Finding all nearest smaller values on a distributed memory machine

Jyrki Katajainen
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen East, Denmark
E-mail jyrki@diku.dk

Abstract

Let $A = [a_j]_{j=1}^n$ be a sequence of n elements drawn from some totally ordered set. In the all nearest smaller values problem the task is to find for each $a_j, j \in \{1, 2, \ldots, n\}$, the elements a_i and a_k such that $i = \max\{h \in \{1, \ldots, j\} \mid a_h < a_j\}$ and $k = \min\{h \in \{j, \ldots, n\} \mid a_h < a_j\}$, if such elements exist. In this paper it is shown that this problem can be solved in $O(n/q)$ time and $O(n)$ space with q processors on a distributed memory machine, provided that $q \in \{1, 2, \ldots, O(n/\log_2 n)\}$. More precisely, the machine considered consists of q processors and q memory modules connected by a communication network, which allows every processor to access any memory module in constant time if only every memory module is accessed by at most one processor at a time. The result proved complements the earlier known results showing that, on a CREW PRAM, $\Omega(\log_2 n)$ is a lower bound for the time complexity of the problem with any number of processors and that, on a hypercube, any $O(\log_2 n)$ algorithm must use $\Omega(n)$ processors.

Keywords all nearest smaller values, parallel algorithms, distributed memory machine.

1 Introduction

In the *all nearest smaller values* problem of size n, denoted ANSV(n) for short, a sequence $[a_j]_{j=1}^n$ of n elements from a totally ordered set is given and the task is to determine for each a_j, $j \in \{1, 2, \ldots, n\}$, the maximal index i, $i \in \{1, 2, \ldots, j\}$, and the minimal index k, $k \in \{j, \ldots, n\}$, such that $a_i < a_j$ and $a_k < a_j$. Element a_i is called the *left match* and element a_k the *right match* of a_j. The importance of this problem in parallel computing was realized by Berkman et al. [2]. However, the dual problem of finding all nearest greater values has also been studied by other researchers under such names as all dominating neighbours of a sequence of elements [4, 17] and all tallest neighbours of a set of rooted vertical line-segments [16].

The sequential complexity of the ANSV(n) problem is clearly $\Theta(n)$. For example, the left matches of the elements are found by scanning the input backwards, keeping the elements for which no left match has not yet been found on a stack, and reporting the current element as a left match for those elements on the top of the stack that are larger than the current element. The right matches can be found in a similar manner.

In this paper the parallel complexity of the ANSV-problem is explored. As our model of computation we use a *distributed memory machine* (DMM) that is basically a Parallel Random Access Machine (PRAM), but the size of its *shared memory* is restricted to q cells, where q is the number of processors available. Variants of this computation model have been studied in the literature under various names such as Local Memory PRAM [11], Module Parallel Computer [12], Direct Connection Machine [20], or Seclusive PRAM [24]. For a detailed discussion of these models, we refer to [8]. The variant we use is an EREW DMM, where the accesses of the processors to the shared memory cells have to obey the Exclusive-Read-Exclusive-Write (EREW) rule, that is, no two processors may read from or write into the same memory cell in the same step.

It is not hard to show that an EREW DMM is equally powerful as a special kind of EREW PRAM, in which the shared memory is viewed as a two-dimensional array with q rows, called *memory modules*, and in which each memory

Proceedings of CATS'96 (Computing: The Australasian Theory Symposium), Melbourne, Australia, January 29–January 30 1996.

module may only be accessed by one processor at a time. Since this formulation makes it easier to understand the DMM model, we use it lateron. Let $s := \lceil n/q \rceil$. We assume that the input is initially partitioned into blocks of size s (except possibly the last block that might be smaller) such that the ith block lies in the ith memory module. In the final output, the pointers to the left and right matches should be stored block-wise together with the corresponding input elements.

