Efficient Cache-oblivious String Algorithms for Bioinformatics *

Rezaul Alam Chowdhury Hai-Son Le Vijaya Ramachandran

UTCS Technical Report TR-07-03

February 5, 2007

Abstract

We present theoretical and experimental results on cache-efficient and parallel algorithms for some well-studied string problems in bioinformatics:

1. *Pairwise alignment.* Optimal pairwise global sequence alignment using affine gap penalty;
2. *Median.* Optimal alignment of three sequences using affine gap penalty;
3. *RNA secondary structure prediction.* Maximizing number of base pairs in RNA secondary structure with simple pseudoknots.

For each of these three 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 also show that these algorithms are easily parallelizable, and we analyze their parallel performance.

We present experimental results that show that all three cache-oblivious algorithms run faster on current desktop machines than the best previous algorithm for the problem. For the first two problems we compare our code to publicly available software written by others, and for the last problem our comparison is to our implementation of Akutsu's algorithm. We also include empirical results showing good performance for the parallel implementations of our algorithms for the first two problems.

Our methods are applicable to several other dynamic programs for string 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.

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

1 Introduction

Algorithms for sequence alignment and for RNA secondary structure are some of the most widely studied and widely-used methods in bioinformatics. Many of these algorithms are dynamic programs that run in polynomial time, and many have been further improved in their space usage, mainly using a technique due to Hirschberg [15]. However, most of these algorithms are deficient with respect to *cache-efficiency*.

Cache-efficiency and cache-oblivious algorithms. 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. Data is transferred in blocks between adjacent levels in order to amortize the access time cost.

The *two-level I/O model* [1] is a simple abstraction of the memory hierarchy that consists of an internal memory (or *cache*) of size M, and an arbitrarily large external memory partitioned into blocks of size B. The *I/O complexity* or *cache-complexity* of an algorithm is the number of blocks transferred between these two levels on a given input.

The *cache-oblivious model* [11] is an extension of the two-level model that assumes an *ideal cache*, i.e., assumes that an optimal cache replacement policy is used, and requires that algorithms do not use knowledge of M and B in their description. A well-designed cache-oblivious algorithm is flexible and portable, and simultaneously adapts to all levels of a multi-level memory hierarchy. Standard cache replacement methods such as LRU allow for a reasonable approximation to an ideal cache.

Our Results. In this paper we present an efficient cache-oblivious framework for solving a general class of recurrence relations in 2- and 3-dimensions that are amenable to solution by dynamic programs with 'local dependencies' (as described in Section 2). In principle our framework can be generalized to any number of dimensions, although we study explicitly only the 2- and 3-dimensional cases. We use this framework to develop cache-oblivious algorithms for three well-known string problems in bioinformatics, and show that our algorithms are faster, both theoretically and experimentally, than previous algorithms for these problems. We also show that these cache-oblivious algorithms can be parallelized with little effort, and present theoretical and experimental results on their parallel performance. The string problems we consider are:

- *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 $\mathcal{O}\left(n^2/(BM)\right)$ cache-misses. When executed with p processors a simple parallel implementation of this algorithm performs $\mathcal{O}\left(n^2\right)$ work and terminates in $\mathcal{O}\left(\frac{n^2}{p} + n^{\log_2 3}\log n\right)$ parallel steps.
- *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 $\mathcal{O}\left(n^3/(B\sqrt{M})\right)$ cache-misses on three sequences of length n each. On a machine with p processors a simple parallel version of our algorithm performs $\mathcal{O}\left(n^3\right)$ work and executes $\mathcal{O}\left(\frac{n^3}{p} + n^2\log n\right)$ parallel steps.
- *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 $\mathcal{O}\left(n^4/(B\sqrt{M})\right)$ cache-misses, and parallel implementation of our algorithm performs $\mathcal{O}\left(n^4\right)$ work and terminates in $\mathcal{O}\left(\frac{n^4}{p} + n^2\log n\right)$ parallel steps when executed with p processors.

