December 1988 Report No. STAN-CS-88-1239

Sorting, Minimal Feedback Sets and Hamilton Paths in Tournaments

by

Amotz Bar-Noy and Joseph Naor

Department of Computer Science

Stanford University

Stanford, California 94305

Sorting, Minimal Feedback Sets and Hamilton Paths in Tournaments

Amotz Bar-Noy * *Joseph Naor* †

Computer Science Department,
Stanford University,
Stanford, CA 94305.

December 15, 1988

Abstract

We present a. general method for translating sorting by comparisons algorithms to algorithms that compute a Hamilton path in a. tournament. The translation is based on the relation between minimal feedback sets and Hamilton paths in tournaments. We prove that there is a one to one correspondence between the set of minimal feedback sets and the set of Hamilton paths. In the comparison model, all the tradeoffs for sorting between the number of processors and the number of rounds hold when a. Hamilton path is computed. For the CRCW model, with $O(n)$ processors, we show the following:(i) Two paths in a tournament can be merged in $O(\log \log n)$ time (Valiant's algorithm [Va]); (ii) a Hamilton path can be computed in $O(\log n)$ time (Cole's algorithm). This improves a previous algorithm for computing a Hamilton path whose running time was $O(\log^2 n)$ using $O(n^2)$ processors.

*Supported in part by a Weizmann fellowship and by contract. ONR N00014-88-K-0166.
†Supported by contract ONR N00014-88-K-0166.

1 Introduction

A tournament $T = (V, D)$ ($|V| = n$) is a directed graph in which every pair of vertices is joined by a directed edge. It can be viewed as a complete graph whose edges are given an orientation. A Hamilton path in a graph is a simple path that contains all the vertices, and each vertex appears exactly once. A well known theorem states that there is a. Hamilton path in every tournament [Re, Be].

In this paper we investigate the complexity of computing a Hamilton path in a tournament and other problems related to it. Our methodology relies on the intimate relationship that exists between Hamilton paths and minimal feedback sets in tournaments, and their connection to sorting algorithms. Sorting by comparisons may be viewed as computing a Hamilton path in a **transitive (acyclic)** tournament. The purpose of this paper is to show the opposite direction, namely how sorting algorithms can be generalized to compute a Hamilton path in an arbitrary tournament.

Sorting by comparisons is a. well investigated problem, perhaps the most in computer science. We show how to exploit the wealth of results on it to design parallel and sequential algorithms for arbitrary tournaments.

Parallel algorith ms to compute a Hamilton path in a tournament have appeared in [Na, So], but were designed by ad **hoc** principles. The key idea. in computing the Hamilton path in both papers is the following: in every tournament there exists a vertex **(separator)** whose indegree and ou tdegree are bounded from below by $|V|/4$; this gives a recursive formulation of the problem with only a. logarithmic number of steps. The difficulty with this approach is that. the best bound known for finding a separator requires $0(n^2)$ processors. Ramachandran [Ra] showed using adversary arguments that a lower bound on the number of edges whose orientation must be known in order to find a separator is $\Omega(n^2)$.

In analogy to sorting, we define two additional problems on tournaments:

- Merging two paths such that the relative order of the vertices in the original paths remain s after t he m erge.
- k-selection, e.g. finding the kth vertex in the Hamilton path in a transitive tournament, is generalized to finding a vertex which is the kth in some Hamilton path in an arbitrary tournamen t.

We now present a summary of our method. A **minimal edge feedback set** F in a directed graph G = (V, D) is a set of edges such that G' = $(V, D - F)$ is acyclic, and F is minimal with respect to containment. We prove that there is a one to one correspondence between the set of minimal feedback sets and the set of Hamilton paths in an arbitrary tournament, and show how a minima.1 feedback set can generate a Hamilton path and vice versa,. We also

show that an edge in a transitive tournament whose orientation is known by implication, cannot appear in the Hamilton path. Assume now that the input to a sorting algorithm is not a transitive tournament but rather any arbitrary tournament.. If a. minimal feedback is computed for each round of comparisons and its orientation flipped, namely the sorting algorithm is "cheated", then the Hamilton path computed by the sorting algorithm will also be a Hamilton path for the original input. Intuitively, this happens because the edges on which we "cheated,' are actually implications. These notions are formalized and proved, and they result in a general method for translating any sorting (or sorting related) algorithm to an algorithm that computes a Hamilton path.

There are two known proofs for the existence of Hamilton paths in tournaments. One proof is Redei's proof [Re], and the other is the aforementioned separator theorem. In view of our results, the first proof corresponds to **insertion-sort,** whereas the second to **quick-sort.** In fact, our results imply that the known sequential sorting algorithms also compute a. Hamilton path in an arbitrary tournament.

The equivalence between sorting and computing a Hamilton path holds both in the comparison model and in PRAMS. Valiant's comparison model [Va] can be easily generalized to arbitrary tournaments. In this model we prove that the complexity of computing a Hamilton path in an arbitrary tournament and in a transitive one is the same. It follows that all the lower bounds, upper bounds and processor-time tradeoffs for sorting, apply also when computing a Hamilton path in an arbitrary tournament. These bounds and tradeoffs have been proved in a series of papers, [AA1, AA2, AAV, Al, AV, BHe, BHo, I']. Hell and Rosenfeld [HR] 1rave also considered the sequential complexity of algorithms in the comparison model for computing generalized. paths in tournaments.

The situation with implementing our translation method in the PRAM model is more complex. The difficulty is that it requires the computation of a minima.] feedback edge set in a. directed graph. It is not known whether an NC algorithm exists for this problem. We consider two PRAM sorting algorithms, and give for these special cases a non trivial procedure that computes a minimal feedback set in constant time. The algorithms are Cole's **merge-sort** [Co] and Valiant's **merging** algorithm [Va.], and our results are in the CRCW model. Hence, a Hamilton path can be computed in O(log n) time, and two paths of length n can be merged in O(loglog n) time; both algorithms use O(n) processors.

These two algorithms achieve an optimal speed (up to a constant factor) with respect to the sequential complexity. Notice that in our case the number of processors is linear in the number of vertices and not edges. This result is interesting by itself.

Our results also imply **an** O(n log n) sequential algorithm for computing a Hamilton path in an arbitrary tournament. As we already have mentioned, **merge-sort** will also output a.

Hamilton path for an arbitrary tournament. This bound also follows from Redei's proof [Re], but only when appropriate data structures are used. It can be proved that Batcher's sorting network [Ba.], also computes a Hamilton path when the input is an arbitrary tournament,. However? it is not obvious whether the AKS sorting network [AKS] can be adapted to tournaments. This motivates the search for a. sorting network whose depth is less than $O(log^2 n)$ and also computes Hamilton paths. It. would also imply better bounds for EREW algorithms.

