An Optimal Parallel Algorithm for the Multiselection Problem

Muhammad H. Alsuwaiyel
Department of Information and Computer Science
King Fahd University of Petroleum & Minerals
Dhahran 31261, Saudi Arabia
e-mail: suwaiyel@ccse.kfupm.edu.sa

Abstract Given a set \( S \) of \( n \) elements drawn from a linearly ordered set, and a set \( K = \{k_1, k_2, \ldots, k_r\} \) of positive integers between 1 and \( n \), the multiselection problem is to select the \( k_i \)th smallest element for all values of \( i, 1 \leq i \leq r \). We present a simple optimal algorithm to solve this problem that runs in \( O(n^\epsilon \log r) \) time on the EREW PRAM with \( n^{1-\epsilon} \) processors, \( 0 < \epsilon < 1 \).

1 Introduction

Let \( S \) be a set of \( n \) elements drawn from a linearly ordered set, and let \( K = \{k_1, k_2, \ldots, k_r\} \) be a set of positive integers between 1 and \( n \), that is a set of ranks. The multiselection problem is to select the \( k_i \)th smallest element for all values of \( i, 1 \leq i \leq r \). If \( r = 1 \), then we have the classical selection problem. On the other hand, if \( r = n \), then the problem is tantamount to the problem of sorting. Recently, Shen[4] presented an optimal parallel algorithm for multiselection that runs in time \( O(n^\epsilon \log r) \) on the EREW PRAM with \( n^{1-\epsilon} \) processors, \( 0 < \epsilon < 1 \). We will show that the multiselection problem can easily be solved using an adaptive algorithm that runs in \( O(n^\epsilon \log r) \) time on the EREW PRAM with \( n^{1-\epsilon} \) processors, \( 0 < \epsilon < 1 \), by slightly modifying the adaptive parallel quicksort algorithm in [1, 2]. The algorithm that will be presented can be thought of as a generalization of the paradigm given in [4].

Due to the resemblance of the multiselection problem to the sorting problem, it is natural to ask if an algorithm for the latter can be modified so that it solves the former. It turns out that a slight modification of the parallel quicksort algorithm results in an optimal algorithm for the multiselection problem. Let \( \text{select} \) be a \( \Theta(n) \) sequential selection algorithm. The algorithm shown below solves the multiselection problem in time \( \Theta(n \log r) \). We will assume that the set of ranks \( K = \{k_1, k_2, \ldots, k_r\} \) is sorted in increasing order. If not, then it can be sorted in \( \Theta(r \log r) \) time.
Henceforth, \( S_j \) will denote a subset of \( S \) and \( S[j] \) will denote the \( j \)th smallest element in \( S \). Similarly, \( K_j \) will denote a subset of \( K \) and \( k_j \) will denote the \( j \)th smallest element in \( K \). For simplicity, we will assume that the elements in \( S \) are distinct.

**Algorithm** \( mselect(S, K) \)

1. if \( K \) is not empty then
2.   if \( K = \{k\} \) then return \( select(S, k) \)
3.   else
4.     \( r \leftarrow |K| \)
5.     \( w \leftarrow k_{\lceil r/2 \rceil} \)
6.     \( select(S, w) \)
7.     \( S_1 = \{x \in S \mid x < S[w]\} \)
8.     \( S_2 = \{x \in S \mid x > S[w]\} \)
9.     \( K_1 = \{k_1, k_2, \ldots, k_{w-1}\} \)
10.    \( K_2 = \{k_{\lceil r/2 \rceil+1} - w, k_{\lceil r/2 \rceil+2} - w, \ldots, k_r - w\} \)
11.    \( mselect(S_1, K_1) \)
12.    \( mselect(S_2, K_2) \)
13. end if
14. end if

Since the recursion depth is \( \lceil \log r \rceil \), the time complexity of Algorithm \( mselect \) is \( \Theta(n \log r) \).

As to the lower bound for multiselection, suppose that it is \( o(n \log r) \). Then, by letting \( r = n \), we would be able to sort \( n \) elements in \( o(n \log n) \) time, contradicting the \( \Omega(n \log n) \) lower bound for sorting on the decision model of computation. This lower bound has been previously established in [3] in the context of heap operations. It follows that the multiselection problem is \( \Omega(n \log r) \), and hence the algorithm given above is optimal.

## 2 The parallel algorithm

In this section we show that the multiselection problem can be solved in \( O(n^\varepsilon \log r) \) time on the EREW PRAM with \( n^{1-\varepsilon} \) processors, \( 0 < \varepsilon < 1 \), by slightly modifying the adaptive parallel quicksort algorithm in [1, 2]. The algorithm uses the parallel selection algorithm in [4], which we will call
select. Algorithm select runs in time $O(n^\epsilon)$ using $N = n^{1-\epsilon}$ processors on the EREW PRAM.

