COMBINATORICA 7 (4) (1987) 315—326

A PARALLEL ALGORITHM FOR THE MAXIMAL PATH PROBLEM

R. ANDERSON

Received 1 July 1985
Revised 1 March 1986

In this paper we present an $O(\log^5 n)$ time parallel algorithm for constructing a Maximal Path in an undirected graph. We also give an $O(\log^{1/2+\varepsilon})$ time parallel algorithm for constructing a depth first search tree in an undirected graph.

1. Introduction

One of the important areas of research in parallel computation is to identify problems that can be sped up significantly using parallelism. The goals of this work are to find fast parallel algorithms for specific problems and to develop techniques for parallel algorithms. There has been quite a bit of progress made recently, with parallel algorithms being developed for such problems as maximal independent set [7] and matching [8]. In this paper we give a fast parallel algorithm for the maximal path problem.

The maximal path problem is: Given a graph $G=(V, E)$ and a vertex $r \in V$, find a simple path P starting at r such that P cannot be extended without encountering a vertex that is already on the path. This problem is rather trivial from the sequential point of view, since it can be solved by a simple greedy algorithm. The main result of this paper is that a maximal path can be found in $\mathcal{RNC}$, that is, there is a probabilistic parallel algorithm that solves the problem in polylog time using a polynomial number of processors. This result is somewhat surprising; the problem appears to be very sequential in nature, because to add a vertex to the path we need to know that the vertex is not already on the path.

The maximal path problem is closely related to depth first search. This was the main reason for our study of the problem. Any branch of a depth first search tree is a maximal path. A fast parallel algorithm for finding a maximal path does not seem to give a fast solution to depth first search, since many maximal paths might be needed to construct a depth first tree. However, the maximal path problem does

This work was supported in part by an IBM Faculty Development Award, an NSF Graduate Fellowship, and NSF grant DCR-8351757.
AMS subject classification (1980): 68 Q 10, 05 C 38.

seem to capture some of the difficulty of depth first search. The techniques that we use for our maximal path algorithm do apply to depth first search. Using them we can construct a depth first search tree in $O(\sqrt{n}\log^c n)$ time for some small c where n is the number of vertices. This is the first sublinear parallel algorithm for depth first seach.

The greedy algorithms for finding a maximal path and for computing a depth first search tree probably cannot be sped up significantly using parallelism. It has been shown that the problems of computing the solution found by the greedy algorithms for these problems are P-complete [9], [1]. For the maximal path problem this creates a similar situation to the maximal independent set problem [3], [7]; the greedy algorithm is probably inherently sequential, but a different approach yields a fast parallel solution.

The maximal path algorithm is quite complicated and uses some fairly powerful machinery. The algorithm uses a divide and conquer strategy. A path is found that reduces the problem to finding a maximal path in a graph of at most half the original size. This path is referred to as a *splitting path.* A set Q of vertex disjoint paths is said to *separate* the graph if $V-Q$ has at least two connected components. A splitting path is constructed from a small set of paths that separate the graph. A major portion of the algorithm is to construct the separating paths. The basic approach is to start with a large number of paths and then to reduce their number until just a few remain. The paths are reduced by finding vertex disjoint paths between them. The vertex disjoint paths are found by reducing the problem to a matching problem. The only use of randomness in this algorithm comes from employing the Karp-Upfal-Wigderson matching algorithm as a subroutine; if matching could be solved in $\mathcal{NC}$, then the maximal path problem also could be solved in $\mathcal{NC}$.

The main body of the maximal path algorithm is described in the next two sections. Section Two covers various reductions that are used to reduce the problem to finding a small set of isolated paths. Section Three gives the algorithm for constructing the isolated paths. Section Four describes the reduction of the problem of finding disjoint paths to matching, completing the $\mathcal{RNC}$ algorithm for finding a maximal path. The final section of the paper gives an application of the disjoint paths routine to depth first search, yielding an $O(\sqrt{n}\log n)$ algorithm.

The underlying model of parallel computation that we use is the P-RAM model [4]. We however, describe the algorithms in a high level manner, ignoring a number of details of parallel implementation. A number of parallel algorithms for simple graph problems and data structure manipulation are used. These include $\mathcal{NC}$ algorithms for finding connected components [11], finding articulation points [10], finding a shortest path between a pair of vertices and maintaining a graph when adding or deleting vertices or edges.

