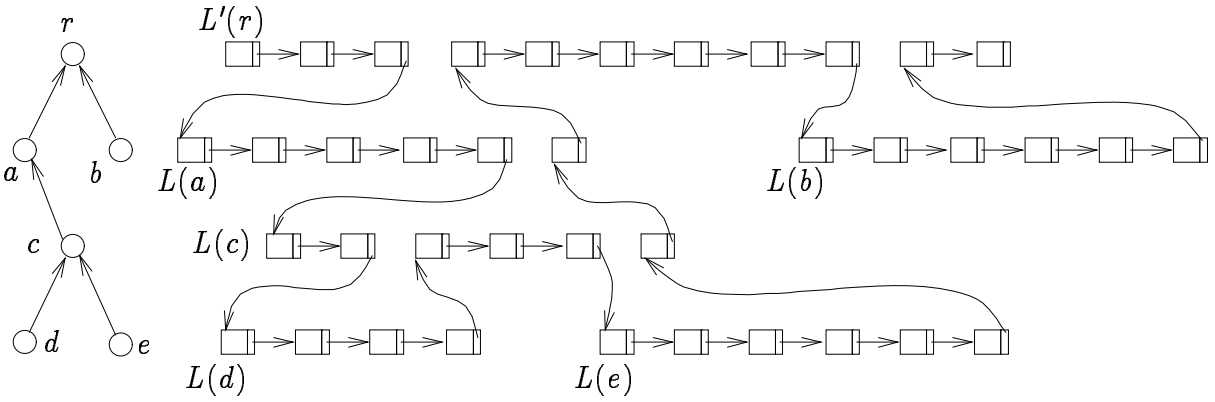

Figure 10: The effect of the plugging step execution by all vertices of a pseudotree but the representative r. On the left is $P = (C, D)$. On the right is $L'(r)$ after the execution of the plugging step.

D, be its outgoing arc in the pseudotree P. According to this scheme, v *plug*s its edge-list $L(v)$ into w's edge-list by redirecting some pointers. The exact place that $L(v)$ is plugged is after the twin edge (w, v) contained in $L(w)$ (Figure 9). This ensures exclusive writing. The edge-plugging is done by having each vertex $v \in C - \{r\}$ execute the *plugging step*:

for each vertex $v \in C - \{r\}$ **in parallel do**
    let $(v, w) \in D$
    let $(w, v) = twin((v, w))$
    $next(last(L(v))) \leftarrow next((w, v))$
    $next((w, v)) \leftarrow first(L(v))$
endfor

We can see that the effect of having all $v \in C - \{r\}$ perform the plugging step simultaneously is to place all the edges in their edge-lists into r's updated edge-list $L'(r)$ (Figure 10). In particular we can prove the following lemma.

Lemma 5 *Let $P = (C, D)$ be a rooted tree with root r. If the edge-lists of all vertices in C are linked lists, then if all vertices in C except r simultaneously execute the edge-plugging step (described above), r's edge-list becomes a linked list containing all edges in D. No write conflicts occurs during the edge-plugging step.*

Proof. First, observe that if all vertices in C simultaneously execute the edge-plugging step, the only processor that accesses pointer $next(twin((v,w)))$ is the processor assigned to vertex v. Therefore there is no writing conflict, and if follows that were the pluggings to be done in sequential order, the order would not matter.

Next, we show that the result of this step is that r's edge list becomes a linked list containing all edges in D. For the sake of the proof, we can assume as noted above that the edge-plugging operations occur in this particular order: first, vertices at distance 1 from the root plug their lists, then vertices at distance 2, and so on, until all vertices (but r) plug. It follows by induction, that an edge in the list of a vertex at distance k ends up in the edge list of r. □

So, we have shown that the edge-list augmentation problem can be solved in constant time once the representative of the pseudotree has been determined.

The connectivity algorithm we present in Section 6 may execute the plugging step before the contraction of a pseudotree to a rooted tree. Thus one may ask what the effect is of executing the edge-plugging step *before* the representative is known and all vertices participate. The following lemma shows what occurs if *all* vertices in the pseudotree execute the edge-plugging step.

Lemma 6 *Let $P = (C, D)$ be a rooted tree with root r. If the edge-lists of all vertices in C are linked lists, then if all vertices in C simoultaneously execute the edge-plugging step, then all the pseudotree's vertices are placed into two linked rings. No write conflicts occurs during the edge-plugging step.*

Proof. As in the proof of the previous lemma, we assume that the edge-pluggings occur in the same order and the root plugs last. It is easy to see that this last operation splits r's edge list into two linked rings (see Figure 11). □

Note, however, that this is a recoverable situation since, in general, the representative can later reverse the effects of its own edge-plugging, thus joining the two rings into a single linked list. Lemma 7 in Section 8 explains how this can be accomplished.

5 Growth-Control Schedule

First, let us illustrate the need for such a schedule. We argued in Section 2 that having a component pick a mate may be time consuming. We now make this statement more precise.

The cycle-reducing rules and the edge-plugging scheme provide the elements of a connectivity algorithm that works correctly on the CREW PRAM model of parallel

