Information Processing Letters 20 (1985) 235–240
North-Holland

12 June 1985

ON EFFICIENT PARALLEL STRONG ORIENTATION *

Uzi VISHKIN **

Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, Washington Square, New York, NY 10003, U.S.A.

Communicated by G.R. Andrews
Received 6 March 1984
Revised 10 September 1984

Keywords: Graph algorithm, parallel algorithm, strongly connected component

1. Introduction

The family of models of computation used in this paper is the family of parallel random access machines (PRAMs). All members of this family employ p synchronous processors all having access to a common memory. We mention three members of the PRAM family in descending order of strength. In a concurrent-read concurrent-write (CRCW) PRAM, simultaneous reading from the same memory location is allowed as well as simultaneous writing. In the latter case, the lowest numbered processor succeeds. A concurrent-read exclusive-write (CREW) PRAM allows simultaneous reading into the same memory location but not simultaneous writing. An EREW PRAM does not allow simultaneous reading or writing (see [12] for a recent survey of results concerning the PRAM family).

Recall that a bridge in a connected graph is an edge whose removal disconnects the graph and a digraph is strongly connected if for every two vertices u and v there is a directed path from u to v. The following problem is considered.

Input. A connected bridgeless undirected graph $G = (V, E)$, $|V| = n$, $|E| = m$.

The strong orientation problem. Assign directions to its edges so that the resulting digraph is strongly connected.

It is known that such a strong orientation exists iff the input graph is connected and bridgeless. Atallah [3] indicates that this problem has a simple linear time serial solution: Apply depth-first search to the graph. Orient tree edges away from the root and back edges towards the root.

Since depth-first search seems to be inherently serial (see [5] for some evidence), another approach is required for designing efficient parallel algorithms. Atallah [3] gave an $O(\log n)$ time algorithm for this problem using n^3 processors on a CRCW PRAM. Atallah's algorithm runs in $O(n^3/p + \log^2 n)$ time using p processors on a CREW PRAM. Our algorithm runs in $O(\log n)$ time using only $n + m$ processors on a CRCW PRAM. An alternative implementation of the algorithm runs in $O(n^2/p)$ time using $p \leqslant n^2/\log^2 n$ processors on a CREW PRAM. This is optimal for dense graphs. A very high-level description of our algorithm is close to Atallah's algorithm. However, only a complete and drastic change in the way parallelism is approached allows the claimed improvements in efficiency. Algorithmic methods that reduce the number of processors without changing the running time are of great importance for the design of parallel algorithms. A survey by the present author [12] mentions many parallel algorithms that achieved a previously known run-

* This research was supported by DOE Grant DE-AC02-76ER03077 and by NSF Grants DCR-8318874 and MCS79-21258.

** Present affiliation: Department of Computer Science, Tel Aviv University, Ramat Aviv, Tel Aviv 69978, Israel.

ning time with less processors. Tarjan and Vishkin [9] introduced a useful technique for computing various functions on trees using an Euler path that visits each edge of the tree exactly twice. Substantial simplifications and amplifications of this technique are also given. They have already been used in a later version of [9] to improve the presentation there. This technique needs O(log n) time using O(n) space and n processors on an EREW PRAM. The article by Tarjan and Vishkin [10] is a preliminary version of both [9] and the present paper.

We assume throughout that our input graph is connected and bridgeless. If we do not know this, we can check connectivity using the algorithms of [7], [4] or [11] depending on the complexity efficiency being sought (O(log n) time using n + m processors or the alternative one). Existence of bridges can be checked using the algorithm of [9] within the efficiencies claimed here.

In order to keep this presentation short, our exposition will focus on the main algorithmic aspects while, like [3], tedious implementation details will be suppressed. Also, there are several cases where a more involved implementation would enable big portions of the algorithm to run on an EREW PRAM instead of the CREW PRAM as presented. The main combinatorial observations required for the algorithm are given in Section 2. A detailed description follows in Section 3.