In this paper we make liberal use of the notation that identifies an induced subgraph with a set of vertices. Throughout the paper, the induced subgraphs are with respect to the graph $G=(V, E)$. We further abuse notation by performing set operations between sets of vertices and sets of paths, in such cases the set of paths is viewed as a set of vertices. For example, if Q is a set of paths, then $V-Q$ denotes the induced subgraph on the vertices that are not on any path in Q.

2. Reductions

The maximal path problem is: Given a graph $G=(V, E)$ and a vertex $r \in V$, find a simple path P starting at r such that P cannot be extended without encountering a vertex that is already on the path. Our algorithm for finding a maximal path is a divide and conquer algorithm. The algorithm finds a *splitting path* which reduces the problem to finding a maximal path in a smaller graph.

Definition 2.1. A path P starting from r is called a *splitting path* if $V-P$ has at least two connected components.

Lemma 2.2. *If splitting paths can be found in time $T(n)$, then a maximal path can be found in time $T(n) \log n$.*

Proof. Suppose $P=ru_1 \ldots u_k$ is a splitting path. Let u_j be the last vertex on P such that u_j is adjacent to at least two components of $V-ru_1 \ldots u_j$. Let C be the smallest of the components of $V-ru_1 \ldots u_j$ adjacent to u_j, and let v be a vertex in C that is adjacent to u_j. If P' is a maximal path from v in C, then $ru_1 \ldots u_j P'$ is a maximal path from r in V, so the problem is reduced to finding a maximal path in C. Since $|C| < n/2$, if a splitting path can be found in time $T(n)$, then the maximal path problem can be solved in $T(n) \log n$ time. ∎

The major portion of the maximal path algorithm is concerned with finding a splitting path. In this section we show how to construct a splitting path from a set of vertex disjoint paths $Q=\{Q_1, \ldots, Q_k\}$ where $V-Q$ has at least two connected components and $k \leqq c$ for some fixed constant c.

If a graph is not biconnected then it is straightforward to find a splitting path. We may as well assume that r has degree at least two. Otherwise, if r has degree one, then we can follow a path from r until we reach a node of degree at least three. Suppose the graph has an articulation point and r has degree at least two. If r is an articulation point, then the trivial path just containing r is a splitting path. If r is not an articulation point, then there is an articulation point v in the same biconnected component as r. Let P be a path of minimum length from r to v. There must be some vertex v' in the biconnected component containing r and v that is not on P, hence P is a splitting path. For the remainder of the section, we assume that G is biconnected.

We show how to construct a splitting path from a small set of vertex disjoint paths that separate the graph. Suppose $Q=\{Q_1, \ldots, Q_k\}$ is a set of vertex disjoint pahts. A subroutine that builds a single path using the paths in Q is used in the construction of a splitting path. The routine *Extend Path* (r, Q, W) construct a path starting from r that cannot be extended to include any more vertices of Q. *Extend Path* works on the subgraph induced on the set of vertices W. In other words, if the constructed path is $P=ru_1 \ldots u_k$, then no vertex lying on any path in Q is contained in any connected component of $W-P$ adjacent to u_k. Such a path is said to be *maximal with respect to Q*. *Extend Path* is essentially a sequential greedy algorithm. It builds the path by adding segments of the paths in Q to the current path for as long as possible. The only use of parallelism is in the low-level routines which find shortest paths and maintain connected components. The routine is:

Extend Path (r, Q, W)
begin
$P \leftarrow \emptyset$; $s \leftarrow r$;
while there is a path in $W-P$ from s to a vertex in Q **do**
begin
Let $su_1 \ldots u_k$ be the shortest path in $W-P$ from s to Q;
Suppose $u_k \in Q_i$, and $Q_i = v_1 q' u_k q'' v_2$ with $|v_1 q'| \leqq |q'' v_2|$;
$P \leftarrow P s u_1 \ldots u_k q''$;
$Q \leftarrow v_1 q'$;
$s \leftarrow v_2$;
end
return P ;
end.