As for the k-selection problem, it is easy to show a. lower bound of $\Omega(\log n/ \log \log n)$ time for any PRAM model even in the case of transitive tournaments. Hence. the best strategy (up to a factor of $O(\log \log n)$ would be first. to compute a Hamilton path, and then solve the k-selection problem. On the other hand, in the comparison model, the result of [AKSSlmp lies an $O(\log \log n)$ upper bound.

2 Terminology and preliminaries

Let $T = (V, E)$ denote a tournamen t, that is a complete graph whose edges are oriented. Let the cardinality of' the vertex set be denotetl by n. If an edge is oriented from v to w, then we say that v is **smaller** than w (w is **greater** than v) and denote it either by v < w, or by v $\rightarrow$ **w.** A **path** $v_0, v_1, \ldots, v_k$ is a. sequence of vertices where $v_i < v_{i+1}$ and $v_i \neq v_j$ for $0 \leq i \neq j \leq k$. A cycle is a path where $v_0 = v_k$. If vertex v precedes vertex w in a. path P, then $v < w$ with respect to the path P or v is **below** w (w is **above** v). The first and last vertices in the path are called **bottom** and top respectively.

A tournament is **transitive** if and only if' it is acyclic. An **implication** in a. transitive tournament is an edge whose orientation is implied by the orientation of other edges in the tournament, namely to avoid cycles.

Let P and Q be two paths in an arbitrary tournamen t and let R be the result of merging P and Q. Then this **merge** has the following property: if $v < w$ with respect to P *(Q),* then also v < w with respect to R.

An **edge feedback set** F in a directed graph $G = (V, D)$ is a set of edges such that G' = $(V, D - F)$ is acyclic. Computing such a set, of minimum cardinality is NP-complete [GJ], whereas computing a minimal such set with respect to containment can be easily done in polynomial time by a greedy algorithm. An easy property of minimal feedback sets is that the graph resulting from inserting any edge of F in G' is cyclic.

The graph G = (V, W, Q) is called a complete path directed bipartite graph (abbreviated CPB) if Q contains all the edges between V and W, and the graph induced by V (W) is a directed Hamilton path.

3 Hamilton paths, minimal feedback sets and the comparison model

In this section we show how Hamilton paths and minimal feedback sets are related to each other in tournaments. We first extend the parallel comparison model to tournaments. This model was first, introduced by Valiant [Va] for the purpose of analyzing sorting algorithms; only comparisons are taken into account in it, whereas internal processor computation and communication are not charged for. The structure of an algorithm in this model is the following: in each round a. set of element pairs are compared until the output is known. The aim of an algorithm is to minimize both the number of rounds and the total number of comparisons.

We extend this model for tournaments in the natural way. The answer to a comparison is the orientation of an edge in the tournament. Hence, in the beginning we have a tournament whose edge orientation is unknown, and at each round, we ask for the orientation of a. set of edges. The algorithm proceeds till the induced graph of the known edges contains the solution. Let $f(p, k)$ denote the minimum number of comparisons needed to compute a function f in k rounds and p processors. As we already have mentioned, sorting can be viewed as computing a. Hamilton path in a transitive tournament.

The next easy lemma. is used by our main theorem that follows immediately.

Lemma 3.1 **Let $e = u \rightarrow v$ be an edge in a transitive tournament whose direction is known by implication. Then e cannot appear on the Hamilton path.**

Proof: If the orientation of e is known by implication then there exists an element w such that u $\rightarrow w$ and w $\rightarrow v$. Hence? e cannot appear on the Hamilton path. ∎

Theorem 3.1 **Let $\mathcal{A}$ be an algorithm that computes a Hamilton path in a transitive tournament with complexity $H(p, k)$. Then there exists an algorithm $\mathcal{B}$ that computes a Hamilton path in a non transitive tournament T with the same complexity $H(p, k)$.**

Proof: Let $Q_1, \ldots, Q_k$ be disjoint. sets of directed edges and let $F = F_1 \cup F_2 \cup \ldots \cup F_k$ be a. set with the followiug properties:

1. $F_i \subseteq Q_i$.
2. F_1 is a. minima.1 feedback set in the graph induced by Q_1.
3. Inductively, F_i is a minimal feedback set in the graph induced by $(Q_1 - F_1) \cup (Q_2 - F_2) \cup \cdots \cup (Q_{i-1} - F_{i-1}) \cup Q_i$.

It is easy to verify that for every i, $F^i = F_1 \cup \ldots \cup F_i$ is a minimal feedback in the graph induced by $Q_1 \cup \cdots \cup Q_i$.

Denote by $\neg F_i$ the set of edges of F_i such that the orientation of each edge in F_i is flipped and let. $Q'_i = Q_i - F_i \cup \neg F_i$. Define $\mathcal{A}(R_1, \ldots, R_i)$ to be the set of comparisons in round $i + 1$ of algorithm $\mathcal{A}$ under the assumption that the comparisons in the first i rounds were $RI,\ldots, R;$. With these notations, the set of comparisons of Algorithm $\mathcal{B}, Q_1, \ldots, Q_k$, will be derived from algorithm $\mathcal{A}$ in the following way:

1. $Q_1 \leftarrow \mathcal{A}(\emptyset)$
2. $Q_{i+1} \leftarrow \mathcal{A}(Q'_1, \ldots, Q'_i)$

First we show that Algorithm $\mathcal{B}$ is well defined by proving that $H_i = Q'_1 \cup \cdots \cup Q'_{i-1}$ is a legal input to round i of algorithm $\mathcal{A}$. It is easy to verify that H_i is a. legal input if it is acyclic. Assume to the contrary that there is a cycle c in it. The edges of H_i are of two types: edges of the minima1 feedback set F^i that were flipped, or unflipped edges. By the definition of a. minimal feedback set, for every edge $e = (u \rightarrow v) \in F^i$, there is a path p_e of unflipped edges from v to u. Now, exchange every edge $e \in$ c that belongs to F^i by p_e and get as a. result a. cycle of edges that were not flipped. If that cycle is not simple, it contains a simple cycle as a. subgraph, hence contradicting F^i being a minimal feedback set.

Now that Algorithm $\mathcal{B}$ is well defined, assume to the contrary that its output p is not a. Hamilton path in I'. The path p is not a Hamilton path only if it contains edges whose orientation was flipped, namely edges that belong to F, the minimal feedback set.. This cannot happen as the edges of F are implications in the graph induced by H_k, and therefore cannot appear on the Hamilton path by the lemma 3.1. ∎

Corollary 3.1 **Let $\mathcal{A}$ be an algorithm that merges two paths in a transitive tournament with complexity $M(p, k)$. Then there exists an algorithm $\mathcal{B}$ that merges two paths in a non transitive tournament with the same complexity.** ∎

Another problem that was extensively studied in transitive tournaments is the k-selection problem: find an element larger than $k - 1$ elements and smaller than $n - k$ elements. This problem can be generalized to non transitive tournaments in two ways. The first one is to find an element in the tournament (if one exists) whose indegree is exactly equal to k. It can be shown that the minimum number of comparisons needed to determine such an element is $\Omega(n^2)$ [Ra.], and therefore the running time cannot be polylogarithmic with a linear number of processors.

