1990

Efficient Parallel Binary Search on Sorted Arrays

Danny Z. Chen

Report Number:
90-1009

EFFICIENT PARALLEL BINARY SEARCH ON SORTED ARRAYS

Danny Z. Chen

CSD-TR-1009
August 1990

Efficient Parallel Binary Search on Sorted Arrays

Danny Z. Chen*

Abstract

Let A be an array of n numbers and B an array of m numbers, where A and B are sorted and $n < m$. We consider the problem of determining for each element $A(j)$, $1 \leq j \leq n$, the element $B(i)$ such that $B(i) \leq A(j) < B(i+1)$, where $0 \leq i \leq m$ (with $B(0) = -\infty$ and $B(m+1) = +\infty$). Efficient parallel algorithms on the EREW-PRAM for this problem have been given [1, 8]. In this paper, we present a parallel algorithm to solve it in $O(\log m)$ time using $O((n \log(m/n))/\log m)$ EREW-PRAM processors. Our solution improves the previous known results either on the time or on the total work complexity, and it can be used to obtain a different parallel algorithm for merging two sorted arrays of size m each in $O(\log m)$ time using $O(m/\log m)$ EREW-PRAM processors.

1 Introduction

Given an array A of n numbers and an array B of m numbers, where A and B are sorted and $n < m$, we consider the problem of finding for each element $A(j)$, $1 \leq j \leq n$, the unique element $B(i)$ such that $B(i) \leq A(j) < B(i+1)$, $0 \leq i \leq m$ (with $B(0) = -\infty$ and $B(m+1) = +\infty$). This problem is called the *multiple search problem* (MSP) [1]. In this paper, we show how to efficiently solve this problem in parallel on the EREW-PRAM computational model. (Recall that the EREW-PRAM is a synchronous parallel computational model in which all processors share a common memory, each processor can access any memory address in constant time, and no more than one processor is allowed to simultaneously access the same memory address.)

Sequentially, the MSP can be easily solved in $O(m)$ time by merging A and B. When n is very small comparing to m, that is, if $O(n)$ is $o(m/\log m)$, it is preferable to solve the problem in $O(n \log m)$ time by performing n binary searches in B. It has also been shown that this problem can be solved in $O(n \log(m/n))$ *expected* time [6] (cited from [1]), which is better than $O(n \log m)$ when $O(\log(m/n))$ is $o(\log m)$. In the next section, we will show

*Dept. of Computer Science, Purdue University, West Lafayette, IN 47907. This research was partially supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, the National Science Foundation under Grant DCR-8451393, and the National Library of Medicine under Grant R01-LM05118.

how to solve the MSP by a sequential algorithm whose time complexity in the *worst case* is $O(n \log(m/n))$.

There have also been parallel solutions to the MSP in the PRAM models. It is trivial to solve the MSP on the CREW-PRAM (in which concurrent reads from the same memory address by multiple processors are allowed) in $O(\log m)$ time using $O(n)$ processors. On the EREW-PRAM, one can solve the problem in $O(\log m)$ time using $O(m/\log m)$ processors by doing parallel merging [3, 4]. There were also parallel solutions that avoid using parallel merging. In [1], two such results based on binary search were presented: one uses n EREW-PRAM processors and takes $O((\log m \log n)/\log\log m)$ time, and the other runs in $O((\log m \log p)/\log\log m + (n \log m)/p)$ time using p EREW-PRAM processors. Another solution was given in [8] that runs in $O(\log m + \log^2 p + (n \log m)/p)$ time using p EREW-PRAM processors.

We present an algorithm solving the MSP in $O(\log m)$ time using $O((n \log(m/n))/\log m)$ EREW-PRAM processors. Hence we improve the results in [1, 8] either on the time or on the total work complexity (the total work of a parallel algorithm is its *time* $\times$ *processor* product). Our parallel algorithm has a total work complexity of $O(n \log(m/n))$, which matches the time bound of the sequential algorithm that we give in Section 2. Using this solution for the MSP, we can easily obtain a parallel algorithm for merging two sorted arrays of size m each in $O(\log m)$ time using $O(m/\log m)$ EREW-PRAM processors. Notice that optimal EREW-PRAM algorithms for merging in parallel have been known [3, 4]. This parallel merge algorithm is different from the ones in [3, 4] and it has the same optimal complexity bounds as those in [3, 4]. Our algorithm can be easily simulated by using $p \leq O((n \log(m/n))/\log m)$ EREW-PRAM processors without any increase on the asymptotic complexity of the total work (by Brent's theorem [2]).