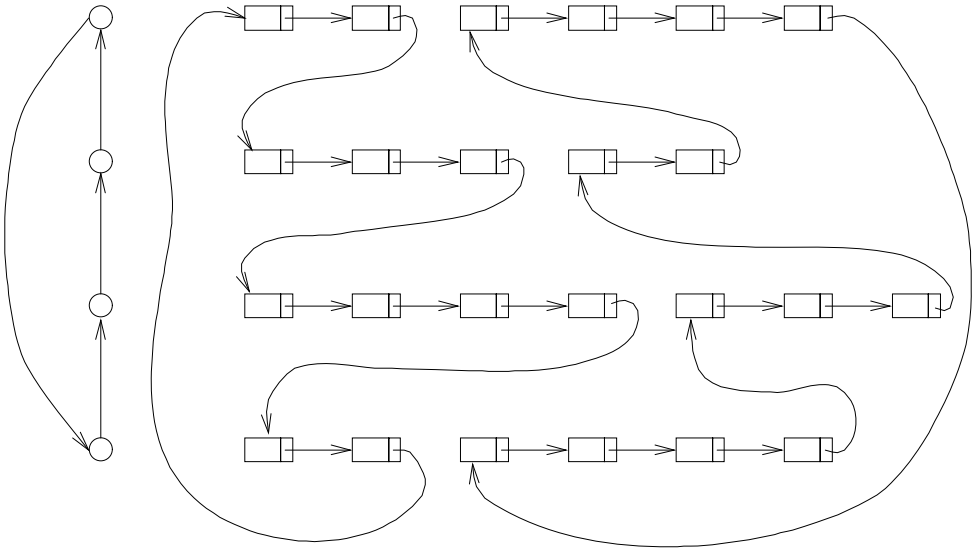

Figure 11: When all the vertices in a cycle execute the plugging step, their edge-lists are connected with two rings of edges. Observe that the *first* pointers of the vertices in the cycle end up in the first ring while the *last* end up in the second. This enables the future root of the cycle to reverse the effect of its own edge-plugging, thus rejoining the two rings into a single edge-list.

computation. Let $T(n)$ be some number that we compute shortly. The algorithm (a first attempt) is as follows:

Algorithm 1.

for $T(n)$ phases **in parallel do**

1. Execute the hooking step by having the representative of each component find a mate, if possible.
2. Try to contract each of the resulting pseudotrees by applying the CR rules for a constant number of times.
3. Identify the roots of any of the resulting rooted trees as new vertices.
4. Perform the plugging step on all the vertices but the representative of each component.
5. Identify internal edges and try to remove them using pointer jumping for a constant number of times.

It is not difficult to see that this algorithm correctly computes the connected components of a graph in $T(n)$ phases, for $T(n)$ sufficiently large. Moreover, we observe that, if we are sure that each component can hook in each phase, then only $\lceil \lg n \rceil$ phases are needed.

However, we cannot be sure that every component will hook in every phase. The reason is that every time two components C_1 and C_2 hook together, the number of internal edges grows. This growth is by a factor of $|C_1| \times |C_2|$ in the worst case, and the time needed to remove them using pointer jumping is $\lg(|C_1| \times |C_2|)$.

As a result of this, some component may attempt to hook many times before it can find a neighboring component. In particular, when components grow at the slowest rate, that is by just pairing up in every hook, the number of internal edges added in the edge-lists in the worst case follows the sequence:

$$1^2, 2^2, 4^2, 8^2, 16^2, \ldots, (\frac{n}{2})^2 = 2^0, 2^2, 2^4, 2^6, 2^8, \ldots, 2^{2\lg(n/2)}$$

So, the time to remove them is:

$$\sum_{i=0}^{\lg(n/2)} \lg 2^{2i} = 2 \cdot \sum_{i=0}^{\lg(n/2)} i = O(\log^2 n)$$

Therefore, the number of phases $T(n)$ in this particular case would be $O(\log^2 n)$. Moreover, as one can see by following the reasoning just described, even if we allow

steps 2 and 5 to be executed more than a constant number of times, say $\log\log n$ or $\log^* n$, $T(n)$ is not reduced asymptotically. So, the crucial observation is the following:

In the beginning, components grow very fast, due to lack of internal edges. Later, when they have grown in size, components having many internal edges may grow much more slowly.

This observation leads to the need for controlling the components' minimum sizes. We introduce the *growth-control schedule* which lowers the running time by a factor of $\sqrt{\lg n}$ without increasing the number of processors involved. We give a brief description of it here and in the next section go into the details (Figure 12).

In order to implement the growth-control schedule, the algorithm is divided into phases. Phase i takes as input a graph G_i, whose components are of size $2^{(i-1)\sqrt{\lg n}}$ (in terms of the number of vertices of the original graph), and which has no internal or redundant external edges. Phase i produces as output a graph G_{i+1}, whose components are of size at least $2^{i\sqrt{\lg n}}$, and which, again, has no internal or redundant external edges. Thus, only $\lceil\sqrt{\lg n}\rceil$ phases are needed.

Each phase consists of a number of subphases, followed by a clean-up process. Each subphase consists of a hooking step, followed by c_1 of pseudotree contraction steps, an edge-plugging step, and c_2 of internal edge removal steps. The constants c_1 and c_2 are chosen so that any component which is not contracted after c_1 steps, or still has internal edges after c_2 edge removal steps, must be of size $2^{i\sqrt{\lg n}}$ already. Any component that is still small after a subphase has no internal edges and, by the observation above, can grow quickly.

After the last subphase of each phase, all components must have grown to the proper size. In the clean-up process, all trees are contracted and internal and redundant external edges are removed. The growth factor, $2^{\sqrt{\lg n}}$, has been chosen so that each phase has $O(\log n)$ running time.

6 Outline of the Algorithm

6.1 Definitions

The algorithm executes a number of *phases*, requiring that each component entering phase i have at least B^i vertices of the original graph, where $B = \lceil 2^{\sqrt{\lg n}}\rceil$. Therefore at most $\lceil\sqrt{\lg n}\,\rceil$ phases are needed. In the beginning of the algorithm all components are of size 1 because they consist of one vertex of the original graph.

We say that some component is *promoted* to phase i, regardless of how many phases have actually been executed, if its size is at least B^i. We should note that the notion of promotion is needed mainly for the analysis; processors may or may not

