Information Processing Letters 113 (2013) 470–476

Contents lists available at SciVerse ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

A divide and conquer approach and a work-optimal parallel algorithm for the LIS problem

Muhammad Rashed Alam, M. Sohel Rahman *

AℓEDA Group, Department of CSE, BUET, Dhaka-1000, Bangladesh

ARTICLE INFO

Article history:
Received 17 August 2012
Received in revised form 15 March 2013
Accepted 22 March 2013
Available online 26 March 2013
Communicated by F.Y.L. Chin

Keywords:
Algorithms
Parallel algorithms
Longest increasing subsequence
Divide and conquer

ABSTRACT

In this paper, we present a divide and conquer approach to solve the problem of computing a longest increasing subsequence. Our approach runs in $O(n \log n)$ time and hence is optimal in the comparison model. In the sequel, we show how we can achieve a work-optimal parallel algorithm using our divide and conquer approach.

1. Introduction

Given a sequence $S = S[1], S[2], \ldots, S[n]$, we get a subsequence by deleting 0 or more symbols from S, keeping the order of the symbols in S intact. A sequence S is said to be increasing if we have $S[i+1] > S[i]$ for all $1 \leqslant i < n$. Let $\pi = \pi(1), \pi(2), \ldots, \pi(n)$ be a permutation of $[1 \ldots n]$ and we are given a sequence $S = S[1], S[2], \ldots, S[n] = \pi(1), \pi(2), \ldots, \pi(n)$, i.e., S is a permutation of $[1 \ldots n]$. The Longest Increasing Subsequence (LIS) problem aims to compute an increasing subsequence (IS) S' from S such that $|S'|$ is maximum.

The LIS problem is related to a more studied problem of computing a longest common subsequence (LCS) of two strings, and to their alignment, in at least two ways. Firstly, it is easy to realize that, the LIS of S is the LCS between S and the sequence representing the identity permutation, i.e., $1, 2, \ldots, n$. This leads to a straightforward $O(n^2)$ time algorithm implementing the standard dynamic programming technique used for computing a longest common subsequence [25] (it can indeed be reduced to $O(n^2/\log n)$ [17,6]). Notably, the LCS computation algorithm of Hunt and Szymanski [14] reduces to an $O(n \log n)$ algorithm for computing LIS under the above setting. Secondly, the LIS question is involved in the solution to the problem of whole-genome comparison proposed by Delcher et al. [8] and in its subsequent variants. Such a comparison is based on maximal exact matches between the two input genome sequences, matches that are additionally constrained to occur only once in each sequence. An LIS is used to extract a long subsequence of matches that are compatible between each other, i.e., they appear in the same order along the two sequences, for producing an alignment of the complete genomes.

The question is also related to the representation of permutations, elements of the symmetric group on $\{1, 2, \ldots, n\}$, with Young tableaux. This is certainly why it has attracted a lot of attention. The readers are referred to [2] for a presentation of Schensted's algorithm [21] in this context.

In parallel to using the LCS algorithms to solve the LIS problem, direct algorithms to solve the problem are also available in the literature. Fredman [10] devised an algorithm running in $O(n \log n)$ time. This solution is clearly optimal if the elements are drawn from an arbitrary

* Corresponding author.
E-mail addresses: rashed.muhammad@yahoo.com (M.R. Alam), msrahman@cse.buet.ac.bd (M.S. Rahman).

0020-0190/$ – see front matter
http://dx.doi.org/10.1016/j.ipl.2013.03.013

set [10]. Parameterized by the LIS length k, the running time becomes $O(n \log k)$. On integer alphabets, the fastest known solution runs in $O(n \log\log n)$ time [26] which relies on a complex priority search tree of van Emde Boas [24]. Very recently, Crochemore and Porat [7] presented an $O(n \log\log k)$ time algorithm for the problem assuming a RAM model. This result improves a 30-year bound of $O(n \log k)$. The algorithm also improves on the previous $O(n \log\log n)$ bound. The question of optimality of the new bound is still open [7]. Note that the algorithm of Crochemore and Porat [7] assumes a permutation of $[1..n]$ as input.