We defer to Sections 4 and 5 a discussion of previous algorithms known for these three problems.

The results in this paper extend and generalize our earlier work in [6], where we presented a relatively simple $\mathcal{O}\left(n^2\right)$ time and $\mathcal{O}\left(n^2/(BM)\right)$ I/O cache-oblivious dynamic program for finding the longest common subsequence (LCS) of two sequences and more involved cache-oblivious dynamic programs that run in $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(n^3/(B\sqrt{M})\right)$ I/O's for other problems including the problems of finding pairwise sequence alignment with general gap costs and RNA secondary structure without pseudoknots.

Two features of our cache-oblivious algorithms are worth noting.

- 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' space-reduced version [15] of these traditional dynamic programs. However, our space reduction is obtained through a divide-and-conquer strategy that is quite different from the method used in [15]. Briefly, our method computes the dynamic program table recursively in sub-blocks and stores only the computed values at the *boundary* of the sub-block. This results in the space saving, and we show that the stored values suffice to compute a solution with optimal value.
- Our algorithms are simpler than the space-reduced versions of the traditional dynamic programs, and hence are easier to code. Further, the recursive structure of our method gives rise to a good amount of parallelizism, which is also very easy to expose using standard parallel constructs such as `fork` and `join`.

We note that often in practice biologists seek not a precise optimal solution but biologically significant solutions that may be sub-optimal under the optimization measure used to define the problem. However, in such cases, an algorithm for the precise optimal solution is often used as a subroutine in conjunction with other methods that determine biological features not captured by the combinatorial problem specification. Therefore, our algorithms are likely to be of use to biologists even when biologically significant solutions are sought that are not necessarily optimal under our definition.

Organization of the paper. In Section 2 we describe the general cache-oblivious strategy that will apply to all three problems we consider in this paper, and in Section 3 we describe and analyze its parallel implementation. In Sections 4.1, 4.2 and 4.3 we describe how to use the cache-oblivious strategy described in Section 2 to obtain cache-oblivious algorithms for global pairwise sequence alignment, median and RNA secondary structure prediction with simple pseudoknots, respectively. It should be noted here that once a problem is mapped to our cache-oblivious framework it also immediately implies a parallel cache-oblivious algorithm for the problem as described in Section 3. In Section 5 we present our experimental results for the three problems.

2 A Cache-Oblivious Framework for a Class of DP with Local Dependencies

Given two sequences $X = x_1x_2 \ldots x_n$ and $Y = y_1y_2 \ldots y_n$ (for simplicity of exposition we consider only equal-length sequences here), and two functions $h_2(\cdot)$ and $f_2(\cdot,\cdot,\cdot)$, we consider dynamic programs that compute the entries of a two-dimensional matrix $c[0:n, 0:n]$ using the following recurrence relation. Such recurrences occur frequently in computational biology [27, 14].

$$c[i,j] = \begin{cases} h_2\left(\langle\ i,\ j\ \rangle\right) & \text{if } i = 0 \text{ or } j = 0, \\ f_2\left(\langle\ i,\ j\ \rangle,\ \langle\ x_i,\ y_j\ \rangle,\ c[i-1:i, j-1:j] \setminus c[i,j]\right) & \text{otherwise.} \end{cases} \tag{2.1}$$

Function f_2 can be arbitrary except that it is allowed to use exactly one cell from $c[i-1:i, j-1:j] \setminus c[i,j]$ to compute the final value of $c[i,j]$ (although it can consider all 3 cells), and we call that

specific cell the *parent cell* of $c[i,j]$. We also assume that f_2 does not access any memory locations in addition to those passed to it as inputs except possibly some constant size local variables and arrays.