Each iteration of the while loop halves the length of some path in Q, so if there are initially c paths, there can be at most $c \log n$ iterations. Each of the steps within the while loop (such as finding shortest paths) can be done in $O(\log^2 n)$ time, so the total time is $O(c \log^3 n)$.

A set Q of vertex disjoint paths *separates* the graph if $V-Q$ has at least two connected components. To show how to construct a splitting path from a set of paths that separate the graph, we begin with the special case where we have a single path which contains all the neighbors of a vertex.

Lemma 2.3. *Let Q_1 be a path and v a vertex different from r that has all of its neighbors on Q_1. There is an $\mathcal{NC}$ algorithm that finds either a splitting path or a maximal path.*

Proof. If v lies on Q_1, let $Q = \{Q', Q''\}$ be the two segments formed by removing v from Q_1, otherwise let $Q = \{Q_1\}$. Construct a path $P = ru_1 \ldots u_k$ that is maximal with respect to Q in $V-v$ by calling *Extend Path* $(r, Q, V-v)$. There are four cases to consider.

1) Suppose u_k is not adjacent to v. If there was a path from u_k to v that contained no nodes of P other than u_k then P could be extended to contain an additional neighbor of v without including v. Hence the component of $V-P$ that contains v is not adjacent to u_k, so P is either maximal, or $V-P$ has more than one component.

Assume that u_k is a neighbor of v and let $P' = Pv$.

2) If all neighbors of v are on P, then P' is a maximal path.
3) If $V-P'$ has more than one component then P' is a splitting path.
4) The remaining case is illustrated below. All nodes in $V-P'$ lie in a single component C that contains a neighbor of v. Since P was maximal with respect to Q, there cannot be an edge from C to u_k. Let u_j be the last vertex on P adjacent to a node in C (there must be one since we are assuming that the graph is biconnected). A maximal path can be formed by going from r to u_j, then into C and back up the path through v to u_{j+1}. ∎

Theorem 2.4. *Let $Q = \{Q_1, \ldots, Q_k\}$ be a set of vertex disjoint paths where $k \leqq c$ for some fixed constant c. If $V-Q$ has more than one connected component then a splitting path or a maximal path can be found by an $\mathcal{NC}$ algorithm.*

Proof. Use *Extend Path* to construct a path P that is maximal with respect to Q. Suppose that P is not a splitting path, so that $V-P$ has a single connected component

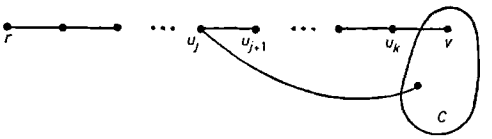

Fig. 1

C. If $Q \not\subseteq P$, then C is not adjacent to the last vertex of P, so P must be a maximal path. Assume that $Q \subseteq P$ and let x_1 and x_2 be vertices in different components of $V-Q$. If both x_1 and x_2 had neighbors in C, then there would be a path from x_1 to x_2 with all of its interior vertices in C, but this would mean that there was a vertex of Q in C. Hence, either x_1 or x_2 has all its neighbors on P. The previous lemma then shows that a splitting path or a maximal path can be constructed. ∎

3. Finding Isolated Paths

In this section we show how to construct a small set of paths that separates the graph. The paths that we construct will be *isolated.*

Definition 3.1. A set of vertex disjoint paths $Q=\{Q_1, ..., Q_k\}$ is *isolated* if every path between endpoints of different paths has at least one interior vertex in Q.

A set of at least two isolated paths can be transformed into a set of paths that separates the graph. Let $Q=\{Q_1, ..., Q_k\}$ be a set of isolated paths, and let x_1 be an endpoint of the path Q_1 and x_2 be an endpoint of Q_2. The paths $Q'=\{Q_1-x_1, Q_2-x_2, Q_3, ..., Q_k\}$ separate the graph with x_1 and x_2 in different componetns of $V-Q'$.

