Studies on Parallel Algorithms for Inherently Sequential Problems

Takaaki NAKASHIMA

March 2003

Abstract

In parallel computational theory, the class NC and P-completeness are known as primary measures of parallel complexity. It is believed that problems in the class NC admit parallelization readily, and a number of cost optimal parallel algorithms have been proposed for the problems. Conversely, P-complete problems are known as a problems which are hard to be parallelized from the definition, and a few cost optimal parallel algorithms have been proposed for the P-complete problems. In addition, we also focus on problems which are not known to be in the class NC or P-complete. These problems seem to be inherently sequential, and parallelizability of the problems have not considered well. In this dissertation, we shall consider parallelizability of the problems and propose cost optimal parallel algorithms for the problems.

In Chapter 3, we consider parallelizability of a representative P-complete problem, which is a stack breadth-first search (stack BFS) problem. We first prove that the stack BFS is P-complete even if the maximum degree of an input graph is 3. Next, we propose the longest path length (LPL) as a measure for P-completeness of the stack BFS. Using the measure, we propose an efficient parallel algorithm for the stack BFS. Assuming the size and LPL of an input graph is n and l respectively, the complexity of the algorithm shows that the stack BFS is in the class NC if $l = O(\log^k n)$ where k is a positive integer. In addition to this, the algorithm is cost optimal if $l = O(n^\epsilon)$ where $0 < \epsilon < 1$.

In Chapter 4, we consider parallelizability of problems which are not known to be in the class NC or P-complete. The problems are the patience sorting and the longest increasing subsequences. Although these problems have no proof of P-completeness, they are inherently sequential and it seem to hard to be parallelized. 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. 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 for patience sorting, for the longest increasing subsequence.

Keywords:
parallel algorithms, PRAM, P-completeness, cost optimal, breadth-first search, patience sorting, longest increasing subsequence

List of Publications

Journals

1. T. Nakashima, A. Fujiwara, "Cost optimal parallel algorithms for the lexicographically first maximal 3 sum problem on the BSP model", IPSJ Journal : Special Issue on Parallel Processing, Vol. 42, No. 4, pp. 724–731, 2001. (in Japanese)

2. T. Nakashima, A. Fujiwara, "A parallel algorithm for the stack breadth-first search", IEICE Transactions on Information and Systems, Vol. E85-D, No. 12, pp. 1955–1958, 2002.

Domestic and International Conferences

1. T. Nakashima, A. Fujiwara, "Cost optimal parallel algorithms for a P-complete problem", Joint Symposium on Parallel Processing (JSPP2000), pp. 35–42, 2000. (in Japanese)

2. T. Nakashima, A. Fujiwara, "Parallelizability of the stack breadth-first search problem", Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'01), pp. 722–727, 2001.

3. T. Nakashima, A. Fujiwara, "Parallel algorithms for patience sorting and longest increasing subsequence", Proceedings of the International Conference in Networks, Parallel and Distributed Processing and Applications (NPDPA'02), pp. 7–12, 2002.

Technical Reports

1. T. Nakashima, A. Fujiwara, "Parallel algorithms for a P-complete problem on the BSP model", General Conference of IEICE, Information and System, p. 1, 2000. (in Japanese)

2. T. Nakashima, A. Fujiwara, "A parallel algorithm for the stack breadth-first search", Technical Report of IEICE, Vol. 101, No. 431 (COMP2001-52), pp. 1-6, 2001.

3. T. Nakashima, A. Fujiwara, "Cost optimal algorithms for patience sorting and longest increasing subsequence", Technical Report of IPSJ, Vol. 2002, No. 42 (2002-AL-84-7), pp. 43-50, 2002.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In parallel computation theory, one of primary complexity classes is the class NC. Let n be the input size of a problem. A problem is in the class NC if there exists a parallel algorithm which solves the problem in $T(n)$ time using $P(n)$ 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 in the class P is P-complete if we can reduce any problem in the class P to the problem using NC-reduction. It is believed that the P-complete problem is inherently sequential and hard to be parallelized.

However polylogarithmic time complexity is not so important for real parallel computation because the number of processors p is usually small in comparison with the size of a problem n, that is, $p << n$. Thus, *cost optimality* is the most important measure for parallel algorithms in practice. The

cost of parallel algorithm is defined as the product of the running time and the number of processors of the algorithm. A parallel algorithm is called cost optimal if its cost is equal to the lower bound of sequential time complexity for the same problem. In other words, the cost optimal parallel algorithm archives optimal speedup, which is equal to the number of processors.

Therefore, one way to parallelize P-complete problems is to find cost optimal parallel algorithms which runs in polynomial time. Let $\Omega(n^k)$ be lower bound of sequential time complexity for a P-complete problem A. It seems that the problem A has no parallel algorithm which runs in polylogarithmic time since A is P-complete. However, the problem A may has a parallel algorithm witch runs in $O(n^{k-\epsilon})$ time using n^{ϵ} processors, where $0 < \epsilon < k$. Since the parallel algorithm is cost optimal, the algorithm probably archives optimal speed up in practice if the number of processors is not so large. Thus, in this dissertation, we first aim to propose cost optimal parallel algorithms, witch run in polynomial time, for P-complete problems.

