Roughly Sorting: Sequential and Parallel Approach

TOM ALTMAN* and YOSHIHIDE IGARASHI**

We study sequential and parallel algorithms on roughly sorted sequences. A sequence $\alpha=(a_1, a_2, \ldots, a_n)$ is k-sorted if for all $1\le i, j\le n$, $i<j-k$ implies $a_i\le a_j$. We first show a real-time algorithm for determining if a given sequence is k-sorted and an $O(n)$-time algorithm for finding the smallest k for a given sequence to be k-sorted. Next, we give two sequential algorithms that merge two k-sorted sequences to form a k-sorted sequence and completely sort a k-sorted sequence. Their running times are $O(n)$ and $O(n \log k)$, respectively. Finally, parallel versions of the complete-sorting algorithm are presented. Their parallel running times are $O(f(2k) \log k)$, where $f(t)$ is the computing time of an algorithm used for finding the median among t elements.

1. Introduction

The concept of roughly sorting has appeared in the context of parallel sorting on a mesh-connected processor array. Igarashi and Sado have designed fast parallel sorting algorithms in which roughly sorted subfiles are merged [9, 10]. Fundamental properties of roughly sorted sequences and some sequential algorithms have been studied in [4, 5]. The notions of presortedness and nearly sorted lists [3, 7, 8] are related to the ideas presented in this paper, but are somewhat different from the roughly sorted lists we will study here.

A number of applications require only roughly or nearly sorted sequences [5]. For example, consider a sorted file in which the item values are occasionally updated. In many cases, the new item values may not differ greatly from the old ones. However, by replacing the old items with new ones, the sorted order may be disturbed. Since re-sorting the entire file is costly, it may be more efficient to leave it in a roughly sorted order. We may then use the algorithms described below to obtain a completely sorted file.

In this paper, we present algorithms, that create and manipulate roughly sorted sequences in both sequential and parallel environments. In Section 2, we formalize our notion of rough sortedness and k-sorted sequences. Algorithms that determine if a sequence is k-sorted and the k-sortedness of a sequence are given in Section 3. In Section 4, we present an algorithm that merges two k-sorted sequences into one k-sorted sequence. Finally, in Section 5, we design sequential and parallel algorithms that completely sort k-sorted sequences.

*Department of Computer Science, University of Kentucky, Lexington, Kentucky 40506, U.S.A.
**Department of Computer Science, Gunma University; Kiryu, 376, Japan.

2. k-Sortedness

We begin by formalizing our notion of roughly sorted sequences. Let $\alpha=(a_1, a_2, \ldots, a_n)$ be a sequence of n items and $\sigma=(a_{\sigma_1}, a_{\sigma_2}, \ldots, a_{\sigma_n})$ the corresponding completely sorted sequence of elements.

Definition 1 A sequence α is k-sorted if and only if the following conditon is satisfied: for all i, j, $1\le i\le j\le n$, $i\le j-k$ implies $a_i\le a_j$.

The above definition was introduced by Igarashi and Wood [5]. The radius of α is define to be the smallest k, such that α is k-sorted, and denoted by $ROUGH(\alpha)$. As shown by Estivill-Castro and Wood [4], the radius presortedness measure satisfies the axioms introduced by Mannila [7].

Observation 1 Suppose that a sequence α has no duplicate entries. If α is k-sorted, then for all i, $|i-\sigma_i|\le k$. Hence, if α is k-sorted, for all i, a_i is no more than k places away from its proper position in a completely, or 0-sorted, sequence.

Observation 2[5] A sequence α is k-sorted if and only if every $(2k+2)$ block of α (i.e., a sequence of $(2k+2)$ consecutive elements of α) is k-sorted. This plays a key role in the design of our algorithms.

3. Determination of the Radius

Several interesting problems arise concerning k-sorted sequences. In particular, we might ask if a given sequence is k-sorted. Second, we might wish to compute the radius of a given sequence. We show that both of these questions can be answered efficiently.

Lemma 1 Given α, a sequence of n elements and a positive integer k, we can decide in real time (i.e., in n steps) whether α is k-sorted.

Proof: Imagine a bus with a passenger capacity of $k+1$. Suppose that the bus started with $k+1$ initial passengers and that at each stop, one passenger gets off and another gets on (in a FIFO fashion). The driver always remembers max, the weight of the heaviest passenger that got off the bus so far. If the weight of the incoming passenger is less than max, the driver stops the bus and declares: These people are not k-sorted.

