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 $c[i, j]$ $(0 \le i \le m, 0 \le j \le n)$ 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]):

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

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 $1 \le i \le m$, $L_f[i]$ 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 $(1 \le j \le m)$ be the smallest index such that $L_f[j] + L_b[j] = \max_{1 \le i \le m}(L_f[i] + L_b[i])$. 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 $i_2 \ge i_1 > 0$ and $j_2 \ge j_1 > 0$, 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 $\mathcal{O}(n^2)$, 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 $c[i \dots i', j \dots j']$.

$b \leftarrow i - j,\ \ r_i \leftarrow i_s - i + 1,\ \ r_j \leftarrow j_s - j + 1,\ \ r \leftarrow \max(r_i, r_j),\ \ Z \leftarrow \emptyset$
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***
    $r' \leftarrow \frac{r}{2}$
    top-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*}
    if $i_s \geq i + r'$ ***and*** $j_s \geq j + r'$ ***then***   {*if the LCS intersects the bottom-right quadrant*}
        bottom-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*}
        top-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*}
        $Z \leftarrow$ RECURSIVE-LCS$(i + r', j + r', i_s, j_s, D)$ # Z {*find LCS fragment in bottom-right quadrant*}
        if $i_s \geq i + r'$ ***then*** $D[b \dots b + r] \leftarrow$ *top-right*$[1 \dots r+1]$ {*restore input boundary of top-right quadrant*}
          elif $j_s \geq j + r'$ ***then*** $D[b - r \dots b] \leftarrow$ *bottom-left*$[1 \dots r+1]$ {*restore input boundary of bottom-left quadrant*}
    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*}
      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*}
    if $i_s \geq i$ ***and*** $j_s \geq j$ ***then***   {*if the LCS intersects the top-left quadrant*}
        $D[b - r' \dots b + r'] \leftarrow$ *top-left*$[1 \dots r+1]$ {*restore input boundary of top-left quadrant*}
        $Z \leftarrow$ RECURSIVE-LCS$(X, Y, i, j, i_s, j_s, D)$ # Z {*find LCS fragment in top-left quadrant*}
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

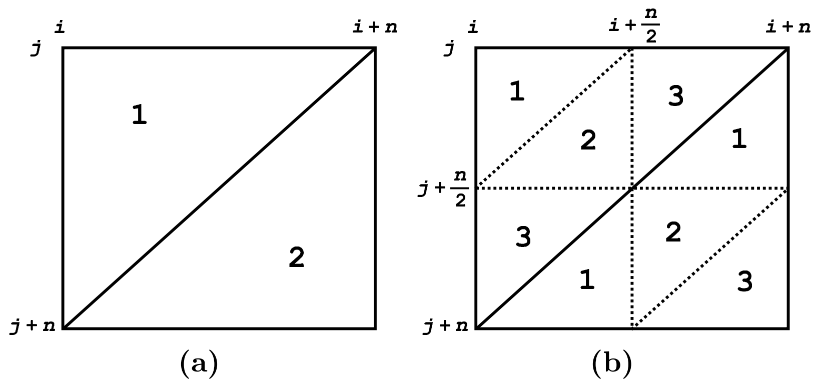


Figure 1: The triangulation method: **(a)** the order in which triangles are processed after the initial LCS matrix is triangulated, **(b)** the order in which sub-triangles are processed in each triangle.

Architecture	Base Case Size
Intel Xeon	256×256
AMD Opteron	512×512
SUN Blade	1024×1024

Figure 2: Size of the base case used for CO and Hi on each architecture.

the running time and space complexity remain $\mathcal{O}(n^2)$ and $\mathcal{O}(n)$, respectively.

It is straightforward to extend this algorithm to handle input strings of unequal lengths m and n, and of lengths that are not powers of 2 with $\mathcal{O}\left(\frac{mn}{BM}\right)$ I/Os while running in $\mathcal{O}(mn)$ time using $\mathcal{O}(m+n)$ space.