Recently, it have been shown that some P-complete problems can be parallelized using efficient parallel algorithms[5, 10, 20, 25]. For example, Uehara[25] considered parallelizability of the lexicographically first maximal independent set (LFMIS) problem, which is a well-known P-complete problem. He proposed a measure, which is called the *longest directed path length* (LDPL), for P-completeness of LFMIS, and proved that LFMIS is P-complete if LDPL is $O(n^{\epsilon})$ where $0 < \epsilon < 1$. He also proposed a cost optimal parallel algorithm which runs in $O(l)$ time using $O(\frac{n+m}{l})$ processors, where n, m and $l$are the numbers of vertices, edges and LDPL, respectively.

The results imply that there are two goals to parallelize P-complete problems. The first one is to find a measure which characterizes P-completeness of the problems. (The measure must be computed in NC.) The second one is proposition of cost optimal parallel algorithms, which run in polynomial time, for the problems.

In Chapter 3, we consider parallelizability for a representative P-complete problem, which is a stack breadth-first search (stack BFS) problem[21, 22]. The stack BFS is a well-known graph searching technique, and proved to be P-complete[13]. We first prove that the stack BFS is P-complete even if the maximum degree of an input graph is 3. Next, we propose a measure for P-completeness of the stack BFS. We call the measure LPL, which means the longest path length of a graph. Finally, we propose an efficient parallel algorithm for the stack BFS using the measure. Assuming the size and LPL of an input graph is n and l respectively, the complexity of the algorithm shows that the stack BFS is in NC if $l = O(\log^k n)$ where k is a positive integer. In addition to this, the algorithm is cost optimal if $l = O(n^\epsilon)$ where $0 < \epsilon < 1$.

Next, we consider parallelizability of problems which are not known to be in the class NC or P-complete. The problems seem to be inherently sequential from their definitions. The parallelizability of the problems have not been considered well, and only a few parallel algorithm have been proposed for the problems. However, we can propose cost optimal parallel algorithms for P-complete problems, then the problems may also have cost optimal parallel algorithms which runs in polynomial computation time. In this disserta-

tion, we consider parallelizability and parallel algorithms for the inherently sequential problems.

The first inherently sequential problems considered in this dissertation is the patience sorting, which was invented as a practical method of sorting a real deck of cards[18]. The second problem is the longest increasing subsequence of n distinct integers[16]. 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, 16], which have been proposed for the two problems, 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 problems[6, 11]. The former algorithm is for the linear array, which is a classical parallel computation model, and the latter is for the CGM model[9], 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 Chapter 4, we propose efficient parallel algorithms for the problems and consider its parallelizability[23]. We first propose a simple algorithm for the patience sorting. The algorithm consists of repetition of the prefix operations, and 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 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 patience sorting, for the longest increasing subsequence.

In Chapter 5, we summarize our result described in this dissertation and consider some future researches.

Chapter 2

Preliminaries

2.1 P-completeness

In this section, we first define two complexity classes of problems. The first class, P, is the class of problems widely used in sequential computations. The second class NC, is also the class of problems used in parallel computations. Let n be the input size of a problem. The above two classes are defined as follows.

Definition 1 *A problem is in the class P if there is a sequential algorithm which solves the problem in $O(n^k)$ computational time, where k is a constant that does not depend on n.* □

Definition 2 *A problem is in the class NC if there exists a parallel algorithm which solves the problem in $O(\log^{k_1} n)$ computational time using n^{k_2} processors, where both of k_1 and k_2 are constants which do not depend on n.* □

The class P is a well-known class which denotes sequential efficiency, that is a problem in the class P has a feasible solution in sequential computation. An analogous class of efficiency for parallel computation is the class NC.A problem in the class NC is believed to have an efficient parallel algorithm.

Next, we define reducibility of problems.

Definition 3 *Let P_1 and P_2 be two problems in the class P. The problem P_1 is NC-reducible to the problem P_2 if there exists an NC-algorithm that transforms an arbitrary instance u_1 for P_1 into an instance u_2 for P_2 such that a solution of P_1 with u_1 is equal to a solution of P_2 with u_2.* □

Note that the reducibility is not symmetric. The existence of the NC transformation implies an algorithm for solving P_1 given any algorithm for solving P_2.

Using the above classes and notation, P-completeness of a problem is defined as follows.

Definition 4 *A problem Q is P-complete if the following two conditions are satisfied.*

(1) *The problem Q is in the class P.*

(2) *For every problem S in the class P, S is NC-reducible to Q.* □

(For details of the P-completeness, see [14].)