A more relaxed definition is to find an element v in the tournament such that v is the k-th element in some Hamilton path. We show that the complexity of this problem is equivalent to the complexity of k-selection in the transitive case.

Corollary 3.2 **Let $\mathcal{A}$ be an algorithm that solves the k-selection problem in transitive tournaments with complexity $S(p, k)$. Then there exists an algorithm $\mathcal{B}$ that solves the &selection problem in non transitive tournaments with the same complexity.**

Proof: Algorithm $\mathcal{A}$ is translated into $\mathcal{B}$ in the same way as in Theorem 3.1. The corollary follows from the observation that any k-selection algorithm can be viewed as a step in a sorting algorithm. ▮

It follows from Corollary 3.2 that the &selection algorithm of [AICSS] in the comparison model whose time complexity is O(loglog n) with a. linear number of processors, can be applied to arbitrary tournaments as well. On the other hand, in the PRAM model, we have the following theorem that implies that the best strategy for k-selection, (up to a factor of O(loglog n)), would be first to compute a Hamilton path.

Theorem 3.2 **There is a lower bound of $\Omega(\log n/ \log\log n)$ on the complexity of the k-selection problem for transitive tournaments in the PRAM model.**

Proof: This follows easily from the lower bound of [Bea] on computing the exclusive OR of a bit vector, as it also implies a lower bound of $\Omega(\log n/ \log\log n)$ on sorting. Assume there was a k-selection algorithm whose time complexity was better than $O(\log n)$. Invoking this . algorithm n times simultaneously for $k = 1, 2, \ldots, n$ would imply a better bound for sorting. Hence, a. contradiction. ▮

Another consequence of Theorem 3.1 is the following relation between Hamilton paths and minimal feedback sets.

Theorem 3.3 **Let $\mathcal{P}$ be the set of all Hamilton paths in a tournament T and let $\mathcal{F}$ be the set of all minimal feedback sets in a tournament T. There is a one to one correspondence between $\mathcal{P}$ and $\mathcal{F}$.**

Proof: We first show how a minimal feedback set F can **be** computed from a given Hamilton path p. For every pair of vertices v, **w** such that v precedes w in the path, add the edge (v, w) to F if it is oriented from w to v. Obviously, F is a minimal feedback set and two different paths cannot generate the same minimal feedback set. We now prove the other direction. Let F be a given minimal feedback set in a tournament T and let T' be the tournament in which the orientation of the edges in F were flipped. By Theorem 3.1, a. Hamilton path in T' is also a Hamilton path in T. let F_1 and F_2 be two minimal feedback sets and let v $\rightarrow$ w be an edge in F_1 and not in F_2. F_1 and F_2 cannot generate the same path as T_1' will contain v $\leftarrow$ w and T_2' will contain v $\rightarrow$ w. ▮

It follows from Theorem 3.3 that all the results on the cardinality of $\mathcal{P}$ [MO] apply to $\mathcal{F}$ as well. A criterion to decide whether there is a path in a tournament that starts at a given vertex v and ends at some other given vertex w also follows from Theorem 3.3. Let

in, (out_v) be the set of incoming (outcoming) edges into v (out of v).

Corollary 3.3 A necessary and sufficient condition for the existence of a Hamilton path from v to w is the existence of a minimal feedback set containing both in_v and out_w.

Proof: Assume that such a minima.1 feedback set exists. After its edges are flipped, v becomes a source and w a sink and hence, according to Theorem 3.1, there is a Hamilton path from v to w. If a Hamilton path from v to w exists, then in, and out_w belong to its corresponding minimal feedback set. (Same construction as in Theorem 3.2). ▌

4 The CRCW model

In this section we exhibit algorithms for sorting and merging in an arbitrary tournament that are based on Theorem 3.1. The complexity of these algorithms matches the complexity of the corresponding algorithms in transitive tournaments.

In the first subsection we present Algorithm MFS that in O(1) time finds a. minimal feedback set in a. CPB graph G = (V, W, E). In the other two subsections, we show how to translate certain algorithms for sorting and merging in transitive tournaments, to algorithms in arbitrary tournaments by calling MFS as a subroutine. It turns out that computing a minimal feedback set in our special cases, reduces to computing it in a CPB graph. We shall prove that in each case, the feedback set computed by repeated calls to Algorithm MFS is indeed the minimal feedback set required by Theorem 3.1. The algorithms chosen are (they both use $O(n)$ processors):

Merging: Valiant's merging algorithm [Va] with $O(\log \log n)$ time complexity.

Sorting : Merge-Sort [Co] wit h O (log n) time complexity.

4.1 Finding the minimal feedback set

Algorithm MFS invokes the two procedures MAXINDEX and MAXPREFIX. We now present them and prove that each can be implemented in O(1) time.

Procedure MAXINDEX(A):

input: A binary sequence $A = a_1 . . . . , a_\ell$; ℓ processors.

output: The maximum index k, $1 \leq k \leq \ell$, such that $a_k = 1$.

Lemma 4.1 MAXINDEX **can be implemented in the CRCW model in** $O(1)$ **time.**

Proof: It is well known that computing the OR (respectively, AND) of a. binary sequence of length ℓ with ℓ processors takes O(1) time in the CRCW model.

First we show how to compute MAXINDEX in O(1) time with ℓ^2 processors. Each element of A such that a; = 1 computes $b_i = \max_{i<j\leq\ell} a(j)$ in O(1) time. (This is possible as each a_i has ℓ processors available). If b_i = 1, then there exists an index j > i such that a_j = 1, and therefore i cannot be the desired answer. In this case we set a_i = 0 and clearly, only one value remains equal to 1. The index of this value is the answer.

Now we show how to compute MAXINDEX in O(1) time in the CRCW model with only ℓ processors. (For clarity, assume that $\sqrt{\ell}$ is an integer.)

1. Divide the ℓ elements of the sequence into $\sqrt{\ell}$ blocks where each contains $\sqrt{\ell}$ consecutive items.
2. For $1 \leq i \leq \sqrt{\ell}$, let c_i denote the OR of the i-th block.
3. Let k be the output of Procedure MAXINDEX when the input is the vector $c_1, \ldots, c_{\sqrt{\ell}}$. This takes 0(1) time as there are ℓ processors and the vector is of length $\sqrt{\ell}$.
4. Let k' be the output of Procedure MAXINDEX when the input is the k-th block. This takes $O(1)$ time as there are ℓ processors and the block is of length $\sqrt{\ell}$.
5. The output is k'.

∎

Procedure MAXPREFIX(A, B**):**

input: A sequence $A = a_1, \ldots, a_\ell$ of integers; ℓ^2 processors.

