

An Optimal Sorting Algorithm for Presorted Sequences

濱村博康 宮尾淳一 若林真一

Hiroyasu Hamamura, Jun'ich Miyao, and Shin'ichi Wakabayashi

広島大学 工学部

Faculty of Engineering, Hiroshima University

Abstract : A sorting algorithm is said to be optimal for presorted sequences if it utilizes the presortedness of the input sequences. In this paper, we propose sequential and parallel sorting algorithms which are optimal with respect to three measures of presortedness, which are *Runs*, *Radius*, and *Rem*. We assume an SM EREW MIMD model for the parallel algorithm.

1 Introduction

It is well known that the time lower bound to sort a sequence with length n is $\Omega(n \log n)$. Several algorithms for sorting n elements achieve the lower bound(see [Knuth 73] for example). In some applications, however, the sequences to be sorted do not randomly consist of elements but are already partially sorted. Most $O(n \log n)$ algorithms do not take the presortedness of the inputs into account. Therefore, the interest in sorting focused on algorithms that exploit the degree of presortedness in the inputs.

A measure of presortedness is an integer function that reflects the difference from the totally sorted sequence. Some measures of presortedness have been proposed until now, which are *Runs* [Knuth 73], *Radius* [Altman 89a] [Castro 89], *Rem* [Mannila 85], and *Inv* [Mehlhorn 79]. Mannila [Mannila 85] gave the formal definition of optimal sorting algorithms for presorted sequences. Until now, sequential and parallel sorting algorithms for

presorted sequences have been proposed [Altman 89a] [Altman 89b] [Castro 89] [Hamamura 91] [Levcopoulos 88] [Levcopoulos 89a] [Levcopoulos 89b] [Melhorn 79].

In this paper, we propose a sequential sorting algorithm and a parallel algorithm, which are optimal with respect to *Runs*, *Radius*, and *Rem*. In Section 3, we describe the sequential algorithm which is based on merge sort. Any conventional merging algorithm takes $\Theta(n)$ time for any input sequences, but the running time of our algorithm varies according to the presortedness of the input. In Section 4, we extend the sequential algorithm to the parallel algorithm. We adopt an MIMD model that has scheduling cost, and we evaluate the upper bound of the number of processors, for which the proposed parallel algorithm is cost optimal.

2 Preliminaries

Let $X = (x_1, x_2, \dots, x_n)$ be a *sequence* of length n from a totally ordered set. For simplicity, we assume that the elements in X are distinct. For two sequences, $X = (x_1, x_2, \dots, x_n)$ and $Y = (y_1, y_2, \dots, y_m)$, their *catenation* XY is the sequence $(x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_m)$. If $Y = (x_{f(1)}, x_{f(2)}, \dots, x_{f(m)})$ and $f : \{1, 2, \dots, m\} \rightarrow \{1, 2, \dots, n\}$ is an injective and monotonically increasing function, then Y is called a *subsequence* of X .

Let $\|X\|$ denote the length of sequence X and $\{X\}$ denote the set, which consists of elements of X . $\text{rank}(e, X)$ denotes the number of elements of X less than e . Furthermore, $|S|$ denotes the cardinality of a set S , and $\log x$ is defined as $\max\{1, \log_2 x\}$.

A measure of presortedness is a function from a sequence X to integer m , that reflects the difference of X from the totally sorted sequence. Let $X = (x_1, x_2, \dots, x_n)$ be the input to be sorted. In this paper, we consider four measures of presortedness shown in Table 2.1. For example in the case $X = (1, 3, 4, 6, 2, 5, 7, 8)$, $\text{Runs}(X) = 2$, $\text{Radius}(X) = 3$, $\text{Rem}(X) = 2$, and $\text{Inv}(X) = 4$.

Mannila [Mannila 85] formalized the concept of an optimal sorting algorithm for sorting presorted sequences. His definition indicates the time lower bound for sorting, evaluated

by not only the length of input sequence but also the presortedness of input sequence. The lower bound of each measure is shown in Table 2.1.

Table 2.1 : Measures of presortedness.