Typically, two types of outputs are expected from dynamic programs evaluating recurrence 2.1:

(i) the value of $c[n,n]$ (or sometimes the values of $c[n,1:n]$ and $c[1:n,n]$), and
(ii) the traceback path starting from $c[n,n]$ (or from some cell in $c[n,1:n]$ or $c[1:n,n]$).

A *traceback path* from a given cell in c is a path through c that starts at that given cell and reaches the first row or the first column of c by following the chain of parent cells.

Each cell of c can have multiple fields and in that case f_2 must compute a value for each field, though as before, it is allowed to use exactly one field from $c[i-1:i,\ j-1:j] \setminus c[i,j]$ to compute the final value of any given field in $c[i,j]$. The definition of traceback path extends naturally.

The recurrence given by 2.1 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. The space complexity can be reduced to $\mathcal{O}\left(n\right)$ using a space-reduction technique due to Hirschberg [15], while keeping the time and cache complexities unchanged. If only the values of the last row and the last column are required (not the traceback path) it is easy to reduce space requirement to $\mathcal{O}\left(n\right)$ even without using Hirschberg's technique (see, e.g., [5]), and the cache-complexity of the algorithm can be improved to $\mathcal{O}\left(n^2/(BM)\right)$ using the cache-oblivious stencil-computation technique from [12].

In [6] we considered two important special cases of recurrence 2.1, namely the recurrences for the *longest common subsequence* and the *basic edit distance* problem, and presented cache-oblivious algorithms for computing both $c[n,n]$ and the traceback path from $c[n,n]$ in $\mathcal{O}\left(n^2\right)$ time, $\mathcal{O}\left(n\right)$ space and $\mathcal{O}\left(n^2/(BM)\right)$ cache-misses.

The 3-dimensional variant of recurrence 2.1 also appears frequently in practice [27, 14]. Two particular problems in bioinformatics where this variant occurs are: median of three sequences and RNA secondary structure prediction with simple pseudoknots (we consider these two problems in Sections 4.2 and 4.3 of this paper). In the 3-dimensional variant we are given three sequences $X = x_1x_2\ldots x_n$, $Y = y_1y_2\ldots y_n$ and $Z = z_1z_2\ldots,z_n$, two functions $h_3(\cdot)$ and $f_3(\cdot,\cdot,\cdot)$, and we are required to fill-in the three-dimensional matrix $c[0:n,0:n,0:n]$ using the following recurrence relation:

$$c[i,j,k] = \begin{cases} h_3\left(\ \langle\ i,\ j,\ k\ \rangle\ \right) & \text{if } i=0 \text{ or } j=0 \text{ or } k=0, \\ f_3\left(\begin{array}{c} \langle\ i,j,k\ \rangle,\ \langle\ x_i,y_j,z_k\ \rangle, \\ c[i-1:i,j-1:j,k-1:k] \setminus c[i,j,k] \end{array}\right) & \text{otherwise.} \end{cases} \tag{2.2}$$

All definitions and conditions for recurrence 2.1 extend naturally to recurrence 2.2.

Straight-forward iterative evaluation of recurrence 2.2 requires $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(n^3\right)$ space, and incurs $\mathcal{O}\left(n^3/B\right)$ cache-misses. If the traceback path is not required the space and cache complexities of the algorithm can be reduced to $\mathcal{O}\left(n^2\right)$ and $\mathcal{O}\left(n^3/(B\sqrt{M})\right)$, respectively [12]. If the traceback path is required Hirschberg's space-reduction technique [15] can be used to reduce the space complexity to $\mathcal{O}\left(n^2\right)$, but the time and cache complexities remain $\mathcal{O}\left(n^3\right)$ and $\mathcal{O}\left(n^3/B\right)$, respectively.

In Section 2.1 we present a cache-oblivious algorithm for solving the general 3-dimensional recurrence 2.2 along with a traceback path in $\mathcal{O}\left(n^3\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(n^3/(B\sqrt{M})\right)$ cache misses. The new algorithm improves over the cache-complexity of all previous algorithms solving the same problem by at least a factor of $\sqrt{M}$, and reduces the space requirement by a factor of n when compared with the traditional iterative dynamic programming solution. In Sections 4.2 and 4.3 we use this algorithm to solve two important problems in bioinformatics, namely median of three sequences and RNA secondary structure prediction with simple pseudoknots.