2. A high-level description of the algorithm

2.1. Atallah's algorithm

Let T be an undirected spanning tree of G, and H be a directed version of T rooted at a vertex r.

(a) Find T and H. Let $q = |E - T|$. Assign arbitrarily serial numbers from 1 to q to edges of $E - T$, so that $E - T = \{e_1, \ldots, e_q\}$.

Let C_k be the fundamental cycle which is induced by e_k, $1 \leqslant k \leqslant q$. Let α be an edge of T and $C_{i_1}, C_{i_2}, \ldots, C_{i_j}$ be all fundamental cycles that contain α, where $1 \leqslant i_1 < i_2 < \cdots < i_j \leqslant q$. Define the lowest numbered cycle, C_{i_1}, to be the *master cycle* of α, and e_{i_1} to be the *master edge* of α (denoted MASTER(α)).

(b) Assign an arbitrary direction to every edge in $E - T$. To each α in T assign the direction which conforms with the direction of MASTER(α) in the master cycle of α.

Theorem (Atallah [3]). *The algorithm produces a strong orientation of* G.

2.2. The new algorithm

Our algorithm introduces a refinement into the assignment of serial numbers to the edges of $E - T$. Perhaps surprisingly, this turns out to be very powerful since it enables the improvements in efficiency. The rest of this section is devoted to a high-level description of the algorithm.

G, T, H, r, and q are as before.

Step 1. Find T.

Step 2. Compute H.

Consider replacing each edge (u, v) of T by two anti-parallel directed edges $u \to v$ and $v \to u$. The new digraph is called T'. Obviously, every edge of H belongs to T' as well. Since in-degree(v) = out-degree(v) for every vertex v in T', T' has an Euler path that starts and ends at the root r.

Remark. Step 2 also computes this path into a vector named D. (Namely, for every edge of T', D will include its successor in the Euler path.) As can be seen in the next section, this path plays an important role in the computation of quite a few instructions of our algorithm. In this section we refer to the Euler path only in the Main Lemma.

Step 3. Compute preorder and postorder numbering of the vertices of H and their levels (lengths of the directed paths from r to them in H). For every vertex v denote these numbers by PREORDER(v), POSTORDER(v), and LEVEL(v), respectively.

Let $u_1, \ldots, u_\ell$ be all vertices which are adjacent to v by an edge of $E - T$. Denote by Ad-LCA(v) the lowest common ancestor in H of $v, u_1, \ldots, u_\ell$. (Ad-LCA(v) = LCA(v, $u_1, \ldots, u_\ell$).) We say that an edge $e = (v, u) \in E - T$ *causes* Ad-LCA(v) if LCA(v, u) = Ad-LCA(v).

Step 4.1. For every vertex v, compute Ad-LCA(v) and if Ad-LCA(v) is different than v, select (arbitrarily) an edge e that causes it.

Step 4.2. Assign to each edge e which was selected at Step 4.1 a new 'serial number' SERIAL(e)

as follows. SERIAL(e) is a pair which consists of the level of Ad-LCA(v) and its original serial number (a number between 1 and m). That is, we have SERIAL(e): = (LEVEL(Ad-LCA(v)), "original serial number of e").

Let e be any edge in E − T such that SERIAL(e) was not defined in Step 4.2. For *explanation purposes only*, set

$$\text{SERIAL}(e) := (\text{LEVEL}(\text{LCA}(e)), \text{"original serial number of e"} + m).$$

The discussion below explains why we can dispense with this assignment (and actually ignore all edges of E − T whose SERIAL(e) was not defined in Step 4.2 in the computation of master edges that follows) without affecting the correctness of the algorithm.

Let e and f be two edges of E − T. Define a lexicographic order $<_1$ between SERIAL(e) and SERIAL(f) as follows:

$$\text{SERIAL}(e) <_1 \text{SERIAL}(f)$$

if $\text{SERIAL}(e_1).1 < \text{SERIAL}(E_2).1$

or

$\text{SERIAL}(e_1).1 = \text{SERIAL}(e_2).1$ and

$\text{SERIAL}(e_1).2 < \text{SERIAL}(e_2).2$,

where, given a pair (a, b), (a, b).1 is a and (a, b).2 is b. $<_1$ is a complete order on the SERIAL(−) values of the edges of E − T.

