1-1-1991

Connected Components in O(lg3/2|V|) Parallel Time for the CREW PRAM

Donald B. Johnson
Dartmouth College

Panagiotis Metaxas
Dartmouth College

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

PCS-TR91-160

Donald B. Johnson
Panagiotis Metaxas

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

Donald B. Johnson* Panagiotis Metaxas†
Dartmouth College‡

Abstract

Computing 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 $O(\lg^2 n)$ time using $n^2/\lg^2 n$ processors [CLC82, HCS79]. For the CRCW PRAM model in which concurrent writing is permitted, the best known algorithm runs in $O(\lg n)$ time using almost $(n+m)/\lg n$ processors [SV82, CV86, AS87]. Unfortunately, simulating this algorithm on the weaker CREW model increases its running time to $O(\lg^2 n)$ [CDR86, KR90, Vis83]. We present here an efficient and simple algorithm that runs in $O(\lg^{3/2} n)$ time using $n + m$ CREW processors. Finding an $o(\lg^2 n)$ 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 $i = 1, \ldots, k-1$. We say that two vertices belong to the same *connected component* if and only if there is a path connecting them.

The problem of finding 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 expressed by a set of pointers p such that, vertices v and w are in the same class if and only if $p(v) = p(w)$ (Figure 1).

*email address: djohnson@cardigan.dartmouth.edu
†email address: takis@dartmouth.edu
‡Department of Mathematics and Computer Science, Hanover, NH 03755

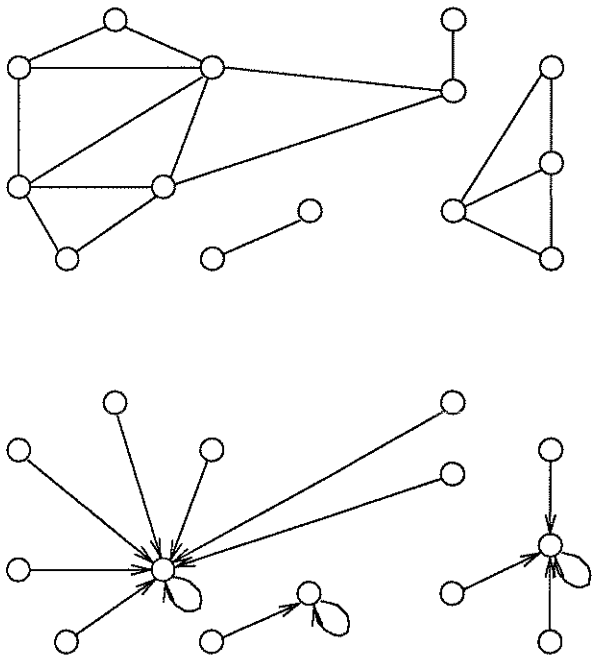

Figure 1: A graph G with three connected components (top) and the pointers p at the end of the computation (bottom).

It is well known that the connectivity problem has a linear-time sequential solution using depth-first search [Tar72], but implementation of this method in parallel seems very difficult [Rei85]. No polylogarithmic-time deterministic parallel algorithm is known for depth-first search, and the best randomized algorithm [AAK89] runs in $O(\log^7 n)$ using almost n^4 processors.

The best known deterministic parallel algorithm for connectivity runs in $O(\lg^2 n)$ time on the CREW PRAM using $n^2/\lg^2 n$ processors [CLC82, HCS79]. (By $\lg n$ we denote $\log_2 n$.) In this model of parallel computation, concurrent writing to any memory location by more than one processor is not allowed [KR90]. For the CRCW PRAM model in which concurrent writing is permitted, the best known algorithm runs in $O(\lg n)$ time using almost $(n+m)/\lg n$ processors [SV82, CV86, AS87]. There is also an optimal randomized algorithm [Gaz91] with the same time complexity. Unfortunately, simulating an algorithm designed for this model on the weaker CREW model increases its running time to $O(\lg^2 n)$ [CDR86, KR90, Vis83].

We present an efficient and simple algorithm that runs in $O(\lg^{3/2} n)$ time using $n+m$ CREW processors. This is a somewhat surprising result because, as Karp and Ramachandran have noted [KR90], "graph problems seem to need at least $\lg^2 n$ time on a CREW and EREW PRAM and $\lg n$ on a CRCW PRAM". This observation was attributed to the fact that many parallel graph algorithms need a connectivity procedure as a subroutine. Indeed, our result improves the running times of algorithms for several graph problems, including ear decomposition [MR86], biconnectivity [TV85], strong orientation [Vis85] and Euler tours [AV84].

Our algorithm is the first parallel connectivity algorithm with running time $o(\lg^2 n)$ for this model. In the process of devising the algorithm, algorithmic techniques were invented which may have more 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 and unique of these is the scheduling of the rate of growth of connected components. We have succeeded not only in defining an optimal rate for components to grow (so as to control the overhead attendant on redundant edge removal) but in enabling the algorithm when necessary to recognize episodically when a component is growing too fast and therefore can be ignored.

We briefly describe here the model of parallel computation we use. 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".) If we don't allow any conflicts in the reading from and writing to the shared memory, the model is

called an EREW PRAM. If we allow only concurrent readings, we have a CREW PRAM. Finally, in the CRCW PRAM simultaneous writing is permitted.

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 discusses how the solutions proposed interact within 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 discusses the conclusions and open problems.

2 Discussion

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 will assume that there is one processor *Proc(i)* assigned to each vertex $i \in V$ and one processor *Proc(i,j)* assigned to each edge $(i, j) \in E$.

The algorithm will deal with *components*, which are sets of vertices found 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 each component is a single vertex. 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 *the 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 with one of its vertices as its representative. Each representative receives into its edge-list all the edges contained in the edge-lists of its pseudotree.

3. Edges internal to components are removed.

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

The existence of 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.

