ペイシェンス・ソートおよび最長昇順部分列問題に対するコスト最適な並列アルゴリズム

中島 孝明　　藤原 暁宏
takaaki@zodiac30.cse.kyutech.ac.jp　fujiwara@cse.kyutech.ac.jp

九州工業大学 情報工学部 電子情報工学科

ペイシェンス・ソートおよび最長昇順部分列問題は互いに密接な関係を持つことが知られている問題である．しかし，これら 2 つの問題が，クラス NC，P 完全問題のどちらに属するのかについては知られていない．本稿では，はじめに n 個の異なる整数に対するペイシェンス・ソートを解くアルゴリズムを 2 つ提案する．1 つ目のアルゴリズムは EREW PRAM モデル上で動作し，その計算量は p 台のプロセッサを用いた場合 $O(m(\frac{n}{p}+\log n))$ となる．ここで，m はペイシェンス・ソートの解に含まれる降順部分列の数である．2 つ目のアルゴリズムは CREW PRAM モデル上で動作し，計算量は $O(\frac{n\log n}{p}+m^2\log\frac{n}{p}+m\log p)$ であり，$1<p<\frac{n}{m^2}$ の範囲でコスト最適な並列アルゴリズムとなる．最後に，ペイシェンス・ソートの解から同一の入力に対する最長昇順部分列を求める並列アルゴリズムを提案し，そのアルゴリズムの計算量がコスト最適となることを示す．

Cost optimal algorithms for patience sorting and longest increasing subsequence

Takaaki NAKASHIMA　and　Akihiro FUJIWARA

Department of Computer Science and Electronics, Faculty of Engineering,
Kyushu Institute of Technology

In this paper, we consider parallel algorithms for the patience sorting and the longest increasing subsequence. These two problems are related each other and are not known to be in the class NC or P-complete. We first 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, and the second algorithm runs in $O(\frac{n\log n}{p}+m^2\log\frac{n}{p}+m\log p)$ time using p processors on the CREW PRAM. If $1<p<\frac{n}{m^2}$ is satisfied, the second algorithm becomes cost optimal. Finally, we propose a procedure which computes the longest increasing subsequence from a solution of the patience sorting, and obtain a parallel algorithm, which runs with the same complexity as the algorithm of the patience sorting, for the longest increasing subsequence.

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 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, some problems in the class P seem to have no parallel algorithm 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 P to the problem using NC-reduction. (For details of the P-completeness, see [10].) It is believed that problems in the class NC admit parallelization readily, and conversely, P-complete problems are inherently sequential and hard to be parallelized.

However, some efficient parallel algorithms have been recently proposed for P-complete problems[5, 14, 15]. 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 some inherently sequential problems have cost optimal parallel algorithms and we can parallelize the problems practically.

In this paper, we consider parallel algorithms for two problems related each other. The first problem is the patience sorting, which was invented as a practical method of sorting a real deck of cards[13]. The second problem is the longest increasing subsequence of n distinct integers. Although these two problems are primitive combinatorial optimization problems, both of them are not known to be in the class NC or P-complete, that is, no NC algorithm have been proposed for the problems, and there is no proof which shows the problems are P-complete.

There are a lot of papers which deal with the patience sorting and the longest increasing subsequence. Sequential algorithms[2, 4, 11], show that we can solve the two problems 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 [6, 9]. The former algorithm is for the linear array, which is a classical parallel computation model, and the latter is for the CGM model[8], which is one of practical parallel computation models. However, the parallel algorithms are not cost optimal since costs of both algorithms are $O(n^2)$.

In this paper, we propose efficient parallel algorithms for the problems and consider there parallelizability. First we propose a simple algorithm for the patience sorting. The algorithm consists of repetition of the prefix operations. 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 shows that the algorithm is cost optimal in case of $m = O(\log n)$. Next we propose another parallel algorithm, which is more complicated, for the patience sorting. The second algorithm runs in $O(\frac{n \log n}{p} + m^2 \log \frac{n}{p} + m \log p)$ time using p processors on the CREW PRAM. From the complexity, the algorithm is cost optimal in case of $1 < p < \frac{n}{m^2}$. Finally we propose a procedure which computes the the longest increasing subsequence from a solution of the the patience sorting. Since the procedure only needs $O(n)$ cost on the EREW PRAM, we obtain a parallel algorithm, which runs with the same complexity as the algorithm of the patience sorting, for the longest increasing subsequence.

