An $O(\log n)$ Parallel Connectivity Algorithm

YOSSI SHILOACH

IBM Israel Scientific Center, Haifa, Israel

AND

UZI VISHKIN

Computer Science Department, Technion–Israel Institute of Technology, Haifa, Israel

Received November 5, 1980; revised July 8, 1981

1. INTRODUCTION

The problem of computing the connected components of a given undirected 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 do not know which. This model is a member in a whole family of models for synchronized parallel computation. This family and the relations among its members are discussed in [2, 3].

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 [1, 4], which 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.

Although their model is weaker (simultaneous writing is not allowed), there is no apparent way to implement their algorithm in our model in depth of $O(\log n)$. The improvement of the depth is mainly due to the algorithm itself. On the other hand it is interesting to see how its extra power is fully exploited. We conjecture (and so does Willie [4]) that this performance cannot be achieved in the weaker model. Moreover, we conjecture that the barrier of $O(\log n)$ cannot be surpassed by any polynomial number of processors.

0196-6774/82/010057–11$02.00/0

2. An $O(\log n)$ Connectivity Algorithm

In this section, an $O(\log n)$ parallel connectivity algorithm is described which uses $2m + n$ processors.

In Section 2.1 the necessary basic definitions and operations are introduced. In Sections 2.2 and 2.3 an informal and an exact descriptions of the algorithm, respectively, are given. A detailed example is given in Section 2.4. The validity proof and depth analysis are given in Section 3.

2.1. *Basics*

Definitions. (1) A *rooted tree* is a directed graph satisfying:

(a) its underlying undirected graph is a tree;

(b) it has a vertex r called *root* such that there exists a directed path from each vertex to r.

(2) A *rooted star* is a rooted tree in which each vertex is connected directly to the root.

During the whole algorithm each vertex v has a pointer field $D(v)$ through which it points to another vertex or to itself. One can regard $v \rightarrow D(v)$ as a directed edge in an auxiliary graph, called the "*pointer graph*" (p.g. for short). The pointer graph keeps changing in the course of the algorithm. However, as will be shown in Section 3, it is always a forest of rooted trees plus self-loops that occur only in the roots. As the algorithm proceeds, the number of trees decreases while each individual tree expands (or disappears). This is caused by a "hooking" operation in which one tree is hooked onto another. The trees are also subject to a shortcut operation that decreases their height. The whole algorithm consists of intermixed applications of these two primitive operations.

At the end of the algorithm the vertices of each connected component form a rooted star in the pointer 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 used frequently.

(1) The *shortcut operation*, namely, $D(v) \leftarrow D(D(v))$. Note that this operation never introduces directed cycles if there were none.

(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

$$D(r_i) \leftarrow v, \qquad \text{where } v \in T_j \text{ and } j \neq i,$$

is called *hooking* of T_i onto T_j.

2.2. *Informal Description of the Algorithm*

In the informal description we try to give the reader a general idea of 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 subsection.

To simplify the description of the algorithm, assume that $V = \{1, 2, \ldots, n\}$. The algorithm performs at most $\lfloor \log_{3/2} n \rfloor + 2$ iterations. The notation $D_s(i) = j$ means that vertex i points to vertex j after the sth iteration. Initially, $D_0(i) = i$, for $i = 1, \ldots, n$.

Description of the sth Iteration

Step 1. Shortcutting:

$$D_s(i) \leftarrow D_{s-1}(D_{s-1}(i)).$$

Step 2. Hooking trees onto smaller vertices of other trees: All the vertices that pointed to a root at the end of the previous iteration check whether their neighbors are pointing to smaller vertices. If one finds such a neighbor j, it tries to hook its tree onto $D_s(j)$. At this point simultaneous writing at the same location is used.

More formally:

$$\text{if } D_s(i) = D_{s-1}(i)$$

then if $\exists j$ such that $(i, j) \in E$ and $D_s(j) < D_s(i)$ then $D_s(D_s(i)) \leftarrow D_s(j)$.

