Cache-Oblivious Dynamic Programming *

Rezaul Alam Chowdhury Vijaya Ramachandran

Abstract

We present efficient cache-oblivious algorithms for several fundamental dynamic programs. These include new algorithms with improved cache performance for *longest common subsequence* (*LCS*), *edit distance*, *gap* (i.e., edit distance with gaps), and *least weight subsequence*. We present a new cache-oblivious framework called the *Gaussian Elimination Paradigm* (*GEP*) for Gaussian elimination without pivoting that also gives cache-oblivious algorithms for Floyd-Warshall all-pairs shortest paths in graphs and *'simple DP'*, among other problems.

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 *cache-oblivious model* [9] is an extension of this model with the additional feature that algorithms do not use 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 possible ('spatial locality'), and also that as much useful work as possible is performed on this data before it is written back to external memory ('temporal locality').

Dynamic programming is a widely-used algorithmic technique [3, 21, 7]. However, standard implementations of these algorithms often fail to exploit the temporal locality of data which leads to poor I/O performance. In this paper we present several results that significantly improve the I/O complexity of dynamic programming algorithms through techniques that take full advantage of both spatial and temporal locality.

The problem of finding the *longest common subsequence* (*LCS*) of two sequences has a classic dynamic programming solution [7] that runs in $\Theta(mn)$ time, uses $\Theta(mn)$ space and performs $\Theta\left(\frac{mn}{B}\right)$ I/Os when working on two sequences of lengths m and n. Linear space implementations of this algorithm [13, 18, 2] also have I/O complexity $\Omega\left(\frac{mn}{B}\right)$. The LCS problem arises in a wide range of applications, and is especially prominent in computational biology in *sequence alignment*.

We present a cache-oblivious implementation of the classic dynamic programming LCS algorithm. Our algorithm continues to run in $\mathcal{O}(mn)$ time, but uses $\mathcal{O}\left(m+n\right)$ space and performs only $\mathcal{O}\left(\frac{mn}{BM}\right)$ block transfers. Experimental results show that this algorithm runs two to six times faster than the widely used linear-space LCS algorithm by Hirschberg [13]. 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 [17, 7] 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.

We also consider the *gap* problem [11, 12, 26] which is a natural generalization of the edit distance problem, and which arises in molecular biology, geology, and speech recognition. Unlike the edit distance problem, however, in this problem a sequence of inserts (or deletes) is treated as a single event and is assigned a cost that is not necessarily equal to the sum of the costs of the individual inserts (or deletes) in the sequence. For $m = n$, the standard dynamic programming solution runs in $\mathcal{O}(n^3)$ time, uses $\mathcal{O}(n^2)$ space and incurs $\mathcal{O}\left(\frac{n^3}{B}\right)$ I/Os. We present a cache-oblivious algorithm that incurs only $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/Os without changing the time and space complexities. The *least weight*

*Department of Computer Sciences, University of Texas, Austin, TX 78712. Email: {shaikat,vlr}@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.

subsequence problem [15, 12] can be viewed as a 1-dimensional version of the gap problem, and we present a cache-oblivious algorithm that runs in $\mathcal{O}(n^2)$ time and $\mathcal{O}\left(\frac{n^2}{BM}\right)$ I/Os under some natural assumptions.

Finally we introduce a general cache-oblivious framework, which we call *GEP* or the *Gaussian Elimination Paradigm*, for problems that can be solved using a construct similar to the computation in Gaussian elimination without pivoting. Standard implementations of these algorithms run in $\mathcal{O}(n^3)$ time, use $\mathcal{O}(n^2)$ space and incur $\mathcal{O}\left(\frac{n^3}{B}\right)$ I/Os. We give a general cache-oblivious implementation for GEP that incurs only $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/Os without changing its running time and space complexity. We use GEP to obtain a cache-oblivious algorithm for Gaussian elimination without pivoting. Our algorithm is in-place, and is arguably simpler than the known cache-oblivious algorithms for solving this problem [27, 4], since it is not based on LU decomposition and does not perform matrix multiplication. We also show that GEP not only gives the cache-oblivious Gaussian elimination algorithm, but it also gives cache-oblivious algorithms for LU decomposition without pivoting, Floyd-Warshall's APSP [8, 25], matrix multiplication, and sequence alignment with gaps; with some modification, it also gives a cache-oblivious algorithm for a class of dynamic programs termed as 'simple-DP' [6] which includes dynamic programming algorithms for RNA secondary structure prediction [19], matrix chain multiplication and optimal binary search trees. The I/O-complexity of each of these algorithms matches the best I/O bound known for the corresponding problem.

Related Work. The linear-space LCS algorithm of Hirschberg [13], when analyzed as a cache-oblivious algorithm, performs $\mathcal{O}\left(\frac{n^2}{B}\right)$ block transfers. While this is considerably better than the naïve bound of $\mathcal{O}(n^2)$ it is considerably larger than $\mathcal{O}\left(\frac{n^2}{BM}\right)$, which is the bound we achieve. If only the length of the LCS is needed, the technique for stencil computation [10] can achieve the same bound as our algorithm. However, that technique does not extend to computing an actual sequence.

Known cache-oblivious algorithms for Gaussian elimination for solving systems of linear equations are based on LU decomposition. In [27, 4] cache-oblivious algorithms performing $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O operations are given for LU decomposition without pivoting, while the algorithm in [23] performs LU decomposition with partial pivoting within the same I/O bound. These algorithms use matrix multiplication and solution of triangular linear systems as subroutines.

In [6], an $\mathcal{O}(n^3)$ time and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O cache-oblivious algorithm based on Valiant's context-free language recognition algorithm [24], is given for simple-DP.

A cache-oblivious algorithm for Floyd-Warshall's APSP algorithm is given in [20]. The algorithm runs in $\mathcal{O}(n^3)$ time and incurs $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache misses.

The rest of the paper is organized as follows. In section 2 we describe and analyze our cache-oblivious algorithm for the LCS problem and present some experimental results. We consider the gap problem in section 3. Finally in section 4 we introduce GEP and show its use in designing cache-oblivious algorithms for various problems.