From the definition, we can prove P-completeness of a problem in the class P if we can reduce a known P-complete problem to the problem using NC-reduction.

2.2 PRAM

PRAM (Parallel Random Access Machine)[15] is a theoretical parallel computational model used in this dissertation.

The PRAM employs an arbitrary number of processors that can access a shared memory. Each processor has its own local memory and can execute its own local program. The processor also knows its processor number and can use the number in algorithms. All processors operate synchronously and execute the same algorithm. The PRAM is illustrated in Figure2.1.

Several models of the PRAM have been proposed with regard to simultaneous reading and writing to single memory cell. The EREW (exclusive read exclusive write) PRAM dose not allow any concurrent access to a single memory cell. The CREW (concurrent read exclusive write) PRAM allows concurrent read access only. Concurrent accesses to the same memory cell for read or write instructions are allowed on the CRCW (concurrent read concurrent write) PRAM.

In the case of the CRCW PRAM, different assumptions are made to resolve the concurrent write conflicts. For each set of concurrent writings into the same memory cell, the common CRCW PRAM assumes that all writings of the set involve the same value, the arbitrary CRCW PRAM allows an arbitrary write succeed, and the priority CRCW PRAM assumes that a writing with the minimum index succeeds. Although other variations of the CRCW PRAM also exist, they do not differ from the above models substantially in their computational powers.

Note that the CRCW PRAM is strictly more powerful than the CREW

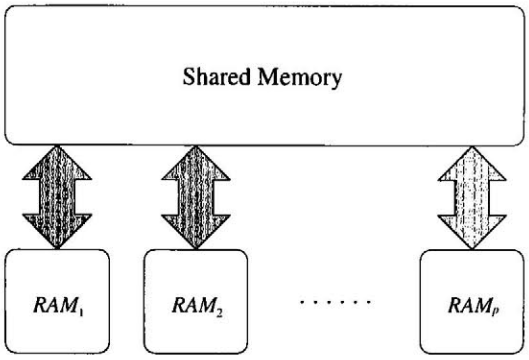


Figure 2.1: The PRAM

PRAM, and the CREW PRAM is strictly more powerful than the EREW PRAM. The reason is as follows. From the above definitions, all EREW PRAM (resp. CREW) algorithms run on the CREW (resp.CRCW) PRAM with the same complexities. In addition, there exist some n input problems which can be solved in $O(1)$ time on the CREW (resp. CRCW) PRAM, and needs $\Omega(\log n)$ time on the EREW (resp. CREW) PRAM. (For example, see [15].)

Chapter 3

Parallelizability for the stack breadth-first search

3.1 Definition of the stack breadth-first search

The breadth-first search (BFS) is one of basic techniques for graph searching. Let $G = (V, E)$ be an input graph. The BFS of G starts by visiting a given vertex $s \in V$. After visiting s, all adjacent vertices to s are visited, and the vertices are stored in a set of vertices V_1. Next, all non-visited vertices which are adjacent to $v \in V_1$ are visited, and stored in a set of vertices V_2. The process is repeated until the entire graph is processed. For each vertex $v \in V_i$, we define that i is the level of v. The level of a vertex denotes a distance on a shortest path from the start vertex s. Using the BFS, we obtain a tree $T = (V, E')$, which is rooted s, by keeping track of the edges which lead to vertices in the next level during the search. We call the tree T the BFS tree.

There are two natural data structures for implementing the BFS, which are a queue and a stack. We assume that an input graph is stored by adja-

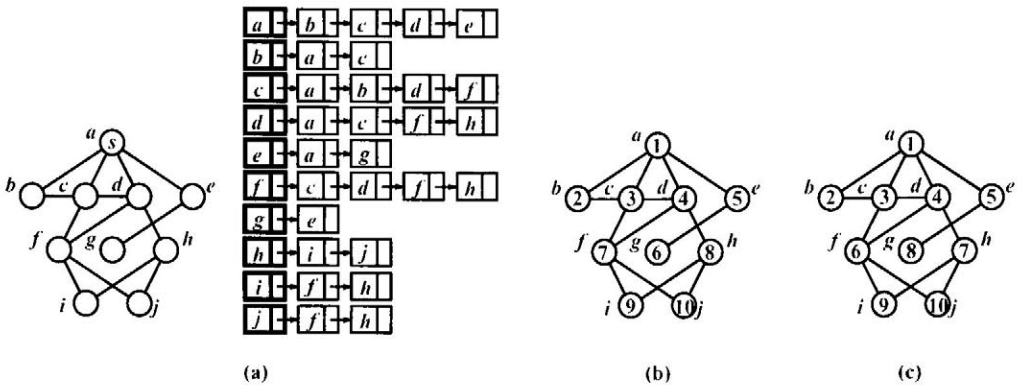


Figure 3.1: Examples of BFS. (a) An input graph and its adjacency lists. (b) Queue BFS numbering. (c) Stack BFS numbering.

cency lists. Figure 3.1 illustrates the difference between the two implementations, the *queue BFS* and the *stack BFS*. The main difference between the two BFSs is order of storing vertices in the searching process. For each level i, all non-visited vertices are stored using a queue in the case of the queue BFS. On the other hand, the vertices are stored using a stack for the stack BFS. It is worth while noticing that the level of each vertex is the same for the two BFSs because the level denotes a distance from the start vertex.