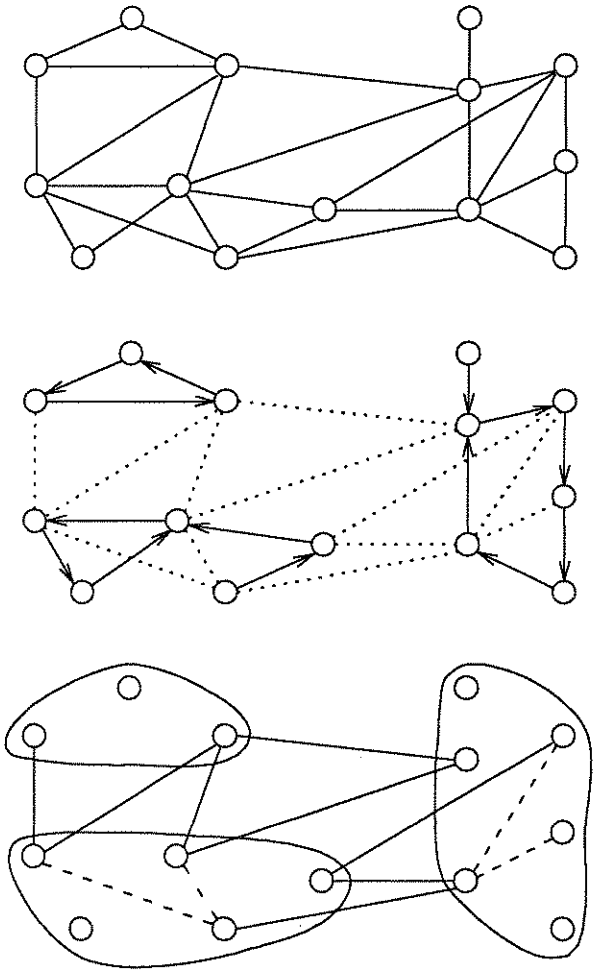

Figure 2: (Top) the input graph G. (Middle) Vertices have picked their mates. Dotted are those edges they were not picked by any vertex. An arc points from a vertex to its mate. Three pseudotrees are shown in this figure. Note that each pseudotree contains a cycle. (Bottom) The new components have been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

Previous algorithms deal with this problem in different ways. [HCS79] spends $O(\lg n)$ time to create trivial pseudotrees while [SV82] uses the power of concurrent write to avoid their creation.

We can solve this problem using the cycle-reducing shortcutting technique we introduce in the 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.

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 [SV82] uses the power of concurrent writing to overcome this problem, while [HCS79] uses an adjacency matrix and $O(n^2)$ processors to solve it in $O(\lg n)$ time.

The edge-plugging scheme we introduce in section 4 achieves the objective 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 attempt to find a mate many times without success if it picks an internal edge. On the other hand, removing all the internal edges before picking an edge may also take a lot of time.

This problem is solved by the *growth-control schedule* we introduce in section 5. Components grow in size in a uniform way that controls their 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 $o(\lg^2 n)$ running time. We have succeeded not only in defining an optimal rate for components to grow (so as to control the overhead attendant on redundant edge removal) but in enabling the algorithm to recognize when necessary whether a component is growing too fast and therefore can be ignored.

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

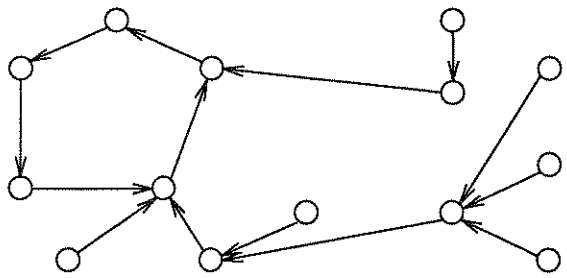

Figure 3: A pseudotree.

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

A *pseudotree* $P = (C, D)$ is a maximal 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* R with root r, is a rooted tree whose arcs are of the form (x, r) with $x \in R$, i.e. all of whose arcs point to r.

A *pseudoforest* $F = (V, A)$ is a collection of pseudotrees. An equivalent definition of a pseudoforest [Ber85] is a *functional graph* $F = (V, f)$, the graph of a finite function f on a set of vertices V. We can think of f as being implemented by a set of pointers p, so we will also call F a *pointer graph* (V, p). We will refer to pseudoforests using any of the three equivalent definitions.

We define the pseudotree contraction problem as follows:

Problem 1 *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 $v \in C$, then $(v, r) \in D'$.*

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

Pseudoforests are especially interesting in parallel computation, since they arise often in parallel graph algorithms when vertices of a graph draw simultaneously a

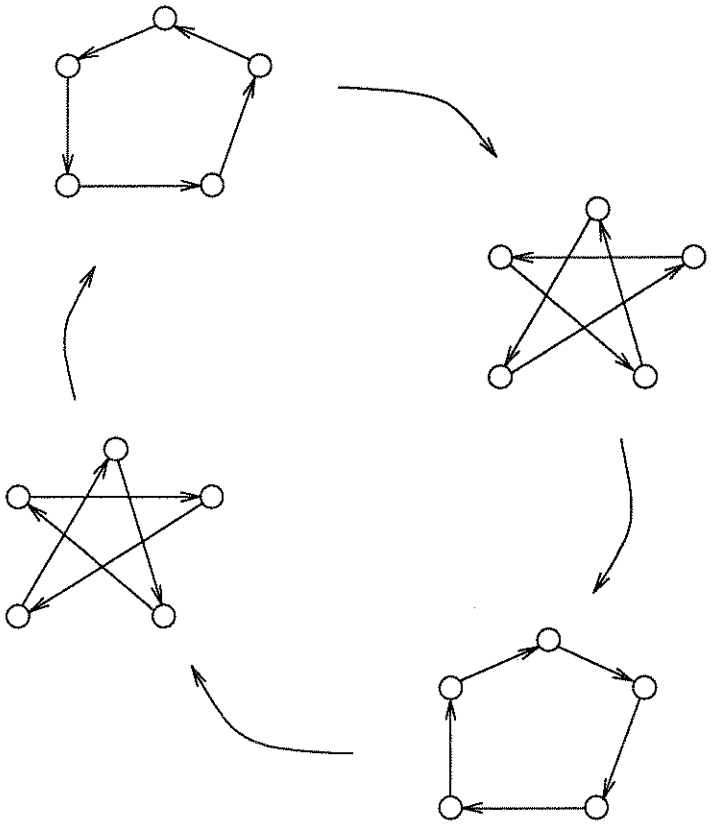

Figure 4: The usual pointer jumping technique cannot deal with cycles.

pointer to a neighbor, an operation called *hooking*. Connectivity, minimum spanning tree, maximal independent set and tree-coloring, [JM91, NM82, SV82, HCS79, AS87, GPS87, Met91, JM92] are among the problems 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 [Wyl81] does not work on a cycle is that it will not terminate (Figure 4). Even if one modifies this technique to recognize a cycle by keeping track of all the vertices pointed to by some pointer, 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) short-cutting rules*. These rules are used to reduce a pseudotree to a rooted star, 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. (Figure 5.)

Let $G = (V, E)$ be a graph. We assume that each vertex $v \in V$ has been assigned an $id(v) \in Id$, which is a distinct integer, for example the id-number of the processor which is responsible for the vertex. 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 will do that using the CR shortcutting rules (Figure 6). These rules assign the vertex r having the smallest id among all the vertices in the cycle to be the root of the future rooted tree. The idea behind the CR rules is that they will not let any of the vertices of the pseudotree shortcut over the future root, even though at the outset the root is not known.

To use the CR rules we will create two kinds of pointers p: *bold* and *light*. Bold pointers belong only to possible roots of a pseudotree. All other vertices have light pointers. Each vertex v of the pseudotree will first execute the *rule enabling* statement:

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.