2 Longest Common Subsequence

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. In the *Longest Common Subsequence* (LCS) problem we are given two sequences X and Y, and we need to find a maximum-length common subsequence of X and Y.

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 $\langle y_1, y_2, \ldots y_j \rangle$. Then $c[m, n]$ is the length of an LCS of X and Y, and can be computed using the following recurrence relation (see, e.g., [7]):

(equation 2.1)

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

The classic dynamic programming solution to the LCS problem 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, since each $c[i, j]$ depends on only $c[i-1, j-1]$, $c[i, j-1]$ and $c[i-1, j]$, 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. The forward pass of this algorithm can be implemented in $\mathcal{O}(\min(m, n))$ space, and thus the length of an LCS can be computed in linear space, but the algorithm needs $\Theta(mn)$ space to compute an actual LCS sequence. Hirschberg [13] gives an $\mathcal{O}(\min(m, n))$ space algorithm, which finds an LCS in $\mathcal{O}\left(\frac{mn}{B}\right)$ I/Os.

LCS-OUTPUT-BOUNDARY(X', Y', T, L)

Input. The top and the left input boundaries of $Q[1\ldots r, 1\ldots r]$ are stored in T and L, respectively, where $r = |X'| = |Y'| = 2^q$ for some nonnegative integer $q \le p$, and $Q[1\ldots r, 1\ldots r] = c[k\ldots k+r-1, l\ldots l+r-1]$, $X' = X[k\ldots k+r-1]$ and $Y' = Y[l\ldots l+r-1]$ for some k and l $(1 \le k, l \le n-r+1)$.

Output. The output is (B, R) where B (R) is the bottom (resp. right) output boundary of $Q[1\ldots r, 1\ldots r]$.

if $r = 1$ ***then*** compute the output boundary directly using equation 2.1
else
    $(B_{11}, R_{11}) \leftarrow$ LCS-OUTPUT-BOUNDARY(X'_1, Y'_1, T_1, L_1) {*process* Q_{11}}
    $(B_{12}, R_{12}) \leftarrow$ LCS-OUTPUT-BOUNDARY$(X'_2, Y'_1, T_2, R_{11})$ {*process* Q_{12}}
    $(B_{21}, R_{21}) \leftarrow$ LCS-OUTPUT-BOUNDARY$(X'_1, Y'_2, B_{11}, L_2)$ {*process* Q_{21}}
    $(B_{22}, R_{22}) \leftarrow$ LCS-OUTPUT-BOUNDARY$(X'_2, Y'_2, B_{12}, R_{21})$ {*process* Q_{22}}
return $(B_{21} \# B_{22}, R_{12} \# R_{22})$

Figure 1: Computing output boundary of an LCS matrix cache-obliviously.

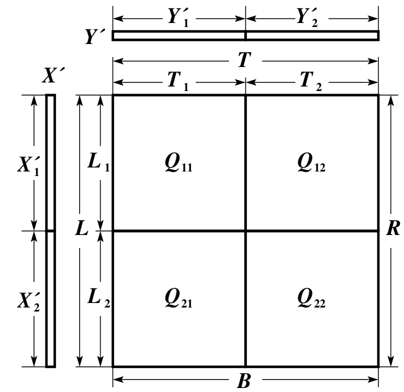

Figure 2: Dividing a square LCS matrix Q into quadrants (Q_{11}, Q_{12}, Q_{21} and Q_{22}).

RECURSIVE-LCS(X', Y', T, L, i, j)

Input. The top and the left input boundaries of $Q[1\ldots r, 1\ldots r]$ are stored in T and L, respectively, where $r = |X'| = |Y'| = 2^q$ for some nonnegative integer $q \le p$, and $Q[1\ldots r, 1\ldots r] = c[k\ldots k+r-1, l\ldots l+r-1]$, $X' = X[k\ldots k+r-1]$ and $Y' = Y[l\ldots l+r-1]$ for some k and l $(1 \le k, l \le n-r+1)$. The entry $c[i, j]$ lies on the output boundary of Q.

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

$Z \leftarrow \emptyset$
if $r = 1$ ***then*** set $Z \leftarrow X'[1]$ if $X'[1] = Y'[1]$ and return the appropriate values for i and j
else
    $(B_{11}, R_{11}) \leftarrow$ LCS-OUTPUT-BOUNDARY(X'_1, Y'_1, T_1, L_1) {*generate output boundary of* Q_{11}}
    if $(i, j) \in$ output-boundary(Q_{22}) ***then*** {*if the LCS intersects* Q_{22}}
        $(B_{12}, R_{12}) \leftarrow$ LCS-OUTPUT-BOUNDARY$(X'_2, Y'_1, T_2, R_{11})$ {*generate output boundary of* Q_{12}}
        $(B_{21}, R_{21}) \leftarrow$ LCS-OUTPUT-BOUNDARY$(X'_1, Y'_2, B_{11}, L_2)$ {*generate output boundary of* Q_{21}}
        $Z \leftarrow$ RECURSIVE-LCS$(X'_2, Y'_2, B_{12}, R_{21}, i, j)$ $\#$ Z {*find LCS fragment in* Q_{22}}
    if $(i, j) \in$ output-boundary(Q_{12}) ***then*** $Z \leftarrow$ RECURSIVE-LCS$(X'_2, Y'_1, T_2, R_{11}, i, j)$ $\#$ Z {*find LCS fragment in* Q_{12}}
    if $(i, j) \in$ output-boundary(Q_{21}) ***then*** $Z \leftarrow$ RECURSIVE-LCS$(X'_1, Y'_2, B_{11}, L_2, i, j)$ $\#$ Z {*find LCS fragment in* Q_{21}}
    if $(i, j) \in$ output-boundary(Q_{11}) ***then*** $Z \leftarrow$ RECURSIVE-LCS$(X'_2, Y'_2, T_1, L_1, i, j)$ $\#$ Z {*find LCS fragment in* Q_{11}}
return Z

Figure 3: Cache-Oblivious computation of an LCS.