It has been showed that parallelizing the queue BFS is not so hard by some efficient NC algorithms[12, 13]. However, the stack BFS has no NC parallel algorithm, and it is proved to be P-complete[13]. The fact shows that the stack BFS is inherently sequential and hard to be parallelized.

3.2 P-completeness for the maximum degree

In this section, we show that the stack BFS is P-complete even if the maximum degree of an input graph is 3. The fact means that degree of an input graph has no relation to parallelizability of the stack BFS.

Theorem 1 *The stack BFS is P-complete for undirected graphs whose maximum degree is 3.*

(Proof)

Since it is obvious that the problem is in the class P, we prove P-completeness of the problem by NC-reduction from the stack BFS for graphs with unlimited degree, which is proved to be P-complete[13].

First, we transform an input graph $G = (V, E)$ to a layered network. The layered network is a graph $N = (V, E')$ such that there exists a division of the vertex set V into l subsets $V_1 \cup V_2 \cup \ldots \cup V_l = V$, and every edge has one end in V_i and the other in V_{i+1}. To obtain the layered network, we compute the queue BFS for an input graph, and divide the vertex set V into subsets according to levels obtained from the BFS tree. Since the level of a vertex is the same between the queue and stack BFSs, we can remove every edge except for edges which connect two vertices in adjacent sets of vertices for computation of the stack BFS.

Next, we transform the layered network so that degree of each vertex becomes exactly 3. Let d be the maximum degree of the layered network. We first create a complete binary tree T such that the number of its leaves is between d and $4d-1$ and its height is odd. (There exists only one tree which

satisfies the condition.) Let us consider vertex $v \in V_i$ $(1 \leq i \leq l-1)$. A star is obtained from vertex v and its adjacent vertices $v_1', v_2', \ldots, v_l' \in V_{i+1}$. We replace the star with a copy of T so that the vertex and its adjacent vertices correspond to a root and leaves respectively. More precisely, we remove all edges from a star and set v to a root of the tree T. The vertices $v_1', v_2', \ldots, v_l'$ are set to the leaves in left-to-right ordering. Although some leaves are redundant, we leave them untouched. It is worth while noticing that the replacement keeps order of vertices $v_1', v_2', \ldots, v_l'$ in the stack BFS numbering before and after the replacement because the height of the tree T is odd. Using the replacement, we can transform a star for each vertex in V_i $(1 \leq i \leq l-1)$ to the tree T in parallel. The transform also keeps orders in whole BFS numbering.

After the above transform, we compute the stack BFS numbering for the layered network, and obtain a result for the original stack BFS by removing results for added vertices in the transform. We can process the removal efficiently in parallel using prefix operations[17].

The size of the layered network is $O(n^2)$, and we can process all of the above reduction in NC. □

3.3 A measure for P-completeness of the stack BFS

In this section we propose a measure for P-completeness of the stack BFS, and show the measure is valid with changing its value.

Let $G = (V, E)$ and $s \in V$ be an input graph and a start vertex for the

stack BFS, respectively. We define the *longest path length* (LPL) for G with s as follows.

Definition 5 *The longest path length (LPL) for a graph G with a vertex s is the maximum length of the shortest paths from s on the graph G.* □

We can compute LPL of a graph in NC using parallel algorithms for the queue BFS[13] or the single source shortest path problem[24].

We obtain the following theorem which shows validity of LPL.

Theorem 2 *The stack BFS is P-complete even on a graph such that LPL is $O(n^\epsilon)$, where ϵ is a constant which satisfies $0 < \epsilon < 1$.*

(Proof)

The stack BFS problem whose LPL is $O(n^\epsilon)$ is in the class P obviously since the problem can be computed in $O(n^2)$ time using optimal sequential algorithm[19]. In addition, the stack BFS problem whose LPL is $O(n)$ is also P-complete from the fact that the general stack BFS problem is P-complete. We show below that the stack BFS whose LPL is $O(n^\epsilon)$ is P-complete by reducing the general stack BFS problem to the objective problem.

We assume that a graph $G = (V, E)$ and a vertex $s \in V$ are the input for the stack BFS problem such that LPL is cn, where c is a constant which satisfies $0 < c < 1$. First, we generate a new vertex set $V' = \{v'_1, v'_2, \ldots, v'_q\}$, where $q = \lceil n^{\frac{1}{\epsilon}} \rceil - n$, and a new edge set $E' = \{(s, v'_k) | v'_k \in V'\}$.

