Algorithms for Testing Occurrences of Length 4 Patterns in Permutations*

Yijie Han
School of Computing and Engineering,
University of Missouri at Kansas City,
Kansas City, MO 64110, USA
E-mail: hanyij@umkc.edu

Sanjeev Saxena
Dept. of Computer Science and Engineering,
Indian Institute of Technology,
Kanpur, INDIA-208 016
Email: ssax@iitk.ac.in

September 14, 2015

Abstract

In this paper we present new sequential and parallel algorithms for testing pattern involvement for all length 4 permutations. Our serial algorithms take $\Theta(n)$ time. For most of these patterns the previous best algorithms require $O(n \log n)$ time. Our parallel algorithms have the complexity of $O(\log n)$ time with $n/\log n$ processors on the CREW PRAM model, $O(\log \log \log n)$ time with $n/\log \log \log n$ processors or constant time and $n \log^3 n$ processors on a CRCW PRAM model. Parallel algorithms were not designed before for some of these patterns and for other patterns the previous best algorithms require $O(\log n)$ time and n processors on the CREW PRAM model.

Keywords: Separable permutations, length 4 permutations, pattern matching, optimal algorithms, parallel algorithms.

*Preliminary version of this paper was published in Y. Han, S. Saxena: Parallel Algorithms for Testing Length Four Permutations, International Symposium on Parallel Architectures, Algorithms and Programming (PAAP'2014), 81-86 and Y. Han, S. Saxena. Algorithms for testing length four permutations. Proceedings 2013 International Frontiers in Algorithmic Workshop (FAW-AAIM'2013) LNCS 7924, 17-23.

1 Introduction

Pattern containment (also called pattern involvement) is a well studied problem in both Computer Science and Combinatorics [1] (see [9] for a survey of results and [10] for a comprehensive introduction to the area of permutation patterns).

Two permutations $P = p_1 p_2 \cdots p_k$ and $P' = p'_1 p'_2 \cdots p'_k$ are said to be order isomorphic if their letters are in the same relative order, i.e., $p_i < p_j$, if and only if, $p'_i < p'_j$. For example permutation $1, 3, 2, 4$ is order isomorphic to $7, 19, 15, 23$.

A permutation $P = p_1 p_2 \cdots p_k$ is said to be present in (or is involved in) another permutation $P' = p'_1 p'_2 \cdots p'_n$ if P' has a subsequence which is order isomorphic to P. The p_i's need not be consecutive in P'. In this context, we say that a permutation P' *contains the pattern* P.

The general problem of testing presence of one permutation in another permutation is NP-complete [4]. However polynomial time algorithms are known when

1. $P = 1, 2, \cdots, k$. This is the largest increasing subsequence problem [5].

2. P' is separable [4]. A permutation is separable if it neither contains the pattern $2, 4, 1, 3$ nor its reverse $3, 1, 4, 2$.

3. k is a constant. Brute force algorithm will take $O(n^k)$ time.

In the case when $k = 3$, linear time algorithms are possible [1]. Further, for $k = 4$, a linear time algorithm exists to test whether a pattern $2, 4, 1, 3$ (or its reverse $3, 1, 4, 2$) is present; this is basically a test to check whether a pattern is separable [4]. Linear time algorithms are also known [5] for monotone patterns $1, 2, 3, 4$ and (its reverse) $4, 3, 2, 1$.

Albert et. al. [1] studied the general problem of permutation involvement and proposed $O(n \log n)$ time algorithms for all patterns of length 4. Their algorithms use orthogonal range queries of the kind:

> Find the smallest number larger than a query item x between positions p and q.

As general orthogonal range queries take $O(\log n)$ query time after $O(n \log n)$ preprocessing time [11, Theorem 2.12], $O(n \log n)$ time algorithms appear to be the best possible using this approach.

Our improvement comes firstly from use of the "usual" range maxima (minima) queries (instead of orthogonal range queries) of the kind:

> Find the largest (smallest) number between positions p and q.

This works, in sequential setting, for all cases except the case $1, 3, 2, 4$, for which we use a particular kind of analysis to achieve $\Theta(n)$ time. In this paper, we describe new linear time serial and optimal parallel algorithms for all patterns of length 4. Parallel algorithms have time complexity $O(\log \log \log n)$ with $n/\log \log \log n$ processors or constant time with $n \log^3 n$ processors on a Concurrent Read Concurrent Write model. On the Concurrent Read

Exclusive Write model, these algorithms will take $O(\log n)$ time and $n/\log n$ processors. Our algorithms are the first parallel algorithms for all length 4 permutations.

In [14] a CREW PRAM parallel algorithm with $O(\log n)$ and n processors is given for the pattern $2, 4, 1, 3$ and its reverse $3, 1, 4, 2$. In [14], the pattern is divided into several groups. Initially each group has very few items, where presence of a pattern can be trivially checked in $O(1)$ time. Then, in each step two groups are combined. Assuming the pattern is not present in either group, the algorithm combines, two groups and checks for the pattern in the combined group. Clearly, any implementation of this algorithm will result in $O(n \log n)$ cost (processor-time product) algorithm, as number of items in all groups put together remain n throughout.

In this paper we use the PRAM (Parallel Random Access Machine) model [8]. On the PRAM model memories are shared among all processors. On a CREW (Concurrent Read Exclusive Write) PRAM multiple processors can read a memory cell concurrently in a step but concurrent write to a memory cell by multiple processor is prohibited. On a CRCW (Concurrent Read Concurrent Write) PRAM both concurrent read and concurrent write are allowed. We use the COMMON CRCW PRAM in which when multiple processors are writing into a memory cell in a step they all have to write the same value.