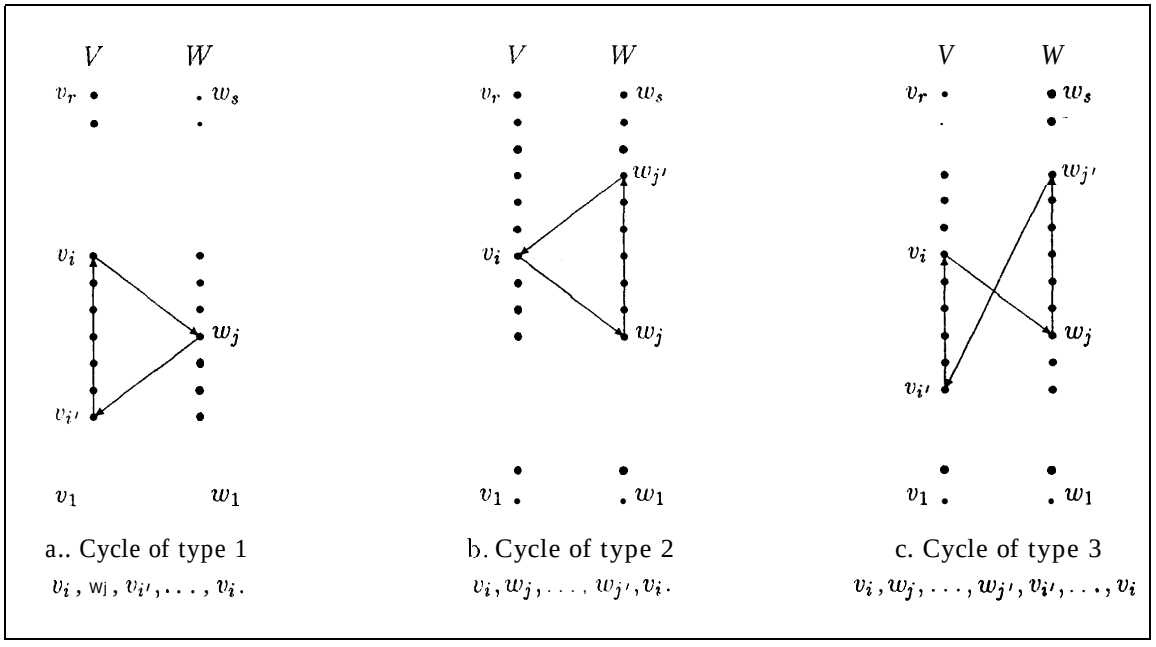
output: A sequence $B = b_1, \ldots, b_\ell$ of integers such that for every i, $1 \leq i \leq \ell$.

$$b_i = \max_{1\leq k\leq i}\{a_k\}.$$

Lemma 4.2 **MAXPREFIX can be implemented in the CRCW model in** O(1) **time.**

Proof: First construct in $O(1)$ time a matrix $M = \{m_{i,j}\}_{1\leq i,j\leq\ell}$ where $m_{i,j}$ = 1 iff $a_i > a_j$) or $i = j$. Each row i calls a variation of procedure MAXINDEX and computes in 0(1) time: the minimum index j in the subrow $m_{i,i+1}, \ldots, m_{i,\ell}$ such that $m_{i,j} = 0$ and $m_{i,i+1} = \cdots = m_{i,j-1}$ = 1. For every k, $1 \leq k < i$ or j $\leq k \leq \ell$ set $m_{i,k}$ = 0. If now $m_{i,j} = 1$, then $a_i > a_j$ and for each k, $i < k < j$, it is also known that $a_i > a_k$.

Therefore, it now remains for each a_j to find the minima.1 index i such that $m_{i,j}$ = 1. Again, this can be done in time 0(1) by calling a variation of procedure MAXINDEX for each column of the matrix M. ∎



a.. Cycle of type 1 v_i, w_j, $v_{i'}, \ldots, v_i$.

b. Cycle of type 2 $v_i, w_j, \ldots, w_{j'}, v_i$.

c. Cycle of type 3 $v_i, w_j, \ldots, w_{j'}, v_{i'}, \ldots, v_i$

Figure **1: The three types of cycles.**

We are now ready to present Algorithm MFS.

Algorithm MFS:

Input: A complete path bipartite graph $G = (V, W, Q)$; $V = v_1, \ldots, v_r$; $W = w_1, \ldots, w_s$; $O(r^2 + rs)$ processors.

Output: **A** minimal feedback set F in G.

For all i, **1** $\leq i \leq r$ **:**

1. Define $a(i)$ as follows:
   (a) If $v_i > w_s$, then $a(i) = s$;
   (b) else, if $v_i < w_1$, then $a(i) = 0$;
   (c) otherwise, let $a(i)$ be the maximum index j such that $w_j < v_i < w_{j+1}$ $(1 \leq j \leq s - 1)$.
2. Let b(i) be the maximum value among $\{a(1), \ldots, a(i)\}$.
3. For all j, $1 \leq j \leq b(i)$: if $v_i < w_j$ then add the edge $v_i \rightarrow w_j$ to F.

Lemma 4.3 The set F computed by Algorithm MFS is a minimal feedback set in Q.

Proof: F is a minimal feedback set if Q' = Q – F is acyclic and if adding an edge of F to Q' generates a cycle.

Assume to the contrary that there is a cycle in Q' and suppose that the cycle is one of the following three types:

1. $v_i, w_j, v_{i'}, \ldots, v_i$ (Figure la);
2. $v_i, w_j, \ldots, w_{j'}, v_i$ (Figure 1b);
3. $v_i, w_j, \ldots, w_{j'}, v_{i'}, \ldots, v_i$ (Figure 1c).

We now show that in the three above cases $b(i) \geq$ j and this leads us to a contradiction, because in Step 3 of algorithm MFS, edge $v_i \rightarrow w_j \in F$.

1. The edge $w_j \rightarrow v_{i'}$ is in Q; therefore the maximality of $a(i')$ implies that $a(i') \geq$ j and consequently, b(i) $\geq b(i') \geq a(i') \geq$ j.
2. The edge $w_{j'} \rightarrow v_i$ is in Q; therefore $a(i) \geq$ j' which implies that $b(i) \geq$ j' > j.
3. The edge $w_{j'} \rightarrow v_{i'}$ is in Q : therefore- b(i) $\geq b(i') \geq a($ i') $\geq$ j' > j.

Now let c be an arbitrary cycle and let v_i be its highest vertex in V. The successor of v_i in c belongs to W, and denote it by w_j. We show that there are three possible cases, and each implies the existence of one of the above three types of cycles.

If w_j is the highest vertex of c in W then its successor in c is $v_{i'} \in V$. It follows that i' < i (from v_i being the highest j and hence $v_i, w_j, v_{i'}, \ldots, v_i$ is a cycle in Q' of type (1). Otherwise, let $w_{j'}$ be the highest vertex of c in W for some j' > j. Denote the successor of $w_{j'}$ in c by $v_{i'} \in V$ for i' $\leq i$. There are two cases. If i' = i then in Q', $v_i, w_j, \ldots, w_{j'}, v_i$ is a cycle of type (2). Else. $i' < i$ and $v_i, w_j, \ldots, w_{j'}, v_{i'}, \ldots, v_i$ is a cycle in Q' of type ('3).