Implementations. We implemented the following three variants of our algorithm based on how the LCS matrix is recursively partitioned:

- **The Quartering Method.** This is the 4-way partitioning algorithm as described above.
- **The Triangulation Method.** In this method we recursively partition the LCS matrix into triangles and process them in the order as shown in Figure 1.
- **The Bisection Method.** We always partition the matrix into halves along the longer dimension.

All three methods have the same asymptotic bounds.

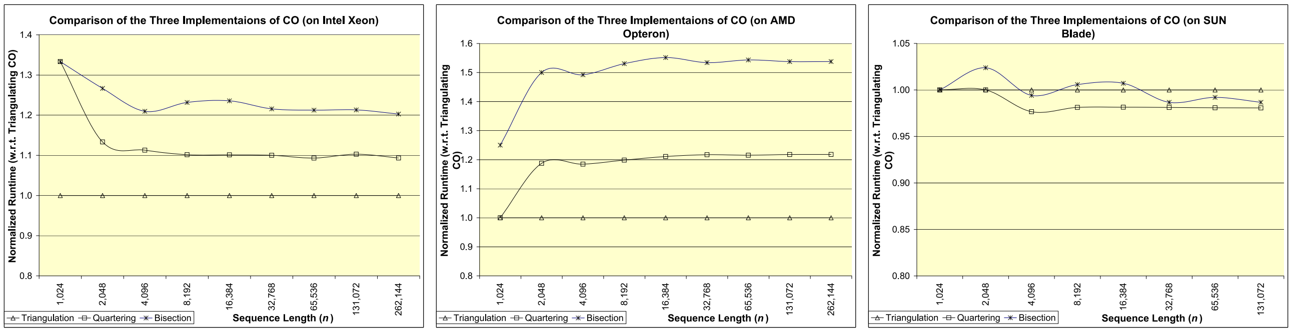
3 Experimental Results

We discuss our experimental results below. In this and subsequent sections we refer to the implementation of the new cache-oblivious algorithm as *CO* and that of Hirschberg's algorithm as *Hi*. In our experiments we have not included Classic-LCS as a stand-alone algorithm since it was unable to handle input strings longer than 27,000 (assuming $m = n$) on any machine.

3.1 Random Sequences. Unless mentioned otherwise all sequences in this section are understood to be drawn from an alphabet of size 26 (the English alphabet), and each numerical figure (i.e., running time, cache misses) is the average of 5 independent runs.

3.1.1 Base Case Size. If the value of base is too small in LCS-Output-Boundary and Recursive-LCS, the overhead of recursion dominates the running times of those routines, and if it is too big the I/O overhead dominates. We empirically determined the best value of base on each of the three architecures (see Figure 2) and used them for all our experiments. However, Hirschberg's algorithm did not seem to be sensitive to the size of the base case unless it is too small since the I/O overhead of Hi dominates the cost of recursion most of the time. We used the same base case size for both CO and Hi in all of our studies.

3.1.2 Comparing the Three Implementations. We executed all three variants of CO on all three architectures. The triangulation method ran the fastest on Intel Xeon (see Fig. 3(a)) and AMD Opteron