Some useful "tools" are described in Section 2. Algorithms for most cases of length 4 patterns are covered in Section 3. In Section 4, we describe algorithms for the case $1, 3, 2, 4$.

2 Preliminaries

We give the definition for range minima and nearest largers problems. We will use routines for range minima and for nearest largers as black boxes.

For the *range minima* problem, we are given an array $A[1 : n]$, which we preprocess to answer queries of the form:

> Given two integers i, j with $1 \leq i \leq j \leq n$ find the smallest item in sub-array $A[i : j]$.

Range minima queries take $O(1)$ time after preprocessing. The preprocessing can be done in $O(n)$ time in sequential case and in $O(\log n)$ time with $\frac{n}{\log n}$ processors on the CREW model and in $O(\log \log n)$ time with $O(\frac{n}{\log \log n})$ processors on the CRCW model [2, 3]. Range maxima can also be solved in this way.

In the right *nearest largers problem* [3],

> for each item i of array $A[1 : n]$, we find $j > i$, closest to i, such that $A[j] > A[i]$ (thus items, $A[i+1], A[i+2], \cdots, A[j-2], A[j-1]$ are all smaller than $A[i]$). Or, $j = \min\{k | A[k] > A[i] \text{ and } k > i\}$.

All right nearest largers can also be found in $O(n)$ time in sequential case and in $O(\log \log \log n)$ time on a CRCW PRAM and $O(\log n)$ time on the CREW PRAM [3] with $O(n)$ operations (processor time product), or alternatively in $O(1)$ with $O(n \log^3 n)$ processors on a CRCW

PRAM [2, 3] in parallel case. The left nearest larger, the left nearest smaller, and the right nearest smaller can also be solved in this way.

Let us assume that the permutation is given in array $P[1:n]$. Thus, if the ith item of the permutation is k, then $P[i] = k$. As P is a permutation, all items of P are distinct. Hence, if $P[i] = k$ then we can define the *inverse mapping*

$$Position[k] = i$$

Thus, item $Y = P(y)$ will be to the right of item $X = P(x)$ in array P, if and only if, $y > x$ or equivalently $Position(Y) > Position(X)$. And item $Z = P(z)$ will be between X and Z if $Position(Z)$ is between (in value) $Position(X)$ and $Position(Y)$, i.e., $Position(X) < Position(Z) < Position(Y)$, or $Position(Y) < Position(Z) < Position(X)$.

Let r be the nearest right larger of k in $Position$ array.

P-value	k	$k+1$	$k+2$	. . .	r	$r+1$	$\cdots$	n
$Position$					*			

Then, as items with P-value $k+1, ..., r-1$ are smaller than those with P-values k, r is the first (smallest) P-value item larger than k and to its right. Moreover, the nearest smaller of k in the $Position$ array to the right is the first (smallest) P-value item larger than k and to its left.

Similarly, s the nearest larger of k in the $Position$ array on the left, is the first (largest) P-value item smaller than k and to its right. And nearest smaller of k in $Position$ array to its left, is the first (largest) P-value item smaller than k and to its left.

Thus we have the following result:

Theorem 1 *If we know $P[i] = k$, then we can find items closest in values (both larger and smaller than k) on either side of position i using nearest smallers or largers on the Position array.* ∎

As all right nearest largers can also be found in $O(n)$ time in sequential case and in $O(\log\log\log n)$ time on a CRCW PRAM and $O(\log n)$ time on the CREW PRAM [3] with $O(n)$ operations (processor time product), or alternatively in $O(1)$ with $O(n\log^3 n)$ processors on a CRCW PRAM [2, 3] in parallel case, we also have the following corollary:

Corollary 1 *After preprocessing, if $P[i] = k$, then we can find items closest in values (both larger and smaller than k) on either side of position i in $O(1)$ time.*

The preprocessing time will be:

1. *$O(n)$ time in sequential case.*
2. *$O(\log n)$ time with $n/\log n$ processors on the CREW model, or alternatively*
3. *$O(\log\log\log n)$ time with $n/\log\log\log n$ processors on a CRCW PRAM, or alternatively*

4. $O(1)$ *time with* $n \log^3 n$ *processors on a CRCW PRAM.* ∎

Let us assume that $P[i] = k$ and $P[j] = l$ with $k < l$.

P-value	k	$k+1$	$k+2$	$\cdots$	l	$l+1$	$\cdots$
Position	i				j		

If x is the $RangeMinimaPosition(k, l)$ on *Position*-array, then x is the smallest (or the leftmost) *Position*-value between $Position(k)$ and $Position(l)$ of items with P-value between k and l.

Similarly, if y is the $RangeMaximaPosition(k, l)$ on *Position*-array, then y is the largest (or the rightmost) *Position*-value between $Position(k)$ and $Position(l)$ of items with P-value between k and l.