In this section we present an optimal cache-oblivious implementation of the LCS dynamic program. Our algorithm uses a procedure LCS-OUTPUT-BOUNDARY that computes LCS-lengths at the 'boundary' of the subproblem being considered. This procedure is used in an algorithm RECURSIVE-LCS that computes an actual LCS. The algorithm performs $\mathcal{O}\left(\frac{mn}{BM}\right)$ I/Os, and we also show that this is optimal for any implementation of the dynamic programming algorithm for LCS. In the following, for convenience we assume that $n = m = 2^p$ where p is a nonnegative integer; the two input sequences are X and Y.

Cache-oblivious LCS Output Boundary. We can compute all entries of a submatrix $c[i_1\ldots i_2, j_1\ldots j_2]$ of c provided we know the entries of $c[i_1-1, j_1\ldots j_2]$ and $c[i_1\ldots i_2, j_1-1]$, 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 Figure 1) when called with sub-sequences X' and Y' where $|X'| = |Y'| = r = 2^q$ for some nonnegative integer $q \le p$, together with costs for the top and the left input boundaries (T and L, respectively) of the corresponding LCS submatrix, computes the output boundary of that submatrix. If $r = 1$, the function

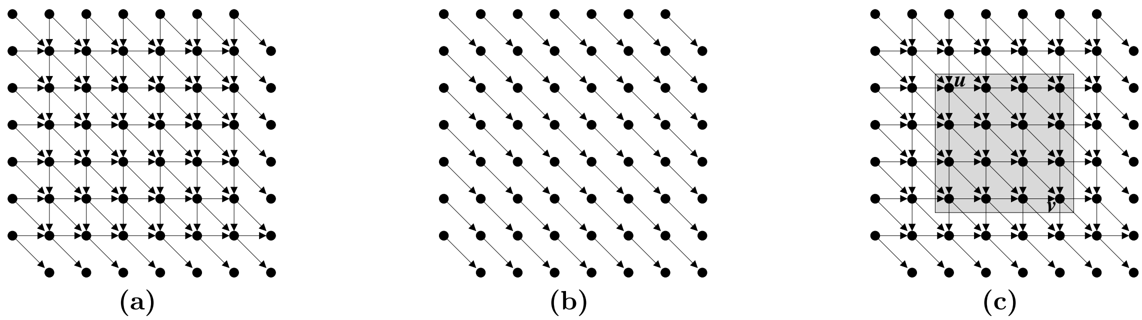

Figure 4: (*I/O lower bound for LCS*) **(a)** Computational DAG of Recursive-LCS, **(b)** Vertex-disjoint paths (lines) from inputs to outputs, **(c)** Computational subDAG (shaded) defined by two nodes at distance $d = 3$ (the two endpoints u and v of the left to right diagonal of the subDAG).

computes the output boundary directly using equation 2.1, otherwise it divides the input submatrix into four quadrants, and recursively computes the output boundary of each quadrant in the following order: top-left, top-right/bottom-left, bottom-right. This sequence ensures that the input boundary of each quadrant has already been computed before processing that quadrant. It then computes the output boundary of the input submatrix by appropriately combining the output boundaries of the quadrants. The initial call is LCS-Output-Boundary(X, Y, T, L), with $T[0\ldots n]$ and $L[0\ldots n]$ initialized to all zeros.

Analysis. Let $I_1(n)$ be the I/O complexity of LCS-Output-Boundary on an input of size n. Then

$$I_1(n) = \begin{cases} \mathcal{O}\left(1 + \frac{n}{B}\right) & \text{if } n \le \alpha M, \\ 4I_1\left(\frac{n}{2}\right) + \mathcal{O}(1) & \text{otherwise;} \end{cases}$$

where α is the largest constant sufficiently small that an input of size αM fits completely in the cache. Hence $I_1(n) = \mathcal{O}\left(1 + \frac{n}{B} + \frac{n^2}{BM}\right)$. The algorithm runs in $\mathcal{O}(n^2)$ time while using $\mathcal{O}(n)$ space.

Computing the Elements of an LCS. 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*.

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 in Figure 3) to construct the LCS. This function is called with parameters X', Y', T, L, i and j where the first four parameters are the same as those of LCS-Output-Boundary, and the last two are indices such that an LCS path intersects the output boundary of $c[k\ldots k+r-1, l\ldots l+r-1]$ at $c[i, j]$. This function traces the fragment of that path through this submatrix, and returns the LCS Z of X and Y along this subpath. It also finds the entry at which this path intersects the input boundary of the given submatrix, and updates i and j to point to that entry, respectively. If $r = 1$, it computes Z by comparing $X'[1]$ and $Y'[1]$, 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 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 and j (i.e., based on which quadrant $c[i, j]$ 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 and j). The output of Recursive-LCS is the concatenation of these LCS fragments in the correct order.

The initial call is Recursive-LCS(X, Y, T, L, n, n), with $T[0\ldots n]$ and $L[0\ldots n]$ initialized to all zeros.

Analysis. Recall that I_1 represents the number of I/Os performed by LCS-Output-Boundary. Let $I_2(n)$ be the I/O complexity of Recursive-LCS on an input of size n. Since an LCS path can intersect at most three quadrants of the input submatrix, we have:

$$I_2(n) \le \begin{cases} \mathcal{O}\left(1 + \frac{n}{B}\right) & \text{if } n \le \alpha M, \\ 3\left(I_1\left(\frac{n}{2}\right) + I_2\left(\frac{n}{2}\right)\right) + \mathcal{O}\left(1 + \frac{n}{B}\right) & \text{otherwise;} \end{cases}$$

where α is the largest constant sufficiently small that an input of size αM fits completely in the cache. Hence $I_2(n) = \mathcal{O}\left(1 + \frac{n}{B} + \frac{n^2}{BM}\right)$. The running time and space remain $\mathcal{O}(n^2)$ and $\mathcal{O}(n)$, respectively.