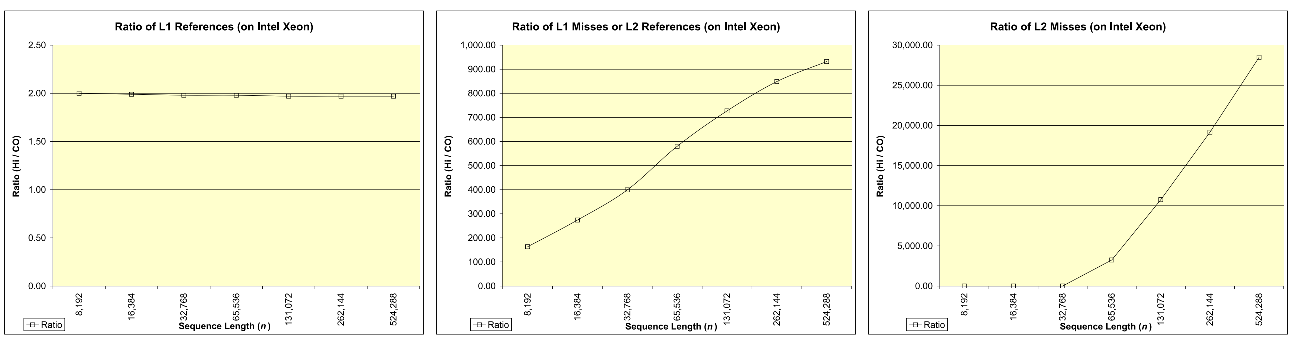


(a) **(b)** **(c)**

Figure 3: Comparison of running times of the three implementations (triangulation, quartering and bisection) of the cache-oblivious algorithm: **(a)** on Intel Xeon, **(b)** on AMD Opteron, and **(c)** on SUN Blade. Each running time is normalized with respect to the corresponding running time of the triangulation method.

(see Fig. 3(b)), and the quartering method was the fastest on SUN Blade (see Fig. 3(c)). On Intel Xeon the quartering method ran around 10% slower compared to the triangulation method, and the bisection method ran around 20% slower. On AMD Opteron those two figures were around 20% and 55%, respectively. It turns out that though all three methods perform almost the same number of memory operations (i.e., operations involving a memory location), both the quartering method and the bisection method perform around 46% more index variable operations (i.e., operations that manipulate index variables) compared to the triangulation method when run on the same pair of sequences. The overhead of manipulating the index variables contribute to the slowdown of the quartering and the bisection methods. The bisection method, however, makes about 50% more recursive function calls compared to the quartering method which explains why the former is slower than the later method. However, on SUN Blade the running times of the three methods were within 2% of one another. This happens because memory operations are slower on SUN Blade (since it has a slower off-board L2 cache) and so the cost of index variable operations and the recursive function calls no longer dominate.

For the remaining experiments in this technical report we used the variant of CO based on triangulation method.

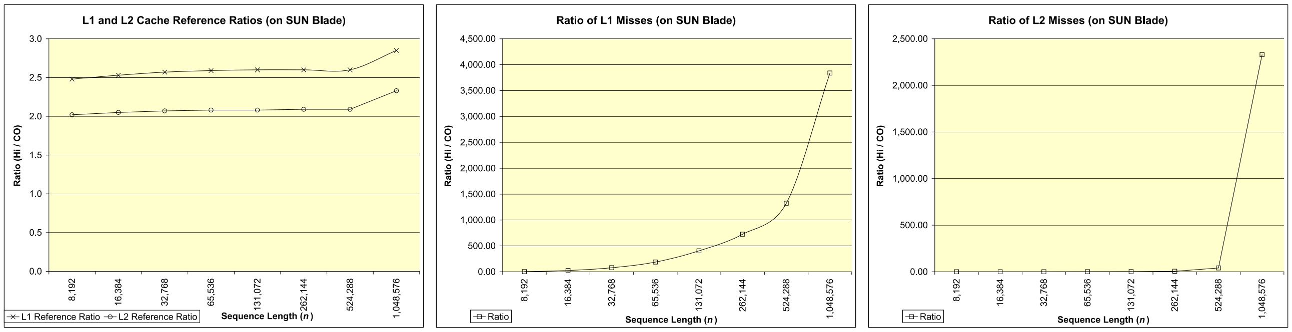


(a) **(b)** **(c)**

Figure 4: Comparison of cache performance on Intel Xeon with an L1 cache of size 8 KB (64 B blocks) and an L2 cache of size 512 KB (64 B blocks): **(a)** ratio of L1 references made by Hi to that made by CO, **(b)** ratio of L1 misses which is also the ratio of L2 references, and **(c)** ratio of L2 misses.

3.1.3 Cache Performance. We compared CO and Hi in terms of the number of L1 and L2 cache references and misses on Intel Xeon and SUN Blade:

- **Intel Xeon.** The data on the Intel Xeon processor was obtained using the *Cachegrind* profiler [23]. The profiler slows down the running program considerably, and so we have not obtained data



(a) **(b)** **(c)**

Figure 5: Comparison of cache performance on SUN Blade with an L1 cache of size 64 KB (32 B blocks) and an L2 cache of size 8 MB (512 B blocks): **(a)** ratio of cache references (L1 and L2 separately) made by Hi to that made by CO, **(b)** ratio of L1 misses, and **(c)** ratio of L2 misses.

for sequences longer than 524,288 (2^{19}).