Thus we have the following theorem:

Theorem 2 *Given any* $P[i] = k$ *and* $P[j] = l$*, we can find the leftmost and the rightmost items with* P*-values between* $P[i]$ *and* $P[j]$ *using range maxima or range minima queries.* ∎

As range minima queries take $O(1)$ time after preprocessing and as preprocessing can be done in $O(n)$ time in sequential case and in $O(\log n)$ time with $\frac{n}{\log n}$ processors on the CREW model and in $O(\log \log n)$ time with $O(\frac{n}{\log \log n})$ processors on the CRCW model [2, 3], we also get the following corollary:

Corollary 2 *If* $P[i] = k$*, and* $P[j] = l$*, then we can find the leftmost and the rightmost items with* P*-values between* k *and* l *in* $O(1)$ *time after preprocessing.*

The preprocessing time will be:

1. *The preprocessing time is* $O(n)$ *in sequential case.*
2. $O(\log n)$ *time with* $n/\log n$ *processors on the CREW model, or alternatively*
3. $O(\log \log \log n)$ *time with* $n/\log \log \log n$ *processors on a CRCW PRAM, or alternatively*
4. $O(1)$ *time with* $O(n \log^3 n)$ *processors on a CRCW PRAM.* ∎

3 Length 4 Permutations except 1324

For length 4 sequences, there will be 24 permutations, but 12 of these will be reverse (i.e., $P_0[i] = P[n - i + 1]$) of some other. Further, 4 out of these 12 will be "complement" (i.e., $P[i] = n - P[i] + 1$), thus in all 8 permutations will be left [1]:

2413 (and its reverse 3142) is the case for testing the separability [7, 6] and linear time algorithm for only these are known [5], the parallel algorithm is given later in Section 3.1. In the Section 4 we will consider the case 1324. First in Section 4.1, a sequential linear time algorithm for the case 1324 (and its reverse 4231) is discussed then in the Section 4.2 the corresponding parallel algorithm is described. Here we sketch how to deal with other permutations. As techniques for these permutations are similar, the description will be a bit brief.

Depending on the case, we try to see if i can be chosen as a "2" or a "3". We will abbreviate this to just "fix 2" (i.e, $i = i_2$) or "fix 3" (i.e., $i = i_3$). We finally get tuple (i_1, i_2, i_3, i_4), as a "witness" for the presence of the permutation if i_1, i_2, i_3, i_4 are in the same relative order as $P[i_1], P[i_2], P[i_3], P[i_4]$ i.e., $P[i_\alpha] < P[i_\beta]$, if and only if, $i_\alpha < i_\beta$. Again, a flag can be set if we have a witness and reset otherwise. Finally a logical "or" will give the answer.

In some of the cases, we have to search for an item (usually 3) which has a still larger item (which can be then chosen as 4). For this to be done efficiently, we define a new array $R[1:n]$ with the elements

$$R[i] = \begin{cases} P[i] & \text{if } i \text{ has a larger item to its right} \\ 0 & \text{otherwise} \end{cases}$$

By using right nearest largers, we can easily identify items which have a larger item to their right. Note that in R each nonzero element has a larger element to its right in P. Let us preprocess array R for range maxima queries.

The technique for various patterns is:

1234 Fix 2. 1, the smallest item to its left, is $i_1 = RangeMinimaP(1, i)$. Item 3 can be obtained by range maxima on array R,
$i_3 = RangeMaximaR(i, n)$. Finally, $i_4 = RangeMaximaP(i_3, n)$.

2134 Fix 2. Index i_1 of 1, the first item less than 2, can be found from right nearest smaller of i_2. And again 3 can be obtained by range maxima on array R, $i_3 = RangeMaximaR(i_1, n)$. Finally,
$i_4 = RangeMaximaP(i_3, n)$.

2341 Fix 3. Index i_4 of 4, the first item more than 3, can be found from right nearest larger of i_3. $i_1 = RangeMinimaP(i_4, n)$ will choose 1 as the smallest item on the right of 4. And we use Corollary 1, to find i_2, the index of 2 as the item on left of i, just smaller than $P[i_3]$.

2314 Fix 3. Again we use Corollary 1, to find i_2, the index of 2 as the item on left of i_3, just smaller than $P[i_3]$. We use Corollary 2, to find i_4, the index of 4 as the rightmost item larger than $P[i_3]$. Finally, $i_1 = RangeMinimaP(i_3, i_4)$.

3412 Fix 4. 3 can be found as the largest element smaller than 4 on the left side of 4 using Corollary 1. We create an array of right near larger. For each element e in P that does not have another element f pointing to it (f's right near larger be e) we will change the value of e to max and we will call this array R_1. We then use $RangeMinimaR_1(i_4, n)$ to find 2. 1 would be the closest element on the left of 2 that uses 2 as its right near larger.

2413 This is reverse of pattern 3142. This is the test of separability of permutation. Linear time serial algorithm is known for this case [5]. Parallel Algorithms are discussed in later in this section (see Section 3.1).

1342 Fix 3. 4 is right nearest larger. 1 can be found by $RangeMinimaP(1, i_3)$. Now in the *Position* array use $RangeMaximaPosition(P[i_1], P[i_3])$ to find the position i_2 (using corollary 2).

