Cache-oblivious Dynamic Programming for Bioinformatics

Rezaul Alam Chowdhury, Hai-Son Le, and Vijaya Ramachandran

Abstract— **We present efficient cache-oblivious algorithms for some well-studied string problems in bioinformatics including *the longest common subsequence, global pairwise sequence alignment* and *3-way sequence alignment (or median)*, both with affine gap costs, and *RNA secondary structure prediction with simple pseudoknots*.**

For each of these problems we present cache-oblivious algorithms that match the best-known time complexity, match or improve the best-known space complexity, and improve significantly over the cache-efficiency of earlier algorithms.

We present experimental results which show that our cache-oblivious algorithms run faster than software and implementations based on previous best algorithms for these problems.

Index Terms— **sequence alignment, median, RNA secondary structure prediction, dynamic programming, cache-efficient, cache-oblivious.**

I. Introduction

ALGORITHMS for sequence alignment and for RNA secondary structure prediction are some of the most widely studied and widely-used methods in bioinformatics. Many of these are dynamic programming algorithms that run in polynomial time under the traditional *von Neumann Model* of computation which assumes a single layer of memory with uniform access cost, and many have been further improved in their space usage, mainly using a technique due to Hirschberg [23]. Modern computers, however, differ significantly from the original von Neumann architecture. Unlike von Neumann machines, memory on these machines is typically organized in a hierarchy with registers in the lowest level followed by several levels of caches (L1, L2 and possibly L3), RAM, and disk. The access time and size of each level increases with its depth, and data is transferred in blocks between adjacent levels. When executed on such a typical modern computer algorithms designed for the traditional model often cause the processor to stall while waiting for data to be transferred from slower levels of memory. The situation is the worst for large datasets that involve block transfers to and from the disk. Therefore, in order to perform well on these machines new algorithms must be designed that reduce the number of block transfers between different levels of the memory hierarchy.

Cache-efficiency and cache-oblivious algorithms. The two-level I/O model [1] is a simple abstraction of the memory hierarchy that consists of a cache of size M, and an arbitrarily large main memory partitioned into blocks of size B. An algorithm is said to have caused a cache-miss if it references a block that does not reside in the cache and must be fetched from the main memory. The *cache complexity* (or *I/O complexity*) of an algorithm is measured in terms of the number of cache-misses it incurs and thus the number of block transfers or I/O operations it causes. Algorithms designed for this model often crucially depend on the knowledge of M and B, and thus do not adapt well when these parameters change.

The ideal-cache model [18] is an extension of the two-level I/O model with an additional requirement that algorithms must remain oblivious of cache parameters, i.e., cache-oblivious. The model assumes an optimal offline cache replacement policy, which can be approximated to within a constant factor by standard cache replacement methods such as LRU and FIFO. A well-designed cache-oblivious algorithm is flexible and portable, and simultaneously adapts to all levels of a multi-level memory hierarchy.

A. Our Results

In this paper we present an efficient cache-oblivious framework that solves a general class of recurrence relations in 2- and 3-dimensions that are amenable to solution by dynamic programs with ‘local dependencies’. In a dynamic program with local dependencies the value of each cell in the DP table depends only on values in adjacent cells. In principle our framework can be generalized to any number of dimensions, although we study explicitly only the 2- and 3-dimensional cases. We describe our methodology using the simple and well-known *longest common subsequence* (LCS) problem. We generalize this framework to develop cache-oblivious algorithms for several well-known string problems in bioinformatics, and show that our algorithms are both theoretically and experimentally more efficient than previous algorithms for these problems. Our results for the string problems we consider are the following (recall that B is the block transfer size, and M is the size of the cache):

- *Global pairwise alignment* with affine gap costs: On a pair of sequences of length n each our cache-oblivious algorithm runs in $\mathcal{O}\left(n^2\right)$ time, uses $\mathcal{O}\left(n\right)$ space and incurs $\mathcal{O}\left(\frac{n^2}{BM}\right)$ cache-misses.
- *Median* (i.e., optimal alignment of three sequences) with affine gap costs: Our cache-oblivious algorithm runs in $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(n^2\right)$ space, and incurs only $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache-misses on three sequences of length n each.
- *RNA secondary structure prediction with simple pseudoknots*: On an RNA sequence of length n, our cache-oblivious algorithm runs in $\mathcal{O}\left(n^4\right)$ time, uses $\mathcal{O}\left(n^2\right)$ space and incurs $\mathcal{O}\left(\frac{n^4}{B\sqrt{M}}\right)$ cache-misses.

Our cache-oblivious algorithms improve on the space usage of traditional dynamic programs for each of the problems we study, and match the space usage of the Hirschberg’s space-reduced version [23] of these traditional DPs. However, our space reduction is obtained through a divide-and-conquer strategy that

This work was supported in part by NSF Grant CCF-0514876 and NSF CISE Research Infrastructure Grant EIA-0303609.

Preliminary versions of Sections II-A and II-C were presented at SODA’06.

Rezaul Alam Chowdhury and Vijaya Ramachandran are with Univ. of Texas at Austin, Email: `{shaikat,vlr}@cs.utexas.edu`.

Hai-Son Le is with Google Inc., Email: `haisonle@google.com`.

is quite different from the method used in [23]. Hirschberg's approach, too, decomposes the problem into subproblems, but uses a method that involves the application of traditional iterative DP in both forward and backward directions. The application of iterative DP results in inefficient cache usage. Moreover, the use of both forward and backward DP, and particularly the need to combine the results obtained from them, sometimes complicates the implementation of Hirschberg's method (e.g., for multiple simultaneous recurrences or recurrences with multiple fields). In contrast, our algorithm always applies DP in one direction and is arguably simpler to implement. A method for applying Hirschberg's space-reduction using forward-only DP is given in [17], but it involves repeated linear scans and thus is not cache-efficient.

In our experimental study of the first two problems we compare our implementations to publicly available software written by others, and for the last one our comparison is to our implementation of the best previous algorithm (due to Akutsu [3]). In general our cache-oblivious algorithms outperformed the other algorithms for all three problems.

In related work, in [13] we present parallel cache-oblivious implementations of these algorithms for distributed and shared caches, and for *multicores*. Additionally, we present cache-oblivious sequential and parallel algorithms for solving several dynamic programming recurrences with 'non-local dependencies'[1] including those for pairwise sequence alignment with general gap costs, and the basic RNA secondary structure prediction (without pseudoknots) problem in [10], [8], [13].

B. Organization of the Paper

In Section II we describe our cache-oblivious framework for solving dynamic programming problems with local dependencies: in Section II-A we use the simple and well-known 2-dimensional dynamic programming recurrence for finding a longest common subsequence (LCS) to describe our methodology, in Section II-B we formulate the general d-dimensional framework, in Section II-C we establish its I/O lower bound, and in Section II-D we apply this framework to obtain cache-oblivious algorithms for global pairwise sequence alignment, median, and RNA secondary structure prediction with simple pseudoknots. In Section III we present our experimental results on the three problems.

Preliminary versions of the LCS results in Section II-A and the I/O lower bound in Section II-C appeared in a conference [10].

II. Cache-oblivious Dynamic Programs with Local Dependencies

A. The LCS DP

In this section we describe our methodology using the simple and well-known dynamic programming recurrence for the longest common subsequence (LCS) problem.

A sequence $Z = z_1 z_2 \ldots z_k$ is called a *subsequence* of another sequence $X = x_1 x_2 \ldots x_n$ if there exists a strictly increasing function $f : [1, 2, \ldots, k] \rightarrow [1, 2, \ldots, n]$ such that for all $i \in [1, k]$, $z_i = x_{f(i)}$. In the *Longest Common Subsequence* (LCS) problem we are given two input sequences, and we need to find a maximum-length subsequence common to both sequences.

[1]In a dynamic program with non-local dependencies values in some or all cells in the DP table depend on values in non-adjacent cells.

Given two sequences $X = x_1 x_2 \ldots x_n$ and $Y = y_1 y_2 \ldots y_n$ (for simplicity, we assume equal-length sequences here), we define $c[i,j]$ $(0 \le i, j \le n)$ to be the length of an LCS of $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$. Then $c[n,n]$ is the length of an LCS of X and Y, and can be computed using the following recurrence relation (see, e.g., [14]):

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

We can rewrite the recurrence above in the following form:

$$c[i,j] = \begin{cases} h\left(\langle\ i,\ j\ \rangle \right) & \text{if } i = 0 \text{ or } j = 0, \\ f\begin{pmatrix} \langle\ i,\ j\ \rangle,\ \ \langle\ x_i,\ y_j\ \rangle, \\ c[\ i-1 : i,\ j-1 : j\ ] \\ \backslash c[i,j] \end{pmatrix} & \text{otherwise.} \end{cases} \quad \text{(II.2)}$$

where $h(\cdot)$ is an initialization function that always returns 0, and $f(\cdot, \cdot, \cdot)$ is the function that computes the value of each cell based on the values in adjacent cells as follows.

$$f\begin{pmatrix} \langle\ i,\ j\ \rangle,\ \ \langle\ x_i,\ y_j\ \rangle, \\ c[\ i-1 : i,\ j-1 : j\ ] \setminus c[i,j] \end{pmatrix} = \begin{cases} c[i-1, j-1] + 1 & \text{if } x_i = y_j, \\ \max \begin{Bmatrix} c[i, j-1], \\ c[i-1, j] \end{Bmatrix} & \text{otherwise.} \end{cases}$$

Function f uses exactly one cell from its third argument to compute the final value of $c[i,j]$, and we call that specific cell the *parent cell* of $c[i,j]$. The *traceback path* from any cell $c[i,j]$ is the path following the chain of parent cells through c that ends at some $c[i', j']$ with $i' = 0 \vee j' = 0$. An LCS can be extracted from the traceback path starting at $c[n,n]$.

All computations above are performed in the domain of non-negative integers (i.e., the set $\mathbb{N}$ of natural numbers).

Recurrence II.2 gives the general form of a DP with local dependencies in 2 dimensions. It can be evaluated iteratively in $\mathcal{O}\left(n^2\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(n^2/B\right)$ cache-misses. It has been shown in [2], [24], [32] that the LCS problem cannot be solved in $o\left(n^2\right)$ 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 $\mathcal{O}\left(\frac{n^2}{\log n}\right)$ time and space [33], though this appears to be impractical to implement. Faster algorithms exist for different special cases of the problem [6]. If the alphabet size is unrestricted, this problem can be solved in $\mathcal{O}\left(\frac{hn^2}{\log n}\right)$ time and space [15], where h is the entropy of the sequences.

In most applications, however, the quadratic space required by an LCS algorithm is a more constraining factor than its quadratic running time [22]. Fortunately, there are linear space implementations [23], [29], [5] of the LCS recurrence, but the cache complexity remains $\Omega\left(\frac{n^2}{B}\right)$ and the running time roughly doubles. Hirschberg's space-reduction technique [23] for the LCS recurrence has become the most widely used method for reducing the space complexity of similar DP-based algorithms in computational biology [34], [36], [42], [26]. However, if a traceback path is not required it is easy to reduce space requirement of the iterative algorithm to $\mathcal{O}(n)$ even without using Hirschberg's technique (see, e.g., [14]), and its cache-complexity can be improved to $\mathcal{O}\left(\frac{n^2}{BM}\right)$ using the cache-oblivious stencil-computation technique [19].

