AN $O(\log n)$ PARALLEL CONNECTIVITY ALGORITHM

by

Yossi Shiloach* and Uzi Vishkin**

Technical Report #178
June 1980

* IBM - Israel Scientific Center, Technion City, Haifa, Israel
** Technion, Israel Institute of Technology, Computer Science Dept. Haifa, Israel.
AN $O(\log n)$ PARALLEL CONNECTIVITY ALGORITHM

by

Yossi Shiloach * and Uzi Vishkin**

June 1980

ABSTRACT

A parallel algorithm which uses $O(n+m)$ processors to find the connected components of an undirected graph with $n$ vertices and $m$ edges in time $O(\log n)$ is presented. We assume that the processors have access to a common memory. Simultaneously access to the same memory location is allowed for both read and write instructions.

---

* IBM-Israel Scientific Center, Technion City; Haifa, Israel

** Technion, Israel Institute of Technology; Computer Science Dept. Haifa, Israel.
1. INTRODUCTION

The problem of computing the connected components of a given un-directed graph \( G = (V,E) \) is considered. The model used is a synchronized parallel computation model in which all the processors have access to a common memory. Simultaneous reading from the same location is allowed as well as simultaneous writing. In the latter case one processor succeeds but we don't known which.

A connectivity algorithm in this model is presented. It has a depth of \( O(\log n) \) using \( n+2m \) processors where \( n = |V| \) and \( m = |E| \).

Our algorithm is considerably faster than those presented in [W79] and [HCS79] that achieve depth of \( O(\log^2 n) \) using \( O(n+m) \) and \( O(n^2/\log n) \) processors respectively. They do, however, use a somewhat weaker model in which simultaneous writing is not allowed.

2. AN \( O(\log n) \) CONNECTIVITY ALGORITHM

In this section, an \( O(\log n) \) parallel connectivity algorithm is described which uses \( 2m+n \) processors. If several processors attempt to write at the same location simultaneously, the model assumes that one of them succeeds.

In Section 2.1 the necessary basic definitions and operations are introduced. In Sections 2.2 and 2.3 an informal and exact description of the algorithm are given respectively.

2.1 Basics

During the whole algorithm each vertex \( v \) has a pointer field \( D(v) \) through which it points on another vertex or to itself. One can regard \( v \rightarrow D(v) \) as a directed edge in an auxiliary graph, so-called the
"pointers graph" (p.g. in short). The pointers graph keeps changing from one phase of the algorithm to the other. However, as will be shown in Section 3, it is always a forest of rooted trees plus self-loops which occur only in the roots. As the algorithm propagates, the number of trees decreases while each individual tree is increasing (or disappearing). This is caused by a "hooking" operation in which one tree is hooked on another. The trees are also subject to another transformation of collapsing towards the root. The whole algorithm consists of intermittent applications of these two primitive operations.

At the end of the algorithm, the vertices of each connected component form a rooted star in the pointers graph. Thus, a question of the form "Do \( v_i \) and \( v_j \) belong to the same connected component?" can be answered in constant time.

During the algorithm the following basic operations will be frequently used.

1. The short-cut operation, namely \( D(v) \leftarrow D(D(v)) \). Note that this operation never introduces directed cycles if there were not any.

2. Let \( P = (V, D) \) be a p.g. with \( k \geq 2 \) rooted trees \( T_1, \ldots, T_k \) and let \( r_i \) be the root of \( T_i \). The operation \( D(r_i) \leftarrow v \) where \( v \in T_j \) and \( j \neq i \) is called hooking of \( T_i \) on \( T_j \).

2.2 Informal Description of the Algorithm

In the informal description we try to give the reader a general idea on how the algorithm works. The exact description of the allocation of
processors to their jobs and the way in which they are scheduled is postponed to the next sub-section.

To simplify the description of the algorithm, assume that \( V = \{1, 2, ..., n\} \). The algorithm performs at most \( \lceil \log_{3/2} n \rceil + 2 \) iterations.

