Discrete Applied Mathematics 9 (1984) 197-207
North-Holland

AN OPTIMAL PARALLEL CONNECTIVITY ALGORITHM

Uzi VISHKIN*
Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012, USA

Received 20 September 1982
Revised 9 August 1983

A synchronized parallel algorithm of depth $O(n^2/p)$ for p $(\leq n^2/\log^2 n)$ processors is given for the problem of computing connected components of an undirected graph. The speed-up of this algorithm is optimal in the sense that the depth of the algorithm is of the order of the running time of the fastest known sequential algorithm over the number of processors used.

1. Introduction

This paper presents a synchronized parallel algorithm for computing connected components of an undirected graph. The algorithm presented has the following favorable property: its depth (running time) is of the order of the fastest known sequential algorithm over the number of processors used. Later in the paper when we say that a parallel algorithm is 'optimal' or has 'optimal speed-up', we refer to this property. A model of computation is used in which all the processors have access to a common memory. Simultaneous reading of several processors from the same memory location is allowed; simultaneous writing in the same memory location is also allowed, provided all processors seek to write the same value. For a detailed description of this model and its basic definitions, see [10].

One can say that models of computation, in which simultaneous writing by several processors in the same memory location is allowed, are fairly acceptable. The papers [1], [6], [9], [11] and [14] allow several processors to seek simultaneous writing in the same location, even of different values (which is an obvious extension of our assumption). In such a case, some of them assume that one of the processors succeeds to write in one time unit, but we do not know in advance which, and others assume that the processor which has the smallest serial number succeeds in one time unit. This family of models of parallel computation is sometimes called concurrent-read concurrent-write parallel random access machines (CRCW PRAM in short). Moreover, [5] and [15] present algorithms for implementing every algorithm in these CRCW PRAM models into models with no simultaneous access conflicts (some-

*Present address: Dept. of Comput. Sci., School of Math. Sci., Tel Aviv Univ., Tel Aviv 69978, Israel. The author visited IBM T.J. Watson Res. Center from Sept. 1981 to Aug. 1982. Part of this research was performed during this visit.

times called exclusive-read exclusive-write (EREW) PRAM) in a way which is not less efficient than any known algorithm for implementing algorithms in models that allow simultaneous access of several processors to the same common memory location for read, but not write, purposes (CREW PRAM) into EREW PRAM. This phenomenon was proven in [16] to hold for a large family of such implementation algorithms provided that they satisfy a few reasonable assumptions. We only sketch the spirit of this proof avoiding its lengthy details. Suppose a simulation of the CREW PRAM by the EREW PRAM is given. Given a time unit of the CREW PRAM we have to simulate it by the EREW PRAM. Instead, a corresponding (legal) time unit of the CREW PRAM is simulated and all the intermediate computations and, in particular, all copying operations of data during the simulation are recorded. These intermediate computations are then used to form a simulation of the original time unit of the CRCW PRAM into the EREW PRAM. In particular, 'fan-outs' of information (which correspond to dissemation of a common memory address to processors that seek to read it) are transferred into 'fan-ins' of information (which correspond to several processors seeking to write into a common memory address). This suggests some justification for comparing results in our model with the results of Chin et al., Hirschberg et al., and Wyllie, although their results were achieved in the slightly weaker CREW PRAM. For a proof that the CREW PRAM is weaker than the CRCW PRAM see [3]. Recently, [13] supported the CRCW PRAM models by establishing a relationship between them and combinational logic circuits that contain AND's, OR's and NOT's, with no bounds on the fan-in of AND-gates and OR-gates. Parallel time and the number of processors for the parallel computation model correspond respectively to depth and size of circuits, where the time-depth correspondence is to within a constant factor and the processor-size correspondence is within a polynomial.