<i>Measure</i>	<i>Definition</i>	<i>Lower bound</i>
<i>Runs</i>	$ \{i \mid 1 \leq i < n \text{ and } x_{i+1} < x_i\} + 1$	$\Omega(n \log m)$
<i>Radius</i>	$\min\{k \mid \forall i, j (1 \leq i \leq n, i < j - k) x_i \leq x_j\}$	$\Omega(n \log m)$
<i>Rem</i>	$n - \max\{t \mid \exists i(1), \dots, i(t) \text{ such that}$ $\leq i(1) < \dots < i(t) \leq n \text{ and } x_{i(1)} \leq \dots \leq x_{i(t)}\}$	$\Omega(n + m \log m)$
<i>Inv</i>	$ \{(i, j) \mid 1 \leq i < j \leq n \text{ and } x_i > x_j\} $	$\Omega(n \log(m/n))$

3 A Sequential Algorithm

3.1 Description of Tree Merge Sort

Mergesort [Knuth 73] is an optimal sorting algorithm in the common sense, i.e., it sorts a sequence with length n in $O(n \log n)$ time. But it takes always $\Theta(n \log n)$ time because mergesort has $\log n$ stages and each stage takes linear time. So it is not optimal for presorted sequences.

We newly propose TMSort, which has $\log n$ stages similarly with general merge sort, but it does not always take linear time at each stage. TMSort uses TMerge as the merging algorithm, which is described bellow.

TMerge does not take linear time exactly to merge the sorted sequences, i.e., the running time of TMerge varies according to the presortedness of the input sequence.

TMerge uses the level linked 2-3 [Brown 80] tree to represent the sorted sequences. The level linked 2-3 tree is a data structure which allows both fast accessing and updating of sorted sequences. Now we give the description of TMerge.

[Algorithm TMerge]

input: two level-linked 2-3 trees T_A and T_B , which represent sorted sequences

$A = (a_1, a_2, \dots, a_k)$ and $B = (b_1, b_2, \dots, b_k)$ respectively

output: a level-linked 2-3 tree, which represents the merged sequence of A and B

Step1: { initialization }

$A_tail \leftarrow k; \quad B_head \leftarrow 1;$

Step2: { find the sequences to be exchanged }

while $a_{A_tail} > b_{B_head}$ **do**

begin

$A_tail \leftarrow A_tail - 1; \quad B_head \leftarrow B_head + 1;$

end

Step3: { insertion }

insert the sequence $\alpha = (a_{A_tail+1}, \dots, a_k)$ to T_B ;

insert the sequence $\beta = (b_1, \dots, b_{B_head-1})$ to T_A ;

Step4: { deletion }

delete the sequence α from T_A ;

delete the sequence β from T_B ;

Step5: { merge two trees }

if the height of $T_A =$ the height of T_B **then begin**

make new internal node N labeled by a_k ;

make vertical links between the root of the trees and N ;

make horizontal links between two trees;

end

else begin

connect the lower tree to the higher;

make horizontal links between two trees;

end

□

After initialization, we select the subsequences α and β in Step2. These subsequences have the property that deletion of these sequences makes AB a sorted sequence. In step3, α and β are inserted to T_B and T_A respectively, and deleted from the original tree in the following step. Finally, the two level-linked trees are merged into one tree by making horizontal links between the same depth nodes.

[Example 3.1] We show an example of TMerge in the case $A = (1, 2, 3, 4, 5, 8, 9, 11)$ and $B = (6, 7, 10, 12, 13, 14, 15, 16)$ in Figure 3.1. In step2(Fig. 3.1 (a)), $\alpha = (9, 11)$ and $\beta = (6, 7)$ are found. In step3, α and β are inserted to T_B and T_A respectively and deleted from the original tree in step4(Fig. 3.1 (b)). Finally, horizontal links between two trees are made.