The notation \( D_s(i) = j \) means that vertex \( i \) points on vertex \( j \) after the \( s \)-th iteration. Initially \( D_0(i) = i, i = 1, ..., n \).

**Step 1:** Collapsing:
\[
D_s(i) = D_{s-1}(D_{s-1}(i)).
\]

**Step 2:** Hooking trees on smaller vertices of other trees: All the vertices that have pointed on a root at the end of the previous iteration check whether their neighbours are pointing on smaller vertices. If one finds such a neighbour \( j \), it tries to hook its tree on \( D_s(j) \). At this point simultaneous writing at the same location is used and one of them succeeds. More formally:

\[
\text{if } D_s(i) = D_{s-1}(i) \text{ then if } \exists j \text{ such that } (i, j) \in E \text{ and } D_s(j) < D_s(i) \text{ then } D_s(D_s(i)) = D_s(j).
\]

**Definitions:** A tree is called **stagnated** in the \( s \)-th iteration if it has not been changed in the first two steps of this iteration, i.e. it has not collapsed, no tree has been hooked on it and it has not been hooked on any other tree. A root of a stagnated tree is a **stagnated root**.

**Step 3:** Hooking stagnated trees:

All the vertices that point on a stagnated root, check whether their neighbours point on a vertex of another tree. If one finds such a vertex \( j \), it tries to hook its tree on \( D_s(j) \). One of them eventually succeeds. The exact implementation of this step is fully described in the next subsection.
2.3 Detailed Description of the Algorithm

Input form: Assuming that the vertices are represented by the numbers 1, ..., n we just keep the number n and a vector $\mathcal{E}$ of length $2m$ in which each edge $(i, j)$ appears twice, once as the ordered pair $<i, j>$ and once as the ordered pair $<j, i>$. The order of the ordered pairs in $\mathcal{E}$ is insignificant.

Example: If $G = \{1, 2, 3, 4\}$ then $\mathcal{E}$ might look like

$\langle 1, 2 \rangle, \langle 2, 4 \rangle, \langle 4, 3 \rangle, \langle 3, 1 \rangle, \langle 3, 2 \rangle, \langle 2, 1 \rangle, \langle 1, 3 \rangle, \langle 3, 4 \rangle, \langle 4, 2 \rangle, \langle 2, 3 \rangle$.

The processors are $P_1, \ldots, P_{n+2m}$.

We will also use an auxiliary vector $Q$ of length $n$. During the algorithm $Q$ will satisfy:

\[ Q(i) = s \quad \text{if after the second step of the } s\text{-th iteration (see informal description) there exists at least one vertex } j \text{ pointing on } i \text{ that did not point on } i \text{ after the } (s-1)\text{-st iteration.} \]

\[ Q(i) < s \quad \text{otherwise.} \]

The significant output of the algorithm is the vector

\[ (D_{s_0}(1), \ldots, D_{s_0}(n)), (s_0 = \lceil \log_3 n \rceil + 2) \]

Step 0: Initialization:

a. Allocation of processors to vertices and ordered pairs. If $i \leq n$ $P_i$ is allocated to the vertex $i$ and called "the processor of vertex $i".

If $i > n$ $P_i$ is allocated to the ordered pair $E(i) = <i_1, i_2>$.

Henceforth this processor is denoted as $P_{i_1 i_2}$ and called "processor of the ordered pair $<i_1, i_2>$".
\[
\text{b: if } i \leq n \\
\text{ then } D_0(i) = i, Q(i) = 0, s = 1, s' = 1.
\]

Comment: An instruction of this form means:
P_i checks whether it is a vertex processor or not. If it is such, it performs the instruction using its own index, e.g. if 7 \leq n then P_7 performs: \( D_0(7) = 7, Q(7) = 0, s = 1, s' = 1 \). Otherwise it remains idle.