For input strings of unequal lengths m and n, and of lengths that are not powers of 2, this algorithm can

n	Intel Xeon CO (c_1)	Intel Xeon Hi (c_2)	Intel Xeon ratio $\left(\frac{c_2}{c_1}\right)$	Sun Blade 2000/1000 CO (c_1)	Sun Blade 2000/1000 Hi (c_2)	Sun Blade 2000/1000 ratio $\left(\frac{c_2}{c_1}\right)$
8,192	0.2	32.6	163	9	20	2
16,384	0.5	136.8	274	19	469	25
32,768	1.4	559.0	399	39	3,037	78
65,536	3.9	2,262	580	79	14,736	187
131,072	12.5	9,084	727	159	64,250	404
262,144	42.9	36,426.9	849	369	267,741	725
524,288	156.7	146,095.1	932	825	1,091,071	1,323
1,048,576	-	-	-	2,033	7,803,878	3,839

Table 1: L1 misses (in millions) on Intel Xeon and Sun Blade. The figures are averages of 5 runs on two equal-length random sequences from an alphabet of size 26.

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

Table 2: Running time on AMD Opteron. The figures give the time for a single run on pairs of CFTR DNA sequences.

be extended to perform $\mathcal{O}\left(\frac{mn}{BM}\right)$ I/Os while running in $\mathcal{O}(mn)$ time using $\mathcal{O}(m+n)$ space.

Lower Bound. We prove that any algorithm that executes mn operations in order to implement the type of computation defined by equation 2.1 (i.e., compute $c[m,n]$), must perform $\Omega\left(1+\frac{m+n}{B}+\frac{mn}{BM}\right)$ I/Os. We use the *red-blue pebble game technique* [16] in order to obtain this lower bound. First we construct a *computation DAG G* given by the computation of the algorithm. Figure 4(a) shows an example of the computation DAG given by equation 2.1. Nodes in G represents operations, and edges represent the data-flow of the algorithm. Nodes with no incoming edges are *inputs* and those with no outgoing edges are *outputs*. In Figure 4(a) the nodes in the top row and the far left column represent inputs, and those in the bottom row and the far right column represent outputs. In this Figure the output nodes form an extra lair, and the edges to the output nodes represents simple copy operations. Figure 4(b) shows a decomposition of G into vertex-disjoint paths from inputs to outputs. These paths are called *lines*. Now for any two nodes u and v in G that lie at least a distance d apart on the same line, the *information speed function* $F_G(d)$ is defined as the number of nodes in G no two of which lie on the same line and each of which belongs to a path connecting u and v. The following theorem gives a lower bound on the number of I/O operations Q required to execute G.

Theorem [16]. For any graph G where all inputs can reach all outputs through vertex-disjoint paths, if the information speed function is $\Omega(F(d))$, where F is monotonically increasing and F^{-1} exists, then $Q \cdot F^{-1}(M) = \Omega(L)$, where L is the total number of vertices on the vertex-disjoint paths or lines.

This theorem assumes that data is transferred to the cache in blocks of size 1. Therefore, for block size B, we will have, $Q = \Omega\left(\frac{L}{BF^{-1}(M)}\right)$. In Figure 4(b) we decomposed G into $\Theta(m+n)$ lines, and $L=\Theta(mn)$. We have $F(d)=\Omega(d)$ (see Figure 4(c)). The inverse of $F(d)$ exists, and $F^{-1}(d)=\mathcal{O}(d)$. Therefore, $Q=\Omega\left(\frac{mn}{BM}\right)$.

Since the algorithm must also read all $\Theta(m+n)$ inputs, the I/O lower bound is $\Omega\left(1+\frac{m+n}{B}+\frac{mn}{BM}\right)$.

2.1 Experimental Results.

We implemented three variants of our algorithm: (*i*) 4-way partitioning as described above, (*ii*) 2-way partitioning along the longer dimension, and (*iii*) triangular partitioning. All three methods have the same asymptotic bounds, but the triangular partitioning gave the best performance experimentally. For comparison we coded the widely used linear-space algorithm of Hirschberg [14]. Both algorithms were tested on both random and real-world 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+. Detailed results of our experiments can be found in [5]. Below we summarize our results, where *CO* and *Hi* denote the new cache-oblivious algorithm and Hirschberg's algorithm, respectively:

- CO incurred considerably fewer cache misses compared to Hi. In Table 1 we tabulate the L1 cache misses incurred by the algorithms on the Intel Xeon and the Sun UltraSPARC-III+.
- CO ran a factor of 2 to 6 times faster than Hi on random sequences. In Table 2 we tabulate running times on the AMD Opteron for CFTR DNA sequences [22], where again, CO performs approximately twice as fast as Hi.
- CO executed 40%-50% fewer 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.

Recursive-Gap(C)

(We assume that C is a square submatrix of D, and the top-left cell of C corresponds to $D[i, j]$ for some $0 \leq j \leq n$. We also assume that the dimension of C is a power of 2. This function recursively computes the entries of C according to equation 3.2.)

if C is a 1×1 matrix ***then*** $D[i, j] \leftarrow \min(D[i, j], D[i-1, j-1])$ ***else***
    Recursive-Gap(C_{11}) {*compute top-left quadrant*}
    Apply-E(C_{12}, C_{11}), Recursive-Gap(C_{12}) {*compute top-right quadrant*}
    Apply-F(C_{21}, C_{11}), Recursive-Gap(C_{21}) {*compute bottom-left quadrant*}
    Apply-E(C_{22}, C_{12}), Apply-F(C_{22}, C_{21}), Recursive-Gap(C_{22}) {*compute bottom-right quadrant*}

Apply-E(A, B)

(A and B are two non-overlapping $2^t \times 2^t$ submatrices of D, where t is a nonnegative integer. $A[1, 1]$ corresponds to $D[i, j]$, and $B[1, 1]$ corresponds to $D[i, q]$ for some $0 \leq i \leq n$ and $0 \leq q < j \leq n$. This function updates the entries of A using the entries of B according to the equation defining $E[i, j]$.)