It is possible to implement the above with two pointers (always $k+1$ positions apart) to the sequence α and a variable max in which the value of the largest encountered element, outside the current $(k+1)$ is stored. Going left to right, the procedure will always identify the first occurrence of a violation of k-sortedness of α. □

Below, we present an efficient algorithm for determining $ROUGH(\alpha)$, i.e., the radius of α.

Definition 2 Let $\alpha=(a_1, a_2, \ldots, a_n)$ be *a* sequence of n items. The *LR* characteristic sequence of α is defined to be $(b_1, \ldots, b_n)$, where for each $i(1 \le i \le n) b_i=\max\{a_1, \ldots, a_i\}$. This sequence is denoted by $LR(\alpha)$. The *RL* characteristic sequence of α is defined to be $(c_1, \ldots, c_n)$, where for each i $(1 \le i \le n)$ $c_i=\min\{a_i, \ldots, a_n\}$. This sequence is denoted by $RL(\alpha)$.

Definition 3 Let $\alpha=(a_1, a_2, \ldots, a_n)$ be *a* sequence of n items. Let $LR(\alpha)=(b_1, \ldots, b_n)$ and $RL(\alpha)=(c_1, \ldots, c_n)$. The disorder measure sequence of α is defined to be $(d_1, \ldots, d_n)$, where for each i $(1 \le i \le n)$ $d_i=\max(\{i-j \mid c_i<b_j\} \cup \{0\})$. This sequence is denoted by $DM(\alpha)$.

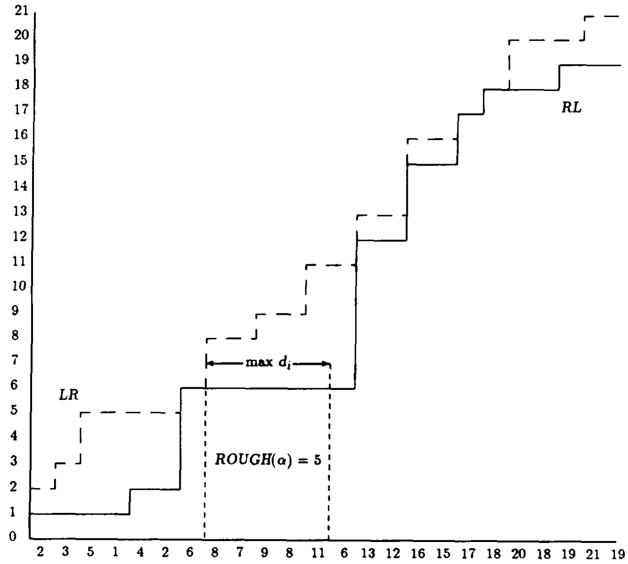
Theorem 1 Let $\alpha=(a_1, a_2, \ldots, a_n)$ be *a* sequence of n items. Then $ROUGH(\alpha)=\max\{d_i \mid d_i$ is an item of $DM(\alpha)\}$.

Proof: Suppose that $ROUGH(\alpha)=k$. If $k=0$, then α is completely sorted and $LR(\alpha)=RL(\alpha)$. Hence, in this case for any $i(1 \le i \le n)$, $d_i=0$, and the assertion of the threorem holds.

Suppose that $k \ge 1$. Then, there exists a pair of i and j such that $i-j=k$ and $a_i<a_j$. Hence, for such i, $d_i \ge i-j=k$. On the other hand, $a_i \ge a_j$ for any pair of i and j such that $i-j \ge k+1$. Therefore, for any $i(1 \le i \le n)$, $d_i<k+1$. □

Below, we present three procedures which construct the *LR*, *RL*, and the *DM* sequences of $\alpha=(a_1, a_2, \ldots, a_n)$.

```
procedure LR(α, B[1 . . n]);
begin
  B[1]:=a_1;
  for i:=2 to n
    if B[i-1]<a_i then B[i]:=a_i
    else B[i]:=B[i-1]
end.
procedure RL(α, C[1 . . n]);
begin
  C[n]:=a_n;
  for i:=n-1 downto 1
    if C[i+1]>a_i then C[i]:=a_i
    else C[i]:=C[i+1]
end.
procedure DM(B[1 . . n], C[1 . . n], D[1 . . n]);
begin
  i:=n;
  for j:=n downto 1 do
    while (j≤i) and (i>0) and (C[i]≤B[j])
        and ((j=1) or (C[i]≥B[j-1])) do
      begin
        D[i]:=i-j;
        i:=i-1;
      end
end.
```



α = (2, 3, 5, 1, 4, 2, 6, 8, 7, 9, 8, 11, 6, 13, 12, 16, 15, 17, 18, 20, 18, 19, 21, 19).
$DM(\alpha)$ = (0, 1, 2, 3, 3, 4, 0, 0, 1, 2, 3, 4, 5, 0, 1, 0, 1, 0, 0, 0, 1, 2, 3, 4).