While \( s' = s \) do

Step 1: if \( i \leq n \)

\[
\text{then } D_s(i) + D_{s-1}(D_{s-1}(i)) \quad \text{(collapsing)} \\
\text{if } D_s(i) \neq D_{s-1}(i) \\
\text{then } Q(D_s(i)) = s
\]

Step 2: if \( i > n \)

\[
\text{then if } D_s(i_1) = D_{s-1}(i_1) \\
\text{then if } D_s(i_2) < D_s(i_1) \\
\text{then } D_s(D_s(i_1)) + D_s(i_2) \\
Q(D_s(D_s(i_1))) = s
\]

Comment: If \( D(i_1) \) has not been changed in Step 1 (i.e. it has pointed on a root) then \( P_i = P_{i_1, i_2} \) checks whether \( i_2 \) is pointing on a smaller vertex. If so it tries to hook the root (which is \( D_s(i_1) \)) on \( D_s(i_2) \). Simultaneously, all the processors of the form \( P_{j_k} \) for which \( D_s(j) = D_s(i_1) \) and \( D_s(k) < D_s(i_1) \) try to update \( D_s(D_s(i_1)) \). According to our model, one of them eventually succeeds and we don't care which one.

Step 3: if \( i > n \)

\[
\text{then if } D_s(i_1) = D_s(D_s(i_1)) \text{ and } Q(D_s(i_1)) < s \\
\text{then if } D_s(i_1) \neq D_s(i_2) \\
\text{then } D_s(D_s(i_1)) + D_s(i_2)
\]

Comment: \( P_i = P_{i_1, i_2} \) checks first whether \( D_s(i_1) \) is a root. If so, it checks (using Q) whether it is a stagnated root and if so it tries to hook it on another tree. This is tried simultaneously by all the processors \( P_{j_k} \) such that \( D_s(j) = D_s(i_1) \).
Step 4: if \( i \leq n \)  
\[ D_s(i) + D_s(D_s(i)) \]

Step 5: \( s \leftarrow s + 1 \).  
if \( i \leq n \) and \( Q(i) = s \)  
then \( s' \leftarrow s' + 1 \). 
\text{od}

Comment: As soon as all the trees are stagnated, \( Q(i) < s \) for all \( i \), \( 1 \leq i \leq n \) and thus \( s' \) will not be incremented resulting \( s' < s \) (since \( s \) is incremented anyway). This causes the algorithm to terminate as soon as all the trees are stagnated. 

Synchronization is required before each line of the program.

2.4 Example

The following example shows a graph \( G \) and the resulting pointer graphs after each iteration of the algorithm. The numbers assigned to the pointers indicate in what step they were formed. Old pointers (from previous iterations) are not numbered.

First iteration
Third Iteration

Second Iteration
There is also a fourth iteration that does not affect the pointers graph and the only thing that happens is that $s$ is incremented while $s'$ is not, and this halts the algorithm.

3. VALIDITY AND DEPTH

In order to prove the validity and the logarithmic depth of the algorithm the following theorem is proved.

**Main Theorem**

a. The algorithm terminates after $s_0 \leq \lceil \log_{3/2} n \rceil + 2$ iterations.

b. $D_{s_0}(i) = D_{s_0}(j)$ iff $i$ and $j$ are in the same connected component.

The proof makes use of the following lemmas and definitions.

Let $R$ be a pointers graph.

**Lemma 3.1** If there exists a (directed) simple path from $u$ to $v$ in $R$ - it is unique.

**Proof.** Follows immediately from the fact that the out-degree of each vertex in $R$ is one.

**Definitions**

1. If there exists a path from $u$ to $v$ in $R$, then $d_R(u,v)$ denotes the number of edges (the distance) in the unique simple path from $u$ to $v$.

2. The **height** of $v$ in $R$, $h_R(v)$, is $\max\{d_R(u,v) | v \text{ is reachable from } u \text{ in } R\}$.

3. The **cardinality** of $v$ in $R$, $c_R(v)$, is $\{|v| | v \text{ is reachable from } u \text{ in } R\}$.

4. A vertex $v$ is a **leaf** in $R$ if its in-degree is zero.

5. Let $R_{s,k}(G)$ denote the pointers graph that the algorithm assigns to $G$ after the $k$-th step of its $s$-th iteration, $(k=1, \ldots, 5)$. We shall use
the abbreviated notation $R_{s,k}$ whenever no confusion can arise regarding $G$'s identity.