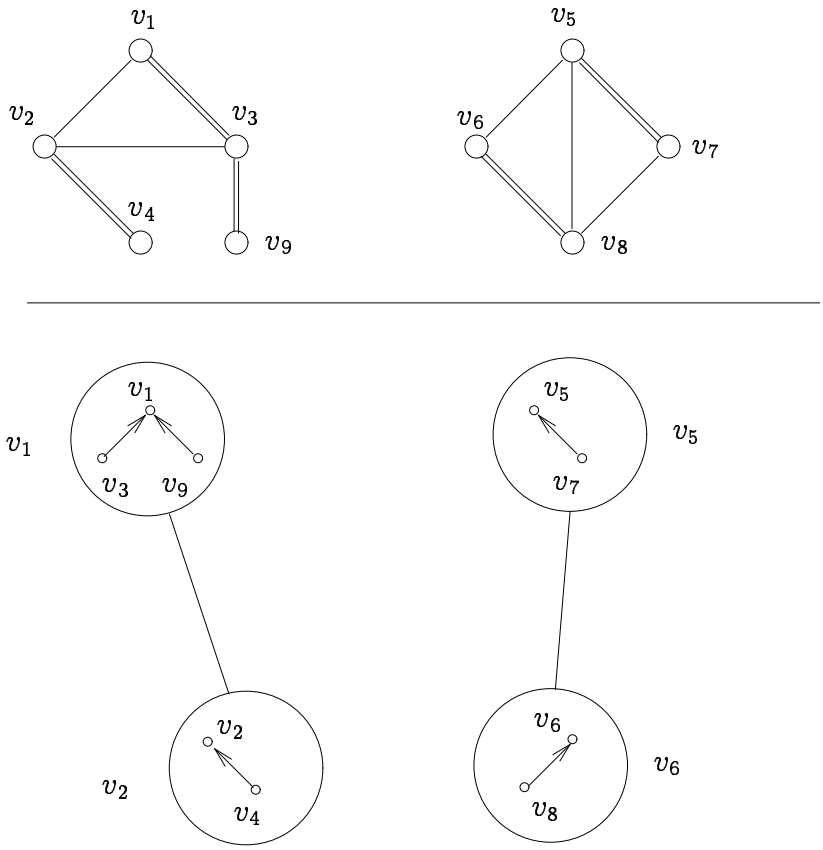

Figure 12: (Top) Graph G_{i-1} in the beginning of phase $i-1$. To simplify the figure, we assume that only the doubly marked edges will be used during phase $i-1$ for hooking. (Bottom) Graph G_i in the beginning of phase i. Vertex v_1 represents the component $\{v_1, v_3, v_9\}$. Edge (v_1, v_2) represents the set of edges $\{(v_1, v_2), (v_2, v_3)\}$.

know whether a component has been promoted during a phase.

Each phase is divided into *subphases*. In each subphase j, components grow in size by hooking to other components. The purpose of a phase can be seen as allowing just enough time for hookings between components, so that all the components have either been promoted to the next phase or they cannot grow any more. If some component cannot grow any more, it is because it is not connected to any other component. In this case it is called *done*, else it is called *active*.

We identify edges that components do not need to keep: *Internal edges* are edges between vertices within the same component. These edges are useless and may be removed. In general, it is difficult to recognize these edges immediately. When an internal edge is recognized as such, it is declared *null*. Since each component contains many vertices, there may be *multiple edges* between two components. For each component pair, only one such edge need be kept in order to hook the two components. Such an edge is called a *useful edge*. The remaining multiple edges are called *redundant edges*. When redundant edges are recognized, they are also declared null and can be removed along with the null internal edges. Of course, it does not matter which of the multiple edges is kept as useful. Any one will do.

6.2 The Subphases of a Phase

As we stated in the previous section, phase i takes as input a graph $G_i = (V_i, E_i)$. Each component $C \in V_i$ contains at least B^i vertices of the original graph G. Note that if a component has fewer than B^i vertices, then it is as big as it can get, and the algorithm will ignore it. The vertices of C are organized in a rooted star, with *representative* the root of the star, denoted by r_C. The other vertices in C were found to belong to C in previous phases, and they do not play any role in phase i or in later phases. So we may assume that in the beginning of a phase each component C is a single vertex r_C, the representative, and that all useful edges are in its edge list. In the remainder of the section, when referring to a fixed phase i, we use the term "component" to refer both to the set of vertices in the component and to the representative of the component. It should be clear from the context which meaning applies.

Each component is equipped with an edge-list. The following invariants are used to prove correctness:

Invariant 1 *In the beginning of a phase i there is at most one edge between any two vertices r_{C_1} and r_{C_2} in G_i. In particular, $(r_{C_1}, r_{C_2}) \in E_i$ if and only if there was an edge $(v, w) \in E$ such that v is in r_{C_1}'s component and w is in r_{C_2}'s component.*

Invariant 2 *Let $G_i = (V_i, E_i)$ be the input graph of phase i and r be the root of a rooted tree $P = (C, D)$ of G_i. Then, each unnullified edge $(v, w) \in E_i$, for which $v \in C$, is in $L(r)$*

The idea of invariant 1 is to keep down the number of internal edges, since redundant edges can become internal, while invariant 2 states that the edges are kept in some rational way — much like the pseudotrees organize the vertices in some rational way.

During each phase, components continually hook to form bigger components. As we have described in previous sections, the hooking is done by having each component pick, if possible, the first edge from its edge-list and point to the indicated neighboring component. This operation is carried out by the representative of the component. If there is no edge left in the component's edge-list, then it is not connected to any other component, and it is done. Each component performs $O(\sqrt{\log n})$ hookings per phase, each one in a different subphase.

The hooking operation creates a pseudoforest, which is then contracted using the cycle-reducing shortcutting technique for $O(\sqrt{\log n})$ steps. The objective is the following: After $O(\sqrt{\log n})$ steps, components with fewer than B vertices have become rooted stars and are ready to hook in subsequent subphases to keep growing. The exact number of CR rule applications that achieve this objective is $\lceil \alpha \sqrt{\lg n} \rceil$, for $\alpha = 1.71$ (as in Section 3).

Components that are rooted trees at the end of a subphase are called *ready*. Those that still do not have a root are called *busy*. If some component is busy at the end of a subphase, the cycle of its pseudotree originally had circumference greater than B, and therefore is promoted to the next phase. This component does not hook in subsequent subphases of this phase. At the end of the phase it is given enough time to become contracted to a star and to prepare for the next phase.

Next, the vertices of the newly formed rooted trees are recognized. Then, all the vertices but the roots of the contracted trees execute the plugging step described in Section 4. Let r be the root of a rooted tree and v be a vertex executing the plugging step. This places the edges of v's edge-list into r's edge-list. However, if v is a vertex of a busy component, this does not work, since there is no root in v's component (Lemma 6). Fortunately, a busy component will be detected in a later subphase and this problem will be fixed.