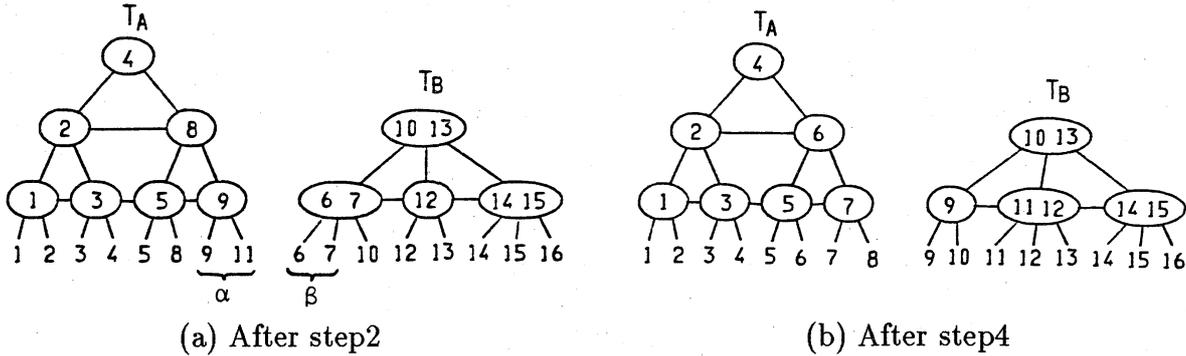


Figure 3.1 : An example of TMerge.

3.2 Analysis of TMSort

We define some notations to analyze the algorithm. Let $tm_{h,j}$ denote the j^{th} merger in the h^{th} stage of TMSort(X), and the input sequences of $tm_{h,j}$ are $A_{h,j}(X)$ and $B_{h,j}(X)$. The execution time of $tm_{h,j}$ is a function of its input sequences A, B and denoted as $t_T(A, B)$. And X^h denotes the output sequence of h^{th} stage mergers. The total execution time of TMSort to sort X is denoted by $T_T(X)$. In the following, let $\Sigma \Sigma$ denote $\sum_{h=1}^{\log n} \sum_{j=1}^{n/2^h}$.

We first analyze the running time of TMerge.

[Lemma 3.1] TMerge merges two sorted sequences $A = (a_1, a_2, \dots, a_k)$ and $B = (b_1, b_2, \dots, b_k)$ in $O(\sum_{i=1}^r \log(\text{rank}(b_{i+1}, A) - \text{rank}(b_i, A)) + \sum_{i=r}^k \log(\text{rank}(a_{i+1}, B) - \text{rank}(a_i, B)) + \log k)$ time where r is an integer satisfying the condition $a_{k-r+1} > b_r$ and $a_{k-r} < b_{r+1}$.

Proof (Refer to [Hamamura 91].) □

We show that TMSort is optimal with respect to *Runs* and *Radius*.

[Lemma 3.2] TMSort sorts the sequence X with $\| X \| = n$ and $Runs(X) = m$ in $O(n \log m)$ time.

Proof Let $R(X)$ be a set of elements defined by the following.

$$R(X) = \{x_j \mid (1 \leq j \leq n) \ x_j > x_{j+1} \text{ or } x_{j-1} > x_j\}$$

If $(\{A_{h,j}(X)\} \cup \{B_{h,j}(X)\}) \cap R(X) = \phi$ then the input sequence $A_{h,j}(X)B_{h,j}(X)$ is already sorted, because if a subsequence of X had no element of $R(X)$ then it would be included by a run of X . So, the running time of the merger is bounded by constant. If $(\{A_{h,j}(X)\} \cup \{B_{h,j}(X)\}) \cap R(X) \neq \phi$, the merger may take $\Theta(2^h)$ time.

Furthermore, $|R(X)| \leq 2m$ holds, then in each stage there are at most $2m$ mergers, each of which takes $\Theta(2^h)$ time and the remaining takes constant. So the total running time of TMSort is bounded as follows.

$$\begin{aligned} T_T(X) &= \sum \sum t_T(A_{h,j}(X), B_{h,j}(X)) \\ &= O(\sum_{h=1}^{\log(n/2m)} \{2m2^h + (n/2^h - 2m) \log 2^h\} + \sum_{h=\log(n/2m)+1}^{\log n} n/2^h (2^h)) \\ &= O(n \log m) \end{aligned} \quad \square$$

For *Radius*, we get the following lemmas. The proofs are described in [Hamamura 91].