Our experimental results show that Hi consistently makes 97% more L1 cache references compared to CO (see Fig. 4(a)), but the ratio of the number of cache misses incurred by Hi to that incurred by CO increases as the sizes of the sequences increase (see Fig. 4(b)). For $n = 8,192 = 2^{13}$, Hi incurs 163 times more cache misses than does CO while for $n = 524,288 = 2^{19}$, this ratio reaches 932. We expect that this ratio will ultimately stabilize at γM for large enough sequences, where γ is a machine dependent constant.

The number L2 cache references is the same as the number of L1 cache misses on Intel Xeon (see Fig. 4(b)). However, it appears that the number of L2 cache misses incurred by either algorithm is insignificant upto $n = 32,768 = 2^{15}$ since upto that point the input is small enough to fit entirely into the L2 cache which has a size of 512 KB. Starting from $n = 65,536 = 2^{16}$, the ratio of of L2 cache misses incurred by Hi to that incurred by CO increases with the increase of the sequence length, reaching 28,487 for $n = 524,288 = 2^{19}$ (see Fig. 4(c)).

- **SUN Blade.** The caching data on the SUN Blade was obtained using the *cputrack* utility that keeps track of hardware counters.

  From our experimental results it appears that Hi consistently makes about 160% more L1 cache references compared to that made by CO (see Fig. 5(a)), and similar to what happened on Intel Xeon the ratio of L1 cache misses incurred by Hi to that incurred by CO increases as the sizes of the input increase. This ratio starts at 2 for $n = 8,192 = 2^{13}$ and reaches 3,839 as n reaces $1,048,576 = 2^{20}$ (see Fig. 5(b)).

  Unlike Intel Xeon, however, SUN Blade has a write-through L1 cache, and so the number of L2 cache references is not the same as the number of L1 misses. In fact, Hi consistently makes 110% more L2 references compared to CO (see Fig. 5(a)). The L2 cache on SUN Blade is large enough (8 MB) to hold inputs upto size $n = 131,072 = 2^{17}$, and so the number of L2 cache misses becomes significant after that point. Starting from $n = 262,144 = 2^{18}$ the ratio of L2 misses incurred by Hi to that incurred by CO increases with the increase of n, and rapidly reaches $2,330$ as n increases to $1,048,576 = 2^{20}$ (see Fig. 5(c)).

3.1.4 Running Time. Across all machines, our cache-oblivious algorithm consistently ran faster than Hirchberg's algorithm on randomly generated equal-length sequence-pairs, with the ratio of running times increasing gradually as the sizes of the sequences increase (see Fig. 6). For example, for sequence pairs of length 1 million ($n = 2^{20}$) our algorithm ran twice as fast as Hirschberg's algorithm on the Intel Xeon and the AMD Opteron, and 5 times faster on the SUN Blade.

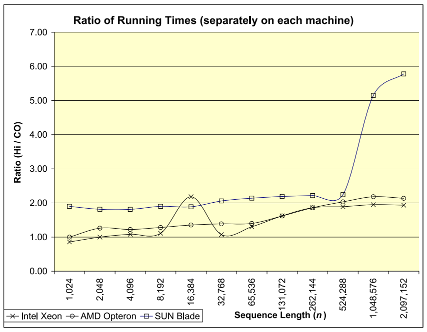


Figure 6: Ratio of running time (separately on Intel Xeon, AMD Opteron and SUN Blade) of Hi to that of CO on equal-length sequences.

