‘‘An Optimal Selection Algorithm’’

by

Richard Cole

Ultracomputer Note #97

Technical Report #209

March, 1986

This work was supported in part by NSF grant DCR-84-01633 and by an IBM faculty development award.

ABSTRACT

We give an optimal parallel algorithm for selection on the EREW PRAM. It requires a linear number of operations and $O(\log n \log^* n)$ time.

1. Introduction

The models of parallel computation used in this paper are the exclusive-read exclusive-write (EREW), and the concurrent-read exclusive-write (CREW) parallel random access machine (PRAM). A PRAM employs p synchronous processors all having access to a common memory. An EREW PRAM does not allow simultaneous access by more than one processor to the same memory location for read or write purposes, while a CREW allows concurrent access for reads but not for writes. See [Vi-83b] for a survey of results concerning PRAMs.

Let $Seq(n)$ be the fastest known worst-case running time of a sequential algorithm, where n is the length of the input for the problem at hand. Obviously, the best upper bound on the parallel time achievable using p processors, without improving the sequential result, is of the form $O(Seq(n)/p)$. A parallel algorithm that achieves this running time is said to have *optimal speed-up* or more simply to be *optimal.* A primary goal in parallel computation is to design optimal algorithms that also run as fast as possible. The following notions of ''fast'' and ''efficient'' are in line with this goal. Given a parallel algorithm that runs in time T using p processors, we say that it performs (a total of) pT operations. (Recall that a theorem due to [B-74] implies that, under quite general conditions, any algorithm that performs x operations in time t can be implemented so as to achieve time $O(t)$ using x/t processors.) Let A (resp. *B*) be a parallel algorithm that performs x_1 operations (resp. x_2 operations) in time T_1 (resp. time T_2) on a given input. We say that algorithm A is *more efficient* than algorithm B if $x_1 \leq x_2$ (regardless of the relation between T_1 and T_2). Also, algorithm A is *faster* than algorithm B if $T_1 \leq T_2$ (regardless of the relation between x_1 and x_2).

In view of our interest in efficiency we shall give all our results in terms of the following two parameters: operations performed and running time. To obtain the number of processors used, simply divide the operation count by the running time. This presentation has two advantages. First it provides for a comparison 'at a glance' with the best sequential algorithm. Second, it may make the description of the algorithm simpler: at times it is convenient to describe the algorithm as if different numbers of processors are being used for different steps of the algorithm, and simply to count the number of operations performed. We are assured, by Brent's theorem [B-74], that it is straightforward to simulate such an algorithm by a uniform number of processors. We remark that Brent's theorem ignores the following implementation issue: how to assign operations to processors. But for the algorithm given here this will present no difficulty. The details will be left to the reader.

We give an EREW PRAM algorithm for selection that uses $O(n)$ operations and $O(\log n \log^* n)$ time; it assumes that the input is provided in an array of length n. The algorithm makes use of the [AKS-83] sorting network. However, if the algorithm is implemented on a CREW PRAM we do not need to use this sorting network; the sorting algorithm of [BH-82] will suffice. The advantage of the latter sorting algorithm is that the constants in the running time are much smaller.

The previous best result for selection on the PRAM was the following: $O(n)$ operations in time $O(\log n \log\log n)$ [Vi-83a]. Much previous work on parallel algorithms for the selection problem has concentrated on Valiant's comparison model [V-75]. In this model, only the time used for comparisons is counted. The following results have been achieved: with n processors, a lower bound of $\Omega(\log\log n)$ time [V-75], an upper bound of $O([\log\log n]^2)$ time [CY-85], and an upper bound of $O(\log\log n)$ time [KSS-86]. Also, in this model, [R-81] gave an $O(1)$ time probabilistic algorithm for n processors.

Our algorithm, like the algorithm of [CY-85], uses a procedure for computing approximations to the k-th smallest item. The selection algorithm itself is described in section 2. We give the approximation algorithm in section 3. In order to simplify the presentation we assume that all the items are distinct. Also, we ignore all rounding errors.

2. The selection algorithm

We use the following notation: $\log^{(i)}$ denotes the i-th iteration of the log function. We define $\log^* n$ to be the least i such that $\log^{(i)} n \le 1$. Also, we introduce the following function f: $f(1) = 1$, and $f(i+1) = 2^{f(i)}$, for $i > 1$. We note that $\log n \le f(\log^* n) < n$, for $n \ge 2$.