**Lemma 3.2** If a vertex $i$ is a leaf in $R_{s_0,k_0}$ for some $s_0, k_0$ then it remains a leaf in $R_{s,k}$ for all $(s,k)$ that follow $(s_0,k_0)$ in the lexicographic order.

**Proof.** Following the algorithm, it is easy to verify that if we "hook" a vertex $i$ on vertex $j$ (i.e. assign $D(i) = j$) then $j$ was not a leaf at the end of the preceding step. □

**Lemma 3.3** If a vertex $i$ is a stagnated root after the second step of the $s$-th iteration (i.e. $D_s(i) = i$ and $Q(i) < s$) then $h_{R_{s_0}}(i) \leq 1$ and $h_{R_{s_2}}(i) \leq 1$.

**Proof.** If $h_{R_{s_1}}(i) > 2$ then Step 1 adds new "sons" to $i$ and $i$ cannot be stagnated. Thus $h_{R_{s_1}}(i) \leq 1$. If $h_{R_{s_2}}(i) > 1$, it means that some vertex was hooked on it in Step 2. The stagnation of $i$ implies that no vertex was hooked on $i$ itself. Moreover, since $h_{R_{s_1}}(i) \leq 1$, all its sons were leaves and Step 2 never hooks vertices on leaves. Thus $h_{R_{s_2}}(i) \leq 1$. The only way to increment the height of $i$ in Step 3 is to hook another (stagnated) vertex on it (since $h_{R_{s_2}}(i) \leq 1$ all its sons are leaves and Step 3 never hooks vertices on leaves).

The next lemma shows that this never happens.

**Lemma 3.4.** If $D_s(i)$ and $D_s(j)$ are different stagnated roots after Step 2 then $i$ and $j$ are not connected by an edge in $G$.

**Proof.** Assume that $D_s(i) < D_s(j)$. If $(i,j) \in E$ then the processor $P_{ji}$ tries to hook $D_s(j)$ on $D_s(i)$ in Step 2, resulting non-stagnation of $D_s(i)$. □
Lemma 3.5 If \( i < D_s(i) \) after Steps 1, 2, 4, then \( i \) is a leaf in \( R_{s,1} \), \( R_{s,2} \) and \( R_{s,4} \) respectively.

Proof. By induction on \( s \). The lemma holds vacuously initially. If the lemma holds after Step 1 of the \( s \)-th iteration, it is also true after Step 2 since in this step we never hook a vertex on a bigger one or a leaf. In Step 3 we may hook a vertex \( i \) on a bigger one. However, in this case \( i \) is a stagnated root and therefore \( h_{R_{s,3}}(i) \leq 1 \), (Lemma 3.3). Hence after the fourth step of the \( s \)-th iteration, \( i \) becomes a leaf. □

Lemma 3.6 \( R_{s,k} \) is a forest of rooted trees with self-loops in their roots, for all \( s \) and \( k \).

Proof. By induction on \( s \). The lemma is obviously true after initialization. Steps 1 and 4 never introduce directed cycles if there weren't any. Assume to the contrary that a simple directe cycle is formed in Step 2. Let \( j \) be the biggest vertex in this cycle and let \( i \) be the vertex in the cycle that points on \( j \). By Lemma 3.5 \( i \) is a leaf - a contradiction. Cycles cannot be formed in Step 3 since by Lemma 3.4 stagnated trees are hooked only on non-stagnated trees which are not further hooked in this step. □

In the following two lemmas, we show that if the algorithm terminates it yields the right output (Part b. of the Main Theorem).

Lemma 3.7 Let \( v \) be a stagnated root after Step 2 of the \( s \)-th iteration, and assume that \( v \) is still a root after the \( s \)-th iteration, then all the vertices in \( v \)'s connected component are pointing on \( v \).

