Experimental Evaluation of an Efficient Cache-Oblivious LCS Algorithm *

Rezaul Alam Chowdhury

UTCS Technical Report TR-05-43

October 05, 2005

Abstract

We present the results of an extensive computational study of an I/O-optimal cache-oblivious LCS (longest common subsequence) algorithm developed by Chowdhury and Ramachandran. Three variants of the algorithm were implemented (*CO* denoting the fastest variant) along with the widely used linear-space LCS algorithm by Dan Hirschberg (denoted *Hi*). Both algorithms were tested on both random and real-world (CFTR DNA) sequences consisting upto 2 million symbols each, and timing and caching data were obtained on three state-of-the-art architectures (Intel Xeon, AMD Opteron and SUN UltraSparc-III+). In our experiments:

- CO ran a factor of 2 to 6 times faster than Hi.
- Hi incurred upto 4,000 times more L1 cache misses and upto 30,000 times more L2 cache misses than CO when run on pairs of sequences consisting of 1 million symbols each.
- CO executed 40%-50% fewer machines instructions than Hi.
- Unlike Hi, CO was able to conceal the effects of caches on its running time; its actual running time could be predicted quite accurately from its theoretical time complexity.
- CO was less sensitive to alphabet size than Hi.

These results suggest that CO and the algorithmic technique employed by CO can be of practical use in many fields including *sequence alignment* in computational biology.

1 Introduction

Memory in modern computers is typically organized in a hierarchy with registers in the lowest level followed by L1 cache, L2 cache, L3 cache, main memory, and disk, with the access time of each memory level increasing with its level. The *two-level I/O model* [1] is a simple abstraction of this hierarchy that consists of an internal memory of size M, and an arbitrarily large external memory partitioned into blocks of size B. The *I/O complexity* of an algorithm is the number of blocks transferred between these two levels. The I/O model successfully captures the situation where I/O operations between two levels of the memory hierarchy dominate the running time of the algorithm. The *cache-oblivious model* [8] is an extension of this model with the additional requirement that algorithms must not use the knowledge of M and B. A cache-oblivious algorithm is flexible and portable, and simultaneously adapts to all levels of a multi-level memory hierarchy. A well-designed cache-oblivious algorithm typically has the feature that whenever a block is brought into internal memory it contains as much useful data as

*Department of Computer Sciences, University of Texas, Austin, TX 78712. Email: {shaikat}@cs.utexas.edu. This work was supported in part by NSF Grant CCF-0514876 and NSF CISE Research Infrastructure Grant EIA-0303609. Chowdhury is also supported by an MCD Graduate Fellowship.

possible ('spatial locality'), and also the feature that as much useful work as possible is performed on this data before it is written back to external memory ('temporal locality').

The problem of finding the *longest common subsequence* (*LCS*) of two given sequences has a classic *Dynamic Programming* (DP) solution [6] that runs in $\Theta(mn)$ time, uses $\Theta(mn)$ space and performs $\Theta\left(\frac{mn}{B}\right)$ block transfers when working on two sequences of lengths m and n. The LCS problem arises in a wide range of applications in many apparently unrelated fields including computer science, molecular biology, mathematics, speech recognition, gas chromatography, bird song analysis, etc [24, 22]. Perhaps the widely used Unix file comparison program *diff* [14] is the most familiar application in computer science that uses an LCS algorithm. The LCS problem is especially prominent in molecular biology in *sequence alignment*. One of the central problems in genome analysis is to find a maximum-length subsequence of two genomes either under the standard LCS metric or under a refined metric that assigns costs to mismatches, or allows insertions and deletions. Many of these variants can be solved by a dynamic programming algorithm with the same structure as the one for LCS, but with somewhat different computation associated with the steps [6, 16, 26, 24, 22].

It has been shown in [2, 12, 17] that the LCS problem cannot be solved in $o(mn)$ time if the elementary comparison operation is of type 'equal/unequal' and the alphabet size is unrestricted. However, if the alphabet size is fixed the theoretically fastest known algorithm runs in $\mathcal{O}\left(\frac{mn}{\log \min(m,n)}\right)$ time [19] which is unfortunately not suitable for practical implementations. Faster algorithms exist for different special cases of the problem [4].