2 Preliminaries

2.1 Patience sorting and longest increasing subsequence

In this subsection, we make some definitions for the patience sorting and the longest increasing subsequence.

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 no more 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 and decreasing if each subsequence is increasing and decreasing, respectively.* □

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.* □

Definition 4 (Longest increasing subsequence) *Let S be a sequence of n distinct integers. The longest increasing subsequence is a problem to compute an increasing subsequence of S such that length of the subsequence is the longest among all increasing subsequences 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 some solutions for an input of the patience sorting and the longest increasing subsequence. In this paper, our objective for the problems is to find one of the solutions.

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

Longest increasing subsequence = (1, 3, 7, 13, 33, 39)

Fig. 1: An example of the patience sorting and the longest increasing subsequence. Each vertical sequence in the patience sorting forms a decreasing subsequence.

Figure 1 shows an example of the patience sorting and the longest increasing subsequence.

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 use the binary search method with 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.

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

2.2 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 a 2-3 tree.

Definition 5 (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.)

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.

(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 ([1]) *Let T, T_1 and T_2 be 2-3 trees whose sizes are $O(n)$, respectively. 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 $(m_0, m_1, \ldots, m_{n-1})$ such that $m_k = \min\{x_h \mid 0 \leq h \leq k\}$, 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 indices are equal to 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 elements 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 remaining elements, and the other decreasing subsequences are obtained from the same reason.

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 $D_i = (d_{i,0}, d_{i,1}, \ldots, d_{i,l})$ $(0 \leq i \leq m-1)$ 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 sum 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 the parallel prefix[12], we can compute 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 of time complexity, Algorithm 2 is usually not efficient because optimal sequential time complexity of the problem is $O(n \log n)$. However, the algorithm becomes cost optimal in case of the number of the subsequences is $O(\log n)$.

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 CREW

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 P_j $(0 \leq j \leq p-1)$ denotes the j-th processor on the PRAM. 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 set of the first decreasing subsequences of 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 make 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 data structures 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 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 $S = (s_0, s_1, \ldots, s_{n-1})$.
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 $D_j^i = D_i \cap D_{j,0}$ are stored in a new 2-3 tree T_j^i, and the other elements $D_{j,0} - D_i$ 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 CREW 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[11] 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[7]. 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 $D_{0,0} \cup D_{1,0} \cup \ldots \cup D_{p-1,0}$.

For simplicity, we assume that E_j denotes $D_{j,0}$ and E denotes D_i. Although we have to prove correctness of the following procedures, we omit the proof due to space limitation.

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 done in $O(\log \frac{n}{p})$ time using MIN operation for a 2-3 tree in parallel. Step 2 can be done in $O(\log p)$ time using $O(\frac{p}{\log p})$ processors using a parallel prefix algorithm[12]. Step 3 can be done 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 $D_{j,0}, D_{j,1}, \ldots, D_{j,m_j-1}$ 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.

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): 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 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 proof for correctness of the procedure is also omitted due to space limitation.

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 which we described in Section 2. Since the number of repetition is 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 $O(\frac{n \log n}{p} + m^2 \log \frac{n}{p} + m \log p)$ time using p processors on the CREW 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 Procedure for longest increasing subsequence

In this section, we describe how to compute the longest increasing subsequence from a solution of the patience sorting of the same input. We first show that the patience sorting and the longest increasing subsequence are closely related each other by giving the following lemma[2].

Lemma 3 ([2]) *Let m be the length of the longest increasing subsequence of a sequence S. Then, the number of decreasing subsequences in a solution of the patience sorting for the same sequence S is also m.* □