Proof. The stagnation of \( v \) after Step 2 implies, by Lemma 3.3, that \( h_{R_{s,2}}(v) \leq 1 \). Assume to the contrary that there exists a vertex \( u \) which is connected to \( v \) and \( D_s(v) \neq v \). Thus, there exists a path from \( u \) to \( v \). Look at the vertex \( w \) on such a path which is the closest to \( v \) among the vertices on the path which satisfy \( D_s(w) \neq v \). But in Step 3, \( D_s(v) \) should
have been assigned with $D_s(w)$. A contradiction to its being a root at the end of the iteration. 

The proof of the first part of the main theorem requires the following lemmas.

Lemma 3.8: If $D_s(v) = u$ for some $s$, then the vertices $u$ and $v$ belong to the same connected component of $G$.

Proof. The proof can be easily obtained by induction on $s$. 

Lemma 3.9: If a tree $T$ in the pointers graph has not been changed during an entire iteration then it remains unchanged until the end.

Proof. Assume that $T$ has not been changed during the $s$-th iteration. Thus, its root has been stagnated after Step 2 and the only reason that it was not hooked on another tree is that all the vertices in its connected component had pointed on it. Obviously, such a tree is not going to be changed anymore. 

Lemma 3.10: Let $v_1, \ldots, v_r$ be roots and let $h_1, \ldots, h_r$ be their heights respectively. If the trees that are rooted at $v_1, \ldots, v_r$ are merged into one tree as a result of hooking in Steps 2 or 3 then its height is at most $h_1 + \ldots + h_r$.

Proof. The proof follows easily from the fact that we never hook a root on the leaf of another tree.

Definition: A root $v$ satisfies the property $w(s)$ at a given time if $c_R(v) \geq (\frac{3}{2})^{s-1} h_R(v)$ at that time.

Lemma 3.11: If a tree is a star after Step 3 of a given iteration then it has also been so after the first step of that iteration.

Proof. Follows from the fact that hooking never results a star.
Lemma 3.12: If \( v \) is a root at the end of the \( s \)-th iteration and was a root of a different tree at the end of Iteration \( s-1 \), then it satisfies \( w(s) \) at the end of the \( s \)-th iteration.

Proof. By induction on \( s \).

For \( s = 1 \), the lemma states that after the first iteration \( c_R(v) \geq h_R(v) \) which is obvious.

Assume that \( v \)'s tree has been changed in the \( s \)-th iteration. By Lemma 3.9 \( v \)'s tree has also been changed during the \( (s-1) \)-st iteration and thus the inductive hypothesis can be applied, yielding that \( v \) has satisfied \( w(s-1) \) at the end of Iteration \( s-1 \).

Let us consider \( v \)'s state before Step 4 of the \( s \)-th iteration and distinguish between two cases.

Case 1: It has been a root of a star at that time.

By Lemma 3.11 \( v \) was the root of the same star after Step 1. Since \( v \)'s tree has been changed in the \( s \)-th iteration, it must have been collapsed in Step 1 and therefore its height decreased by a factor of at least 3/2 in this step. Since \( v \)'s tree has satisfied \( w(s-1) \) before Step 1, it has satisfied \( w(s) \) after it. Since it has not been changed in the subsequent steps, \( v \) satisfies \( w(s) \) at the end of the \( s \)-th iteration.

Case 2: \( v \) was a root of a non-star tree before Step 4. The inductive hypothesis and Lemma 3.10 imply that \( v \) satisfies \( w(s-1) \) after Step 3 of the \( s \)-th iteration. Since \( v \) is a root of a non-star tree after this step, its tree collapses in Step 4 yielding a shrinking factor of at least 3/2 in its height. Thus \( v \) satisfies \( w(s) \) after Step 4 and at the end of the \( s \)-th iteration. \( \square \)
Proof of the Main Theorem - Part a

By Lemma 3.12, a root of a tree that has been changed during the s-th iteration satisfies \( w(s) \) at its end. Since any tree can have at most \( n \) vertices and height of at least one, it means that if something has been changed in the s-th iteration then \( s \leq \lceil \log_3 n \rceil + 1 \).

Since there is only one iteration in which nothing is changed - the algorithm iterates at most \( \lceil \log_3 n \rceil + 2 \) times. \( \square \)

REFERENCES