A few parallel algorithms also have been proposed for the LIS problem in the literature. A generic approach is to reduce the problem to computing the longest common subsequence (LCS) of two strings of length n. For example in [12] the authors presented one such approach having cost $O(n^2/p)$ on p processors. On the EREW PRAM model with p processors, Nakashima and Fujiwara [18,19] presented two algorithms with $O(m(\frac{n}{p} + \log n))$ and $O(\log n + \frac{n \log n}{p} + m^2 \log \frac{n}{p} + m \log p)$ time,[1] respectively. Semé [22] gave a CGM algorithm that runs in $O(n \log(n/p))$ time. Krusche and Tiskin [15] have also given a parallel algorithm obtaining a computational cost of $O(n^{1.5}/p)$ in BSP model [23].

In this paper, we take a different approach to solve the LIS problem. In particular we attack the problem using a divide and conquer approach. Using our approach we are able to devise a novel algorithm to solve LIS that also runs in $O(n \log n)$ time. In the sequel, we show how our approach provides us with a parallel work optimal algorithm considering the comparison model. The contribution of this paper is as follows. Firstly, since many multithreaded algorithms involving nested parallelism follow naturally from the divide-and-conquer paradigm, our approach opens a new and hitherto unexplored avenue to get direct multiprocessor solutions for the LIS problem. And indeed the parallel algorithm devised in this paper based on the serial divide and conquer algorithm presented outperforms all the parallel algorithms for LIS in the literature. Secondly, all the sequential algorithms for the LIS problem in the literature are online. As a result, being offline, our approach may turn out to be at least theoretically interesting and may present many enthusiastic researchers with some new ideas to devise even more efficient offline algorithms.

The rest of the paper is organized as follows. In Section 2 we will recall the basic well-known algorithm for solving the LIS problem. In Section 3 we present our divide and conquer approach to solve the problem. In Section 4 we discuss the parallel algorithm and a brief comparison with other parallel algorithms. Finally we conclude in Section 5.

2. Basic algorithm

In our divide and conquer approach we make use of the basic algorithm, referred to as BAlg henceforth, for computing an LIS. For the sake of completeness, in this section we briefly discuss how BAlg works. In BAlg, the elements are processed in the order $\pi(1), \pi(2), \ldots, \pi(n)$. Conceptually, we compute for each length $\ell = 1, 2, \ldots$, the smallest last element that can end an increasing subsequence of that length. It is called the *best element* for that length and denoted by $B[\ell]$. Note that best elements $B[1], B[2], \ldots, B[\ell]$ form an increasing sequence. This fact is used for the choice of a data structure to implement the list and is essential for efficient computation.

BAlg works as follows. Consider the ith iteration where $1 \leqslant i \leqslant n$. Element $\pi(i)$ can extend any increasing subsequence ending at an element of B (say, $B[j]$) such that $B[j]$ is smaller than $\pi(i)$. Suppose, up to now, i.e., for $\pi(1), \pi(2), \ldots, \pi(i-1)$, we have computed $B[1] \ldots B[\ell]$. If $\pi(i) > B[\ell]$, then we must also have $\pi(i) > B[i]$, $1 \leqslant i \leqslant \ell$. In this case, $\pi(i)$ can produce an IS longer than any previous one. So, we set $B[\ell + 1] = \pi(i)$. Otherwise, $\pi(i)$ becomes the best element for an existing length: it replaces the smallest element greater than $\pi(i)$, i.e., the *successor* of $\pi(i)$ in B. In both cases, we set the parent of $\pi(i)$, namely $P(i)$, to the position of largest element smaller than $\pi(i)$, i.e., its *predecessor* in B. We can find the original LIS by traversing the P-array backward. Notably, $B[0]$ is set to 0.

The runtime analysis of BAlg is straightforward. At the ith iteration the index of the successor/predecessor of $\pi(i)$ can be found using binary search in $O(\log n)$ time. Hence the $O(n \log n)$ running time of BAlg follows readily.

3. A divide and conquer approach

In this section, we discuss our divide and conquer approach. Without the loss of generality we can assume n to be even. In order to compute the LIS of S, we divide S into two subsequences S_1 and S_2 of equal length $n/2$. Now we solve the LIS problem for S_1 and S_2. In other words, using BAlg, we compute B_k and P_k, to compute the LIS of S_k, where $k \in \{1, 2\}$. Then we compute the B and P arrays for S using B_k, P_k, $k \in \{1, 2\}$. Notably, for our divide and conquer approach we will slightly extend the structure of the B array as follows. In particular, we will assume B to be an array of 2-tuple, where each entry of B will have two attributes, namely, *val* and *pos*. Hence, inserting S_i in B at some index k implies that $B[k+1].val = S[i]$ and $B[k+1].pos = i$. However, since S is a permutation, the elements in S_1 and S_2 are distinct and without multiple occurrences. Hence, the parent array P can be global and we don't need two separate parent arrays P_1 and P_2. Our divide and conquer algorithm, referred to as the D&C algorithm henceforth works as follows.