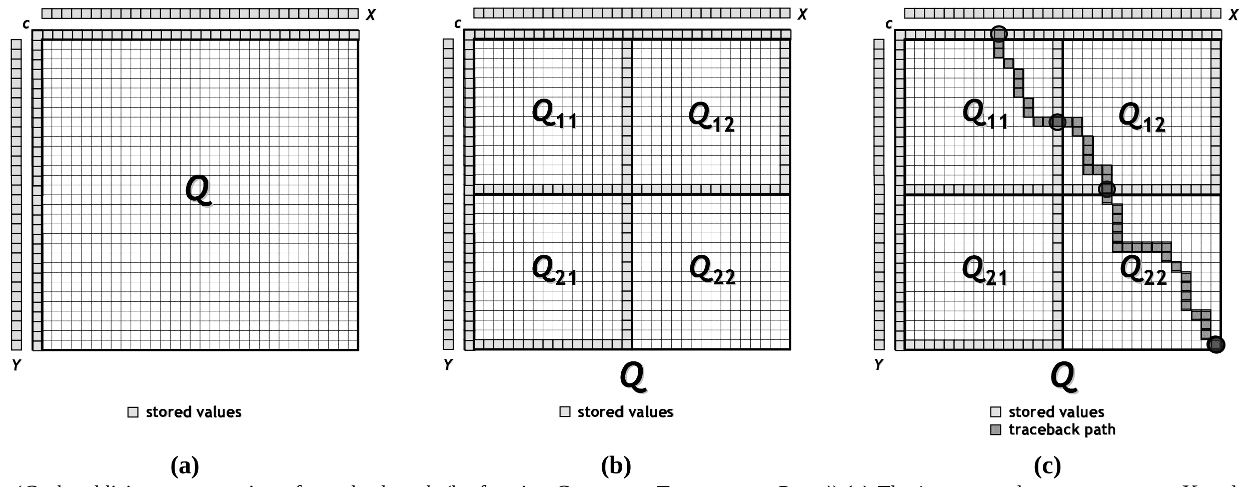

Fig. 1. (*Cache-oblivious computation of traceback path (by function* COMPUTE-TRACEBACK-PATH*)*) **(a)** The inputs are the two sequences X and Y, and the entries on the left and the top boundaries of matrix $Q \equiv c[\ 1 : n,\ 1 : n\ ]$. **(b)** Forward Pass: The output boundaries of three of the four quadrants of Q are computed recursively (by calling COMPUTE-BOUNDARY) in the following order: Q_{11}, Q_{12} and Q_{21}. This order ensures that the input boundaries of each quadrant are already available at the time it is processed by COMPUTE-BOUNDARY. **(c)** Backward Pass: The fragments of the traceback path are extracted from the quadrants by calling COMPUTE-TRACEBACK-PATH on them recursively in the opposite order of the forward pass.

In the rest of this section we present a cache-oblivious algorithm for solving recurrence II.2 along with a traceback path in $\mathcal{O}\left(n^2\right)$ time, $\mathcal{O}\left(n\right)$ space and $\mathcal{O}\left(\frac{n^2}{BM}\right)$ cache misses. It improves over the previous best cache-miss bound by at least a factor of M, and reduces space requirement by a factor of n when compared with the traditional iterative solution.

Cache-oblivious Algorithm for Solving Recurrence II.2. Our algorithm COMPUTE-TRACEBACK-PATH works by decomposing the given matrix $c[\ 1 : n,\ 1 : n\ ]$ into smaller submatrices, and then recursively extracting the fragments of the traceback path from them. For any such submatrix one can recursively compute the entries on its output boundary (i.e., on its right and bottom boundaries) provided the entries on its input boundary (i.e., entries immediately outside of its left and top boundaries) are already known. Since the submatrices share boundaries, when the output boundaries of all submatrices are computed the problem of finding the traceback path through the entire matrix is reduced to the problem of recursively finding the fragments of the path through the submatrices. Though we compute all n^2 entries of c, at any stage of recursion we only need to save the entries on the boundaries of the submatrices and thus we use only $\mathcal{O}\left(n\right)$ space. The divide and conquer strategy also improves locality of computation and consequently leads to an efficient cache-oblivious algorithm.

We describe below the two parts of our algorithm. The pseudocode for both parts are given in Figure 2. The initial call is to COMPUTE-TRACEBACK-PATH on the two input sequences, which computes the traceback path through the input matrix starting at its bottom-right corner. This function uses the COMPUTE-BOUNDARY routine for decomposing the input matrix into submatrices each representing a smaller instance of the original problem.

COMPUTE-BOUNDARY. Given the input boundary of $c[\ i_1 : i_2,\ j_1 : j_2\ ]$ this function (Function 2.2) recursively computes its output boundary. For simplicity of exposition we assume that $i_2 - i_1 = j_2 - j_1 = 2^q - 1$ for some integer $q \geq 0$.

If $q = 0$, the function can compute the output boundary directly using recurrence II.2, otherwise it decomposes its quadratic computation space Q (initially $Q \equiv c[\ 1 : n,\ 1 : n\ ]$) into 4 quadrants $Q_{i,j}$, $1 \leq i, j \leq 2$, where $Q_{i,j}$ denotes the quadrant that is i-th from the top and j-th from the left. It then computes the output boundary of each quadrant recursively as the input boundary of the quadrant becomes available during the process of computation. After all recursive calls terminate, the output boundary of Q is composed from the output boundaries of the quadrants.

Analysis. Let $I_1(n)$ be the cache-complexity of COMPUTE-BOUNDARY on input sequences of length n each. If the sequences are small enough so that the entire input of size $\mathcal{O}\left(n\right)$ completely fits into the cache, then the only cache-misses incurred by the function will be $\mathcal{O}\left(1 + \frac{n}{B}\right)$ in order to read the input into the cache initially, and write the output to the main memory at the end. In this case, the intermediate recursive function calls will incur no additional cache-misses since the entire input is already in the cache. However, if the the input is too large to fit into the cache, the total number of cache-misses incurred by the function is the sum of the cache-misses incurred by the recursive function calls, and $\mathcal{O}\left(1 + \frac{n}{B}\right)$ additional misses incurred while saving/restoring intermediate outputs between recursive function calls. Thus we have

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

where α is a suitable constant such that computation involving two input sequences of length αM each can be performed completely inside the cache. Solving the recurrence we obtain $I_1(n) = \mathcal{O}\left(1 + \frac{n}{B} + \frac{n^2}{BM}\right)$ for all n. It is straight-forward to show that the algorithm runs in $\mathcal{O}\left(n^2\right)$ time and uses $\mathcal{O}\left(n\right)$ space, and the cache complexity reduces to $\mathcal{O}\left(\frac{n^2}{BM}\right)$ when the input is too large for the cache (i.e., $n = \Omega\left(M\right)$). In contrast, though the standard iterative dynamic programming approach for computing the output boundary has the same time and space complexities (see, e.g., [14] for a standard technique that allows the DP to be implemented in $\mathcal{O}\left(n\right)$ space), it incurs a factor of M more cache-misses.

COMPUTE-TRACEBACK-PATH. This is the main algorithm,

```
FUNCTION 2.1: COMPUTE-TRACEBACK-PATH( $X,\ Y,\ L,\ T,\ P$ )
Input. Here $r = |X| = |Y| = 2^t$ for some integer $t \in [1,p]$, and $Q[\ 0:r,\ 0:r\ ] \equiv c[\ u-1:u+r-1,\ v-1:v+r-1\ ]$, $X = x_u x_{u+1} \ldots x_{u+r-1}$
and $Y = y_v y_{v+1} \ldots y_{v+r-1}$ for some $u$ and $v$ $(1 \le u, v \le n-r+1)$. The left and top boundaries of $Q[\ 1:r,\ 1:r\ ]$ are in $L$ $(\equiv Q[\ 0,\ 0:r\ ])$
and $T$ $(\equiv Q[\ 0:r,\ 0\ ])$, respectively. Current traceback path is given in $P$.
Output. Returns the updated traceback path.
  1.  if $P \cap Q = \emptyset$ return $P$
  2.  if $r = 1$ then update $P$ using recurrence II.2
  3.  else     { For $i, j \in [1,2]$, the left, right, top and bottom boundaries of quadrant $Q_{ij}$ are denoted by $L_{ij}$, $R_{ij}$, $T_{ij}$
               and $D_{ij}$, respectively. $X_1$ and $X_2$ denote the 1st and the 2nd half of $X$, respectively (similarly for $Y$).}
  4.    Extract $L_{1,j}$ from $L$, and $T_{i,1}$ from $T$, where $i, j \in [1,2]$      { $L_{2,j} \equiv R_{1,j}$ and $T_{i,2} \equiv D_{i,1}$ for $i, j \in [1,2]$ }
  5.    $quadrant[\ 1:4\ ] \leftarrow \langle\ \langle\ 1,1\ \rangle,\ \langle\ 1,2\ \rangle,\ \langle\ 2,1\ \rangle,\ \langle\ 2,2\ \rangle\ \rangle$
        Forward Pass ( Compute Boundaries ):
  6.      for $l \leftarrow 1$ to $3$ do
  7.        $\langle\ i,\ j\ \rangle \leftarrow quadrant[\ l\ ]$, $\langle\ R_{ij},\ D_{ij}\ \rangle \leftarrow$ COMPUTE-BOUNDARY( $X_i,\ Y_j,\ L'_{ij},\ T'_{ij}$ )
               { $L'_{ij}$ is the same as $L_{ij}$ except that it contains one additional cell at the top.
                 Similarly, $T'_{ij}$ contains one more cell to its left than $T_{ij}$. }
        Backward Pass ( Compute Traceback Path ):
  8.      for $l \leftarrow 4$ downto $1$ do
  9.        $\langle\ i,\ j\ \rangle \leftarrow quadrant[\ l\ ]$, $P \leftarrow$ COMPUTE-TRACEBACK-PATH( $X_i,\ Y_j,\ L'_{ij},\ T'_{ij},\ P$ )
 10.  return $P$
COMPUTE-TRACEBACK-PATH ENDS
```

```
FUNCTION 2.2: COMPUTE-BOUNDARY( $X,\ Y,\ L,\ T$ )
Input. Same as the input description of COMPUTE-TRACEBACK-PATH (Function 2.1).
Output. Returns an ordered tuple $\langle R, D \rangle$, where $R$ $(\equiv Q[\ r,\ 1:r\ ])$ and $D$ $(\equiv Q[\ 1:r,\ r\ ])$ are the right and bottom boundaries of $Q[\ 1:r,\ 1:r\ ]$,
respectively.
  1.  if $r = 1$ then $R = D \leftarrow f_2\ (\ \langle\ u,\ v\ \rangle,\ \langle\ X,\ Y\ \rangle,\ L \cup T\ )$
  2.  else
  3.    Extract $L_{1,j}$ from $L$, and $T_{i,1}$ from $T$, respectively, where $i, j \in [1,2]$
  4.    $quadrant[\ 1:4\ ] \leftarrow \langle\ \langle\ 1,1\ \rangle,\ \langle\ 1,2\ \rangle,\ \langle\ 2,1\ \rangle,\ \langle\ 2,2\ \rangle\ \rangle$
  5.    for $l \leftarrow 1$ to $4$ do
  6.      $\langle\ i,\ j\ \rangle \leftarrow quadrant[\ l\ ]$, $\langle\ R_{ij},\ D_{ij}\ \rangle \leftarrow$ COMPUTE-BOUNDARY( $X_i,\ Y_j,\ L'_{ij},\ T'_{ij}$ )
  7.    Compose $R$ from $R_{2,j}$, and $D$ from $D_{i,2}$, respectively, where $i, j \in [1,2]$
  8.  return $\langle\ R,\ D\ \rangle$
COMPUTE-BOUNDARY ENDS
```

Fig. 2. Cache-oblivious algorithm for evaluating recurrence II.2 along with the traceback path. For convenience of exposition we assume that we only need to compute $c[\ 1:n,\ 1:n\ ]$ where $n = 2^q$ for some nonnegative integer q. The initial call to COMPUTE-TRACEBACK-PATH is made with $X = x_1 x_2 \ldots x_n$, $Y = y_1 y_2 \ldots y_n$, $L \equiv c[\ 0,\ 0:n\ ]$, $T \equiv c[\ 0:n,\ 0\ ]$ and $P = \langle (n,n) \rangle$.

which recursively calls both itself and COMPUTE-BOUNDARY. Given the input boundary of $c[\ i_1 : i_2,\ j_1 : j_2\ ]$ and the entry point of the traceback path on the output boundary, this function (Function 2.1) recursively computes the entire path. Recall that a traceback runs backwards, that is, it enters the cube through a point on the output boundary and exits through the input boundary.