Let $\mathcal{M}$ be a family {M(1),M(2),...} of parallel machines, where M(q) has q processors. Further, let $\mathcal{P}$ and $\mathcal{Q}$ be two problems to be solved in $\mathcal{M}$. Moreover, let f, g, and h be some functions mapping natural numbers to natural numbers. We use notation $\mathcal{P}(n) \leq_{t(n,q)}^{M(q)} \mathcal{Q}(f(n))$, if it is the case that any input for problem $\mathcal{P}$ (input size n, output size $g(n)$) can be transformed to an input for problem $\mathcal{Q}$ (input size $f(n)$, output size $h(f(n))$) and the output for $\mathcal{Q}$ can again be transformed to the output for $\mathcal{P}$ in overall time $t(n,q)$ on an M(q). Intuitively, this means that probelm $\mathcal{P}$ is easier than problem $\mathcal{Q}$. Especially, if there exists an algorithm $\mathcal{A}$ that solves problem $\mathcal{Q}(m)$ in time $s(m,q)$ on an M(q), then by using the above transformations and algorithm $\mathcal{A}$, any instance of problem $\mathcal{P}(n)$ can be solved in time $t(n,q) + s(f(n),q)$ on an M(q).

Berkman et al. [2] proved the following results

MINIMUM(n) $\leq_{O(n/q)}^{\text{EREW PRAM}(q)}$ ANSV(n), for all $q \in \{1, 2, \ldots, n\}$,

MERGING(n,n) $\leq_{O(n/q)}^{\text{EREW PRAM}(q)}$ ANSV($2n$), for all $q \in \{1, 2, \ldots, n\}$, and

( )-MATCHING $\leq_{O(n/q)}^{\text{EREW PRAM}(q)}$ ANSV(n), for all $q \in \{1, 2, \ldots, O(n/\log_2 n)\}$,

where MINIMUM(n) denotes the problem of computing the *minimum* of n elements, MERGING(n,n) that of *merging* two sorted sequences each of size n, and ( )-MATCHING(n) that of determining the *matching right parentesis* for each *left parenthesis* in a legal sequence of n left and right parentheses. It is not difficult to show that these reductions are also valid on an EREW DMM(q), or even on some weaker models of computation.

On a CREW PRAM(q), the ANSV(n) problem can be solved in $O(n/q)$ time, that is *cost-optimally*, for any $q \in \{1, 2, \ldots, O(n/\log_2 n)\}$ [2]. Since

$$\text{MINIMUM}(n) \leq_{O(1)}^{\text{CREW PRAM}(n)} \text{ANSV}(n),$$

$\Omega(\log_2 n)$ is a lower bound for the time complexity of the problem regardless of the number of processors available in any model weaker than the CREW PRAM model [6]. Independently of the present work, algorithms solving the ANSV(n) problem in $O(n/q)$ time with q processors on an EREW PRAM(q), for any $q \in \{1, 2, \ldots, O(n/\log_2 n)\}$, have been reported [4, 16, 17]. On a WEAK CRCW PRAM(q), the problem is solvable cost-optimally, for any $q \in \{1, 2, \ldots, O(n/\log_2 \log_2 n)\}$ [2]. On the other hand, it is known that $\Omega(n)$ processors are necessary for any algorithm to solve the ANSV-problem in $O(\log_2 n)$ time on the HYPERCUBE [18].

In this paper these results are complemented by proving that the ANSV(n) problem can be solved in $O(n/q)$ time and $O(n)$ space on an EREW DMM(q), for any $q \in \{1, 2, \ldots, O(n/\log_2 n)\}$. Due to the self-simulation property of PRAMs, on an EREW PRAM an algorithm running in time t with q processors translates immediately to an algorithm running in $O(t \cdot q/p)$ time with $p \leq q$ processors. The same needs not be true on an EREW DMM, because this model is not known to be self-simulating. However, the result of this paper shows that for the ANSV-problem this kind of translation is possible with no loss of efficiency.

By using the above mentioned problem transformations, the result of this paper also implies that the problems MERGING(n,n) and ( )-MATCHING(n) can be solved within the same resource bounds as ANSV(n) on an EREW DMM(q). This result for merging was already proved by Katajainen et al. [15], but the result for parentheses-matching is apparently new. For other papers describing cost-optimal logarithmic-time algorithms for merging and parentheses-matching on an EREW PRAM, see [1, 3, 7, 9, 10, 15, 21, 22, 23].