Fig. 1 The *LR* and *RL* sequences, and the max d_i from *DM*.

Using procedure *LR*, *RL*, and *DM*, we can decide $\max\{d_i \mid d_i$ is an item in $DM(\alpha)\}$ in linear time to n. From Theorem 1, that value is equal to $ROUGH(\alpha)$. An example of a 5-sorted sequence α, its *LR* and *RL* sequences, and the maximal element d_i, from the sequence *DM*, is shown in Fig. 1.

4. Sequential k-Sorting

In this section, we present three algorithms that operate on k-sorted sequences. First, we describe procedure *HALVE*, which takes as input a $2k$-sorted sequence γ and returns a $(k-1)$-sorted sequence δ. Next, in procedure *MERGE*, we show how two k-sorted sequences, α and β, can be merged to produce a k-sorted sequence γ. Finally, procedure *QMSORT* shows how a k-sorted sequence α is sorted in time $O(n \log k)$.

```
procedure HALVE(γ, δ, k);
{Suppose γ=(a_1, a_2, . . . , a_n). Assume n=2kr. If n is
not a multiple of 2k, the procedure needs a minor
modification.}
begin
  1. for i:=1 to r
     begin
```

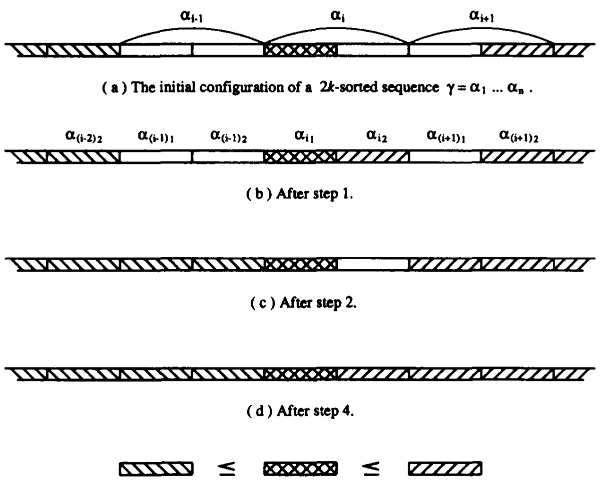


Fig. 2 The order of α_{i_1} during the computation by $HALVE(\gamma, \delta, k)$.

$\alpha_i := a_{2k(i-1)+1} \ldots a_{2ki}$;
$PARTITION(\alpha_i, \alpha_{i_1}, \alpha_{i_2})$
end;
2. **for** $i:=1$ **to** $r-1$
begin
$\beta_i := \alpha_{i_2}\alpha_{(i+1)_1}$;
$PARTITION(\beta_i, \beta_{i_1}, \beta_{i_2})$
end;
3. $\alpha_1 := \alpha_{1_1}\beta_{1_1}$; $\alpha_r := \beta_{(r-1)_2}\alpha_{r_2}$;
4. **for** $i:=2$ **to** $r-1$
$\alpha_i := \beta_{(i-1)_2}\beta_{i_1}$;
5. **for** $i:=1$ **to** r
$PARTITION(\alpha_i, \alpha_{i_1}, \alpha_{i_2})$;
6. $\delta := \alpha_{1_1}\alpha_{1_2} \ldots \alpha_{r_1}\alpha_{r_2}$
end.

The median of n items is defined as that item which is less than or equal to half of the n items and which is larger than or equal to the other half of the n items. Here, $PARTITION(\alpha, \beta, \gamma)$ finds the median m of α and constructs a partition (β, γ) of α by m (i.e., any item in $\beta \leq m \leq$ any item in γ). A computation process by $HALVE(\gamma, \delta, k)$ is depicted in Fig. 2.

Theorem 2 Let γ be *a* $2k$-sorted sequence of length n. Then $HALVE(\gamma, \delta, k)$ returns a $(k-1)$-sorted sequence δ of γ in $O(n)$ time.