[**Lemma 3.3**] TMSort sorts the sequence X with $\|X\| = n$ and $Radius(X) = m$ in $O(n \log m)$ time. □

[**Lemma 3.4**] TMSort sorts the sequence X with $\|X\| = n$ and $Rem(X) = m$ in $O(n + m \log m)$ time. □

The problem whether TMSort is *Inv*-optimal or not is open.

From Lemma 3.2, 3.3, 3.4 and the time lower bounds of each measure, we can show the next theorem.

[**Theorem 3.1**] TMSort is optimal with respect to *Runs*, *Radius*, and *Rem*. □

4 A Parallel Algorithm

4.1 Model of Computation

We assume an MIMD model for parallel computation. Figure 4.1 illustrates the structure of this model. This MIMD model consists of a scheduler, p processors and a shared memory. The scheduler manages the created *processes* using an FIFO queue, and distributes processes to idle processors. The processor executes the assigned processes and becomes

idle state if finished. We put the following assumptions on this model.

- (i) Each processor has the same performance.
- (ii) The concurrent read and concurrent write are not allowed.
- (iii) It takes $d \cdot \log p$ time between the time a processor becomes idle and the time a new process is assigned to this processor (d is a constant).

The assumption of (iii) is reasonable if the scheduling is done using parallel prefix sum computation [Hamamura 91]. The following observation holds with respect to this model.

[Observation 4.1] Suppose that there are k processes in the queue of scheduler and each process needs t time. When p processors execute these processes, the sum of the time in which each processor is in idle state is not greater than $p(t + d \log p) + k \cdot d \cdot \log p$. □

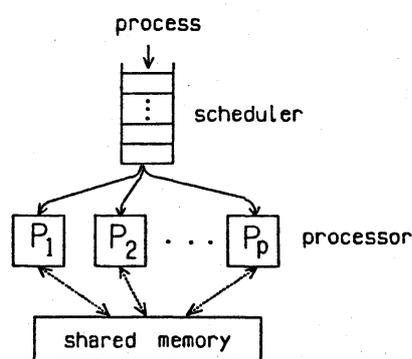


Figure 4.1: Model of computation.

4.2 Parallel Tree Merge Sort

PTMSort is a parallel implementation of TMSort on the MIMD model. The same stage mergers of TMSort can be executed in parallel, so we decompose TMSort to n merging processes and execute these processes concurrently. The algorithm is as follows.

[Algorithm PTMSort]

Process PTMSort(X, n)

begin

for $j := 1$ to $n/2$ **do**

 create process PTMerge($X, 1, j, n$);

end

Process PTMerge(X, h, j, n)

begin

 TMerge(X, h, j);

if $h < \log n$ **then**

if $Wait_{\lfloor (n/2^h + j - 1)/2 \rfloor} = complete$ **then**

 create TMerge($X, h + 1, \lceil j/2 \rceil, n$)

else

$Wait_{\lfloor (n/2^h + j - 1)/2 \rfloor} = complete$;

end

□

At first, PTMSort creates the first stage mergers. Each merging process executes TMerge sequentially, and creates the merging process of the above stage if finished. When creating the above stage process, each process checks an array Wait to examine whether the partner of merging process has been finished or not.

4.3 Analysis of PTMSort

We analyze the running time of PTMSort.

[Lemma 4.1] PTMSort sorts the sequence X in $O(n \log m/p + n)$ time, where p is the number of available processors ($p \leq n$), $n = \|X\|$ and $m = Runs(X)$.

Proof Between the start and the end of sorting, an processor executes merging process or is in idle state. The sum of the execution time of merging processes is $O(n \log m)$ because the sequential algorithm is optimal with respect to $Runs$. So if we can say that the sum of idle time is bounded by $(n \log m)$, the total running time is $O((n \log m)/p)$.

In the h^{th} stage of PTMSort, there are $n/2^h$ merging processes each of which takes $O(2^h)$. Therefore, from observation 4.1 we can say that the sum of the idle time in the h^{th} stage is $O(p(2^h + \log p) + (n/2^h) \log p)$.