if $t = 0$ ***then*** $D[i, j] \leftarrow \min(D[i, j], D[i, q] + w(q, j))$
else {*update in the order:* $A_{11}, A_{12}, A_{21}, A_{22}$}
    Apply-E(A_{11}, B_{11}), Apply-E(A_{11}, B_{12})
    Apply-E(A_{12}, B_{11}), Apply-E(A_{12}, B_{12})
    Apply-E(A_{21}, B_{21}), Apply-E(A_{21}, B_{22})
    Apply-E(A_{22}, B_{21}), Apply-E(A_{22}, B_{22})

Apply-F(A, B)

(A and B are two non-overlapping $2^t \times 2^t$ submatrices of D, where t is a nonnegative integer. $A[1, 1]$ corresponds to $D[i, j]$, and $B[1, 1]$ corresponds to $D[p, j]$ for some $0 \leq p < i \leq n$ and $0 \leq j \leq n$. This function updates the entries of A using the entries of B according to the equation defining $F[i, j]$.)

if $t = 0$ ***then*** $D[i, j] \leftarrow \min(D[i, j], D[p, j] + w'(p, i))$
else {*update in the order:* $A_{11}, A_{12}, A_{21}, A_{22}$}
    Apply-F(A_{11}, B_{11}), Apply-F(A_{11}, B_{21})
    Apply-F(A_{12}, B_{12}), Apply-F(A_{12}, B_{22})
    Apply-F(A_{21}, B_{11}), Apply-F(A_{21}, B_{21})
    Apply-F(A_{22}, B_{12}), Apply-F(A_{22}, B_{22})

Figure 5: Cache-oblivious algorithm for the gap problem.

2.2 The Edit Distance Problem. Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ over a finite alphabet Σ, the *edit distance* of X and Y is the minimum cost of an *edit sequence* that transforms X into Y [17]. The edit operations are: **delete**(x_i) of cost $D(x_i)$ that deletes x_i from X, **insert**(y_j) of cost $I(y_j)$ that inserts y_j into X, and **substitute**(x_i, y_j) of cost $S(x_i, y_j)$ that replaces x_i with y_j in X.

Our LCS algorithm can be adopted directly to solve this problem cache-obliviously in $\mathcal{O}(mn)$ time and $\mathcal{O}\left(\frac{mn}{BM}\right)$ I/Os provided that Σ is small enough so that S can be stored in internal memory, or $S(x_i, y_j)$ can be computed on the fly in constant time and without incurring any additional cache misses.

3 The Gap Problem

The *gap* problem [11, 12, 26] is a generalization of the edit distance problem that arises in molecular biology, geology, and speech recognition. When transforming a string $X = x_1 x_2 \ldots x_m$ into another string $Y = y_1 y_2 \ldots y_n$, a sequence of consecutive deletes corresponds to a gap in X, and a sequence of consecutive inserts corresponds to a gap in Y. In many applications the cost of such a gap is not necessarily equal to the sum of the costs of each individual deletion (or insertion) in that gap. In order to handle this general case two new cost functions w and w' are defined, where $w(p, q)$ ($0 \leq p < q \leq m$) is the cost of deleting $x_{p+1} \ldots x_q$ from X, and $w'(p, q)$ ($0 \leq p < q \leq n$) is the cost of inserting $y_{p+1} \ldots y_q$ into X. The substitution function $S(x_i, y_j)$ is the same as that of the standard edit distance problem.

Let $D[i, j]$ denote the minimum cost of transforming $X_i = x_1 x_2 \ldots x_i$ into $Y_j = y_1 y_2 \ldots y_j$ (where $0 \leq i \leq m$ and $0 \leq j \leq n$) under this general setting. Then

(equation 3.2)
$$D[i, j] = \begin{cases} 0 & \text{if } i = j = 0, \\ w(0, j) & \text{if } i = 0,\ 1 \leq j \leq n, \\ w'(0, i) & \text{if } j = 0,\ 1 \leq i \leq m, \\ G[i, j] & \text{if } i, j > 0; \end{cases}$$

where $G[i, j] = \min(D[i-1, j] + S(x_i, y_j), E[i, j], F[i, j])$, $E[i, j] = \min_{0 \leq q < j} \{D[i, q] + w(q, j)\}$ and $F[i, j] = \min_{0 \leq p < i} \{D[p, j] + w'(p, i)\}$.

Assuming $m = n$, this problem can be solved in internal memory [11] in $\mathcal{O}(n^3)$ time using $\mathcal{O}(n^2)$ space; this algorithm incurs $\Theta\left(\frac{n^3}{B}\right)$ I/Os.

Cache-Oblivious Algorithm. The Recursive-Gap function given in Figure 5 is a cache-oblivious algorithm for the gap problem. When called with matrix D it computes the entries of D recursively assuming that $m = n = 2^p$ for some integer $p \geq 0$. It splits the input matrix C into four quadrants, and solves each quadrant recursively in the following order: top-left (C_{11}), top-right (C_{12}), bottom-left (C_{21}) and bottom-right (C_{22}). This order of processing ensures that no entry $D[i, j]$ is

1. ***for*** $k \leftarrow \kappa_1$ ***to*** κ_2 ***do*** $\{n \geq \kappa_2 \geq \kappa_1 \geq 1\}$
2. ***for*** $i \leftarrow \iota_1(k)$ ***to*** $\iota_2(k)$ ***do*** $\{n \geq \iota_2(k) \geq \iota_1(k) \geq 1\}$
3. ***for*** $j \leftarrow \zeta_1(k,i)$ ***to*** $\zeta_2(k,i)$ ***do*** $\{n \geq \zeta_2(k,i) \geq \zeta_1(k,i) \geq 1\}$
4. $c[i,j] \leftarrow f(c[i,j], c[i,k], c[k,j], c[k,k])$ $\{$*a problem specific function*$\}$

Figure 6: General dynamic program in GEP.

