Slowing Down Sorting Networks to Obtain Faster Sorting Algorithms

RICHARD COLE

New York University, New York, New York

Abstract. Megiddo introduced a technique for using a parallel algorithm for one problem to construct an efficient serial algorithm for a second problem. This paper provides a general method that trims a factor of $O(\log n)$ time (or more) for many applications of this technique.

Categories and Subject Descriptors: F.1.2 **[Computation by Abstract Devices]**: Modes of Computation—*parallelism*; *relations among modes*; F.2.2 **[Analysis of Algorithms and Problem Complexity]**: Nonnumerical Algorithms and Problems—*computations on discrete structures*; *geometrical problems and computations*; *sequencing and scheduling*; *sorting and searching*; G.2.1 **[Discrete Mathematics]**: Combinatorics—*combinatorial algorithms*; G.2.2 **[Discrete Mathematics]**: Graph Theory—*network problems*; *path and circuit problems*; *trees*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Algorithms on trees, ham sandwich theorem, min-ratio cycle, parallel algorithms, scheduling, sorting network, spanning tree

1. *Introduction*

We solve a somewhat unusual sorting problem optimally; it is described in Section 2. The sorting problem was motivated by and derived from its application. The application is an improvement of Megiddo's ingenious technique [9], which uses an efficient parallel algorithm for one problem to produce an efficient serial algorithm for a second problem. However, Megiddo's technique is more general than our improvement. That is, there are problems to which Megiddo's technique can be applied, but to which our improvement cannot be applied. Some of the ideas involved in Megiddo's technique were first presented in [8]. The general idea of Megiddo's technique is as follows: Suppose a certain problem A is solved in T_p time units on a P-processor machine. Also suppose a serial algorithm for problem A running in time T_s could be applied to designing a serial algorithm for problem B with a running time of $O(CT_s)$, where essentially the algorithm for problem B carries out each step of the algorithm for A, taking time C per step. By using the parallel algorithm for A as a serial algorithm, we would obtain a serial algorithm for B running in time $O(CT_pP)$. Megiddo showed that in many cases problem B can actually be solved in time $O(CT_p \log P)$, if P is not too large. Our contribution

This work was supported in part by the National Science Foundation under grant DCR 84-01633 and by an IBM Faculty Development Award.

Author's address: Courant Institute of Mathematical Science, New York University, 251 Mercer Street, New York, New York 10012.

is to reduce this time to $O(CT_p)$ in many cases. We describe Megiddo's technique and the improvement in Section 3. Using the improvement, we trim the running times of several algorithms, which we describe in Section 4.

Remark. Although our technique trims the running times of several algorithms, it is at the expense of using the Ajtai–Komlos–Szemeredi (AKS) sorting network [2], which involves enormous constants.

2. *The Sorting Problem*

We describe the sorting problem as a game. There are two players: the sorter and the adversary. The game is played on a sorting network for sorting n items; the network has width $n/2$ and depth $f(n)$. The idea of the game is to carry out a sort on the network. The game is played in turns. In a turn, first the sorter requests the adversary to resolve certain comparisons (i.e., to determine which of two inputs is the larger); then the adversary resolves some of these comparisons (we are more specific below). By "sorting on the network" we mean that the sorter must obtain the result of exactly those comparisons that arise when the network is used, and no others (except those that can be deduced by transitivity); also, the sorter must follow the ordering of the comparisons created by the network (that is, if an output of comparator D is an input to comparator C, then the result of the comparison at D must be known before the comparison at C is attempted).

Before giving the precise rules for the game, we need a few definitions. We define an input to a comparator C to be known, either if the input is a circuit input (i.e., the input wire for the input to comparator C is an input wire for the circuit), or if the input was the output of some other comparator D and the comparison at D has been resolved. A comparator is defined to be *active* if both its inputs are known and the order of the inputs has not yet been determined; a comparator is *inactive* if it is not active.

Now, we define the game precisely. The two players alternate their moves, which are as follows.

(1) The sorter assigns weights to all the active comparators. Let the sum of the weights assigned be W (the *active* weight). Let C be the comparison corresponding to comparator $\bar{C}$. If $\bar{C}$ is assigned weight w, we consider C to have been assigned weight w also.
(2) The adversary is obliged to resolve sufficiently many of the weighted comparisons so that the sum of the weights of the resolved comparisons is at least $W/2$.