If $q = 0$, the traceback path can be updated directly using recurrence II.2, otherwise the function performs two passes: forward and backward. In the forward pass it computes the output boundaries of all quadrants except $Q_{2,2}$ as in COMPUTE-BOUNDARY. After this pass the algorithm knows the input boundaries of all four quadrants, and the problem reduces to recursively extracting the fragments of the traceback path from each quadrant and combining them. In the backward pass the algorithm starts at $Q_{2,2}$ and updates the traceback path by calling itself recursively on the quadrants in the reverse order of the forward pass. This backward order of the recursive calls is essential since in order to find the traceback path through a quadrant the algorithm requires an entry point on its output boundary through which the path enters the quadrant and initially this point is known for only one quadrant. The quadrants are processed in the backward order because it ensures that the exit point of the traceback path from one quadrant can be used as the entry point of the path to the next quadrant in the sequence.

Analysis. Let $I_2(n)$ be the cache-complexity of COMPUTE-TRACEBACK-PATH on input sequences of length n each. We observe that though the algorithm calls itself recursively 4 times in the backward pass, at most 3 of those recursive calls will actually be executed and the rest will terminate at line 1 of the algorithm (see Figure 2)) since the traceback path cannot intersect more than 3 quadrants. Then using arguments similar to those used in determining $I_1(n)$, we have,

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

where γ is a suitable constant such that the computation involving sequences of length γM each can be performed completely inside the cache. Solving the recurrence we obtain $I_2(n) =$

$\mathcal{O}\left(1 + \frac{n}{B} + \frac{n^2}{BM}\right)$ for all n. The algorithm runs in $\mathcal{O}\left(n^2\right)$ time and uses $\mathcal{O}(n)$ space, and $I_2(n)$ reduces to $\mathcal{O}\left(\frac{n^2}{BM}\right)$ provided the inputs are too large to fit into the cache. When compared with the cache-complexity of any existing algorithm for finding the traceback path our algorithm improves it by at least a factor of M, and improves the space complexity by a factor of n when compared against the standard dynamic programming solution.

Our algorithm can be easily extended to handle lengths that are not powers of 2 within the same performance bounds. Thus we have the following theorem.

Theorem 2.1: Given two sequences X and Y of length n each, recurrence II.2 can be solved and a traceback path can be computed cache-obliviously in $\mathcal{O}\left(n^2\right)$ time, $\mathcal{O}(n)$ space and $\mathcal{O}\left(1 + \frac{n}{B} + \frac{n^2}{BM}\right)$ cache misses.

If the sequences are long enough $\left(\text{i.e., } n = \Omega(M)\right)$, the cache complexity of the algorithm reduces to $\mathcal{O}\left(\frac{n^2}{BM}\right)$.

B. A General Framework for DPs with Local Dependencies

Suppose we are given the following.

- $d \geq 2$ sequences $S_i = s_{i,1}s_{i,2}\ldots s_{i,n}$, $1 \leq i \leq d$, of length n each, with symbols chosen from an arbitrary finite alphabet Σ. We define the following (to be used later).
  - Given integers $i_j \in [0, n]$, $j \in [1, d]$, we denote by $\mathbf{i}$ the sequence of d integers $i_1, i_2, \ldots, i_d$; and by $\langle\, \mathbf{i} \,\rangle$ we denote the d-dimensional vector $\langle\, i_1, i_2, \ldots, i_d \,\rangle$.
  - By $\langle\, \mathbf{S_i} \,\rangle$ we denote the d-dimensional vector $\langle\, s_{1,i_1},\ s_{2,i_2}, \ldots,\ s_{d,i_d} \,\rangle$ containing the i_j-th symbol of S_j in j-th position, where each $i_j \in [1, n]$.
- An arbitrary set $\mathcal{U}$.
- An initialization function $h(\cdot)$ that accepts a vector $\langle\, \mathbf{i} \,\rangle$ as input and outputs an element from $\mathcal{U}$.
- A function $f(\cdot,\cdot,\cdot)$ that accepts vectors $\langle\, \mathbf{i} \,\rangle$ and $\langle\, \mathbf{S_i} \,\rangle$, and an ordered set of $2^d - 1$ elements from Σ, and returns an element of $\mathcal{U}$.

Now suppose $c[\ 0 : n,\ 0 : n,\ \ldots,\ 0 : n\ ]$ is a d-dimensional matrix that can store elements from the given set $\mathcal{U}$, and we want to compute the entries of c using the following dynamic programming recurrence.

$$c[\,\mathbf{i}\,] = \begin{cases} h\left(\,\langle\, \mathbf{i} \,\rangle\,\right) & \text{if } \exists\ i_j = 0, \\ f\left(\langle\, \mathbf{i} \,\rangle,\ \langle\, \mathbf{S_i} \,\rangle,\ c\begin{bmatrix} i_1 - 1 : i_1, \\ i_2 - 1 : i_2, \\ \ldots, \\ i_d - 1 : i_d \end{bmatrix} \setminus c[\,\mathbf{i}\,]\right) & \text{otherwise.} \end{cases} \tag{II.3}$$

Function f can be arbitrary except that it is allowed to use exactly one cell from its third argument to compute the final value of $c[i_1, i_2, \ldots, i_d]$ (though it can consider all cells), which we call the parent cell of $c[i_1, i_2, \ldots, i_d]$. We also assume that f does not access any memory locations in addition to those passed to it as inputs except possibly some constant size local variables.

Typically, two types of outputs are expected when evaluating this recurrence: (i) the value of $c[n, n, \ldots, n]$, and (ii) the traceback path starting from $c[n, n, \ldots, n]$. As in the case of LCS recurrence, the traceback path from any cell $c[i_1, i_2, \ldots, i_d]$ is the path following the chain of parent cells through c that ends at some $c[i'_1, i'_2, \ldots, i'_d]$ with $\exists\ i'_j = 0$.

Each cell of c can have multiple fields and in that case f must compute a value for each field, though as before, it is allowed to use exactly one field from its third argument to compute the final value of any field in $c[i_1, i_2, \ldots, i_d]$. The definition of traceback path extends naturally to this case, i.e., when the cells have multiple fields.

Recurrence II.2 in Section II-A gives the 2 dimensional version of recurrence II.3 and the h and f functions for the LCS recurrence are listed below that recurrence.

Recurrence II.3 can be solved in $\mathcal{O}\left(n^d\right)$ time, $\mathcal{O}\left(n^{d-1}\right)$ space and $\mathcal{O}\left(n^d/B\right)$ cache-misses using Hirschberg's technique [23].

A straight-forward extension of the cache-oblivious algorithm given in Section II-A and Figure 2 for solving the 2D recurrence II.2 solves the general recurrence II.3 along with a traceback path for any arbitrary dimension $d \geq 2$. The algorithm is similar to the 2D algorithm, but the computation space is a d-dimensional hypercube, and the input and output boundaries are of dimension $d-1$. The algorithm works by decomposing the hypercube into 2^d sub-hypercubes, and computing the output boundaries of the sub-hypercubes recursively in a sequence so that the output boundaries of a sub-hypercube are computed only after its input boundaries become available (possibly as outputs of recursive calls earlier in the sequence). After the output boundaries of all sub-hypercubes are computed, we can find the traceback path through the entire hypercube by recursively extracting the fragments of the path through the sub-hypercubes and stitching them together. Thus we have the following theorem.

Theorem 2.2: Given $d \geq 2$ sequences S_i, $1 \leq i \leq d$, of length n each, with symbols chosen from an arbitrary finite alphabet, recurrence II.3 can be solved and a traceback path can be computed cache-obliviously in $\mathcal{O}\left(n^d\right)$ time, $\mathcal{O}\left(n^{d-1}\right)$ space and $\mathcal{O}\left(\frac{n^d}{BM^{\frac{1}{d-1}}}\right)$ cache misses provided $n = \Omega\left(M^{\frac{1}{d-1}}\right)$ and $M = \Omega\left(B^{d-1}\right)$.

Details of the cache-oblvious algorithm for $d = 3$ as well as its pseudocode can be found in [9] and in the PhD thesis of the first author [8].

C. I/O Lower Bound

The following theorem establishes that our cache-oblivious algorithm for solving recurrence II.3 is cache-optimal:

Theorem 2.3: For any $d \geq 2$, any algorithm that implements the computation defined by recurrence II.3, must perform $\Omega\left(\frac{n^d}{BM^{\frac{1}{d-1}}}\right)$ block transfers.

We obtain the lower bound in Theorem 2.3 using the I/O lower bound proved by Hong & Kung [25] for executing the DAG obtained by taking the product of d directed line graphs. Let $L_1 = (V, E)$ be a directed line graph, where $V = \{\ 1, 2, \ldots, n\ \}$ and $E = \{\ (i, i+1) \mid i \in [1, n-1]\ \}$. Nodes in L_1 represent operations, and edges represent data-flow. The node with no incoming edges (i.e., node 1) is the unique *input* and the node with no outgoing edges (i.e., node n) is the unique *output*. For $d \geq 2$, L_d is obtained by taking the product of d such L_1's. Figure 3(b) shows L_2. Corollary 7.1 in [25] gives the following lower bound on the number of I/O operations Q required to execute L_d.

Corollary 7.1 in [25]. *For the product L_d with $d \geq 2$, $Q = \Omega\left(\frac{n^d}{M^{\frac{1}{d-1}}}\right)$.*

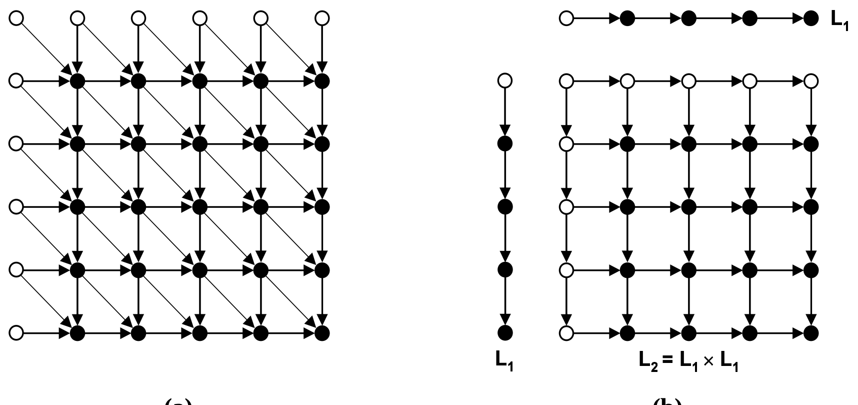

Fig. 3. (*I/O lower bound for DP implementing recurrence II.3*) **(a)** Computational DAG G_2 implementing recurrence II.3 for $d = 2$. The nodes colored *white* represent input nodes. **(b)** Product graph L_2 of two line graphs (L_1), which is a subDAG of DAG G_2 shown in Figure 3(a). Hence, I/O lower bound for executing L_2 also holds for G_2.

The corollary above assumes that data is transferred to and from the cache in blocks of size 1. For block size B, $Q = \Omega\left(\frac{n^d}{BM^{\frac{1}{d-1}}}\right)$.

Now consider the computation DAG G_d given by recurrence II.3 for dimension d. Figure 3(a) shows this DAG for $d = 2$. It is easy to see that L_d is, in fact, a subDAG of G_d, and hence I/O lower bound for executing L_d also holds for G_d. Therefore, Theorem 2.3 follows from the corollary above under the assumption that data is transferred in blocks of size B.