The problem of obtaining parallel algorithms for computing connected components of an undirected graph received considerable attention in literature. An algorithm of depth $O(\log^2 n)$ for n^2 processors was suggested in [8]. (Note that this is the same as saying, depth of $O((n^2 \log^2 n)/p)$ for p $(\leq n^2)$ processors.) An improved version of the same algorithm is given in [7], with the same depth achieved by only $n^2/\log n$ processors. Still another version of this algorithm is suggested in [17] with depth $O(\log^2 n)$, this time by $n+m$ processors; n being the number of vertices and m that of edges. It remained to be asked whether optimization of the 'speed up' is feasible. In [4], this question is answered in the affirmative for dense graphs with $m \approx n^2$. The algorithm which is another version of the above algorithm is of depth $O(\log^2 n)$ for $n^2/\log^2 n$ processors. These results were achieved in the CREW PRAM.

[11] introduces a new algorithm that achieves a depth of $O(\log n)$ by $n+m$ processors. The model of computation allows simultaneous writing by several processors in the same memory location; in such a case one of these processors succeeds, but we do not know which.

All the works mentioned so far assume models of computation that take all the overheads into account, including allocation of processors to their jobs. Actually, the main contribution of [1] in the merging algorithm, [4] in the connectivity algorithm and [10] in the algorithm for finding the maximum among n elements is in solving the problem of allocating processors to their jobs in previously known algorithms.

The algorithm presented here is of depth $O(\log^2 n)$ for $n^2/\log^2 n$ processors as in [4].

The algorithm takes advantage of the stronger model of computation due to considerable changes in the algorithm of [7]. The changes are: Closer adherence to the basic notion of the algorithm of shrinking the original graph with gradual reduction of the number of vertices, a more careful assignment of processors to their jobs, and data-structure that may be useful in other parallel algorithms.

The general outlook at the algorithm as repeatedly shrinking the original graph and collecting the relevant information after the shrinking was finished (that is known in the theory of designing sequential algorithms), together with the technique of assigning processors to their jobs in the shrinking graphs (that enables smooth shrinking), seems a promising method in the theory of designing efficient parallel algorithms.

Our solution for the important tasks of assigning processors to their jobs is simpler than in [4]. This is due to two things.

(1) It uses a technique that makes the life of the algorithm designer much easier. This technique was already shown to be useful for several other parallel algorithms. Specifically, see [2] and [12] for parallel algorithms where the algorithm is given first without taking care of the full assignment of processors to their jobs. This problem is later resolved by applying Brent's [2] theorem. This simple technique seems to be basic for design of parallel algorithms.

(2) The advantage of allowing concurrent write (in Step 9 of the algorithm below) frees us from the apparent need to apply sorting in each application of the loop of Steps 2 through 9, as in [4]. Besides the simplicity point this repeated application of sorting increases the constant multiplicatives in the time complexity estimates of their algorithm. For more on the application of sorting in their algorithm see the last paragraph of this paper.

On the other hand, one may favor the [4] algorithm as it does not use concurrent write. My answer to that is to refer the reader to the arguments presented above in favor of the most permissive model of computation that does not increase the fine for implementation into restrictive models of computation.

2. The algorithm

In Subsection 2.1, a useful data-structure is presented. Subsection 2.2 is devoted to a detailed description of the algorithm. The efficient implementation of the algorithm is described in Section 3.

2.1. Data-structure

The following simple data-structure, which enables efficient assignment of processors to their jobs, is reminiscent to some extent of the one given in [12]. It is used here, however, more effectively.

Given n numbers $a_1 a_2, \dots, a_n$ we associate with them a complete binary tree $T(a_1, a_2, \dots, a_n)$, a so-called *partial star tree,* or P*-tree for short, the asterisk standing for any associative binary operation. $T(a_1, a_2, \dots, a_n)$ contains $2^{\lceil \log n \rceil}$ leaves (all logarithms in this paper are to the base of 2). Every node in the tree is represented by a combination $[h, j]$, h being its height in the tree and j its serial number among the other nodes at the same height (Fig. 1).