The game ends when the sort is complete. The aim of the sorter is to end the game quickly. A turn consists of one move of each player; we show the sorter can end the game in $O(\log n + f(n))$ turns. The sorter's strategy is to assign weights to the comparators according to the following rule. An active comparator at depth j in the network is given weight 4^{-j}. We prove the following invariant.

LEMMA 1. *At the start of the* $k + 1$*st turn the active weight is bounded by* $(3/4)^k \cdot n/2$, *for* $k \geq 0$.

PROOF. We prove the result by induction on k. At the start of the first turn there are $n/2$ active comparators at depth 0, and all other comparators are inactive. So for $k = 0$ the result holds. To prove the inductive step, it is sufficient to show that at each turn the active weight is reduced by at least one quarter. We now show this.

Consider an active comparator $\bar{C}$ of weight w, and suppose the corresponding comparison C is resolved. Then $\bar{C}$ ceases to be active, and up to two comparators, each of weight $w/4$, may become active. So the resolution of C reduces the active weight by at least $w/2$. Let the active weight be W. In one turn, we are guaranteed that the comparisons resolved by the adversary have combined weight at least $W/2$. Thus, in one turn, the active weight is reduced from W to at most $3W/4$. □

We now show that this process terminates reasonably quickly.

LEMMA 2. *For $k \geq 5(j + 1/2 \log n)$, during the $k+1$st turn there are no active comparators at depth j.*

PROOF. At the start of the $k+1$st turn the total active weight W is bounded by $(3/4)^{5(j+1/2\log n)} \cdot n/2$ (by Lemma 1). We note $(3/4)^5 < 1/4$. So $W < (1/4)^{j+1/2\log n} \cdot n/2 = (1/4)^j \cdot 1/2$. But an active comparator at depth j has weight $(1/4)^j$. So there is no such comparator. □

COROLLARY. *The game ends after $O(f(n) + \log n)$ turns.*

PROOF. After $5(f(n) + 1/2 \log n)$ turns there are no active comparators (by Lemma 2). That is, all the comparisons are resolved, so the sort is completed. Hence the game ends after $O(f(n) + \log n)$ turns. □

Remark. It is of interest to play this game on arbitrary directed acyclic graphs. Lemma 1 and the corollary hold even in the case of unbounded fanin (respectively, fanout) as long as the fanout (respectively, fanin) is bounded. It suffices to redefine the weights as follows: Give each output a weight of 1, and let the weight of each internal node be twice the sum of the weights of its immediate descendants (then scale to make the initial weight equal to $n/2$). Using the same weight assignment, it is easy to obtain a bound of $O(\log w + d \log \delta)$ turns for playing this game on the class of graphs of width w, depth d, and $\min\{\max \textit{fanin}, \max \textit{fanout}\} \leq \delta$.

3. *Improving Megiddo's Technique*

In general terms, Megiddo's technique provides a way to search a partially ordered space, possibly of superpolynomial size, without actually constructing the space. Instead an *implicit* binary search of the space is made. We use a sorting algorithm to guide this search (Megiddo's technique is not restricted to sorting algorithms). Typically, a sorted order corresponds to a region of the space being searched. So resolving a comparison in the sort corresponds to reducing the size of the space in which a solution is known to lie. As might be expected, a single comparison is expensive; but surprisingly, for some problems, several comparisons can be batched relatively cheaply. This leads Megiddo to use a parallel sorting algorithm to guide the search, for in the searching algorithm he can batch the comparisons that are performed simultaneously in the parallel algorithm.

More precisely, suppose that we have a fast parallel algorithm for one problem, problem A say, which is used to construct a fast serial algorithm for a second problem, problem B say. Further, suppose that problem A is sorting and problem B has the following unusual features.