Thus, we have the following theorem:

Theorem 3 *Given any length* 4 *permutation (other than* 1324 *and its reverse* 4231*), we can test whether it is present (involved) in another permutation of length* n *in* $\Theta(n)$ *sequential time.*

And in parallel case, given any length 4 *permutation (other than* $3, 1, 4, 2$ *and its reverse* $2, 4, 1, 3$*, and* $1, 3, 2, 4$ *and its reverse* $4, 2, 3, 1$*), we can test whether it is present (involved) in another permutation of length* n *in* $O(\log n)$ *time and* $O(n)$ *operations on the CREW PRAM or in* $O(\log\log\log n)$ *time and* $O(n)$ *operations on the CRCW PRAM.* ∎

3.1 Parallel Algorithm for Permutation 3142 (Non-Separable Permutation)

We use a property of separable permutations to get an extremely simple optimal algorithm to check whether the given permutation is separable or not. The simplified algorithm uses just two calls to range minima (or maxima) and a call to nearest largers. If the permutation P is not separable, the permutation, will contain four numbers corresponding to either $3, 1, 4, 2$ or corresponding to $2, 4, 1, 3$. We only test the permutation for the case of $3, 1, 4, 2$, the other case $2, 4, 1, 3$ can be treated similarly (just reverse the pattern).

Let us assume that the pattern is not separable, and $P[i]$, the ith item of P corresponds to 1. Then, r, the item corresponding to 2 must be on the right of $P[i]$ (i.e., $r > i$) and should be larger than $P[i]$ (i.e., $P[r] > P[i]$). We will show that it is sufficient to test the smallest such item $P[r]$. In other words we want the smallest item $P[r] > P[i]$, with index $r > i$. We will fix this item to 2. Rest is routine. We can take the largest item between 1 and 2 as the item corresponding to 4. And finally, the smallest indexed item (the first item from beginning) whose value is between 2 and 4 as 3, if this item is to the left of 1.

We first prove, that the smallest item, larger than the item corresponding to 1 and to its right can be taken as 2.

Lemma 1 *Assume that the permutation P is not separable and contains the pattern $3, 1, 4, 2$. Then there is an index i such that the item $P[i]$ corresponds to 1, and we can get a witness to non-separability of P by fixing item*

$$P[r] = \min\{P[j] | P[j] > P[i] \text{ and } j > i\}$$

to be the item corresponding to 2.

Proof: Basically, we consider that choice in which 1 is closest to 4 in P. For this choice, all items between 1 and 4 will be larger than the item corresponding to 2; otherwise we can take any item between them as 1 and decrease the distance between 1 and 4. Thus the item to the right of 1 which is just larger than 1 (in value) must be to the right of position 4.

In more detail, let us assume that the permutation contains $3, 1, 4, 2$ and hence is not separable. Then there must be a 4-tuple of indices (i_1, i_2, i_3, i_4) such that $i_3 < i_1 < i_4 < i_2$ and $P[i_1] < P[i_2] < P[i_3] < P[i_4]$ (i.e., the permutation $3, 1, 4, 2$ is present).

In case there are several such 4-tuples, first fix i_4 and pick a tuple in which i_1 and i_4 are closest in P (i.e., $(i_4 - i_1)$ is minimum). Then all items between i_1 and i_4 will be larger than $P[i_2]$, otherwise we can take any item between them as i_1 and decrease $(i_4 - i_1)$, the distance between i_1 and i_4.

Thus item r to the right of i_1 (i.e., with index more than i_1) which is just larger than $P[i_1]$ (in value) must be to the right of the position i_4. Hence, as $P[i_1] < P[r] \leq P[i_2] < P[i_3] < P[i_4]$ and $i_1 < i_4 < r$, the 4-tuple $(i - 1, r, i_3, i_4)$ is a witness to presence of $3, 1, 4, 2$.

The lemma follows by choosing $i = i_1$. ∎

As we do not know i_1 of the lemma, we will try all position of permutation P in parallel.

In more detail, we assume that the permutation which we have to test appears in array P. Thus, if i-th item of the pattern is k, then $P[i] = k$. As P is a permutation, all items of P are distinct. To find the location of character k (see Section 2), we create a new array *Position* such that if $P[i] = k$ then $Position[k] = i$. This will give the index or location of each number in P [14].

We preprocess arrays P and *Position* for the range maxima and range minima queries respectively. We also find nearest right largers on the array P.

We will assume that we have n processors, and one processor is assigned to each item of $P[i]$. The following algorithm is executed for each index i in parallel, to test whether the i-th item of P, i.e. $P[i]$ can be 1 (in our witness). The algorithm will take $O(1)$ parallel time.

For each $i \in 1, 2, \cdots, n$ in parallel do:

1. /* Use Theorem 1 to find a "2" after i (recall $P[i]$ is "1") just larger than $P[i]$ */
   If $P[i] = k_i$, then let the nearest right larger of k_i on *Position* be r_i.

   REMARK $P[r_i]$ will be the item after i in P which is just larger than $P[i]$; i.e., $P[r_i]$ will be the smallest item larger than $P[i]$ among all items with index more than i. We will assume that $P[r_i]$ is "2", and see if we can find a suitable "3" and a "4".

2. /* Take the largest item between "1" and "2" as "4" */
   Let $P[t_i]$ be the range maxima on $P[i : r_i]$.

   REMARK $P[t_i]$ will the largest item between items which we have assumed are "1" and "2". We will take item $P[t_i]$ as "4".

3. /* Use Theorem 2 to check if "3" is present */
   Let s_i be the range minima on $Position[P[r_i] : P[t_i]]$.

   REMARK s_i will be the index of the first item $P[s_i]$ which is between 2 and 4 in value. If this item is to the left of "1" then it is a "3" and the permutation is not separable.

4. If $s_i < i$, the permutation is not separable, so we will set Flag[i] =true, otherwise we will set Flag[i] =false.

   REMARK As $s_i < i < t_i < r_i$, tuple $(P[s], P[i], P[t], P[r])$ is a witness to presence of $(3, 1, 4, 2)$.