Let f be an edge of T. Our goal is to compute MASTER(f). It is the edge $e \in E - T$ with minimum SERIAL(e) whose fundamental cycle contains f.

Claim. *Let* $e = (u, v) \in E - T$ *be any edge whose* SERIAL(e) *was not defined in Step* 4.2. *It is impossible that* e *is the master edge of any edge of* T.

Proof. Let $w \to u$ be the father edge of u in H. If (u, w) is included in the fundamental cycle of (u, v), then due to the fact that (u, v) was not selected in Step 4.2 there must be another edge which is adjacent to u, say (u, x), such that LCA(u, x) is an ancestor of or equal to LCA(u, v). Therefore, $\text{SERIAL}((u, x)) <_1 \text{SERIAL}((u, v))$ and (u, x) 'beats' (u, v) in the contest to become the master edge of each of the edges along the path from u to LCA(u, v) in H. Similar considerations imply that (u, v) can never be the master edge of any edge on the path from v to LCA(u, v) in H. □

Step 5. Let f = (u, v) be an edge of T and say that u is the father of v in H. For each such edge initialize MASTER(f) to MIN{SERIAL(e)} where the minimum (with respect to $<_1$) is taken over all edges e = (w, v) in E − T whose SERIAL(e) is defined in Step 4.2.

The *interval* of $f (\in T)$ in the Euler path of T′ is the portion of the Euler path that starts at the $u \to v$ copy of f and ends at the $v \to u$ copy. Our presentation identifies MASTER(f) with both MASTER($u \to v$) and MASTER($v \to u$). No confusion will arise.

Main Lemma. *Let* f = (u, v) *be an edge in* T *where* u *is the father of* v *in* H, *as before. Then, the master edge of* f *is the edge of* E − T *that provides the minimum* MASTER(−) *value over the edges in the interval of* f.

Proof. 'Disconnect' T by 'deleting' f. The interval of f in the Euler path of T′ scans the component of v in the disconnected tree. The master edge of f comes from the edges in E − T which have one endpoint in the component of v and the other at the component of u. All these edges are taken into account in the minimum computation for f. Let e be an edge of E − T whose both endpoints are in the component of v. Observe that SERIAL(e) is taken into account in the minimum computation for f. This seems incorrect! However, we show that fortunately SERIAL(e) must loose in the minimum computation to an edge that has only one endpoint in the component of f. (This explains the role of our refined assignment of serial number to edges of E − T.) There must be an edge g in E − T that has only one endpoint at the component of v, or f is a bridge of the input graph. LCA(g) is an ancestor of v. Thus, LCA(g) is an ancestor of LCA(e), and therefore, $\text{SERIAL}(g) <_1 \text{SERIAL}(e)$. □

Steps 6 *and* 7. Compute the minima described in the Main Lemma for every interval of an edge in T.

The last two steps of the algorithm are the same as in [3].

Step 8. Assign (arbitrarily) a direction to every edge in $E - T$.

Step 9. To every $(u, v) \in T$, assign a direction that agrees with the fundamental cycle of its master. This is done as follows: Let $(i, j) \in E - T$ be the master of (u, v) and assume, w.l.o.g., that u is the father of v in H and that (i, j) was given the direction $i \to j$ during the previous step of the algorithm. One of two cases can occur: (1) If v is an ancestor of i, then the $u \to v$ direction is assigned to (u, v). (2) If v is an ancestor of j, then the $v \to u$ direction is assigned to (u, v).

3. The algorithm

Our algorithm has two implementations. The first runs in $O(\log n)$ time using $n + m$ processors on a CRCW PRAM. The second runs in $O(n^2/p)$ time using p processors on a CREW PRAM. (The present author [11] achieved the same efficiency by a simpler algorithm. However, it uses a CRCW PRAM.)