We consider a new graph $G' = (V \cup V', E \cup E')$. The number of vertices of G' is $n' = \lceil n^{\frac{1}{\epsilon}} \rceil$. On the other hand, LPL of the graph G' is cn, which is

the same as the input graph. Since $cn \leq c(n'^{\epsilon})$ holds, G' is a graph whose number of vertices and LPL are n' and $O(n'^{\epsilon})$.

We compute the stack BFS of the graph G', and assume that $N'(v)$ denotes the BFS numbering for a vertex v on the graph G'. The BFS numbering $N(v)$ on the graph G is given by a simple subtraction $N(v) = N'(v) - |V'|$. (We assume that added vertices are searched earlier than original vertices adjacent to the start vertex s.) We can implement the above reduction in NC since n' is a polynomial function for n. □

In the next section, we show that the stack BFS is in NC if its LPL is $O(\log^k n)$ where k is a positive constant.

3.4 A parallel algorithm for the stack BFS

In this section, we describe a parallel algorithm for the stack BFS on the CREW PRAM. We make brief definitions before showing our algorithm. Let T be a stack BFS tree obtained from an input graph. If a vertex w is adjacent to a vertex v on the path from the root of a tree T to v, then w is a *parent* of v, and v is a *child* of w.

The outline of the algorithm is as follows. Let $G = (V, E)$ with a start vertex s be an input graph. We transform the graph G to a layered network N, which is described in Section 3. We assume that the layered network is $N = (V, E')$ such that $V = V_0 \cup V_1 \cup \cdots \cup V_l$ and $V_0 = \{s\}$. (The l denotes LPL of the graph.) Then, we set the BFS number of s to 1, and repeat the following substeps from $i = 1$ to l. First, for each vertex $v \in V_i$, we select an adjacent vertex $w \in V_{i-1}$ whose BFS number is the maximum among

adjacent vertices. (The vertex w is a parent of v in the stack BFS tree of G.) Second, we compute the number of children of w in the stack BFS tree. Finally, for each vertex $v \in V_i$, we compute the stack BFS numbering from the obtained numbers of children using the prefix operations.

Details of the algorithms are described below. (We assume that all arrays used in the algorithm are initialized with 0 for convenience.)

Algorithm 1 (Parallel algorithm for the stack BFS)

Input: a graph $G = (V, E)$ and a start vertex $s \in V$.

Output: the stack BFS numbering for each vertex. (We assume that $N(v)$ denotes the stack BFS number of a vertex v.)

Step 1: Execute the following substeps.

(1-1) Transform the input graph G to a layered network $N = (V, E')$ using the queue BFS algorithm or the single source shortest path algorithm. (We assume that $L(v)$ denotes the level of the vertex v, and V is divided into $V_0 \cup V_1 \cup \cdots \cup V_l$ such that $V_0 = \{s\}$.)

(1-2) For each vertex v_j $(1 \leq j \leq n)$, set $NL_i(v_j) = 1$ if $L(v_j) = i$, otherwise, set $NL_i(v_j) = 0$ for i $(1 \leq i \leq l)$. (The $NL_i(v_j)$ denotes whether v_j is in the level i.)

(1-3) For each level i $(1 \leq i \leq l)$, compute the prefix sums for $NL_i(v_j)$ $(1 \leq j \leq n)$, and store the results in the same array $NL_i(v_j)$ $(1 \leq j \leq n)$. (After this substep, the $NL_i(v_j)$ means order of the vertex v_j in the level i.)

(1-4) For each vertex v_j $(1 \leq j \leq n)$, set $LL_i(NL_i(v_j)) = v_j$ if $L(v_j) = i$. (The array LL_i is a list of vertices in the level i.)

Step 2: Set $N(s)$=1, and repeat following substeps from i=1 to l.

(2-1) For each vertex $v_k \in V_{i-1}$, set $CH_{v_k}(v_j) = 0$ for all $1 \leq j \leq n$. Then, for each vertex $v_j \in V_i$, determine $w \in V_{i-1}$ which satisfies the following condition, and set $P(v_j) = w$ and $CH_w(v_j) = 1$.

$$N(w) = \max\{N(x) | x \in V_{i-1}, (x, v_j) \in E'\}.$$

(The $CH_w(v_j)$ denotes whether v_j is a child of w, and the $P(v_j)$ denotes a parent of v_j in the BFS tree.)

(2-2) For each vertex $v_k \in V_{i-1}$, compute the prefix sums of $CH_{v_k}(v_j)$ $(1 \leq j \leq n)$, and store the result in the same array $CH_{v_k}(v_j)$ $(1 \leq j \leq n)$. In addition to this, set $NCH(v_k) = CH_{v_k}(v_n)$. (After this substep, the $CH_w(v_j)$ denotes order of v_j among children of w in the BFS tree, in case that v_j is a child of w, and the $NCH(v_k)$ denotes the number of children of a vertex v_k in the BFS tree.)