A set of isolated paths can be constructed from a set of paths by repeatedly combining paths. A maximum set of disjoint paths is found between the endpoints of the paths. If a path P is found between endpoints of Q_i and Q_j, then the paths Q_i, P and Q_j are joined to form a single path. Phases of joining paths are repeated until no more paths can be joined. The routine *Join Paths* constructs a set of isolated paths. The input to *Join Paths* is a set of vertex disjoint paths $Q=\{Q_1, ..., Q_k\}$ and a set of vertices T not on the paths. *Join Paths* constructs a set of isolated paths $Q'=\{Q_1, ..., Q_j\}$. Every path in Q is a segment of some path in Q'.

Join Paths (Q, T)
begin
while the paths in Q are not isolated **do**
begin
Construct an auxiliary graph G' with vertices x_i for each $Q_i \in Q$, and vertices v_i for each $v_i \in T$. The edge (v_i, v_j) is in G' if (v_i, v_j) is an edge in the original graph; (x_i, v_j) is in G' if there is an edge from an endpoint of Q_i to v_j, and (x_i, x_j) is in G' if there is an edge between endpoints of Q_i and Q_j, $i \neq j$;
Find a *Maximum Set of Disjoint Paths* $P=\{P_1, ..., P_j\}$ in G' with their endpoints in $\{x_1, ..., x_k\}$;

for each $P_l \in P$ **do in parallel**
begin
Suppose $P_l = x_i P_l' = x_j$ with $i < j$;
$Q_i \leftarrow Q_i P_l' Q_j$, (joined appropriately)
$Q_j \leftarrow \emptyset$;
$T \leftarrow T - P_l'$;
end
end

The next section shows how a maximum set of disjoint paths can be computed in $\mathscr{RNC}$ by using matching. The manipulation of paths and the construction of the auxiliary graph can be done easily in $\mathscr{NC}$. To show that *Join Paths* is an $\mathscr{RNC}$ algorithm, we show that the number of phases (i.e. iterations of the outer loop), is $O(\log n)$.

Lemma 3.2. *The process of joining the paths* $\{Q_1, \ldots, Q_m\}$ *requires* $O(\log m)$ *phases.*

Proof. Suppose k joins are performed at phase j. Any subsequent join must involve at least one unjoined endpoint of a path joined at phase j. Thus there will be at most $2k$ subsequent joins. The number of paths that are joined at a phase is no greater than the number joined the previous phase. Therefore, there are at most $k/2$ joins at phase $j+4$. Because there are at most $m/2$ joins during the first phase, there will be $O(\log m)$ phases. ∎

In the maximal path algorithm, it will be important to insure that the joining process will not result in a single path. The following lemma gives a simple case where the joining process will not form a single path.

Lemma 3.3. *If at most* $m/3-1$ *joins are performed in the first phase of the joining process, then there will be at least* 2 *paths left when the joining process is done.*

Proof. The total number of joins performed is at most $m/3-1+2(m/3-1)=m-3$. The number of joins required to combine the paths into a single path is $m-1$. ∎

The set of paths joined in a single phase depends upon the particular set of disjoint paths found by the probabilistic matching algorithm. The number of paths joined in a phase is however a fixed number, independent of the probabilistic choices made. We denote the number of paths joined in the first phase of *Join Paths* by $J(Q_1, \ldots, Q_k)$. The *Join Paths* procedure does not actually require finding a *maximum* set of disjoint paths to construct isolated paths, it is sufficient to find a *maximal* set of disjoint paths. However, for technical reasons explained later, our maximal path algorithm relies on the fact that *Join Paths* does find a maximum set of disjoint paths. A second reason for using a maximum set of disjoint paths is that it is not known how to construct a maximal set of disjoint paths without constructing a maximum set of disjoint paths.

A major portion of the maximal path algorithm is to construct a small set of isolated paths. We construct a set of isolated paths $Q=\{Q_1, \ldots, Q_k\}$ where $2 \leqq k \leqq 5$. An initial set of isolated paths is constructed by calling *Join Paths* $(V, \emptyset)$ (treating each vertex as a path of length zero). If at most five paths are constructed a splitting path can be found directly, otherwise the number of paths is reduced. The basic idea

is to discard some of the paths and use the nodes of the discarded paths as well as the nodes not on any of the paths to joint the remaining paths. This process is repeated until between two and five isolated paths remain. It is easy to reduce the number of paths significantly with a phase, since any number of paths may be discarded. It is necessary to make sure that at least two paths remain when the paths are joined.