Finally we will compute logical "or" of bits in the Flag[ ] array in parallel. On a CRCW PRAM, this will take $O(1)$ time with n processors.

We can test for $2, 4, 1, 3$ either in a similar manner, or by executing the algorithm again on an array P' which is reverse of our array P (i.e., $P'[i] = P[n - i + 1]$ for each i).

Clearly the preprocessing time for the algorithm is dominated by that of range maxima and nearest largers computation. The algorithm after preprocessing clearly takes $O(1)$ time with n processors. If we have only $p < n$ processors, the time will increase to $O(n/p)$.

From Section 2, as the preprocessing time for both these problems is:

- $O(\log n)$ time with $n/\log n$ processors on the CREW model, or alternatively,
- $O(\log\log\log n)$ time with $n/\log\log\log n$ processors on a CRCW PRAM, or alternatively,
- $O(1)$ time with $n\log^3 n$ processors on a CRCW PRAM.

We have the following theorem:

Theorem 4 *Given a permutation, we can test whether it is separable with the following (time, processor) trade-offs:*

$$(1, n\log^3 n), (\log\log\log n, n/\log\log\log n), (\log n, n/\log n)$$

The first two trade-off are for a CRCW PRAM and last for the CREW PRAM.

Proof: On the CREW PRAM, logical "or" can be computed in $O(\log n)$ time with $n/\log n$ processors, and the algorithm will be "slowed" down by a factor of $O(\log n)$; basically the i-th processor will now be responsible for indices $i\log n$ to $(i+1)\log n - 1$. ∎

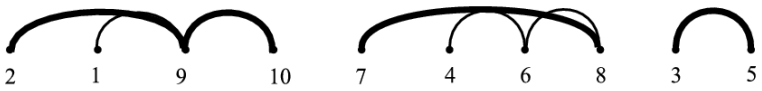


Figure 1: Forest of right Nearest larger. Chains are shown in boldface

4 Algorithms for Permutation 1324

Let us finally consider the remaining case 1324. First consider the following algorithm:

> Fix 3. 1, the smallest item to its left, is $i_1 = RangeMinimaP(1, i)$. We use Corollary 2, to find i_4, the index of 4 as the rightmost item larger than $P[i_3]$.

The only problem is locating a 2. If i_2, the right nearest smaller of 3 is larger than 1, then we have got a "2". However, this method will fail if $P[i_2]$ is a prefix minima (smallest item in $P[1 : i_2]$).

Alternatively, we can proceed as follows:

> Fix 2. 1, the leftmost item smaller than 2, can be found using Corollary 2. i_4, the index of 4 is the largest item right of 2 can be obtained as $i_4 = RangeMaximaP(i, n)$.

This time, the problem is of locating a 3. If i_3, the left nearest larger of 2 is smaller than 4, then we have got a "3". However, this method will fail if $P[i_3]$ is a suffix maxima (largest item in $P[i_2 : n]$).

Thus, in the rest of this section, we assume that these methods do not work.

4.1 Serial Algorithm for Permutation 1324

In array P we use right nearest largers to build a forest. Within each tree of the forest we define the chain of the tree consisting of the root r of the tree, the largest child c_1 of r, the largest child c_2 of c_1, ..., the largest child c_{i+1} of c_i, ..., etc. This is illustrated in Fig. 1.

Our intention is to let roots of trees serve as 4 and nodes on the chains to serve as 3, 1 will be found using range minima and 2 will be served by non-chain nodes.

We traverse a tree this way: when we are at node d, we visit the subtrees of d in the order from the subtree rooted at the smallest child of d to the subtree rooted at the largest child of d. After that we visit d. We start at the root of the tree. This traversal will label the nodes of the tree.

We start from the rightmost tree, visiting nodes in this tree in the order of the above traversal. Let d be a non-chain node we are visiting and let $d, d_1, d_2, \cdots, d_t$ be the path in the tree from d to the nearest ancestor d_t where d_t has another child c larger than d. In this case c will be larger than $d, d_1, d_2, \cdots, d_{t-1}$ and c will be on the left side of $d, d_1, \cdots, d_{t-1}, d_t$ in array P. Thus we can let d_{t-1} serve as 2, c serve as 3, and d_t serve as 4. 1 will be found using $RangeMinimaP(1, i_3)$. If $RangeMinimaP(1, i_3)$ is larger than 2, then we label $d, d_1, \cdots, d_{t-1}$ as they cannot serve as 2. They cannot serve as 1 or 3 because of our traversal order (i.e. they may have tried to serve as 1 or 3 before in the traversal).

Thus after we examined all non-chain nodes, they cannot serve as 1, nor 2, nor 3. They need not serve as 4 because the root of the tree can serve as 4. Thus these non-chain nodes can be removed.

The chain nodes may later serve as 2 but they cannot serve as 1 or 3 because there are no qualifying 2 for them. All of them except the root need not serve as 4 because the root can serve as 4 for them.

Next we examine the second rightmost tree. After we did the same examination for the tree as we did for the rightmost tree only the chain nodes remain to be tested as 2's. However, we have to test the chain nodes in the first tree as 2's using nodes in the second tree as 3's.

In order to do this we have the two lists of nodes each sorted in ascending order. One is the list L_1 of the chain nodes of the first tree and the second list L_2 is the one containing all the tree nodes of the second tree. Let r_1 be the root of the first tree and r_2 be the root of the second tree. We visit these two lists from smallest nodes.