In most applications, however, the quadratic space required by an LCS algorithm is a more contraining factor than its quadratic running time [10]. For example, one can wait for a week or even a month for finding an LCS of two sequences of length 1 million each, but one can hardly afford the several terabytes of RAM required by the algorithm. Fortunately, there are linear space implementations [11, 15, 3] of the LCS algorithm, but the I/O complexity remains $\Omega\left(\frac{mn}{B}\right)$ and the running time roughly doubles. Hirschberg's space-reduction technique [11] for the DP-based LCS algorithm has become the most widely used trick for reducing the space complexity of similar DP-based algorithms in computational biology [18, 20, 26, 16].

In [7] we present a cache-oblivious implementation of the basic dynamic programming LCS algorithm. Our algorithm continues to run in $\mathcal{O}(mn)$ time and uses $\mathcal{O}(m+n)$ space, but it performs only $\mathcal{O}\left(\frac{mn}{BM}\right)$ block transfers. We show that our algorithm is I/O-optimal in that it performs the minimum number of block transfers (to within a constant factor) of any implementation of the dynamic programming algorithm for LCS. This algorithm can be adapted to solve the *edit distance* problem [13, 6] within the same bounds; this latter problem asks for the minimum cost of an edit sequence that transforms a given sequence into another one with the allowable edit operations being insertion, deletion and substitution of symbols each having a cost based on the symbol(s) on which it is to be applied.

Our Results. We present the results of an extensive empirical study of the cache-oblivious LCS algorithm given in [7]. We implemented three variants of the algorithm (denoting the fastest variant by *CO*) along with the widely used linear-space LCS algorithm by Hirschberg (denoted *Hi*) [11]. We ran both algorithms on both random and real-world sequences consisting upto 2 million symbols each, and obtained timing and caching data on three state-of-the-art architectures: Intel Xeon, AMD Opteron and SUN UltraSparc-III+.

On random sequences CO ran a factor of 2 to 6 times faster than Hi and consistently executed 40%-50% fewer machines instructions. As expected, CO exhibited a far better cache performance than Hi: when run on sequences of length 1 million or more CO incurred only a small fraction of cache misses ($\frac{1}{4000}$ for L1 cache and $\frac{1}{30{,}000}$ for L2 cache) compared to Hi. Unlike Hi, CO was able to conceal the effects of caches on its running time; its actual running time could be predicted quite accurately from

its theoretical time complexity. Moreover, CO appeared to be less sensitive to alphabet size than Hi. On real-world DNA sequences (CFTR gene sequences [25]), too, CO ran about 2 times faster than Hi.

Our experimental results suggest that CO and the algorithmic technique employed by CO can be of practical value in many application areas including *sequence alignment* in computational biology.

Related Work. In [4], Bergroth et al. conducted an empirical study of the running times of different LCS algorithms. However, they excluded linear-space algorithms from their study, and considered sequences of length at most 4000. If only the length of the LCS is needed, and not an actual subsequence, the technique for cache-oblivious stencil computation presented in [9] can achieve the same time, space and I/O bounds as our algorithm. Experimental results show that the algorithm runs upto 7 times faster than the standard algorithm [9].

In [5] an empirical study of a DP-based cache-oblivious optimal matrix chain multiplication algorithm was conducted. A cache-oblivious algorithm for Floyd-Warshall's APSP algorithm is given in [21] and experimental results show that the algorithm runs upto 10 times faster than the standard Floyd-Warshall algorithm.

2 Experimental Setup

2.1 Test Data Sets. In our experiments we used two kinds of sequences:

- **Random Sequences.** We used random sequences of lengths ranging from 2^{10} ($\approx$ one thousand) to 2^{21} ($\approx$ 2 million) drawn from alphabets of sizes 26 (i.e., the English alphabet) and 4 (i.e., the set $\{A, C, G, T\}$ of DNA bases).
- **Real-world Sequences.** We used sequences from 11 species (baboon, cat, chicken, chimpanzee, cow, dog, fugu, human, mouse, rat and zebrafish) generated for the genomic segment harboring the cystic fibrosis transmembrane conductance regulator (CFTR) gene [25]. The lenghts of the sequences ranged from 0.42 million to 1.80 million.