For completeness, we present in the Appendix a cache-oblivious algorithm that solves the 2-dimensional recurrence 2.1 and computes a traceback path in $\mathcal{O}\left(n^2\right)$ time, $\mathcal{O}\left(n\right)$ space and $\mathcal{O}\left(\mathcal{O}\left(n^2/(BM)\right)\right)$ cache misses. It improves over the previous best cache-complexity by a factor of at least M, and also improves over the space complexity of the standard iterative DP by a factor of n. This algorithm is a generalization of the cache-oblivious algorithm we presented in [6] for solving the LCS recurrence (a special case of recurrence 2.1), and is simpler than the 3-dimensional case. In Section 4.1 we use this algorithm for global pairwise sequence alignment with affine gap costs.

2.1 Cache-Oblivious Algorithm for Solving the 3D Recurrence 2.2 with Traceback Path. Our algorithm works by decomposing the given cube $c[1:n, 1:n, 1:n]$ into smaller subcubes, and is based on the observation that for any such subcube we can compute the entries on its *output boundary* (i.e., on its right, front and bottom boundaries) provided we know the entries on its *input boundary* (i.e., entries immediately outside of its left, back and top boundaries). Since the subcubes share boundaries and the input boundary of c is known we can compute the input boundaries of all subcubes by recursively computing the output boundaries of the subcubes in an appropriate order. When the input boundaries of all subcubes are computed the problem of finding the traceback path through the entire cube is decomposed into smaller subproblems of finding the fragments of the path through the smaller subcubes which can be solved recursively. Though we compute all n^3 entries of c, at any stage of recursion we only need to save the entries on the boundaries of the subcubes and thus use only $\mathcal{O}\left(n^2\right)$ space. The divide and conquer strategy also improves locality of computation and consequently leads to an efficient cache-oblivious algorithm.

As noted before, Hirschberg's technique [15] can also be used to solve recurrence 2.2 along with a traceback path. Though Hirschberg's approach is also divide and conquer and has the same time and space complexities as ours, unlike our algorithm it always decomposes the problem into two subproblems of typically unequal size, and uses a complicated process involving the application of the traditional iterative DP in both forward and backward directions to perform the decomposition. The application of the iterative DP along with the fact that the subproblems are often unequal in size contributes to its inefficient cache usage.

Our algorithm is given in Figure 1. Before describing and analyzing our algorithm for finding a traceback path, we describe and analyze the method we use for computing the output boundaries of subcubes which will be used as a subroutine when we retrieve a traceback path. Our method for computing the output boundary has some similarity to the cache-oblivious stencil computation algorithm described in [12] which, however, uses a different decomposition scheme.

Computing the Output Boundary in 3D (Compute-Boundary-3D **in Figure 1).** Given the input boundary of $c[1:n, 1:n, 1:n]$ the function Compute-Boundary-3D recursively computes its output boundary. For simplicity of exposition we assume that $n = 2^q$ for some integer $q \geq 0$.

If c is a $1 \times 1 \times 1$ matrix, the function can compute the output boundary directly using recurrence 2.2, otherwise it decomposes its cubic computation space Q (initially $Q \equiv c[1:n, 1:n, 1:n]$) into 8 subcubes: $Q_{1,1,1}$ (left-back-top), $Q_{2,1,1}$ (right-back-top), $Q_{1,2,1}$, $Q_{2,2,1}$, $Q_{1,1,2}$, $Q_{2,1,2}$, $Q_{1,2,2}$ and $Q_{2,2,2}$ (right-front-bottom). It then computes the output boundary of each subcube recursively in the order given above. Observe that though the left boundary of $Q_{1,2,1}$ is not known initially, the recursive call on $Q_{1,1,1}$ earlier in the order computes this boundary. After all recursive calls terminate, the output boundary of Q is composed from the output boundaries of the subcubes.