Edges (x, y) can now recognize their new endpoints $p(x)$ and $p(y)$ and can be renamed accordingly. Those having both of their endpoints pointing at the same root are internal and *nullify* themselves. Then, the edge-list of the root is cleared of null edges by $O(\sqrt{\log n})$ *null-edge removal* steps. This is a simple application of the pointer-doubling technique (see also Figure 13):

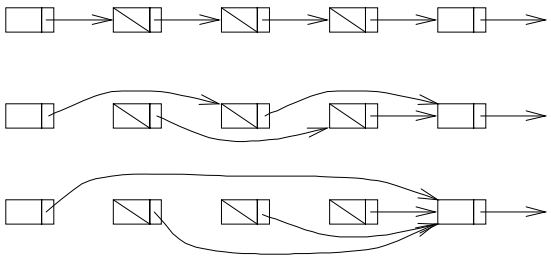

Figure 13: Example of two null-edge removal steps. The three null edges in the middle are being removed from the edge-list.

for all edges e **do in parallel**
  if $null(next(e))$ **then** $next(e) \leftarrow next(next(e))$

The exact number of null-edge removal steps is $2\lceil\sqrt{\lg n}\rceil + 1$. This number is chosen so that any component having fewer than B vertices (and therefore fewer than $\lfloor 2^{2\sqrt{\lg n}}\rfloor$ null edges) can remove all its internal edges, and therefore it can find a non-null edge in the next subphase. This ends a subphase.

In the next subphase, the roots of ready components try to hook again. We say that a vertex (root) v had a *successful hooking*, if its mate w belongs to a different component. Observe that it is possible for a promoted component not to become a rooted star at the end of some subphase j, because it contained a path longer than B. As a consequence, some internal edge e may not be nullified at the end of subphase j, and the root of the component may pick e for hooking in a later subphase. This is called *internal hooking*.

A root having an internal hooking may or may not detect it. For example, some unnullified internal edge (x, y) may be recognized by the root r at the time of the hooking by checking if $r \neq x$. If this is the case, x was at a distance greater than B from r in the tree, and so x did not have the time to reach r and rename (x, y) to (r, y). In this case r does not hook, since its component is known to be promoted.

An (undetected) internal hooking can only create a pseudotree, and the cycle-reduction rules will be called again to deal with it. So, before the application of the CR rules, components have to execute the rule enabling statement, described in Section 3.

The idea behind $O(\sqrt{\log n})$ subphases per phase is that after that many successful hookings a component is promoted in any case. On the other hand, one internal hooking means that a component is already promoted.

Finally, there is another case we should address. Consider a component C having

more than B vertices, but whose height is less than B. The CR shortcutting process will contract it to a rooted star during the subphase of its formation. However, C may have more than B^2 internal edges, and the edge-removal process may not remove them all. This component may be unable to hook if it picks one of the remaining unremoved null edges in the next subphase. However, failure to hook is not harmful because the component is promoted.

Each subphase takes $O(\sqrt{\log n}\,)$ steps and there are at most $O(\sqrt{\log n}\,)$ subphases, summing up to a total of $O(\log n)$ steps per phase. We can prove that after $O(\sqrt{\log n})$ subphases all components have been promoted. They may not have been contracted, though. In the final (clean-up) part of the phase:

- All components are contracted to rooted stars, and representatives are identified.
- Edges are renamed by their new endpoints.
- All internal edges are identified and nullified.
- All multiple edges are identified. One member of each of the sets of identical edges is kept as useful while the rest are nullified as redundant. This is done as follows: First, we sort the edge list of each component in lexicographical order. We note that there are $O(\log n)$ time, n processor, sorting algorithms for both the CREW PRAM model [8, 2, 14] and the CREW PPM model [13]. Then, blocks of redundant edges are identified and nullified. This step takes $O(\log n)$ time using m processors.
- The edge-list of each component is prepared by deleting all the null edges.

Each of these steps take $O(\log n)$ parallel running time. So, the total running time of the algorithm is $O(\log^{3/2} n)$ using $n + m$ CREW PRAM processors.

7 The Algorithm

The important ideas have been presented in the previous sections. We now present the algorithm in detail.

In the beginning, each component contains a single vertex $v \in V$ of the input graph $G = (V, E)$, so we initialize by setting $root(v)$ to true for each $v \in V$. The edge-list of each component is also formed as described above, with *first*, *last*, and *twin* pointers. Each edge list is terminated by a fake edge. Then, the procedure `phase` is executed $\lceil \sqrt{\lg n}\,\rceil$ times. At the end, the vertex set V has been divided into a number of equivalence classes containing the connected components.

Procedure phase

1. INITIALIZATION

   for each vertex v **do in parallel**
   if $root(v)$ **then** $not(promoted(v))$
   $subphase(v) \leftarrow 0$
   $mate(v) \leftarrow v$

2. COMPONENT PROMOTION

   for $i \leftarrow 1$ **to** $\lceil \sqrt{\lg n} \rceil$ **do**
   execute `subphase(i)`.

   COMMENT: At the end of this step, each component is either promoted or done. Each subphase takes time $O(\sqrt{\log n})$.

3. CONTRACT THE PSEUDOFOREST TO ROOTED STARS

   for each vertex v such that $not(root(v))$ **do in parallel**
   $bold(v) \leftarrow id(v) < id(p(v))$

   for $i \leftarrow 1$ **to** $\lceil \alpha \cdot \lg n \rceil$ **do**
   for each vertex v **in parallel do**
   v executes the appropriate CR rule

   COMMENT: At the end of the last subphase components were rooted stars, rooted trees or pseudotrees. This step gives enough time to the last two categories to become rooted stars before they enter the next phase. First, the rule enabling step is executed and then, the CR rules are applied.

4. RENAME EDGES AND IDENTIFY INTERNAL EDGES

   for each edge (v, w) **in parallel do**
   rename (v, w) to $(p(v), p(w))$

   for each edge (v, v) **in parallel do**
   $null((v, v))$

   COMMENT: Internal edges of rooted stars are easily recognized and nullified.

5. IDENTIFY REDUNDANT EDGES

Run list-ranking on the edge-list of each component to find the distance of each edge from the end of its list.

Copy each edge-list in an array using as index the results of the list ranking.