2 Preliminaries

In this section we recall some basic facts on the power of DMMs and the complexity of the ANSV-problem in some special cases.

2.1 On the power of DMMs

Let SORT(n) denote the problem of *sorting* n elements in non-decreasing order according to the key values of the elements. By viewing the EREW DMM(q) as an EREW PRAM(q) with $O(q)$ shared memory cells, one may sort q elements in $O(\log_2 q)$ time by using the pipelined mergesort algorithm devised by Cole [5].

Fact 1 *The problem SORT(n) can be solved in* $O(\log_2 n)$ *time and* $O(n)$ *space on an EREW DMM(n).*

In the problem of *computing all prefix sums*, PREFIX-$\oplus(n)$, a sequence $[a_i]_{i=1}^n$ of n elements and an associative operation $\oplus$ are given and the task is to compute another sequence $[b_i]_{i=1}^n$ such that $b_i := \oplus_{j=1}^i a_j$. The standard algorithm based on the partitioning technique (see, e.g., [19]) runs also on an EREW DMM. Therefore, we have

Fact 2 *The problem PREFIX-*$\oplus(n)$ *can be solved in* $O(n/q + \log_2 n)$ *time and* $O(n)$ *space on an EREW DMM(q), for any associative operation* $\oplus$ *and* $q \in \{1, 2, \ldots, n\}$.

By using the standard simulation techniques (see, e.g., [13, Section 10.1]) and the above facts, we get the following

Fact 3 *A concurrent read instruction of a CREW PRAM(q) can be simulated on an EREW DMM(q) to run in* $O(\log_2 q)$ *time. The space requirement is increased in this simulation by an additive term* $O(q)$.

Yet another problem, the solution of which we need, is MULTIPREFIX-$\oplus(m, n)$ defined as follows. Given m sequences of elements $[a_{ik}]_{i=1}^n$, $k \in \{1, 2, \ldots, m\}$, each containing n elements such that a_{ik} is stored in the ith memory module. The goal is to compute the prefix sums $[b_{ik}]_{i=1}^n$ so that $b_{ik} = \oplus_{j=1}^i a_{jk}$, for all $i \in \{1, 2, \ldots, n\}$ and $k \in \{1, 2, \ldots, m\}$. These m independent prefix computations can be performed efficiently in parallel as stated in the following

Fact 4 *The problem MULTIPREFIX-*$\oplus(m, n)$ *can be solved in* $O(m + \log_2 n)$ *time and* $O(m \cdot n)$ *space on an EREW DMM(n), for any associative operation* $\oplus$.

Proof Let $s := \lceil \log_2 n \rceil$ and $t := \lceil n/(2s) \rceil$. We show here how the s first prefix-sum computations can be carried out in $O(\log_2 n)$ time with the n processors available. From this, the time bound for general m will immediately follow. The space bound is obvious so we will not discuss about it any further.

Let $[a_{ij}]_{i=1}^n$ be the input sequences for the prefix computations, $j \in \{1, 2, \ldots, s\}$. Now we divide each of these sequences into blocks B_{hj} of size $2s$ (except possibly the last block which can be smaller), $h \in \{1, 2, \ldots, t\}$. Observe that the number of these blocks is less than or equal to n, so we can allocate one processor for each block to compute the sum of the elements in B_{hj}. To avoid conflicts when accessing the memory modules, the sums $B_{h1}, B_{h2}, \ldots, B_{hs}$ are computed in a pipelined fashion. The processor assigned to B_{hj} starts its computation at time step j and finally stores its output c_{hj} into the $((j-1) \cdot t + h)$th memory module. All this takes clearly $O(s)$ time.

Next the prefix sums of sequences $[c_{hj}]_{h=1}^t$ are computed, for all $j \in \{1, 2, \ldots, s\}$, in parallel. For the jth task we allocate the t processors whose numbers are in the range $\{(j-1) \cdot t + 1, \ldots, j \cdot t\}$. These independent prefix computations are performed by using the algorithm establishing Fact 2. So also in this phase the memory module conflicts are avoided. By Fact 2 the total time used here is $O(\log_2 t)$.