Let a be the current smallest node in L_1 and b be the current smallest node in L_2. If $b < a$ then we remove b from further comparison and get next smallest node from L_2. In this case b is less than all remaining nodes in L_1 and therefore cannot serve as 3, for the remaining nodes in L_1 to serve as 2's. If $r_1 > b > a$ then we let r_1 serve as 4, b serve as 3 and a serve as 2 and use $RangeMinimaP(1, i_3)$ to find 1. If $RangeMinimaP(1, i_3) > 2$ then if a's chain parent p is less than b then p can replace a to serve as 2 and a can be deleted. If $p > b$ then b can be removed. In either case we remove one node. If $r_1 < b$ then we can stop because what remains cannot serve as 3 because there is no 4 for them.

Thus we visit nodes in the second tree at most twice. In the remaining we have the chain for the second tree left and some nodes on the chain for the first tree left. We can merge these nodes from two chains together and form an ascending list. The merging is not done by examining all nodes of the two chains as doing this way will be too costly. We maintain the above two ascending lists in linked lists. Thus once we find $r_1 < b$ for b as a chain node then we insert the remaining nodes in L_1 between b and b's chain child (here insert into the chain of the second tree and not inserting into L_2). Thus the merge takes constant time. Note now that all b's chain descendants are smaller than the remaining nodes in L_1.

Then we view the merged chain node as one chain and we continue working on the third tree from right.

What we have described is the linear time algorithm for the pattern 1324.

Theorem 5 *The pattern* 1324 *can be tested whether it is present (involved) in another permutation of length* n *in* $\Theta(n)$ *time.* ∎

4.2 Parallel Algorithm for Permutation 1324

First each item finds its leftmost smaller. Let $M_{LS} = \{a_1, a_2, a_3, ..., a_t\}$ be the set of items that have no left smallers, where t is some integer and $a_1 < a_2 < \cdots < a_t$.

Let item a be in set S_i if a has a_i as its leftmost smaller. We also put a_i in S_i. We have

Fact 1 *$S_1 < S_2 \cdots < S_t$, where $S_i < S_{i+1}$ means $\max S_i < \min S_{i+1}$.*

We also have

Fact 2 *We can restrict that 1 and 2 be in the same set S_i.*

This is because for any 2 in row i we can always use a_i as 1.

Next each item finds its rightmost larger. Let $M_{RL} = \{b_1, b_2, \cdots, b_s\}$ be the set of items that have no right larger, where s is some integer and $b_1 < b_2 < \cdots < b_s$. Let item a in set T_i if the rightmost larger of a is b_i. We also put b_i in T_i.

Fact 3 *$T_1 < T_2 ... < T_s$.*

Fact 4 *We can restrict that 3 and 4 be in the same set T_i.*

Proof: This is because for any 3 in T_i we can always use b_i as 4. ∎

Fact 5 *For any T_i (S_i), there are at most two S_j's (T_j's): S_k and S_l (T_k and T_l) such that $S_k \cap T_i \neq \phi$ ($T_k \cap S_i \neq \phi$) and $S_l \cap T_i \neq \phi$ ($T_l \cap S_i \neq \phi$) yet $S_k \not\subseteq T_i$ and $S_l \not\subseteq T_i$ ($T_k \not\subseteq S_i$ and $T_l \not\subseteq S_i$).*

Proof: Fact 5 comes from Fact 1 and Fact 3. ∎

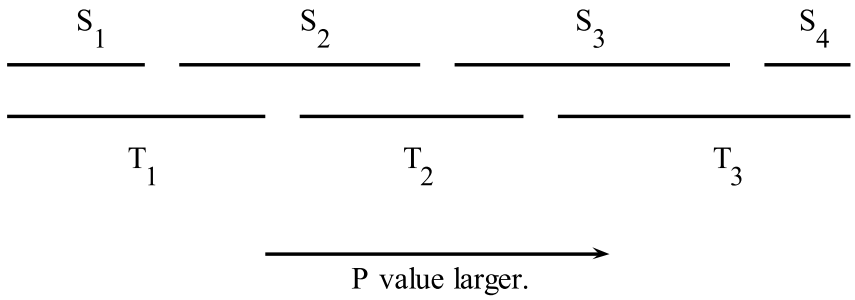
The situation for S_i's and T_j's is shown in Fig. 2a. First let each item a finds the indices i and j such that $a \in S_i$ and $a \in T_j$. Then decide the situation: either $S_i \not\subseteq T_j$ and $T_j \not\subseteq S_i$, or $S_i = T_j$, or $S_i \subset T_j$, or $T_j \subset S_i$. We can then partition S_i's and T_j's such that S_i and T_i becomes identical. This is shown in Fig. 2b.

Note that when we partition S_i into two sets we will have a copy of a_i in both sets and when we partition T_i into two sets we will have b_i in both sets.

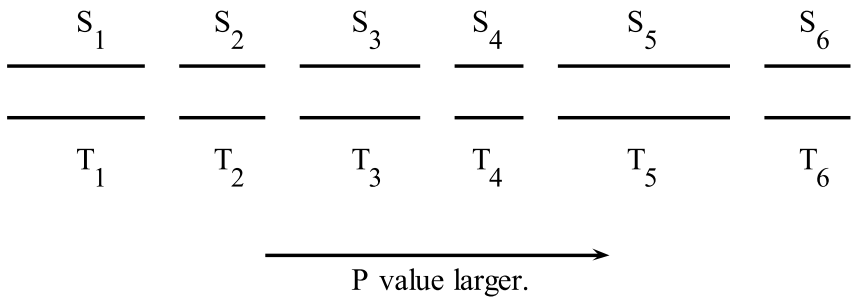
Items $i \in S_j$ and $i \in T_j$ is in row j. a's row will be indicated by $row(a)$. See Fig. 3. for illustration.