Sort the array [8, 13] and use the results to form a sorted linked list.

for each edge (v, w) **in parallel do**
    if $next((v, w)) = (v, w)$ **then** $null((v, w))$

COMMENT: The sorting places all multiple edges in blocks of consecutive identically named edges. The last edge in a block of consecutive edges having identical (v, w) names is kept as useful. The rest nullify themselves as redundant. Since the useful edge (v, w) found in $L(v)$ may, in general, differ from the useful edge (w, v) found in $L(w)$, some care must be taken for the *twin* function to be recomputed correctly. Step 7 below takes care of this.

6. REMOVE INTERNAL AND REDUNDANT EDGES.

for $j \leftarrow 1$ **to** $2\lceil \lg n \rceil$ **do**
    for each edge e **do in parallel**
        if $null(next(e))$
            then $next(e) \leftarrow next(next(e))$

for each vertex v such that $root(v)$ **in parallel do**
    if $null(first(L(v)))$
        then $first(L(v)) \leftarrow next(first(L(v)))$

COMMENT: This step removes blocks containing up to m consecutive null edges. However, if the first edge on some list was null, no pointer could have jumped over it, and it cannot have been removed. The last step explicitly removes any null edge from $first(L(v))$. The remaining edges satisfy Invariant 1 (Section 6.2).

7. RECOMPUTATION OF THE *twin* FUNCTION.

for all edges (v, w) such that $not(null((v, w)))$ **in parallel do**
    let $(v', w') = next((v, w))$
    let $prev((v', w')) = (v, w)$
    $twin((v, w)) \leftarrow prev(next(twin((w, v))))$

COMMENT: This final step recomputes the *twin* function of the useful edges (v, w) in constant time as follows: First, observe that, after removing redundant edges from an edge-list, all edges named (v, w), useful and redundant, point at the same location. This location is the edge (v', w') that comes lexicographically after (v, w). The useful edge (v, w) passes its address to a field $prev((v', w')$. From there, the useful edge (w, v) reads it, by following pointer $next(twin((w, v)))$.

`Procedure subphase(i)`

1. THE HOOKING STEP

for each vertex v such that $root(v)$ **and** $active(v)$ **and** $not(promoted(v))$
do in parallel
let $(x, y) \leftarrow first(L(v))$
if $(x, y) = nil$ **then** $done(v)$
else if $x \neq v$ **then** $promoted(v)$
else if $null((x, y))$ **then** $promoted(v)$
else $mate(v) \leftarrow y$
$p(v) \leftarrow y$
$not(root(v))$

COMMENT: Roots of still active and possibly unpromoted components try to pick an edge from their edge-list. If there is no edge in $L(v)$, i.e. $first(L(v)) = nil$, its component is not connected to any other component and it is done. If $x \neq v$ then $p(x) \neq v$, and $d_x^0 > B$ (d_x^0 is defined in Section 3). This indicates that v was the root of a tree, not a star. If the edge found was null, v's component had more than B^2 null edges and therefore more than B vertices. In the last two cases, v's component is promoted. Otherwise, v can hook to its mate vertex y. Also, note that $mate(v)$ is used in our analysis but, in fact, it need not be saved since it is defined by $L(v)$ whenever it is needed.

2. PSEUDOTREE CONTRACTION

for each vertex v such that $not(root(v))$ **do in parallel**
$bold(v) \leftarrow id(v) < id(p(v))$

for $j \leftarrow 1$ **to** $\lceil \alpha\sqrt{\lg n} \rceil$ **do**
for each vertex v **do in parallel**
v executes the appropriate CR rule

COMMENT: Vertices execute the rule enabling statement and then apply the CR rules for $\lceil \alpha\sqrt{\lg n} \rceil$ times, which forces all components with fewer than B members to become rooted stars. Observe that after this step components with more than B members may become rooted trees or non-rooted pseudotrees. This step takes $O(\sqrt{\log n})$ time.

3. ROOT RECOGNITION STEP

for each vertex v such that $p(v) = v$ **do in parallel**
    $root(v)$
    $mate(v) \leftarrow v$
    if $subphase(v) = i - 1$
        then $subphase(v) \leftarrow i$
        else $promoted(v)$
            $next(twin(first(L(v)))) \leftarrow next(last(L(v)))$
            $next(last(L(v))) \leftarrow nil$

COMMENT: The new roots of the newly formed trees or stars identify themselves. If root v was also root in the previous subphase, its component may still be unpromoted. But, if there was at least one subphase j, where $subphase(v) < j < i$, during which v did not hook, then during subphase j vertex v belonged to either a busy component or a rooted tree with height more than B that had an internal hooking. In either case the component was promoted. Note that v performed the edge-plugging step (step 4, below) during subphase $subphase(v)$. Lemma 7 of the next subsection explains why the effect of v's plugging step can be reversed by the last two statements.

4. THE EDGE-PLUGGING STEP

for each vertex v such that $not(root(v))$ **and** $subphase(v) = i - 1$
    do in parallel
        $next(last(L(v))) \leftarrow next(twin((v, w)))$
        $next(twin((v, w))) \leftarrow first(L(v))$

COMMENT: Non-root vertices that were roots in the previous subphase and therefore hold an edge-list, plug it into their mate's edge-list. At this point each unpromoted star has all the edges of its component members contained in its root's edge-list (Lemmas 7, 6).

5. EDGE RENAMING AND IDENTIFICATION OF INTERNAL EDGES

for each edge (v, w) **do in parallel**
  if $p(v) = r$ **and** $root(r)$
    then rename (v, w) to (r, w)
  if $p(w) = r$ **and** $root(r)$
    then rename (v, w) to (v, r)

for each edge (r, r) **do in parallel**
  $null((r, r))$

for each vertex v such that $not(root(v))$ **and** $root(p(v))$ **in parallel do**
  $null(last(L(v)))$

COMMENT: Edges identify their new endpoints. Those having both endpoints on the same root are internal and so nullify themselves. The $root(r)$ condition assures that lists of non-rooted pseudotrees are not altered. Finally, the last statement explicitly nullifies the unnecessary fake edges at the end of the edge-lists.