The algorithm for reducing the number of paths works in phases, where each phase reduces the number of paths on hand by at least a factor of 1/4. A phase starts with a set of isolated paths $Q=\{Q_1, \ldots, Q_m\}$. To reduce the number of paths in Q, we choose a suitable k and replace Q by $\{Q_1, \ldots, Q_k\}$ and join the paths in Q. The first value of k that we try is $k=3m/4$. If $J(Q_1, \ldots, Q_{3m/4})<m/4-1$, then Lemma 2 guarantees that the paths won't collapse to a single path. Suppose $J(Q_1, \ldots, \ldots, Q_{3m/4})\geqq m/4$. Since $J(Q_1, \ldots, Q_m)=0$, there is a $k>3m/4$ such that $J(Q_1, \ldots, \ldots, Q_k)\geqq m/4-1$ and $J(Q_1, \ldots, Q_{k+1})<m/4-1$. This value can be found by checking all values of k in parallel. This value of k still might not be satisfactory, since joining $Q_1, \ldots, Q_k$ could cause the paths to collapse to a single path, while joining $Q_1, \ldots, Q_{k+1}$ might not give enough reduction. If this is the case, we find a segment P of Q_{k+1} such that $m/4-2\leqq J(Q_1, \ldots, Q_k, P)\leqq m/4-1$. If we remove a node from Q_{k+1}, we change the number of joins in the first phase of *Join Paths* by at most two. This is because we change the auxiliary graph used in *Join Paths* by two nodes. When finding a maximum set of disjoint paths in a graph that has been altered in this way, the number of paths found changes by at most two. This "continuity" property is why we chose to use a maximum set of disjoint paths instead of settling for maximal. We can find this segment P by testing each of the initial segments of Q_{k+1} in parallel. When we join paths, we guarantee that the number of paths is reduced by at least $m/4-2$ since the first phase will perform at least that many joins.

Reduce Paths $(Q=\{Q_1, \ldots, Q_m\}, V')$
begin
  if $J(\{Q_1, \ldots, Q_{3m/4}\})<m/4-1$ **then**
    Join Paths $(\{Q_1, \ldots, Q_{3m/4}\}, V'\cup\{Q_{3m/4+1}, \ldots, Q_m\})$;
  else
    begin
      Find a $k>3m/4$ such that $J(Q_1, \ldots, Q_k)\geqq m/4-1$ and $J(Q_1, \ldots, Q_{k+1})<$
        $<m/4-1$;
      Suppose $Q_{k+1}=p_1 \ldots p_j$;
  Find an initial segment $P=p_1 \ldots p_{j'}$ of Q_{k+1} such that
        $m/4-2\leqq J(Q_1, \ldots, Q_k, P)\leqq m/4-1$;
      Join Paths $(\{Q_1, \ldots, Q_k, P\}, V'\cup\{Q_{k+2}, \ldots, Q_m\}\cup\{p_{j'+1} \ldots p_j\})$;
    end
end.

4. Finding a Maximum Set of Disjoint Paths

The Maximum Set of Disjoint Paths Problem (MDP) is: Given a graph $G=(V, E)$ and a set of vertices U, find a maximum cardinality set of non-trivial vertex disjoint paths that have their endpoints in U. Finding disjoint paths is the central step in joining paths. In this section we show that MDP is in $\mathscr{RNC}$ by reducing it to matching. The reduction constructs a graph $G'=(V', E')$ that has a

complete matching if and only if k pairs of vertices of U can be joined by vertex disjoint paths. The paths can be identified directly from a complete matching. To find a maximum set of disjoint paths, each value of k between 1 and $1/2|U|$ is tested independently in parallel.