FUNCTION 2.1. COMPUTE-BOUNDARY-3D($X,\ Y,\ Z,\ L,\ B,\ T$)

Input. See the input description of COMPUTE-TRACEBACK-PATH-3D.

Output. Returns an ordered triple $\langle R, F, D\rangle$, where R ($\equiv Q[r, 1:r, 1:r]$), F ($\equiv Q[1:r, r, 1:r]$) and D ($\equiv Q[1:r, 1:r, r]$) are the right, front and bottom boundaries of $Q[1:r, 1:r, 1:r]$, respectively.

1. ***if*** $r = 1$ ***then*** $R = F = D \leftarrow f_3(\ \langle\ u,\ v,\ w\ \rangle,\ \langle\ X,\ Y,\ Z\ \rangle,\ L \cup B \cup T\ )$
2. ***else***
3. Extract $L_{1,j,k}$ from L, $B_{i,1,k}$ from B, and $T_{i,j,1}$ from T, respectively, where $i, j, k \in [1, 2]$
4. $subcube[\ 1:8\ ] \leftarrow \langle\ \langle\ 1,1,1\ \rangle,\ \langle\ 2,1,1\ \rangle,\ \langle\ 1,2,1\ \rangle,\ \langle\ 2,2,1\ \rangle,\ \langle\ 1,1,2\ \rangle,\ \langle\ 2,1,2\ \rangle,\ \langle\ 1,2,2\ \rangle,\ \langle\ 2,2,2\ \rangle\ \rangle$
5. ***for*** $l \leftarrow 1$ ***to*** 8 ***do***
6. $\langle\ i,\ j,\ k\ \rangle \leftarrow subcube[\ l\ ],\ \langle\ R_{ijk}, F_{ijk},\ D_{ijk}\ \rangle \leftarrow$ COMPUTE-BOUNDARY-3D($X_i,\ Y_j,\ Z_k,\ L'_{ijk},\ B'_{ijk},\ T'_{ijk}$)
7. Compose R from $R_{2,j,k}$, F from $F_{i,2,k}$, and D from $D_{i,j,2}$, respectively, where $i, j, k \in [1, 2]$
8. ***return*** $\langle\ R,\ F,\ D\ \rangle$

COMPUTE-BOUNDARY-3D ENDS

FUNCTION 2.2. COMPUTE-TRACEBACK-PATH-3D($X,\ Y,\ Z,\ L,\ B,\ T,\ P$)

Input. Here $r = |X| = |Y| = |Z| = 2^t$ for some nonnegative integer $t \le q$, and $Q[0:r, 0:r, 0:r] \equiv c[u-1:u+r-1, v-1:v+r-1, w-1:w+r-1]$, $X = x_u x_{u+1} \ldots x_{u+r-1}$, $Y = y_v y_{v+1} \ldots y_{v+r-1}$ and $Z = z_w z_{w+1} \ldots z_{w+r-1}$ for some u, v and w ($1 \le u, v, w \le n - r + 1$). The left, back and top boundaries of $Q[1:r, 1:r, 1:r]$ are stored in L ($\equiv Q[0, 0:r, 0:r]$), B ($\equiv Q[0:r, 0, 0:r]$) and T ($\equiv Q[0:r, 0:r, 0]$), respectively. The 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 2.2
3. ***else*** { *For $i, j, k \in [1, 2]$, the left, right, front, back, top and bottom planes of subcube Q_{ijk} are denoted by L_{ijk}, R_{ijk}, F_{ijk}, B_{ijk}, T_{ijk} and D_{ijk}, respectively. X_1 and X_2 denote the 1st and the 2nd half of X, respectively (similarly for Y and Z).*}
4. Extract $L_{1,j,k}$ from L, $B_{i,1,k}$ from B, and $T_{i,j,1}$ from T, respectively, where $i, j, k \in [1, 2]$ { *$L_{2,j,k} \equiv R_{1,j,k}$, $B_{i,2,k} \equiv F_{i,1,k}$ and $T_{i,j,2} \equiv D_{i,j,1}$ for $i, j, k \in [1, 2]$* }
5. $subcube[\ 1:8\ ] \leftarrow \langle\ \langle\ 1,1,1\ \rangle,\ \langle\ 2,1,1\ \rangle,\ \langle\ 1,2,1\ \rangle,\ \langle\ 2,2,1\ \rangle,\ \langle\ 1,1,2\ \rangle,\ \langle\ 2,1,2\ \rangle,\ \langle\ 1,2,2\ \rangle,\ \langle\ 2,2,2\ \rangle\ \rangle$