D. Applications of the Cache-oblivious Framework

In this section we apply the cache-oblivious framework described in Section II-B to obtain cache-oblivious algorithms for pairwise sequence alignment, median of three sequences, and RNA secondary structure prediction with simple pseudoknots.

1) **PAIRWISE GLOBAL SEQUENCE ALIGNMENT WITH AFFINE GAP PENALTY***:* Sequence alignment plays a central role in biological sequence comparison, and can reveal important relationships among organisms. Given two strings $X = x_1x_2 \ldots x_m$ and $Y = y_1y_2 \ldots y_n$ over a finite alphabet Σ, an *alignment* of X and Y is a matching M of sets $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ such that if $(i, j), (i', j') \in M$ and $i < i'$ hold then $j < j'$ must also hold [26]. The i-th letter of X or Y is said to be in a *gap* if it does not appear in any pair in M. Given a *gap penalty* g and a mismatch cost $s(a, b)$ for each pair $a, b \in \Sigma$, the *basic (global) pairwise sequence alignment problem* asks for a matching M_{opt} for which $(m+n-|M_{opt}|)\times g+\sum_{(a,b)\in M_{opt}} s(a, b)$ is minimized [26].

For simplicity of exposition we will assume $m = n$ for the rest of this section.

The formulation of the basic sequence alignment problem favors a large number of small gaps while real biological processes favor the opposite. The alignment can be made more realistic by using an *affine gap penalty* [20], [4] which has two parameters: a *gap introduction cost* g_i and a *gap extension cost* g_e. A run of k gaps incurs a total cost of $g_i + g_e \times k$.

In [20] Gotoh presented an $\mathcal{O}\left(n^2\right)$ time and $\mathcal{O}\left(n^2\right)$ space DP algorithm for solving the global pairwise alignment problem with affine gap costs. The algorithm incurs $\mathcal{O}\left(\frac{n^2}{B}\right)$ cache misses. The space complexity of the algorithm can be reduced to $\mathcal{O}\left(n\right)$ using Hirschberg's space-reduction technique [34] or the diagonal checkpointing technique described in [21]. Gotoh's algorithm solves the following DP recurrences.

$$D(i,j) = \begin{cases} G(0,j) + g_e & \text{if } i = 0 \wedge j > 0 \\ \min\begin{Bmatrix} D(i-1,j), \\ G(i-1,j)+g_i \end{Bmatrix} + g_e & \text{if } i > 0 \wedge j > 0. \end{cases} \tag{II.4}$$

$$I(i,j) = \begin{cases} G(i,0) + g_e & \text{if } i > 0 \wedge j = 0 \\ \min\begin{Bmatrix} I(i,j-1), \\ G(i,j-1)+g_i \end{Bmatrix} + g_e & \text{if } i > 0 \wedge j > 0. \end{cases} \tag{II.5}$$

$$G(i,j) = \begin{cases} 0 & \text{if } i = 0 \wedge j = 0 \\ g_i + g_e \times j & \text{if } i = 0 \wedge j > 0 \\ g_i + g_e \times i & \text{if } i > 0 \wedge j = 0 \\ \min\begin{Bmatrix} D(i,j),\ I(i,j), \\ G(i-1,j-1) \\ +s(x_i,y_j) \end{Bmatrix} & \text{if } i > 0 \wedge j > 0. \end{cases} \tag{II.6}$$

The optimal alignment cost is $\min\{G(n,n), D(n,n), I(n,n)\}$ and an optimal alignment can be traced back from the smallest of $G(n,n)$, $D(n,n)$ and $I(n,n)$.

Cache-oblivious Implementation. Recurrences II.4 - II.6 can be viewed as evaluating a single matrix $c[\ 0 : n,\ 0 : n\ ]$ with three fields: D, I and G. These recurrences can be treated as a single recurrence matching the general recurrence II.3 for $d = 2$ by defining functions h and f to output triplets with their 1st, 2nd and 3rd entries containing values for the D, I and G fields of c, respectively. For example, $f\big(\ \langle\ i,\ j\ \rangle,\ \langle\ x_i,\ y_j\ \rangle,\ c[\ i-1 : i,\ j-1 : j\ ] \setminus c[i,j]\big)$ returns a triplet $\langle\ v_D,\ v_I,\ v_G\ \rangle$, where,

$$\begin{aligned} v_D &= \min\{\ c[\ i-1,\ j\ ].D,\ c[\ i-1,\ j\ ].G + g_i\ \} + g_e, \\ v_I &= \min\{\ c[\ i,\ j-1\ ].I,\ c[\ i,\ j-1\ ].G + g_i\ \} + g_e \\ \text{and } v_G &= \min\begin{Bmatrix} c[\ i,\ j\ ].D,\ c[\ i,\ j\ ].I, \\ c[\ i-1,\ j-1\ ].G + s(x_i,y_j) \end{Bmatrix}. \end{aligned}$$

Therefore, function COMPUTE-BOUNDARY in Figure 2 can be used to compute the optimal alignment cost cache-obliviously, and COMPUTE-TRACEBACK-PATH can be used to extract the optimal alignment. Thus the following claim follows from Theorem 2.1 in Section II-A.

Claim 2.1: Optimal global alignment of two sequences of length n each can be performed cache-obliviously using an affine gap cost in $\mathcal{O}\left(n^2\right)$ time, $\mathcal{O}\left(n\right)$ space and $\mathcal{O}\left(\frac{n^2}{BM}\right)$ cache misses.

The cache-complexity of the our cache-oblivious algorithm is a factor of M improvement over previous algorithms [20], [34].

2) **ALIGNMENT OF THREE SEQUENCES (MEDIAN)***:* Given three sequences X, Y and Z, the *median problem* asks for a sequence W such that the sum of the pairwise alignment costs of W with X, Y and Z is minimized. The sequence W is called the *median* of the three given sequences. In this section we will assume affine gap costs for the alignments. The 3-way sequence alignment can be obtained as the pairwise alignment of each of X, Y and Z with the median sequence W. Hence we will focus on finding the median sequence.

In [28] Knudsen presented a dynamic programming algorithm for optimal multiple alignment of any number of sequences related by a tree under affine gap costs. The input sequences are assumed to be at the leaves of the tree, and the optimal alignment cost is the minimum sum of pairwise alignment costs of the sequence pairs at the ends of each edge of the tree over all possible ancestral sequences (i.e., the unknown sequences at the internal nodes of the tree). For N sequences of length n each, the algorithm runs in $\mathcal{O}\left(16.81^N n^N\right)$ time and uses $\mathcal{O}\left(7.442^N n^N\right)$ space. For $N = 3$, Knudsen's algorithm solves the median problem in $\mathcal{O}\left(n^3\right)$ time and space, and incurs $\mathcal{O}\left(\frac{n^3}{B}\right)$ cache-misses. An Ukkonen-based algorithm for the median problem is presented in [38], which performs well especially for sequences whose (3-way) edit distance δ is small. This method performs DP on the edit distance instead of sequence lengths. On average, it requires $\mathcal{O}\left(n + \delta^3\right)$ time and space. A space-reduced version of the algorithm uses $\mathcal{O}\left(n + \delta^2\right)$ space, but runs in $\mathcal{O}\left(n \log \delta + \delta^3\right)$ time on average [38].

Knudsen's algorithm [28] for three sequences (say, $X = x_1 x_2 \ldots x_n$, $Y = y_1 y_2 \ldots y_n$ and $Z = z_1 z_2 \ldots, z_n$) is a dynamic program over a three-dimensional matrix K. These three sequences are assumed to be at the leaves of a star-shaped tree, the root of which corresponds to the ancestor/median sequence W. Each entry $K(i, j, k)$ is composed of several fields, each of which corresponds to an indel configuration that keeps track of ongoing insertions/deletions in the alignment. In a multiple alignment, if we compare the symbols x and w at location l of X and W, respectively, they will be in one of the following three states: (i) x is a residue and w is a gap (i.e., an insertion to X), (ii) x is a gap and w is a residue (i.e., a deletion from X), and (iii) either both of them are residues or both are gaps. Thus if all three input sequences are considered, we will have a total of $3^3 = 27$ such possibilities, each of which is called an indel configuration. However, 4 of those 27 configurations are considered invalid since they lead to contradictory state assignments for the sequences. Hence $K(i, j, k)$ consists of only 23 fields. A residue configuration defines how the next three symbols of the sequences will be matched. Each configuration is a vector $e = (e_1, e_2, e_3, e_4)$, where $e_i \in \{0, 1\}$, $1 \le i \le 4$. The entry e_i, $1 \le i \le 3$, is 1 if the aligned symbol of sequence i is a residue, and 0 otherwise. The last entry e_4 corresponds to the aligned symbol of the median sequence. A residue configuration is valid provided at least one of e_1, e_2 and e_3 is 1, and if more than one of them is 1 then e_4 is also 1 (see [28] for the reasoning behind these conditions). Thus only 10 of the $2^4 = 16$ possible residue configurations are valid. We define $\nu(e, q') = q$ if applying the residue configuration e to the indel configuration q' leads to the indel configuration q. Knudsen's algorithm uses the following recurrence relation which for any indel configuration q computes the field $K(i, j, k)_q$ from all fields $K(i', j', k')_{q'}$ such that for some residue configuration e, $\nu(e, q') = q$, $i' = i - e_1$, $j' = j - e_2$ and $k' = k - e_3$.

$$K(i,j,k)_q = \begin{cases} 0 & \text{if } i = j = k = 0 \land q = q_o \\ \infty & \text{if } i = j = k = 0 \land q \neq q_o \\ \min\limits_{e, q' : q = \nu(e, q')} \left\{ \begin{array}{l} K(i', j', k')_{q'} + G_{e,q'} \\ +M_{(i',j',k') \to (i,j,k)} \end{array} \right\} & \text{otherwise.} \end{cases} \tag{II.7}$$

where q_o is the indel configuration where all symbols match, $M_{(i',j',k') \to (i,j,k)}$ is the matching cost when going from alignment $(x_{i'}, y_{j'}, z_{k'})$ to alignment (x_i, y_j, z_k), and $G_{e,q'}$ is the gap cost of applying e on q'.

The M and G matrices can be pre-computed. Therefore, Knudsen's algorithm runs in $\mathcal{O}\left(n^3\right)$ time and space with $\mathcal{O}\left(\frac{n^3}{B}\right)$ cache-misses.

Cache-oblivious Algorithm. In order to make recurrence II.7 match the general recurrence II.3 for $d = 3$ given in Section II, we shift all symbols of X, Y and Z one position to the right, introduce a dummy symbol in front of each of those three sequences, and obtain the following recurrence by modifying recurrence II.7, where $c[i, j1, k]_q = K(i-1, j-1, k-1)_q$ for $1 \le i, j, k \le n+1$ and any q.

$$c[i,j,k]_q = \begin{cases} \infty & \text{if } i = 0 \ \lor \ j = 0 \ \lor \ k = 0 \\ 0 & \text{if } i = j = k = 1 \ \land \ q = q_o \\ \infty & \text{if } i = j = k = 1 \ \land \ q \neq q_o \\ \min\limits_{e, q' : q = \nu(e, q')} \left\{ \begin{array}{l} c[i', j', k']_{q'} + G_{e,q} \\ +M_{(i',j',k') \to (i,j,k)} \end{array} \right\} & \text{otherwise.} \end{cases}$$

If $i = 0$ or $j = 0$ or $k = 0$ then $c[i, j, k]_q$ can be evaluated using a function $h(\langle\ i,\ j,\ k\ \rangle) = \infty$ as in the general recurrence II.3. Otherwise the value of $c[i, j, k]_q$ depends on the values of i, j, and k, values in some constant size arrays (G and M), and on the cells to its left, back and top. Hence, in this case, $c[i, j, k]_q$ can be evaluated using a function similar to f in recurrence II.3 for $d = 3$. This function is defined simply by the last three rows of the recurrence for $c[i, j, k]_q$. Therefore, the above recurrence matches the 3-dimensional version of the general recurrence II.3, and we claim the following using Theorem 2.2.