Step 1. A spanning tree T of G is computed. For the first implementation we use an adaptation of the connectivity algorithm of [7] as described in [2] and [9]. It runs in $O(\log n)$ time using $n + m$ processors on a CRCW PRAM. The second implementation uses the $O(n^2/p)$ time algorithm of [4] that uses $p \leqslant n^2/\log^2 n$ processors on a CREW PRAM.

The efficiency of the rest of the algorithm is optimal for graphs with $m = \Omega(n \log n)$: It takes $O(\log n)$ time using $(n \log n + m)/\log n$ processors on a CREW PRAM. So, if it will be possible in the future to find a spanning tree within this optimal efficiency (for Step 1) then the efficiency of the present algorithm will be improved as well.

Step 2. Find H, a directed version of T which is rooted at some vertex r. We summarize below a solution of [9] for Steps 2 and 3 since extentions of this solution are needed for later steps of the present algorithm.

Step 2.1. Replace each edge (u, v) of T by two anti-parallel directed edges $u \to v$ and $v \to u$ to from T'.

Step 2.2. For each vertex v of T we do the following. (Let degree(v) = d in T and let the d adjacent edges of v in T be $(v, u_1), \ldots, (v, u_d)$.)

$$D(u_i \to v) := v \to u_{i+1 \bmod d} \quad \text{for } 1 \leqslant i \leqslant d.$$

Now D has an Euler cycle. The 'correction',

$$D(u_d \to r) := \text{'end of list'} \quad (\text{where } d = \text{degree}(r))$$

gives an Euler path that starts and ends at r.

Step 2.3. (The distance of each edge of T' from the end of the path is computed into a vector R using a 'doubling' procedure.)

Initialize: $R(e) := 1$ for all edges e in T', and R ('end of list'): = 0.

Apply $\lceil \log(2n - 2) \rceil$ iterations in parallel ($2n - 2$ is the length of the Euler path):

$$R(e) := R(e) + R(D(e)), \quad D(e) := D(D(e)).$$

Step 2.4. (Determine H.)

if $R(e) > R$ ('anti-parallel edge of e')

then e (and not its anti-parallel edge) is oriented away from the root r.

Remarks. (1) Let f be an edge in T. For better understanding of some of the following steps of the algorithm try to think about the (directed) copy of $f \in H$ in the Euler path as a left parenthesis and its anti-parallel edge as a right parenthesis. If we do so for every such f, the Euler path will correspond to a legal sequence of parentheses, where matching pairs of parentheses will represent the two copies of an edge of T.

(2) In order to obtain initially a 'desired' representation of the tree, Tarjan and Vishkin [9] propose to consider application of the sorting algorithm of [1] (respectively [6]) to the edges of the tree. This algorithm runs in $O(\log n)$ time (respectively almost surely) using n processors on an EREW PRAM. We avoid elaborating on this.

Step 3. (Compute POSTORDER(v), PREORDER(v), and LEVEL(v) for every vertex v.)

For all edges $e \in H$ (respectively $e \in T' - H$) initialize $R(e) := 1$ (respectively $R(e) := 0$).

The doubling procedure assigns to R(e) of each edge $e \in T'$ how many vertices were entered for

the first time by the Euler path following the appearance of e on this path. It is a simple exercise to compute preorder numbering from this information. Computation of postorder numbering is obtained similarly:

For all edges $e \in H$ (respectively $e \in T' - H$) initialize R(e): = 0 (respectively R(e): = 1). Proceed similarly to the preorder computation.

Computation of levels: For all edges $e \in H$ (respectively $e \in T' - H$) initialize R(e): = 1 (respectively R(e): = −1). Proceed similarly to the preorder computation.

Step 4.1. (For every vertex v compute Ad-LCA(v) and if Ad-LCA(v) is different than v, select (arbitrarily) an edge e that causes it.)