Proof: We use the following notation: For x and y, a pair of sequences, $x \leq y$ means that any item in x is not greater than any item in y. Since γ is initially $2k$-sorted, after Step 1, for each i,

$$\alpha_{1_1} \ldots \alpha_{(i-2)_2} \leq \alpha_{i_1} \leq \alpha_{(i+1)_2} \ldots \alpha_{r_2} \text{ and } \alpha_{i_1} \leq \alpha_{i_2}.$$

Hence, after Step 2, for each i,

$$\alpha_{1_1} \ldots \alpha_{(i-1)_2} \leq \alpha_{i_1} \leq \alpha_{(i+1)_1} \ldots \alpha_{r_2}.$$

Then, after Step 3, for each i,

$$\alpha_{1_1} \ldots \alpha_{(i-1)_2} \leq \alpha_{i_1} \leq \alpha_{i_2} \ldots \alpha_{r_2}.$$

Furthermore, we can show

$$\alpha_{1_1} \ldots \alpha_{i_1} \leq \alpha_{i_2} \leq \alpha_{(i+1)_1} \ldots \alpha_{r_2}.$$

Therefore, δ is a $(k-1)$-sorted sequence of γ after Step 6. Since the median of n items can be found in $O(n)$ time (e.g., see [1]), the computing time of $HALVE(\gamma, \delta, k)$ is $O(n)$. □

Procedure *MERGE* below takes as input two k-sorted sequences $\alpha=(a_1, a_2, \ldots, a_n)$ and $\beta=(b_1, b_2, \ldots, b_n)$, and returns the resulting merged k-sorted sequence γ of length $2n$. For simplicity, we assume that k is even and $n=kr$. For n and k not satisfying these conditions, the procedure with a minor modification is still valid.

procedure $MERGE(\alpha, \beta, \gamma, k)$;
{γ is a queue and initially empty}
begin
1. $HALVE(\alpha, \alpha', k/2)$; $HALVE(\beta, \beta', k/2)$;
{$\alpha' = a_1, \ldots, a_n$ and $\beta' = b_1, \ldots, b_n$}
2. **for** $i:=1$ **to** $2r$
begin
$\alpha_i := a_{k(i-1)/2+1} \ldots a_{ki/2}$;
$\beta := b_{k(i-1)/2+1} \ldots b_{ki/2}$;
$amax_i := max(\alpha_i)$; $bmax_i := max(\beta_i)$
end;
3. $p:=q:=1$;
4. **while** $(p \leq 2r$ and $q \leq 2r)$
begin
if $amax_p \leq bmax_q$ **then**
begin
add α_p to γ;
all elements in β_q not greater than $amax_p$ are removed from β_q and added to γ;
$p:=p+1$; **if** β_q is empty **then** $q:=q+1$
end
else
begin
add β_q to γ;
all elements from α_p not greater than $bmax_q$ are removed from α_p and added to γ;
$q:=q+1$; **if** α_p is empty **then** $p:=p+1$
end
end;
5. **if** $p \leq 2r$ **then** $\alpha_p, \ldots, \alpha_{2r}$ are added to γ;
6. **if** $q \leq 2r$ **then** $\beta_q, \ldots, \beta_{2r}$ are added to γ
end.

Theorem 3 Let α and β be two k-sorted sequences of length n. Then $MERGE(\alpha, \beta, \gamma, k)$ returns in $O(n)$ time a k-sorted sequence of length $2n$ which is merged from α and β.

Proof: After Step 2, for any pair of i and j, such that $1 \leq i \leq j \leq 2r$, $\alpha_i \leq \alpha_j$ and $\beta_i \leq \beta_j$ (see Fig. 2). For each t, at the beginning of t-th iteration of **while** statement of Step 4, any element in γ is not greater than any element in $\alpha_p, \ldots, \alpha_{2r}, \beta_q, \ldots, \beta_{2r}$. On the other hand, during the t-th iteration, the number of items transferred from α_p and β_p to γ is at most k. Therefore, the sequence in γ is always k-sorted. Hence, at the end of computation, γ is a k-sorted sequence of length $2n$.