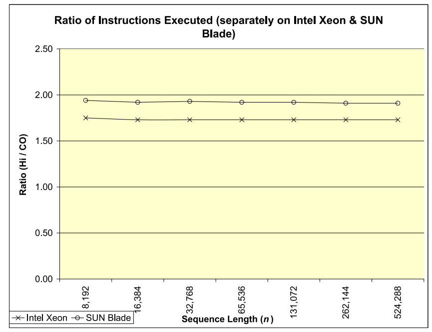


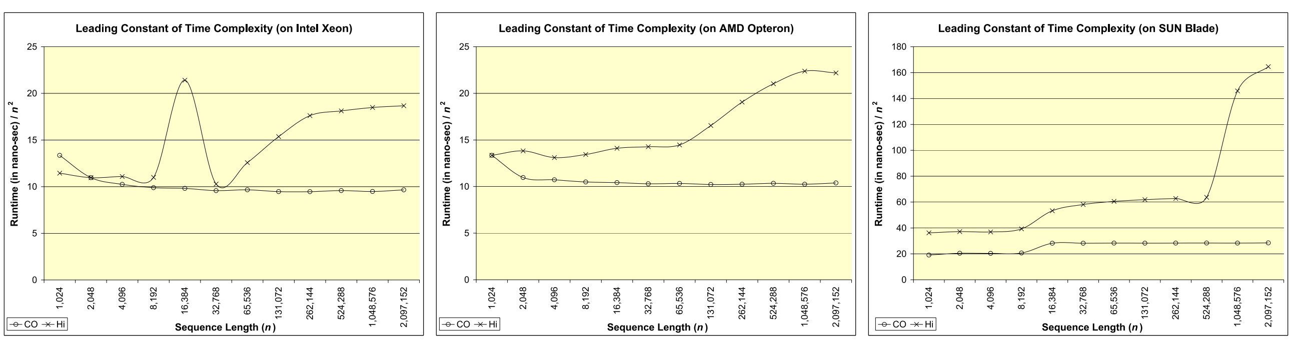
Figure 7: Ratio of instructions executed (separately on Intel Xeon and SUN Blade) for Hi to that executed for CO.

On SUN Blade Hirschberg's algorithm dramatically slowed down compared to the cache-oblivious algorithm as soon as n reached 2^{20}. We believe this happened because at that point the input became too large to completely fit in the L2 cache and Hi started to incur significantly more L2 misses than CO did (see Fig. 5(c)). The RAM in the SUN Blade is slower than that in either of the two more recent machines (Intel Xeon and AMD Opteron), and that causes programs to slow down considerably if the RAM is not used I/O-efficiently.

The graph for Intel Xeon in Fig. 6 shows a sudden spike at $n = 16,384 = 2^{14}$. We think this happened because data in Hirschberg's algorithm did not align suitably with the L2 cache.

Since disks have very large access latencies, we expect that CO will run far faster than Hi if the sequences are too large to fit in the RAM.

3.1.5 Instruction Count. Hirschberg's algorithm executed a larger number of machine instructions than the cache-oblivious algorithm; for input sizes from 2^{13} to 2^{19} the ratio stayed consistently around 1.73 on Intel Xeon, and on Sun Blade it stayed around 1.92 (see Fig. 7). We also counted the number of C operations performed by both algorithms and found that Hi consistently performed 33% more memory operations and 50% more index variable operations compared to CO.

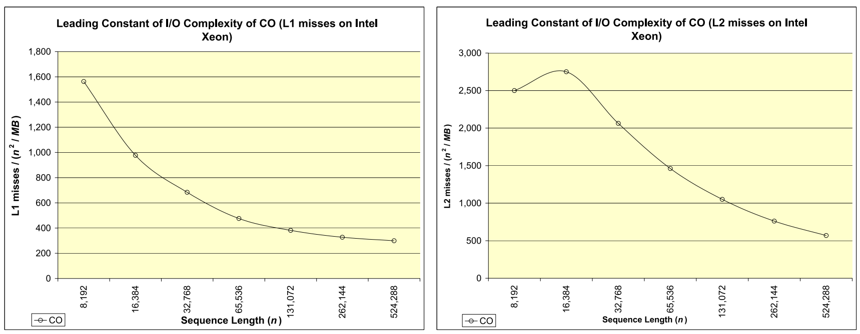


(a) **(b)** **(c)**

Figure 8: Leading constant of time complexity (i.e., value of $\frac{\text{runtime in ns}}{n^2}$): **(a)** on Intel Xeon, **(b)** on AMD Opteron, and **(c)** on SUN Blade.