An important fact about the *Position* array is that it groups S_i (T_i) together. We will call the $S_i(T_i)$ in *Position* array as PS_i (PT_i). See Fig. 4 for illustration.

Thus for $a = Position[i]$ with $P[a] \in S_j$, the right nearest larger of a in $PS_{j+1}, PS_{j+2}, \cdots$, (note that PS_j is excluded) is the smallest $P[k] \in S_{j+1}$ on the right side of $P[a]$ in array P. The right nearest smaller of a in $PS_{j+1}, PS_{j+2}, \cdots$, is the smallest $P[k] \in S_{j+1}$ on the left side of $P[a]$ in array P. If we reverse every PS_i and write them as RPS_i then the nearest larger of a in $RPS_{j+1}, RPS_{j+2}, \cdots$, is the largest $P[k] \in S_{j+1}$ on the right side of $P[a]$ in array P. The right nearest smaller of a in $RPS_{j+1}, RPS_{j+2}, \cdots$, is the largest $P[k] \in S_{j+1}$ on the left side of $P[a]$ in array P. By the nature of the algorithm of Berkman et al. [2][3] the right nearest larger and right nearest smaller here can be found is constant time with $O(n)$ preprocessing operation (with $O(\log n)$ time and $n/\log n$ processors on the CREW PRAM and $O(\log\log\log n)$ time and $n/\log\log\log n$ processor on a CRCW PRAM), or with constant preprocessing time with $n\log^3 n$ processors on a CRCW PRAM. Note also we can look at the left side of PS_j, namely $PS_1, PS_2, \cdots, PS_{j-1}$ and found corresponding element



(a)



(b)

Figure 2: Situation for S_is and T_js

on $P[k]$ in S_{j-1}. Note that the observation made in this paragraph is a key for us to design efficient parallel algorithms for the 1324 case. We will call the right nearest larger and right nearest smaller here as the right nearest skip larger and right nearest skip smaller because they skip S_j.

First note that the case for both 2 and 3 being in row j can be treated easily. By Fact 2 and Fact 4 we have all $1, 2, 3, 4$ are in row j. Thus we use a_j as 1 and b_j as 4. Now in *Position* array PS_j are grouped together. Say PS_j is in $Position[k]$ to $Position[l]$. We then first remove $Position[a_j]$ and $Position[b_j]$ and then compare $Position[v]$ with $Position[v+1]$, $k \leq v < l$. If $position[v] > Position[v+1]$ then use $v+1$ as 3 and v as 2.

Second consider the case that the *Position* values for both 3 and 2 are smaller than $Position[b_t]$, where t is the largest index for b_i, as shown in Fig. 5. In this case we fix a as 2, say that $a \in S_i$, then use range maxima between $Position[a_i]$ and $Position[a]$ as 3, use a_i as 1 and b_t as 4.

Third consider the case that the *Position* values for both 3 and 2 are larger than $Position[a_1]$, as shown in Fig. 6. In this case we fix a as 3, say that $a \in S_i$(i.e. $a \in T_i$ because S_i and T_i are identical). Use range minima between $Position[a]$ and $Position[b_i]$ as 2, use a_1 as 1 and b_i as 4.

Thus we left with the case that *Position* value for 3 is smaller than $Position[a_1]$ and

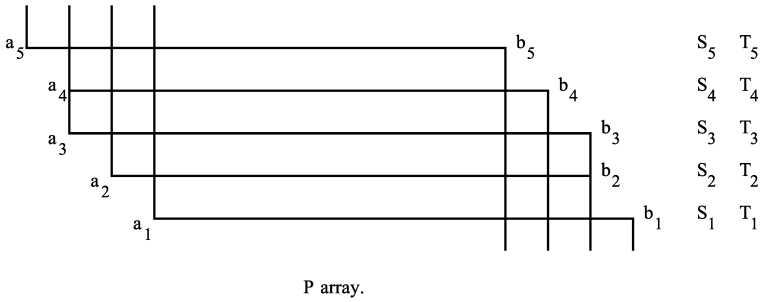


Figure 3: Illustration for a's row $row(a)$

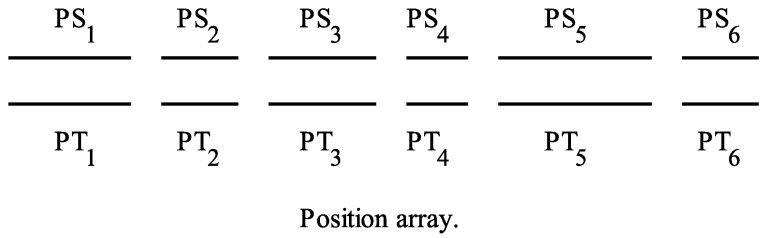


Figure 4: Position array PS

$Position$ value for 2 is larger than $Position[b_t]$ and this is the fourth case. We use the $Position$ array as $PS_1, PS_2, \cdots$. We change the value $Position[i]$ to 0 (minimum) if $Position[i] > Position[b_t]$ and call the changed $Position$ array as $CPosition$ array and each PS_i will now become CPS_i. Now fix a with $Position[a] > Position[b_t]$ as 2, say $a \in S_i$, then find the right nearest skip larger b of $CPosition[a_i]$ in the $CPosition$ array. b can now serve as 3 because $Position[b] < Position[b_t]$ since we set $Position[i]$ to 0 (minimum) if $Position[i] > Position[b_t]$ in $CPosition$.