From Theorem 2, the computing time at Step 1 is $O(n)$. Step 2 obviously takes $O(n)$ time as well. For each iteration of the **while** statement, the computing time is $O(k)$. Since $r=O(n/k)$, the computing time at Step 4 is $O(n/k)O(k)=O(n)$. Therefore, the time for $MERGE(\alpha, \beta, \gamma, k)$ is $O(n)$. □

Using procedure *HALVE*, we can design a very simple algorithm that completely sorts a k-sorted sequence in time $O(n \log k)$. It is a variation of the quicksort algorithm in which the partitioning element is chosen to be the median of a given subsequence. For this reason, we call the algorithm *QMSORT*. As shown in [4] and [5], the running time $O(n \log k)$ is optimal within a constant factor. The proof is based on the decision tree argument. Algorithm *RHEAPSORT* [5] also completely sorts a k-sorted sequence in $O(n \log k)$ time. Its constant factor is smaller than the constant factor for *QMSORT*. However, as shown in the next section, *QMSORT* has a very natural and direct implementation for parallel environments, whereas the parallel implementation of *RHEAPSORT* seems to be impractical.

procedure *QMSORT*(α, k);
begin
  for $i:=k/2, k/4, \ldots$ **downto** 1 $HALVE(\alpha, \alpha, i)$
end.

Observe that the procedure *HALVE* reduces a $2k$-sorted into a $(k-1)$-sorted sequence. Hence it is pointless to invoke $HALVE(\alpha, \alpha, 0)$. Moreover, to sort 1-sorted sequences, one may use algorithm *ONESORT* [5], which has been shown to be optimal in the worst case and to be close to the known lower bound in the average case.

The next theorem is an immediate consequence of Theorem 2.

Theorem 4 *QMSORT* sorts a k-sorted sequence in time $O(n \log k)$.

QMSORT may, of course, be used to sort an arbitrary sequence of n elements, which by definition is at least $(n-1)$-sorted, in time $O(n \log n)$.

5. Parallel k-Sorting

In this section, our model of computation is the standard PRAM without concurrent reads or writes. First, let us examine the problem of transforming a $2k$-sorted sequence of n elements into a $(k-1)$-sorted sequence.

The procedure *PHALVE* takes as input a $2k$-sorted sequence γ and returns a $(k-1)$-sorted sequence δ.

procedure *PHALVE*(γ, δ, k);
{Suppose $\gamma=(a_1, a_2, \ldots, a_n)$. Assume $n=2kr$. If n is not a multiple of $2k$, the procedure needs a minor modification.}
begin
1. **for** $i:=1$ **to** r **do in parallel**
   begin
   $\alpha_i:=a_{2k(i-1)+1} \ldots a_{2ki}$;
   $PPARTITION(\alpha_i, \alpha_{i_1}, \alpha_{i_2})$
   end;
2. **for** $i:=1$ **to** $r-1$ **do in parallel**
   begin
   $\beta_i:=\alpha_{i_2}\alpha_{(i+1)_1}$;
   $PPARTITION(\beta_i, \beta_{i_1}, \beta_{i_2})$
   end;
3. $\alpha_1:=\alpha_{1_1}\beta_{1_1}$; $\alpha_r:=\beta_{(r-1)_2}\alpha_{r_2}$;
4. **for** $i:=2$ **to** $r-1$ **do in parallel**
   $\alpha_i:=\beta_{(i-1)_2}\beta_{i_1}$;
5. **for** $i:=1$ **to** r **do in parallel**
   $PPARTITION(\alpha_i, \alpha_{i_1}, \alpha_{i_2})$;
6. $\delta:=\alpha_{1_1}\alpha_{1_2} \ldots \alpha_{r_1}\alpha_{r_2}$
end.

Let $f(t)$ denote the time for finding the median of t elements used in procedure *PPARTITION*.

Lemma 2 The computing time for $PHALVE(\gamma, \delta, k)$ by the *PRAM* is $3f(2k)+O(1)$.

We now present a parallel algorithm that sorts a k-sorted sequence α.

procedure *PQMSORT*(α, k);
begin
  for $i:=k/2, k/4, k/8, \ldots,$ **downto** 1
  $PHALVE(\alpha, \alpha, i)$
end.

Theorem 5 *PQMSORT* sorts a k-sorted sequence of size n in time $O(f(2k) \log k)$, using $O(n)$ processors.

Proof: The proof of correctness follows directly from Theorem 2. The overall running time for *PQMSORT* is $O(f(2k) \log k)$ by Lemma 2. □

As stated in Theorem 5 the computing time of *PQMSORT* depends on the efficiency of the median finding algorithm used. For example, if we choose an $O(\log k)$ median finding algorithm, the time complexity of *PQMSORT* becomes $O(\log^2 k)$.