(2-3) Using the array LL_{i-1}, compute a list of vertices in V_{i-1}, which is $(v_{k_1}, v_{k_2}, \ldots, v_{k_r})$ such that $N(v_{k_1}) > N(v_{k_2}) > \ldots > N(v_{k_r})$. Then, compute the prefix sums of $NCH(v_{k_1}), NCH(v_{k_2}), \ldots, NCH(v_{k_r})$, and store the results in $OS(v_{k_1}), OS(v_{k_2}), \ldots, OS(v_{k_r})$. (The $OS(v_k)$ means offset for BFS numbering in each level for a vertex whose parent is v_k.)

(2-4) For each vertex $v_j \in V_i$, compute the stack BFS numbering using the

following expression.

$$\begin{aligned} N(v_j) &= \sum_{k=0}^{i-1} |V_k| + OS(P(v_j)) \\ &\quad -NCH(P(v_j)) + CH_{P(v_j)}(v_j) \end{aligned}$$

□

Now we consider complexity of the above parallel algorithm in two cases. Let p be the number of processors.

The case of $n \leq p$:

First we consider complexity of Step 1. In the substep (1-1), we compute a level of each vertex in the layered network by the queue BFS algorithm[13], which runs in $O(\frac{n^3}{p} + \log^2 n)$ time on the CREW PRAM. (Redundant edges are removed easily in parallel.) Since the substep (1-2) consists of simple assignments for the size of n^2, this substep can be executed in $O(\frac{n^2}{p})$ time. The substep (1-3) can be executed in $O(\frac{l \times n}{p} + \log n)$ time because this substep consists of l independent prefix sums of the size n. We can compute the substep (1-4) in $O(\frac{n^2}{p})$ time easily.

Next, the complexity of Step 2 is as follows. We consider complexity of each iteration of Step 2. In the substep (2-1), we can execute all operations in $O(|V_i| \times \frac{n}{p} + \log n)$ time using a simple parallel algorithm[17], which computes the prefix operation, and basic operations. In the same way, the substep (2-2) can be executed in $O(|V_{i-1}| \times \frac{n}{p} + \log n)$. Similarly, the substep (2-3) runs in $O(\frac{|V_{i-1}|}{p} + \log |V_{i-1}|)$ time per iteration, and the substep (2-4) runs in $O(\frac{n}{p})$ time easily. Consequently, we can prove that complexity of each iteration is

$O((|V_{i-1}| + |V_i|) \times \frac{n}{p} + \log n)$.

Since the number of repetition l of Step 2 is equal to LPL of the graph G, total complexity of Step 2 is given by,

$$\begin{aligned} & O(\sum_{i=1}^{l}\{(|V_{i-1}| + |V_i|) \times \frac{n}{p} + \log n\}) \\ \leq\ & O(\frac{2n}{p}\sum_{i=0}^{l}\{|V_i|\} + l\log n) \\ =\ & O(\frac{n^2}{p} + l\log n). \end{aligned}$$

Therefore the stack BFS algorithm runs in $O(\frac{n^3}{p} + \log^2 n + l\log n)$ time totally, and we obtain the following theorem and corollary.

Theorem 3 *We solve the stack BFS in $O(\frac{n^3}{p} + \log^2 n + l\log n)$ time if $p \geq n$ where l is LPL of an input graph on the CREW PRAM.* □

Corollary 1 *The stack BFS is in NC on a graph such that LPL is $O(\log^k n)$ where k is a positive constant.* □

The case of $p < n$:

In Step 1, we compute a level of each vertex by a cost optimal parallel algorithm for computing the single source shortest path problem[24]. The algorithm[24] runs in $O(\frac{n^2}{p})$ time using p processors, where $1 \leq p \leq n^\epsilon$ and ϵ is a constant which satisfies $0 < \epsilon < 1$. We can compute Step 2 in $O(\frac{n^2}{p} + l\log n)$ time by the same way mentioned above. Since $\frac{n^2}{p} > n\log n \geq l\log n$ asymptotically holds in case of $1 \leq p \leq n^\epsilon$ where $0 < \epsilon < 1$, we obtain the following theorem.

Theorem 4 *We solve the stack BFS in* $O(\frac{n^2}{p})$ *time if* $1 \leq p \leq n^{\epsilon}$ *where* $0 < \epsilon < 1$ *on the CREW PRAM.* □

The above algorithm is cost optimal because the product of its time complexity and the number of processors is equal to the sequential time complexity.

Chapter 4

Parallel algorithms for patience sorting and longest increasing subsequences

4.1 Definitions

4.1.1 Patience sorting and longest increasing subsequence

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

The patience sorting is known as a traditional card game in British. 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 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 optimal solution for the problem.

