A Cost Optimal Parallel Algorithm for Patience Sorting *

TAKAAKI NAKASHIMA
Tottori University of Environmental Studies,
Wakabadai-Kita 1-1-1, Tottori, Tottori, 689-1111, Japan

and

AKIHIRO FUJIWARA
Department of Computer Science and Electronics
Kyushu Institute of Technology,
Kawazu 680-4, Iizuka, Fukuoka, 820-8502, Japan

Received January 2004
Revised January 2006
Communicated by Selim G. Akl

ABSTRACT

In this paper, we consider a parallel algorithm for the patience sorting. The problem is not known to be in the class NC or P-complete. We propose two algorithms for the patience sorting of n distinct integers. The first algorithm runs in $O(m(\frac{n}{p} + \log n))$ time using p processors on the EREW PRAM, where m is the number of decreasing subsequences in a solution of the patience sorting. The second algorithm runs in $O(\log n + \frac{n \log n}{p} + m^2 \log \frac{n}{p} + m \log p)$ time using p processors on the EREW PRAM. If $1 < p < \frac{n}{m^2}$ is satisfied, the second algorithm becomes cost optimal.
Keywords: Patience sorting, parallel algorithms, P-completeness.

1. Introduction

In parallel computational theory, one of major goals is to find a parallel algorithm which runs as fast as possible. For example, many problems are known to have efficient parallel algorithms which run in $\Theta(1)$ or $\Theta(\log n)$ computational time, where n is the input size of problems. From the point of view of the complexity theory, the class NC is used to denote the measure. A problem is in the class NC if there exists a parallel algorithm which solves the problem in $O(T(n))$ time using $O(P(n))$ processors, where $T(n)$ and $P(n)$ are polylogarithmic and polynomial functions for n, respectively. Many problems in the class P, which is the class of problems solvable in polynomial time sequentially, are also in the class NC. On the other hand, a number of problems in the class P seem to have no parallel algorithm

*This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grand-in-Air for Encouragement Scientists, 14780229, 2003.

which runs in polylogarithmic time using a polynomial number of processors. Such problems are called P-complete. A problem is P-complete if the problem is in the class P and we can reduce any problem in the class P to the problem using NC-reduction. (For details of the P-completeness, see [11].) It is believed that problems in the class NC admit parallelization readily, and conversely, P-complete problems are inherently sequential and difficult to be parallelized.

However, a number of efficient parallel algorithms have been recently proposed for P-complete problems [4,15,16]. In the above papers, the other well-known measure, cost optimality, is used to denote parallelizability of problems. The cost of a parallel algorithm is defined as the product of the running time and the number of processors required in the algorithm, and a parallel algorithm is called cost optimal if its cost is asymptotically equal to the time complexity of the fastest known sequential algorithm for the same problem. The above results mean that a number of inherently sequential problems have cost optimal parallel algorithms, and we can parallelize the problems practically.

In this paper, we consider parallel algorithms for the patience sorting, which was invented as a practical method of sorting a deck of cards [14]. Although the patience sorting is a primitive combinatorial optimization problem, it is not known to be in the class NC or P-complete, that is, no NC algorithm has been proposed for the problem, and there is no proof which shows the problem is P-complete.

The patience sorting has a number of important applications in computer science. For example, the patience sorting can be reduced to the longest increasing subsequence [2,3,5,10], which is a well-known problem in mathematics. Since the reduction is simple, we can solve the longest increasing subsequences from the result of the patience sorting with the same complexity.

Therefore, there are a number of papers which deal with the problem. Sequential algorithms [2,3] show that we can solve the problem in $\Theta(n \log n)$ time sequentially in case that its input is a set of distinct integers. As for parallel algorithms, two algorithms have been proposed for the problem [5,10]. The former algorithm is for the linear array, which is a classical parallel computation model, and runs in $O(n)$ time using n processors. The latter algorithm is for the CGM (Coarse Grained Multicomputer) model [8], which is a practical parallel computation model, and runs in $O(n + \frac{n^2}{p})$ computation time using p processors. The cost of the both algorithms is $O(n^2)$, and the algorithms are not cost optimal.

In this paper, we propose efficient parallel algorithms for the problem and consider their parallelizability. We first propose a simple algorithm for the patience sorting. The algorithms consists of repetition of the prefix operations, which is a well-known basic operation in parallel algorithms. The algorithm runs in $O(m(\frac{n}{p}+\log n))$ time using p processors on the EREW PRAM, where m is the number of decreasing subsequences in a solution of the patience sorting. The complexity indicates that the algorithms is cost optimal in case that $m = O(\log n)$ and $p = O(\frac{n}{\log n})$. Next, we propose another parallel algorithm, which is a little more complicated, for the patience sorting. The second algorithm runs in $O(\log n + \frac{n \log n}{p} + m^2 \log \frac{n}{p} + m \log p)$ time using p processors on the EREW PRAM. From the complexity, the algorithm is cost optimal in case of $1 < p < \frac{n}{m^2}$.