6. NULL-EDGE REMOVAL

for $j \leftarrow 1$ **to** $2\lceil\sqrt{\lg n}\rceil + 1$ **do**
  for each edge e **do in parallel**
    if $null(next(e))$
      then $next(e) \leftarrow next(next(e))$

for each vertex v such that $root(v)$ **do**
  if $null(first(L(v)))$
    then $first(L(v)) \leftarrow next(first(L(v)))$

COMMENT: Blocks composed of up to $\lfloor 2^{2\sqrt{\lg n}} \rfloor$ consecutive null edges are removed. Unpromoted stars now contain no null edges. This ensures that they will have a successful hooking at the next subphase.

8 Correctness and Time Bounds

Theorem 2 *The algorithm correctly computes the connected components of a graph in $O(\log^{3/2} n)$ parallel running time without concurrent writing.*

Proof. Correctness follows from Lemma 14 below. The running time comes from the fact that there are $\lceil \sqrt{\lg n} \rceil$ phases, each taking $O(\log n)$ parallel time. □

We prove that in the beginning of each subphase j, the root r of each rooted tree P holds in $L(r)$ all the edges (v, w) which in the beginning of phase i belonged to the edge-lists $L(v)$ of vertices $v \in P$ and were not deleted as internal in previous subphases.

Let $G_i = (V_i, E_i)$ be the input graph of phase i. We define $M_j = (V_i, mate)$ to be the pointer graph composed of the *mate* pointers of V_i at the beginning of subphase j. Note that M_j is a pseudoforest.

Lemma 7 *At the beginning of subphase j each root r of a rooted tree $(C, mate)$ in M_j satisfies invariant 2.*

Proof. We prove the lemma by induction on j. In the beginning of the first subphase M_1 is composed of $|V_1|$ vertices, and the lemma holds true.

We assume that the lemma is true at the beginning of subphase j. During subphase j the unpromoted roots of M_j hook to form larger components (step 1 of procedure subphase). Then, in step 3, some roots recognize themselves as the roots of M_{j+1}. We must prove that these roots satisfy invariant 2 (see page 25) at the beginning of subphase $j + 1$.

Let r be a root at the beginning of subphase $j + 1$. We distinguish two cases:

(1) r was also a root in the beginning of subphase j. Then, for every vertex v that belonged to a tree which during the hooking step hooked on r's tree, there is a path of mate pointers from v to r. So, after the plugging step (step 4) Lemma 5 applies. Moreover, note that step 6 removes only null edges. Therefore, at the beginning of subphase $j + 1$, root r satisfies invariant 2.

(2) r was not a root in the beginning of subphase j, therefore r was a part of a promoted component. We have seen (Lemma 6) that the effect of having all vertices in a cycle execute the plugging step (Figure 14) is to break the edge-lists in two rings. To reverse the effect of plugging, re-join these two rings of edges into a chain. This can be done by r in subphase j by executing the following statements:

$next(twin(first(L(r)))) \leftarrow next(last(L(r)))$

$next(last(L(r))) \leftarrow nil$

To prove that the above statements correctly re-join the rings, we have to show that (a) $first(L(r))$ still points to edge (r, w), the edge that r chose during its most

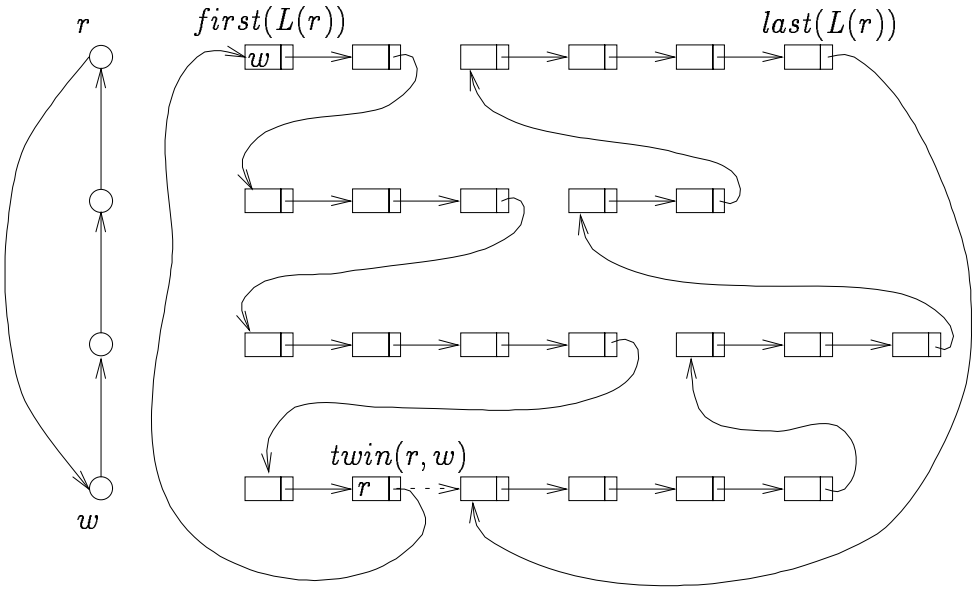

Figure 14: Assume that, at a later subphase j, vertex r becomes the root of the pseudotree. Then, r can easily reverse its edge-plugging. In the figure, the dashed line denotes one of the two pointers that must be changed. The other is the pointer out of $last(L(r))$, which should become *null*.

recent hooking subphase $j' < j$, (b) $twin((r,w)) = (w,r)$, and (c) no edge shortcutted over (r,w), (w,r), or $last(L(r))$ during subphases j' through j.

The $first(L(r))$ pointer is only altered in step 6 when $root(r)$. However, $root(r)$ was false in every subphase after j' and before j; so (a) is true. Twin functions are only computed at the end of a phase, not during subphases, so (b) is also true. Finally, as one can see by examining step 5 of procedure `subphase`, these three edges were never nullified; so, (c) is also true. □

We should note here that since only the edge-list of an already promoted component is ever divided into two rings, one can actually postpone dealing with them until the end of the phase. Then one can construct the edge-lists of the components from scratch. This takes $O(\log n)$ time, so it can be done at no extra cost. The reason that we chose to describe the rejoining steps as we did in Lemma 7 instead, was to provide the details for an implementation of Algorithm 1 (Section 5).