Divide: We divide S in two subsequences S_1 and S_2 as follows. We delete from S the elements that are greater than (less than or equal to) $n/2$ to get S_1 (S_2). In other words, elements that are less than or equal to (greater than) $n/2$ appear in S_1 (S_2).

Conquer: We perform the LIS computation for the two subsequences S_1 and S_2 recursively, i.e., compute B_1, B_2 and the P array (globally) for S_1 and S_2.

[1] Here, m is the number of decreasing subsequences in the solution.

Combine: In this phase, we iterate over the elements of S. In the Combine phase, we will use the function $Insert(x, y)$ to insert the element x in B after the position of y in it. So, if at some iteration i, $x = S[i]$ and $B[j].value = y$, then, executing $Insert(x, y)$ sets $B[j+1].val = x$ and $B[j+1].pos = i$. Clearly, any element previously stored at position $j+1$ is replaced by this operation. The case when $y = 0$ is treated specially as follows. $Insert(x, 0)$ put x at the first position of B (i.e., $B[1].val = x$ and $B[1].pos = i$). Now, at the ith iteration, for an element $x = S[i]$, we do the following. Note that we have B_1, B_2 and the P array at our disposal and our aim now is to compute B and update P. We have the following two cases.

1. **Case 1: $x \leqslant n/2$ (i.e., x is in S_1):** If the parent of x, i.e., $y = S[P[i]]$ is currently present in B, then we execute $Insert(x, y)$. Note that, if $y = 0$ then we execute $Insert(x, 0)$. Notably, if $y \neq 0$ then y will always be present in B (Lemma 1).
2. **Case 2: $x > n/2$ (i.e., x is in S_2):** Similar to the case before, if the parent of x, i.e., $y = S[P[i]]$ is present in B, then we execute $Insert(x, y)$. If, however, $y = 0$ or y is not currently present in B, then we execute $Insert(x, z)$ where z is the largest element of S_1 currently in B. Additionally, we update parent of x, $P[i]$ by setting it to the index of z.

For efficient implementation, we will have to use an array *isPresent* to check the presence of an element y in B as follows. If j is the index of y in S, $isPresent[j] = 0$ if y is not present in B and $isPresent[j] = \ell$ if $B[\ell] = y$.

Now we prove the correctness of the D&C algorithm and deduce its running time. Clearly, the key operation of this algorithm is the *Combine* operation. We start with the following lemma.

Lemma 1. *During the* Combine *operation, suppose that $x = S[i]$, $j = P[i]$, and $y = S[j]$, i.e., y is the parent of x. If $0 < y \leqslant n/2$ then y will always be present in B.*

Proof. Since every element of S_1 is less than or equal $n/2$, an element y of S_1 will be the predecessor of x if and only if y also belongs to S_1 and $y < x$. Thus if y is the predecessor of x during the computation of S_1, then it is also predecessor of x during the computation of LIS of S. Hence the result follows. □

Lemma 2. *The* Combine *operation correctly computes in linear time B and P arrays for S from the LIS computation of the two subsequences S_1 and S_2.*

Proof. Recall that, while computing an LIS, at some iteration i, we always place an element $x = S[i]$ immediately after it's *predecessor* in the queue B. Now we have two cases as discussed below.

Case 1: x belongs to the set S_1. In this case, the predecessor of x must also belong to the set S_1. Clearly, if y is the predecessor of x in the computation of LIS of S_1, then y is also the predecessor of x in the computation of LIS of S. As we have already computed the LIS of S_1, we have already known the predecessor y of x. Hence, we correctly place x immediately after the position of y in B by executing $Insert(x, y)$ as mentioned in Case 1 of the *Combine* operation.

Case 2: x belongs to the set S_2. In this case, the predecessor of X either belongs to the set S_2 or is the largest element of S_1 currently present in B. Now, we have already computed the LIS of S_2. Suppose y' has been found to be the predecessor of x during the computation of LIS of S_2. While computing the LIS of S, we need to check whether y' is currently present in B. If yes, then we correctly execute $Insert(x, y')$ as stated in Case 2 of the *Combine* operation. If on the other hand, y' is not present in B, then the largest element z of the set S_1 that is currently present in B will be predecessor of x and we correctly place x immediately after z executing $Insert(x, z)$ as stated in Case 2 of the *Combine* operation.