The paper is organized as follows. In Section 2, we make some definitions for

the patience sorting. In Section 3, a parallel algorithm using the prefix operations is described. In Section 4, we explain a cost optimal parallel algorithm for the patience sorting. Finally, we summarize the paper in Section 6.

2. Preliminaries

The patience sorting is known as a traditional card game in Britain. An overview of the game is as follows. (To simplify the description, we assume cards in a deck are indexed $1, 2, \ldots, n$.)

(1) Shuffle the deck.

(2) Turn up one card and deal into piles on the table, according to the following rule: A card with a smaller index may be placed on a card with a larger index, or may be put into a new pile to the right of the existing piles.

At each stage in the second step, we check the top card on each pile. If the turned up card has a larger index than all of the top cards, it must be put into a new pile to the right of the others. The objective of the game is to finish with as few piles as possible.

As a matter of fact, we can achieve the objective using the following greedy method in the second step. (Optimality of the greedy method has been proved in [2].)

(2') Turn up one card and deal into piles on the table, according to the following rule: A card is placed on the leftmost possible pile, whose top card has a larger index than the turned up card. Otherwise, the card is put into a new pile to the right of existing piles.

Our main goal for the patience sorting is to obtain the above optimal solution for the problem.

Now we describe a precise definition of the patience sorting. We make some related definitions before describing the problem.

Definition 1 (Subsequence) *Given a sequence S of n distinct integers, a subsequence of S is a sequence which can be obtained from S by deleting zero or some integers. The subsequence is called increasing if each element of the subsequence is larger than the previous element. Conversely, the subsequence is called decreasing if each element of the subsequence is less than the previous element.* □

Definition 2 (Cover) *Given a sequence S of n distinct integers, a cover of S is a set of subsequences of S such that every element in S is contained in one of the subsequences. The size of the cover is the number of subsequences in it. The cover is called increasing or decreasing if every subsequence in the cover is increasing or decreasing, respectively.* □

Using the above two definitions, the patience sorting is defined as follows.

Definition 3 (Patience sorting) *Let S be a sequence of n distinct integers. The patience sorting is a problem to compute a decreasing cover of S such that the size of the cover is the smallest among all covers of S.* □

It is worth while noticing that each element is not contained in two subsequences of the same cover, and each decreasing subsequence of the patience sorting means a pile in case of the card game. In addition, there may be several solutions for an input of the patience sorting. In this paper, our objective for the problem is to find one of the solutions. Figure 1 shows an example of the patience sorting. In the figure, each vertical sequence denotes a decreasing subsequence of the cover.

Input sequence = (10, 8, 23, 1, 3, 37, 7, 21, 35, 13, 2, 33, 39, 4, 20, 9)

	1	2	4	9	20	
	8	3	7	13	33	
Patience sorting =	10	23	37	21	35	39

Figure 1: An example of the patience sorting.

We can solve the patience sorting using the following greedy algorithm [2]. (Correctness of the algorithm is also proved in [2].)

Algorithm 1 (Greedy algorithm for the patience sorting)
Input : a sequence of n distinct integers $S = (s_0, s_1, \ldots, s_{n-1})$.
Output : a decreasing cover of S. (We assume that $D_0 \cup D_1 \cup \ldots \cup D_{m-1}$ denotes the decreasing cover of S, and each D_i $(0 \leq i \leq m-1)$ denotes the i-th decreasing subsequence of the cover.)

Step 1: Set $j = 1$, $i = 0$, and add s_0 to D_0.

Step 2: Repeat the following substeps until $j = n$.

(2.1): Find the smallest indexed decreasing subsequence whose last element is larger than s_j, and add s_j to the subsequence. If there is no such subsequence, set $i = i+1$, create a new subsequence D_i, and add s_j to the subsequence D_i.

(2.2): Set $j = j + 1$.

We now consider the time complexity of the above greedy algorithm. It is obvious that the number of repetition in Step 2 is $n - 1$. There are two methods of finding the lowest indexed decreasing subsequence in substep (2.1). One of the methods is to examine all decreasing subsequences in order. However, the method takes $O(n)$ time in the worst case and time complexity of the algorithm becomes $O(n^2)$. The alternative method uses the characteristic of the last elements of subsequences, that is, a feature that the last elements are ordered in increasing order. We can execute the binary search [7] for the last elements in $O(\log n)$ time if we use any data structure which can be accessed to the last element of each subsequences in $O(1)$ time. In this case, we can execute the greedy algorithm in $O(n \log n)$ time. Since it is shown that the lower bound of the patience sorting is $\Omega(n \log n)$ [9], we obtain the following lemma.