Finally, we allocate again one processor for each block B_{hj} to compute the prefix sums of the elements inside the block in parallel. Before these computations the first element a_f of B_{hj} is assigned to be equal to $a_f + c_{h-1,j}$ (c_{0j} being equal to 0). Due to these assignments, we can be sure about the correctness of the whole algorithm. The prefix-sum computations within the blocks are pipelined as done when computing the sums. Again these pipelined computations take at most $O(s)$ time.

To summarize, the s first sequences can be handled in $O(s + \log_2 t)$ time, that is in $O(\log_2 n)$ time. □

The problem, COPYING(n), of *creating n copies* of a datum x, is a special prefix computation, where the associative operation is the normal addition and the input $[a_i]_{i=1}^n$ is of the form $a_1 := x$ and $a_i := 0$, for $i \in \{2, \ldots, n\}$. The problem SEGMENTED-PREFIX-$\oplus(n)$, where the input sequence is divided into segments of consecutive elements and the prefix sums within each segment is to be computed, is still another special case of the standard PREFIX-$\oplus$ problem (see, e.g., [13, Exercise 2.6]). Therefore, also the problem SEGMENTED-MULTICOPYING(m, n), a special case of the MULTIPREFIX-$\oplus(m, n)$ problem, in which the prefix operation corresponds to that for segmented copying, is solvable within the resource bounds stated in Fact 4. This result for segmented multicopying was already proved by Katajainen et al. [15, Section 4].

2.2 Special ANSV-problems

Consider the very special case, where we want to compute the left and right matches for one element a_j of the sequence $[a_i]_{i=1}^m$ only. We use x-ANSV(m) to denote such a problem if the size of the sequence is m.

Assume now that we have m processors available for solving this problem. One way to use these to determine the right match of a_j is as follows. The left match of a_j can be found in an analogous manner. First, m copies of a_j are created. Second, each element a_k in the input is compared with a copy of a_j and, if $(j < k)$ and $(a_k < a_j)$ then a 1-bit is stored into the kth entry of a new bit sequence $[b_i]_{i=1}^m$, otherwise a 0-bit is stored into that location. Third, the prefix sums of $[b_i]_{i=1}^m$ are computed. Fouth, the element with the prefix value 1 (i.e., the first 1-bit) is reported as the right match of a_j. By Fact 2 we have

Fact 5 *The x-ANSV(m) problem can be solved in $O(\log_2 m)$ time and $O(m)$ space on an EREW DMM(m).*

Another special case that has to be solved repeatedly in our algorithm is the following. We are given two sequences C and D of m and n elements, respectively. The elements of C are originally in ascending order, whereas the elements of D are in descending order. Now the task is to determine the right matches in D for all C-elements and the left matches in C for all D-elements. We use $(/,\backslash)$-ANSV(m,n) to denote this special case of the ANSV-problem.

Assume now that $m + n$ processors are available for solving the problem. The computation is started be reversing the order of elements in D. Let D^R denote the outcome of this operation. Then the sorted sequences C and D^R are merged, e.g., by using the odd-even merging algorithm (for a linear space implementation, see [14]). In the resulting sequence E, every C-element is labelled with number $n + 1$ and every D-element with number i, where i is its index in D. By performing a prefix-min computation with respect to these labels, the index of a D-element will be broadcasted to those C-elements that lie between it and the next D-element. This way the left matches of all D-elements are found. The right matches of all C-elements can be determined in a similar manner. By Fact 2 and the properties of the odd-even merging algorithm, we have

Fact 6 *The problem $(/,\backslash)$-ANSV(m, n) can be solved in $O(\log_2(m+n))$ time and $O(m+n)$ space on an EREW DMM$(m + n)$.*

3 Algorithm and its analysis

In this section we describe two algorithms for solving the ANSV(n) problem. The first algorithm in Section 3.2 uses n processors to solve the problem, whereas the second one in Section 3.3 carries out the task with only q processors, $q \in \{1, 2, \dots, n\}$. Both algorithms rely on the same partitioning paradigm, so we describe this first in Section 3.1.

3.1 The paradigm

Our algorithms as many other parallel algorithms are based on the partitioning technique. First, the original problem is partitioned into smaller subproblems, these are solved, and finally these partial solutions are combined to get a solution for the original problem. In the two algorithms to be described, the size of the subproblems vary as well as how these are solved and how the partial solutions are combined to obtain the final answer.