Let $N$ be the number of processors used in the multiselection algorithm, where $1 < N < n$. Write $N = n^{1-\epsilon}$. Let $q$ be an appropriately chosen small positive integer greater than 1, say $q = \min\{r, 8\}$. The algorithm works as follows. Let $p = \lceil r/q \rceil$. For brevity, we will define $k_0 = 0$ and $S[k_0] = -\infty$.

First, Algorithm select finds and outputs the elements

$$S' = \{S[k_{jp}] \mid 1 \leq j \leq q - 1\}$$

of ranks in the set

$$K' = \{k_{jp} \mid 1 \leq j \leq q - 1\}.$$  

The set $S - S'$ is then partitioned into $q$ subsets: $S_1, S_2, \ldots, S_q$, where

$$S_j = \{x \in S \mid S[k_{(j-1)p+1}] < x < S[k_{jp}]\}$$

for $1 \leq j \leq q - 1$, and

$$S_q = \{x \in S \mid x > S[k_{(q-1)p}]\}.$$  

Similarly, the set of ranks $K - K'$ is partitioned into $q$ subsets $K_1, K_2, \ldots, K_q$, where

$$K_j = \{k \in K \mid (j-1)p < k < jp\}$$

for $1 \leq j \leq q - 1$, and

$$K_q = \{k \in K \mid (q-1)p < k \leq r\}.$$  

The algorithm is then recursively called in parallel on the $q$ pairs $(S_j, K_j)$, $1 \leq j \leq q$, where the number of processors used in each recursive call is proportional to the size of the subset $S_j$, that is $N[S_j]/|S|$.

Thus, the underlying principle is a generalization of the paradigm given in [4], in which both the set of elements and the set of ranks are divided into two parts. Here, they are divided into $q$ parts, and $q$ can be tuned for optimum performance. The detailed algorithm is given below.

**Algorithm mselect($S, K, N$)**

1. if $|K| \leq q$ then
2. for $j \leftarrow 1$ to $|K|$ do
3. \hspace{1em} select($S, k_j, N$)
4. output $S[k_j]$
5. end for
6. else
7. $p \leftarrow \lceil |K|/q \rceil$
8. $w \leftarrow k_0$
9. for $j \leftarrow 1$ to $q - 1$
do
10. select($S, k_j p, N$)
11. output $S[k_j p]$
12. $S_j \leftarrow \{x \in S \mid S[k_{(j-1)p}] < x < S[k_j p]\}$
13. $K_j \leftarrow \{k_{(j-1)p+1} - w, k_{(j-1)p+2} - w, \ldots, k_{jp-1} - w\}$
14. $w \leftarrow k_{jp}$
15. end for
16. $S_q \leftarrow \{x \in S \mid x > S[k_{(q-1)p}]\}$
17. $K_q \leftarrow \{k_{(q-1)p+1} - w, k_{(q-1)p+2} - w, \ldots, k_r - w\}$
18. for $j \leftarrow 1$ to $q$ do in parallel
19. mselect($S_j, K_j, N \mid S_j \mid / \mid S \mid$)
20. end for
21. end if

It is not hard to see that Algorithm mselect works correctly. We now analyze its time complexity. Each call to Algorithm select in lines 3 and 10 takes $O(n^\epsilon)$ using $n^{1-\epsilon}$ processors [4]. After each call select($S, k_j p, N$), $1 \leq j < q$, in line 10, we extract $S_j$ by marking those elements between (and not including) $S[k_{(j-1)p}]$ and $S[k_{jp}]$, and extracting them using the parallel prefix algorithm and compaction using all allocated processors. That is, each processor works on $n^\epsilon$ elements and marks those elements between $S[k_{(j-1)p}]$ and $S[k_{jp}]$. This is followed by applying parallel prefix and compaction. Hence, the time required to construct all $S_j$’s is $O(q(n^\epsilon + \log n^{1-\epsilon})) = O(qn^\epsilon)$. Since $K$ is sorted, $K_j$ is constructed by extracting those elements greater than $j_{p-1}$ and less than $j_p$ in $O(q)$ time. For each recursive call, the number of processors is

$$\frac{N \mid S_j \mid}{n} = \frac{n^{1-\epsilon} \mid S_j \mid}{n} = \frac{\mid S_j \mid}{n^\epsilon}.$$
Hence, the ratio of the number of elements to the number of processors is

\[
\frac{|S_j|}{|S_j|/n^\epsilon} = n^\epsilon.
\]

As shown in [4], each call to Algorithm SELECT takes \(O(n^\epsilon)\) time. It follows that the overall running time of Algorithm MSELECT is governed by the recurrence:

\[
t(r, n) = t(r/q, n) + O(qn^\epsilon) + O(q \log n).
\]

The solution to this recurrence is

\[
t(r, n) = O(qn^\epsilon \log_q r) = O(n^\epsilon \log r),
\]

and hence the cost of the algorithm is \(O(n \log r)\).

References