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** $bold(v) \leftarrow false$
$p(v) \leftarrow p(p(v))$
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 }

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

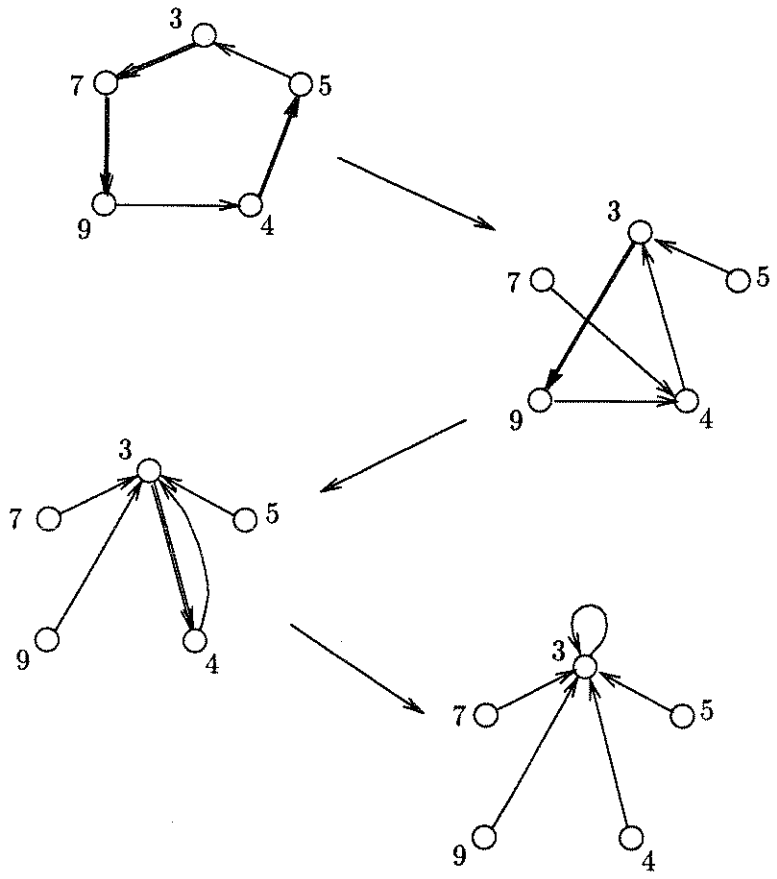

Figure 5: Using the CR shortcutting rules the cycle can be reduced to a rooted tree in logarithmic number of steps.

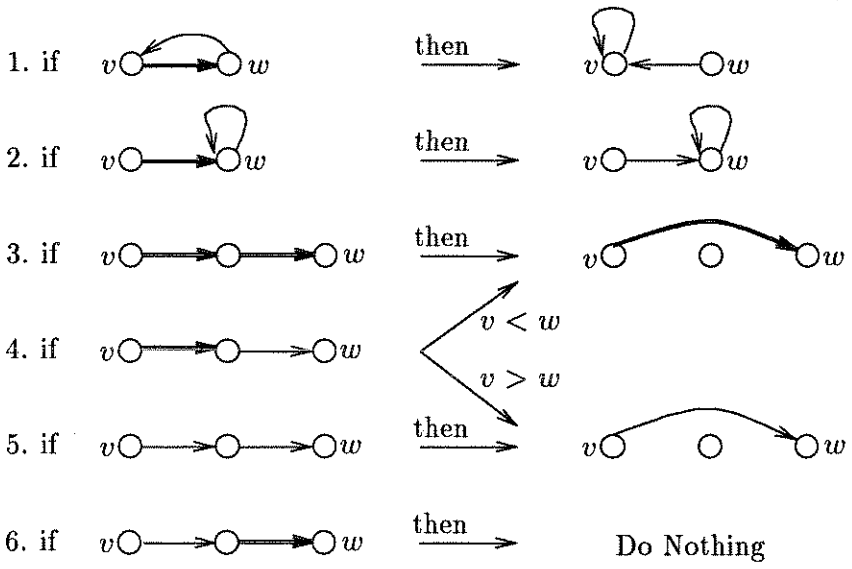

Figure 6: Cycle-Reducing rules. (1) and (2) Terminating rules, (3) Bold-bold rule, (4) Bold-light rule, (5) Light-light rule, (6) Light-Bold rule. Comparison between vertices means comparison between their corresponding id's.

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 will have 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. So, no vertex in c will shortcut over r because the light-bold rule applies. Moreover, r will continually shortcut over the other pointers in the cycle since the bold-bold and bold-light rules permit it. Eventually, the first terminating rule will apply and r will reach itself, effectively becoming the root of a rooted tree. □

Let P be a pseudotree composed of n vertices and n pointers (arcs) 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 *distance* d_v of a vertex $v \in C$ to be the number of pointers on the shortest directed path from v to r that uses a last pointer. Also, we define the *k'th distance* 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 will 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. More specifically, we will show that:

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. 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:

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

With this hypothesis we now prove:

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

We will 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).

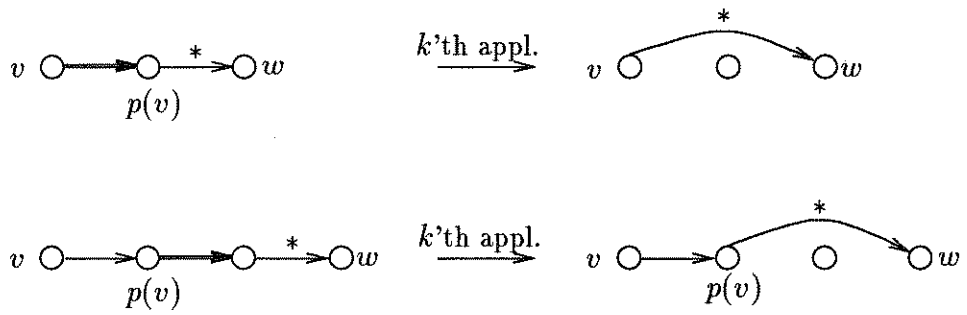

Figure 7: Top figure: Case 1 of the Lemma. Bottom Figure: Case 2 of the Lemma. An asterisk (*) over a pointer means that this pointer could be either bold or light.

CASE 1: Let's assume that v's pointer is bold or $p(v)$'s pointer is light. Since in all cases v's pointer will shortcut (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 $d_w^k \leq \lceil \frac{2d_w^{k-1}}{3} \rceil$. We want to prove that $d_v^k \leq \lceil \frac{2d_v^{k-1}}{3} \rceil$.