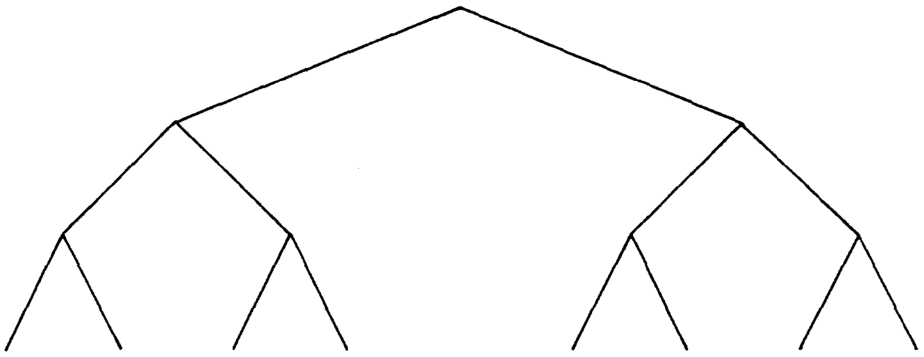
Fig. 1.

A neutral element of the $*$ operation is denoted by 0: for example, the neutral element for the + operation is the number zero; any number greater than the possible values of a_i may serve as a neutral element for the Min operation (the minimum between two numbers).

Every node $[h, j]$ in the tree is associated with two numbers, $A(h, j)$ and $B(h, j)$. The A numbers satisfy

$$(1) \qquad A(0, j) = \begin{cases} a_j & \text{for } j \le n, \\ 0 & \text{for } j > n. \end{cases}$$

$$(2) \qquad A(h, j) = A(h-1, 2j-1) * A(h-1, 2j) \quad \text{for } 0 < h \le \lceil \log n \rceil.$$

Another way to put the second equation is: each internal node (h, j) is the root of a complete subtree; $A(h, j)$ is the 'sum' of the $a_{j_1}, a_{j_1+1}, \dots, a_{j_2}$ attached to its leaves (we refer to the $*$ operation as summation). Therefore, having $(p=)$ $2^{\lceil \log n \rceil}$ processors numbered from 1 to $2^{\lceil \log n \rceil}$, the following straightforward procedure computes the A's:

Procedure A

```
        Processor i:
            1. if $i \le n$
                  then $A(0,i) \leftarrow a_i$
                  else $A(0,i) \leftarrow 0$
            2. $h \leftarrow 1$
            3. while $h \le \lceil \log n \rceil$ and $2^h i \le 2^{\lceil \log n \rceil}$ do
                        $A(h,i) \leftarrow A(h-1,2i-1) * A(h-1,2i)$
                        $h \leftarrow h+1$
                  od
```

We would like the B numbers to satisfy:

$$B(0,j) = A(0,1) * A(0,2) * \cdots * A(0,j) = (\text{for } j \le n)\ a_1 * a_2 * \cdots * a_j.$$

For this purpose, let us define $B(\lceil \log n \rceil, 1) = 0$ and

$$B(h,j) = \begin{cases} B(h+1, \frac{1}{2}j) * A(h, j-1) & \text{if } j \text{ is even,} \\ B(h+1, \lceil \frac{1}{2}j \rceil) & \text{else.} \end{cases}$$

On a more intuitive level, we may say that the B of a left-hand son is identical to the B of its father and the B of a right-hand son is (the B of its father) $*$ (the A of its left brother). Obviously this recursive definition implies:

$$B(0,j) = A(0,1) * A(0,2) * \cdots * A(0,j-1).$$

So, by the instruction

$$B(0,j) \leftarrow B(0,j) * A(0,j)$$

we are finished.

With the A's known, the procedure for computing the B's is as follows (assume $p = 2^{\lceil \log n \rceil}$):

Procedure B

```
        1. $B(\lceil \log n \rceil, 1) \leftarrow 0$
        2. $h \leftarrow \lceil \log n \rceil$
    3. Processor i:
            wait until $i \le 2^{\lceil \log n \rceil - h}$
            while $h \ge 0$ do
                    if $2 \lceil \frac{1}{2} i \rceil = i$
                    then $B(h,i) \leftarrow B(h+1, \frac{1}{2} i) * A(h, i-1)$
                    else $B(h,i) \leftarrow B(h+1, \lceil \frac{1}{2} i \rceil)$
                    h $\leftarrow h-1$
                od
                $B(0,i) \leftarrow B(0,i) * A(0,i)$
```

Note that the depth (computation time) of both procedures is proportional to the height of the tree, i.e., $O(\lceil \log n \rceil)$.