We first give a parallel algorithm which runs in $O(\log m)$ time using n EREW-PRAM processors (Subsection 3.1). The basic idea of this algorithm comes from the parallel searching scheme on 2-3 trees of [9]. In our problem, however, the scheme of [9] can not be directly applied since we do not want to explicitly construct a 2-3 tree for array B (such construction would already require $O(m)$ total work which in general can not be accomplished by n processors in $O(\log m)$ time). Notice that in a 2-3 tree the data in many leaves may have multiple copies of their values stored in the internal nodes of the tree, while in the case of arrays, there are no multiple copies of the array elements available. Hence we must modify the scheme of [9]. What we do is to make some copies of appropriate elements

in B in order to avoid read conflicts in the parallel search, and such copies are created when they are needed in the process of searching (see Subsection 3.1). Then we use this algorithm to obtain another algorithm which still takes $O(\log m)$ time but uses less (i.e., $O((n \log(m/n))/\log m)$) processors (Subsection 3.2). A parallel merge algorithm is outlined in Section 4.

2 Preliminaries

We let the n EREW-PRAM processors be $P(1)$, $P(2)$, ..., $P(n)$. We denote a consecutive sequence of processors $P(i)$, $P(i+1)$, ..., $P(j)$, $i \leq j$, by $P[i,j]$ and we call $P[i,j]$ a *P-block*. Similarly, we denote a consecutive sequence of elements in A (resp., B) by $A[i,j]$ (resp., $B[i,j]$) and call it an *A-block* (resp., *B-block*).

Without loss of generality, we assume A and B are sorted in nondecreasing order. We also assume $A(i) < A(j)$ for all $i < j$. That is, A contains distinct elements. (If it is not the case, then we can use parallel prefix [5, 7] to eliminate the repetitions of the elements in A with $O(n/\log n)$ processors and in $O(\log n)$ time.) We still use n to denote the size of A so resulted. We let each processor $P(i)$ have element $A(i)$.

It is sufficient to find, for each index j of A, the unique index i such that $B(i) \leq A(j) < B(i+1)$, $0 \leq i \leq m$ (with $B(0) = -\infty$ and $B(m+1) = +\infty$). This is because after the appropriate indices of B are known to the elements of A, the elements of A can find the values of the corresponding elements of B by parallel prefix [5, 7].

As in the searching algorithm of [9], the processors are grouped into disjoint *P-blocks*. During the computation, a *P-block* is often bisected into two *P-blocks* of equal size (e.g., $P[i,j]$, $i < j$, is bisected into $P[i,k]$ and $P[k+1,j]$). For each *P-block* $P[i,j]$, only one processor, say $P(i)$, is active (i.e., it is the only processor that does the primary computation for the search of $P[i,j]$ so long as $P[i,j]$ is not bisected); the rest processors in $P[i,j]$ are all idle. When $P[i,j]$, $i < j$, is bisected into $P[i,k]$ and $P[k+1,j]$, $P(i)$ "wakes" $P(k+1)$ up and let $P(k+1)$ handle $P[k+1,j]$ while $P(i)$ handles $P[i,k]$.

Here we show how to solve the MSP sequentially in $O(n \log(m/n))$ time. We combine the merge and binary search algorithms together as follows: first partition B into n sub-arrays of size m/n each by selecting n elements from B (this is easy to do in arrays); then merge the n selected elements from B with A (hence we know for each element $A(j)$ which sub-array of B into that $A(j)$ falls); finally do a binary search for every element of A in the

sub-array of B into that it falls in order to find for it the appropriate element of B. This method obviously requires $O(n \log(m/n))$ time in the worst case because the partition and merge take only $O(n)$ time and n binary searches are performed (each binary search is of $O(\log(m/n))$ time).

Notice that the $O(n \log(m/n))$ sequential time algorithm is better than the $O(n \log m)$ time algorithm when $O(\log(m/n))$ is $o(\log m)$.

3 The Parallel Search Algorithms

We present two algorithms solving the MSP. The first one, which runs in $O(\log m)$ time using $O(n)$ processors, is described in Subsection 3.1. Then we use this algorithm to obtain another algorithm in Subsection 3.2 which has the same time complexity as the one in Subsection 3.1 but uses only $O((n \log(m/n))/\log m)$ processors.

3.1 The First Algorithm for the MSP

The following is the algorithm with $O(\log m)$ time and $O(n)$ processors.

Procedure *Alg-MSP*$(B[i,j], P[k,l], F, L)$.

Input. $B[i,j]$, $i \leq j$, $P[k,l]$, $k \leq l$, and values F and L (F is a copy of $B(i)$ and L a copy of $B(j)$).