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

$$\begin{aligned} d_v^k &\leq 1 + d_w^k \\ &\leq 1 + \lceil \frac{2d_w^{k-1}}{3} \rceil \\ &= 1 + \lceil \frac{2(d_v^{k-1} - 2)}{3} \rceil \\ &= 1 + \lceil \frac{2d_v^{k-1}}{3} - \frac{4}{3} \rceil \\ &\leq 1 + \lceil \frac{2d_v^{k-1}}{3} \rceil - 1 \\ &= \lceil \frac{2d_v^{k-1}}{3} \rceil. \end{aligned}$$

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

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

$$d_v^k \leq 2 + d_w^k$$

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

□

Now we prove the following lemmas that will be useful in the connectivity algorithm. For increased efficiency, we will assume that the rules are applied in two steps. In the first, the two terminating rules apply, and in the second step, the remaining rules apply. Let $\alpha = 1.71 > 1/(\lg \frac{3}{2})$.

Lemma 3 *When the cycle-reducing rules are applied $\lceil \alpha k \rceil$ times to a rooted tree, any vertex within distance 2^k 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 k times to a rooted tree, any vertex within distance $(\frac{3}{2})^k - 2$ from the root reaches the root of the tree. The Lemma follows by observing that $\lceil \alpha k \rceil > k/(\lg \frac{3}{2})$, therefore $(\frac{3}{2})^{\lceil \alpha k \rceil} > 2^k$. □

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

Proof. We observe that $d_r(r) = circ(P)$. If $circ(P) < 2^k$, then r's pointer will reach itself in $\lceil \alpha k \rceil$ steps (Lemma 3) and r will become the root of a rooted tree. On the other hand, any vertex at distance 2^k from r will reach r after $\lceil \alpha k \rceil$ applications of the CR rules. □

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

Theorem 1 *A pseudotree P with $h = \max_{v \in P}\{d_v\}$ will be 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.*

□

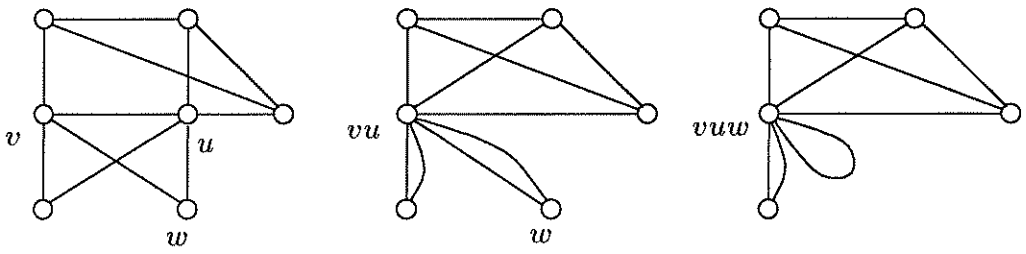

Figure 8: Contraction of the vertices v, u, w.

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 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 [Wil85]. 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.[1] 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 we discussed in the previous section, the vertex subsets that appear naturally in parallel computation are pseudotrees. 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 *edge-list augmentation* problem naturally arises:

Problem 2 *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 will show how to solve this problem in constant time without memory access conflicts once the representative of the pseudotree is known.

[1] Sometimes this new vertex is called a *supervertex*.

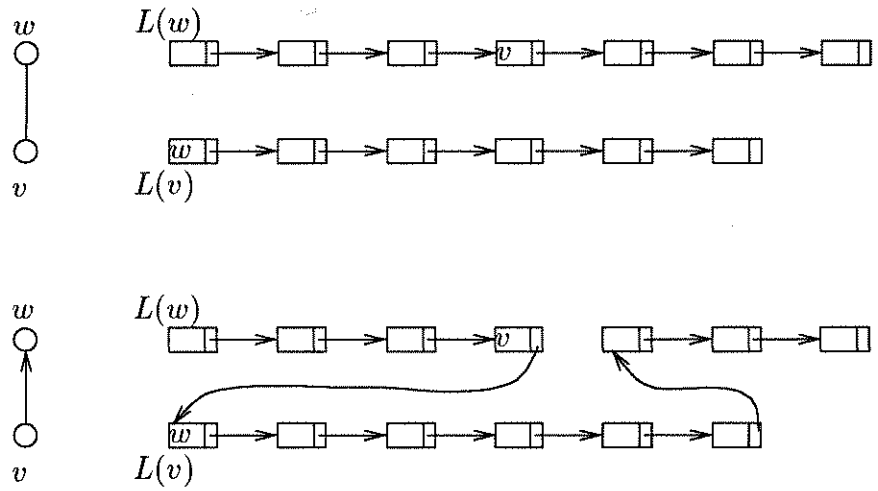

Figure 9: Edge lists of nodes v and w before (top) and after the edge plugging step (bottom).

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 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.

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 $(v, w) \in D$ be its outgoing arc of the pseudotree P. According to this scheme, v will *plug* its edge-list $L(v)$ into w's edge-list by redirecting a couple of 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 $(w, v) = twin(v, w)$
  $next(last(L(v))) \leftarrow next(w, v)$
  $next(w, v) \leftarrow first(L(v))$

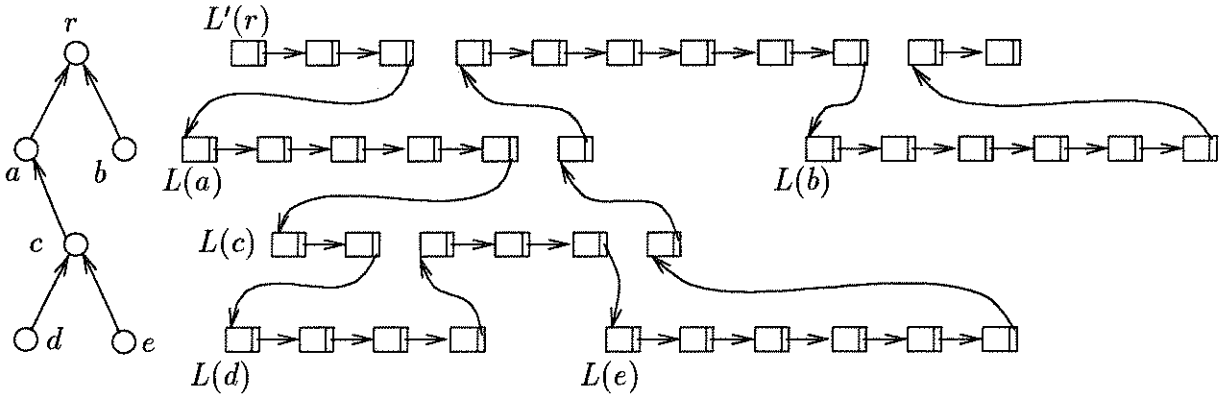

Figure 10: The effect of the plugging step execution by all vertices of a pseudotree but the representative r. On the left is $P = (C, D)$. On the right is $L'(r)$ after the execution of the plugging step.