Lemma 1 *We can solve the patience sorting in $\Theta(n \log n)$ time sequentially.* □

2.1. 2-3 tree

In the following sections, we use a balanced search tree, called a 2-3 tree, to support our parallel algorithm for the patience sorting. We introduce a definition and a lemma for the 2-3 tree.

Definition 4 (2-3 tree) *A 2-3 tree is a rooted tree in which each internal node has two or three children and every path from a root to a leaf is of same length.* □

We can easily prove that the height of a 2-3 tree is $\Theta(\log n)$ in case that the number of leaves is n. When using a 2-3 tree as a data structure, all elements of a sorted sequence are stored into leaf nodes from left to right, and each internal node v holds two variables $L[v]$ and $M[v]$, which store values of the maximum elements in the leftmost and the second subtrees of v, respectively. Using $L[v]$ and $M[v]$, we can search any element in a 2-3 tree in $O(\log n)$ time using a similar technique to the binary search. We can construct a 2-3 tree which stores a sorted sequence, whose size is n, in $O(n \log n)$ time sequentially. (See [1] for details.) Figure 2 shows an example of a 2-3 tree.

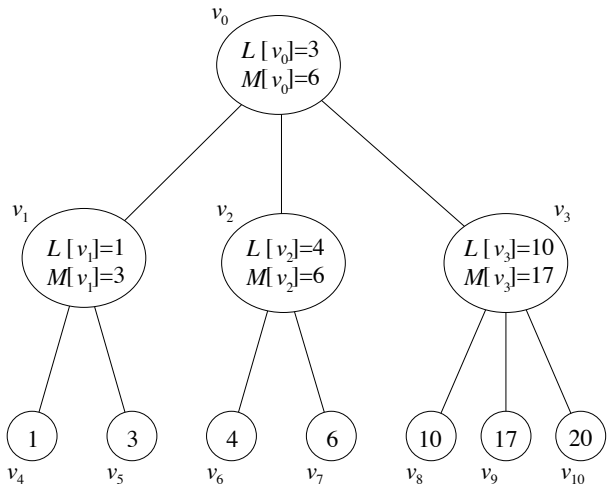

Figure 2: An example of a 2-3 tree for a sequence (1, 3, 4, 6, 10, 17, 20)

Let T, T_1 and T_2 be 2-3 trees which store sorted sequences S, S_1 and S_2, respectively. We use the following four operations on 2-3 trees in this paper.

(1) MIN: $MIN(T)$ is an operation that outputs the minimum element in a 2-3 tree T.

(2) DELETE: Let x be an element in S. $DELETE(T, x)$ is an operation that deletes x from a 2-3 tree T if x exists in the tree.

(3) IMPLANT: Assume each element in S_1 is less than every element in S_2. $IMPLANT(T_1, T_2)$ is an operation that implants T_2 in T_1 so that T_1 stores the concatenated sequence S_1S_2.

(4) SPLIT: Let x be an element in S. $SPLIT(T, x)$ is an operation that outputs two trees T_1 and T_2 which satisfy $S_1 = \{y \mid y \leq x, y \in S\}$ and $S_2 = \{z \mid z > x, z \in S\}$, respectively.

It is known that the above four operations can be processed efficiently on 2-3 trees [1].

Lemma 2 *Let T, T_1 and T_2 be 2-3 trees whose sizes are $O(n)$. We can execute each of four operations MIN, DELETE, IMPLANT and SPLIT in $O(\log n)$ time sequentially.* □

3. First algorithm using prefix operations

In this section, we describe our first algorithm, which consists of repetition of *prefix minima* and *prefix sum* operations, for the patience sorting. The prefix minima of a sequence $(x_0, x_1, \ldots, x_{n-1})$ is defined as the sequence $(pm_0, pm_1, \ldots, pm_{n-1})$ such that $pm_k = \min\{x_h \mid 0 \leq h \leq k\}$, and the prefix sum of a sequence $(x_0, x_1, \ldots, x_{n-1})$ is defined as the sequence $(ps_0, ps_1, \ldots, ps_{n-1})$ such that $ps_k = \sum_{h=0}^{k} x_h$.

The algorithm uses the prefix minima operation as follows. Let $S = (s_0, s_1, \ldots, s_{n-1})$ be an input sequence for the patience sorting. We first compute the prefix minima of S, select elements whose input values are equal to the results of the prefix minima, and store the selected elements in an array D. In case of the sequential greedy algorithm (Algorithm 1), an element s_k is added to the first decreasing subsequence D_0 if s_k is smaller than the last element of D_0. Therefore each element s_k in D_0 satisfies $s_k = \min\{s_h \mid 0 \leq h \leq k\}$, and D is equal to D_0. We repeat the prefix minima operation for the remaining elements, and the other decreasing subsequences are obtained in the same way.