(1) Problem B can be solved by "sorting" but each "comparison" is expensive, that is, it takes time $C(n)$ rather than time $O(1)$. Typically $C(n)$ is $O(n)$ or $O(n \log n)$.
(2) (The *batching* rule.) If we consider m of these "comparisons," $C_1, \ldots, C_m$, we can order the comparisons, $C_{\pi(1)} \leq \cdots \leq C_{\pi(m)}$, in the following sense. (We

think of a comparison C as being the question "Is $c_1 < c_2$?," which has answer either "yes" or "no.") An answer of "yes" to $C_{\pi(j)}$ forces $C_{\pi(1)}, \ldots, C_{\pi(j-1)}$ to have answer "yes," while an answer of "no" to $C_{\pi(j)}$ forces $C_{\pi(j+1)}, \ldots, C_{\pi(m)}$ to have answer "no." Also we can determine the relative order of two comparisons fairly quickly, typically in time $O(1)$.

We give an example to illustrate this definition (this example was previously given in [9]). This example is not intended to be solved by the methods described below; it is being given solely to illustrate the method. Suppose we are given n increasing, linear functions of x: $f_i(x) = a_i x + b_i$, $a_i > 0$, $1 \le i \le n$. We define the median function $f(x) = \text{median}[f_1(x), \ldots, f_n(x)]$. We note $f(x)$ is an increasing function of x. The problem is to find the unique x^* such that $f(x^*) = 0$. One (unorthodox) way of solving this problem is to find an i such that $f_i(x^*) = f(x^*)$ (without knowing x^*). Having found i, it is then easy to deduce x^*.

How do we find i? To do this we could compute an index i such that $f_i(x^*) = \text{median}[f_1(x^*), \ldots, f_n(x^*)]$. One way of finding this index is to sort the (symbolic) values $f_1(x^*), \ldots, f_n(x^*)$, without knowing x^*, and then pick out the median: Its index is the i we seek.

So the question becomes: How do we sort the values $f_1(x^*), \ldots, f_n(x^*)$, without knowing x^*? We explain how to compare an arbitrary pair of these values, $f_1(x^*)$ and $f_2(x^*)$, without knowing x^*; then we can insert this comparison method as a subroutine into any standard sorting algorithm.

If $f_1(x)$ and $f_2(x)$ represent parallel lines, then either $f_1(x) > f_2(x)$, or $f_1(x) = f_2(x)$, or $f_1(x) < f_2(x)$, for all x, and in particular for $x = x^*$. So suppose f_1 and f_2 are not parallel; in fact, without loss of generality suppose $a_1 > a_2$. Then $f_1(x) = f_2(x)$ for some unique value $x = x_0$; so for $x > x_0$, $f_1(x) > f_2(x)$, while for $x < x_0$, $f_1(x) < f_2(x)$. We conclude that, if $x^* > x_0$, $f_1(x^*) > f_2(x^*)$, if $x^* = x_0$, $f_1(x^*) = f_2(x^*)$, and if $x^* < x_0$, $f_1(x^*) < f_2(x^*)$. Thus resolving a comparison reduces to determining whether $x^* > x_0$, $x^* < x_0$, or $x^* = x_0$. To do this, we simply evaluate $f(x_0)$ (in $O(n)$ time). Since $f(x)$ is an increasing function, we deduce that if $f(x_0) < 0$, $x^* > x_0$, if $f(x_0) = 0$, $x^* = x_0$, and if $f(x_0) > 0$, $x^* < x_0$. Thus in $O(n)$ time ($=O(C(n))$ time) we can resolve a comparison.

In addition, we show that comparisons can be ordered. Suppose we have m comparisons C_i of the form "Is $f_{i_1}(x^*) < f_{i_2}(x^*)$?", where $a_{i_1} > a_{i_2}$, $1 \le i \le m$. Each comparison determines a value x_i, for which determining whether $x^* > x_i$, $x^* = x_i$, or $x^* < x_i$ resolves the comparison. We can order these values: $x_{\pi(1)} \le \cdots \le x_{\pi(m)}$. (We give the comparisons the corresponding order $C_{\pi(1)} \le \cdots \le C_{\pi(m)}$.) We note that if $x^* > x_{\pi(j)}$, then $x^* > x_{\pi(i)}$, for $i \le j$, whereas if $x^* < x_{\pi(j)}$, then $x^* < x_{\pi(i)}$, for $i \ge j$. Hence by obtaining the answer to $C_{\pi(j)}$, we also resolve either $C_{\pi(1)}, \ldots, C_{\pi(j-1)}$ or $C_{\pi(j+1)}, \ldots, C_{\pi(m)}$. More precisely, if $f_{j_1}(x^*) < f_{j_2}(x^*)$ (answer "yes"), then $f_{i_1}(x^*) < f_{i_2}(x^*)$ for $i < j$ (answer "yes"), whereas if $f_{j_1}(x^*) > f_{j_2}(x^*)$ (answer "no"), then $f_{i_1}(x^*) > f_{i_2}(x^*)$ for $i > j$ (answer "no"). Finally, we observe that the relative order of two comparisons can be determined in $O(1)$ time.