Let $[a_i]_{i=1}^n$ be the sequence of n elements, for which the left and right matches are to be found. Without loss of generality, we can assume that n is a power of 2, i.e., $n := 2^K$, for some natural number K. If this was not the case, we could increase the problem size to the nearest power of 2 larger than n and add some dummy elements to the end of the input to fill the empty lots thus created. After the computation the matches of these dummy elements could just be ingnored. Since this operation will at most double the problem size, it does not have any effect on the asymptotical performance of our algorithms.

The value of the blocking factor s is to be determined later. However, we assume that s is a power of 2. Further, let $t := n/s$. Now the ANSV-problem for $[a_i]_{i=1}^n$ is solved as follows.

Step 1: Divide the input into t blocks of size s. Let B_j denote the jth block, $j \in \{1, 2, \dots, t\}$.

Step 2: Solve the ANSV(s) problem for each B_j. This way we can identify the following subsets of B_j

(1) the set L_j containing those elements that do not yet have a left match but whose right match is in B_j,

(2) the set R_j containing those elements that do not yet have any right match but whose left match is in B_j,

(3) the minimum element in B_j that is still without matches, and

(4) the set of all the remaining elements that have their matches within B_j.

Step 3: Following the terminology used in [2], we let $\ell(k)$ denote the index of the left match of a_k, $r(k)$ the index of the right match of a_k, $g\ell(k)$ the block containing $a_{\ell(k)}$, and $gr(k)$ the block containing $a_{r(k)}$. Moreover, we let $b(j)$ denote the index of the minimum element of B_j. Let M be the set of all $a_{b(j)}$'s. Solve now the ANSV(t) problem for M. This gives us the index $g\ell(b(j))$ of the block containing $a_{\ell(b(j))}$ and the index $gr(b(j))$ of the block containing $a_{r(b(j))}$.

Step 4: To avoid read conflicts when accessing blocks $B_{g\ell(b(j))}$ and $B_{gr(b(j))}$, generate two copy requests $\langle j, g\ell(b(j))\rangle$ and $\langle j, gr(b(j))\rangle$, for each $j \in \{1, 2, \ldots, t\}$. Sort these with respect to their second component, i.e., we have to solve a SORT$(2t)$ problem here.

Step 5: Create a proper number of copies of each block. After solving this SEGMENTED-MULTICOPYING$(2t, s)$ problem, use the copies created to determine $\ell(b(j))$ and $r(b(j))$, for all $j \in \{1, 2, \ldots, t\}$. Here we have to solve $2t$ x-ANSV$(s{+}1)$ problems in parallel.

Step 6: By using $\ell(b(j))$'s and $r(b(j))$'s, compute the remaining matches. In this step, we shall rely on the tools developed in [2]. Now there are four cases.

Case 1: $gr(b(j)) = j + 1$. For each $j \in \{1, 2, \ldots, t\}$, it is checked whether this condition is true. For those blocks for which this is the case the $(/,\backslash)$-ANSV problem for $B_{j1} := R_j$ and $B_{j2} := L_{j+1}$ is solved, thus giving the right matches for all R_j-elements and the left matches for all L_{j+1}-elements

Case 2: $g\ell(b(j)) = j - 1$. This case is handled symmetrically with Case 1, so the description is omitted here.

Case 3: $i := g\ell(b(j))$, $k := gr(b(j))$, $g\ell(b(i)) = g\ell(b(j))$. For each $j \in \{1, 2, \ldots, t\}$, it is checked whether this condition is true. Observe that this checking may involve concurrent reading. Therefore, this step should be simulated as described in Fact 3.

Let B_{j1} denote the subsequence containing the R_i-elements between indices $b(i)$ and $\ell(b(j))$. Further, let B_{j2} denote the subsequence containing the L_k-elements between indices $r(b(j))$ and $r(b(i))$. The processors assigned to B_j are used to solve the $(/,\backslash)$-ANSV problem for B_{j1} and B_{j2}, for all j in parallel. Observe that to avoid read conflicts in these computations copies of the whole blocks B_i and B_k are to be provided as in Steps 4 and 5.