Lemma 8 *If at the end of subphase j some component is busy, it has been promoted.*

Proof. By definition, a component is busy if at the end of a subphase it is still a pseudotree. Of course, such a component will not pick a mate in the beginning of the next subphase because it has no root to do the operation. Procedure `subphase` contracts the components for $\lceil \alpha \lg B \rceil$ steps. So, according to Lemma 4, pseudotrees with circumference less than or equal to B will be rooted trees at the end of subphase j and therefore not busy. Thus, a busy component had more than B members, and so it has been promoted to the next phase. □

Lemma 9 *Let C be a component which in subphase j has an internal hooking. Then C has been promoted.*

Proof. Recall that internal hooking happens when C picks as a mate an internal edge (without knowing it). Also note that such a hooking cannot happen in subphase 1 because all components enter the first subphase without internal edges. So, $j > 1$.

At the end of subphase $j-1$ all components within distance B from the root have reached the root (Lemmas 3 and 4) and have nullified the appropriate entries in the root's edge list (step 5 of procedure `subphase`). So, an internal edge must be connecting the root of the component to some vertex v in the tree, which was at a distance more than B from the root, since it did not have enough time to reach the root. Thus, there are at least B components that reached the root (namely, those in the path from v to the root), and so C has been promoted. □

Lemma 10 *If a component C fails to find a mate at some subphase j, then either it has been promoted or it is not connected to any other component.*

Proof. Let C be a component that cannot find a mate at some subphase j of phase i. We distinguish two cases: (a) C found no edges in its edge list, and (b) C found a null edge in its edge list.

(a) According to Lemma 7, in the beginning of each subphase the root of a component C holds all the edges of its members that have not been removed as null. So, if an edge of C was not null, it would be in C's edge list. Therefore, C is not connected to any other component in the graph.

(b) First we observe that $j > 1$. Note that at the end of subphase $j-1$ the algorithm performed the null edge removal step for $2\lceil \lg B \rceil + 1$ times. This removes any block containing up to B^2 null edges from the edge list of the component's root. Next we observe that any component with fewer than B members cannot have more than $B(B-1)/2$ internal edges. This is a consequence of Invariant 1. So, a root may find a null edge in its edge list only if its component is bigger than B and therefore is promoted. □

Lemma 11 *Every active, non-promoted component at subphase j will have a successful hooking at subphase $j+1$.*

Proof. Let C be a component that is not promoted at the end of subphase j. C is a rooted star because $|C| < B$. Also, by Lemma 7, its root holds all the edges that belonged to the edge-lists of its vertices and were not deleted in previous subphases. Moreover, $L(r)$ contains no internal edges because they were all identified and deleted. So, if $L(r)$ contain any edges, r will have a successful hooking at the next subphase. □

Lemma 12 *After $\lceil \lg B \rceil$ successful hookings in some phase, a component has been promoted.*

Proof. First we show that if a root r is not promoted after performing k successful hookings, it was continuously hooking to components having successful hookings. For, if one of these components had an internal hooking, it was promoted; therefore, r's component was part of a promoted component.

Next, we can prove by induction that, after each successful hooking at subphase j, components have sizes at least 2^j. Therefore after $\lceil \lg B \rceil$ successful hookings, r is the root of a component of size B and thus has been promoted. □

Lemma 13 *At the end of phase i each component is either promoted or not connected to any other components.*

Proof. Each phase is composed of $\lceil \lg B \rceil$ subphases. In the beginning of a subphase each component is either ready or busy. A busy component cannot pick a mate, but, according to Lemma 8, it is a promoted pseudotree. On the other hand, a ready component is a rooted tree which can pick a mate from its edge list that contains all the edges of its members (Lemma 7). So, the reason for which a ready component may not be able to find a mate (according to Lemma 10), is that the component is promoted or done. Otherwise the component finds a mate.

A hooking may either be successful or internal. An internal hooking, according to Lemma 9, can only happen to an already promoted component. So, we only have to follow components which have successful hookings for $\lceil \lg B \rceil$ subphases. But these components (Lemma 12) have been promoted at the end of the last subphase. □

Lemma 14 *In the beginning and at the end of each phase i (a) The components are rooted stars. (b) The size of each active component is at least B^i. (c) Invariant 1 is preserved. (d) There are no internal edges. (e) There is no concurrent writing.*

Proof. (a) This is obviously true in the first phase, where components are composed of a single vertex, the root. During the subphases, these components are hooked to form a pseudoforest. Then, at step 3 of procedure `phase`, the pseudoforest is transformed to a set of rooted stars. The remaining steps do not affect the structure of the components, and so, in the beginning of the next phase, the components are stars.

(b) This is immediate from Lemma 13.

(c) Again, invariant 1 (see page 25) is true for the first phase. For the remaining phases, step 4 of procedure `phase` uses merge sort to identify multiple edges and the step 6 removes them.

(d) Internal edges are nullified in step 4 and are removed in step 6 of the procedure `phase`.

(e) The elimination of concurrent writing has been discussed at the points where preventing concurrent writing required new techniques. The absence of concurrent writing follows from examination of the algorithm in Sections 6 and 7. □

9 Conclusions

We have presented an algorithm that finds the connected components of an undirected graph for the CREW PRAM model of parallel computation. This algorithm works in $O(\log^{3/2} n)$ time, and narrows the gap of the performance between several CREW and CRCW PRAM graph algorithms by a factor of $\log^{1/2} n$.

This result settles a question that remained unresolved for many years because a connectivity algorithm for this model with running time $o(\log^2 |V|)$ was a challenge

that had thus far eluded researchers [19, page 894]. Recently, Chong and Lam [7] have used a recursive version of our growth control schedule (Section 5) to improve the running time to $O(\log n \log\log n)$. An apparently necessary idea of theirs not present in our algorithm is to hook vertices of largest degree first. Also, since the results of this paper were reported, Nisan, Szemerédi, and Wigderson [24] have described an $O(\log n^{3/2})$ *space* algorithm for the single connectivity problem. This result subsumes our time bound, but not our processor bound. A paper by Karger, Nisan, and Parnas [18] which relates to this latter result has bounds equal to ours. Despite these several results, however, a conjecture posed by Wyllie [33] and Shiloach and Vishkin [26] remains open. The conjecture states that no $O(\log n)$-time algorithm exists for the exclusive-write PRAM model.