We use an approximate selection algorithm with the following input/output relation, for $i \ge 2$.

Input: A set of $n/[f(i)]^2$ items in an array, and an integer k.
Output: Two items, one smaller than the k-th input item, and one larger than the k-th input item. We also specify the accuracy of the approximation. The smaller item is at least the $[k - \frac{n}{2[f(i+1)]^2}]$-th input item, while the larger item is at most the $[k + \frac{n}{2[f(i+1)]^2}]$-th input item.

The approximation algorithm requires $O(n/f(i))$ operations and time $O(\log n)$.

The selection algorithm has the following form.

1) **do** $i = 1$ **to** $\log^* n - 1$

(i) Compute the 2 approximations to the k-th smallest item, one smaller and one larger than the k-th smallest item.

(ii) Let them have ranks k_1 and k_2.

(iii) Extract the items strictly between the k_1-th and k_2-th items.

(iv) Place the extracted items into an array of size $n/[f(i+1)]^2$.

(v) $k := k - k_1$.

od;

2) Sort the remaining $\le n/\log^2 n$ items using the [AKS-83] sorting network; the k-th item is the item sought.

The i-th iteration of the selection algorithm requires $O(n/f(i))$ operations and time $O(\log n)$. The final sort requires $O(n/\log n)$ operations and time $O(\log n)$. Thus, the selection algorithm requires $O(n)$ operations and time $O(\log n \log^* n)$.

Remark 1: We can achieve a runtime of $O(h \log n)$ at the cost of using $O(n \log^{(h)} n)$ operations, as follows. We use essentially the algorithm given above; however, the first iteration of the loop will produce approximations to the k-th item with an error of at most $\frac{n}{2[f(\log^* n - h)]^2}$. This is achieved by using $n\,[f(\log^* n - h - 1)]^2 < n[\log^{(h+1)} n]^2 \le n \log^{(h)} n$ processors to compute the initial approximations.

Remark 2: The structure of the selection algorithm is very similar to that of the list ranking algorithm of [CV-86a]. In [CV-86b] this structure is called an accelerating cascade.

3. The approximation algorithm

We discuss how to find a lower approximation to the k-th item. Finding a higher approximation is a symmetric operation. Our algorithm will use $O(n/f(i))$ operations and time $O(\log n)$ if the input is of size at most $n/[f(i)]^2$; the error in the approximation will be at most $\frac{n}{2[f(i+1)]^2}$. (In the event that $k \le \frac{n}{2[f(i+1)]^2}$, we choose the lower approximation to be the (non-existent) 0-th item with a value of $-\infty$.) It is assumed that the input is presented in an array of size equal to the input size. If $i = \log^* n$ the selection is achieved by sorting the input using the [AKS-83] sorting network. For smaller values of i we proceed as follows. We use a reduction procedure with the following input/output relation.

Input: A set S_i of size at most $n/[f(i)]^2$, and an integer k_i. (A lower approximation to the k_i-th item in S_i is being sought; the error is to be at most $\frac{n}{2[f(i)]^2[f(i+1)]^2}$.)

Output: A set S_{i+1} of size at most $n/[f(i+1)]^2$, and an integer k_{i+1}. (A lower approximation to the k_{i+1}-th item in S_{i+1} is to be found; the error is to be at most $\frac{n}{2[f(i+1)]^2[f(i+2)]^2}$. This approximation will also be a lower approximation to the k_i-th item in S_i, with error at most $\frac{n}{2[f(i)]^2[f(i+1)]^2}$.)

The input and output will be provided in arrays of size $|S_i|$ and $|S_{i+1}|$ respectively. The reduction procedure runs in time $O(f(i))$, and uses $O(n/f(i))$ operations.

Method: Form sets of size $4[f(i+1)]^4$ items (take contiguous segments of items from the input array). Sort each set using an [AKS-83] sorting network. The output set comprises every $[f(i+1)]^2$-th item in each sorted set. The output is provided in an array: namely every $[f(i+1)]^2$-th item in the input array, following the sorting.