Case 4: $i := g\ell(b(j))$, $k := gr(b(j))$, $g\ell(b(j)) = g\ell(b(k))$. This case is again symmetric with Case 3, so its description is omitted.

The proof of correctness of the described algorithm is established in [2, Theorem 3.5], where it was proved that all elements having their matches outside their own block will get the proper matches in one of the cases 1-4.

3.2 Non-optimal algorithm

In this subsection our aim is to solve the ANSV(n) problem on an EREW DMM(n) by using the $\sqrt{n}$-divide-and-conquer technique. Here the description of the algorithm is recursive. However, it is not difficult to devise an iterative implementation of the algorithm that will avoid the complications in syncronization that might be problematic, when running recursive programs in the EREW DMM model.

Recall that $n = 2^K$, for some non-negative integer K. Now we let $t := 2^{\lfloor K/2 \rfloor}$ and $s := 2^{\lceil K/2 \rceil}$. First, we divide the input $[a_i]_{i=1}^n$ into blocks $B_1, B_2, \ldots, B_t$ of size s (Step 1). Second, we solve the ANSV(s) problems for each B_j recursively in parallel, by allocating s processors for each subproblem (Step 2). After this we have identified the sets L_j, R_j, and the minimum element $a_{b(j)}$ in each block B_j. As earlier, let M denote the sequence of all minumum elements $a_{b(j)}$. Third, we create t copies of M (cf. Fact 2) and use these copies to solve the t x-ANSV(t) problems in parallel (cf. Fact 5). Hence, we have determined the blocks in which the actual matches of $a_{b(j)}$ lie, for all $j \in \{1, 2, \ldots, t\}$ (Step 3).

Consider next how the left match is found for each $a_{b(j)}$ (Steps 4 and 5). The right matches can then be determined analogously. Many of the minimum elements might have their matches in the same block. To avoid concurrent reads when accessing the blocks, we use again copying. Since $a_{b(j)}$ needs for a copy of block $B_{g\ell(b(j))}$, we generate a copy request $\langle j, g\ell(b(j))\rangle$. Then we sort these requests with respect to their second component (cf. Fact 2). This way the requests wanting to access the same block are gathered together. The location of a request in the resulting sorted sequence is called its *rank*. Let us now see the shared memory as a two-dimensional array T of size $s \times t$. After the current stage, the hth column of T should contain block B_j if the second component of the request with rank h is j. This is accomplished as follows.

First, the segmentation information, i.e., the sorted sequence of requests, is copied to each row of T. Every element in each row can be copied independently of each others by using the processors "lying" on the columns. Any entry t_{ij} of T is called a *starter* if the segment of $t_{i-1,j}$ differs from that of t_{ij}, i.e., the block in the $(i-1)$st column and ith column should not be the same. Second, an element from the original block is moved to each starter position of T. Third, a segmented copying computation is carried out in each row. Fourth, if the first component corresponding to some column equals to i, this column is used to determine $\ell(b(i))$. Here we have to solve t x-ANSV(b) problems in parallel (cf. Fact 5).

At this point, $\ell(b(j))$ and $r(b(j))$ are known for each $i \in \{1, 2, \ldots, t\}$. To find the missing matches (Step 6), we do as described in Section 3.1. Observe that in Cases 1-4 separate copies of the set B_{j1} and B_{j2} are actually not needed, since all these subsequences are disjoint and since all the elements are stored in separate memory modules.

As to the time complexity of the algorithm described above, it is clear that all subroutines used require at most logarithmic time with linear number of processors. Moreover, since an ANSV(2) problem can be solved in constant time, the running time of the algorithm satisfies the recurrence

$$T(2^K) \leq \begin{cases} T(2^{\lceil K/2 \rceil}) + O(K) & \text{for } K > 1 \\ O(1), & \text{for } K = 1 \end{cases}$$

It is easy to verify by induction that $T(2^K) \in O(K)$.

As to the space complexity of the algorithm, only $O(n)$ space is used at each recursion level. If the algorithm is implemented iteratively in a bottom-up manner, the space used at each level can be reused at the next level. Thus, the total space consumption is also $O(n)$.