Now at each iteration, the only operations are to check the presence of *predecessor* y of x using the array *isPresent*. So, the *Combine* operation runs in $O(n)$ time. □

The D&C algorithm is initiated by calling the function $COMPUTE\text{-}LIS(S, P)$ where S is the input sequence and P is the parent array to be computed. The complete algorithm is formally given below:

COMPUTE-LIS(S, P)

```
1: if |S| > 1 then
2:     n ← |S|
3:     Divide S into S1 and S2 each of which has length n/2.
4:     COMPUTE-LIS(S1, P)
5:     COMPUTE-LIS(S2, P)
6:     COMBINE-LIS(S, P)
7: end if
```

COMBINE-LIS(S, P)

```
1: n ← |S|; max ← 0; max_si ← 0; ℓ ← 0
2: for i ← 1 to n do
3:     x ← S[i]; j ← P[i]; y ← S[j]; k ← isPresent[j]
4:     if k + 1 > ℓ then
5:         ℓ ← ℓ + 1
6:     end if
7:     if x ⩽ n/2 then
8:         isPresent[B[k + 1].pos] ← 0; P[i] ← k
9:         B[k + 1].val ← x; B[k + 1].pos ← i; isPresent[i] ← k + 1
10:        if (max < x) then
11:            max ← x; max_si ← k + 1
12:        end if
13:    else
14:        if k = 0 then
15:            k ← max_si
16:        end if
17:        isPresent[B[k + 1].pos] ← 0; P[i] ← k
18:        B[k + 1].val ← x; B[k + 1].pos ← i; isPresent[i] ← k + 1
19:    end if
20: end for
```

To find the longest increasing subsequence of $S = S[1], S[2], \ldots, S[n]$, we make the initial call *COMPUTE-LIS*(S, P). The algorithm consists in combining pairs of subsequence of length one to compute an LIS of a subsequence of length 2, combining the pairs of subsequence of length 2 to find an LIS of subsequence of length 4, and so on, until two subsequences of length $n/2$ are combined to find the LIS of the original (input) sequence of length n. As the running time of combine operation is $\Theta(n)$, the running time $T(n)$ for the algorithm can be evaluated using the following well known recursive equation:

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n). \quad (1)$$

3.1. An illustrative example

Example 1. We consider the sequence $S = (8, 9, 2, 6, 3, 7, 10, 4, 1, 5)$ where $n = 10$. To clearly identify the positions of the elements of the subsequences in the original sequence, in the rest of the example we denote each element of a subsequence S' of S as a two-tuple $(value, pos)$, where $S'[i].value$ indicate the ith element and $S'[i].pos$ denote its original position in S. Under this convention, a subsequence $S' = 9, 6, 10$ of S may also be denoted as follows:

$$S' = (9, 2), (6, 4), (10, 7).$$

If we compute the LIS of S using the basic algorithm then we will have $B = (1, 9), (3, 5), (4, 8), (5, 10)$ and $P = 0, 1, 0, 3, 3, 5, 6, 5, 0, 8$. In what follows, we say x belongs to a sequence S if we have $S[i] = x$ for some $1 \leqslant i \leqslant |S|$.

Divide: Now we divide the S into S_1 and S_2 as follows:

$$S_1 = (2, 3), (3, 5), (4, 8), (1, 9), (5, 10),$$

$$S_2 = (8, 1), (9, 2), (6, 4), (7, 6), (10, 7).$$

Conquer: After computing the LIS of S_1 and S_2 recursively, we will have $B_1 = (1, 9), (3, 5), (4, 8), (5, 10)$, $B_2 = (6, 4), (7, 6), (10, 7)$ and $P = 0, 1, 0, 0, 3, 4, 6, 5, 0, 8$.

Combine: We iterate over the elements of S, given that B is initially empty.

At iteration 1, $S[1] = 8$ and parent of 8, $P[1] = 0$. We insert 8 into B and hence we have $B = (8, 1)$. Nothing changes in P.

At iteration 2, $S[2] = 9$, $P[2] = 1$ and $S[P[2]] = S[1] = 8$, which is currently present in B. So we insert 9 after 8 in B. Now, $B = (8, 1), (9, 2)$. Nothing changes in P.