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 *If there is a directed path p from v to r in $P = (C, D)$, then after the execution of the plugging step by all the vertices in p but r,*

$$(v, w) \in L(v) \Rightarrow (v, w) \in L'(r).$$

No writing conflict occurs during this operation.

Proof. The path p in P composed of edges in D corresponds to a unique sequence of *next* pointers in $L'(r)$, through which any edge in $L(v)$ is accessible. Moreover, the only processor that will access pointer $next(twin(v, w))$ is the processor assigned to vertex v. Therefore there is no writing conflict. □

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.

One may ask what the effect is of having the edge-plugging step being executed by the vertices of the pseudotree *before* determining the representative. (In particular, the connectivity algorithm we will present in section 6 may execute the plugging step before the contraction of a pseudotree to a rooted tree.) In other words, what if the representative also executes the edge-plugging step.

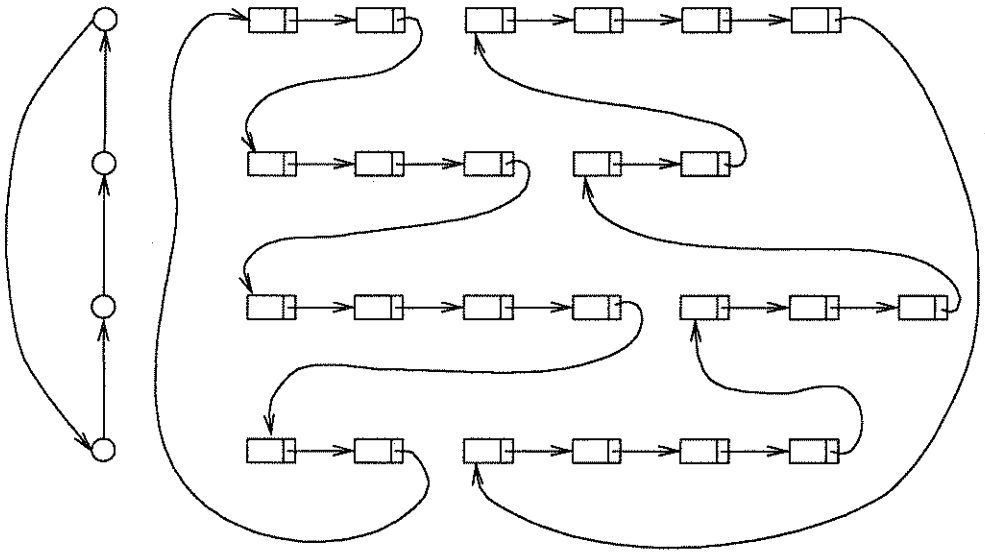

Figure 11: When all the vertices in a cycle execute the plugging step, their edge-lists are connected with two rings of edges. Observe that the *first* pointers of the vertices in the cycle end up in the first ring while the *last* in the second. This will enable the future root of the cycle to reverse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

The effect of this is to place the edges of the pseudotree's vertices into two linked rings (Figure 11). However, 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 6 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 that having a component pick a mate may be time consuming. We will make this statement more precise.

The cycle-reducing rules and the edge-plugging scheme provide the elements of a greedy-like connectivity algorithm that works correctly on the CREW PRAM model of parallel computation. Let $T(n)$ be some number that we will compute shortly. The algorithm is as follows:

Algorithm 1.

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

1. The representative of each component performs the hooking step if it can find a mate.
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 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. Moreover, we observe that, if we are sure that each component can hook in each phase, then only $\lg n$ 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:

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

Therefore, the number of phases $T(n)$ in this particular case would be $O(\lg^2 n)$. So, the crucial observation is the following:

In the beginning, components grow very fast. Later, when they have grown and there are lots of internal edges, there is a slowdown on the growth rate.

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

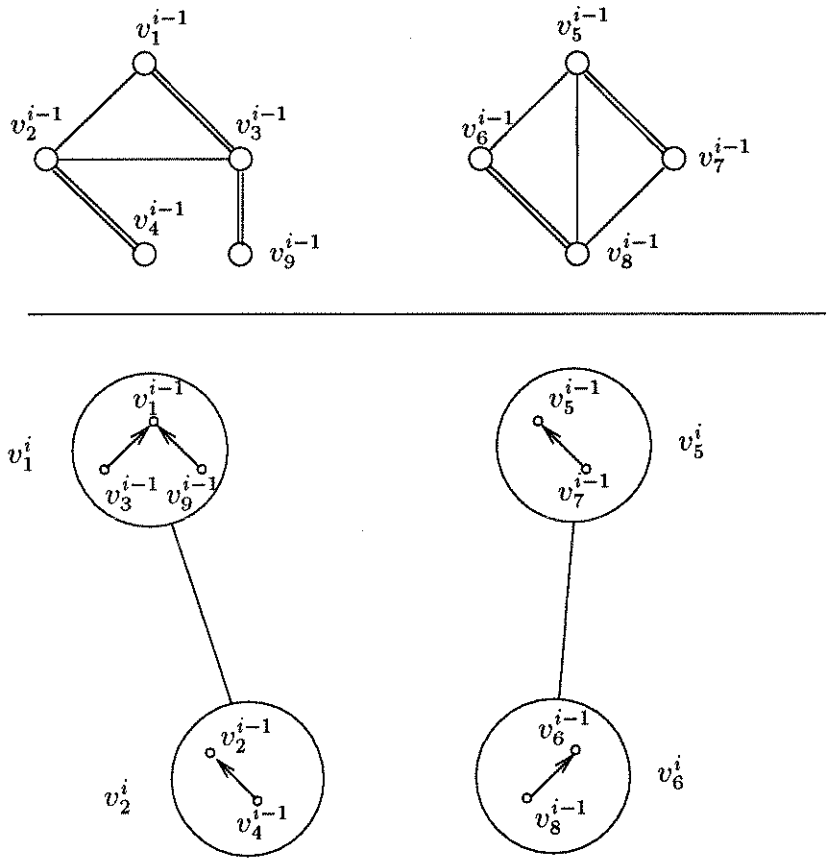

Figure 12: (Top) Graph G_{i-1} in the beginning of phase $i-1$. To simplify the figure, we assume that only the doubly marked edges will be used during phase $i-1$ for hooking. (Bottom) Graph G_i in the beginning of phase i. Vertex v_1^i represents the component $\{v_1^{i-1}, v_3^{i-1}, v_9^{i-1}\}$.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

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.