2.2. Computing connected components

We asume here that n^2 processors are available. The more efficient implementation is discussed in Section 3. It is recommended that the verbal description of the algorithm given below should be read in conjunction with the more formal one in Fig. 2.

A is the adjacency matrix of an undirected graph. $V = \{1, 2, \ldots, n\}$. Each operation

```
Input: $A = A_0$, the $n \times n$ adjacency matrix for an undirected graph.
Output: $F(v)$ (for all $v \in V$), the smallest serial number vertex in the connected component of vertex $v$.
    1.  $k \leftarrow 0$, $L(0) \leftarrow n$
        for all $v \in V$,
            $N_0(v) \leftarrow v$
while $L(k) > 0$ do (Steps 2 through 9)
    2.  $k \leftarrow k+1$
    3.  for all $v \le L(k-1)$ do (Steps 3 through 8)
            $C_k(v) \leftarrow \text{Min}\{u\colon A_{k-1}(u,v) = 1$ and $u \ne v$ and $u \le L(k-1)\}$
                          if none then $v$
    4.      $D_k(v) \leftarrow C_k(v)$
    5.      for $\lceil \log L(k-1) \rceil$ iterations do
                $C_k(v) \leftarrow C_k(C_k(v))$
            od
    6.      $C_k(v) \leftarrow \text{Min}\{C_k(v), D_k(C_k(v))\}$
    7.      if $C_k(v) = v$ and $D_k(v) \ne v$
            then $S_k(v) \leftarrow$ 'the serial number of $v$ among the vertices of the $k$th graph'
            else $S_k(v) \leftarrow$ QUIT
            $L(k) \leftarrow$ the number of vertices of the $k$th graph
    8.      if $C_k(v) = v$ and $D_k(v) \ne v$
            then $N_k(S_k(v)) \leftarrow N_{k-1}(v)$
        od
    9.  for all $u, v \le L(k-1)$ such that $D_k(u) \ne u$ and $D_k(v) \ne v$ do
            $A_k(S_k(C_k(u)), S_k(C_k(v))) \leftarrow A_{k-1}(u, v)$
        od
od
    10. $l(v) \leftarrow 1$, $P(v) \leftarrow v$
        for all $v \in V$ do
                    while $S_{l(v)}(C_{l(v)}P(v))) \ne$ QUIT do
                        $P(v) \leftarrow S_{l(v)}(P(v)))$
                        $l(v) \leftarrow l(v) + 1$
                    od
                    $F(v) \leftarrow N_{l(v)}(P(v)))$
        od
```

Fig. 2. Algorithm for computing connected components.

of the loop of Steps 2 through 9 shrinks the undirected graph represented by A_j ($j = 0, 1, \ldots$) (hereinafter the *large graph*) into one represented by A_{j+1} (*shrunken graph*). Whenever a set of vertices is shrunk into a single vertex, it can be shown that all these vertices belong to the same connected component. Assume that a connected component was shrunk into a single vertex following several performances of the loop. The next performance of the loop would cause the disappearance of this vertex. The graph that is represented by A_k is called the kth *graph*. Let v be a vertex of the kth graph. $N_k(v)$ denotes the vertex of the original graph that was shrunk during the k performances of the loop into v. The definition and the technique for attaching vertices of the large graph to vertices of the shrunken graph will become clear later.

In order to proceed with the description of the algorithm we need to define rooted trees and rooted stars:

(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.

Let us follow the rules of the kth performance of the shrinking process. The large graph has $L(k-1)$ vertices $\{1, 2, \ldots, L(k-1)\}$, and its adjacency matrix is A_{k-1}. In *Step 3*, vertex v is set to point (by the value assigned to $C_k(v)$), to the smallest vertex adjacent to it; if there is no such vertex, v points to itself. The implementation of this step is discussed later. In *Step 4*, we record $C_k(i)$ in $D_k(i)$ for later use. The directed graph, currently represented by the vector C_k has the following properties:

(1) *the out-degree of each vertex is exactly one, and moreover;*

(2) *its edges can be partitioned into a forest of rooted trees and a set of edges such that each of which emanates from a root of the forest, and is anti-parallel to an edge in the forest.*