The followings are details of the algorithm.

Algorithm 2 (Algorithm using prefix operations)
Input: a sequence of n distinct integers $S = (s_0, s_1, \ldots, s_{n-1})$.
Output: a decreasing cover of S. (We assume that $D_0 \cup D_1 \cup \ldots \cup D_{m-1}$ denotes the decreasing cover of S, and each D_i $(0 \leq i \leq m-1)$ denotes the i-th decreasing subsequence of the cover.

Step 1: Set $i = 0$.

Step 2: Repeat the following substeps until $s_0 = s_1 = \ldots = s_{n-1} = \infty$.

(2.1): Compute the prefix minima of S, and store the result in an array $Q = (q_0, q_1, \ldots, q_{n-1})$.

(2.2): For each j $(0 \leq j \leq n-1)$, if $s_j = q_j \neq \infty$ set $r_j = 1$, otherwise set $r_j = 0$. Then, compute the prefix minima of $R = (r_0, r_1, \ldots, r_{n-1})$, and store the result in the same array R.

(2.3): For each j $(0 \leq j \leq n-1)$, if $s_j = q_j \neq \infty$, set $d_{i,r_j} = s_j$, and then set $s_j = \infty$.

(2.4): Set $i = i + 1$. □

Now we discuss the complexity of the above algorithm. Let m be the number of decreasing subsequences of the cover. Obviously, all of substeps in Step 2 consist of a constant number of primitive operations and the prefix operations. Using a known parallel algorithm for parallel prefix [13], we can compute the the prefix operation of n elements in $O(\frac{n}{p} + \log n)$ time using p processors on the EREW PRAM. Since the number of repetition of Step 2 is m, we obtain the following theorem.

Theorem 1 *Algorithm 2 solves the patience sorting of n elements in $O(m(\frac{n}{p} + \log n))$ time using p processors on the EREW PRAM.* □

In respect to time complexity, Algorithm 2 is usually not efficient because the optimal sequential time complexity of the problem is $O(n \log n)$. However, the algorithm becomes cost optimal when $m = O(\log n)$ and $p = O(\frac{n}{\log n})$.

4. Second algorithm for the patience sorting

4.1. Outline of the algorithm

In this section, we describe the second parallel algorithm for the patience sorting on the EREW PRAM. We assume that $D_0 \cup D_1 \cup \ldots \cup D_{m-1}$ denotes the decreasing cover of S, and each D_i $(0 \leq i \leq m-1)$ denotes the i-th decreasing subsequence of the cover. We also assume that P_j $(0 \leq j \leq p-1)$ denotes the j-th processor on the PRAM and $1 \leq p \leq n$. The algorithm basically consists of m repetitions of a procedure. In the i-th procedure, we compute the i-th decreasing subsequence D_i.

An outline of the algorithm is as follows. Let S be an input sequence. First, we divide S into p blocks whose sizes are $\frac{n}{p}$, and assign the j-th block to the j-th processor. Then, on each processor, we compute the patience sorting sequentially for each block. We assume that $D_{j,0} \cup D_{j,1} \cup, \ldots, \cup D_{j,m_j-1}$ denotes a result of the patience sorting for a block assigned to a processor P_j.

Next, we compute the first decreasing subsequence D_0 using the above results. We can prove that D_0 is a subset of $D_{0,0} \cup D_{1,0} \cup \ldots \cup D_{p-1,0}$, that is, a subset of the first decreasing subsequences of the divided blocks. We can compute D_0 from $D_{0,0} \cup D_{1,0} \cup \ldots \cup D_{p-1,0}$ using the prefix minima operation. (Correctness and details of this substep are shown in the following subsection.) After computing D_0, we remove elements in D_0 from each block, and reconstruct a decreasing cover for each block. Then, we can compute remaining decreasing subsequences $D_1, D_2, \ldots, D_{m-1}$ by repeating the above procedure $m-1$ times. However, a simple implementation of this step makes time complexity of the algorithm $O(m(\frac{n}{p} \log \frac{n}{p}))$ since reconstruction of a decreasing cover of each block needs $O(\frac{n}{p} \log \frac{n}{p})$ computation time. To reduce the complexity, we use 2-3 trees as a data structure which store a decreasing cover of each block. We assume that each decreasing subsequence $D_{j,k}$, which is the k-th decreasing subsequence for processor P_j, is stored into a 2-3 tree $T_{j,k}$. Since we reconstruct a decreasing cover on each processor efficiently using 2-3

trees, we can reduce the complexity of the algorithm sufficiently. (The details of the reconstruction are also described in the following subsection.)

We now summarize an outline of the algorithm.

