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 $S = S[1], S[2], \dots, S[n]$, 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 $\pi = \pi(1), \pi(2), \dots, \pi(n)$ be a permutation of $[1 \dots n]$ and we are given a sequence $S = S[1], S[2], \dots, S[n] = \pi(1), \pi(2), \dots, \pi(n)$, i.e., S is a permutation of $[1 \dots n]$. 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., $1, 2, \dots, n$. 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 $\{1, 2, \dots, n\}$, 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 © 2013 Elsevier B.V. All rights reserved.
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.