Case 1. $B[i,j]$ has no more than two elements. This case is easy. (For example, one can first do binary search for each element of $B[i,j]$ in $A[k,l]$ to find the appropriate location of the element of B in $A[k,l]$, and then use parallel prefix to assign the indices of B so found to the elements of $A[k,l]$.)

Case 2. $P[k,l]$ has at least two processors. Let m_a (resp., m_b) be the index of A (resp., B) that partitions $A[k,l]$ (resp., $B[i,j]$) into two sub-blocks of equal size. Let v_b be a copy of $B(m_b)$. There are five subcases.

Subcase 2.1. $A(l) < F$. Then the index for all elements of $A[k,l]$ is $i-1$.

Subcase 2.2. $L \leq A(k)$. Then the index for all elements of $A[k,l]$ is j.

Subcase 2.3. $v_b \leq A(m_a)$. Then in parallel, call *Alg-MSP*$(B[i,j], P[k,m_a], a_1, a_2)$, where a_1 is a copy of F and a_2 a copy of L, and call *Alg-MSP*$(B[m_b+1,j], P[m_a+1,l], a_3, a_4)$, where a_3 is a copy of $B(m_b+1)$ and a_4 a copy of L.

Subcase 2.4. $A(m_a) < v_b < A(m_a + 1)$. Then in parallel, call *Alg-MSP*$(B[i, m_b - 1], P[k, m_a], a_1, a_2)$, where a_1 is a copy of F and a_2 a copy of $B(m_b - 1)$, and call *Alg-MSP*$(B[m_b + 1, j], P[m_a + 1, l], a_3, a_4)$, where a_3 is a copy of $B(m_b + 1)$ and a_4 a copy of L.

Subcase 2.5. $A(m_a+1) \leq v_b$. Then in parallel, call *Alg-MSP*$(B[i, j], P[m_a+1, l], a_1, a_2)$, where a_1 is a copy of F and a_2 a copy of L, and call *Alg-MSP*$(B[i, m_b - 1], P[k, m_a], a_3, a_4)$, where a_3 is a copy of F and a_4 a copy of $B(m_b - 1)$.

Case 3. $P[k,l]$ has only one processor. There are four subcases. These subcases are handled similarly as those in Case 2 and, in fact, are simpler than those in Case 2. Hence they are omitted here.

The algorithm initially calls *Alg-MSP*$(B[1, m], P[1, n], b_1, b_2)$, where b_1 is a copy of $B(1)$ and b_2 a copy of $B(m)$.

The algorithm proceeds in stages, as shown in [9]. If a call is processed at stage s, then the recursive calls made by this call are to be processed at stage $s + 1$. Since $P[1, n]$ can be bisected $O(\log n)$ times and a binary search in B can take $O(\log m)$ steps, there are totally $O(\log m + \log n) = O(\log m)$ stages in the algorithm. Therefore it is easy to see that the algorithm takes $O(\log m)$ time.

The algorithm correctly solves the MSP. This can be shown by an easy induction on the number of stages of the algorithm. Suppose in stage $s \geq 1$, a call to *Alg-MSP* with input arguments $B[i, j]$ and $P[k, l]$ is being processed. Then one of two possible things is done: either we correctly find the indices of B for $A[k, l]$ (e.g., in Case 1, Subcases 2.1 and 2.2) and make no recursive calls, or we ensure the following condition being held for the recursive calls (which are to be processed in stage $s + 1$) generated by this call: the indices of B for the *P-block* arguments (hence, the *A-blocks*) to those recursive calls can all be determined using the *B-block* arguments to those calls (this can be easily checked for Subcases 2.3, 2.4, and 2.5).

The only thing left is to show that no read conflict occurs in this algorithm. Case 1 is easy, because the reads are caused only by $O(1)$ sequential binary searches on $A[k, l]$ (notice that the *A-blocks* are disjoint). We only show for Case 2, since Case 3 is just a simple case of Case 2.

In a stage, we say $P[k, l]$ is *at* $B[i, j]$ if both $P[k, l]$ and $B[i, j]$ are input arguments to a call made by the algorithm. There are two situations: one is when two *P-blocks* are at two

distinct B-blocks in a stage, and the other is when two *P-blocks* are at the *same B-block* in a stage. We have the following facts.