To complete the proof, we have to show that if an edge $v_i \rightarrow w_j \in F$ is added to Q', then a cycle is generated. This follows from the following two facts:

Fact 1: The edge $w_{a(i)}$ -- v_i is always an edge in Q' as' long as $a(i) > 0$.

Fact 2: if $v_i \rightarrow w_j \in F$ then j $\leq$ b(i).

If j < n(i), then $v_i, w_j, \ldots, w_{a(i)}, v_i$ is a cycle (type (2)). Otherwise, by the above two facts $a(i) <$ j $\leq$ b(i). If j = $b(i)$, then there exists $i' < i$ such that $a(i')$ = b(i). By Fact 1, $w_{b(i)} \rightarrow v_{i'} \in$ Q' and $v_i, w_j, v_{i'}, \ldots, v_i$ is a cycle (type (1)). Else, j < $b(i)$ and again, there exists i' < i such that $a(i')$ = b(i) > j and by Fact 1 $w_{b(i)} \rightarrow v_{i'} \in$ Q'. It follows that $v_i, w_j, \ldots, w_{b(i)}, v_{i'}, \ldots, v_i$ is a cycle (type (3)). ∎

Lemma 4.4 Algorithm MFS can be computed in time O(1).

Proof: It is easy to see how step 1c can be computed with the help of procedure MAXINDEX with rs processors. Hence, it takes time O(1) by Lemma 4.2. Step 2 calls Procedure MAXPREFIX with r^2 processors and can also be computed in time O(1) by Lemma 4.3. Obviously, the rest of the algorithm can be computed in time $O(1)$. ∎

4.2 Merging

In this subsection we show how Valiant's [Va] merging algorithm can be modified to a CRCW algorithm that merges two paths in a tournament. We elaborate on its description for two reasons: (i) to simplify the proof of Theorem 4.1; (ii) to the best of our knowledge, a detailed description of Valiant's algorithm in the PRAM model does not exist. The algorithm uses a. linear number of processors (linear in the length of both paths) and its time complexity remains O(log log n) (n is the length of X, the longer path). Borodin and Hopcroft [BH] proved a lower bound of $\Omega(\log\log n)$ on the complexity of merging two paths and hence, our results are tight. Hereafter, denote by $X = x_1, \ldots, x_n$ and $Y = y_1, \ldots, y_m$ the two paths to be merged. We omit ceiling and floor for clarity.

Valiant's algorithm employs a. **divide-and-conquer** method: in O(1) time, merging two paths is divided into merging many pairs of subpaths from X and Y, where in each pair, the length of the subpath belonging to X is $\sqrt{n}$. Then. in 0(1) time, the paths produced by merging the pairs are concatenated to output the desired merge. The basic step of the recursion is when n is a. constant and then, the merge is completed by performing a.11 possible comparisons. The time complexity $O(\log\log n)$ is the solution of the recursive equation:

- $T(\alpha)$ = O(1) ; $\alpha = O(1)$;
- T(n) = $T(\sqrt{n})$ + O(1).

We now explain how the divide-a.nd-conquer is achieved. Each of the two paths, X and Y, is divided into subpaths whose lengths are a. square root of their original length. The top of every subpath is called its leader, and denote by LX and LY the set of leaders in X and Y respectively. We may sometimes refer to LX and LY also as paths.

The first step of comparisons in the algorithm is between all the leaders of LX with all the leaders of LY. (This can be clone in one step as $\sqrt{nm} < n + m$.) At this point, for each leader $y \in LY$, there are two successive leaders $x' < x \in LX$, such that $y > x'$ and $y < x$. This pair of leaders is uniquely defined in the case of a transitive tournament and can be computed in time 0(1). The exact place in X where y is eventually inserted, will be called the **insertion point** of y. Now, for two consecutive leaders in LY, $y' < y$, there is a. subpath $LX(y)$ in LX and a. subpath $Y(y)$ in Y, such that each vertex in these subpaths is smaller

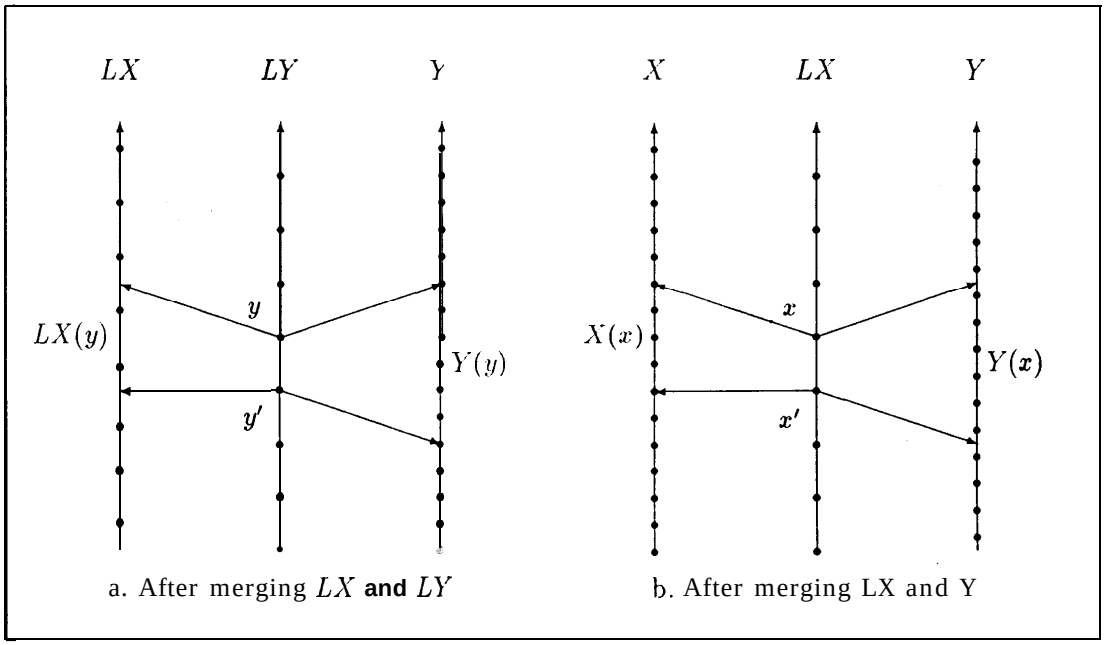


Figure 2: **Proof of Theorem 4.1**

. than y and greater than y' (Figure 2a).