$\mathrm{F}(X, k_1, k_2)$ {F can be any of the nine functions $(A, B_1, B_2, C_1, C_2, D_1, D_2, D_3, D_4)$ in column 1 of Figure 9.}

(X is a square submatrix of c such that $X[1,1] = c[i_1, j_1]$ and $X[2^q, 2^q] = c[i_2, j_2]$ where $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ for some integer $q \geq 0$. The top-left, top-right, bottom-left and bottom-right quadrants of X are denoted by X_{11}, X_{12}, X_{21} and X_{22}, respectively. The function calls in lines 4 and 5 are determined from Figure 9.)

1. ***if*** $k_1 = k_2$ ***then*** $c[i_1, j_1] \leftarrow f(c[i_1, j_1], c[i_1, k_1], c[k_1, j_1], c[k_1, k_1])$
2. ***else***
3. $k_m \leftarrow \lfloor \frac{k_1 + k_2}{2} \rfloor$
4. $F_{11}(X_{11}, k_1, k_m)$, $F_{12}(X_{12}, k_1, k_m)$, $F_{21}(X_{21}, k_1, k_m)$, $F_{22}(X_{22}, k_1, k_m)$ $\{$*forward pass*$\}$
5. $F'_{22}(X_{22}, k_m + 1, k_2)$, $F'_{21}(X_{21}, k_m + 1, k_2)$, $F'_{12}(X_{12}, k_m + 1, k_2)$, $F'_{11}(X_{11}, k_m + 1, k_2)$ $\{$*backward pass*$\}$

Figure 7: Cache-oblivious implementation of the dynamic program in Figure 6.

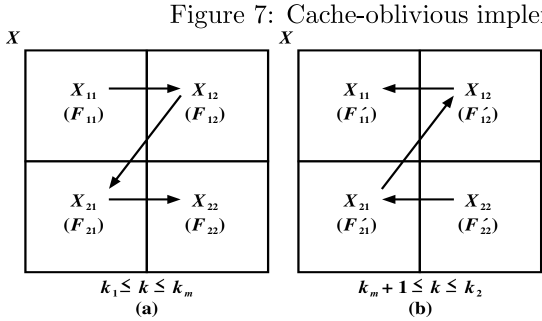

Figure 8: Processing order of quadrants of X by F (processing function in parentheses): **(a)** forward pass, **(b)** backward pass.

F	F_{11}	F_{12}	F_{21}	F_{22}	F'_{22}	F'_{21}	F'_{12}	F'_{11}
A	A	B_1	C_1	D_1	A	B_2	C_2	D_4
B_i $(i = 1, 2)$	B_i	B_i	D_i	D_i	B_i	B_i	D_{i+2}	D_{i+2}
C_i $(i = 1, 2)$	C_i	D_{2i-1}	C_i	D_{2i-1}	C_i	D_{2i}	C_i	D_{2i}
D_i $(i \in [1, 4])$	D_i	D_i	D_i	D_i	D_i	D_i	D_i	D_i

Figure 9: Function specific recursive calls.

F	A	B_1	B_2	C_1	C_2	D_1	D_2	D_3	D_4
$P(F)$	$i_1 = k_1$ $\wedge\ j_1 = k_1$	$i_1 = k_1$ $\wedge\ j_1 > k_2$	$i_1 = k_1$ $\wedge\ j_2 < k_1$	$i_1 > k_2$ $\wedge\ j_1 = k_1$	$i_2 < k_1$ $\wedge\ j_1 = k_1$	$i_1 > k_2$ $\wedge\ j_1 > k_2$	$i_1 > k_2$ $\wedge\ j_2 < k_1$	$i_2 < k_1$ $\wedge\ j_1 > k_2$	$i_2 < k_1$ $\wedge\ j_2 < k_1$

Figure 10: Function specific pre-condition $P(F)$.

used to update some other entry before completing the computation of $D[i,j]$ itself. The function Apply-E (Apply-F) is used to update the entries of a quadrant using the entries of another one based on the equation defining $E[i,j]$ (resp. $F[i,j]$).

I/O Complexity. Let $I(n)$ and $I'(n)$ be the I/O complexities of Recursive-Gap and Apply-E/Apply-F, respectively, on an input of size $n \times n$. Then

$$I'(n) = \begin{cases} \mathcal{O}(n + \frac{n^2}{B}) & \text{if } n^2 \leq \alpha' M, \\ 8I'\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases}$$

$$\text{and} \quad I(n) = \begin{cases} \mathcal{O}(n + \frac{n^2}{B}) & \text{if } n^2 \leq \alpha M, \\ 4I\left(\frac{n}{2}\right) + 4I'\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases}$$

where α' and α are suitable constants. Solving the recurrences we obtain $I(n) = \mathcal{O}\left(\frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$, assuming a tall cache, i.e., $M = \Omega(B^2)$.

The Least Weight Subsequence Problem. The *least weight subsequence* problem [15, 12] which arises in optimum paragraph formation and in finding minimum height B-trees, can be solved cache-obliviously in $\mathcal{O}(n^2)$ time, $\mathcal{O}(n)$ space and $\mathcal{O}(\frac{n^2}{BM})$ I/Os using a 1-dimensional version of Recursive-Gap, where n is the length of the input sequence.

4 The Gaussian Elimination Paradigm (GEP)

In this section we present a general cache-oblivious framework for problems that can be solved using a triply-nested ***for*** loop as shown in Figure 6. In view of the structural similarity between this construct and the computation in Gaussian elimination without piv-

$$c = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n-1} & b_1 \\ a_{2,1} & a_{2,2} & \dots & a_{2,n-1} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-1,1} & a_{n-1,2} & \dots & a_{n-1,n-1} & b_{n-1} \\ 0 & 0 & \dots & 0 & 0 \end{pmatrix}$$

(a)

1. ***for*** $k \leftarrow 1$ ***to*** $n-2$ ***do***
2. ***for*** $i \leftarrow k+1$ ***to*** $n-1$ ***do***
3. ***for*** $j \leftarrow k+1$ ***to*** n ***do***
4. $c[i,j] \leftarrow c[i,j] - \frac{c[i,k]}{c[k,k]} \times c[k,j]$

(b)

Figure 11: **(a)** System of equations as a matrix, **(b)** First phase of Gaussian elimination.

oting, we will refer to this paradigm as the *Gaussian Elimination Paradigm* or *GEP*. Many practical problems fall in this category, for example: all-pairs shortest paths, LU decomposition, and Gaussian elimination without pivoting. Other problems can be solved using GEP through structural transformation, for example: the gap problem, simple dynamic program [6], and matrix multiplication.