Let the predicate ANCESTOR(u, v) be 'true' iff u is the ancestor of v in H. Recall (see [8]) that

$$\text{ANCESTOR}(u, v) = [\text{PREORDER}(u) < \text{PREORDER}(v)] \text{ AND } [\text{POSTORDER}(u) > \text{POSTORDER}(v)].$$

Each of the ℓ edges $(v, u_1), \ldots, (v, u_\ell)$ of E − T which are adjacent to v falls into one of four cases.

Case 1:

$$[\text{PREORDER}(u_k) > \text{PREORDER}(v)] \text{ AND } [\text{POSTORDER}(u_k) < \text{POSTORDER}(v)].$$

In this case, v is an ancestor of u_k. Therefore, these edges cannot affect Ad-LCA(v) and no operation is performed.

Case 2:

$$[\text{PREORDER}(u_k) < \text{PREORDER}(v)] \text{ AND } [\text{POSTORDER}(u_k) < \text{POSTORDER}(v)].$$

Find among these edges the one which has minimum PREORDER(u_k). (Denote this edge by (v, u_{ℓ_1}).)

Case 3:

$$[\text{PREORDER}(u_k) > \text{PREORDER}(v)] \text{ AND } [\text{POSTORDER}(u_k) > \text{POSTORDER}(v)].$$

Find among these edges the one which has maximum POSTORDER(u_k). (Denote this edge by (v, u_{ℓ_2}).)

Find LCA(v, u_{ℓ_1}) and LCA(v, u_{ℓ_2}). (The LCA computation is described after Step 7. The complexity analysis uses the fact that the LCAs of at most 2n edges may be required for this instruction.)

Case 4:

$$[\text{PREORDER}(u_k) < \text{PREORDER}(v)] \text{ AND } [\text{POSTORDER}(u_k) > \text{POSTORDER}(v)].$$

All these u_k's, as well as LCA(v, u_{ℓ_1}) and LCA(v, u_{ℓ_2}), are ancestors of v. So, they are all on the path from v to the root in H and therefore ANCESTOR(−,−) is a complete order on them.

Find the lowest ancestor among them by applying a minimum computation. This ancestor is the required Ad-LCA(v) and an edge that caused it is the required e.

Implementation. n processors are used. The classification into the cases takes O(1) time. The minimum and maximum computations in Cases 2, 3, and 4 can be done in O(log n) time using the obvious algorithm (that is, compare, iteratively, disjoint pairs of 'winners').

Step 4.2. Assign to each edge e which was selected at Step 4.1 a new 'serial number' SERIAL(e),

$$\text{SERIAL}(e) := (\text{LEVEL}(\text{Ad-LCA}(v)), \text{"original serial number of e"}).$$

Step 5. Let f = (u, v) be an edge of T and say that u is the father of v in H. For each such f,

$$\text{MASTER}(f) := \text{MIN}\{\text{SERIAL}(e)\},$$

where the minimum (with respect to $<_1$) is taken over all edges e = (w, v) in E − T whose SERIAL(e) is defined in Step 4.2. (There are $\leqslant n$ such edges.)

Assume, w.l.o.g., that the length of the Euler path (2n − 2) is a power of two.

Step 6. Compute MIN{MASTER(−)} over all edges of T′ (the edges of the Euler path). For reasons that will become clear shortly we insist on computing this minimum as follows.

Apply the obvious parallel algorithm which uses a balanced binary tree whose leaves correspond to directed edges of the Euler path in the order of the path. This algorithm finds first the minimum of successive pairs of edges, then of successive fours, then of successive eights, and so on. This can be illustrated by an upward motion in Fig. 1: Each level of the binary computation tree corresponds to a stage in the computation of Step 6. Each level of Fig. 1 gives the segments of the Euler path

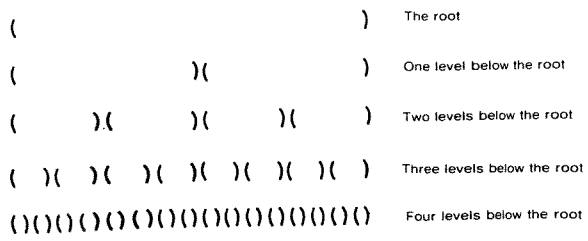

Fig. 1. The binary computation tree.

whose minima are computed in the corresponding stage of Step 6. During the course of the algorithm we save all these intermediate minimum computations.