DEFINITIONS. A tree is called *stagnant in the sth iteration* if it has not been changed in the first two steps of this iteration; i.e., it has not been subject to any shortcut operation, no tree has been hooked onto it, and it has not been hooked onto any other tree. A root of a stagnant tree is a *stagnant root.*

Step 3. Hooking stagnant trees: All the vertices that point to a stagnant root check whether their neighbors point to a vertex of another tree. If one finds such a vertex j, it tries to hook its tree onto $D_s(j)$. One of them succeeds. The exact implementation of this step is fully described in the next subsection.

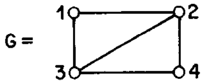
Step 4. Second shortcutting:

$$D_s(i) \leftarrow D_s(D_s(i)).$$

2.3. *Detailed Description of the Algorithm*

Input form: The vertices are represented by the numbers $1, \dots, n$. The edges are represented by a vector **E** of length $2m$ in which each edge (i, j) appears twice, once as the ordered pair $\langle i, j\rangle$ and once as the ordered pair $\langle j, i\rangle$. The order of the ordered pairs in **E** is immaterial.

EXAMPLE. if



then **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 $P_1, \dots, P_{n+2m}$. We will also use an auxiliary vector Q of length n. During the algorithm Q will satisfy:

$Q(i) = s$ if after the second step of the sth iteration (see informal description) there exists at least one vertex j pointing to i that did not point to i after the $(s-1)$st iteration.

$Q(i) < s$ otherwise.

The significant output of the algorithm is the vector

$$(D_{s_0}(1), \dots, D_{s_0}(n)),$$

where s_0 is the index of the last iteration.

Step 0. *Initialization*: (a) Allocation of processors to vertices and ordered pairs. If $i \le n$, processor P_i is allocated to vertex i and called "the processor of vertex i." If $i > n$, processor P_i is allocated to the ordered pair $\mathbf{E}(i-n) = \langle i_1, i_2\rangle$. Henceforth this processor is denoted as P_{i_1, i_2} and called "processor of the ordered pair $\langle i_1, i_2\rangle$."

```
(b) If $i \le n$
    then $D_0(i) \leftarrow i$, $Q(i) \leftarrow 0$, $s \leftarrow 1$, $s' \leftarrow 1$.
```

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. For example, if $7 \le n$ then P_7 performs: $D_0(7) \leftarrow 7$, $Q(7) \leftarrow 0$, $s \leftarrow 1$, $s' \leftarrow 1$. Otherwise, it remains idle.

```
While $s' = s$ do
    Step 1: if $i \le n$
            then $D_s(i) \leftarrow D_{s-1}(D_{s-1}(i))$   (shortcutting)
                 if $D_s(i) \neq D_{s-1}(i)$
                 then $Q(D_s(i)) \leftarrow s$           (updating $Q$)
```

```
Step 2: if $i > n$
        then if $D_s(i_1) = D_{s-1}(i_1)$
             then if $D_s(i_2) < D_s(i_1)$
                  then $D_s(D_s(i_1)) \leftarrow D_s(i_2)$
                       $Q(D_s(i_2)) \leftarrow s$.
```

Comment. If $D_s(i_1)$ has not been changed in Step 1 (i.e., it has pointed to a root) then $P_i = P_{i_1, i_2}$ checks whether i_2 is pointing to a smaller vertex. If so it tries to hook the root (which is $D_s(i_1)$) onto $D_s(i_2)$. 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 succeeds and we do not care which one.

```
Step 3: if $i > n$
        then if $D_s(i_1) = D_s(D_s(i_1))$ and $Q(D_s(i_1)) < s$
             then if $D_s(i_1) \neq D_s(i_2)$
                  then $D_s(D_s(i_1)) \leftarrow D_s(i_2)$
```

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 stagnant root and if so it tries to hook it onto another tree. This is tried simultaneously by all the processors $P_{j,k}$ such that $D_s(j) = D_s(i_1) \neq D_s(k)$.

```
Step 4: if $i \leq n$
        then $D_s(i) \leftarrow D_s(D_s(i))$.

Step 5: if $i \leq n$ and $Q(i) = s$
        then $s' \leftarrow s' + 1$
        $s \leftarrow s + 1$
od
```