Algorithm 3 (Second algorithm for the patience sorting)
Input: a sequence of n distinct integers $S = (s_0, s_1, \ldots, s_{n-1})$. (For simplicity, we assume that $n = kp$ where k is an integer.)
Output: a decreasing cover of S. (We assume that $D_0 \cup D_1 \cup \ldots \cup D_{m-1}$ denotes the decreasing cover of S, and each D_i $(0 \leq i \leq m-1)$ denotes the i-th decreasing subsequence of the cover.)

Step 1: Divide S into p blocks S_j $(0 \leq j \leq p-1)$ of size $\frac{n}{p}$.

Step 2: On each processor P_j $(0 \leq j \leq p-1)$, compute a decreasing cover of S_j sequentially. (We assume that $D_{j,0} \cup D_{j,1} \cup \ldots \cup D_{j,m_j-1}$ denotes the decreasing cover for S_j.) Then, store each decreasing subsequence $D_{j,k}$ in a 2-3 tree $T_{j,k}$ $(0 \leq k \leq m_j - 1)$.

Step 3: Set $i = 0$, and repeat the following substeps until $S_0 = S_1 = \ldots = S_{p-1} = \phi$.

(3.1): Compute the i-th decreasing subsequence D_i from a set of decreasing subsequences $D_{0,0} \cup D_{1,0} \cup \ldots \cup D_{p-1,0}$. On each processor P_j $(0 \leq j \leq p-1)$, elements in $D_j^i = D_i \cap D_{j,0}$ are stored in a new 2-3 tree T_j^i, and the other elements in $D_{j,0} - D_i$ are stored in $T_{j,0}$ again. (A set of elements $D_0^i \cup D_1^i \cup \ldots \cup D_{p-1}^i$ is equal to D_i.)

(3.2): On each processor P_j $(0 \leq j \leq p-1)$, set $S_j = S_j - D_j^i$, and reconstruct 2-3 trees $T_{j,0}, T_{j,1}, \ldots, T_{j,m_j-1}$ so that the set of 2-3 trees denotes a decreasing cover of S_j.

(3.3): Set $i = i + 1$.

Step 4: Execute the following substeps to obtain D_i $(0 \leq i \leq m-1)$ from $D_0^i, D_1^i, \ldots, D_{p-1}^i$.

(4.1): On each processor P_j $(0 \leq j \leq p-1)$, extract all leaf elements of T_j^i $(0 \leq i \leq m_j - 1)$ and store the elements into an array C_j with a key index i.

(4.2): Sort elements $C_0 \cup C_1 \cup \ldots \cup C_{p-1}$ with the key indices and their values, and store the elements with the key index i into D_i. □

We now consider the complexity of the above algorithm on the EREW PRAM. Step 1 can be easily executed in $O(\frac{n}{p})$ time using p processors. In Step 2, we can compute the decreasing cover on each processor in $O(\frac{n}{p} \log \frac{n}{p})$ using sequential algorithm [12] since the number of elements of each block is $O(\frac{n}{p})$, and store the results into 2-3 trees with the same complexity using a sequential algorithm for construction of a 2-3 tree [1]. In Step 4, the substep (4.1) can be executed in $O(\frac{n}{p} \log \frac{n}{p})$ time using MIN and $DELETE$ operations for a 2-3 tree $\frac{n}{p}$ times on

each processor, and the substep (4.2) can be executed in $O(\log n + \frac{n \log n}{p})$ using a well-known sorting algorithm [6]. Let $T_3(n)$ be the time complexity of substeps (3.1) and (3.2). Since the number of repetition of Step 3 is m, where m is the number of decreasing subsequences of the cover, complexity of the algorithm becomes $O(\log n + \frac{n \log n}{p} + mT_3(n))$. In the following two subsections, we consider complexities of substeps (3.1) and (3.2), respectively.

4.2. Computation of the i-th decreasing subsequence

In this subsection, we explain details of the substep (3.1), which computes the decreasing subsequence D_i from a set of the first decreasing subsequences of each block $D_{0,0} \cup D_{1,0} \cup \ldots \cup D_{p-1,0}$.

First of all, we prove that each element in D_i is in one of the first decreasing subsequences of each block, that is, D_i is a subset of $D_{0,0} \cup D_{1,0} \cup \ldots \cup D_{p-1,0}$. Let $S = (s_0, s_1, \ldots, s_{n-1})$ be an input sequence. We first consider the case of $i = 0$. We assume that $D_0 = (s_{i_0}, s_{i_1}, \ldots, s_{i_k}, \ldots, s_{i_m})$ and s_{i_k} is an element which is not in $D_{0,0} \cup D_{1,0} \cup \ldots \cup D_{p-1,0}$. From the first algorithm we described in Section 3, s_{i_k} satisfies,

$$s_{i_k} = \min\{s_h \mid 0 \leq h \leq i_k\}.$$

Now we also assume that s_{i_k} is in a block S_j and s_{j_0} is the first element of S_j. From the above expression, we obtain the following expression directly.

$$s_{i_k} = \min\{s_h \mid j_0 \leq h \leq i_k\}$$

The expression implies s_{i_k} is in $D_{j,0}$, and the fact is in contradiction to the hypothesis. We can prove D_i is a subset of $D_{0,0} \cup D_{1,0} \cup \ldots \cup D_{p-1,0}$ in case of $1 \leq i \leq m-1$ in the same fashion.