Now we describe a precise definition of the patience sorting and the longest increasing subsequence. We make some related definitions before describing the problems.

Definition 6 (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 7 (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.* □

Using the above two definitions, the patience sorting and the longest increasing subsequence is defined as follows.

Definition 8 (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 9 (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.

Figure 4.1 shows an example of the patience sorting and the longest increasing subsequence. In the description, 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

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

Figure 4.1: An example of the patience sorting and the longest increasing subsequence. Each vertical sequence in the patience sorting forms a decreasing 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[8] 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.* □

4.1.2 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 10 (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

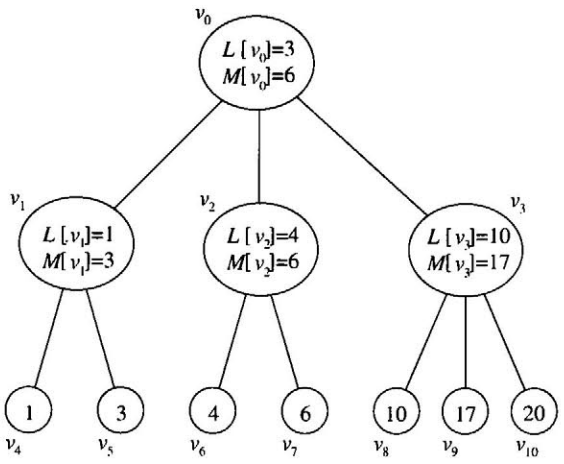


Figure 4.2: An example of a 2-3 tree for a sequence (1, 3, 4, 6, 10, 17, 20)

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 4.2 shows an example of a 2-3 tree.

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.* □

4.2 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 parallel prefix[17], 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 5 *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 $O(\log n)$.

4.3 Second algorithm for the patience sorting

4.3.1 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[16] 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.

4.3.2 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 $1 \leq i \leq m-1$ with 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 and 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 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[17]. 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.

4.3.3 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.

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 no more 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 4.3 shows an example of the above idea. We assume that the following sequence S is an input for the example,

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

and the elements $1, 8$ were removed in (3.1) of Algorithm 3. After the reconstruction, the obtained decreasing subsequences are same as the decreasing cover of S.

We prove F'_i consists of F_i and F_{i+1} for the correctness of the above idea. We 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 $F_0, F_1, \ldots, F_{m-1}$ was 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

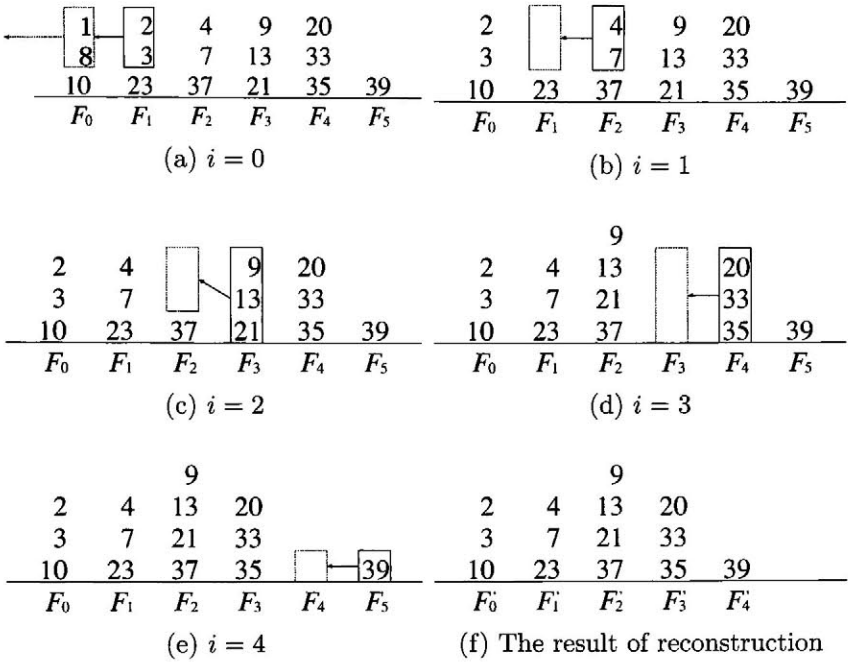


Figure 4.3: An example of reconstruction of 2-3 trees

sorting.) The s_h must be included in F'_0 from definition of the decreasing cover. Then s_g is not included in F'_0, and the fact is in contradiction to the hypothesis. 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): 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 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 $O(m \log \frac{n}{p})$.

4.3.4 Complexity of the algorithm

As we described in Subsection 4.3.1, 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 6 *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 $\epsilon < \frac{1}{2}$.

4.4 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

$s_{l_{m-1}} \in D_{m-1}$, 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 subsequence of the patience sorting. Figure 4.4 illustrates an example of the above idea. In Figure 4.4, each pointer denotes parent relation, and each number in parenthesis denotes index of the element in the input sequence. 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.