Step 7. The interval of each edge f of T can be represented as a union of O(log n) intermediate minima segments which are pairwise disjoint. We find the minimum of SERIAL(−) over the interval of f by computing the minimum of the minima of these segments.

Complexity. Assign a processor to each of the $n-1$ edges of T. It has to compute which segments should be considered and compute the minimum. All this needs O(log n) time.

Next, we show how this technique can be used to compute the lowest common ancestor of any two vertices u and v in H (as required before). Assume, w.l.o.g., that PREORDER(u) < PREORDER(v). Consider the interval of the Euler path that starts (exactly) after the edge of the form $w_1 \to u$ and ends before the edge of the form $v \to w_2$. Observe that the vertex which yields the minimum in a computation of minimum level over the vertices of this interval is LCA(u, v). Levels of vertices in H were computed earlier. The rest of the computation of LCA(u, v) is similar to Steps 6 and 7.

We do do not repeat here the last two steps of the algorithm which where given in the previous section.

Complexity. Step 1: O(log n) time using $n+m$ processors or $O(\log^2 n)$ time using $(n/\log n)^2$ processors.

Using n processors, Steps 2, 3, 4, 5, 6, and 7 take O(log n) time each. Step 8 needs O(1) time using $m-(n-1)$ processors. Step 9 needs O(1) time using n processors. And the complexity results claimed after Step 1 of this section follow.

Conclusion

The reader is invited to check that the space requirement of our main implementation is O(n + m). Most of our use of the doubling procedure was in order to rank a weighted linked list. It took us O(log n) time using n processors. The present author [13] presents randomized algorithms that do it almost surely in O(log n log*n) (respectively O(log n)) time using n/(log n log*n) (respectively n log log n/log n) processors.

References

[1] M. Ajtai, J. Komlós and E. Szemerédi, An O(n log n) sorting network, Combinatorica 3 (1) (1983) 1–19.

[2] B. Awerbuch and Y. Shiloach, New connectivity and MSF algorithms for Ultracomputer and PRAM, Proc. 1983 Internat. Conf. on Parallel Processing, 1983.

[3] M. Atallah, Parallel strong orientation of an undirected graph, Inform. Process. Lett. 18 (1984) 37–39.

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

[5] J.H. Reif, Depth-first search is inherently sequential, Inform. Process. Lett. 20 (5) (1985) 229–234 (this issue).

[6] J.H. Reif and L.G. Valiant, A logarithmic time sort for linear size network, Proc. 15th ACM Symp. on Theory of Computing (1983) 10–16.

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

[8] R.E. Tarjan, Finding dominators in directed graphs, SIAM J. Comput. 3 (1974) 62–89.

[9] R.E. Tarjan and U. Vishkin, An efficient parallel biconnectivity algorithm, Tech. Rept. 69, Dept. of Computer Science, Courant Institute, New York Univ., 1983; also: SIAM J. Comput., to appear.

[10] R.E. Tarjan and U. Vishkin, Finding biconnected components and computing tree functions in logarithmic parallel time, Proc. 25th IEEE Symp. on Foundations of Computer Science (1984) 12–20.

[11] U. Vishkin, An optimal parallel connectivity algorithm, Discrete Appl. Math. 9 (2) (1984) 197–207.

[12] U. Vishkin, Synchronous parallel computation—a survey, Tech. Rept. 71, Dept. of Computer Science, Courant Institute, New York Univ., 1983.

[13] U. Vishkin, Randomized speed-ups in parallel computation, Proc. 16th ACM Symp. on Theory of Computing (1984) 230–239.