Showing that the first property holds is trivial and left to the reader. For the second property take any vertex $v \in \{1, 2, \ldots, L(k-1)\}$. *Claim:* $v \geq C_k(C_k(v))$. This is because v is adjacent to $C_k(v)$ and $C_k(C_k(v))$ is the smallest vertex adjacent to $C_k(v)$. Thus, the sequence obtained by applying C_k even number of times with v as its first element is non-increasing. A closer look would show that it first decreases and then the same element, say u, is repeated, implying that the edge $(u, C_k(u))$ is anti-parallel to $(C_k(C_k(u)), C_k(u)) = (u, C_k(u))$. The length of a directed simple path $v \to C_k(v) \to C_k(C_k(v)) \to \cdots$ does not exceed the number of vertices which is $L(k-1)$. Applying the loop of *Step 5* i times, in a synchronous fashion for all vertices, implies that $C_k(v)$ is assigned with a value which equals 2^i applications of D_k (C_k before Step 5). So the $\lceil \log L(k-1) \rceil$ iterations of Step 5 sets each vertex, x, to point either to its root, r (the root of its tree), or to $D_k(x)$ in the digraph. After *Step 6*, C_k forms a digraph consisting of rooted stars plus self-loops in the roots (note that the serial number of each root is minimal among the vertices in its star).

The implementation of *Step 7* is given later. Its outcome, however, is that for each root v of the rooted stars graph that is not a single vertex in that graph, it assigns a number $S_k(v)$. $S_k(v)$ is the serial number of v among these roots. It is going to be the vertex in the kth graph that v (or its star or any other vertex in its star as we may say), shrinks into it. $S_k(v)$ is set to QUIT for the other vertices. The reason for computing $L(k)$ here (the number of vertices of the kth graph), will be clear after reading the implementation of this step.

Step 8. $N_{k-1}(v)$ is the vertex in the original graph that is associated with v in the $(k-1)$th graph (note that $N_k(v)$ has the minimal serial number among the vertices shrunken into v). v is therefore assigned to $N_k(S_k(v))$, implying that this vertex of the original graph is associated in return with the vertex $S_k(v)$ of the kth graph. The rule for forming the adjacency matrix of the shrunken graph from the large graph (*Step 9*) is: a pair of vertices u, v in the shrunken graph are connected by an edge if and only if a connected pair u_1, v_1 exists in the large graph such that the roots of $u_1 v_1$ were shrunk into u, v respectively. Here is where simultaneous writing of the same value in the same memory location may occur. Obviously, the shrunken graph possesses no more than half the number of vertices than the large graph $(2L(k) \le L(k-1))$. Hence, after at most $\lceil \log n \rceil$ performances of the loop we should arrive at *Step 10.*

$l(v)$ is an auxiliary counter that is attached to vertex v of the original graph. At the beginning of each performance of the loop of Step 10, the variable $P(v)$ contains the name of the vertex of the $(l(v)-1)$th graph (for the present $l(v)$) that v had shrunk into. The value of $l(v)$ on exiting the loop equals:

$$\operatorname*{Max}_{k} \left\{ \begin{array}{l} \text{the } k\text{th graph contains vertices that are associated with} \\ \text{vertices in the connected component of the vertex } v. \end{array} \right.$$

The connected component was shrunken into exactly one vertex $P(v)$ in the $l(v)$th graph. $P(v)$ is associated with vertex $N_{l(v)}(P(v))$ in the original graph. $N_{l(v)}(P(v))$ is the smallest serial number vertex in the connected component of v. This is exactly what we need to assign to $F(v)$ according to the definition of the output in Fig. 2. The loop of Step 10 is performed no more than $\lceil \log n \rceil$ times.

Implementation remarks

(1) The computation of $C_k(v)$ for each $v \le L(k-1)$ in Step 3 is done as follows: Attach $L(k-1)$ processors to v and compute the A numbers as per Subsection 2.1 for the Min operation. $A(\lceil \log L(k-1) \rceil, 1)$ yields the anticipated result that is assigned to $C_k(v)$.