The growth-control schedule allows the size of the connected components of a graph to increase in a uniform way. To implement the schedule, the algorithm is divided into a number of phases. Phase i takes as input a graph $G_i = (V_i, E_i)$. Each vertex $v^i \in V_i$ represents a component formed in the previous phase $i-1$ containing several vertices $v_l^{i-1} \in V_{i-1}$, where $l \in Id$. We will write $v_l^{i-1} \in v^i$ to denote the membership of a component found in this way. Each edge $(v^i, w^i) \in E_i$ represents the set

$$\{(v_l^{i-1}, w_k^{i-1}) | (v_l^{i-1}, w_k^{i-1}) \in E_{i-1}, \text{ where } v_l^{i-1} \in v^i \text{ and } w_k^{i-1} \in w^i, \text{ with } l, k \in Id\}$$

of edges between distinct components v_l^{i-1} and w_k^{i-1}, which were found to belong to components v^i and w^i respectively of G^i during phase $i-1$. (See Figure 12.)

The purpose of each phase i is to allow just enough time for the components constituting the input graph G_i to grow by a factor of at least $2^{\sqrt{\lg n}}$. Therefore only $\lceil \sqrt{\lg n} \rceil$ phases are needed. When this growth has been achieved, the components are contracted to rooted stars using the CR rules, while useless edges (internal to the components and multiple among components) are removed from the graph. Then, a new graph G_{i+1} is created, induced by the components formed during the i'th phase. The next phase will operate on this graph.

A phase is further divided into stages. Stage j contains one hooking step followed by a number c_1 of pseudotree contraction steps and a number c_2 of useless-edges removal steps. The constants c_1 and c_2 are chosen so that components that have grown enough to enter the next phase can be recognized immediately. The growth factor has been chosen so that each phase has $O(\lg n)$ running time.

6 Outline of the Algorithm

6.1 Definitions

Let *critical bound* B be the quantity $2^{\sqrt{\lg n}}$. Some component C is in *category* i if $\lfloor B^i \rfloor \leq |C| < \lfloor B^{i+1} \rfloor$, $i = 0, 1, \ldots$, where $|C|$ denotes the number of vertices in C. No component may be in a category $i > i_{max} = \lceil \sqrt{\lg n} \rceil$.

The algorithm executes a number of *phases*, requiring that each component entering phase i is at least in category i. Therefore at most i_{max} phases are needed.

We say that some component has been *promoted* to phase i, if its size is at least $\lfloor B^i \rfloor$. We should note that the notion of promotion is needed mainly for the analysis; a component may or may not know whether it has been promoted during a phase.

In the beginning of the algorithm all components (which are simply single vertices), are in category 0. Each phase is divided into *stages*. In each stage 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 will contain many vertices, there may be *multiple edges* between two components. For each component pair, only one such edge needs to be kept in order to hook the two components. It is called the *useful edge*. The remaining 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 Stages 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[2] B^i vertices of the original graph G organized in a rooted star, with *representative* the root of the star denoted by v_C^i. These vertices were found to belong to C in previous phases, and they will 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 v_C^i, the representative. In the remainder of the section, when referring to a fixed phase i, we will use the term "component" to refer both to the set C of vertices in the component and to the representative v_C^i of the component. It should be clear from the context which meaning applies.

Each component is equipped with an edge-list. The following invariant will be used to prove correctness:

Invariant 1 *In the beginning of a phase i there is at most one edge between any two distinct components v^i and w^i. In particular,*

$$(v^i, w^i) \in E_i \text{ iff there was an edge } (v, w) \in E \text{ and } v \in v^i, w \in w^i$$

During each phase i, components continually hook to form bigger components. As we have described in previous sections, the hooking is done by having each component

[2]If a component has fewer than B^i vertices, then it is as big as it will get.

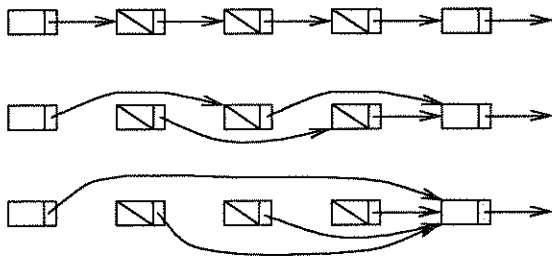

Figure 13: Example of two null-edge removal steps. The three null edges in the middle are being removed from the edge-list.

C pick, if possible, the first edge (v, w), $v \in C$ from its edge-list and point to its mate w. This operation is carried out by the representative of C. If there is no edge left in C's edge-list, then C is not connected to any other component, and it is done. At the end of the last phase all components are done.

The hooking operation creates a pseudoforest having bold and light pointers as needed by the cycle-reducing technique. Any component will perform $O(\sqrt{\lg n})$ hookings per phase, each one in a different stage.

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

Components that are rooted trees at the end of a stage are called *ready*. Those that still do not have a root are called *busy*. If some pseudotree C is busy at the end of a stage, its cycle had circumference greater that B, and therefore it has been promoted to the next phase. C will not hook in subsequent stages of this phase. At the end of the phase it will be 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 v but the roots of the contracted trees will 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 will place the edges of v's edge-list into r's edge-list. However, if v is a vertex of a busy component, this will not work, since there is no root in v's component and it will plug too. Fortunately, a busy component will be detected in a later stage 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 apparently internal and *nullify* themselves. Then, the edge-list of the root is cleared of null edges by $O(\sqrt{\lg n})$ *null-edge removal* steps. This is a simple application of the pointer-doubling technique (Figure 13):

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 B^2 null edges) can remove all its internal edges, thus it can find a non-null edge in the next stage. This ends a stage.

In the next stage, the roots of ready components will 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 stage j, because it contained a path longer than B. As a consequence, some internal edge e may not be nullified at the end of j. Moreover, the root of the component may pick e for hooking in a later stage. This is called *internal hooking*.

A root having an internal hooking may or may not detect it. For example, some un-nullified 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 will not hook, since its component is known to be promoted.

An (undetected) internal hooking will 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 the $O(\sqrt{\lg n})$ stages per phase is that after that many successful hookings a component has been promoted in any case. On the other hand, one internal hooking means that a component has already been 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 stage 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 stage. However, failure to hook is not harmful because the component has been promoted.

Each stage takes $O(\sqrt{\lg n})$ steps and there are at most $O(\sqrt{\lg n})$ stages, summing up to a total of $O(\lg n)$ steps per phase. We can prove that after $O(\sqrt{\lg n})$ stages all

components have been promoted. They may not have been contracted, though. In the final 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 of them 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 optimal sorting algorithms for both the CREW PRAM model [Col88, AKS83, Hag87] and the CREW PPM model [GK89] that run in $O(\lg n)$ time. Then, blocks of redundant edges are identified and nullified. This step takes $O(\lg n)$ time using m processors.
- The edge-list of each component is prepared by deleting all the null edges.

Each of these steps take $O(\lg n)$ parallel running time. So, the total running time of the algorithm is $O(\lg^{3/2} n)$ using at most $n + m$ 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 $v \in V$. We also form the edge-list of each component. Then, we perform the procedure `phase` $i_{max} = \lceil \sqrt{\lg n} \rceil$ times. Finally, we execute procedure `component identification`, after which the vertex set V has been divided into a number of equivalence classes

$$CC_k = \{v | v \in V \text{ and } p(v) = v_k\}, \; k \in Id$$

containing the connected components.