The second step of comparisons in the algorithm is performed between all the vertices in LX(y) and Y(y) for each leader y $\in LY$. After this step, the insertion point in Y of each leader $x \in$ LX is known, namely there are two successive vertices in Y, y' < y, such that x > y' and x < y. Again, this pair of vertices is uniquely defined in the case of a transitive tournament and can be computed in time $O(1)$. Now for two consecutive leaders in LX, $x' < x$, there is a subpath $X(x)$ in X and a. subpath Y(x) in Y, such that each vertex in these subpaths is smaller than x and greater than x' (Figure 2b).

Now the subpaths can be merged simultaneously. For every leader $x \in LX$, the algorithm merges recursively the paths $X(x)$ and Y(x) into the path $Z(x)$. After the recursion is completed, x (x') can be inserted above (below) $Z(x)$.

Thus, by merging $\sqrt{n}$ pairs of subpaths and then concatenating them through the leaders of X , the desired merge is achieved.

The comparisons steps that precede the recursive calls in Valiant's algorithm incur two problems in an arbitrary tournament. A leader y $\in LY$ ($x \in$ LX) may have several insertion points in LX (Y); furthermore, if y' < y (x' < x) then the insertion point of y' (x') is not necessarily smaller than that of y (x).

To overcome these problems, we compute a minimal feedback set in the graph induced by the comparisons, as suggested by Theorem 3.1. Whenever a set of vertices is compared,

a. minimal feedback set is computed and the orientation of its edges is flipped. In all cases, the minimal feedback set is either the output of Algorithm MFS, or the union of outputs of different calls to Algorithm MFS. There are four cases summarized below:

1) **The base case of the recursion:** Here, n is a constant, and the minimal feedback set is the output of Algorithm MFS computed on the whole graph.

2) **The first round of comparisons before the recursive calls:** Here, the leaders of LX and LY are compared and again, the minimal feedback set is the output of Algorithm MFS.

3) **The second round of comparisons before the recursive calls:** The graph induced by these comparisons is a union of CPB graphs and Algorithm MFS is invoked for each CPB graph. The minimal feedback set is the union of the minimal feedback sets computed for each CPB graph.

4) **The rounds during the recursive calls:** The structure of the comparison graph is the same as in case (3) and the minimal feedback set is the union of the minimal feedback sets computed for each CPB graph.

Theorem 4.1 The modification of Valiant's merge algorithm according to Theorem 3.1 merges two paths in an arbitrary tournament.

Proof: We prove it by induction on k, the depth of the recursion. When k = 1 (case 1), namely the length of one of the merged paths is a constant. the correctness follows as Algorithm MFS computes a minimal feedback set (Lemma 4.3). Assume that the theorem is true when the recursion depth is less than k.

In the first step of comparisons (case 2), the leaders of X and Y are compared. It. follows again from lemma 4.3 that a minimal feedback set is indeed computed. Hence, the picture depicted by Figure 2a is valid. Now assume that the second round of comparisons before the recursive calls took place (case 3) and assume also that the union of minimal feedback sets computed for edge disjoint subgraphs is not a feedback set. To disprove this last assumption, it is enough to show for every cycle that all of its edges belong to the same subgraph . We prove the following claim:

Claim: For each leader y $\in LY$, if a vertex from LX(y) (or $Y(y)$) takes part in a cycle, then that cycle is completely contained in LX(y) $\cup$ $Y(y)$.

Proof of the claim: Construct the following graph H: for each leader y $\in LY$ associate a vertex $f(y)$ in H which is the contraction of Lx(y) and Y(y) e.g., a vertex in H can be viewed as a set of "old" vertices. Let a and b be two vertices in H; the edge $a \rightarrow b$ exists if before the contraction, there was an edge oriented from a vertex in the set a to a. vertex in the set b.

Notice that the leader y is the only vertex in f(y) that was compared to vertices in $LX(y')$ for any leader y' < y. The way the segments in LX were chosen implies that y is greater than all the vertices in $LX(y')$ and clearly y is greater than all the vertices in $Y(y')$. Hence, H is well defined and is isomorphic to a path. The correctness of the claim follows immediately. ∎

In the rounds during the recursion (case 4), similar arguments to the above hold. The analogous claim is that for each leader $x \in$ LX, if a vertex from $X(x)$ (or $Y(x)$) takes part in a cycle, then that cycle is completely contained in LX(y) $\cup$ $Y(y)$. Therefore, it is enough to compute a minimal feedback set in each CPB graph. ∎

Theorem *4.2* **The time complexity of the modified algorithm is** $O(\log\log n)$ **when** $O(n)$ **processors are available.**

Proof: Define *T(n, m)* to be the time complexity of the modified algorithm. By Lemma, *4.4,* Algorithm MFS can be implemented in time O(1) if the number of available processors is $O(r^2 + rs)$, where r and s denote the lengths of the input paths. We need to show that whenever Algorithm MFS is invoked there is a sufficient number of processors available.

If n is a constant, the claim follows from the fact that $O(n + m)$ = 0(nm). When LX and LY are compared, the claim follows from the inequality $\sqrt{nm} < n + m$.

If $\sum \alpha_i = \sqrt{n}$, then $\sum \alpha_i^2 \leq$ n. Therefore, there are enough processors for the second step of comparisons that precedes the recursive call. Each subpath $Y(y)$ from Y that is merged with α leaders from LX, receives $O(\alpha^2 + \alpha\sqrt{m})$ processors.

For the simultaneous recursive calls there are enough processors, as each submerge of two paths of length n' and m' gets $O(n' + m')$ processors.

Clearly, the rest of the algorithm can be implemented in constant time same as the transitive case. Hence the recursive equation is:

$$T(n, m) = T(\sqrt{n}, \boldsymbol{mj} + c,$$

for some constant c. The solution of the above equality is $O(\log\log n)$. ∎

4.3 Sorting

In this subsection we show how a. path in a. tournament can be found in the CRCW model in O(log n) time using O(n) processors. We rely on Cole's merge-sort [Co] algorithm and compute a minimal feedback set in $O(1)$ time for each round of comparisons. We shall give only an outline of Cole's algorithm, and elaborate on the steps where comparisons are performed. We refer the reader to the description of Cole's algorithm in [GR] as our outline depends on it and uses its terminology.