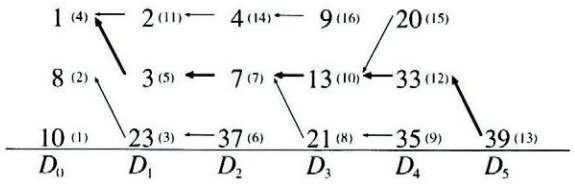
$$s_{l_{j-1}} = s_k \quad \text{s.t.} \quad k = \max\{k' \mid k' < l_j, s_{k'} \in D_{j-1}\}$$

Then, $s_{l_{j-1}} < s_{l_j}$ holds because of definition of the patience sorting. Since indices of elements in each decreasing subsequences 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.

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



Longest increasing subsequence = 1 3 7 13 33 39

Figure 4.4: An example of the idea. Each pointer denotes a parent relation and each number in parenthesis denotes index of the element in an input sequence.

(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 paral-

lel 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 7 *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.* □

Chapter 5

Conclusions

In this dissertation, we presented some efficient parallel algorithms for inherently sequential problems.

In Chapter 3, we first proved that the stack BFS is P-complete even if its maximum degree is 3, and suggested the longest path length as the measure of the stack BFS. Using the measure, we next proposed an efficient parallel algorithm for the stack BFS. The algorithm solves the stack BFS in $O(\frac{n^2}{p})$ time if $1 \leq p \leq n^{\epsilon}$ where $0 < \epsilon < 1$ on the CREW PRAM. The algorithm is cost optimal because the product of its time complexity and the number of processors is equal to the sequential time complexity.

In Chapter 4, we proposed two algorithms for the patience sorting. The first algorithm is a parallel algorithm which consists of repetition of the prefix operations. The second algorithm 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 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, an algorithm for 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 does not seem to be easy.

We summarize our results in Table 5.1.

In the future research, we shall investigate a measure for other P-complete problems. We think that other P-complete problems also have such parameters, and it may be useful criteria for the classification and the choice of parallel techniques. We are also interested in considering parallelizability of some problems which have similar properties to the patience sorting : inherently sequential problems. We think it is important to know whether these problems are in the class NC or not.

Table 5.1: List of results.

Problem	Time	Processors	Model
stack BFS	$O(\frac{n^2}{p})$	$p(1 \leq p \leq n^{\epsilon})$	CREW PRAM
patience sorting	$O(\frac{n \log n}{p} + m^2 \log \frac{n}{p} + m \log p)$	$p(1 < p < \frac{n}{m^2})$	CREW PRAM
longest increasing subsequence	$O(\frac{n \log n}{p} + m^2 \log \frac{n}{p} + m \log p)$	$p(1 < p < \frac{n}{m^2})$	CREW PRAM

Acknowledgements

I would like to express my gratitude for continual encouragement and invaluable suggestions received from Associate Professor Akihiro Fujiwara.

I also would like to thank Professor Kyoki Imamura, Professor Tsutomu Sasao and Professor Ken-ichi Kakizaki, who are my dissertation committee at Kyushu Institute of Technology, for many comments. Their comments are very useful for the improvement of my presentation.

I wish to thank Mr. Masashi Ito and the rest of the members of my laboratory for their kindly supports and suggestions.

Bibliography

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

[5] 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.

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

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, 2001.

[9] 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.

[10] A. Fujiwara, M. Inoue, and M. Toshimitsu. Parallelizability of some P-complete problems. In *Proc. Workshop on Advances in Parallel Computational Models (Lecture Notes in Computer Science, 1800)*, pages 116–122, 2000.

[11] 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.

[12] H. Gazit and G.L. Miller. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Information Processing Letters*, 28(2):61–65, 1988.

[13] R. Greenlaw. A model classifying algorithms as inherently sequential with applications to graph searching. *Information and Computation*, 97(2):133–149, 1992.

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

[15] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, 1992.

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

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

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

[19] Edward F. Moore. The shortest path through a maze. In *Proc. the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[20] T. Nakashima and A. Fujiwara. Cost optimal parallel algorithms for the lexicographically first maximal 3 sum problem on the BSP model. *IPSJ Journal : Special Issue on Parallel Processing*, 42(4):724–731, 2001.

[21] 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.

[22] T. Nakashima and A. Fujiwara. A parallel algorithm for the stack breadth-first search. *IEICE Transactions on Information and Systems*, E85-D(12):1955–1958, 2002.

[23] T. Nakashima and A. Fujiwara. Parallel algorithms for patience sorting and longest increasing subsequence. In *The International Conference in Networks, Parallel and Distributed Processing and Applications (NPDPA'02)*, pages 7–12, 2002.

[24] P. C. Paige and C. P. Kruskal. Parallel algorithms for shortest path problems. In *Proc. the 1985 International Conference on Parallel Processing*, pages 14–20, 1985.

[25] 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.