`Procedure phase`

1. { `Initialization` }

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

2. { `Component Promotion` }

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

COMMENT: This step tries to promote the components to the next phase. At the end of this step, each component is either promoted or done. Each stage takes time $O(\sqrt{\lg n})$.

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 stage 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 we execute the rule enabling step and then apply the CR rules.

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 [Wyl81] 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 [Col88, GK89] and use the results to form a sorted linked list.

COMMENT: The sorting places all multiple edges in blocks of consecutive identically named edges.

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

COMMENT: 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 $next(e) \leftarrow next(next(e))$

COMMENT: This step tries to remove blocks containing up to B^2 consecutive null edges. However, if the first edge, say e_v on some list $L(v)$ was null, it has not been removed. This is so because no pointer jumped e_v.

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 explicitly removes any null e_v from $first(L(v))$. The remaining edges satisfy Invariant 1.

7. { `Recomputation of the` *twin* `function.` }

for all useful edges (v, w) **in parallel do**
    let $(v', w') = next(v, w)$
    (v, w) writes its address to $prev(v', w')$
    $twin(v, w) \leftarrow prev(next(twin(w, v)))$

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 stage(i)`

1. { `The hooking step` }

for each active vertex v such that $root(v)$ **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 $(x, y) = (v, w)$ **and** $null(v, w)$
then $promoted(v)$
else $mate(v) \leftarrow w$
$p(v) \leftarrow w$
$not(root(v))$

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 $p(x) \neq v$, then $d_x^0 > B$. 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, r's component is promoted. Otherwise, v can hook to its mate vertex w.

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: First we 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 $O(\sqrt{\lg n})$ time.

3. { `Root recognition step` }

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

COMMENT: The new roots of the newly formed trees or stars identify themselves. If root v was also root in the previous stage, its component may still be unpromoted. But, if there was at least one stage $j : stage(v) < j < i$ during which v did not hook, then during stage 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 during stage $stage(v)$. Lemma 6 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** $stage(v) = i - 1$
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 stage 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.

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 apparently internal and so nullify themselves. The $root(r) = true$ 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 B^2 consecutive null edges are removed. Unpromoted stars now contain no null edges. This ensures that they will have a successful hooking at the next stage.

`Procedure component identification`

for $i \leftarrow 1$ **to** $\lceil \lg \sqrt{\lg n}\,\rceil$ **do**
  for each vertex $v \in V$ **in parallel do**
    $p(v) \leftarrow p(p(v))$

COMMENT: Let's assume that the input graph was composed of k connected components. The execution of the $\sqrt{\lg n}$ phases has created a pseudoforest $F = (V, p)$ composed of k rooted trees, one for each connected component. The depth of these trees is at most $\sqrt{\lg n}$. At the deepest level lie the vertices hooked in the first phase; at the next level lie the vertices hooked in the second phase, etc.

The execution of this procedure contracts each of these trees to rooted stars. After this step, vertices v and w are in the same connected component CC_k if and only if $p(v) = p(w) = v_k$.

8 Correctness and Time Bounds

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

Proof. Correctness follows from Lemma 13 below. The running time comes from the fact that there are $\lceil \sqrt{\lg n} \rceil$ phases, each taking $O(\lg n)$ parallel time. □

We will prove that in the beginning of each stage 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 stages.

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 stage j. Note that each of the M_j's is a pseudoforest.

We say that the root r of a rooted tree $P = (C, mate)$ *satisfies invariant* (2) if

$$\forall (v, w) \in E_i \text{ such that } v \in C \text{ and } not(null(v, w)) \Rightarrow (v, w) \in L(r) \qquad (2)$$

Lemma 6 *At the beginning of stage j each root r of a rooted tree $P = (C, mate) \in M_j$ satisfies (2).*

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

We assume that the lemma is true at the beginning of stage j, that is, every root of M_j satisfies (2). We will show that the invariant holds true at the beginning of stage $j + 1$.

During stage j the unpromoted roots of M_j hook to form larger components (step 1). Then, in step 3, some roots will recognize themselves as the roots of M_{j+1}. We have to prove that these roots satisfy invariant (2) at the beginning of stage $j+1$.

We will distinguish two cases:

(1) Root r was also a root in the beginning of stage 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

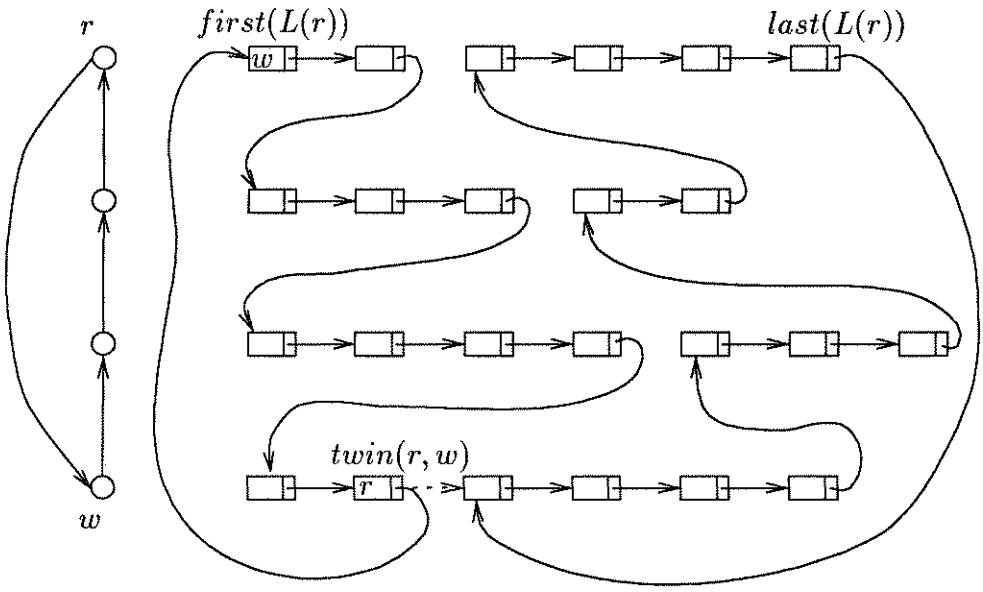

Figure 14: Assume that, at a later stage j, vertex r becomes the root of the pseudotree. Then, r can easily reverse its edge-plugging. In the figure, the dashed line denotes the pointer that must be changed.

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 stage $j+1$, root r satisfies (2)

(2) Root r was not a root in the beginning of stage j, therefore r was a part of a promoted component. We can see 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 stage j.