(2) The practice of attaching serial numbers to the vertices that 'survive' the shrinking (roots of 'non-degenerate' components) in Step 7 can be done readily, using our data structure. The a_v numbers would satisfy $(v = 1, \ldots, L(k-1))$:

$$a_v = \begin{cases} 1, & \text{if } C_k(v) = v \text{ and } D(v) \ne v, \\ 0, & \text{else.} \end{cases}$$

Thus, for every v such that $a_v = 1$, $B(0, v)$ is the serial number of v among these surviving vertices, and is assigned to $S_k(v)$. $S_k(v)$ of the vertices v such that $a_v = 0$ is assigned by QUIT. $L(k)$ is set to the value of $A(\lceil \log L(k-1) \rceil, 1)$.

(3) It can be shown that all the initializations of the required variables affect neither the depth analysis nor the number of required processors. This comment applies to the next section as well.

Depth evaluation

Step 1:	$O(1)$,	Step 6:	$O(\log n)$,
Step 2:	$O(\log n)$,	Step 7:	$O(\log^2 n)$,
Step 3:	$O(\log^2 n)$,	Step 8:	$O(\log n)$,
Step 4:	$O(\log n)$,	Step 9:	$O(\log n)$,
Step 5:	$O(\log^2 n)$,	Step 10:	$O(\log n)$.

Hence, the total depth of the algorithm is $O(\log^2 n)$.

Proof of the correctness of the algorithm is dispensed with as it is a modified version of [7]. The modifications that lead to better depth in the next section are in Steps 7 and 10, in the presentation of the A_k matrices (which become smaller as the shrinking process goes on), and in the introduction and efficient use of the data structure (see Section 3 below).

3. Optimal implementation

The scheme of optimal implementation is based on the following theorem and its proof.

Theorem 3.1. (Brent [2]). *Any synchronized parallel algorithm of depth d, that consists of x elementary operations, can be implemented by p processors with a depth of $\lceil x/p \rceil + d$.*

Proof. Let x_i denote the number of operations performed by the algorithm in time i, ($\sum_1^d x_i = x$). We now use the p processors to 'simulate' the algorithm. Since all operations in time i can be executed simultaneously, they can be computed by the p processors in $\lceil x_i/p \rceil$ units of time. Thus, the whole algorithm can be implemented by p processors in time

$$\sum_1^d \lceil x_i/p \rceil \le \sum_1^d (x_i/p + 1) \le \lceil x/p \rceil + d. \qquad \square$$

The proof of Brent's theorem entails two implementation problems: how to evaluate x_i at the beginning of time i in the algorithm, and how to assign the processors to their jobs without increasing the depth that is yielded by Brent's theorem by an order of magnitude. The reader is invited to verify for himself that the problems can be solved easily in each of the steps of the algorithm. After Lemma 3.2, however, there is a hint that may help to do it in Step 10.

Having shown earlier that d is $O(\log^2 n)$, we now claim that x (of Brent's theorem) equals $O(n^2)$. The following two lemmas will help us to prove this claim.

Lemma 3.1. *The total number of operations that is required for Step* 3 *is* $O(n^2)$.

Proof. Whenever the data structure of Subsection 2.1 is used, the number of operations involved is proportional to that of nodes of the P*-tree.

In the first performance of the 'main loop' (Steps 2 through 9), we have a tree of $2.2^{\lceil \log n \rceil}$ nodes (since there are $2^{\lceil \log n \rceil}$ leaves) for each vertex v, $v = 1, 2, \ldots, n$. $2.2^{\lceil \log n \rceil} < 4n$, implying that the total number of nodes is bounded by $4n^2$. $2L(k) \leq L(k-1)$ implies that the 1st graph that is the input to the second performance of the main loop contains no more than $\frac{1}{2}n$ vertices, i.e., the total number of nodes is bounded by n^2. In the next performance, the number is again reduced by a factor of at least 4. Hence, the total number of elementary operations needed for Step 3 is $O(n^2)$. □

A similar reasoning shows that the total number of operations required for Step 7 is $O(n)$.

Lemma 3.2. *The total number of operations required for Step* 9 *is* $O(n^2)$.