Next, we explain how to compute D_i from $D_{0,0}, D_{1,0}, \ldots, D_{p-1,0}$. As we described in Section 3, we can compute D_i using the prefix minima operation for $D_{0,0}, D_{1,0}, \ldots, D_{p-1,0}$. Since each $D_{j,0}$ $(0 \leq j \leq p-1)$ is a decreasing sequence which is stored in a 2-3 tree, we can compute the prefix minima efficiently from the following reason.

For simplicity, we assume that E_j denotes $D_{j,0}$ and E denotes D_i. As we described above, an element s_{j_k} in E_j is in E if and only if the following condition holds.

$$s_{j_k} = \min\{s_h \mid 0 \leq h \leq j_k\}$$

Let $s_{j_{min}}$ and s_{j_0} be the smallest and the first elements in E_j, respectively. We can modify the above condition using $s_{j_{min}}$ and s_{j_0}.

$$s_{j_k} = \min\{\min\{s_{g_{min}} \mid 0 \leq g \leq j-1\}, \min\{s_h \mid j_0 \leq h \leq j_k\}\}$$

Since each E_j is a decreasing sequence, the latter expression $s_{j_k} = \min\{s_h \mid j_0 \leq h \leq j_k\}$ always holds. Therefore,we finally obtain the following condition.

$$s_{j_k} < \min\{s_{g_{min}} \mid 0 \leq g \leq j-1\}$$

Once we can find such an element s_{j_k} in E_j, the following elements in E_j are also in E since E_j is a decreasing sequence. We can use the $SPLIT$ operation to compute the set of elements because each decreasing sequence is stored in a 2-3 tree.

Based on the above idea, we obtain the following simple procedure.

Procedure 1 (Computation of the i-th decreasing subsequence)
Input: A set of decreasing subsequences $E_0, E_1, \ldots, E_{p-1}$. Each decreasing subsequence E_j $(0 \leq j \leq p-1)$ is stored in a 2-3 tree T_j, and its size is $O(\frac{n}{p})$.
Output: The first decreasing subsequence E such that $E = E'_0 \cup E'_1 \cup \ldots \cup E'_{p-1}$ and $E'_j \subseteq E_j$ for each j $(0 \leq j \leq p-1)$. (Elements in E'_j are stored in a new 2-3 tree T'_j, and the other elements $E_j - E'_j$ are stored in T_j again.)

Step 1: On each processor P_j $(0 \leq j \leq p-1)$, find the smallest element in the tree T_j, and store the element into q_j.

Step 2: Compute the prefix minima of the array $Q = (q_0, q_1, \ldots, q_{p-1})$, and store the result into the same array Q.

Step 3: On each processorP_j $(0 \leq j \leq p-1)$, split T_j into two 2-3 trees T'_j and T_j using q_{j-1}. □

The complexity of Procedure 1 is as follows. Step 1 can be executed in $O(\log \frac{n}{p})$ time using MIN operation for a 2-3 tree in parallel. Step 2 can be executed in $O(\log p)$ time using $O(\frac{p}{\log p})$ processors using a parallel prefix algorithm [13]. Step 3 can be executed in $O(\log \frac{n}{p})$ time using $SPLIT$ operation for a 2-3 tree. Thus the procedure can be executed in $O(\log p + \log \frac{n}{p})$ time using p processors.

4.3. Reconstruction of 2-3 trees

In this subsection, we explain details of the subsection (3.2), which executes reconstruction of 2-3 trees. For each processor P_j, an input of this substep is a set of decreasing subsequences $D_{j,0}, D_{j,1}, \ldots, D_{j,m_j}$ such that each $D_{j,i}$ is stored in a 2-3 tree $T_{j,i}$. Since the reconstruction is executed on each processor in parallel, we describe a sequential procedure for one processor, and assume that F_i $(0 \leq i \leq m-1)$ denotes $D_{j,i}$ and a 2-3 tree T_i stores F_i.

The simplest implementation of this substep is to compute the decreasing cover for $F_0 \cup F_1 \cup \ldots \cup F_{m-1}$ again. However, computation of a decreasing cover needs $O(\frac{n}{p} \log \frac{n}{p})$ computation time, and the algorithm does not become cost optimal. To avoid this, we reconstruct the decreasing subsequences using the following idea. ($F'_0, F'_1, \ldots, F'_{m'-1}$ denote reconstructed decreasing subsequences.)