   Forward Pass (Compute Boundaries):
6. ***for*** $l \leftarrow 1$ ***to*** 7 ***do***
7. $\langle\ i,\ j,\ k\ \rangle \leftarrow subcube[\ l\ ],\ \langle\ R_{ijk}, F_{ijk},\ D_{ijk}\ \rangle \leftarrow$ COMPUTE-BOUNDARY-3D($X_i,\ Y_j,\ Z_k,\ L'_{ijk},\ B'_{ijk},\ T'_{ijk}$) { *if $L_{ijk} \equiv Q[i', j_1:j_2, k_1:k_2]$ then $L'_{ijk} \equiv Q[i', j_1-1:j_2, k_1-1:k_2]$; similarly for B_{ijk} and T_{ijk}.* }

   Backward Pass (Compute Traceback Path):
8. ***for*** $l \leftarrow 8$ ***downto*** 1 ***do***
9. $\langle\ i,\ j,\ k\ \rangle \leftarrow subcube[\ l\ ],\ P \leftarrow$ COMPUTE-TRACEBACK-PATH-3D($X_i,\ Y_j,\ Z_k,\ L'_{ijk},\ B'_{ijk},\ T'_{ijk},\ P$)
10. ***return*** P

COMPUTE-TRACEBACK-PATH-3D ENDS

Figure 1: Cache-oblivious algorithm for evaluating recurrence 2.2 along with the traceback path. For convenience of exposition we assume that we only need to compute $c[1:n, 1:n, 1:n]$ where $n = 2^q$ for some nonnegative integer q. The initial call to COMPUTE-TRACEBACK-PATH-3D is made with $X = x_1 x_2 \ldots x_n$, $Y = y_1 y_2 \ldots y_n$, $Z = z_1 z_2 \ldots z_n$, $L \equiv c[0, 0:n, 0:n]$, $B \equiv c[0:n, 0, 0:n]$, $T \equiv c[0:n, 0:n, 0]$ and $P = \langle (n, n, n)\rangle$.

Analysis. Let $I_1(n)$ be the cache-complexity of COMPUTE-BOUNDARY-3D on input sequences of length n each. Then

$$I_1(n) = \begin{cases} \mathcal{O}\left(1 + \frac{n^2}{B}\right) & \text{if } n \leq \sqrt{\alpha M}, \\ 8I_1\left(\frac{n}{2}\right) + \mathcal{O}\left(1 + \frac{n^2}{B}\right) & \text{otherwise;} \end{cases}$$

where α is the largest constant sufficiently small that computation involving three input sequences of length $\sqrt{\alpha M}$ each can be performed completely inside the cache. Solving the recurrence we obtain $I_1(n) = \mathcal{O}\left((n^3/M) + (n^3/(B\sqrt{M}))\right)$. It is straight-forward to show that the algorithm runs in $\mathcal{O}\left(n^3\right)$ time and uses $\mathcal{O}\left(n^2\right)$ space. In contrast, though the standard iterative dynamic programming approach for computing the output boundary has the same time and space complexities (see, e.g., [5] for a standard technique that allows the DP to be implemented in $\mathcal{O}\left(n^2\right)$ space), it incurs a factor of $\sqrt{M}$ more cache-misses.