Megiddo does not describe his technique as applying to the type of problem formalized above, for his technique is more general, and in fact so is our improvement. Nonetheless, we use this formulation both for simplicity and because many of the problems we consider have this form. Next we describe how Megiddo solves problem B, and then we explain our improvement.

Suppose we were to use a standard efficient sorting algorithm (running time $O(n \log n)$) to solve B. Then we would obtain an algorithm with running time $O(n \log nC(n))$. Megiddo's idea is to use a parallel sorting algorithm using $P(n)$

processors and parallel time $T(n)$. At each time step of the parallel algorithm, we have up to $P(n)$ comparisons to resolve. Instead of evaluating them one by one, we solve the median one. This immediately resolves half the comparisons. We repeat $\log(P(n))$ times ($=O(\log n)$ typically) and we thereby resolve all $P(n)$ comparisons. So we achieve a running time of $O(T(n)C(n) \log n)$ plus overheads for running the parallel computation and finding medians of sets of comparisons. The overheads for running the parallel algorithm are $O(P(n)T(n))$. In fact, as Megiddo observed, we can use a parallel algorithm which runs in time $T(n)$ in Valiant's model [16] so long as the overheads can be performed efficiently in a *serial* simulation. To find the median comparison, we use the fast median algorithm [1, pp. 97–99], running in time $O(P(n))$, assuming ordering two comparisons takes time $O(1)$. Since each time we halve the size of the set of unresolved comparisons, the time taken to find all $\log(P(n))$ medians that we need is also $O(P(n))$. So over $T(n)$ parallel time steps we take time $O(P(n)T(n))$ to find medians.

Thus the total running time of the algorithm for problem B is $O(P(n)T(n) + T(n)C(n) \log n)$. Typically, $P(n) = O(n \log n)$, $T(n) = O(\log n)$ [13], or $P(n) = O(n)$, $T(n) = O(\log n)$ [2]. (In the latter case the constant is very large.) When $C(n) = O(n)$ ($O(n \log n)$, respectively) each of these sorting algorithms gives an algorithm for problem B running in time $O(n \log^2 n)$ ($O(n \log^3 n)$, respectively).

Our improvement is to trim a factor of $\log n$ from these running times. We use the network of [2]; instead of performing the comparisons as described above, we play the game described in Section 2. We have to provide an adversary, which we do as follows. When the adversary is required to resolve a weighted half of the comparisons, we resolve the weighted median comparison, which by (2) above immediately resolves a weighted half of the comparisons. (Finding a weighted median of n items takes $O(n)$ time [15].) It is easy to see that the time taken to play the game, apart from the comparisons, is $O(n \log n)$. (The AKS network can be built in deterministic time $O(n \log n)$ [2, 6]—the constant is rather large, however.) Since the depth of the AKS network [2] is $O(\log n)$, we need perform only $O(\log n)$ comparisons. So we have a running time of $O(C(n) \log n + n \log n)$; for $C(n) = O(n)$ this is $O(n \log n)$, and for $C(n) = O(n \log n)$ it is $O(n \log^2 n)$.

In several of the applications problem A is not sorting; instead it is the problem of finding the minimum or of finding the median. Here Megiddo would use algorithms having $O(\log \log n)$ [15a, 16] and $O((\log \log n)^2)$ [3] parallel steps, respectively, and using $O(n/\log \log n)$ and $O(n)$ processors, respectively. These yield running times of $O(C(n) \log \log n \log n + n)$ and $O(C(n)(\log \log n)^2 \log n + n(\log \log n)^2)$, respectively, for problem B. Instead, by using a sorting algorithm we achieve a running time of $O(C(n) \log n + n \log n)$ for problem B, in both cases. (Note: We are making no assumptions about the size of $C(n)$, though for all problems considered so far $C(n) \geq O(n)$.)