Proof. The first performance of Step 9 takes n^2 operations; the second $L^2(1)$ $(\leq \frac{1}{4}n^2)$ operations and so on. The subsequent reasoning is as in the preceding proof. □

In order to both calculate smoothly the number of elementary operations of Step 10 and the assignment of processors to their jobs during that step, we attach a processor to each of the vertices during Step 10. A few processors may exit the loop of Step 10 before others and, in that case, they will carry out the instruction: 'remain idle' until the algorithm is over. Namely, Step 10 elapses $O(\log n)$ time units, independently of the input and requires no more than $O(n \log n)$ operations. The number of operations needed for all the other steps is also easy to calculate, therefore, the following corollary can be stated.

Corollary 3.3. *The total number of elementary operations in the algorithm is* $O(n^2)$. *We conclude by reminding the reader that we actually proved, with the aid of Brent's theorem, that the depth of the algorithm is* $O(n^2/p)$ *for* $p \leq n^2/\log^2 n$.

At last we would like to go back and defend one of our claims that our algorithm is simpler than [4]. We do it by pointing out where sorting is used in their algorithm. Due to the concurrent-write assumption Step 9 is performed instantaneously regardless of the number of processors attempting to write into the same memory location or whether this number is known. On the other hand it seems impossible to compute this OR function in a CREW PRAM simultaneously for all entries of the matrix into which concurrent-write should be performed without powerful and time consuming techniques like sorting. They show that this can be done without multiplying the total running time by more than a constant by giving a lengthy and time consuming solution for this problem.

Acknowledgement

I would like to thank Y. Shiloach for stimulating discussions.

References

[1] A. Borodin and J.E. Hopcroft, 'Routing, merging and sorting on parallel models of computation, Proc. 14th ACM Symp. Theory of Computing (1982) 338-344.

[2] R.P. Brent, The parallel evaluation of general arithmetic expressions, J. ACM 21 (1974) 201-206.

[3] S. Cook and C. Dwork, Bounds on the time for the parallel RAM's to compute simple functions, Proc. 14th ACM Symp. Theory of Computing (1982) 231-233.

[4] F.Y., Chin, J. Lam and I. Chen, Optimal parallel algorithms for the connected component problem, Proc. 1981 Internat. Conf. Parallel Processing (1981) 170-175.

[5] D.M. Eckstein, Simultaneous memory access, TR-79-6, Computer Science Dept., Iowa State University, Ames, IA (1979).

[6] L.M. Goldschlager, A unified approach to models of synchronous parallel computation, Proc. 10 ACM Symp. Theory of Computing (1978) 89-94.

[7] D.S. Hirschberg, A.K. Chandra and D.V., Sarwate, Computing connected components on parallel computers, Comm. ACM 22 (8) (August 1979) 461-464.

[8] D.S. Hirschberg, Parallel algorithms for the transitive closure and the connected component problem, Proc. 8th ACM Symp. Theory of Computing, Hershey, PA (1976) 55-57.

[9] J.T. Schwartz, Ultracomputers, ACM Trans. Programming Languages and Systems 2 (1980) 848-521.

[10] Y. Shiloach and U. Vishkin, Finding the maximum, merging and sorting in a parallel computation model, J. Algorithms 2 (1981) 88-102.

[11] Y. Shiloach and U. Vishkin, An O(log n) parallel connectivity algorithm, J. Algorithms 3 (1982) 57-67.

[12] Y. Shiloach and U. Vishkin, An O(n^2 log n) parallel max-flow algorithm, J. Algorithms 3 (1982) 128-146.

[13] L.J. Stockmeyer and U. Vishkin, Simulation of parallel random access machines by circuits, SIAM J. Comput. 13 (2) (1984) 409-422.

[14] R.E Tarjan and U. Vishkin, An efficient parallel biconnectivity algorithm, TR-69, Dept. of Computer Science, Courant Institute, NYU (1983), SIAM J. Comput., to appear.

[15] U. Vishkin, Implementing simultaneous memory address access in models that forbid it, J. Algorithms 4 (1983) 45-50.

[16] U. Vishkin, On choice of a model of parallel computation, TR-61, Dept. of Computer Science, Courant Institute, NYU (1983), J. Comput. System Sci., to appear.

[17] J.C. Wyllie, The complexity of parallel computation, Ph.D. Thesis, Dept. of Computer Science, Cornell University, Ithaca, NY (1979).