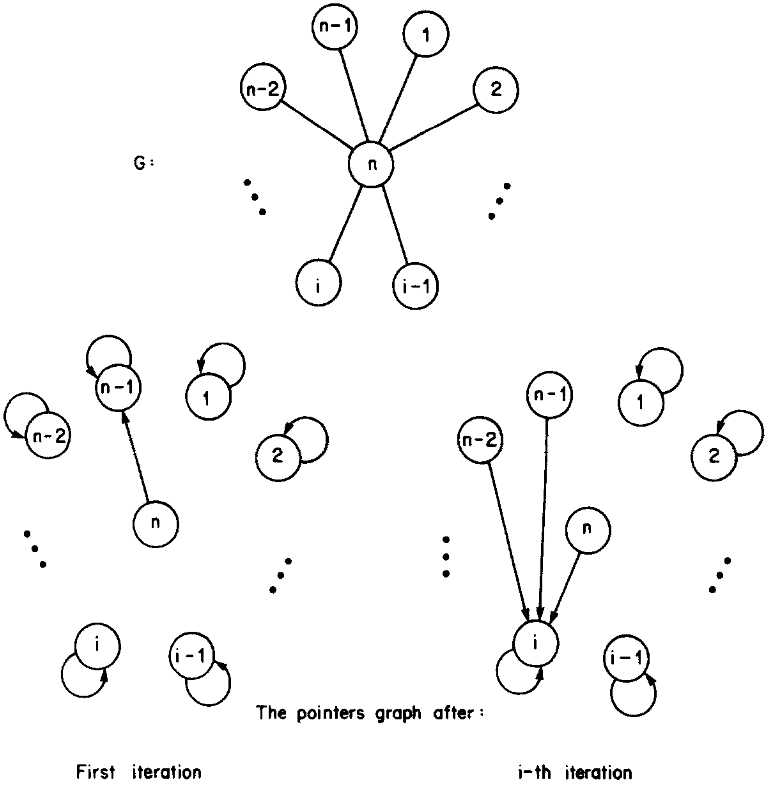
Comment. As soon as all the trees are stagnant, $Q(i) < s$ for all i, $1 \leq i \leq n$, and thus s' will not be incremented resulting $s' < s$ (since s is incremented anyway). This causes the algorithm to terminate as soon as all the trees are stagnant.

Synchronization is required before each line of the program; i.e., all the processors start together the execution of each line.

Actually, Step 4 can be excluded. However, it enables us to obtain a neater depth analysis and its exclusion may double the number of iterations.

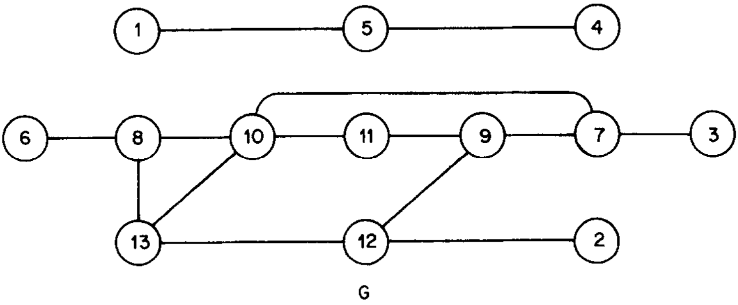
The example given below shows that Step 3 cannot be excluded. Its exclusion may lead to an algorithm of $n - 1$ iterations.