In practice another approach can be taken. There are probabilistic parallel algorithms, running in constant time on $O(n)$ processors, for finding the minimum and the median [14]. Using these, and applying Megiddo's technique, we would solve problem B in $O(C(n) \log n + n)$ probabilistic time. The constant is much smaller than the one for the algorithm described in the previous paragraph, and in addition the algorithm is considerably simpler. (I am indebted to Megiddo for drawing this to my attention—I shall refer to this as Megiddo's probabilistic improvement.)

Remark. At this point we can explain the title. By contrast with Megiddo's technique, our solution initially slows down the network, in that some comparisons

at the top level of the network are resolved later. However, the complete sort proceeds faster.

There is another type of problem we encounter. It has two salient features:

(1) The problem can be solved by m independent binary searches, each on a set of n items, but each comparison takes $C(n)$ time.
(2) The batching rule applies (see the rules for problem B at the start of Section 3).

We use the same terminology as above. We solve this problem by playing a slightly different game. We (the player) assign weights to the active comparisons. Let the total weight of the active comparisons be W. The adversary is obliged to resolve comparisons having total weight $\geq W/2$. We assign weight 2^{1-j} to a comparison at level j in a binary search (i.e., the jth comparison in that particular search). One can show:

LEMMA 3. *At the start of the* $(k+1)$*st turn the active weight is bounded by* $(3/4)^k \cdot m$, *for* $k \geq 0$.

LEMMA 4. *For* $k \geq 3(j + \log m)$, *during the* $(k+1)$*st turn there are no active comparisons at level* j.

COROLLARY. *The game ends after* $O(\log m + \log n)$ *turns.*

It is straightforward to use this game to produce an algorithm with running time $O(C(n) \cdot (\log m + \log n) + m \cdot (\log m + \log n))$.

4. *Applications*

We discuss algorithms for several problems to which we have applied these improvements. They are drawn from [4], [9], [10], and [12]. Our descriptions of the algorithms are brief. To fully understand each of these algorithms the reader should refer to the original solution (given in the appropriate reference).

(1) *The Ham Sandwich Theorem in Two Dimensions* [4]. In the discrete ham sandwich theorem we are given a set of n blue points and a set of n red points, and we are to find a straight line dividing both sets into equal sized halves. The algorithm in [4] uses a parallel algorithm for finding medians, with $C(n) = O(n)$. So it runs in time $O(n \log n(\log \log n)^2)$. Our improvement yields a running time of $O(n \log n)$. If the two sets can be separated by a straight line, the problem can be solved in linear time [11].

(2) *Finding a Point in the Center in Two Dimensions* [4]. A center of a set of n points has the property that any line through it has at least $\lceil n/3 \rceil$ points to either side. Cole et al. [4] first solve the problem B_1 of determining on which side of a straight line the center lies, in time $O(n \log^3 n)$. Their solution uses a parallel sorting algorithm, with $C(n) = O(n \log n)$. Our method improves the $O(n \log^3 n)$ to a time of $O(n \log^2 n)$. Cole et al. [4] next solve B_2, a second sorting problem; the solution uses a parallel sorting algorithm with B_1 used to resolve comparisons. We save another factor of $O(\log n)$. So we solve B_2 in time $O(n \log^3 n)$ rather than $O(n \log^5 n)$. B_2 dominates the running time of the main algorithm.

(3) *Finding a Point in the Center in Three Dimensions* [4]. Here, a center of a set of n points has the property that any plane through it has at least $\lceil n/4 \rceil$ points to either side. The algorithm is similar to the one above. Cole et al. [4] first solve the problem B_1 of determining on which side of a plane the center lies; their

solution uses n parallel sorting algorithms, running in parallel, with $C(n) = O(n^2 \log n)$. Using essentially the solution for problem (2) they achieve a running time of $O(n^2 \log^5 n)$; using the improvements described in (2), we reduce this to $O(n^2 \log^3 n)$. Cole et al. [4] also solve B_2, a second sorting problem; we save a factor of $O(\log n)$ here, too. So we reduce the running time from $O(n^2 \log^7 n)$ to $O(n^2 \log^4 n)$.