Note that the r-th item in S_{i+1} is at least the $r[f(i+1)]^2$-th item in S_i, and is less than the $r[f(i+1)]^2 + \frac{n}{4[f(i)]^2[f(i+1)]^2}$-th item in S_i. We choose $k_{i+1} = \frac{k_i}{[f(i+1)]^2} - \frac{n}{4[f(i)]^2[f(i+1)]^4}$. So the k_{i+1}-th item in S_{i+1} is at least the $k_i - \frac{n}{4[f(i)]^2[f(i+1)]^2}$-th item in S_i, and is less than the k_i-th item in S_i.

We apply the reduction procedure recursively to obtain the approximation; let e be the item selected.

Lemma 1: Item e is at least the $k_i - \frac{n}{2[f(i)]^2[f(i+1)]^2}$-th item in S_i.

Proof: We prove the result by induction on $j = \log^* n - i$. Clearly, the base case, $j = 0$ ($i = \log^* n$), is true, since the set $S_{\log^* n}$ is sorted, so the item selected is the desired item. For the inductive step, suppose that e is at least the $k_{i+1} - \frac{n}{2[f(i+1)]^2[f(i+2)]^2}$-th item in S_{i+1}. Then, in S_i, e is at least the $k_i - \frac{n}{4[f(i)]^2[f(i+1)]^2} - \frac{n}{2[f(i+2)]^2}$-th item, that is, at least the $k_i - \frac{n}{2[f(i)]^2[f(i+1)]^2}$-th item. □

Corollary 1: Item e is at least the $k_i - \frac{n}{2[f(i+1)]^2}$-th item in S_i.

Lemma 2: Item e is smaller than the k_i-th item in S_i, for $i < \log^* n$.

Proof: This follows immediately from the fact that the k_{i+1}-th item in S_{i+1} is less than the k_i-th item in S_i. □

We note that the approximation algorithm takes $O(\log n)$ time and performs $O(n/f(i))$ operations.

Remark: We can avoid using the [AKS-83] network, by replacing it with Borodin and Hopcroft's parallel sorting algorithm, which runs in time $O(\log n)$ on $n \log n$ processors in the CREW PRAM model. In the selection algorithm above, we replace each term $[f(i)]^2$ by a term $[f(i)]^3$, and each term $[f(i)]^4$ by a term $[f(i)]^6$. The analysis is identical (on performing the textual substitutions there also).

Acknowledgements.

I would like to thank Uzi Vishkin for having read and commented on an earlier version of this paper.

References

[AKS-83] M. Ajtai, J. Komlos, E. Szemeredi, ‘‘An $O(n \log n)$ sorting network’’, *Combinatorica*, 3(1983), 1-19.

[B-74] P. Brent, ‘‘The optimal evaluation of general arithmetic expressions’’, *JACM* 21,2(1974), 201-206.

[BH-82] A. Borodin and J. Hopcroft, ‘‘Routing, merging and sorting on parallel models of computation’’, *Proc. Fourteenth Annual Symp. on Theory of Computing*, 338-344.

[CV-86a] R. Cole and U. Vishkin, ‘‘Deterministic coin tossing with application to optimal list ranking’’, submitted.

[CV-86b] R. Cole and U. Vishkin, ‘‘Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms’’, to be presented, *Eighteenth Annual ACM Symp. on the Theory of Computing*.

[CY-85] R. Cole and C. Yap, ‘‘A parallel median algorithm’’, *IPL*, 20(1985), 137-139.

[KSS-86] J. Komlos, W.L. Steiger, E. Szemeredi, ‘‘Deterministic selection in $O(\log \log n)$ time’’, to be presented, *Eighteenth Annual ACM Symp. on the Theory of Computing*.

[R-81] R. Reischuk, ‘‘A Fast Probabilistic Sorting Algorithm’’, *Proc. Twenty Second Annual Foundations of Computer Science Conf.*, 212-219.

[V-75] L. Valiant, ‘‘Parallelism in Comparison Problems’’, *SIAM J. Comput.*, Vol.4, No.3, 348-355.

[Vi-83a] U. Vishkin, ‘‘An optimal parallel algorithm for selection’’, manuscript.

[Vi-83b] U. Vishkin, ‘‘Synchronous parallel computation - a survey’’, TR 71, Dept. of Computer Science, NYU, 1983.