Claim 2.2: Optimal alignment of three sequences of length n each can be performed and the median sequence under the optimal alignment can be computed cache-obliviously using an affine gap cost in $\mathcal{O}\left(n^3\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache misses.

3) **RNA SECONDARY STRUCTURE PREDICTION WITH SIMPLE PSEUDOKNOTS***:* A single-stranded RNA can be viewed as a string $X = x_1 x_2 \ldots x_n$ over the alphabet $\{A, U, G, C\}$ of bases. An RNA strand tends to give rise to interesting structures by forming *complementary base pairs* with itself. An *RNA secondary structure* (w/o pseudoknots) is a planar graph with the nesting condition: if $\{x_i, x_j\}$ and $\{x_k, x_l\}$ form base pairs and $i < j$, $k < l$ and $i < k$ hold then either $i < k < l < j$ or $i < j < k < l$ [42], [39], [3]. An *RNA secondary structure with pseudoknots* is a structure where this nesting condition is violated [39], [3].

In [3] Akutsu presented a DP to compute RNA secondary structures with maximum number of base pairs in the presence of *simple pseudoknots* (see [3] for definition) which runs in $\mathcal{O}\left(n^4\right)$ time, $\mathcal{O}\left(n^3\right)$ space and $\mathcal{O}\left(\frac{n^4}{B}\right)$ cache-misses. In this Section we improve its space and cache complexities to $\mathcal{O}\left(n^2\right)$ and $\mathcal{O}\left(\frac{n^4}{B\sqrt{M}}\right)$, respectively, without changing its time complexity.

We list below the DP recurrences used in Akutsu's algorithm [3]. For every pair (i_0, k_0) with $1 \le i_0 \le k_0 - 2 \le n - 2$, recurrences II.8 - II.12 compute the maximum number of base pairs in a pseudoknot with endpoints at the i_0-th and k_0-th residues. The value computed by recurrence II.12, i.e., $S_{pseudo}(i_0, k_0)$, is the desired value. Recurrences II.8 - II.10 consider three locations (i, j, k) $(i_0 - 1 \le i < j \le k \le k_0)$ on the RNA at a time. Recurrences II.8 and II.9 correspond to cases where $\{x_i, x_j\}$ and $\{x_j, x_k\}$ form base pairs, respectively, while recurrence II.10 handles the case where neither $\{x_i, x_j\}$ nor $\{x_j, x_k\}$ forms a base pair. The variable $S_{MAX}(i, j, k)$ in recurrence II.11 contains the maximum score for the triple (i, j, k). In recurrences II.8 and II.9, $v(x_s, y_t) = 1$ if $\{x_s, y_t\}$ form a base pair, otherwise $v(x_s, y_t) = -\infty$. All uninitialized entries are assumed to have value 0.

$$S_L(i,j,k) = \begin{cases} v(x_i, x_j) & \text{if } i_0 \le i < j \ge k, \\ \begin{array}{c} v(a_i, a_j) \\ +S_{MAX}(i-1, j+1, k) \end{array} & \text{if } i_0 \le i < j < k. \end{cases} \tag{II.8}$$

$$S_R(i,j,k) = \begin{cases} v(x_j, x_k) & \text{if } i_0 - 1 = i < j - 1 = k - 2, \\ \begin{array}{c} v(a_j, a_k) \\ +S_{MAX}(i, j+1, k-1) \end{array} & \text{if } i_0 \le i < j < k. \end{cases} \tag{II.9}$$

$$S_M(i,j,k) = \max \left\{ \begin{array}{c} S_L(i-1, j, k), \\ S_M(i-1, j, k), \\ S_{MAX}(i, j+1, k), \\ S_M(i, j, k-1), \\ S_R(i, j, k-1) \end{array} \right\} \text{ if } i_0 \le i < j < k. \tag{II.10}$$

$$S_{MAX}(i,j,k) = \max\{\ S_L(i,j,k),\ S_M(i,j,k),\ S_R(i,j,k)\ \} \tag{II.11}$$

$$S_{pseudo}(i_0, k_0) = \max_{i_0 \le i < j < k \le k_0} \{\ S_{MAX}(i,j,k)\ \} \tag{II.12}$$

After computing all entries of S_{MAX} for a fixed i_0, all $S_{pseudo}(i_0, k_0)$ values for $k_0 \ge i_0 + 2$ can be computed using equation II.12 in $\mathcal{O}\left(n^3\right)$ time and space and $\mathcal{O}\left(\frac{n^3}{B}\right)$ cache-misses. Since there are $n - 2$ possible values for i_0, all $S_{pseudo}(i_0, k_0)$ can be computed in $\mathcal{O}\left(n^4\right)$ time, $\mathcal{O}\left(n^3\right)$ space and $\mathcal{O}\left(\frac{n^4}{B}\right)$ cache-misses.

Finally, the following recurrence computes the optimal score $S(1, n)$ for the entire structure.

$$S(i,j) = \max \left\{ \begin{array}{c} S_{pseudo}(i,j),\ S(i+1, j-1) + v(a_i, a_j), \\ \max_{i<k\le j}\{S(i, k-1), S(k, j)\} \end{array} \right\} \tag{II.13}$$

Iterative evaluation of recurrence II.13 requires $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(n^2\right)$ space, and incurs $\mathcal{O}\left(\frac{n^3}{B}\right)$ cache-misses [3] which is sufficient for our purposes. This recurrence can be evaluated in only $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache-misses without changing the other bounds using the cache-oblivious GEP (Gaussian Elimination Paradigm) framework we presented in [10], [11], [12].

Space Reduction. We now describe our space reduction result. A similar method was suggested in [31]. Observe that evaluating recurrence II.12 requires retaining all $\mathcal{O}\left(n^3\right)$ values computed by recurrence II.11. We avoid using this extra space by computing all required $S_{pseudo}(i_0, k_0)$ values on the fly while evaluating recurrence II.11. We achieve this goal by introducing recurrence II.14, replacing recurrence II.12 with recurrence II.15 for S'_{pseudo}, and using S'_{pseudo} instead of S_{pseudo} for evaluating recurrence II.13. Intuitively, the variable $S_P(i,j,k)$ in recurrence II.14 stores the maximum score among all triples (i, j', k) with $j' \ge j$. All uninitialized entries in recurrences II.14 and II.15 are assumed to have value $-\infty$.

$$S_P(i,j,k) = \begin{cases} \max \left\{ \begin{array}{c} S_{MAX}(i,j,k), \\ S_P(i, j+1, k) \end{array} \right\} & \text{if } i_0 \le i < j < k, \\ S_P(i, j+1, k) & \text{if } i_0 \le i \ge j < k. \end{cases} \tag{II.14}$$

$$\begin{aligned} & S'_{pseudo}(i_0, k_0) \\ & = \max \left\{ \begin{array}{c} S'_{pseudo}(i_0, k_0 - 1), \\ \max\limits_{i_0 \le i < k_0 - 1} \Big\{ S_P(i, i_0 + 1, k_0) \Big\} \end{array} \right\} \text{ if } k_0 \ge i_0 + 2. \end{aligned} \tag{II.15}$$

We claim that recurrence II.15 computes exactly the same values as recurrence II.12.

Claim 2.3: For $1 \le i_0 \le k_0 - 2 \le n - 2$, $S'_{pseudo}(i_0, k_0) = S_{pseudo}(i_0, k_0)$.

Proof: (sketch) We obtain the following by simplifying recurrence II.14.

$$\begin{aligned} & S_P(i,j,k) \\ & = \begin{cases} \max\limits_{\max\{i+1, j\} \le j' < k} \Big\{\ S_{MAX}(i, j', k)\ \Big\} & \text{if } i_0 \le i\ \wedge\ j < k, \\ -\infty & \text{otherwise.} \end{cases} \end{aligned}$$

Therefore,

$$\max_{i_0 \le i < k_0 - 1} \{S_P(i, i_0 + 1, k_0)\} = \max_{i_0 \le i < j < k_0} \{\ S_{MAX}(i, j, k_0)\ \}$$

We can now evaluate $S'_{pseudo}(i_0, k_0)$ by induction on k_0. For $k_0 \ge i_0 + 2$,

$$\begin{aligned} S'_{pseudo}(i_0, k_0) &= \max \left\{ \begin{array}{c} S'_{pseudo}(i_0, k_0 - 1), \\ \max\limits_{i_0 \le i < k_0 - 1} \Big\{ S_P(i, i_0 + 1, k_0) \Big\} \end{array} \right\} \\ &= \max \left\{ \begin{array}{c} \max\limits_{i_0 \le i < j < k \le k_0 - 1} \Big\{\ S_{MAX}(i,j,k)\ \Big\}, \\ \max\limits_{i_0 \le i < j < k_0} \Big\{\ S_{MAX}(i, j, k_0)\ \Big\} \end{array} \right\} \\ &= \max_{i_0 \le i < j < k \le k_0} \Big\{\ S_{MAX}(i,j,k)\ \Big\} \\ &= S_{pseudo}(i_0, k_0) \end{aligned}$$

■

Now observe that in order to evaluate recurrence II.15 we only need the values $S_P(i,j,k)$ for $j = i_0 + 1$, and each entry (i,j,k) in recurrences II.8 - II.11 and II.14 depends only on entries $(\cdot, j, \cdot)$ and $(\cdot, j+1, \cdot)$. Therefore, we will evaluate the recurrences for $j = n$ first, then for $j = n - 1$, and continue downto $j = i_0 + 1$. Observe that in order to evaluate for $j = j'$ we only need to retain the $\mathcal{O}\left(n^2\right)$ entries computed for $j = j' + 1$. Thus for a fixed i_0 all $S_P(i, i_0 + 1, k)$ and consequently all relevant $S'_{pseudo}(i_0, k_0)$

Machine	Processors	Speed	L1 Cache	L2 Cache	RAM
Intel P4 Xeon	2	3.06 GHz	8 KB (4-way, $B = 64$ B)	512 KB (8-way, $B = 64$ B)	4 GB
AMD Opteron 250	2	2.4 GHz	64 KB (2-way, $B = 64$ B)	1 MB (8-way, $B = 64$ B)	4 GB
AMD Opteron 850	8	2.2 GHz	64 KB (2-way, $B = 64$ B)	1 MB (8-way, $B = 64$ B)	32 GB

TABLE I
MACHINES USED FOR EXPERIMENTS; ON ALL MACHINES ONLY 1 PROCESSOR WAS USED.

Algorithm	Comments	Time	Space	Cache Misses
PA-CO	our cache-oblivious algorithm (see Section II-D.1)	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(n\right)$	$\mathcal{O}\left(\frac{n^2}{BM}\right)$
PA-FASTA	linear-space implementation of Gotoh's algorithm [34] available in *fasta2* [35]	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(n\right)$	$\mathcal{O}\left(\frac{n^2}{B}\right)$

TABLE II
PAIRWISE SEQUENCE ALIGNMENT ALGORITHMS USED IN OUR EXPERIMENTS.

can be computed using only $\mathcal{O}\left(n^2\right)$ space, and the same space can be reused for all n values of i_0.

Cache-oblivious Algorithm. The evaluation of recurrences II.8 - II.11 and II.14 can be viewed as evaluating a single $n \times n \times n$ matrix c with five fields: S_L, S_R, S_M, S_{MAX} and S_P. If we replace all j with $n-j+1$ in the resulting recurrence it conforms to recurrence II.3 for $d = 3$. Therefore, for any fixed i_0 we can use our cache-oblivious algorithm to compute all entries $S_P(i, i_0 + 1, k)$ and consequently all relevant $S'_{pseudo}(i_0, k_0)$ values. in $\mathcal{O}\left(n^3\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache-misses. All $S'_{pseudo}(i_0, k_0)$ values can be computed by n applications (i.e., once for each i_0) of the algorithm.