2.2 Computing Environment. We ran our experiments on the following three architectures:

- **Intel Xeon.** A four processor 3.06 GHz Intel Xeon shared memory machine with 4 GB of RAM and running Linux 2.4.29. Each processor had an 8 KB L1 data cache (4-way set associative) and an on-chip 512 KB unified L2 cache (8-way). The block size was 64 bytes for both caches.
- **AMD Opteron.** A dual processor 2.4 GHz AMD Opteron shared memory machine with 4 GB of RAM and running Linux 2.4.29. Each processor had a 64 KB L1 data cache (2-way) and an on-chip 1 MB unified L2 cache (8-way). The block size was 64 bytes for both caches.
- **SUN Blade.** A 1 GHz Sun Blade 2000/1000 (UltraSPARC-III+) with 1 GB of RAM and running SunOS 5.9. The processor had an on-chip 64 KB L1 data cache (4-way) and an off-chip 8 MB L2 cache (2-way). The block sizes were 32 bytes for the L1 cache and 512 bytes for the L2 cache.

We used the *Cachegrind* profiler [23] for simulating cache effects on Intel Xeon. The caching data on the Sun Blade was obtained using the *cputrack* utility that keeps track of hardware counters. All algorithms were implemented in C using a uniform programming style and compiled using *gcc* with optimization parameter -O3. Each machine was exclusively used for experiments (i.e., no other programs were running on them), and on multi-processor machines only a single processor was used.

2.3 Overview of Algorithms Implemented. A sequence $Z = \langle z_1, z_2, \ldots z_k \rangle$ is called a *subsequence* of another sequence $X = \langle x_1, x_2, \ldots x_m \rangle$ if there exists a strictly increasing function $f : [1, 2, \ldots, k] \rightarrow [1, 2, \ldots, m]$ such that for all $i \in [1, k]$, $z_i = x_{f(i)}$. A sequence Z is a *common subsequence* of sequences X and Y if Z is a subsequence of both X and Y. Given two sequences X and Y, the *Longest Common Subsequence* (LCS) problem asks for a maximum-length common subsequence of X and Y.

2.3.1 Classic Dynamic Programming Algorithm. Given two sequences $X = \langle x_1, x_2, \ldots x_m \rangle$ and $Y = \langle y_1, y_2, \ldots y_n \rangle$, we define $c[i, j]$ $(0 \leq i \leq m, 0 \leq j \leq n)$ to be the length of an LCS of $\langle x_1, x_2, \ldots x_i \rangle$ and $\langle y_1, y_2, \ldots y_j \rangle$, which can be computed using the following recurrence relation (see, e.g., [6]):

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases} \tag{2.1}$$

The classic dynamic programming solution to the LCS problem which we will refer to as CLASSIC-LCS, is based on this recurrence relation, and computes the entries of $c[0 \ldots m, 0 \ldots n]$ in row-major order in $\Theta(mn)$ time and incurs $\mathcal{O}\left(\frac{mn}{B}\right)$ cache misses. Further, after all entries of $c[0 \ldots m, 0 \ldots n]$ are computed, we can trace back the sequence of decisions that led to the value computed for $c[m, n]$, and thus recover an LCS of X and Y in $\mathcal{O}(m+n)$ additional time, while incurring $\Theta(m+n)$ I/Os. Observe that since the entries of any row of c depends only on the entries in the previous row, $c[m, 0 \ldots n]$ can be computed using only $\mathcal{O}(n)$ extra space, and thus the length of an LCS can be computed in linear space. However, the algorithm needs $\Theta(mn)$ space to compute an actual LCS sequence.

2.3.2 Hirschberg's Linear Space Algorithm. Hirschberg [11] gives an $\mathcal{O}(m+n)$ space algorithm, which finds an LCS in $\mathcal{O}(mn)$ time and $\mathcal{O}\left(\frac{mn}{B}\right)$ I/Os. We briefly describe the algorithm below.

If $n = 1$, the LCS can be determined in a single scan of X. Otherwise, let $k = \left\lfloor \frac{n}{2} \right\rfloor$, $Y_f = \langle y_1, y_2, \ldots y_k \rangle$ and $Y_b = \langle y_n, y_{n-1}, \ldots y_{k+1} \rangle$. Hirschberg computes two arrays $L_f[1 \ldots m]$ and $L_b[1 \ldots m]$, where for $1 \leq i \leq m$, $L_f[i]$ is the length of an LCS between $\langle x_1, x_2, \ldots x_i \rangle$ and Y_f, and $L_b[i]$ is the length of an LCS between $\langle x_1, x_2, \ldots x_i \rangle$ and Y_b. Observe that both L_f and L_b can be computed in $\mathcal{O}(mn)$ time using only $\mathcal{O}(m)$ extra space (see section 2.3.1). Let j $(1 \leq j \leq m)$ be the smallest index such that $L_f[j] + L_b[j] = \max_{1 \leq i \leq m} (L_f[i] + L_b[i])$. Then Hirschberg recursively computes an LCS Z_1 between $\langle x_1, x_2, \ldots x_j \rangle$ and $\langle y_1, y_2, \ldots y_k \rangle$, and Z_2 between $\langle x_{j+1}, x_{j+2}, \ldots x_m \rangle$ and $\langle y_{k+1}, y_{k+2}, \ldots y_n \rangle$, and returns $Z = Z_1 || Z_2$ as an LCS of X and Y.