(4) *A Spanning Tree Problem* [9]. A parallel sorting algorithm is used to sort the edges of the graph with each comparison costing $O(E \log \beta(V, E/V))$, where $\beta(x, y) = \textit{least } i : \log^{(i)} x \leq y$ (a comparison consists of finding a minimum spanning tree, which can be done in time $O(E \log \beta(V, E/V))$ [7]). So Megiddo obtains a running time of $O(E(\log V)^2 \log \beta(V, E/V))$, and our improvement yields a running time of $O(E \log V \log \beta(V, E/V))$. (In fact, in [9] a bound of $O(E(\log V)^2 \log \log V)$ is given, for the best algorithm for the minimum spanning tree, known at that time, ran in time $O(E \log \log V)$.)

(5) *A Scheduling Problem* [9]. A parallel sorting algorithm is used, with each comparison costing $O(n \log n)$. So Megiddo obtains a running time of $O(n \log^3 n)$, and our improvement yields a running time of $O(n \log^2 n)$.

(6) *The Minimum Ratio Cycle Problem* [9]. The problem is to find a cycle in a network with edge costs and edge times such that the ratio of the total cost to the total time of the edges on the cycle is minimized. We improve Megiddo's approach No. 2 [9, p. 858]. He uses a parallel algorithm to find the minimum of V items, each comparison taking $O(EV)$ time. This algorithm is iterated $O(\log V)$ times, taking time $O(V^3 \log V + EV(\log V)^2 \log \log V)$ overall, where $O(V^3 \log V)$ is the time for computations other than comparisons. Instead we use a sorting algorithm to find the minimum; together with our improvement of Megiddo's technique, this reduces the running time to $O(V^3 \log V + EV(\log V)^2)$.

We could apply Megiddo's probabilistic improvement instead to obtain a probabilistic algorithm for this problem, having the same running time as above, but with a smaller constant.

(7) *Max-Min Tree k-Partitioning Problem* [9]. The problem is: Given a tree T with n edges and a nonnegative weight associated with each vertex, delete k edges of T so as to maximize the weight of the lightest of the resulting subtrees (formed by the remaining edges). Megiddo solves the partitioning problem for a path in time $O(n \log^2 n)$; his solution requires n binary searches to be done in parallel on a set of n items, where each comparison takes time $O(n)$. We carry these out using the method described at the end of Section 3 and obtain an algorithm solving the partitioning problem for a path in time $O(n \log n)$. The partitioning problem for a tree is solved recursively, using the solution to the path partitioning problem to put the pieces together; Megiddo obtains a running time of $O(n \log^3 n)$; our improvement of the path partitioning algorithm reduces the running time to $O(n \log^2 n)$.

(8) *p-Center Problems* [5, 12]. See [12] for the problem definition. The solution in [12] is built on a searching problem described in the appendix of [12]. It has two phases: a sorting phase and a series of n binary searches running in parallel. For both of these the comparisons take $O(n)$ time. We apply the method of Section 2 to the sorting phase, and the improvement described at the end of Section 3 to the binary searches; this reduces their running time of $O(n \log^2 n)$ for the searching problem to $O(n \log n)$. As in (7), the *continuous p-center problem* is solved recursively, using the solution to the searching problem to put the pieces together. Again, Megiddo obtains a running time of $O(n \log^3 n)$ for finding the *continuous p-center*; our improvement to the algorithm for the searching problem reduces this

to $O(n \log^2 n)$. For $p \leq \log n$, the solution due to Fredrickson and Johnson [5] is better: it achieves time $O(np \log(n/p))$, for $p = o(n)$.

Remark. U. Vishkin (private communication) has found a method for doing n binary searches in parallel; it is as follows. Let L be the list of items being searched and let M be a sorted list of the items being sought. M and L are merged using Valiant's parallel algorithm [16]. It runs in $O(\log \log n)$ parallel steps. One could apply Megiddo's technique to this algorithm, but not our improvement. For both problems 7 and 8 this would yield slightly slower algorithms.

5. *Further Remarks*

We comment now on the efficiency of the algorithms described above. All these algorithms use the sorting algorithm based on the AKS network [2]. This has a very large constant, making these algorithms impractical. However, we could implement the AKS network [2] probabilistically and then the constants become less unreasonable, while the order of the expected running time is unchanged. In addition, when a median is required, as a rule an approximation to the median will suffice; so if we are using a probabilistic algorithm we can take a random item instead. This may not improve the order of the run time, but in practice it will result in a simpler and more efficient algorithm. Also, we could apply Megiddo's probabilistic improvement.