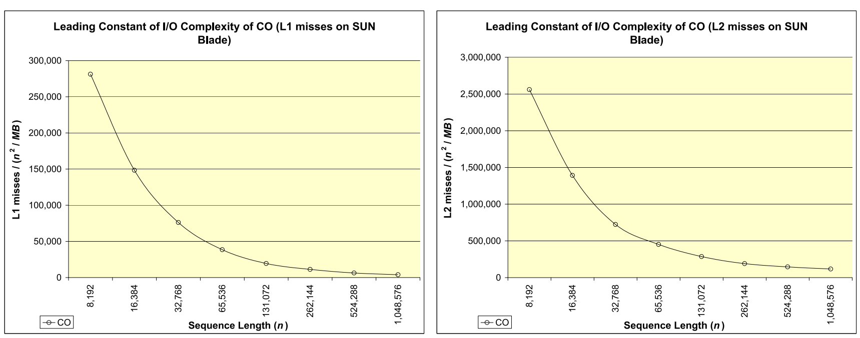
3.1.6 Predicting Running Times from Theoretical Time Complexities. In Fig. 8 we plot the value of $\left(\frac{\text{runtime in ns}}{n^2}\right)$ for both CO and Hi on all three architectures. For CO this value is almost constant on both Intel Xeon and AMD Opteron, and its running time (in ns) on these two machines can be approximated by $10n^2$ for two random sequences of length n each. Thus on these machines CO can effectively conceal the effect of different caches on its running time. However, on SUN Blade its running time (in ns) can be approximated by $20n^2$ for $n \le 2^{13}$ and by $30n^2$ for $n > 2^{13}$, i.e, CO cannot completely conceal the effect of L2 cache on this machine. We believe this happens because the SUN Blade has a slower (off-chip) L2 cache compared to that on the Intel Xeon or the AMD Opteron.

Across all machines Hi has a larger value of $\left(\frac{\text{runtime in ns}}{n^2}\right)$ than CO, and this ratio is not constant for all values of n. On Intel Xeon the ratio remains around 11 until $n = 2^{15}$ (i.e., until L2 cache misses start to occur, ignoring the spike at $n = 2^{14}$), and then it gradually increases, stabilizing at around 18 after $n = 2^{18}$. On AMD Opteron the ratio lies around 14 until $n = 2^{16}$ (L2 cache misses start to occur after that point), and then it slowly increases to 22 at $n = 2^{20}$. On SUN Blade the ratio remains at around 37 upto $n = 2^{13}$ (after that L1 cache misses start to occur), lies around 60 between $n = 2^{14}$ and $n = 2^{19}$ (L2 cache misses occur after $n = 2^{19}$), and exceeds 140 after that.



(a) **(b)**

Figure 9: Leading constant of I/O complexity (i.e., value of $\frac{\text{cache misses}}{n^2/MB}$) of CO on Intel Xeon: **(a)** for L1 cache ($M = 8$ KB, $B = 64$ B, 4-way), and **(b)** for L2 cache ($M = 512$ KB, $B = 64$ B, 8-way).

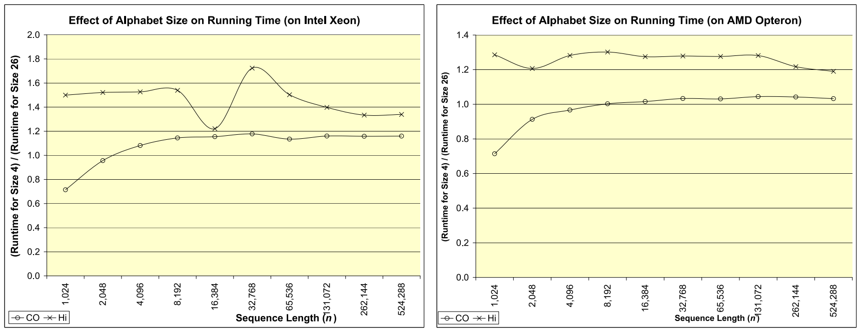


(a) **(b)**

Figure 10: Leading constant of I/O complexity (i.e., value of $\frac{\text{cache misses}}{n^2/MB}$) of CO on SUN Blade: **(a)** for L1 cache ($M = 64$ KB, $B = 32$ B, 4-way, write-through), and **(b)** for L2 cache ($M = 8$ MB, $B = 512$ B, 2-way).