Computing a Traceback Path in 3D (COMPUTE-TRACEBACK-PATH-3D in Figure 1). Given the input boundary of $c[1:n, 1:n, 1:n]$ and the entry point of the traceback path (which is typically $c[n, n, n]$) the function COMPUTE-TRACEBACK-PATH-3D 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 c is a $1 \times 1 \times 1$ matrix, the traceback path can be updated directly using recurrence 2.2, otherwise it performs two passes: forward and backward. In the forward pass it computes the output boundaries of all subcubes (except $Q_{2,2,2}$) as in COMPUTE-BOUNDARY-3D. After this pass the algorithm knows the input boundaries of all eight subcubes, and the problem reduces to recursively extracting the fragments of the traceback path from each subcube and combining them. In the backward pass the algorithm does precisely that. It starts at $Q_{2,2,2}$ and updates the traceback path by calling itself recursively on all subcubes 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 subcube the algorithm requires an entry point on its output boundary through which the path enters the subcube and initially this point is known for only one subcube. When the subcubes are processed in the backward order the exit point of the traceback path from one subcube acts as the entry point of the path to the next subcube in the sequence.

Analysis. Let $I_2(n)$ be the cache-complexity of COMPUTE-TRACEBACK-PATH-3D on input sequences of length n each. We observe that though the algorithm calls itself recursively 8 times in the backward pass, at most 4 of those recursive calls will actually be executed and the rest will terminate at line 1 of the algorithm (see Figure 1)) since the traceback path cannot intersect more than 4 subcubes. Then,

$$I_2(n) = \begin{cases} \mathcal{O}\left(1 + \frac{n^2}{B}\right) & \text{if } n \leq \sqrt{\gamma M}, \\ 4I_2\left(\frac{n}{2}\right) + 7I_1\left(\frac{n}{2}\right) + \mathcal{O}\left(1 + \frac{n^2}{B}\right) & \text{otherwise;} \end{cases}$$

where γ is the largest constant sufficiently small that computation involving sequences of length $\sqrt{\gamma M}$ each can be performed completely inside the cache. Solving the recurrence we obtain $I_2(n) = \mathcal{O}\left((n^3/M) + (n^3/(B\sqrt{M}))\right)$. The time and space complexities of the algorithm can be shown to be $\mathcal{O}\left(n^3\right)$ and $\mathcal{O}\left(n^2\right)$, respectively. 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 $\sqrt{M}$, 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 three sequences X, Y and Z of length n each any recurrence relation of the same form as recurrence 2.2 can be solved and a traceback path can be computed 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.*

As mentioned before, our approach of finding the traceback path as described above is different from Hirschberg's approach [15], and as explained below it is also simpler. Hirschberg's algorithm splits the computation space into two equal halves, and applies traditional iterative DP in the forward direction on one half and in backward direction on the other. It then combines the results of both halves to find the point at which the traceback path intersects their common plane. This point of intersection allows one to discard portions of the computation space from both halves and recursively find the rest of the traceback path from the remaining portions of the two halves. However, it is not always obvious how to combine the results of the two halves to find the point of intersection and the process depends on the recurrence relation at hand. Finding this point can be particularly difficult when the recurrence relation has multiple fields (equivalently, if there are multiple recurrence relations). In contrast our algorithm always applies DP in the natural (i.e., forward) direction, and there is no tricky step of combining incompatible intermediate results. The complexity of the recurrence relation does not increase the complexity of coding our algorithms.

3 Parallel Implementation of the Cache-Oblivious Framework

In this section we consider parallel implementations of the cache-oblivious algorithms given in Figure 7 (in the Appendix) and Figure 1 (in Section 2) for evaluating recurrences 2.1 and 2.2 of Section 2, respectively.