At iteration 3, $S[3] = 2$ and $P[3] = 0$. Since 2 belongs to S_1, we insert 2 at the first index of B replacing the previous entry. Now $B = (2, 3)(9, 2)$. Nothing changes in P.

At iteration 4, $S[4] = 6$ and $P[4] = 0$. Now, 6 belongs to S_2 and the largest element of S_1 that is currently present in B is 2. So, we insert 6 after 2 in B and set parents of 6, $P[4] = 3$, which is position of 2 in S. Now $B = (2, 3)(6, 4)$ and $P = 0, 1, 0, \mathbf{3}, 3, 4, 6, 5, 0, 8$.

At iteration 5, $S[5] = 3$, and $P[5] = 3$ which is the position of 2. So we place 3 after 2 in B and hence $B = (2, 3)(3, 5)$. Nothing changes in P.

At iteration 6, $S[6] = 7$, $P[6] = 4$ and $S[4] = 6$, which is currently not present in B. Now, 7 belongs to S_2. So we place 7 after the largest element of S_1 currently present in B, which is 3 and set $P[6]$ to 5, i.e., the position of 3. Now $B = (2, 3)(3, 5), (7, 6)$ and $P = 0, 1, 0, 3, 3, \mathbf{5}, 6, 5, 0, 8$.

At iteration 7, $S[7] = 10$, $P[7] = 6$ and $S[6] = 7$, which is currently present in B. So, we insert 10 after 7 in B. Now $B = (2, 3)(3, 5), (7, 6), (10, 7)$. Nothing changes in P.

At iteration 8, $S[8] = 4$, $P[8] = 5$, and $S[5] = 3$, which is currently present in B. So we insert 4 after 3 in B replacing the previous entry. Now, $B = (2, 3)(3, 5), (4, 8), (10, 7)$. Nothing changes in P.

At iteration 9, $S[9] = 1$, and $P[9] = 0$. Since, 1 belongs to S_1, we insert 1 at the first index of B replacing the previous entry. Now $B = (1, 9)(3, 5), (4, 8), (10, 7)$. Nothing changes in P.

At iteration 10, $S[10] = 5$, $P[10] = 8$ and $S[8] = 4$, which is currently present in B. So we insert 5 after 4 in B replacing the previous entry. Now $B = (1, 9)(3, 5), (4, 8), (5, 10)$. Nothing changes in P.

Thus after the *Combine* operation we have:

$$B = (1, 9)(3, 5), (4, 8), (5, 10)$$

and

$$P = 0, 1, 0, 3, 3, 5, 6, 5, 0, 8.$$

3.2. Non-permutations

So far we have discussed our algorithms considering a permutation $\pi = \pi(1), \pi(2), \ldots, \pi(n)$ of set $[1 \ldots n]$. However our algorithm can be easily adapted to work on a general sequence that is not a permutation of $[1..n]$. To do this, we first select the median of the input sequence using a linear time algorithm and use that median to divide the sequence into two subsequences, each of length $n/2$. So our slightly revised divide and conquer algorithm will be as follows:

COMPUTE-LIS-General(S, P**)**

```
1: if |S| > 1 then
2:     n ← |S|
3:     x = MEDIAN(S)
4:     Divide S based on x into S1 and S2 such that |S1| =
       |S2| = n/2.
5:     COMPUTE-LIS-General(S1, P)
6:     COMPUTE-LIS-General(S2, P)
7:     COMBINE-LIS(S, P)
8: end if
```

We conclude this section with another illustrative example that considers a general sequence, i.e., non-permutations and show how *COMPUTE-LIS-General*(S, P) works.

Example 2. We consider the sequence $S = 51, 62, 34, 56, 42$, where $n = 5$. Like the example in Section 3.1, here as well we follow the following convention. To clearly identify the positions of the elements of the subsequences in the original sequence, we denote each element of a

subsequence S' of S as a two-tuple $(value, pos)$, where $S'[i].value$ indicate the ith element and $S'[i].pos$ denote its original position in S. Now, if we compute the LIS of S using the basic algorithm then we get, $B = (34, 42)$ and $P = 0, 1, 0, 3, 3$.

Divide: We first compute the median of S, which is 51. we divide the queue S into two subsequences $S_1 = (51, 1)$, $(34, 3)$, $(42, 5)$ and $S_2 = (62, 2)$, $(56, 4)$.

Conquer: After computing the LIS of S_1 and S_2 recursively, we get $B_1 = (34, 3), (42, 5)$, $B_2 = (56, 4)$ and $P = (0, 0, 0, 0, 3)$.