The next procedure is a variation of *PMQSORT*, but a hybrid of parallel and serial computation for sorting k-sorted sequences.

procedure *HQMSORT*(α, k)
{Suppose $\alpha=(a_1, \ldots, a_n)$. Assume $n=2kr$. If n is not a multiple of $2k$, the procedure needs a minor modification.}
begin
1. **for** $i:=1$ **to** r **do in parallel**
   begin
   $\alpha_i:=a_{2k(i-1)+1} \ldots a_{2ki}$;
   $PARTITION(\alpha_i, \alpha_{i_1}, \alpha_{i_2})$
   end;
2. **for** $i:=1$ **to** $r-1$ **do in parallel**
   begin

$\beta_i := \alpha_{i_2}\alpha_{(i+1)_1}$;
PARTITION$(\beta_i, \beta_{i_1}, \beta_{i_2})$
end;
3. **for** $i:=1$ **to** $r-1$ **do in parallel**
begin
$\alpha_{i_2} := \beta_{i_1}$;
$\alpha_{(i+1)_1} := \beta_{i_2}$
end;
4. **for** $i:=1$ **to** r **do in parallel**
begin
QMSORT(α_{i_1}, k);
QMSORT(α_{i_2}, k)
end;
5. $\alpha := \alpha_{1_1}\alpha_{1_2} \ldots \alpha_{r_1}\alpha_{r_2}$
end.

Theorem 6 *HQMSORT* sorts a k-sorted sequence of size n in time $O(k \log k)$, using (n/k) processors.

Proof: At Step 1 and Step 2 the determination of the medians of each α_i and β_i can be performed by a single processor in $O(k)$ time. Step 4. takes $O(k \log k)$ steps. Therefore, the computing time is $O(k \log k)$. At Step 4 of *HQMSORT* the subsequence in each block of size k is sorted sequentially by *QMSORT*(α_{i_1}, k) and *QMSORT*(α_{i_2}, k). Hence, the number of processors needed is $O(n/k)$. □

6. Concluding Remerks

We have designed a number of algorithms for roughly sorted sequences. These algorithms, with the exception of *PQMSORT* and *HQMSORT*, are optimal to within constant factors. We do not yet know the optimal factors for the time complexities of these problems except for the algorithms given in the proof of Lemma 1. We are interested in accurate evaluations of these factors. It would also be of interest to redesign our algorithms using a simpler parallel model, e.g., the mesh-connected processor array, rather than the PRAM model of computation.

Acknowledgements

The authors wish to thank Mrs. Jaleh Razaie for checking and testing the algorithms in this paper. They also wish to thank an anonymous referee for pointing out errors in an earlier version of this paper.

References

1. AHO, A. V., HOPCROFT, J. E. and ULLMAN, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974.
2. BILARDI, G. and PREPARATA, F., A Minimum Area VLSI Architecture for $O(\log N)$ Time Sorting, *Proc. 16th Annual ACM Symp. on Theory of Computing* (1984), 64–70.
3. COOK, C. R. and KIM, D. J., Best Sorting Algorithms for Nearly Sorted Lists, *CACM*, **23** (1980), 620–624.
4. ESTIVILL-CASTRO, V. and WOOD, D., A New Measure of Presortedness, Technical Report CS-87-58, Department of Computer Science, University of Waterloo, 1987.
5. IGARASHI, Y. and WOOD, D., Roughly Sorting: A Generalization of Sorting, Technical Report CS-87-55, Department of Computer Science, University of Waterloo, 1987.
6. LEIGHTON T., Tight Bounds on the Complexity of Parallel Sorting, *IEEE Trans. Comput.*, C-**34** (1985), 344–354.
7. MANNILA, H., Measures of Presortedness and Optimal Sorting Algorithms, *IEEE Trans. Comput.*, C-**34** (1985), 318–325.
8. MEHLHORN, K., Sorting Presorted Files, *4-th GI Conf. on Theory of Computer Science* (1979), 199–212.
9. SADO, K. and IGARASHI, Y., A Divide-and-Conquer Method of the Parallel Sort, Tech. Report AL84-68, IECEJ, 1985.
10. SADO, K. and IGARASHI, Y., A Fast Parallel Pseudo-Merge Sort Algorithm, Tech. Report AL85-16, IECEJ, 1985.

(Received June 13, 1988; revised January 19, 1989)