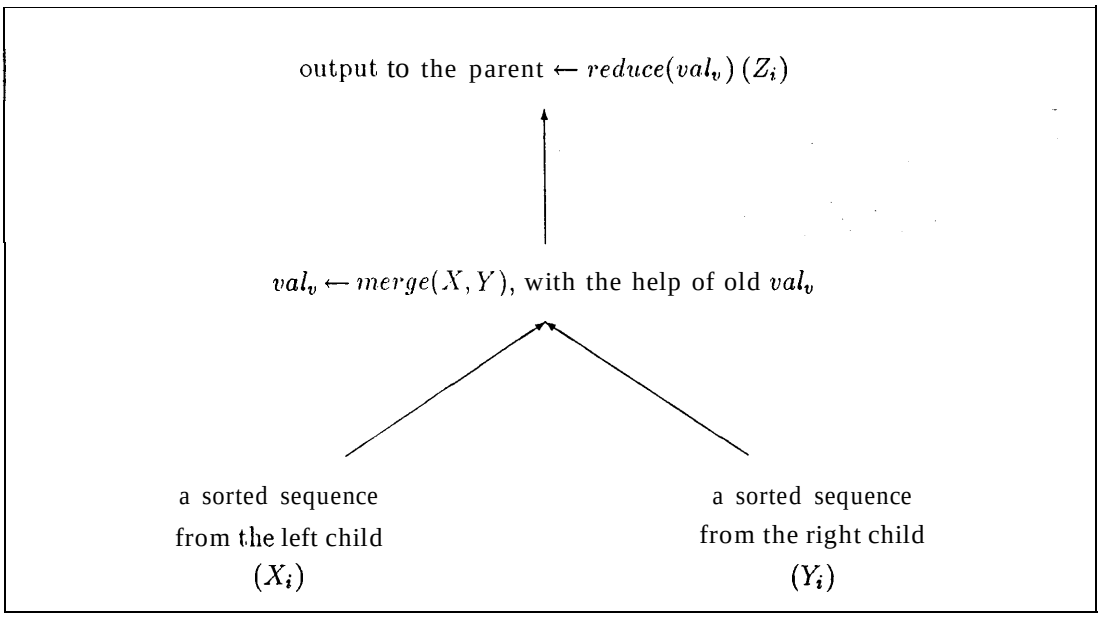


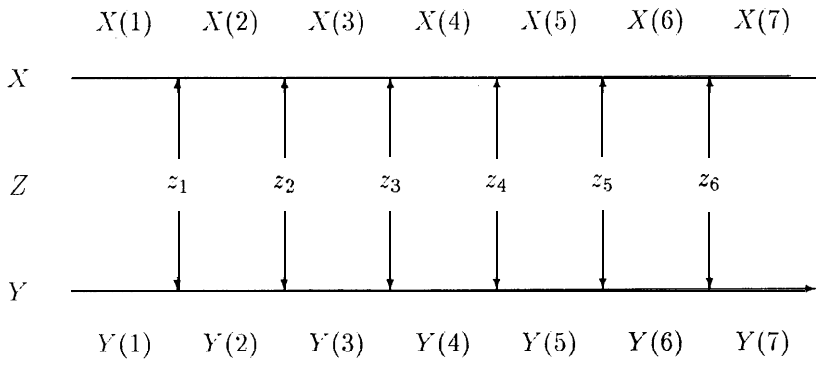
Figure **3: The local action in an internal node v in the tree.**

In the previous subsec tion, the merging algorithm implies a sorting algorithm whose complexity is $O(\log\ n\ \log\log n)$. To improve the complexity to O(log n), *Cole* showed how to pipeline the merging steps. Let T be the complete binary tree that describes the standard **merge-sort** algorithm; a, typical node v in T merges two lists X and Y when both X and Y are sorted. The novelty of Cole's algorithm is that node v starts processing the lists X and Y before they are sorted. Namely, at the ith step, two lists X_i and Y_i are merged, where X_i is **a** sorted sample of X, and Y_i is a sorted sample of Y. This merge can be computed in constant time if the results of the $i-$ 1st step are known.

For the sake of simplicity, assume that n is a. power of 2. Let T_v be the subtree rooted at v and $list_v$ be the list of elements stored initially at the leaves of T_v. Let val_v be the current list associated with node v of the tree. The sequence val_v will always be a sorted su bsequencc of list,, , and will double its size in each round. We say that a. node is **complete** if and only if $val_v = list_v$. Let us now describe what happens in a typical internal node v of the tree during the course of the algorithm (Figure 3).

In the ith step, node v receives from its left child and right child sequences X_i and Y_i respectively. It merges the two sequences to a sorted sequence val_v with the help of the old val_v. If v is incomplete, then it sends every fourth element of' val_v to its parent. During the first step after v becomes complete. every second element is sent up, whereas in the step after that, every element is sent up. The sequence Z_i sent by v to its pa-rent is called $reduce(val_v)$.

'I he algorithm begins when all the lea?.\-es send their value to their parents. A node



$Z = (z_1, z_2, z_3, z_4, z_5, z_6)$

$X(i)$ = the set of elements of X lying between the $i-1$th and ith elements of Z.

$Y(i)$ = the set of elements of Y lying between the $i-1$th and ith elements of Z.

$merge(X, Y) = merge(X(1), Y(1)) \ldots merge(X(7), Y(7))$

Figure 4: **Merging X and Y with the help of a good sampler Z.**

terminates two steps after it became complete, and the algorithm ends when the root becomes complete. It follows from the way $reduce(val_v)$ is defined, that two rounds after v became complete, its parent becomes complete. Therefore, the algorithm takes $O(\log n)$ rounds of internal computation in the nodes of the tree. In order to achieve the desired complexity, Cole showed how to implement each such round in constant time.

We now need some notations. The **rank** of an element x in a. sequence X, $rank(x, X)$, is the number of elements in X preceding x. The **cross rank** from X to Y, denoted by $R[X, Y]$, is the function for which $R[X, Y](x) = rank(x, Y)$ for each $x \in X$. A sorted sequence X is a good **sampler** of the sorted sequence **Y** if and only if, between any $k+1$ **adjacent** elements of $\{-\infty\} \cup X \cup \{\infty\}$ there are at most $2k+1$ elements of Y. In our case, assume that $k = 1$. i.e., between any two element of Y there are at most three elements of X. Note that it is always true that $reduce(X)$ is a good sampler of X.

The motivation behind these notations is as follows. The cross rank $R[X, Y]$ ($R[Y, X]$) enables us to merge the sorted sequences X and Y in time **O(1).** In this sense? the merge of two sequences and their cross ranks, are equivalent. Our description relies heavily on this fact. It is also easy to merge in constant time two sorted sequences X and Y with the help of a. sequence Z which is a. **good** sampler of both (Figure 4).

The basic property of the algorithm that entails its correctness is well demonstrated by the following invariant preserved at each step:

Main invariant: if each X_i is a good sampler of X_{i+1}, and each Y_i is a good sampler of Y_{i+1} then each Z_i is a good sampler of Z_{i+1}.

In terms of cross ranks, the merge of X_{i+1} and Y_{i+1} is performed in three steps:

1. The four cross ranks: $R[X_i, Y_{i+1}]$, $R[Y_i, X_{i+1}]$, $R[X_{i+1}, Y_i]$ and $R[Y_{i+1}$, X;] are computed with the help of $R[X_i, X_{i+1}]$ and $R[Y_i, Y_{i+1}]$.
2. The two cross ranks: $R[X_{i+1}, Y_{i+1}]$ and $R[Y_{i+1}, X_{i+1}]$ are computed with the help of the cross ranks of the previous step. (As stated before, this is equivalent to merging X_{i+1} and Y_{i+1}).
3. The cross rank $R[Z_i, Z_{i+1}] = R[reduce(X_i \cup Y_i), reduce(X_{i+1} \cup Y_{i+1})]$ is computed with the help of the previous calculated cross ranks.