3.1.7 Empirical and Theoretical I/O Complexities.

In Figure 9 we plot the value of $\left(\frac{\text{cache misses}}{n^2/MB}\right)$ for both caches (L1 and L2) when CO is run on the Intel Xeon. We plot the same for the SUN Blade in Figure 10. The theoretical I/O complexity of the algorithm has been derived for a fully associative cache that employs an optimal offline cache-replacement policy. The caches on the Intel Xeon and the SUN Blade are, however, only 2 to 8-way set-associative, and they do not use optimal cache-replacement strategies. Moreover, the L1 cache on the SUN Blade is write-through. For these reasons the plotted values are quite large (ideally we would expect a value of 1) though in all cases they are decreasing with the increase of n.



(a) **(b)**

Figure 11: Effect on the running time of CO and Hi when the alphabet size is changed from 26 to 4: **(a)** on Intel Xeon, and **(b)** on AMD Opteron.

Sequence pairs with lengths (10^6)	Running time CO (t_1)	Hi (t_2)	ratio ($\frac{t_2}{t_1}$)	Sequence pairs with lengths (10^6)	Running time CO (t_1)	Hi (t_2)	ratio ($\frac{t_2}{t_1}$)
cat/dog (1.16/1.05)	$5h$ $51m$	$8h$ $39m$	1.48	human/chicken (1.80/0.42)	$2h$ $47m$	$4h$ $59m$	1.79
rat/mouse (1.50/1.49)	$8h$ $0m$	$15h$ $54m$	1.99	human/chimp (1.80/1.32)	$7h$ $37m$	$17h$ $34m$	2.31
baboon/chimp (1.51/1.32)	$7h$ $22m$	$14h$ $46m$	2.00	human/cow (1.80/1.46)	$9h$ $27m$	$18h$ $55m$	2.00
human/zebrafish (1.80/0.16)	$1h$ $8m$	$1h$ $45m$	1.56	human/rat (1.80/1.50)	$9h$ $40m$	$20h$ $33m$	2.13
human/fugu (1.80/0.27)	$1h$ $54m$	$3h$ $8m$	1.65	human/baboon (1.80/1.51)	$8h$ $52m$	$19h$ $18m$	2.18

Table 1: The figures give the time for a single run on pairs of CFTR DNA sequences on AMD Opteron.

3.1.8 Effect of Alphabet Size on Running Times. If the size of the alphabet decreases running times of both algorithms increase. However, change of alphabet size has a much smaller impact on the running time of CO than on Hi (see Fig. 11). This happens because Hirschberg's algorithm has a code segment (reversing the LCS's obtained for base cases) that directly depends on the length of the LCS, and with the decrease of alphabet size the length of the LCS increases for any given input size.

3.2 Real-world Sequences. In Table 1 we tabulate running times on the AMD Opteron for pairs of sequences from the CFTR DNA sequences [25], where again, our algorithm performs approximately twice as fast as Hirschberg's algorithm.

Acknowledgement. We would like to thank Vijaya Ramachandran for her support, encouragement and valuable comments. We also thank Mike Brudno for providing us with the CFTR DNA sequences.

References

[1] A. Aggarwal and J.S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31:1116–1127, September 1988.

[2] A.V. Aho, D.S. Hirschberg, and J.D. Ullman. Bounds on the complexity of the longest common subsequence problem. *Journal of the ACM*, 23(1):1–12, 1976.

[3] A. Apostolico, S. Browne, and C. Guerra. Fast linear-space computations of longest common subsequences. *Theoretical Computer Science*, pp. 3–17, 1992.