The construction is relatively simple. Suppose we want to find k disjoint paths. The set of vertices V' is made up of three subsets: X, Y and Z. The vertices X correspond to vertices of U, the vertices Y correspond to vertices of $V-U$, and the vertices Z are used to determine the number of disjoint paths. For a vertex $u_i \in U$ there is a vertex $x_i \in X$. For a vertex $v_i \in V-U$, there are vertices $y_i', y_i'' \in Y$ with an edge $(y_i', y_i'') \in E'$. There are also edges in E' that correspond to the edges of E. If $(u_i, u_j) \in E$, then $(x_i, x_j) \in E'$ if $(u_i, v_j) \in E$, then $(x_i, y_j'), (x_i, y_j'') \in E'$, and if $(v_i, v_j) \in E$, then $(y_i', y_j''), (y_i'', y_j') \in E'$. The set Z consists of $|U|-2k$ vertices $Z_1, \ldots, Z_{|U|-2k}$. There is a complete bipartite graph between X and Z, so there are edges $(x_i, z_j) \in E'$ for $i=1, \ldots, |U|$ and $j=1, \ldots, |U|-2k$.

We now prove that this construction allows us to find vertex disjoint paths.

Theorem 4.1. *G' has a complete matching if and only if k pairs of vertices of U can be joined by vertex disjoint paths.*

Proof. We first show that we can construct a perfect matching from a set of vertex disjoint paths joining vertices of U. We assume that no vertices of U are interior vertices of the paths.

Since there are $|U|-2k$ vertices of U not on paths, the vertices $x_i \in X$ corresponding to verices not on paths are matched with the vertices of Z. For vertices $v_i \in V-U$, with v_i not on a path, the vertices y_i' and y_i'' are matched. The only vertices not matched so far are the ones corresponding to path vertices. For the path $u_{i_1} v_{j_1} \ldots v_{j_s} u_{i_2}$, x_{i_1} is matched with y_{j_1}', x_{i_2} is matched with y_{j_s}'' and y_{j_r}'' is matched with $y_{j_{r+1}}'$ for $r=1, \ldots, s-1$.

For the other direction, assume that the graph has a complete matching M. Let M' be the partial matching consisting of all edges of the form (y_i', y_i''). Consider the graph $(V', M \oplus M')$, where $\oplus$ denotes the symetric difference, i.e., the edges in exactly one of M or M'. The vertices of Y have degree 0 or 2, while the vertices of X and Z have degree 1. The connected components of this graph are either paths or cycles. There are $|U|-2k$ matched pairs of vertices between X and Z. Thus the remaining $2k$ vertices of X must be endpoints of paths, so there are k pairs of vertex disjoint paths. ∎

5. Depth First Search

In this section a parallel algorithm for depth first search is described. The depth first search problem is: Given a graph $G=(V, E)$ and a vertex $r \in V$, find a spanning tree T of G that could be constructed by some depth first search of G with root r. The algorithm runs in $O(\sqrt{n} \log^k n)$ and uses a polynomial number of processors. The original motivation for looking at the maximal path problem was its relation to depth first search. Any branch from the root to a leaf in a depth first search tree is a maximal path. The maximal path algorithm does not apply directly to give a fast parallel algorithm for depth first search, since $O(n)$ maximal paths might be necessary to

construct a depth first search tree. However, the general techniques and tools developed for the maximal path problem are used for depth first search.

The depth first search algorithm uses the same strategy as was used for finding a maximal path. A partial solution is found that allows the problem to be reduced to smaller problems which are solved recursively. The algorithm finds a set Q of disjoint paths such that the size of the largest connected component of $V-Q$ is less than $\frac{1}{2}n$. These paths will be referred to as a *separating set* of pahts. Note that this usage of the term *seperate* differs from our usage in the description of the maximal path algorithm. An initial segment of a depth first search tree containing Q is constructed. Depth first search trees are then found for the remaining components. Since the problem is halved at each level, the depth of recursion is at most $\log n$. The procedures for finding separating paths and constructing an initial segment both take $O(\sqrt{n}\log^k n)$ time. We first describe the construction of the initial segment and then discuss the more complicated step of finding the separating paths.