Our Implementation. Our implementation of Hirschberg's algorithm differs from the original algorithm in how the base case is handled. Once both m and n drop below some preset threshold value (BASE) we solve the problem using CLASSIC-LCS. Provided BASE $= \mathcal{O}\left(\sqrt{m+n}\right)$, the space usage remains linear, but the algorithm runs faster.

2.3.3 Cache-Efficient LCS Algorithm. In [7] we present an I/O-optimal cache-oblivious implementation of the dynamic programming algorithm defined by recurrence 2.1. The algorithm runs in $\mathcal{O}(mn)$ time, uses $\mathcal{O}(m+n)$ space and incurs $\mathcal{O}\left(\frac{mn}{BM}\right)$ I/Os. We describe the algorithm below, where we assume for convenience that $n = m = 2^p$ where p is a nonnegative integer.

For any submatrix $c[i_1 \ldots i_2, j_1 \ldots j_2]$ of c where $i_2 \geq i_1 > 0$ and $j_2 \geq j_1 > 0$, we refer to $c[i_1 - 1, j_1 \ldots j_2]$ and $c[i_1 \ldots i_2, j_1 - 1]$ as the *input boundary* of the submatrix, and $c[i_2, j_1 \ldots j_2]$ and $c[i_1 \ldots i_2, j_2]$ as the *output boundary*. The function LCS-OUTPUT-BOUNDARY (given in [7]) when called with parameters i, j, r, and a one dimensional array D containing the input boundary of the submatrix $c[i \ldots i + r - 1, j \ldots j + r - 1]$ returns the output boundary of that submatrix in D. For simplicity of exposition, we assume that D can have negative indices, and entry $c[i', j']$ of c is stored in $D[i' - j']$. The function works by recursively subdividing the input matrix into quadrants, and for an input matrix of dimension $n \times n$ runs in time $\mathcal{O}\left(n^2\right)$, uses $\mathcal{O}(n)$ space and incurs $\mathcal{O}\left(\frac{n^2}{BM}\right)$ cache misses.

Recall that if all entries of $c[1 \ldots n, 1 \ldots n]$ are available, one can trace back the sequence of decisions that led to the value computed for $c[n, n]$, and thus retrieve the elements on an LCS of X and Y. We can view this sequence of decisions as a path through c that starts at $c[n, n]$ and ends at the input boundary of $c[1 \ldots n, 1 \ldots n]$. We call this path an *LCS Path*.

RECURSIVE-LCS$(X, Y, i, j, i_s, j_s, D)$

Input. The input boundary of $c[i \dots i+r-1, j \dots j+r-1]$, where $r = \max(i_s - i + 1, j_s - j + 1)$, is stored in D, with boundary entry $c[i', j']$ in $D[i' - j']$. The entry $c(i_s, j_s)$ lies on the output boundary of $c[i \dots i+r-1, j \dots j+r-1]$.

Output. Let i' and j' be the values supplied in i_s and j_s in the input. Returns an LCS Z of $X[i \dots i']$ and $Y[j \dots j']$, updates i_s and j_s to return the point at which the *LCS Path* starting at $c[i', j']$ intersects the input boundary of $c[i \dots i', j \dots j']$.

1. $b \leftarrow i - j,\ \ r_i \leftarrow i_s - i + 1,\ \ r_j \leftarrow j_s - j + 1,\ \ r \leftarrow \max(r_i, r_j),\ \ Z \leftarrow \emptyset$

2. ***if*** $r \leq$ BASE ***then*** compute in Z the LCS of the subproblem and return the appropriate values for i_s and j_s ***else***

3. $r' \leftarrow \frac{r}{2}$

4. $top\text{-}left[1 \dots r+1] \leftarrow D[b - r' \dots b + r']$ {*save input boundary of top-left quadrant*}

LCS-OUTPUT-BOUNDARY(X, Y, i, j, r', D) {*generate output boundary of top-left quadrant*}

5. ***if*** $i_s \geq i + r'$ ***and*** $j_s \geq j + r'$ ***then*** {*if the LCS intersects the bottom-right quadrant*}