For any given pair (i_0, k_0) the pseudoknot with the optimal score can be traced back cache-obliviously in $\mathcal{O}\left(n^3\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ cache-misses using our algorithm. Thus from Theorem 2.2 we obtain the following claim.

Claim 2.4: Given an RNA sequence of length n, a secondary structure that has the maximum number of base pairs in the presence of simple pseudoknots can be computed cache-obliviously in $\mathcal{O}\left(n^4\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(\frac{n^4}{B\sqrt{M}}\right)$ cache misses.

Extensions. In [3] the basic dynamic program for simple pseudoknots has been extended to handle energy functions based on adjacent base pairs within the same time and space bounds. Our cache-oblivious technique as described above can be adapted to solve this extension within the same improved bounds as for the basic DP. An $\mathcal{O}\left(n^{4-\delta}\right)$ time approximation algorithm for the basic DP has also been proposed in [3], and our techniques can be used to improve the space and cache complexity of the algorithm to $\mathcal{O}\left(n^2\right)$ (from $\mathcal{O}\left(n^3\right)$) and $\mathcal{O}\left(\frac{n^{4-\delta}}{B\sqrt{M}}\right)$ (from $\mathcal{O}\left(\frac{n^{4-\delta}}{B}\right)$), respectively.

III. EXPERIMENTAL RESULTS

In this section we present experimental results for the three bioinformatics applications discussed in section II-D. We used the machines listed in Table I for our experiments. All machines ran Ubuntu Linux 5.10. All our algorithms were implemented in C++ (compiled with *g++* 3.3.4), while some software packages we used for comparison were written in C (compiled with *gcc* 3.3.4). Optimization parameter -O3 was used in all cases. Each machine was exclusively used for experiments, and only one processor was used. The *Cachegrind* profiler [40] was used for simulating cache effects.

In order to reduce the overhead of recursion in the cache-oblivious algorithms, in our implementations we did not run the recursion all the way down to sequence length $r = 1$. Instead we stopped the recursion at a larger value of r, and solved the subproblem at that size using the traditional iterative method. This is a commonly-used methodology: if we were to keep the base case size at 1, the cost of the overheads associated with a recursive call would far exceed the useful computation performed during execution of the base case. Note that this unrelated to the cache-size, except that one would want to use a base-case size that is sufficiently small so that it does not overflow the cache. The values of r that we used were $r = 256$ for pairwise alignment (PA-CO), and $r = 64$ for median (MED-CO) and for RNA secondary structure with simple pseudo-knots (RNA-CO). Details on the iterative methods used to solve the base cases can be found in the theses [8], [30].

Our cache-oblivious algorithms outperformed currently available software and methods for all three applications. We describe details of our experimental results below.

A. Pairwise Global Sequence Alignment with Affine Gap Penalty

We performed experimental evaluation of the implementations listed in Table II: PA-CO is our implementation of our linear-space cache-oblivious algorithm given in Section II-D.1, and PA-FASTA is the implementation of the linear-space version of Gotoh's algorithm [34] available in the *fasta2* package [35].

In most cases, for input sequence sizes ranging from 1,000 to over a million, PA-FASTA ran about 20%-30% slower than PA-CO on AMD Opteron and up to 10% slower on Intel Xeon. We note that most bioinformatics applications use relatively short sequences of length less than $10,000$, and this sequence length falls within the range considered in Figure 4, though our experiments also considered much longer sequence pairs.

Random Sequences. We ran on randomly generated equal-length string-pairs over $\{\ A, C, G, T\ \}$ on AMD Opteron 250 (see Figure 4(a)) and Intel P4 Xeon (see Figure 4(b)). We varied string lengths from 1 K to 512 K. In our experiments PA-FASTA always ran slower than PA-CO on AMD Opteron (around 27% slower for sequences of length 512 K) and generally the relative

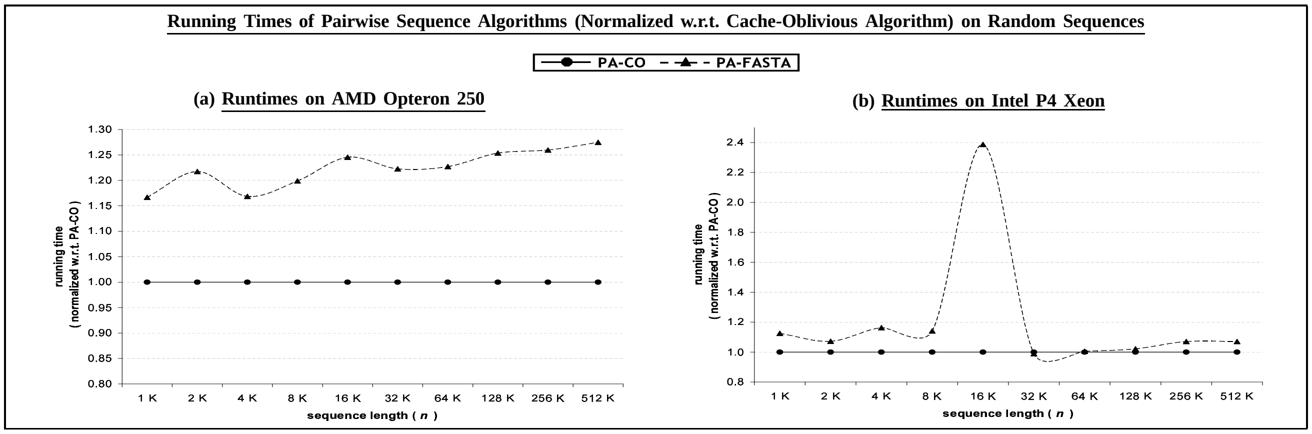

Fig. 4. Pairwise alignment on random sequences: All running times are normalized w.r.t. that of PA-CO. Each data point is the average of 3 independent runs on randomly generated strings over $\{ A, T, G, C \}$.

Sequence pairs with lengths (10^6)	Running times of pairwise alignment algorithms on *CFTR DNA Sequences* [41] (on AMD Opteron)		
	PA-CO (t_1)	PA-FASTA (t_2)	ratio (t_2/t_1)
human/baboon (1.80/1.51)	$17h\ 23m$	$20h\ 34m$	1.18
human/chimp (1.80/1.32)	$15h\ 25m$	$19h\ 51m$	1.29
baboon/chimp (1.51/1.32)	$12h\ 43m$	$16h\ 43m$	1.31
human/rat (1.80/1.50)	$18h\ 16m$	$24h\ \ 1m$	1.31
rat/mouse (1.50/1.49)	$13h\ 55m$	$16h\ 49m$	1.21

TABLE III

PAIRWISE ALIGNMENT ALGORITHM REAL DATA: EACH ENTRY IN COLUMNS 2 AND 3 IS THE TIME FOR A SINGLE RUN.

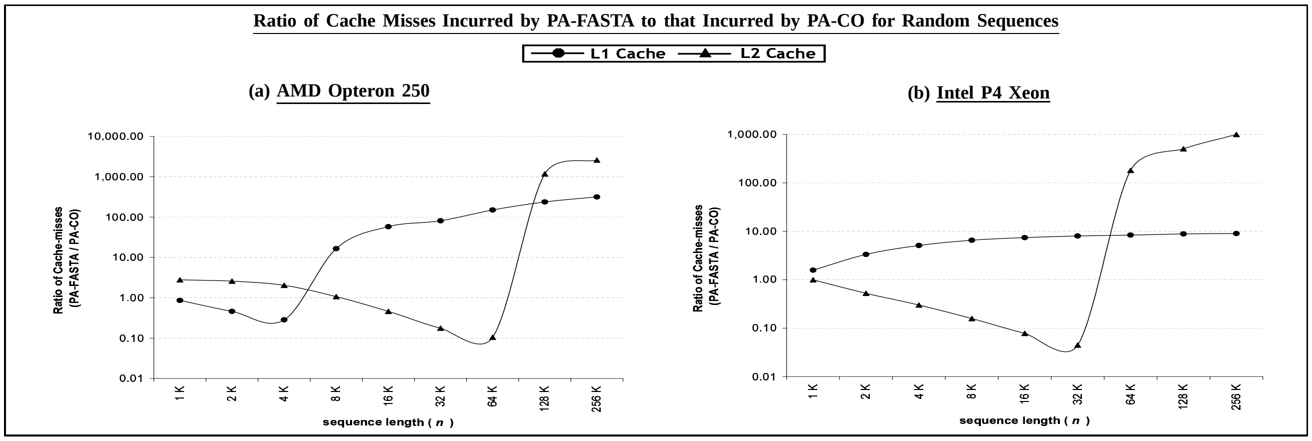

Fig. 5. Ratio of cache-misses incurred by PA-FASTA to that incurred by PA-CO (see Table II) on random sequences for both L1 and L2 caches. Data was obtained using Cachegrind [40].

performance of PA-CO improved over PA-FASTA as the length of the sequences increased. The trend was almost similar on Intel Xeon except that the improvement of PA-CO over PA-FASTA was more modest. We also obtained some anomalous results for $n \approx 10,000$ which we believe is due to architectural affects (cache misalignment of data in PA-FASTA).

Real-World Sequences. We ran PA-CO and PA-FASTA on CFTR DNA sequences of lengths between 1.3 million to 1.8 million [41] on AMD Opteron, and PA-FASTA ran 20%-30% slower than PA-CO on these sequences (see Table III). Though proper alignment of these genomic sequences require more sophisticated cost functions, running times of PA-CO and PA-FASTA on these sequences give us some idea on the relative performance of these implementations on very long sequences.

Cache Performance. We measured the number of L1 and L2 cache-misses incurred by both PA-FASTA and PA-CO on random sequences (see Figure 5). Though PA-FASTA causes fewer cache-misses than PA-CO when the input fits into the cache, it incurs significantly more misses than PA-CO as the input size grows beyond cache size. On AMD Opteron PA-FASTA incurs up to 300 times more L1 misses and 2500 times more L2 misses than PA-CO while on Intel Xeon the figures are 10 and 1000, respectively. Observe that the larger the cache the larger the cache-miss ratio which follows theoretical predictions since we know that PA-CO should incur fewer cache-misses on larger caches while cache performance of PA-FASTA should be independent of cache size.

B. Median of Three Sequences

In this section we report our experimental results for the median problem (i.e., the problem of determining 3-way global

Algorithm	Comments	Time	Space	Cache Misses
MED-CO	our cache-oblivious algorithm (see Section II-D.2)	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$
MED-Knudsen	Knudsen's implementation of his algorithm [27]	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(\frac{n^3}{B}\right)$
MED-ukk.alloc	Powell's implementation [37] of an $\mathcal{O}\left(\delta^3\right)$-space Ukkonen-based algorithm (δ = 3-way edit distance of sequences)	$\mathcal{O}\left(n+\delta^3\right)$ (avg.)	$\mathcal{O}\left(n+\delta^3\right)$	$\mathcal{O}\left(\frac{\delta^3}{B}\right)$
MED-ukk.checkp	Powell's implementation [37] of an $\mathcal{O}\left(\delta^2\right)$-space Ukkonen-based algorithm (δ = 3-way edit distance of sequences)	$\mathcal{O}\left(n\log\delta+\delta^3\right)$ (avg.)	$\mathcal{O}\left(n+\delta^2\right)$	$\mathcal{O}\left(\frac{\delta^3}{B}\right)$
MED-H	our implementation of MED-Knudsen with Hirschberg's space-reduction (used in study of space reduction and cache efficiency)	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(\frac{n^3}{B}\right)$

TABLE IV

MEDIAN ALGORITHMS USED IN OUR EXPERIMENTS.

sequence alignment with affine gap penalty). We performed experimental evaluation of the implementations listed in Table IV: MED-CO implements our quadratic-space cache-oblivious median algorithm described in Section II-D.2, MED-Knudsen is Knudsen's cubic-space median algorithm [28] implemented by Knudsen himself [27], MED-ukk.alloc is the $\mathcal{O}\left(n+\delta^3\right)$-space (where δ is the 3-way edit distance of sequences) Ukkonen-based median algorithm described in [38], and MED-ukk.checkp is the space-reduced version of MED-ukk.alloc based on checkpointing technique. Both Ukkonen-based algorithms were implemented by Powell [37]. Finally, MED-H is our quadratic-space implementation of Knudsen's algorithm based on Hirschberg's space-reduction technique, which we consider at the end of this section.