Lemma 1 *In a stage, let $P[k,l]$ be at $B[i,j]$. Then the only possible read accesses to $B[i,j]$ made by $P[k,l]$ occur only at $B(m_b-1)$, $B(m_b)$, or $B(m_b+1)$, where m_b is the index for the middle element of $B[i,j]$. Furthermore, a read access to one of these three elements is necessary only if the index q for that element in B satisfies $i<q<j$.*

Proof. The subcases in Case 2 are identified using the values of $B(i)$, $B(m_b)$, and $B(j)$. The values of $B(m_b-1)$ and $B(m_b+1)$ are needed for the recursive calls. These five values are all that are needed from $B[i,j]$ by the search of $P[k,l]$ in this stage. The values of $B(i)$ and $B(j)$ are already available from the input arguments F and L. Notice that each *P-block* has its own copies of F and L (even if the copies represent the same elements in B). Therefore no read conflict can occur when reading from F and L. Let $q \in \{m_b-1, m_b, m_b+1\}$. If $q<i$ or $j<q$, then q is out of the index range of $B[i,j]$ and thus no read is needed from $B(q)$ by $P[k,l]$. If, say, $q=i$, then there is no need to read from B for the value of $B(i)$ because the value is available from F. The same is for $q=j$. □

Lemma 2 *In a stage, if $P[k_1,l_1]$ is at $B[i_1,j_1]$ and $P[k_2,l_2]$ at $B[i_2,j_2]$, where $B[i_1,j_1] \neq B[i_2,j_2]$, then $P[k_1,l_1]$ and $P[k_2,l_2]$ do not read from the same element of B.*

Proof. Let the index for the middle element of $B[i_1,j_1]$ be m_{b_1} and the index for the middle element of $B[i_2,j_2]$ be m_{b_2}. Since $B[i_1,j_1]$ is different from $B[i_2,j_2]$, there is at most one index $q \in \{m_{b_1}-1, m_{b_1}, m_{b_1}+1\} \cap \{m_{b_2}-1, m_{b_2}, m_{b_2}+1\}$ such that $B(q)$ is in $B[i_1,j_1] \cap B[i_2,j_2]$. If this is the case, then $B(q)$ must be one of the first or last element of $B[i_1,j_1]$ or $B[i_2,j_2]$, and by Lemma 1 no read from $B(q)$ is necessary. □

The above two lemmas show that if two *P-blocks* are at distinct *B-blocks* in a stage, then no read conflict can occur between the two *P-blocks* from their read accesses to B. The following lemma shows that in the situation when more than one *P-block* is at the same *B-block* in a stage, no read conflict will happen at that *B-block*.

Lemma 3 *In a stage, there can be only $O(1)$ (in fact, at most* three *in our algorithm)* P-blocks *at the same* B-block.

Proof. It has been shown in [9] that there can be at most *four P-blocks* at the same node of a 2-3 tree in a stage during the parallel search. The same proof is for this lemma and hence is omitted here. □

From the discussion for the two situations above, we can conclude that no read conflict occurs in this algorithm.

3.2 Reducing the Processor Complexity

We now present a solution which takes $O(\log m)$ time using $O((n \log(m/n))/\log m)$ EREW-PRAM processors. This algorithm uses less processors than the one in Subsection 3.1 if $O(\log(m/n))$ is $o(\log m)$ (it has the same complexity bounds as the one in Subsection 3.1 if $O(\log(m/n))$ is $\Theta(\log m)$). For example, when $m = O(n \log^c n)$ for some constant $c > 0$, only $O((n \log\log n)/\log n)$ processors are needed here. The total work of this parallel algorithm is $O(n \log(m/n))$, matching the time bound of the sequential algorithm in Section 2.

The algorithm is given below.

(1) Select n elements from B to form an array B'. The elements in B' partition B into n portions of size m/n each.

(2) Partition A into $(n \log(m/n))/\log m$ sub-arrays of size $\log m/\log(m/n)$ each by selecting $(n \log(m/n))/\log m$ elements from A. Let these selected elements form an array A'.

(3) Call *Alg-MSP* to search for A' in B. Let C be the array of the elements of B determined by the call of searching for A' in B. (The elements of C partition B into $O((n \log(m/n))/\log m)$ disjoint portions; no two distinct sub-arrays of A fall into the same portion of B so resulted.)

(4) Merge B' and C into array D. D partitions B into $n + (n \log(m/n))/\log m = O(n)$ sub-arrays of size $\leq m/n$ each (some sub-arrays may have size 0).

(5) Merge A and D into array E (E has a size of $O(n)$).

(6) Find for every element of A which sub-array of B into that it falls (for example, for each element of E that is from A, mark it as 0, and for each element of E from D, mark it by its own index in D; then do a parallel prefix on E).