The routine *Initial Segment* is given a set of disjoint paths Q and constructs a subtree T' of some depth first serach tree T with all the vertices of Q contained in T'. In the depth first search algorithm, there will be $O(\sqrt{n}\log n)$ paths in Q when *Initial Segment* is called. *Initial Segment* is essentially a sequential algorithm; however it uses the parallel routine *Extend Path* described above. *Initial Segment* maintains the connected components of the nodes not in the subtree T'. A component is said to be *active* if it contains a path from the set Q.

```
InitialSegment (Q, r)
  begin
    T' ← r;
    while there is an active component of V−T' do
      begin
        Let v be the lowest node on T' adjacent to an active component C of V−T';
        P ← Extend Path (v, Q, C);
        Add P to T';
        Recompute the connected components of V−T';
      end
  end.
```

Each phase of the routine *Extend Path* reduces the length of some path in Q by a factor of at least one half. If there are initially m paths in the set Q, then *Initial Segment* will take $O(m\log^k n)$ time.

Lemma 5.1. *The tree T' constructed by Initial Segment can be extended to a depth first search tree.*

Proof. It suffices to show that there are no pahs between separate branches of T' that have all their interior nodes in $V-T'$. This condition holds throughout the execution of *Initial Segment* since the extensions are made at the lowest node adjacent to some component. ∎

We now show how to find a separating set of paths. We construct a set Q of disjoint paths where Q contains $O(\sqrt{n}\log n)$ paths and the largest component of

$V-Q$ has size at most $n/2$. The procedure will run in $O(\sqrt{n}\log^k n)$ time. The routine *Separate* constructs a set of paths and then attempts to reduce the number of paths while keeping the connected components of the nodes not on the paths small. The paths will be reduced by joining them together or by removing nodes from them. The routine maintains several sets of paths. The set Q contains paths that will be in the separator. Once a path is put into Q it is committed to the separating set and will not be removed. The set S stores the paths that are currently being worked on. The set R is used for temporary storage; paths are put into R to set them aside for the next phase. The nodes not on any of the paths in Q, R, or S are in a set T. The size of the largest component of T is at most $n/2$. The algorithm runs until R and S are empty.

In *Separate* an iteration of the outer while loop is referred to as a phase and an iteration of the inner while loop is a subphase. Each phase halves the number of paths in S. A subphase reduces the length of each path in S by one. Subphases are repeated until all of the nodes in paths of S have been moved to R or T. The paths of R are then moved back to S for the next phase. Phases are repeated until R and S are empty. In the routine below, *Join* (S, T) is a procedure that performs the first phase of *Join Paths* (S, T). *Join* (S, T) finds a maximum set of vertex disjoint paths in T between the endpoints of the paths in S. The paths in S are then joined using the disjoint paths that were found.

```
Separate
  begin
    Q←∅; R←∅; T←∅;
    S←V;
    while S≠∅ do
      begin
        while S≠∅ do
          begin
            Join (S, T);
            Move the joined paths from S to R;
            Move one endpoint of each path in S to T;
            if there is a component in T of size ≧n/2 then
              Fix the component size by moving a node from T to Q;
          end
        S←R; R←∅;
        Move paths of length ≧√n from S to Q;
      end
  end.
```

The goal of the routine *Separate* is to remove all the paths from S and R while making sure that the largest connected component of T remains of size at most $n/2$. *Separate* accomplishes this by alternately joining paths of S and by moving some nodes from paths in S to T. The paths are removed in two different ways. When paths are joined, any paths of length at least $\sqrt{n}$ are put into Q. The other way paths can be removed is if all of their nodes are put into the set T.

A subphase begins by joining as many paths as possible using the nodes of T. The paths of S that are joined are then put into R and set aside until the next phase.

The next step is to move one endpoint of each of the paths remaining in S to T. The moving of endpoints accomplishes a dual purpose; it makes new nodes of S endpoints, allowing additional joins in the next subphase, and reduces the lengths of the paths in S. Before the endpoints are removed, each connected component is adjacent to the endpoints of at most one path. This means that when the endpoints are moved from S to T, the only way components of T are merged is if they are adjacent to the endpoints of the same path in S. At most one component of size at least $n/2$ can be formed each subphase. If a large component is formed, then removing the endpoint that caused it to merge will reduce the components to size less than $n/2$. When the node is removed, it is placed into Q as a path of length one.

The following lemmas establish that *Separate* constructs the desired separator in $O(\sqrt{n} \log n)$ time.