Combine: Now let us iterate over the elements of S, given that B is initially empty.

At iteration 1, $S[1] = 51$ and $P[1] = 0$. So we inset 51 into B and we get $B = (51, 1)$. Nothing changes in P.

At iteration 2, $S[2] = 62$ and $P[2] = 0$. Now, 62 belongs to the set S_2. Since, the largest element of S_1 currently present in B is 51, we insert 62 after 51 in B. We further set 51 to be the parent of 62 by setting $P[2] = 1$. Now $B = (51, 1), (62, 2)$ and $P = 0, \mathbf{1}, 0, 0, 3$.

At iteration 3, $S[3] = 34$ and $P[3] = 0$. Since, 34 belongs to the set S_1, we insert 34 at the first index of B replacing the previous entry. Now $B = (34, 3), (62, 2)$. Nothing changes in P.

At iteration 4, $S[4] = 56$ and $P[4] = 0$. Now, 56 belongs to the set S_2. Since the largest element of S_1 currently present in B is 31, we insert 56 after 31 in B replacing the previous entry. We further set 34 as the parent of 56 by setting $P[4] = 3$. Now $B = (34, 3), (56, 4)$ and $P = (0, 1, 0, \mathbf{3}, 3)$.

At iteration 5, $S[5] = 42$, $P[5] = 3$ and $S[3] = 34$. So, we place 42 after 34 in B replacing the previous entry. So, we get $B = (34, 3)(42, 5)$. Nothing changes in P.

Thus after the *Combine* operation $B = (34, 3)(42, 5)$ and $P = (0, 1, 0, 3, 3)$.

4. Dynamic multithreaded LIS

Many multithreaded algorithms involving nested parallelism follow naturally from the divide-and-conquer paradigm. In this section, we will adopt our D&C algorithm to work in a multithreaded environment. We will use the multithreading model discussed in [5]. Although we mainly focus on *COMPUTE-LIS*(S, P), the same strategy applies for *COMPUTE-LIS-General*(S, P) (of Section 3.2) as well. The following discussion briefly presents the model and is borrowed from [13,5].

Dynamic-multithreading supports two features: nested parallelism and parallel loops. Nested parallelism allows a subroutine to be *spawned* allowing the caller to proceed while the spawned subroutine is computing its result. This model is a simple extension of our serial programming model. We can describe a multithreaded algorithm by adding to our pseudocode just three *concurrency* keywords, namely, *parallel*, *spawn*, and *sync*. Moreover, if we delete these concurrency keywords from the multithreaded pseudocode, the resulting text becomes a serial pseudocode for the same problem. This process is referred to as the *serialization* of the multithreaded algorithm. It provides a theoretically clean way to quantify parallelism based on the notions of *work* and *span*. A growing number of concurrency platforms support one variant or another of dynamic multithreading, including Cilk [3,11], Cilk++ [1], OpenMP [4], Task Parallel Library [16], and Threading Building Blocks [20].

Below, we augment the pseudocode of our serial D&C algorithm to make it a multithreaded algorithm (referred to as Mult-D&C henceforth) by adding the concurrency keywords **spawn** and **sync**.

PCOMPUTE-LIS(S, P)

```
1: if |S| > 1 then
2:     n ← |S|
3:     Divide S into S1 and S2 each of which has length
   n/2.
4:     spawn PCOMPUTE-LIS(S1, P)
5:     spawn PCOMPUTE-LIS(S2, P)
6:     sync
7:     COMBINE-LIS(S, P)
8: end if
```

Nested parallelism occurs when the keyword spawn precedes a procedure call, as in Lines 4 and 5. The semantics of a spawn differs from an ordinary procedure call in that the procedure instance that executes the spawn (the parent) may continue to execute in parallel with the spawned subroutine (its child) instead of waiting for the child to complete, as would normally happen in a serial execution. The keyword *spawn* does not say, however, that a procedure must execute concurrently with its spawned children, only that it may. The concurrency keywords express the logical parallelism of the computation, indicating which parts of the computation may proceed in parallel. At runtime, it is up to a scheduler to determine which subcomputations actually run concurrently by assigning them to available processors as the computation unfolds. A procedure cannot safely use the values returned by its spawned children until after it executes a *sync* statement, as in Line 6. The keyword *sync* indicates that the procedure must wait as necessary for all its spawned children to complete before proceeding to the statement after the sync.