[4] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the 7th International Symposium on String Information Retrieval* , pp. 39–48, 2000.

[5] C. Cherng and R.E. Ladner. Cache efficient simple dynamic programming. In *Proceedings of the International Conference on the Analysis of Algorithms*, pp. 49–58, Barcelona, Spain, June 2005.

[6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[7] R.A. Chowdhury and V. Ramachandran. Cache-Oblivious Dynamic Programming. To appear in the *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms*, Miami, Florida, January 2006.

[8] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[9] M. Frigo and V. Strumpen. Cache-oblivious stencil computations. In *Proceedings of the 19th ACM International Conference on Supercomputing*, Cambridge, Massachusetts, USA, June 2005.

[10] D. Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, New York, 1997.

[11] D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[12] D. S. Hirschberg. An information theoretic lower bound for the longest common subsequence problem. *Information Processing Letters*, 7(1):40–41, 1978.

[13] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms* W. H. Freeman & Co., New York, 1998.

[14] J.W. Hunt and M.D. McIlroy. An algorithm for differential file comparison. CSTR #41, Bell Laboratories, Murray Hill, New Jersey, June 1976.

[15] S.K. Kumar and C.P. Rangan. A linear-space algorithm for the LCS problem. *Acta Informatica*, 24:353–362, 1987.

[16] J. Kleinberg and E. Tardos. *Algorithm Design.* Addison-Wesely Publishing Co., Reading, MA, 2005.

[17] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, 1978.

[18] E.W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

[19] W. J. Masek, M.S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[20] C.N.S. Pedersen. *Algorithms in Computational Biology.* PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.

[21] J.-S. Park, M. Penner and V.K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, vol. 15(9), pp. 769–782, 2004.

[22] D. Sankoff and J.B. Kruskal. *Time Wraps, String Edits, and Macro-molecules: the Theory and Practice of Sequence Comparison.* Addison-Wesely Publishing Co., Reading, MA, 1983.

[23] J. Seward and N. Nethercote. Valgrind (debugging and profiling tool for x86-Linux programs). http://valgrind.kde.org/index.html

[24] I. Simon. Sequence comparison: some theory and some practice. In M. Gross and D. Perrin, editors, *Electronic Dictionaries and Automata in Computational Linguistics*, LNCS 377, pp. 79–92, Berlin, 1989.

[25] J.W. Thomas, J.W. Touchman, R.W. Blakesley, G.G. Bouffard, S.M. Beckstrom-Sternberg, E.H. Margulies, M. Blanchette, A.C. Siepel, P.J. Thomas, J.C. McDowell, B. Maskeri, N.F. Hansen, M.S. Schwartz, R.J. Weber, W.J. Kent, D. Karolchik, T.C. Bruen, R. Bevan, D.J. Cutler, S. Schwartz, L. Elnitski, J.R. Idol, A.B. Prasad, S.-Q. Lee-Lin, V.V.B. Maduro, T.J. Summers, M.E. Portnoy, N.L. Dietrich, N. Akhter, K. Ayele, B. Benjamin, K. Cariaga, C.P. Brinkley, S.Y. Brooks, S. Granite, X. Guan, J. Gupta, P. Haghihi, S.-L. Ho, M.C. Huang, E. Karlins, P.L. Laric, R. Legaspi, M.J. Lim, Q.L. Maduro, C.A. Masiello, S.D. Mastrian, J.C. McCloskey, R. Pearson, S. Stantripop, E.E. Tiongson, J.T. Tran, C. Tsurgeon, J.L. Vogt, M.A. Walker, K.D. Wetherby, L.S. Wiggins, A.C. Young, L.-H. Zhang, K. Osoegawa, B. Zhu, B. Zhao, C.L. Shu, P.J. De Jong, C.E. Lawrence, A.F. Smit, A. Chakravarti, D. Haussler, P. Green, W. Miller, and E.D. Green. Comparative analyses of multi-species sequences from targeted genomic regions. *Nature*, vol. 424, pp. 788–793, 2003.

[26] M.S. Waterman. *Introduction to Computational Biology* Chapman & Hall, London, UK, 1995.