(1) Let s_{min} be the smallest element in F_0. Split F_1 into F_1 and F'_0 so that every element in F_1 is larger than s_{min} and every element in F'_0 is less than s_{min}. (Note that F'_0 and F_1 are decreasing sequences.)

(2) Concatenate F_0 and F'_0. (The concatenated sequence is stored in F'_0.)

(3) Repeat (1) and (2) for F_i and F_{i+1} $(1 \leq i \leq m-2)$.

Figure 3 shows an example of the above idea. We assume that the following sequence S is an input for the example on a processor.

$$S = (10,\ 8,\ 23,\ 1,\ 3,\ 37,\ 7,\ 21,\ 35,\ 13,\ 2,\ 33,\ 39,\ 4,\ 20,\ 9)$$

We also assume that elements $1, 8$ have been removed in (3.1) of Algorithm 3. Then, $s_{min} = 10$, and elements $2, 3$ are moved from F_1 to F_0. After repetition of the reconstruction, the obtained decreasing subsequences are the decreasing cover of the remaining elements.

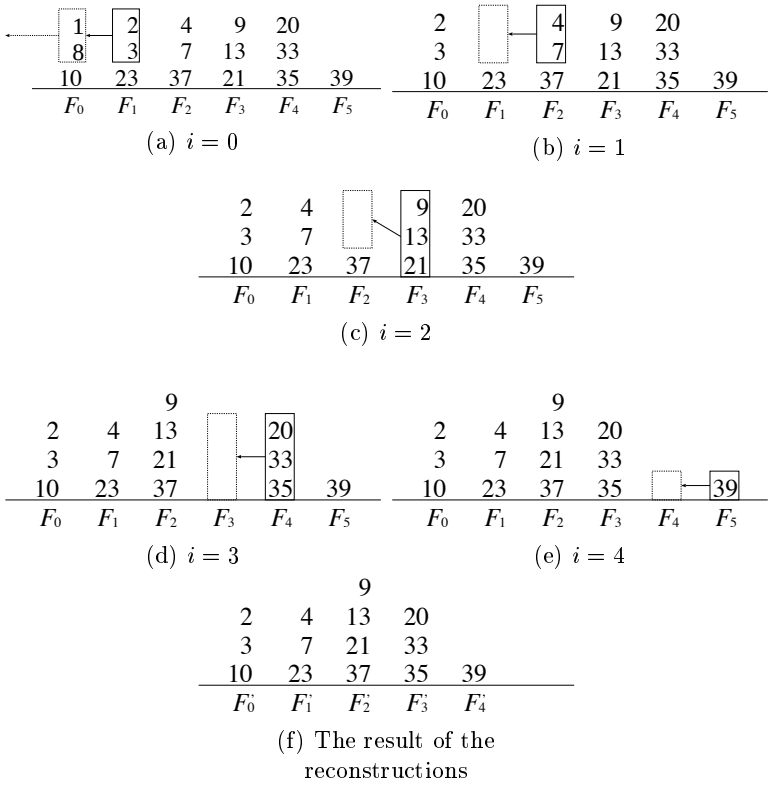

Figure 3: An example of reconstructions of 2-3 trees

For the correctness of the above idea, we prove F'_i only consists of elements in F_i and F_{i+1}. We first consider the case of $i = 0$, and assume that there exists an element $s_g \in F'_0$ which is not in $F_0 \cup F_1$. Since $F_0, F_1, \ldots, F_{m-1}$ is the decreasing cover before (3.1) of Algorithm 3, there exists an element $s_h \in F_1$ which satisfies $s_h < s_g$ and $h < g$. (Recall $S = (s_0, s_1, \ldots, s_{n-1})$ is the input sequence of the patience sorting.) Then, the existence of s_h contradicts to the condition of $s_g \in F'_0$ because each element s_k in the first decreasing subsequence must satisfy $s_k = \min\{s_i \mid 0 \leq i \leq k\}$. We can prove in case of $1 \leq i \leq m-2$ inductively.

Since each decreasing subsequence is stored in a 2-3 tree, implementation of the above idea is not difficult. We show details of the procedure as follows.

Procedure 2 (Reconstruction of 2-3 trees on a processor)
Input: A set of decreasing subsequences $F_0, F_1, \ldots, F_{m-1}$ obtained for a processor

after the substep (3.1) of Algorithm 3. Each decreasing subsequence F_j $(0 \leq j \leq m-1)$ is stored in a 2-3 tree T_j.
Output: A set of decreasing subsequences $F'_0, F'_1, \ldots, F'_{m-1}$ such that the set of decreasing subsequences is the decreasing cover of $F_0 \cup F_1 \cup \ldots \cup F_{m-1}$. Each decreasing subsequence F'_j $(0 \leq j \leq m-1)$ is stored in a 2-3 tree T_j.