We used a gap insertion cost of 3 (i.e., $g_i = 3$), a gap extension cost of 1 (i.e., $g_e = 1$) and a mismatch cost of 1 in all experiments.

We first compare the performance of our cache-oblivious algorithm MED-CO with the three publicly available code: MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp. Overall, MED-Knudsen ran about 1.5-2.5 times slower than MED-CO on both machines, and MED-ukk.alloc and MED-ukk.checkp were even slower. Furthermore, due to their high space overhead MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp could not be run on any machine for sequences longer than 640, while MED-CO ran to completion on sequences of length over 1,000. We summarize our results below.

Random Sequences. We ran all implementations on random (equal-length) sequences of length $64i$, $1 \leq i \leq 16$ on AMD Opteron (see Figure 6(a)) and Intel Xeon (see Figure 6(b)). On both machines MED-CO ran the fastest, and MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp crashed as they ran out of memory for sequences longer than 384, 256 and 640, respectively.

On Intel Xeon, MED-CO ran at least 1.45 times faster than MED-Knudsen. Both MED-ukk.alloc and MED-ukk.checkp ran at least 2 times slower than MED-CO for length 64, and continued to slow down even further with increasing sequence length. They ran up to 3.3 times (for length 256) and 4.8 times (for length 640) slower than MED-CO, respectively. The trends were similar on AMD Opteron and MED-CO ran at least 2.5, 3.4 and 4.2 times faster than MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp, respectively.

Real-World Sequences. We ran the algorithms on triplets of 16S bacterial rDNA sequences from the *Pseudanabaena group* [16] (see Table V).

The triplets in Table V were formed by choosing at random three sequences of length less than 500 from the group. We report results for the same 5 triplets on both the Intel Xeon and AMD Opteron 850. The trends were very similar on both machines.

The results show that our cache-oblivious algorithm MED-CO has the best performance, since it runs to completion on all triplets and is considerably faster than the other 3 methods in most cases.

The results also closely track theoretical predictions for all 4 algorithms. The theoretical analysis shows that the number of steps, space usage and cache misses for both MED-Knudsen and MED-CO should increase with the product of the lengths of the three sequences, and we note the overall running time for both of these algorithms increases with this product on both the Intel and AMD machines. However, MED-Knudsen ran around 35–50% slower than MED-CO on the Xeon and over twice as slow on the Opteron 850.

On the other hand, the resource usage of the two Ukkonen-based methods increases with the alignment cost, and hence these methods are best suited for sequences with small alignment cost. This again shows up in our results: On the Opteron 850, MED-ukk.alloc is actually faster than our cache-oblivious MED-CO on the triple 2, which has smallest alignment cost, but because of its cubic space dependence on alignment cost, it is unable to run to completion on the other triples. The more space-efficient MED-ukk.checkp runs to completion on all triples, but its performance degrades with the alignment cost, and even for triple 2, it runs about 20% slower than MED-CO.

We ran the same triples on Opteron 850 with mismatch cost set to 2 instead of 1, and we summarize the results here. This increases the alignment cost, and as expected, the two Ukkonen-based methods degraded significantly: MED-ukk.alloc did not complete on any of the 5 triples, and MED-ukk.checkp ran 5 to 10 times slower than MED-CO. Also as expected, the runtimes for MED-Knudsen and MED-CO were virtually unchanged from the timings in Table V.

Overall, our experimental results suggest that MED-CO is always a better choice than MED-Knudsen, and a better choice than the two Ukkonen-based algorithms (MED-ukk.alloc and MED-ukk.checkp) except when the alignment cost is very small.

Effects of Space-reduction and Cache-efficiency. In Figures 7(a) and 7(b) we plot the running times on triples of random sequences of Knudsen's algorithm (MED-Knudsen), our implementation of a Hirschberg-style space-reduced version of the same algorithm (MED-H), and our space-efficient cache-oblivious algorithm (MED-CO). As the plots show, after simply reducing the space usage from $\mathcal{O}\left(n^3\right)$ (MED-Knudsen) to $\mathcal{O}\left(n^2\right)$ (MED-H), the median algorithm runs faster and can handle much longer sequences. For space-intensive algorithms reducing space usage can improve its cache performance significantly since now the

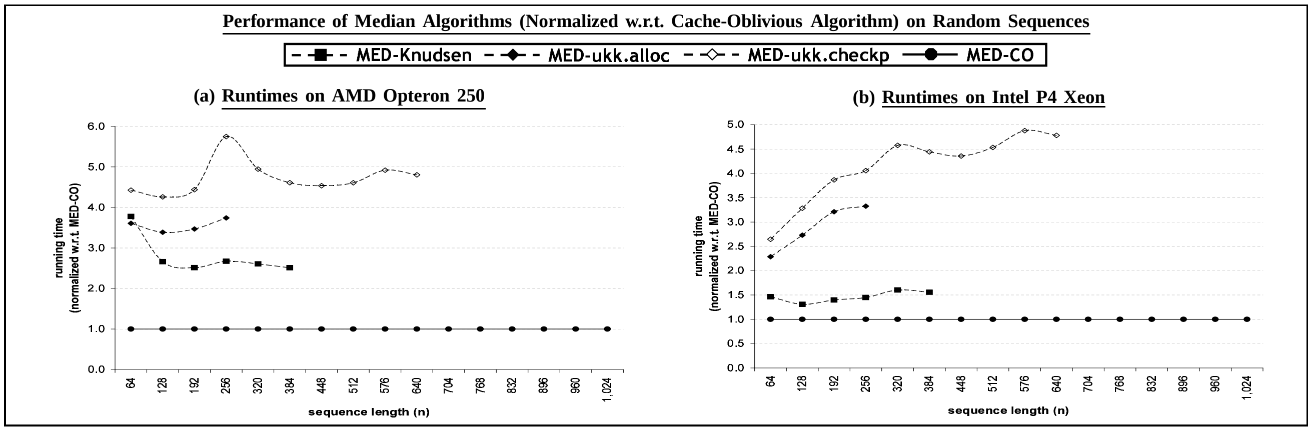

Fig. 6. Median on random data: Each data point is the average of 3 independent runs on random strings over $\{ A, T, G, C \}$.

Random triples of *16S Bacterial rDNA Sequences* from the *Pseuanabaena Group* [16]							
Intel Xeon: Running times in seconds (runtime w.r.t. MED-CO)							
No.	Lengths	Product of Lengths	MED-CO	MED-Knudsen	MED-ukk.alloc	MED-ukk.checkp	Alignment Cost
1	342, 367, 389	$\approx 49 \times 10^6$	451 (1.00)	611 (1.35)	– (–)	863 (1.91)	339
2	367, 387, 388	$\approx 55 \times 10^6$	487 (1.00)	722 (1.48)	512 (1.05)	601 (1.23)	299
3	378, 388, 403	$\approx 59 \times 10^6$	529 (1.00)	752 (1.42)	– (–)	769 (1.45)	324
4	342, 370, 474	$\approx 60 \times 10^6$	531 (1.00)	764 (1.44)	– (–)	1,701 (3.20)	432
5	370, 388, 447	$\approx 64 \times 10^6$	553 (1.00)	824 (1.49)	– (–)	– (–)	336
Opteron 850: Running times in seconds (runtime w.r.t. MED-CO)							
No.	Lengths	Product of Lengths	MED-CO	MED-Knudsen	MED-ukk.alloc	MED-ukk.checkp	Alignment Cost
1	342, 367, 389	$\approx 49 \times 10^6$	445 (1.00)	937 (2.11)	– (–)	831 (1.87)	339
2	367, 387, 388	$\approx 55 \times 10^6$	493 (1.00)	1,057 (2.14)	427 (0.87)	572 (1.16)	299
3	378, 388, 403	$\approx 59 \times 10^6$	528 (1.00)	1,133 (2.15)	– (–)	740 (1.40)	324
4	342, 370, 474	$\approx 60 \times 10^6$	528 (1.00)	1,151 (2.18)	– (–)	1,636 (3.10)	432
5	370, 388, 447	$\approx 64 \times 10^6$	562 (1.00)	– (–)	– (–)	798 (1.42)	336

TABLE V

MEDIAN ON REAL DATA: THE TRIPLETS WERE FORMED BY CHOOSING RANDOM SEQUENCES OF LENGTH LESS THAN 500. EACH NUMBER OUTSIDE PARENTHESES IN COLUMNS 4–7 IS THE TIME FOR A SINGLE RUN, AND THE RATIO OF THAT RUNNING TIME TO THE CORRESPONDING RUNNING TIME FOR MED-CO IS GIVEN WITHIN PARENTHESES. A '–' IN A COLUMN DENOTES THAT THE CORRESPONDING ALGORITHM COULD NOT BE RUN DUE TO HIGH SPACE OVERHEAD.

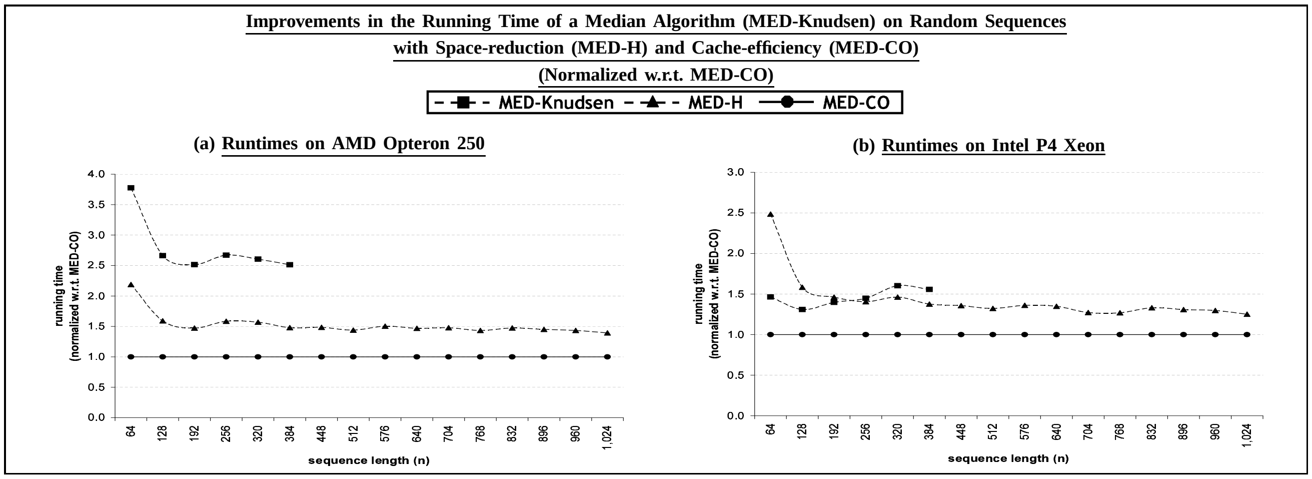

Fig. 7. Improvements in the performance of a median algorithm (i.e., MED-Knudsen: Knudsen's implementation of his algorithm [28]) as its space requirement is reduced (with a Hirschberg-style implementation of Knudsen's algorithm (MED-H)), and as both its space usage and cache performance are improved using our cache-oblivious median algorithm (MED-CO). Each data point is the average of 3 independent runs on random strings over $\{ A, T, G, C \}$.