This operation is actually simple: r can reverse the steps of its own edge-plugging 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 recent hooking stage $j' < j$, (b) $twin(r, w) = (w, r)$, and (c) no edge shortcutted over (r, w), (w, r), or $last(L(r))$ during stages j' to j.

The $first(L(r))$ pointer is only altered by a hooking step, so (a) is true. Twin functions are only computed at the end of a phase, not during stages, so (b) is also true. Finally, as one can see by examining step 5 of procedure **stage**, these three edges were never nullified; so, (c) is also true. □

REMARK. Since only the edge-list of an already promoted component will ever be 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 $O(\lg n)$ 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 6 instead, was to provide the details for an implementation of Algorithm 1 (section 4.3).

Lemma 7 *If at the end of stage j some component is busy, it has been promoted.*

Proof. By definition, a component is busy if at the end of a stage it is still a pseudotree. Of course, such a component will not pick a mate in the beginning of the next stage because it has no root to do the operation. Procedure **stage** 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 stage 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 8 *Let C be a component which in stage 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 stage 1 because all components enter the first stage without internal edges. So, $j > 1$.

At the end of stage $j-1$ 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. 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 9 *If a component C fails to find a mate at some stage 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 stage j of phase i. We will 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 6, in the beginning of each stage 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 stage $j-1$ 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. 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 10 *Every active, non-promoted component at stage j will have a successful hooking at stage $j+1$.*

Proof. Let C be a component that is not promoted at the end of stage j. C is a rooted star because $|C| < B$. Also, by Lemma 6, its root holds all the edges that belonged to the edge-lists of its vertices and were not deleted in previous stages. 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 stage. □

Lemma 11 *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 easily prove by induction that, after each successful hooking at stage 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 12 *At the end of phase i each component is either promoted or not connected to any other components.*

Proof. Each phase is composed of $\lceil \lg B \rceil$ stages. In the beginning of a stage each component is either ready or busy. A busy component cannot pick a mate, but, according to Lemma 7, 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 6). So, the reason for which a ready component may not be able to find a mate (according to Lemma 9), is that the component is promoted or done. Otherwise the component will find a mate.

A hooking may either be successful or internal. An internal hooking, according to Lemma 8, can only happen to an already promoted component. So, we only have to follow components which have successful hookings for $\lceil \lg B \rceil$ stages. But these components (Lemma 11) have been promoted at the end of the last stage. □

Lemma 13 *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 $\lfloor B^i \rfloor$. (c) Invariant 1 is preserved. (d) There are no internal edges.*

Proof. (a) This is obviously true in the first phase, where components are composed of a single vertex, the root. During the stages, these components are hooked to form a pseudoforest. Then, at step 2 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 12.

(c) Again, this is true for the first phase. For the remaining phases, step 4 of procedure `phase` uses bucket sort to identify multiple edges and the step 5 removes them.

(d) Internal edges are nullified in step 3 and are removed in step 5 of the procedure `phase`. □

9 Conclusions

We have presented an algorithm which finds the connected components of an undirected graph for the CREW PRAM model of parallel computation. This algorithm works in $O(\lg^{3/2} n)$ time, and narrows the gap of the performance between several CREW and CRCW PRAM graph algorithms by a factor of $\lg^{1/2} n$. This result settles a question that remained unresolved for many years: a connectivity algorithm for this model with running time $o(\log^2 |V|)$ was a challenge that had thus far eluded researchers [KR90].

The techniques presented in this paper have been used to design new parallel algorithms for the minimum spanning tree problem [JM92]. Other algorithms having running times that depend on the connectivity algorithm include the Euler tour on graphs [AV84, AIS84], biconnectivity [TV85], the ear decomposition [MSV86, MR86] and its applications on 2-edge connectivity, triconnectivity, strong orientation, s-t

numbering etc. See the surveys by Karp and Ramachandran [KR90] and by Vishkin [Vis91] 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 [GK89]. The modification is to substitute the sorting routine we use at the end of each phase by the optimal sorting algorithm of Goodrich and Kosaraju [GK89]. 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 the generalization of Knuth's linking automaton.

Acknowledgements. The authors would like to thank Adonis Simvonis for his careful reading of an earlier version of this paper and his correction of a subtle point.

References

[AAK89] A. Aggarwal, R.J. Anderson, and M.-Y. Kao. Parallel depth-first search in general directed graphs. In *Proc. of the 21st Annual ACM Symposium on the Theory of Computing*, pages 297–308, May 15-17 1989.

[AIS84] 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.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[AS87] 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.

[AV84] M Atallah and U. Vishkin. Finding Euler tours in parallel. *Journal of Computer and System Sciences*, 29:330–337, 1984.

[Ber85] C. Berger. *Graphs.* North-Holland, Amsterdam, The Netherlands, 2nd edition, 1985.

[CDR86] 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.

[CLC82] 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.

[Col88] R. Cole. Parallel merge sort. *SIAM Journal of Computing*, 17(4):770–785, August 1988.

[CV86] 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.

[Gaz91] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, pages 1046–1067, 1991. Also: Proc. 27th FOCS 1986.

[GK89] 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.

[GPS87] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparce graphs. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages 315–324, 1987.

[Hag87] T. Hagerup. Towards optimal parallel bucket sorting. *Informaton and Computation*, 75:39–51, 1987.

[HCS79] 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.

[JM91] D.B. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} n)$ parallel time for the CREW PRAM. In *Proc. of 32nd IEEE Symposium on the Foundations of Computer Science (FOCS'91)*, October 1991.

[JM92] D.B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. In *Proc. of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'92)*, June 1992.

[KR90] R. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. *Handbook for Theoretical Computer Science*, 1:869–941, 1990.

[Met91] P. Metaxas. *Parallel Algorithms for Graph Problems.* PhD thesis, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH, July 1991.

[MR86] G.L. Miller and V. Ramachandran. Efficient parallel ear decomposition with applications. Technical report, MSRI, Berkeley, CA, 1986.

[MSV86] 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.

[NM82] 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.

[Rei85] J. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20:229–234, 1985.

[SV82] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[Tar72] R.E Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), 1972.

[TV85] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 14(4):862–874, 1985.

[Vis83] U. Vishkin. Implementation of simultaneous memory address access in models that forbid it. *Journal of Algorithms*, 4:45–50, 1983.

[Vis85] U. Vishkin. On efficient parallel strong orientation. *Information Processing Letters*, 20:235–240, June 1985.

[Vis91] U. Vishkin. Structural parallel algorithms. Technical Report UMIACS-TR-91-53, CS-TR-2652, University of Maryland, College Park, Maryland 20742, April 1991.

[Wil85] R.J. Wilson. *Introduction to Graph Theory.* Longman, Inc., New York, 3rd edition, 1985.

[Wyl81] J.C. Wyllie. *The Complexity of Parallel Computation.* PhD thesis, Computer Science Dept., Cornell University, Ithaca, NY, August 1981.