If $p \leq \log m$ holds, the total idle time is computed as $\sum_{h=1}^{\log n} O(p(2^h + \log p) + (n/2^h) \log p) = O(n \log m)$.

So, if $p \leq \log m$ holds, the running time is $O(n \log m/p)$. Even if p becomes greater than $\log m$, The running time does not increase than linear time. Therefore, we get $T_T(X) = O(n \log m/p + n)$. \square

By the same way, we get the following lemmas for the other measures. The proofs are described in [Hamamura 91].

[Lemma 4.2] PTMSort sorts the sequence X in $O(n \log m/p + n)$ time, where p is the number of available processors ($p \leq n$), $n = \|X\|$ and $m = Radius(X)$. \square

[Lemma 4.3] PTMSort sorts a sequence with $\|X\| = n$ and $Rem(X) = m$ in $O((n + m \log m)/p)$ using p processors where $p \leq 1 + m \log m/n$. \square

From Lemma 4.1, 4.2, 4.3, we can show the next theorem.

[Theorem 4.1] PTMSort is cost optimal with respect to measures, shown below if the number of processors satisfies the following inequations.

$$\text{Runs} : \quad 1 \leq p \leq \log \text{Runs}(X)$$

$$\text{Radius} : \quad 1 \leq p \leq \log \text{Radius}(X)$$

$$\text{Rem} : \quad 1 \leq p \leq 1 + \text{Rem}(X) \log \text{Rem}(X)/n \quad \square$$

Until now, some parallel sorting algorithms for presorted sequences have been proposed [Altman 89a] [Altman 89b] [Levcopoulos 88] [Levcopoulos 89a]. These algorithms run on PRAM, and TMSort run on the MIMD model, so we can not compare the proposed algorithm with conventional algorithms directly. But, there exists no PRAM sorting algorithm which is shown to be optimal with respect to *Runs*, *Radius*, and *Rem*.

5 Conclusion

In this paper, we proposed sequential and parallel sorting algorithms which are optimal with respect to *Runs*, *Radius*, and *Rem*. We are now interested in the optimality of TMSort with respect to *Inv*.

Acknowledgement

The authors would like to thank Prof. Noriyoshi Yoshida for his kind support and encouragement.

References

- [Altman 89a] T. Altman, and Y. Igarashi: "Roughly sorting : Sequential and parallel approach," Journal of Information Processing, Vol. 12, pp.154-158 (1989).
- [Altman 89b] T. Altman: "Sorting roughly sorted sequences in parallel," Information Processing Letter, Vol. 33, pp.297-300 (1989).
- [Brown 80] M. R. Brown and R. E. Tarjan: "Design and analysis of a data structure for representing sorted lists," SIAM Journal of Comput., Vol.9, No.3, pp.594-614 (1980).

- [Castro 89] V. E. Castro *and* D. Wood: "A new measure of presortedness," Information and Computation, 83, pp.111-119 (1989).
- [Hamamura 91] H. Hamamura: "Optimal sorting algorithms for presorted sequences on an MIMD model," Thesis for the degree of Master of Engineering, Hiroshima University (1991).
- [Knuth 73] D. E. Knuth: "The Art of Computer Programming, Vol.3: Sorting and Searching," Addison-Wesley (1973).
- [Levcopoulos 88] C. Levcopoulos *and* O. Peterson: "An optimal parallel sorting algorithm for presorted files," Proc. 8th Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science, 338, pp.154-160, Springer-Verlag (1988).
- [Levcopoulos 89a] C. Levcopoulos *and* O. Peterson: "Heapsort-Adapted for presorted files," 1989 Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science, 356, Springer-Verlag (1989).
- [Levcopoulos 89b] C. Levcopoulos *and* O. Peterson: "A note on adaptive parallel sorting," Information Processing Letters, Vol. 33, pp.187-191 (1989).
- [Mannila 85] H. Mannila: "Measure of presortedness and optimal sorting algorithms," IEEE Trans. on Computer, Vol.C-34, No.4, pp.318-325 (1985).
- [Mehlhorn 79] K. Mehlhorn: "Sorting presorted files," Proc. 4th GI Conference on Theory of Computer Science, pp.199-212 (1979).