Using the above lemma, we obtain a solution of the longest increasing subsequence from a solution of the patience sorting for the same input $S = (s_0, s_1, \ldots, s_{n-1})$ as follows. We assume that $D_0 \cup D_1 \cup \ldots \cup D_{m-1}$ denotes the decreasing cover of S, and each D_j $(0 \leq j \leq m-1)$ denotes the i-th decreasing subsequence of the cover. Let s_{l_j} be an element in D_j $(1 \leq j \leq m-1)$. Then there exists an element $s_{l_{j-1}} \in D_{j-1}$ which satisfies $s_{l_{j-1}} < s_{l_j}$ and $l_{j-1} < l_j$, and we call $s_{l_{j-1}}$ a *parent element* of s_{l_j}. Given an element $s_{l_{m-1}} \in D_{m-1}$ for s_{l_j}, the sequence of elements connected with the parent relation $S_L = (s_{l_0}, s_{l_1}, \ldots, s_{l_{m-1}})$ is increasing. Therefore S_L is the longest increasing subsequence whose length is equal to the number of decreasing subsequences of the patience sorting. Once parent relations are obtained for all elements, we can find the longest increasing subsequence by tracing the parent relation from an element in D_{m-1} to an element in D_0.

We now consider how to find a parent element for each element in D_j $(1 \leq j \leq m-1)$. Although there may be some candidates for a parent element for each element, we define a parent element $s_{l_{j-1}}$ of s_{l_j} using the following expression.

$$s_{l_{j-1}} = s_k \text{ s.t. } k = \max\{k' \mid k' < l_j, s_{k'} \in D_{j-1}\}$$

Then, $s_{l_{j-1}} < s_{l_j}$ holds because of definition of the patience sorting. Since indices of elements in each decreasing subsequence are increasing, we can find parent relations between two decreasing subsequences using a ranking operation for merging[7]. Moreover we can execute the search operation for each pair of decreasing subsequences in parallel.

We summarize the idea in the followings.

Procedure 3 (Procedure for the longest increasing subsequence)
Input: A solution of the patience sorting for a sequence of distinct integers S.
(We assume that a solution of the patience sorting consists of m decreasing subsequences $D_0, D_1, \ldots, D_{m-1}$.)
Output: A longest increasing subsequence $S_L = (s_0', s_1', \ldots, s_{m-1}')$ for S.

Step 1: For each element s_{l_j} in D_j $(1 \leq j \leq m-1)$, find the parent element $s_{l_{j-1}}$ which satisfies $s_{l_{j-1}} = s_k$ s.t. $k = \max\{k' \mid k' < l_j, s_{k'} \in D_{j-1}\}$.

Step 2: Trace the parent relation from an element in D_{m-1} to an element in D_0, and store traced elements in S_L in reverse order. □

The complexity of Procedure 3 is as follows. Step 1 consists of m independent ranking operations for merging[7]. Since we can execute ranking operation in $O(\log n + \frac{n}{p})$ time using p processors for two sequences whose sizes are $O(n)$, we can execute Step 1 in $O(\log n + \frac{n}{p})$ time p processors on the CREW PRAM. Step 2 can be done in $O(\log n + \frac{n}{p})$ time using a parallel list ranking algorithm[3] because parent relations make a tree structure. Therefore we obtain the following lemma and theorem for Procedure 3 and the longest increasing subsequence, respectively.

Lemma 4 *Procedure 3 computes the longest increasing subsequence from a solution of the patience sorting in $O(\log n + \frac{n}{p})$ time using p processors on the CREW PRAM.* □

Theorem 3 *We can solve the longest increasing subsequence of n elements in $O(\frac{n \log n}{p} + m^2 \log \frac{n}{p} + m \log p)$ time using p processors on the CREW PRAM.* □

6 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 $O(\frac{n \log n}{p} + m^2 \log \frac{n}{p} + m \log p)$ time using p processors on the CREW PRAM. The algorithm is cost optimal in case of $1 < p < \frac{n}{m^2}$. Finally, we have proposed a procedure which computes the longest increasing subsequence from a solution of the patience sorting, and obtain a parallel algorithm, which runs with the same complexity as the patience sorting, for the longest increasing subsequence.

Although P-completeness of both problems have not been proven yet, a proposition of efficient parallel algorithms for the problems is not easy.We are now considering parallelizability of some problems which have similar properties.

References

[1] A. Aho, J. Hopcroft, and J. 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] R. Anderson and G. Miller. Deterministic parallel list ranking. In *Third Aegean Workshop on Computing, AWOC 88*, pages 81–90. Springer-Verlag, 1988.

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

[5] C. 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.

[6] 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.

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

[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] T. Garcia, J. 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.

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

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

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

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

[14] 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.

[15] 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.