Therefore, we have proved the following

Theorem 7 *The ANSV(n) problem can be solved in $O(\log_2 n)$ time and $O(n)$ space on an EREW DMM(n).*

3.3 Cost-optimal algorithm

In this subsection we present and analyse a cost-optimal algorithm for solving the ANSV(n) problem. More precisely, we prove

Theorem 8 *The ANSV(n) problem can be solved in $O(n/q + \log_2 n)$ time and $O(n)$ space on an EREW DMM(q), for any $q \in \{1, 2, \ldots, n\}$.*

Let us assume that we have $q = 2^L$ processors available for solving the problem. The basic approach when utilizing these processors is similar to that used already in the previous subsection. Now the input is partitioned into q blocks, each of size $s := n/q$ (Step 1). The left and right matches within each block are found by using a sequential algorithm, but again the blocks are handled in parallel (Step 2). After this we have identified within each block B_j, the set L_j of elements without a left match, the set R_j of elements without a right match, the minimum element $a_{b(j)}$ without any matches, and the elements with both matches within their own block. The elements knowing their matches are discarded from further consideration. However, any element in L_j and R_j as well as $a_{b(j)}$ should remember their original index in the input sequence.

The hearth of our construction is how the remaining matches are determined. This is again done in three stages. First, $g\ell(b(j))$ and $gr(b(j))$ are found for each $a_{b(j)}$ (Step 3). Second, $\ell(b(j))$ and $r(b(j))$ are found for each $a_{b(j)}$ (Step 4 and 5). Finally, by using $\ell(b(j))$'s and $r(b(j))$'s as described in Section 3.1, the remaining matches are found (Step 6). Let us now take a closer look at all these three steps.

We collect all the minimum elements $a_{b(j)}$ together to form set M of size q. Then we solve the ANSV(q) problem for M by using the algorithm developed in Section 3.2. Since we have q processors available, this computation takes $O(\log_2 q)$ time. Hence, we can assume that indices $g\ell(b(j))$ and $gr(b(j))$ are known, for all $j \in \{1, 2, \ldots, q\}$.

Let us next consider how the actual index $\ell(b(j))$ is determined, for all j. The right indices can then be found in a similar manner. To determine $\ell(b(j))$ within block $B_{g\ell(b(j))}$, the jth processor would need its own copy of the block. If such were provided for it, the memory module conflicts could be avoided.

For each $j \in \{1, 2, \ldots, q\}$, we generate a copy request $\langle j, g\ell(b(j))\rangle$. Again these requests are sorted according to their second component (cf. Fact 1). In this sorted sequence, an element is marked as a starter, if the second component at this particular position is not equal to that at the previous position (the first position being always a starter). If a starter position k contains a request $\langle \cdot, h\rangle$, the kth processor copies the original elements from the hth memory module to the kth memory module. Finally, the segmentation information is duplicated for all elements in each memory module. Now the task we want to carry out is that of segmented copying described in Section 2.1. The algorithm establishing Fact 4 will do this efficiently. When these copies are available, the left match of $a_{b(j)}$ is easily determined sequentially, for all j, in parallel. By Facts 1 and 4, the time requirement of these computations is $O(n/q + \log_2 q)$.

At this point $\ell(b(j))$ and $r(b(j))$ are known, for all $j \in \{1, 2, \ldots, q\}$. Now these pointers can be used in the very similar manner as described in Section 3.1. There are four cases and each case generates an $(/,\backslash)$-ANSV problem, where the total size of the subsequences is at most $2n/q$. To solve these problems, the copying technique used above when determining $\ell(b(j))$ is again needed in order to avoid memory module conflicts. When the subsequences B_{j1} and B_{j2} are stored in the same memory module, one processor can solve the corresponding $(/,\backslash)$-ANSV problem for B_{j1} and B_{j2} sequentially.

At last, the final output should be produced as required in the specification of the problem. Now every memory module contains matches for at most $O(n/q)$ elements forming at most $O(1)$ consecutive sequences of size $O(n/q)$. Therefore, we can give $O(1)$ time steps for each processor so that it can move the matches of the elements with index i mod $s = j$ to their final memory module, for each $j \in \{0, 1, 2, \ldots, s-1\}$. These syncronized steps can clearly be performed in $O(n/q)$ time.