In our algorithm like its serial counterpart, *PCOMPUT-LIS* finds the LIS of subsequences S_1 and S_2. After the two recursive subroutines in Lines 4 and 5 have completed, which is ensured by the *sync* statement in Line 6, *PCOMPUT-LIS* calls the same *COMBINE-LIS* procedure as its serial counter part. As the *COMBINE-LIS* procedure is serial, it does $\Theta(n)$ work. Assume that $T_p(n)$ denotes the asymptotic running time a parallel algorithm running on p processors. Then, the following recurrence characterizes the asymptotic running time $T_1(n)$ of *PCOMPUT-LIS* on n elements running on a single processor:

$$T_1(n) = 2T_1(n/2) + \Theta(n) = \Theta(n \log n). \tag{2}$$

Clearly this is the same as the serial running time of D&C algorithm. Since, the two recursive calls of *PCOMPUTE-LIS* can now run in parallel, the asymptotic runtime

$T_p(n)$ of *PCOMPUT-LIS* is given by the following recurrence:

$$T_p(n) = T_p(n/2) + \Theta(n) = \Theta(n). \quad (3)$$

Thus the parallelism of *PCOMPUTE-LIS* turns out to be $T_1(n)/T_p(n) = \Theta(\log n)$. Now, we utilize the notion of the *linear speedup* as defined in [13] as follows. Suppose that for a problem, the best known serial algorithm has an asymptotic runtime of $S(n)$ and a parallel algorithm *PAlg* for the same problem has an asymptotic runtime of $T_p(n)$. Here n is the problem size and p is the number of processors for the parallel algorithm. If $S(n)/T_p(n) = \Theta(p)$, then *PAlg* will be said to have achieved a linear speedup.

Note that, we can easily augment the pseudocode of *COMPUTE-LIS-General*(S, P) of Section 3.2 as well to make it a multithreaded algorithm following the same strategy to parallelize COMPUTE-LIS of Section 2. Now, if we consider that the elements are drawn form an arbitrary set (of integers), the $O(n \log n)$ time serial algorithm for computing LIS is clearly optimal. In that case, we have $O(n \log n)/T_p(n) = \Theta(\log n)$. Hence, if we are running on a multiprocessor machine with $\Theta(\log n)$ number of processors, *PCOMPUT-LIS* will achieve a linear speedup. And as a parallel algorithm is said to be *work-optimal* if and only if it has linear speed up [13], *PCOMPUTE-LIS* is a work-optimal algorithm considering the above setting.

4.1. Comparison with other parallel algorithm

In Section 1, we have cited the parallel algorithms on LIS that exist in the literature to the best of our knowledge [12,18,19,22,15]. However, these algorithms either do not achieve work-optimality with respect to the fastest sequential running time, or have rather restrictive slackness conditions. In [12] the presented parallel algorithm in CGM (Coarse Grained Multicomputer) model has $O(n^2/p)$ cost on p processor, which is not work optimal since the fastest sequential algorithm runs in $O(n \log n)$ time. For the same reason the algorithm by Krusche and Tiskin in [15] obtaining computational cost of $O(n^{1.5}/p)$ is also not work optimal. On the EREW PRAM model the given algorithm by Nakashima and Fujiwara [18,19] with $O(m(\frac{n}{p} + \log n))$ time is not work optimal as well. Their second algorithm with $O(\log n + \frac{n \log n}{p} + m^2 \log \frac{n}{p} + m \log p)$ time becomes work optimal only if $p < n/k^2$ where k is the length of an LIS. This algorithm becomes asymptotically sequential once $k > \sqrt{(}n)$. Any sequence of n numbers must have either a monotonically increasing or a monotonically decreasing subsequence of minimum length $\sqrt{(}n)$ [9]. Therefore, for any sequence of numbers, the condition $p < n/k^2$ definitely inhibits parallelism either for running the algorithm on the sequence itself, or on its reversal. The CGM algorithm presented in [22] with $O(n \log(n/p))$ time is not asymptotically faster than the sequential algorithm because the number of processors p only appear in a lower order polynomial term. Notably, in [15] the authors posed an open question as to whether it would be possible to obtain a generally work optimal parallel algorithm for the LIS problem running in time $O((n \log n)/p)$ in the comparison-based model. Our algorithm thus answers this question positively.

5. Conclusion

In this paper, we have presented a divide and conquer approach for the longest increasing subsequence problem. Our algorithm runs in $O(n \log n)$ time which is optimal in the comparison model. We have also shown that our divide and conquer algorithm provides us with a work optimal parallel algorithm.