Algorithm	Comments	Time	Space	Cache Misses
RNA-CO	cache-oblivious algorithm (see Section II-D.3)	$\mathcal{O}\left(n^4\right)$	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(\frac{n^4}{B\sqrt{M}}\right)$
RNA-CS	Akutsu's original cubic-space algorithm [3]	$\mathcal{O}\left(n^4\right)$	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(\frac{n^4}{B}\right)$
RNA-QS	our iterative quadratic -space implementation of Akutsu's algorithm (see Section II-D.3)	$\mathcal{O}\left(n^4\right)$	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(\frac{n^4}{B}\right)$

TABLE VI

ALGORITHMS FOR RNA SECONDARY STRUCTURE PREDICTION USED IN OUR EXPERIMENTS.

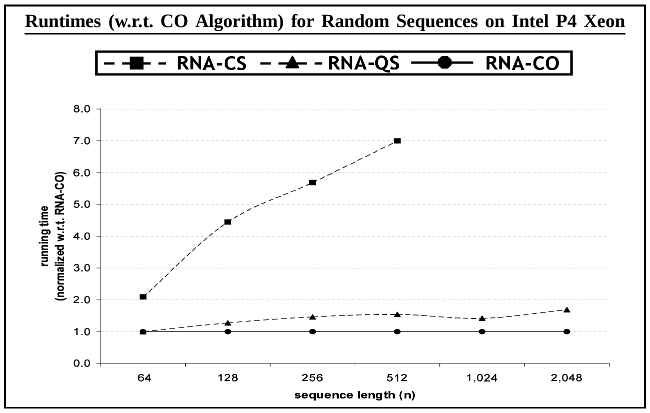

Fig. 8. RNA secondary structure prediction with simple pseudoknots on random data: All running times are normalized w.r.t. RNA-CO. Each data point is the average of 3 independent runs on random strings over $\{A, U, G, C\}$.

Runtimes of algorithms for RNA secondary structure prediction with simple pseudoknots on Intel Xeon for *Bacterial (Spirochaetes) 16S rRNA Sequences* [7]				
Organism	Length (n)	Cache-oblivious (RNA-CO)	Quadratic Space (RNA-QS)	RNA-QS/RNA-CO
Brevinema andersonii	1443	$1h\ 22m$	$2h\ 14m$	1.64
Borrelia burgdorferi	1530	$1h\ 44m$	$2h\ 48m$	1.62
Borrelia burgdorferi	1537	$1h\ 45m$	$2h\ 48m$	1.60
Borrelia hermsii	1523	$1h\ 41m$	$2h\ 43m$	1.61
Brachyspira hyodysenteriae	1463	$1h\ 27m$	$2h\ 21m$	1.61
Cristispira CP1	1491	$1h\ 33m$	$2h\ 30m$	1.62
Leptonema illini	1526	$1h\ 42m$	$2h\ 45m$	1.61
Leptospira interrogans	1508	$1h\ 38m$	$2h\ 37m$	1.61
Spirochaeta aurantia	1520	$1h\ 41m$	$2h\ 42m$	1.60
Treponema pallidum (rRNA A)	1549	$1h\ 48m$	$2h\ 54m$	1.60
Average	1509	$1h\ 38m$	$2h\ 38m$	1.61

TABLE VII

RNA SECONDARY STRUCTURE PREDICTION WITH SIMPLE PSEUDOKNOTS ON REAL DATA: INPUTS ARE BACTERIAL (SPIROCHAETES) 16S RRNA SEQUENCES [7] WITH AN AVERAGE LENGTH OF 1509. EACH NUMBER IN COLUMNS 3 AND 4 REPRESENTS TIME FOR A SINGLE RUN.

data fits in lower cache levels and thus incurs fewer cache-misses.

Although we were able to improve the performance of Knudsen's algorithm significantly by reducing its space usage to $O(n^2)$ we observe that our cache-oblivious, space-efficient algorithm for median has better performance. On AMD Opteron the running time of Knudsen's algorithm improves by 40% after space-reduction (MED-H), and our cache-oblivious implementation (MED-CO) gives a *further* 30% improvement. A similar trend is observed on Intel P4 Xeon.

C. RNA Secondary Structure Prediction with Pseudoknots

We implemented the algorithms in Table VI for computing all values of S_{pseudo} or S'_{pseudo} (i.e., we compute the optimal scores only, we do not traceback the pseudoknots). We ran all experiments on Intel Xeon using a single processor.

Overall, RNA-QS ran about 50% slower than RNA-CO and RNA-CS ran up to 7 times slower than RNA-CO for sequence lengths it could handle. For sequences longer than 512 RNA-CS could not be run due to lack of memory space. We summarize our results below.

Random Sequences. We ran all three algorithms on randomly generated string-pairs over $\{A, U, G, C\}$ (see Figure 8). The lengths of the strings were varied from 64 to 2048. However, due to lack of space RNA-CS could not be run for strings longer than 512. In our experiments RNA-CO ran the fastest while RNA-CS was the slowest. Though both RNA-QS and RNA-CS have the same time and cache-complexity, RNA-QS ran significantly faster than RNA-CS (e.g., ≈ 4.5 times faster for length 512). We believe this happened because even for small sequence lengths RNA-CS overflows the L2 cache, and most of its data reside in the slower RAM, while both RNA-QS and RNA-CO still work completely inside the faster L2 cache. For strings of length 512 RNA-CO ran about 35% faster than RNA-QS and about 7 times faster than RNA-CS. The performance of RNA-CO improved over that of both RNA-CS and RNA-QS as the length increased.

Real-World Sequences. We ran all three implementations on a set of 24 bacterial 5S rRNA sequences obtained from [7]. The average length of the sequences was 118, and the average running times of RNA-CS, RNA-QS and RNA-CO on each sequence were 1.46 sec, 0.45 sec and 0.35 sec, respectively. We also ran RNA-QS and RNA-CO on a set of 10 bacterial (spirochaetes) 16S rRNA sequences of average length 1509. The RNA-CS implementation could not be run on these sequences due to space limitations. On these sequences RNA-CO took 1 hour 38 minutes while RNA-QS took 2 hours 38 minutes on the average (see Table III-B).

IV. Conclusion

In this paper we have presented a general cache-oblivious framework for a class of widely encountered dynamic programming problems with local dependencies, and applied it to obtain efficient cache-oblivious algorithms for three important string problems in bioinformatics, namely global pairwise sequence alignment and median (both with affine gap costs), and RNA secondary structure prediction with simple pseudoknots. We have shown that our algorithms are faster, both theoretically and experimentally, than previous algorithms for these problems.

Our framework can be applied to several other dynamic programming problems in bioinformatics including local alignment, generalized global alignment with intermittent similarities, multiple sequence alignment under several scoring functions such as 'sum-of-pairs' objective function and RNA secondary structure prediction with simple pseudoknots using energy functions based on adjacent base pairs.

Acknowledgment

We thank Mike Brudno for the CFTR DNA sequences, Robin Gutell for the rRNA sequences, and David Zhao for the MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp code.

References

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

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

[3] T. Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Appl. Math.*, 104:45–62, 2000.

[4] S. Altschul and B. Erickson. Optimal sequence alignment using affine gap costs. *Bull. Math. Biol.*, 48:603–616, 1986.

[5] A. Apostolico, S. Browne, and C. Guerra. Fast linear-space computations of longest common subsequences. *Theor. Comp. Sci.*, 92(1):3–17, 1992.

[6] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proc. 7th SPIRE*, pages 39–48, 2000.

[7] J. Cannone, S. Subramanian, M. Schnare, J. Collett, L. D'Souza, Y. Du, B. Feng, N. Lin, L. Madabusi, K. Muller, N. Pande, Z. Shang, N. Yu, and R. Gutell. The comparative RNA web (CRW) site: An online database of comparative sequence and structure information for ribosomal, intron, and other RNAs. *BioMed Central Bioinform.*, 3:2, 2002. url: http://www.rna.icmb.utexas.edu/.

[8] R. Chowdhury. *Algorithms and Data Structures for Cache-efficient Computation: Theory and Experimental Evaluation*. PhD thesis, Dept. of Computer Sciences, The University of Texas at Austin, 2007.

[9] R. Chowdhury, H. Le, and V. Ramachandran. Efficient cache-oblivious string algorithms for Bioinformatics. Technical Report TR-07-03, Dept. of Computer Sciences, University of Texas at Austin, February 2007.

[10] R. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proc. 17th ACM-SIAM SODA*, pages 591–600, 2006.

[11] R. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: Theoretical framework and experimental evaluation. Technical Report TR-06-04, Dept. of Computer Sciences, University of Texas at Austin, March 2006.

[12] R. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. In *Proc. 19th ACM SPAA*, pages 71–80, 2007.

[13] R. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores (to appear). In *Proc. 20th ACM SPAA*, June 2008.

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

[15] M. Crochemore, G. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM J. Comput.*, 32(6):1654–1673, 2003.

[16] T. DeSantis, I. Dubosarskiy, S. Murray, and G. Andersen. Comprehensive aligned sequence construction for automated design of effective probes (cascade-p) using 16S rDNA. *Bioinform.*, 19:1461–1468, 2003. url: http://greengenes.llnl.gov/16S/.

[17] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.

[18] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annu. IEEE Symp. FOCS*, pages 285–297, 1999.

[19] M. Frigo and V. Strumpen. Cache-oblivious stencil computations. In *Proc. 19th ACM ICS*, pages 361–366, 2005.

[20] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.

[21] J. Grice, R. Hughey, and D. Speck. Reduced space sequence alignment. *Comput. Appl. Biosci.*, 13(1):45–53, 1997.

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

[23] D. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

[24] D. Hirschberg. An information theoretic lower bound for the longest common subsequence problem. *Inf. Process. Lett.*, 7(1):40–41, 1978.

[25] J. Hong and H. Kung. I/O complexity: the red-blue pebble game. In *Proc. 13th Annu. ACM STOC*, pages 326–333, 1981.

[26] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesely, 2005.

[27] B. Knudsen. Multiple parsimony alignment with "affalign". Software package `multalign.tar`.

[28] B. Knudsen. Optimal multiple parsimony alignment with affine gap cost using a phylogenetic tree. In *Proc. WABI*, pages 433–446, 2003.

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

[30] H. Le. Algorithms for identication of patterns in biogeography and median alignment of three sequences in bioinformatics, 2006. Undergraduate Honors Thesis, CS-TR-06-29, The University of Texas at Austin, Dept. of Computer Sciences.

[31] R. Lyngsø and C. Pedersen. RNA pseudoknot prediction in energy-based models. *J. Comput. Biol.*, 7(3-4):409–427, 2000.

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

[33] W. Masek and M. Paterson. A faster algorithm for computing string edit distances. *J. Comput. Sys. Sci.*, 20(1):18–31, 1980.

[34] E. Myers and W. Miller. Optimal alignments in linear space. *Comput. Appl. Biosci.*, 4(1):11–17, 1988.

[35] W. Pearson and D. Lipman. Improved tools for biological sequence comparison. In *Proc. Natl. Acad. Sci. U.S.A.*, volume 85, pages 2444–2448, 1988.

[36] C. Pedersen. *Algorithms in Computational Biology*. PhD thesis, Dept. of Computer Science, University of Aarhus, Denmark, 1999.

[37] D. Powell. Software package `align3str_checkp.tar.gz`.

[38] D. Powell, L. Allison, and T. Dix. Fast, optimal alignment of three sequences using linear gap cost. *J. Theor. Biol.*, 207(3):325–336, 2000.

[39] E. Rivas and S. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *J. Mol. Biol.*, 285(5):2053–2068, 1999.

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

[41] J. Thomas et al. Comparative analyses of multi-species sequences from targeted genomic regions. *Nature*, 424:788–793, 2003.

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