To sum up, the time complexity of the algorithm is $O(n/q + \log_2 q)$, which is $O(n/q)$ if $q \in \{1, 2, \ldots, O(n/\log_2 n)\}$.

Up to now, we have not mentioned anything about the space complexity of this algorithm. However, it should be obvious that each memory module will never contain more than $O(n/q)$ elements or other data items. Therefore, the total space requirement is $O(n)$. This concludes the proof of Theorem 8.

Acknowledgements

The author would like to thank Marianne Kaufmann, Jens Mathiesen, and Jesper Larsson Träff for helpful discussions. This work was partially supported by the Danish Natural Science Research Council under contract No. 9400952 (Project "Computational Algorithmics").

References

[1] R. J. Anderson, E. W. Mayr, and M. K. Warmuth. Parallel approximation algorithms for bin packing. *Information and Computation* **82** (1989) 262–277.

[2] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms* **14** (1993) 344–370.

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

[4] D. Z. Chen. Efficient geometric algorithms on the EREW PRAM. *IEEE Transactions on Parallel and Distributed Systems* **6** (1995) 41–47.

[5] R. Cole. Parallel merge sort. *SIAM Journal on Computing* **17** (1988) 770–785. Correction *SIAM Journal on Computing* **22** (1993) 1349.

[6] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower bounds for parallel random access machines without concurrent writes. *SIAM Journal on Computing* **15** (1986) 87–97.

[7] E. Dekel and I. Ozsvath. Parallel external merging. *Journal of Parallel and Distributed Computing* **6** (1989) 623–635.

[8] M. Dietzfelbinger. Simulations between different models of parallel computers. In *Proceedings of the 9th International Conference on Fundamentals of Computation Theory*, Lecture Notes in Computer Science **710**, Springer-Verlag, Berlin/Heidelberg, Germany (1993) 16–30.

[9] X. Guan and M. A. Langston. Time-space optimal parallel merging and sorting. *IEEE Transactions on Computers* **C-40** (1991) 596–602.

[10] T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters* **33** (1989) 181–185.

[11] Y. Han. An optimal linked list prefix algorithm on a local memory computer. *IEEE Transactions on Computers* **C-40** (1991) 1149–1153.

[12] T. J. Harris. A survey of PRAM simulation techniques. *ACM Computing Surveys* **26** (1994) 187–206.

[13] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA (1992).

[14] J. Katajainen. Constants in parallel sorting. In *Proceedings of the 19th Australasian Computer Science Conference*, to appear.

[15] J. Katajainen, C. Levcopoulos, and O. Petersson. Space-efficient parallel merging. *Informatique théorique et Applications* **27** (1993) 295–310.

[16] S. K. Kim. Optimal parallel algorithms on sorted intervals. Technical Report **90-01-04**, Department of Computer Science and Engineering, University of Washington, Seattle, WA (1990).

[17] D. G. Kirkpatrick and T. M. Przytycka. An optimal parallel minimax tree algorithm. Unpublished manuscript.

[18] D. Kravets and C. G. Plaxton. All nearest smaller values on hypercube with applications. In *DIMACS Workshop on Parallel Algorithms: From Solving Combinatorial Problems to Solving Grand Challenge Problems* (1993) 34

[19] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers* **C-34** (1985) 965–968.

[20] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science* **71** (1990) 95–132.

[21] C. Levcopoulos and O. Petersson. Matching parentheses in parallel. *Discrete Applied Mathematics* **40** (1992) 423–431.

[22] W. Rytter and K. Diks. On optimal parallel computations for sequences of bractets. Unpublished manuscript.

[23] W. W. Tsang, T. W. Lam, and F. Y. L. Chin. An optimal EREW parallel algorithm for parenthesis matching. In *Proceedings of the 1989 International Conference on Parallel Processing*, Volume III (1989) 185–192.

[24] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen (Ed.). *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. North-Holland, Amsterdam, The Netherlands (1990) 943–971.