6. $bottom\text{-}left[1 \dots r+1] \leftarrow D[b - r \dots b]$ {*save input boundary of bottom-left quadrant*}

LCS-OUTPUT-BOUNDARY$(X, Y, i, j + r', r', D)$ {*generate output boundary of bottom-left quadrant*}

7. $top\text{-}right[1 \dots r+1] \leftarrow D[b \dots b + r]$ {*save input boundary of top-right quadrant*}

LCS-OUTPUT-BOUNDARY$(X, Y, i + r', j, r', D)$ {*generate output boundary of top-right quadrant*}

8. $Z \leftarrow$ RECURSIVE-LCS$(i + r', j + r', i_s, j_s, D)$ # Z {*find LCS fragment in bottom-right quadrant*}

9. ***if*** $i_s \geq i + r'$ ***then*** $D[b \dots b + r] \leftarrow top\text{-}right[1 \dots r+1]$ {*restore input boundary of top-right quadrant*}

10. ***elif*** $j_s \geq j + r'$ ***then*** $D[b - r \dots b] \leftarrow bottom\text{-}left[1 \dots r+1]$ {*restore input boundary of bottom-left quadrant*}

11. ***if*** $i_s \geq i + r'$ ***then*** $Z \leftarrow$ RECURSIVE-LCS$(X, Y, i + r', j, i_s, j_s, D)$ # Z {*find LCS fragment in top-right quadrant*}

12. ***elif*** $j_s \geq j + r'$ ***then*** $Z \leftarrow$ RECURSIVE-LCS$(X, Y, i, j + r', i_s, j_s, D)$ # Z {*LCS fragment in bottom-left quadrant*}

13. ***if*** $i_s \geq i$ ***and*** $j_s \geq j$ ***then*** {*if the LCS intersects the top-left quadrant*}

14. $D[b - r' \dots b + r'] \leftarrow top\text{-}left[1 \dots r+1]$ {*restore input boundary of top-left quadrant*}

15. $Z \leftarrow$ RECURSIVE-LCS$(X, Y, i, j, i_s, j_s, D)$ # Z {*find LCS fragment in top-left quadrant*}

16. ***return*** Z

Our algorithm traces an LCS path without storing all entries of c; instead it only stores the boundaries of certain subproblems. It uses a recursive function RECURSIVE-LCS (given below and in [7]) to construct the LCS. When called with parameters i, j, i_s, j_s and D, RECURSIVE-LCS assumes that the input boundary of the submatrix $c[i \dots i + r - 1, j \dots j + r - 1]$ is stored in the one dimensional array D of length $2r + 1$, i.e., entry $c[i', j']$ on the boundary is stored in $D[i' - j']$, where $r = \max(i_s - i + 1, j_s - j + 1) = 2^p$ for some non-negative integer p. It also assumes that an LCS path intersects the output boundary of $c[i \dots i + r - 1, j \dots j + r - 1]$ at $c[i_s, j_s]$. This function traces the fragment of that path through this submatrix, and returns the LCS of X and Y along this subpath. It also finds the entry $c[i', j']$ at which this path intersects the input boundary of the given submatrix, and updates i_s and j_s to i' and j', respectively. If r is sufficiently small it solves the problem iteratively using CLASSIC-LCS, otherwise it solves the problem recursively by dividing the input submatrix into four quadrants. It first calls LCS-OUTPUT-BOUNDARY at most three times (at most once for each quadrant except the bottom-right one) in order to generate the input boundaries of the top-right and the bottom-left quadrants, and if required (if $i_s \geq i + \frac{r}{2}$ and $j_s \geq j + \frac{r}{2}$), for the bottom-right quadrant. Observe that the LCS path can pass through at most three quadrants of the current submatrix. This function locates those quadrants one after another based on the current values of i_s and j_s (i.e., based on which quadrant $c[i_s, j_s]$ belongs to), and calls itself recursively in order to trace the fragment of the LCS path that passes through that quadrant. (note that the recursive calls modify i_s and j_s). The output of RECURSIVE-LCS is the concatenation of these LCS fragments in the correct order.

The initial call is to RECURSIVE-LCS$(X, Y, 1, 1, n, n, D)$, with $D[-n \dots n]$ initialized to all zeros.

Complexities. It has been shown in [7] that the algorithm incurs $\mathcal{O}\left(1 + \frac{n}{B} + \frac{n^2}{BM}\right)$ cache misses while