In the triply-nested loop of Figure 6 the range of i in step 2 is a function of k, and that of j in step 3 is a function of k and i. We assume w.l.o.g. that the smallest value taken by any of i, j or k is 1, and the largest value taken is $n = 2^p$ for some integer $p \geq 0$. Therefore, the running time of the dynamic program is $\mathcal{O}(n^3)$ provided f can be computed in constant time.

A General Cache-Oblivious Implementation. In Figure 7, we give a template for a recursive function F which can be instantiated to any of the nine functions (A, B_1, B_2, C_1, C_2, D_1, D_2, D_3, and D_4) given in Figure 9. The inputs to F are a square submatrix X of $c[1 \dots n, 1 \dots n]$, and two indices k_1 and k_2. The top-left cell of X corresponds to $c[i_1, j_1]$, and the bottom-right cell corresponds to $c[i_2, j_2]$. These indices satisfy the following constraints:

(a) $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ for some integer $q \geq 0$

(b) $[i_1, i_2] \neq [k_1, k_2] \Rightarrow [i_1, i_2] \cap [k_1, k_2] = \emptyset$ and $[j_1, j_2] \neq [k_1, k_2] \Rightarrow [j_1, j_2] \cap [k_1, k_2] = \emptyset$

(c) $P(F)$ (see Figure 10)

The base case occurs when $k_1 = k_2$, and the function updates $c[i_1, j_1]$ to $f(c[i_1, j_1], c[i_1, k_1], c[k_1, j_1], c[k_1, k_1])$. Otherwise it splits X into four quadrants (X_{11}, X_{12}, X_{21} and X_{22}), and recursively updates the entries in each quadrant in two passes: forward (line 4) and backward (line 5). The processing order of the quadrants and the processing function for each quadrant in each pass are shown in Figures 8 and 9, respectively. A given computation need not necessarily make all recursive calls in lines 4 and 5. Whether a specific recursive call to a function F' (say) will be made or not depends on $P(F')$ and the dynamic program at hand. For example, if the ***for*** loop in line 2 of Figure 6 starts with $i \leftarrow k$, then we do not make any recursive call to function C_2 since the indices of the dynamic program cannot satisfy $P(C_2)$.

The function call $A(c, 1, n)$ (F instantiated to A) solves the dynamic program in Figure 6.

I/O Complexity. Let $I(n)$ be an upper bound on the number of I/O operations performed by any of the nine functions on an input of size $n \times n$. Since all functions have the same base case, we have,

$$I(n) \leq \begin{cases} \mathcal{O}(n + \frac{n^2}{B}) & \text{if } n^2 \leq \alpha M, \\ 8I\left(\frac{n}{2}\right) & \text{otherwise;} \end{cases}$$

where α is the largest constant sufficiently small that four $\alpha M \times \alpha M$ submatrices fit in the cache. The solution to the recurrence is $I(n) = \mathcal{O}\left(\frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right) = \mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ (assuming a tall cache).

I/O Lower Bound. Later in this section we show that the problem of multiplying two $n \times n$ matrices (i.e., computing $C = A \times B$, where A, B and C are $n \times n$ matrices) can be cast as a problem in GEP. Since any algorithm that executes $\Theta(n^3)$ operations given by the definition of matrix multiplication ($C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$) must perform $\Omega\left(\frac{n^3}{B\sqrt{M}}\right)$ I/Os [16], the same lower bound holds for GEP, too.

Gaussian Elimination without Pivoting. Gaussian elimination without pivoting is used in the solution of systems of linear equations and LU decomposition of symmetric positive-definite or diagonally dominant real matrices [7]. We represent a system of $n-1$ equations in $n-1$ unknowns $(x_1, x_2, \dots, x_{n-1})$ using an $n \times n$ matrix c as shown in Figure 11(a), where the i'th $(1 \leq i < n)$ row represents the equation $a_{i,1}x_1 + a_{i,2}x_2 + \dots + a_{i,n-1}x_{n-1} = b_i$. The method proceeds in two phases. In the first phase, as shown in Figure 11(b), an upper triangular matrix is constructed from c by successive elimination of variables from the equations. This phase requires $\mathcal{O}(n^3)$ time. The loop in step 3 starts with $j = k+1$ instead of $j = k$ which does not affect the correctness of the algorithm, but is crucial for a recursive implementation. In the second phase, the values of the unknowns are determined from this

```
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            c[i, j] ← min (c[i, j], c[i, k] + c[k, j])
```

(a)

```
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            c[i, j] ← c[i, j] + a[i, k] × b[k, j]
```

(b)

Figure 12: **(a)** Floyd-Warshall's APSP algorithm, **(b)** Modified matrix multiplication algorithm.

```
for k ← 0 to n − 1 do
    for i ← 1 to n do
        for j ← 1 to n do
            if (i ≤ k ∧ j ≥ k + 1) ∨ (i ≥ k + 1 ∧ j ≤ k) then
                D[i, j] ← min(D[i, j],  D[i − 1, j − 1] + S(x_i, y_j))
                if i ≤ k and j ≥ k + 1 then
                    D[i, j] ← min(D[i, j],  D[i, k] + w(k, j))
                else D[i, j] ← min(D[i, j],  D[k, j] + w′(k, i))
```

(a)

```
for i ← 1 to n do D[i, i] ← x_i
for d ← 2 to n do
    for i ← 1 to n − d + 1 do
        j ← d + i − 1
        for k ← i to j − 1 do
            D[i, j] ← D[i, j] + D[i, k] · D[k + 1, j]
```

(b)

Figure 13: **(a)** Alternate DP for the gap problem, **(b)** The $\mathcal{O}(n^3)$ Simple DP algorithm.

matrix by back substitution. It is straight-forward to implement this phase in $\mathcal{O}(n^2)$ time and $\mathcal{O}\left(\frac{n^2}{B}\right)$ I/Os.