Notably, for a permutation our sequential algorithm has worse running time than the existing solution. However, we believe our divide and conquer approach is interesting and useful at least from two different angles. Firstly, since many multithreaded algorithms involving nested parallelism follow naturally from the divide-and-conquer paradigm, our approach opens a new and hitherto unexplored avenue to get direct multiprocessor solutions for the LIS problem. Secondly, as all the sequential algorithms for this problem in the literature are online, being offline, our approach may turn out to be at least theoretically interesting.

Acknowledgements

The authors are grateful to the anonymous reviewers for their constructive comments and suggestions.

References

[1] Cilk arts, Inc., Burlington, MA, Cilk++ programmer's guide, 2008.
[2] B.L.A. Lascoux, J.-Y. Thibon, The plactic monoid, in: M. Lothaire (Ed.), Algebraic Combinatorics on Words, Cambridge University Press, Cambridge, UK, 2002, pp. 164–196.
[3] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, Y. Zhou, Cilk: An efficient multithreaded runtime system, J. Parallel Distrib. Comput. 37 (1) (1996) 55–69.
[4] B.M. Chapman, J. LaGrone, Openmp, in: D.A. Padua (Ed.), Encyclopedia of Parallel Computing, Springer, 2011, pp. 1365–1371.
[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3 ed., MIT Press, 2009.
[6] M. Crochemore, G.M. Landau, M. Ziv-Ukelson, A subquadratic sequence alignment algorithm for unrestricted scoring matrices, SIAM J. Comput. 32 (6) (2003) 1654–1673.
[7] M. Crochemore, E. Porat, Fast computation of a longest increasing subsequence and application, Inform. and Comput. 208 (9) (2010) 1054–1059.
[8] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, S.L. Salzberg, Alignment of whole genomes, Nucleic Acid Research 27 (11) (1999) 2369–2376.
[9] P. Erdös, G. Szekeres, A combinatorial problem in geometry, Compos. Math. 2 (1935) 463–470.
[10] M.L. Fredman, On computing the length of longest increasing subsequences, Discrete Math. 11 (1975) 29–35.
[11] M. Frigo, C.E. Leiserson, K.H. Randall, The implementation of the cilk-5 multithreaded language, in: PLDI, 1998, pp. 212–223.
[12] T. Garcia, J.F. Myoupo, D. Semé, A work-optimal cgm algorithm for the lis problem, in: SPAA, 2001, pp. 330–331.
[13] E. Horowitz, S. Sahni, S. Rajasekaran, Fundamentals of Computer Algorithms, Galgotia Publications Pvt. Ltd., 1998.
[14] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest subsequences, Commun. ACM 20 (5) (1977) 350–353.
[15] P. Krusche, A. Tiskin, Parallel longest increasing subsequences in scalable time and memory, in: PPAM (1), 2009, pp. 176–185.
[16] D. Leijen, J. Hall, Optimize managed code for multi-core machines, MSDN Magazine 22 (10) (2007) 79–90.
[17] W.J. Masek, M. Paterson, A faster algorithm computing string edit distances, J. Comput. System Sci. 20 (1) (1980) 18–31.
[18] T. Nakashima, A. Fujiwara, Parallel algorithms for patience sorting and longest increasing subsequence, in: J. Li, K. Kato, H. Kameda

(Eds.), Proceedings of Networks, Parallel and Distributed Processing, and Applications (NPDPA 2002), 2002, pp. 368–374.
[19] T. Nakashima, A. Fujiwara, A cost optimal parallel algorithm for patience sorting, Parallel Process. Lett. 16 (1) (2006) 39–52.
[20] J. Reinders, Intel Threading Building Blocks: Outfitting C++ for Multi-core ProcessorParallelism, O'Reilly Media, Inc., 2007.
[21] C. Schensted, Largest increasing and decreasing subsequences, Canad. J. Math. 13 (1961) 179–191.
[22] D. Semé, A cgm algorithm solving the longest increasing subsequence problem, in: ICCSA (5), 2006, pp. 10–21.
[23] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[24] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Syst. Theory 10 (1977) 99–127.
[25] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.
[26] I.-H. Yang, C.-P. Huang, K.-M. Chao, A fast algorithm for computing a longest common increasing subsequence, Inform. Process. Lett. 93 (5) (2005) 249–253.