Thus we have that

Theorem 6 *All length four permutations can be tested in $O(\log\log\log n)$ time with $n/\log\log\log n$ processors or in constant time with $n\log^3 n$ processors on a CRCW PRAM, or in $O(\log n)$ time with $n/\log n$ processors on the CREW PRAM.* ∎

5 Concluding Remarks

We have described $\Theta(n)$ time (sequential) algorithms for testing involvement of all length 4 patterns. Previously most of these patterns had only $O(n\log n)$ time algorithms.

We have also described an $O(\log n)$ time and $n/\log n$ processor CREW PRAM algorithm and an $O(\log\log\log n)$ time and $n/\log\log\log n$ processor and constant time and $n\log^3 n$ processor CRCW PRAM algorithm for testing involvement of all length 4 patterns. Thus parallel computations for length 4 permutations are now basically solved.

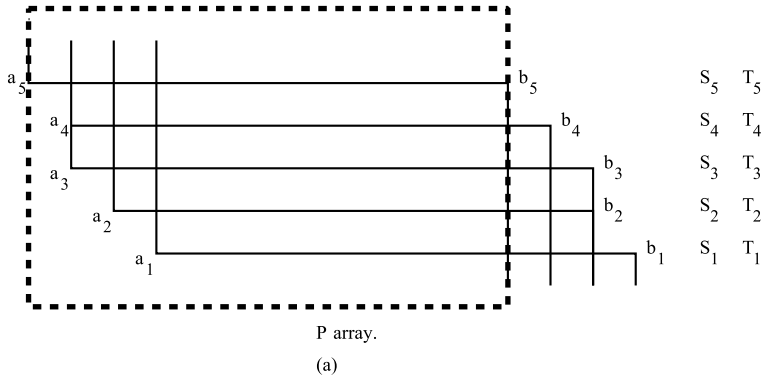


Figure 5: Position values for 3 and 2 smaller than $Position[b_t]$

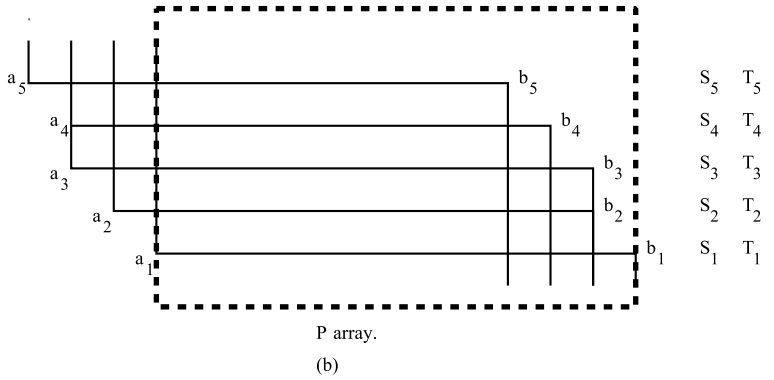


Figure 6: Position values for 3 and 2 larger than $Position[a_1]$

References

[1] Albert M.H., Aldred R.E.L., Atkinson M.D., and Holton, D.A. Algorithms for Pattern Involvement in Permutations. Proc. ISAAC 2001, Springer Lecture Notes Computer Sc. vol. 2223 (2001), 355–367.

[2] Berkman, O., Matias, Y., and Ragde, P. Triply-logarithmic parallel upper and lower bounds for minimum and range minima over small domains. Journal of Algorithms 28 (August 1998), 197–215.

[3] Berkman, O., Schieber, B., and Vishkin, U. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. Journal of Algorithms 14 (May 1993), 344–370.

[4] Bose, P., Buss, J. F., and Lubiw, A. Pattern matching for permutations. Information Processing Letters 65 (1998), 277–283.

[5] Fredman, M.L. On Computing the length of longest increasing subsequences. Discrete Mathematics 11 (1975), 29–35.

[6] Y. Han, S. Saxena, X. Shen. An efficient parallel algorithm for building the separating tree. Journal of Parallel and Distributed Computing, Vol. 70, Issue 6, 625-629(2010).

[7] Ibarra, L. Finding pattern matchings for permutations. Information Processing Letters 61 (1997), 293–295.

[8] R. Karp, V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD 88/408, Computer Science Division (EECS), University of California, Berkeley, 1988.

[9] Kitaev, S. and Mansour, T. A survey on certain pattern problems. Available at http://www.ru.is/kennarar/sergey/index_files/Papers/survey.ps

[10] Kitaev, S. Patterns in permutations and words. Springer-Verlag, 2011

[11] Preparata, F.P. and Shamos, M.I. Computational Geometry, An Introduction. Springer-Verlag, 1985

[12] Saxena, S. Dominance made simple. Information Processing Letters 109 (2009), 419–421.

[13] Tamassia, R. and Vitter, J.S. Optimal cooperative search in fractional cascaded data structures. Algorithmica 15 (1996), 154–171.

[14] Yugandhar, V., and Saxena, S. Parallel algorithms for separable permutations. Discrete Applied Mathematics 146 (2005), 343–364.