Lemma 5.2. *The largest connected component of T has size at most $n/2$.*

Proof. The only time nodes are added to T is when endpoints of paths in S are moved to T. When this is done, if a component of size greater than $n/2$ is formed, a node can be moved from T to Q to reduce the components to size at most $n/2$. ∎

Lemma 5.3. *There are at most $\sqrt{n}$ subphases per phase.*

Proof. At the start of a phase, the paths in S have length at most $\sqrt{n}$. Each subphase reduces the lengths of all the paths remaining in S by one. ∎

Lemma 5.4. *The number of phases is at most $\log n$.*

Proof. Each path at the start of a phase is the join of two paths from the previous phase, so the number of paths in S is halved by each phase. ∎

Lemma 5.5. *When Separate is finished, there are $O(\sqrt{n} \log n)$ paths in Q.*

Proof. At most $\sqrt{n}$ paths of length $\sqrt{n}$ can be placed in Q. Each subphase places at most one singleton in Q. Since there are at most $\sqrt{n} \log n$ subphases altogether, there are $O(\sqrt{n} \log n)$ paths in Q. ∎

6. Conclusions and Open Problems

In this paper, we have presented parallel algorithms for finding a maximal path and for computing a depth first search tree. Both of these problems appear to be very sequential in nature, so it is interesting that a substantial speed up is possible using parallelism.

The runtime for the maximal path algorithm is $O(\log^5 n)$. The algorithm has three levels of recursion or iteration, each requiring $O(\log n)$ phases. Gabow [5] has observed that one of these levels can be removed by reorganizing the algorithm, reducing the runtime by a factor of $\log n$. The inner loop involves finding a complete matching, which can be done in $O(\log^2 n)$ time [8], giving the $O(\log^5 n)$ time bound. As described, the number of processors used is n^2 times the number of processors

required for matching. This number can be reduced at the expense of a small increase in the runtime.

An open question is whether fast deterministic algorithms can be found for these problems. An $\mathcal{NC}$ algorithm is known for the maximal path problem if all vertices have low degree [2], so it would be interesting to know if the general problem could also be solved in $\mathcal{NC}$. It would also be interesting to find a solution to these problems that does not involve matching. The number of processors used by our algorithms is very large (although polynomial); a final question is whether the processor requirement can be reduced to a more reasonable number.

Acknowledgement. I wish to thank Ernst Mayr for many very helpful discussions. I also with to thank Joan Feigenbaum, Peter Hochschild, Anna Karlin, Yoram Moses and Alexandro Schaeffer for suggestions which have greatly improved this paper.

References

[1] R. ANDERSON and E. MAYR, Parallelism and greedy algorithms, *Technical Report No. STAN CS-84-1003, Computer Science Department, Stanford University,* April 1984.
[2] R. ANDERSON and E. MAYR, Parallelism and the maximal path problem, *Information Processing Letters,* **24** (1987), 121—126.
[3] S. A. COOK, A taxonomy of problems with fast parallel algorithms, *Information and Control,* **64** (1985), 2—22.
[4] S. FORTUNE and J. WILLIE, Parallelism in random access machines, *Proc. 10th ACM STOC* (1978), 114-118.
[5] H. GABOW, *Private communication,* 1985.
[6] R. M. KARP, E. UPFAL and A. WIGDERSON, Constructing a maximum matching is in random NC, *Combinatorica,* **6** (1986), 35—48.
[7] R. M. KARP and A. WIGDERSON, A fast parallel algorithm for the maximal independent set problem, *JACM,* **32** (4) (1985), 762—773.
[8] K. MULMULEY, U. VAZARANI and V. VAZARANI, Matching is as easy as matrix inversion, *Combinatorica,* **7** (1987), 105—113.
[9] J. REIF. Depth first search is inherently sequential, *Information Processing Letters,* **20** (1985), 229—234.
[10] R. E. TARJÁN and U. VISHKIN, Finding biconnected components and computing tree function in logarithmic parallel time, *Proc. 25th FOCS,* (1984) 12—20.
[11] U. VISHKIN, An optimal parallel connectivity algorithm, *RC 9149, IBM T. J. Watson Research Center, Yorktown Heights, N. Y.,* 1981.

Richard Anderson

Department of Computer Science
University of Washington
Seattle, Washington, 98195