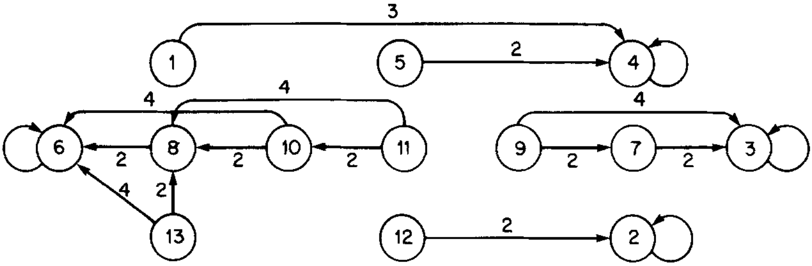
EXAMPLE.



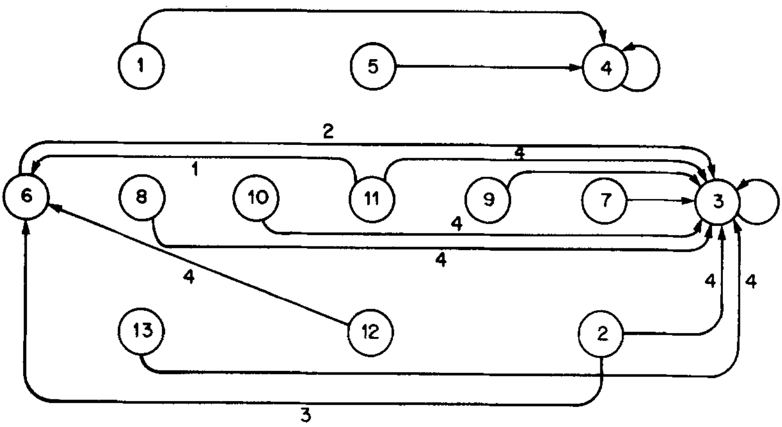
2.4. *Example*

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.

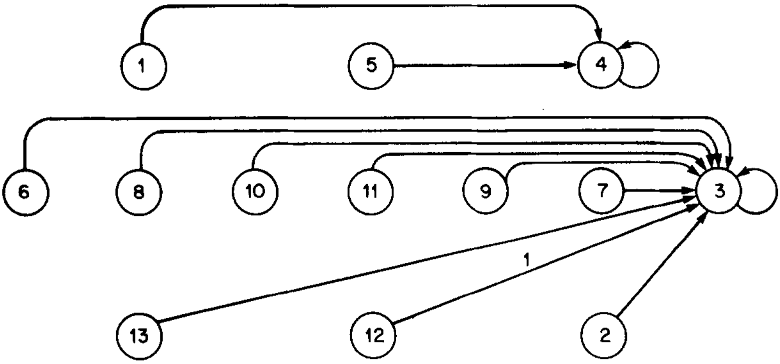




First Iteration



Second Iteration



Third Iteration

There is also a fourth iteration that does not affect the pointer graph. 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* $s_0 \leq \lfloor \log_{3/2} n \rfloor + 2$ *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 pointer 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 $\max\{d_R(u, v) \mid v$ is reachable from u in $R\}$ and the *cardinality* of v in R, $c_R(v)$, is $\{u \mid v$ is reachable from u in $R\}$.

(3) A vertex v is a *leaf* in R if its in-degree is zero.

(4) Let $R_{s,k}(G)$ denote the pointer graph that the algorithm assigns to G after the kth step of its sth 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_*, k_*} *for some* s_*, k_* *then it remains a leaf in* $R_{s,k}$ *for all* (s, k) *that follow* (s_*, k_*) *in the lexicographic order.*

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

Lemma 3.3. *If* $D_s(i)$ *and* $D_s(j)$ *are different stagnant 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 $P_{j,i}$ tries to hook $D_s(j)$ onto $D_s(i)$ in Step 2, resulting in nonstagnation of $D_s(i)$. □

LEMMA 3.4. *If a vertex i is a stagnant root after the second step of the sth iteration (i.e., $D_s(i) = i$ and $Q(i) < s$) then $h_{R_{s,2}}(i) \leq 1$ and $h_{R_{s,3}}(i) \leq 1$.*

Proof. If $h_{R_{s,1}}(i) \geq 2$ then Step 1 adds new "sons" to i and i cannot be stagnant. Thus, $h_{R_{s,1}}(i) \leq 1$. If $h_{R_{s,2}}(i) > 1$, it means that some vertex was hooked onto it in Step 2. The stagnation of i implies that no vertex was hooked onto i itself. Moreover, since $h_{R_{s,1}}(i) \leq 1$, all its sons were leaves and Step 2 never hooks vertices onto 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 (stagnant) vertex onto it. (Since $h_{R_{s,2}}(i) \leq 1$ all its sons are leaves and Step 3 never hooks vertices onto leaves.) By the previous lemma, this never happens.