(7) In parallel, process every sub-array A_i of A using one processor. Each element a of A_i knows exactly which sub-array B_j of B that it falls in. Thus a binary search for a in

B_j will find the answer for a in B. The processor does such a binary search for every element of A_i one by one. (No read conflict can occur here because of Step (3).)

The correctness of this algorithm is very easy to see. This algorithm solves the MSP in $O(\log m)$ time using $O((n \log(m/n))/\log m)$ EREW-PRAM processors. It is trivial to compute Steps (1) and (2) in these complexity bounds. Steps (4) and (5) use parallel merging on totally $O(n)$ numbers and hence can be done in $O(\log n)$ time using $O(n/\log n)$ processors [3, 4]. Step (6) is parallel prefix [5, 7] and can also be computed in $O(\log n)$ time using $O(n/\log n)$ processors. Step (3) uses $O((n \log(m/n))/\log m)$ processors and runs in $O(\log m)$ time (by Subsection 3.1). Step (7) uses $O((n \log(m/n))/\log m)$ processors. Because every processor takes care of $\log m/\log(m/n)$ elements (which form a sub-array) of A, and each element costs $O(\log(m/n))$ time (by a binary search for that element in a sub-array of B whose size is $\leq m/n$), the time for Step (7) is $O(\log(m/n)) \times \log m/\log(m/n)$ $= O(\log m)$. Since $O(\log m) = \Theta(\log(m/n) + \log n)$, the time complexity is $O(\log m)$ and the processor complexity is $((n \log(m/n))/\log m)$.

4 A Parallel Merge Algorithm

For the problem of merging two sorted arrays V and W, each of which has size m, efficient parallel algorithms on the EREW-PRAM have been known [1, 3, 4]. The best solutions run in $O(\log m)$ time using $O(m/\log m)$ processors [3, 4]. Here we give another solution on the EREW-PRAM for parallel merging which also takes $O(\log m)$ time using $O(m/\log m)$ processors. Our algorithm uses *Alg-MSP* as a subroutine.

We briefly outline the algorithm for parallel merging.

(1) Select $m/\log m$ elements from V (resp., W) that partition V (resp., W) into $m/\log m$ portions of size $\log m$ each. Let these selected elements form an array V' (resp., W').

(2) Call *Alg-MSP* to search for V' (resp., W') in W (resp., V). Let the output be in an array W'' (resp., V''). That is, W'' (resp., V'') contains elements of W (resp., V) as the result of searching for V' (resp., W') in W (resp., V).

(3) Now V' and V'' (resp., W' and W'') together partition V (resp., W) into $O(m/\log m)$ sub-arrays of size $\leq \log m$ each (a sorted set of $V' \cup V''$ can be found by doing parallel prefix on V). In this way, each sub-array of V is paired off with one and only one sub-array of W.

(4) In parallel, assign a processor to each pair of sub-arrays of V and W. This processor merges the two sub-arrays sequentially.

It is straightforward to see the correctness of this algorithm and its complexity bounds; hence we omit the analysis here.

Acknowledgement. The author is very grateful to Professor Mikhail Atallah for his great support and valuable suggestions to this work.

References

[1] S. G. Akl and H. Meijer. "Parallel binary search." *IEEE Trans. Parallel and Distributed Systems* Vol. 1 No. 2 (1990), 247–250.

[2] R. P. Brent. "The parallel evaluation of general arithmetic expressions." *J. of the ACM* 21 (2) (1974), 201–206.

[3] G. Bilardi and A. Nicolau. "Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines." *SIAM J. Comput.* 18 (1989), 216–228.

[4] T. Hagerup, and C. Rub. "Optimal merging and sorting on the EREW PRAM." *Inform. Process. Lett.* 33 (1989), 181–185.

[5] C. P. Kruskal, L. Rudolph and M. Snir. "The power of parallel prefix." *IEEE Trans. Comput.* C-34 (1985), 965–968.

[6] S. D. Lang and N. Deo. "Recursive batched binary searching of sequential files." *The Comput. Journal*, to be published.

[7] R. E. Ladner and M. J. Fischer. "Parallel prefix computation." *J. of the ACM* 27 (4) (1980), 831–838.

[8] H. Meijer and S. G. Akl. "Parallel binary search with delayed read conflicts." Tech. Rep. 89-257, Dept. Comput. Inform. Sci., Queen's University, Kingston, Ont., Canada, June 1989.

[9] W. Paul, U. Vishkin and H. Wagener. "Parallel dictionaries on 2-3 trees." *Proc. 10th Coll. on Autom., Lang., and Prog. (ICALP), LNCS 154*, Springer, Berlin, 1983, pp. 597–609.