The techniques presented in this paper have been used to design new parallel algorithms for the minimum spanning tree problem [17]. Other algorithms having running times that depend on the connectivity algorithm include the Euler tour on graphs [3, 4], biconnectivity [28], the ear decomposition [20, 22] and its applications on 2-edge connectivity, triconnectivity, strong orientation, s-t numbering etc. See the surveys by Karp and Ramachandran [19] and by Vishkin [31] for more details on this.

We should also mention that, with a minor modification our algorithm works on the weaker CREW PPM (Parallel Pointer Machine) model [13]. The modification is to substitute the sorting routine we use at the end of each phase by the asymptotically optimal sorting algorithm of Goodrich and Kosaraju [13]. In the PPM model, the memory can be viewed as a directed graph whose vertices correspond to memory cells, each having a constant number of fields. The PPM is based on a generalization of Knuth's linking automaton.

Acknowledgements. The authors would like to thank Prof. Adonis Simvonis and the anonymous referee for their careful reading of this paper and substantial help in improving the presentation.

References

[1] A. Aggarwal, R.J. Anderson, and M.-Y. Kao. Parallel depth-first search in general directed graphs. In *Proc. 21st Annual ACM Symposium on the Theory of Computing*, pages 297–308, May 15-17 1989.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[3] M. Atallah and U. Vishkin. Finding Euler tours in parallel. *Journal of Computer and System Sciences*, 29:330–337, 1984.

[4] B. Awerbuch, A. Israeli, and Y. Shiloach. Finding Euler circuits in logarithmic parallel time. In *Proc. 16th Annual ACM Symposium on Theory of Computing*, pages 249–257, 1984.

[5] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, C-36:1258–1263, 1987.

[6] F.Y. Chin, J. Lam, and I-N. Chen. Efficient parallel algorithms for some graph problems. *Communications of ACM*, 25(9):659–665, September 1982.

[7] K.W. Chong and T.W. Lam. Finding connected components in $O(\log n \log \log n)$ time on the EREW PRAM. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms*, January 1993.

[8] R. Cole. Parallel merge sort. *SIAM Journal of Computing*, 17(4):770–785, August 1988.

[9] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *Proc. 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 478–491, 1986.

[10] S. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal of Computing*, 15(1):87–97, February 1986.

[11] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM Journal of Computing*, pages 1046–1067, 1991.

[12] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages 315–324, 1987.

[13] M.T. Goodrich and S.R. Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. In *Proc. 30th IEEE Symposium on Foundations of Computer Science*, pages 190–195, 1989.

[14] T. Hagerup. Towards optimal parallel bucket sorting. *Informaton and Computation*, 75:39–51, 1987.

[15] D.S. Hirschberg, A.K. Chandra, and D.V. Sarwate. Computing connected components on parallel computers. *Communications of ACM*, 22(8):461–464, August 1979.

[16] D.B. Johnson and P. Metaxas. Connected components in $O(\log^{3/2}|V|)$ parallel time for the CREW PRAM (extended abstract). In *Proc. of 32nd IEEE Symposium on the Foundations of Computer Science*, pages 688–697, October 1991.

[17] D.B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. In *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 363–372, June 1992.

[18] D. R. Karger, N. Nisan, and M. Parnas. Fast connected components algorithms for the EREW PRAM. In *Proc. of 4th Symposium on Parallel Algorithms and Architectures*, pages 373–381, June 1992.

[19] R. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. *Handbook for Theoretical Computer Science*, 1:869–941, 1990.

[20] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (EDS) and s-t numbering in graphs. *Theoretical Computer Science*, 47:277–298, 1986.

[21] P. Metaxas. *Parallel Algorithms for Graph Problems.* PhD thesis, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH, July 1991.

[22] G.L. Miller and V. Ramachandran. Efficient parallel ear decomposition with applications. Manuscript, 1986. An updated version appears in Chapter 7 of J.H. Reif, ed., *Synthesis of Parallel Algorithms*, Morgan Kaufman, 1993.

[23] D. Nath and S.N. Maheshwari. Parallel algorithms for the connected components and minimal spanning tree problems. *Information Processing Letters*, 14(1):7–11, March 1982.

[24] N. Nisan, E. Szemerédi, and A. Wigderson. Undirected connectivity in $O(log^{1.5}n)$ space. In *Proc. of 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 24–29. IEEE, October 1992.

[25] J. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20:229–234, 1985.

[26] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[27] R.E Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), 1972.

[28] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 14(4):862–874, 1985.

[29] U. Vishkin. Implementation of simultaneous memory address access in models that forbid it. *Journal of Algorithms*, 4:45–50, 1983.

[30] U. Vishkin. On efficient parallel strong orientation. *Information Processing Letters*, 20:235–240, June 1985.

[31] U. Vishkin. Structural parallel algorithms. Technical Report UMIACS-TR-91-53, CS-TR-2652, University of Maryland, College Park, Maryland 20742, April 1991.

[32] R.J. Wilson. *Introduction to Graph Theory.* Longman, Inc., New York, 3rd edition, 1985.

[33] J.C. Wyllie. *The Complexity of Parallel Computation.* PhD thesis, Computer Science Dept., Cornell University, Ithaca, NY, August 1981.

Symbols Used in Paper.

$O(\cdots)$	Big "Oh", italic, in formulas
O	capital "Oh" in text
o	lower case "oh" in text
$o(\cdots)$	Little "oh", italic, in formulas
$0, 1, 2, 3, \ldots$	Arabic numerals. Must distinguish 0 (zero) from "Oh"s
a,b,...,k,l,m,n	lower case English letters in text
lg	logarithm symbol in math formulas: Roman type as shown
log	logarithm symbol in math formulas: Roman type as shown
$k, l, m, n, \ldots$	italic lower case English letters in math formulas
$G, E, V, \ldots$	italic upper case English letters in math formulas
α	Greek alpha in math formulas