The three steps can be implemented in O(1) time with $O(|X_{i+1}| + |Y_{i+1}|)$ processors. The reason is that the good sampling property insures us that in each CPB graph, at least one of the paths is of length at most three.

We now explain how to modify the above algorithm according to Theorem 3.1 so that it computes a Hamilton path in an arbitrary tournament. Whenever a. set of vertices is compared, a minimal feedback set is computed and the orientation of its edges is flipped. Each stage of comparisons can be decomposed to edge disjoint CPB graphs and in each one a minimal feedback set is computed by invoking Algorithm MFS. The union of the minimal feedback sets computed in each CPB graph will be a minimal feedback set in the graph induced by the set of comparisons.

Theorem 4.3 The modification of Cole's sort algorithm according to Theorem 3.1 finds a Hamilton path in an arbitrary tournament.

Proof: For arguments similar to those in the proof of the merging algorithm, it follows that the union of the outputs of the calls for algorithm MFS in each node of the tree T, is a. minimal feedback set in the graph induced by the comparisons performed at that node.

The theorem follows from the next claim proved by induction on $\langle k, i \rangle$, where k is the 'height of the tree and i is the current step.

- In the ith step, in a tree of height k, the algorithm computes a. minimal feedback set.
- The sequence sent by v, the root of this tree, in the ith step is consistent with the previous sequences sent by v. Namely, if val_v contains the relation $x < y$ for a, $y \in list_v$, then in all previous steps, val_v never contained the relation $y > x$.

The induction holds clearly for $k = 1$, namely the leaves. Now let T' be a. tree of height k, let v be its root and let i be the current, step. We prove the first part of the inductive claim by showing that there cannot be a cycle after the edges of the minimal feedback set

in T', computed in the ith step, are flipped. Let X and Y be the sequences that v received from its left and right children respectively. It follows from the induction assumption on the height of the tree that any cycle contained completely in X or in Y was handled by-the children of v. On the other hand, a cycle containing vertices from both X and Y could only be generated by v. Assume there is such a cycle and w.l.o.g. it contains an edge between vertices $a, b \in$ X where a precedes b with respect to X. Because of the second part of the induction hypothesis, this edge is $a \rightarrow b$. However, such cycles are detected by Algorithm MFS.

After flipping the minimal feedback set, the graph induced by $list_v$ contains an instance of Cole's merge-sort algorithm. Therefore, the correctness of the second claim follows from the correctness of Cole's algorithm. ∎

Theorem 4.4 The time complexity of the modified algorithm is $O(\log n)$.

Proof: A more detailed inspection of the algorithm shows that in each CPB graph at least one of the paths is of length at most three. In Cole's algorithm each node in the tree gets $O(|X_{i+1}| + |Y_{i+1}|)$ processors, this number is sufficient to calculate the calls for algorithm MFS. ∎

Acknowledgment

We would like to thank Eli Gafni, Nati Linial and Moni Naor for valuable discussions.

References

[AA1] N. Alon and Y. Azar, *Sorting, approximate sorting,, and searching in rounds*, Siam J. Disc. Math., Vol. 1, No. 3, pp. 269-280 (1988).

[AA2] N. Alon and Y. Azar, *The average complexity of deterministic and randomized parallel comparison sorting algorithms*, Proceedings 28th Annual IEEE Foundations of Computer Science, Los Angeles, CA (1987), pp. 489-498; also: Siam J. Comput., to appear.

[AAV] N. Alon, Y. Azar and U. Vishkin, *Tight complexity bounds for parallel comparison sorting*, Proceedings 27th Annual IEEE Foundations of Computer Science. Toronto, Ontario, Canada (1986), pp. 502-510;

[AKS] M. Ajtai, J. Komlos and E. Szemeredi, *Sorting in c* $\log n$ *parallel.* steps, Combinatorica 3 (1) (1983) pp. 1-19.

[AKSS] M. Ajtai et al., *Deterministic selection in* $O(\log \log n)$ *parallel time*, Proceedings 18th Annual ACM Symposium on Theory of Computing, Berkeley, CA, (1986), pp. 188-195.

[Al] *N.* Alon, *Eigenvalues, geometric expa nders, sorting in rounds and Ramsey theory,* Combinatorica, 6 (3) (1986) pp. 207-219.

[AV] *Y.* Azar and U. Vish kin, *Tight comparison bounds on the complexity of parallel sorting*, Siam J. Comput ., Vol. 3, (1987), pp. 458-464.

[Ba] K. E. Batcher, *Sorting networks and their applications, in* Proc. AFIPS Spring Joint, Comput. Conf., Vol. 32, April 1968, pp. 307-314.

[Be] *C.* Berge, *Graphs and Hypergraphs*, North Holland Publishing Company, 1973.

[Bea] P. Beame, *Limits on the power of concurrent-write parallel machines*, Proceedi ngs ACM 18th Symposium on Theory of Computing, 1986, pp. 169-176.

[BHe] B. Bollobas and P. Hell, *Sorting and graphs,* in Graphs and Orders, I. Rival. ed., D. Reidel, Boston, MA, 1985, pp. 169-184.

[BHo] A. Borodin and J. Hopcroft, *Routing, merging and sorting on parallel models oj computation*, J. of Computers Systems and Sciences ...

[Co] Richard Cole, *Parallel Merge Sort,* Siam J. Cornput., Vol. 17, No. 4, pp. 770-785 (1988).

[GJ] M. Carey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman, San Francisco. 1979.

[GR] A. Gibbons and W. Rytter, *Efficient parallel algorithms,* Cambridge University Press, 1988.

[HR] P. Hell and M. Rosenfeld, *The complexity of finding generalized paths in tournaments*, J. of Algorithms, Vol. 4, pp. 303-309 (1983).

[Mo] J. W. Moon, *Topics on tournaments*, Holt, Rinehart and Winston Inc., 1968.

[Na] Joseph Naor, *Two parallel algorithms in graph theory*, Technical Report CS-86-6, Dept. of Computer Science, Hebrew University, June 1986.

[P] N. Pippenger, *Sorting and selecting in rounds*, Siam J. Comput. 16 (1987) 1032-1038.

[Ra] V. Ramachandran, *Private communication.*

[Re] L. Redei, *Ein kombinatorischer satz*, Acta Litt. Sci. Szeged, 7 (1934) pp. 39-43.

[So] D. Soroker, *Fast parallel algorithms for finding hamilton paths and cycles in a tournament*, Journal of Algorithms, 9, (1988) 276-286.

[Va] L. G. Valiant, *Parallelism in Comparison problems*, Siam J. Comput. 4 (1975) 348-355.