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 sth 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 stagnant root and therefore $h_{R_{s,3}}(i) \leq 1$ (Lemma 3.4). Hence after the fourth step on the sth 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 were none. Assume to the contrary that a simple directed 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 3.3 stagnant trees are hooked only on nonstagnant 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 stagnant root after Step 2 of the sth iteration, and assume that v is still a root after the sth iteration. Then all the vertices in v's connected component are pointing to v.*

Proof. The stagnation of v after Step 2 implies, by Lemma 3.4, that $h_{R_{s,2}}(v) \leq 1$. Assume to the contrary that there exists a vertex u which is connected to v and $D_s(u) \neq v$. 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 $D_s(w)$ or other vertex $\neq v$. Thus v cannot be a root at the end of the iteration—a contradiction. □

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. □

The proof of the first part of the Main Theorem requires the following lemmas.

LEMMA 3.9. *If a tree T in the pointer 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 sth iteration. Thus, its root has been stagnant after Step 2 and the only reason that it was not hooked onto another tree is that all the vertices in its connected component had pointed to it. Obviously, such a tree is not going to be changed any more. □

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 Step 2 or Step 3 then its height is at most $h_1 + \cdots + h_r$.*

Proof. The proof follows easily from the fact that we never hook a root onto 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 iteration $s \geq 2$ then it has not been changed since the end of the first step of that iteration.*

Proof. Follows from the fact that hooking never results a star from the second iteration and on. □

LEMMA 3.12. *If v is the root of a tree that changed during the sth iteration, then it satisfies $w(s)$ at the end of the sth 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 sth 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 to show 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 sth iteration and distinguish between two cases.

Case 1. It was 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 sth iteration, it must have been shortcut in Step 1 and therefore its height decreased by a factor of at least $\frac{3}{2}$ 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 sth iteration.

Case 2. v was a root of a nonstar tree before Step 4. The inductive hypothesis and Lemma 3.10 imply that v satisfies $w(s-1)$ after Step 3 of the sth iteration. Since v is a root of a nonstar tree after this step, its tree is shortcut in Step 4 yielding a shrinking factor of at least $\frac{3}{2}$ in its height. Thus v satisfies $w(s)$ after Step 4 and at the end of the sth iteration. □

Proof of the Main Theorem—Part (a). By Lemma 3.12, a root of a tree that has been changed during the sth iteration satisfies $w(s)$ at its end. Since any such tree has at most n vertices and height at least one, it means that if something has been changed in the sth iteration then $s \leq \lfloor \log_{3/2} n \rfloor + 1$.

Since there is only one iteration in which nothing is changed, the algorithm iterates at most $\lfloor \log_{3/2} n \rfloor + 2$ times. □

ACKNOWLEDGMENTS

The authors wish to thank Professor S. Even and the devoted referee for their valuable comments.

REFERENCES

1. D. S. HIRSCHBERG, A. K. CHANDRA, and D. V. SARWATE, Computing connected components on parallel computers, *Comm. ACM* **22**, No. 8 (1979), 461–464.
2. U. VISHKIN, "Synchronized Parallel Computation," D.Sc. thesis, Computer Science Department, Technion, Haifa, Israel, February 1981. [In Hebrew].
3. U. VISHKIN, Implementing simultaneous memory access in nonsimultaneous memory access models, to appear.
4. J. C. WILLIE, "The Complexity of Parallel Computations," Ph.D. thesis, Cornell University, Ithaca, N.Y., August 1979.