Step 1: Set $k = 0$, and repeat the following substeps until $k = m-1$.

(1.1): Find the smallest element in the tree T_k, and store the result in s_{min}

(1.2): Split T_{k+1} into T'_k and T_{k+1} using s_{min} so that every element in F_{k+1} is larger than s_{min} and every element in F'_k is no more than s_{min}.

(1.3): Implant T_k in T'_k, and then, set $k = k+1$. □

The complexity of each substep in the above procedure is $O(\log \frac{n}{p})$ because all of the substeps consist of a constant number of MIN, $IMPLANT$, and $SPLIT$ operations described in Section 2. Since the number of repetition is at most m, the time complexity of the above procedure is $O(m \log \frac{n}{p})$.

4.4. Complexity of the algorithm

As we described in Subsection 4.1, complexity of the algorithm is $O(\log n + \frac{n \log n}{p} + mT_3(n))$, where $T_3(n)$ is the time complexity of substeps (3.1) and (3.2). In addition, complexities of (3.1) and (3.2) are $O(\log p + \log \frac{n}{p})$ and $O(m \log \frac{n}{p})$ from Subsections 4.2 and 4.3, respectively. Then, $T_3(n) = O(\log p + m \log \frac{n}{p})$.

In consequence, we obtain the following theorem.

Theorem 2 *Algorithm 3 solves the patience sorting of n elements in $O(\log n + \frac{n \log n}{p} + m^2 \log \frac{n}{p} + m \log p)$ time using p processors on the EREW PRAM.* □

From the above theorem, the complexity of the algorithm becomes $O(\frac{n \log n}{p})$ in case of $\frac{n}{p} > m^2$, namely $1 \leq p < \frac{n}{m^2}$. In other words, we can solve the patience sorting cost optimally if $m = n^{\epsilon}$ and $1 \leq p < n^{1-2\epsilon}$ where ϵ is a constant which satisfies $\epsilon < \frac{1}{2}$.

5. Conclusion

In this paper, we have proposed two algorithms for the patience sorting. The first algorithm is a parallel algorithm which consists of repetition of the prefix operations. The second one is a parallel algorithm which improves the complexity of the first algorithm, and runs in $O(\log n + \frac{n \log n}{p} + m^2 \log \frac{n}{p} + m \log p)$ time using p processors on the EREW PRAM. The algorithm is cost optimal in case of $1 < p < \frac{n}{m^2}$.

Although P-completeness of the problems has not been proven yet, the proposal of an efficient parallel algorithm for the problem is not easy. We are now considering parallelizability of a number of problems which have similar properties.

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] A. Aldous and P. Diaconis. Longest increasing subsequences: From patience sorting to the baik-deift-johansson theorem. *BAMS: Bulletin of the American Mathematical Society*, 36:413–432, 1999.

[3] S.N. Bespamyatnikh and M. Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76(1–2):7–11, 2000.

[4] C.D. Castanho, W. Chen, K. Wada, and A. Fujiwara. Polynomially fast parallel algorithms for some P-complete geometric problems. In *Proc. Workshop on Computational Geometry*, 2000.

[5] C. Cerin, C. Dufourd, and J. F. Myoupo. An efficient parallel solution for the longest increasing subsequence problem. In *Fifth International Conference on Computing and Information (ICCI'93)*, pages 220–224. IEEE Press, 1993.

[6] R. J. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithm*. The MIT Press, second edition, 2001.

[8] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. In *ACM Symposium on Computational Geometry*, pages 298–307, 1993.

[9] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11:29–35, 1975.

[10] T. Garcia, J.F. Myoupo, and D. Semé. A work-optimal CGM algorithm for the longest increasing subsequence problem. In *The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'01)*, pages 563–569, 2001.

[11] R. Greenlaw, H.J. Hoover, and W.L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford university press, 1995.

[12] D. E. Knuth. *Sorting and Searching*. Volume 3 of The Art of Computer Programming. Addison-Wesley, 1973.

[13] R.E. Ladner and M. J. Fisher. Parallel prefix computation. *Journal of ACM*, 27:831–838, 1980.

[14] C.L. Mallows. Patience sorting. *Bulletin of the Institute of Mathematics and its Applications*, 9:216–224, 1973.

[15] T. Nakashima and A. Fujiwara. Parallelizability of the stack breadth-first search problem. In *The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'01)*, pages 722–727, 2001.

[16] R. Uehara. A measure for the lexicographically first maximal independent set problem and its limits. *International Journal of Foundations of Computer Science*, 10(4):473–482, 1999.