Unfortunately, we cannot apply our improvement to the sorting algorithms of Valiant [16] or Preparata [13], for these do not yield sorting networks of small enough depth. Alternatively, we can consider these algorithms to define directed acyclic graphs (see the remark at the end of Section 2): A node is a pair $[C, t]$ comprising a comparison C at the tth step. An edge connects node $[C, t-1]$ to node $[D, t]$, if an output of comparison C at time $t-1$ might be an input to comparison D at time t. The graphs induced by these algorithms have maximum fanin and fanout equal to $\Theta(n^{1/2})$. Thus, when applying the remark at the end of Section 2, our technique will produce an algorithm that uses $O(\log^2 n)$ turns. This results in an algorithm no faster than the algorithm produced by Megiddo's technique. An efficient parallel sorting algorithm ($O(n)$ processors, $O(\log n)$ running time), with bounded fanout (or fanin), would enable all the algorithms described above to be implemented more efficiently.

We believe that other searching problems, particularly those with a geometric flavor, will benefit from applying these techniques.

ACKNOWLEDGMENTS. It is a pleasure to acknowledge the conversations I had with Micha Sharir at the start of this work. He contributed to a precise definition of the problem. Also, I am grateful to both Uzi Vishkin and Nimrod Megiddo for carefully reading the paper and making suggestions which led to considerable improvements in the presentation. The referees also provided several helpful suggestions. Any remaining errors are solely the author's responsibility.

REFERENCES

1. AHO, A., HOPCROFT, J., AND ULLMAN, J. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. AJTAI, M., KOMLOS, J., AND SZEMEREDI, E. An $O(n \log n)$ sorting network. *Combinatorica 3* (1983), 1–19.
3. COLE, R., AND YAP, C. A parallel median algorithm. *Inf. Process. Lett. 20* (1985), 137–139.
4. COLE, R., SHARIR, M., AND YAP, C. On k-hulls and related problems. *SIAM J. Comput.* to appear.

5. FREDERICKSON, G., AND JOHNSON, D. Finding k-th paths and p-centers by generating and searching good data structures. *J. Algorithms 4* (1983), 61–80.
6. GABBER, O., AND GALIL, Z. Explicit construction of linear size superconcentrators. *J. Comput. Syst. Sci. 22* (1981), 407–420.
7. GABOW, H., GALIL, Z., AND SPENCER, T. Efficient implementation of graph algorithms using contraction. In *Proceedings of the 25th Annual Symposium on the Foundations of Computer Science.* IEEE, New York, 1984, 347–357.
8. MEGIDDO, N. Combinatorial optimization with rational objective functions. *Math. Oper. Res. 4* (1979), 414–424.
9. MEGIDDO, N. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM 30*, 4 (1983), 852–865.
10. MEGIDDO, N. Linear time algorithms for linear programming in R^3 and related problems. *SIAM J. Comput. 12* (1983), 759–776.
11. MEGIDDO, N. Partitioning with two lines in the plane. *J. Algorithms 6* (1985), 430–433.
12. MEGIDDO, N., AND TAMIR, A. New results on the complexity of p-center problems. *SIAM J. Comput. 12*, 4 (1983), 751–758.
13. PREPARATA, F. New parallel sorting schemes. *IEEE Trans. Comput. C-27* (1978), 669–673.
14. REISCHUK, R. A fast probabilistic sorting algorithm. In *Proceedings of the 22nd Annual Symposium on the Foundations of Computer Science.* IEEE, New York, 1981, 212–219.
15. REISER, A. A linear selection algorithm for sets of elements with weights. *Inf. Process. Lett. 7* (1978), 159–162.
15a. SHILOACH, Y., AND VISHKIN, U. Finding the maximum, merging and sorting in a parallel computation model. *J. Algorithms 2* (1981), 88–102.
16. VALIANT, L. Parallelism in comparison problems. *SIAM J. Comput. 4* (1975), 348–355.

RECEIVED NOVEMBER 1984; REVISED JANUARY 1985; ACCEPTED JUNE 1986