The first phase is an instantiation of the GEP dynamic program in Figure 6. Observe that we always have $i > k$ and $j > k$ when we reach line 4 in Figure 11(b). Therefore, comparing these indices with the preconditions in Figure 10, we can eliminate all recursive function calls except those to the following four: A, B_1, C_1 and D_1. Thus this phase can be executed in $\mathcal{O}(n^3)$ time incurring $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache misses by calling $A(c, 1, n)$ with the reduced set of function calls.

A similar method solves LU decomposition without pivoting within the same bounds. Both algorithms are in-place. Our algorithm for Gaussian elimination is arguably simpler than existing algorithms since it does not use LU decomposition as an intermediate step, and thus does not invoke subroutines for multiplying matrices or solving triangular linear systems, as is the case with other cache-oblivious algorithms for this problem [27, 4, 23].

All-pairs Shortest Paths. The Floyd-Warshall algorithm [8, 25] for computing all-pairs shortest paths (see Figure 12(a)) is another instance of a dynamic program in GEP. In this case, however, we cannot eliminate any recursive function calls, and a little inspection reveals that all 9 functions are actually the same. Therefore, we can take only one function, say A, and substitute A for each recursive call it makes. The resulting algorithm is exactly the same as that in [20].

Matrix Multiplication. We consider the problem of computing $C = A \times B$, where A, B and C are $n \times n$ matrices. Though standard matrix multiplication does not fall into GEP, it does after the small structural modification shown in Figure 12(b) (index k is in the outermost loop in the modified algorithm, while in the standard algorithm it is in the innermost loop). The cache-oblivious algorithm we obtain for this modified algorithm by applying the transformations in this section is similar to that obtained for Floyd-Warshall's algorithm except that we need to pass all three matrices (A, B and C) to the recursive functions instead of one.

The Gap Problem. We presented a cache-oblivious algorithm for the gap problem (section 3). Here we show that we can obtain an alternate cache-oblivious algorithm for this problem by casting it as a dynamic program in GEP. In Figure 13(a) we give an alternate implementation of equation 3.2 assuming that row 0 and column 0 of $D[i, j]$ have already been initialized. Though the dynamic program in Figure 13(a) does not exactly match the pattern given in Figure 6 (for example: some of the indices are off by a constant, and additional functions w, w' and S are used), the GEP method continues to apply, and we obtain an $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O cache-oblivious algorithm.

Simple Dynamic Programs. In [6], the term *Simple dynamic program* was used to denote a class of dynamic programming problems over a nonassociative semi-ring $(S, +, \cdot, 0)$ which can be solved in $\mathcal{O}(n^3)$ time using the dynamic program shown in Figure 13(b). Its applications include RNA secondary structure prediction, optimal matrix chain multiplication, construction of optimal binary search trees, and optimal polygon triangulation. An $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O cache-oblivious algorithm based on Valiant's context-free language recognition algorithm [24] was given in [6] for this class of problems.

We can transform a simple dynamic program into a dynamic program in GEP using the decomposition technique given in [12]. The upper triangular matrix D is decomposed into (forward) diagonal strips of horizontal width $n^{\frac{1}{4}}$, and the entries in D are computed one strip at a time starting from the largest strip. The computation for each strip involves min-plus matrix multiplication and dynamic programs that can be solved with GEP. The resulting algorithm runs in $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/Os.

Acknowledgement. We thank Mike Brudno for providing us with the CFTR DNA sequence data.

References

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

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

[3] R. Bellman. *Dynamic Programming*. The Princeton University Press, Princeton, New Jersey, 1957.

[4] R.D. Blumofe, M. Frigo, C.F. Joerg, C.E. Leiserson, and K.H. Randall. An analysis of DAG-consistent distributed shared-memory algorithms. In *Proc. of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 297–308, 1996.

[5] R.A. Chowdhury. Experimental Evaluation of a Cache-Oblivious LCS Algorithm. Tech. Rep. TR-05-43, Dept. of Comp. Sci., University of Texas at Austin, Oct. 2005.

[6] C. Cherng and R.E. Ladner. Cache efficient simple dynamic programming. In *Proc. of the International Conf. on the Analysis of Algorithms*, pp. 49–58, 2005.

[7] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd ed., 2001.

[8] R.W. Floyd. Algorithm 97 (SHORTEST PATH). *Communications of the ACM*, 5(6):345, 1962.

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

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

[11] Z. Galil, and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64:107–118, 1989.

[12] Z. Galil, and K. Park. Parallel algorithms for dynamic programming recurrences with more than $\mathcal{O}(1)$ dependency. *Journal of Parallel and Distributed Computing*, vol. 21, pp. 213–222, 1994.

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

[14] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *JACM*, 40(4):664-675, 1977.

[15] D.S. Hirschberg, and L.L. Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, vol. 16(4), pp. 628–638, 1987.

[16] J.-W. Hong and H.T. Kung. I/O complexity: the red-blue pebble game. In *Proc. of the 13th Annual ACM Symp. on Theory of Computing*, pp. 326–333, 1981.

[17] E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms*. W. H. Freeman & Co., NY, USA, 1998.

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

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

[20] 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.

[21] M. Sniedovich. *Dynamic Programming*. The Marcel Dekker, Inc., New York, NY, USA, 1992.

[22] 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.

[23] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, vol. 18(4), pp. 1065–1081, 1997.

[24] L.G. Valiant. General context-free recognition in less than cubic time. *Journal of Compute and System Sciences*, vol. 10, pp. 308–315, 1975.

[25] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

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

[27] D. Womble, D. Greenberg, S. Wheat, and R. Riesen. Beyond core: making parallel computer I/O practical. In *Proc. of the DAGS/PC Symposium*, pp. 56–63, 1993.