3.1 Parallel Evaluation of the 2-Dimensional Recurrence 2.1. Let $T_{P_2}(n) = T_\infty(n)$ denote the parallel running time of function COMPUTE-TRACEBACK-PATH-2D when invoked with an unbounded number of processors on two sequences of length n each. Let $T_{B_2}(n)$ denote the same for COMPUTE-BOUNDARY-2D. (Both of these functions are in the Appendix.) We observe that in function COMPUTE-BOUNDARY-2D (lines 4–6) and in the forward pass of function COMPUTE-TRACEBACK-PATH-2D (lines 5–8) quadrants $Q_{1,2}$ and $Q_{2,1}$ can be evaluated in parallel while maintaining the correctness of the computation. We also observe that in the backward pass of COMPUTE-TRACEBACK-PATH-2D (lines 9–11) at most three recursive calls will be made since the traceback path can pass through at most three quadrants. These three recursive calls must be made sequentially. Before making the recursive calls in the backward pass COMPUTE-TRACEBACK-PATH-2D must save the input boundaries of all quadrants except $Q_{2,2}$. Thus we have the following recurrences:

$$T_{B_2}(n) \leq 3 \cdot T_{B_2}\left(\frac{n}{2}\right) + 4 \quad \text{and} \quad T_{P_2}(n) \leq 2 \cdot T_{B_2}\left(\frac{n}{2}\right) + 3 \cdot T_{P_2}\left(\frac{n}{2}\right) + 3n + 6$$

Now assuming for simplicity that n is a power of 2, and $T_{B_2}(1) = T_{P_2}(1) \leq 1$, we obtain the following by solving the recurrences above:

$$T_{B_2}(n) \leq 3 \cdot n^{\log_2 3} - 2 \quad \text{and} \quad T_{P_2}(n) \leq 2 \cdot (4 + \log_2 n) \cdot n^{\log_2 3} - 6n - 1$$

Therefore, $T_\infty(n) = T_{P_2}(n) \leq 2 \cdot (4 + \log_2 n) \cdot n^{\log_2 3} - 6n - 1$. Now assuming that $T_p(n)$ denotes the running time of COMPUTE-TRACEBACK-PATH-2D with p processors on a pair of sequences of length n each, we obtain the following theorem:

THEOREM 3.1. *When executed with p processors parallel implementation of* COMPUTE-TRACEBACK-PATH-2D *performs $T_1(n) = \mathcal{O}\left(n^2\right)$ work and terminates in $T_p(n) \leq \frac{T_1(n)}{p} + T_\infty(n) = \mathcal{O}\left(\frac{n^2}{p} + n^{\log_2 3} \log n\right)$ parallel steps on two sequences of length n each.*

Now let $p(n) = \frac{T_1(n)}{T_\infty(n)} \approx \frac{n^{\log_2 (4/3)}}{2 \cdot (4 + \log_2 n)}$. Then we have the following from Theorem 3.1.

$$T_p(n) = \begin{cases} \mathcal{O}\left(\frac{n^2}{p}\right) & \text{if } p \leq p(n), \\ \mathcal{O}\left(n^{\log_2 3} \log n\right) & \text{otherwise.} \end{cases} \tag{3.3}$$

In other words, the $\frac{n^2}{p}$ term in the parallel running time of COMPUTE-TRACEBACK-PATH-2D given in Theorem 3.1 dominates when $p \leq p(n)$, otherwise the $n^{\log_2 3} \log n$ term dominates and consequently p has little impact on the overall running time once $p > p(n)$.

3.2 Parallel Evaluation of the 3-Dimensional Recurrence 2.2. Let $T_{P_3}(n) = T_\infty(n)$ and $T_{B_3}(n)$ be the parallel running times of COMPUTE-TRACEBACK-PATH-3D and COMPUTE-BOUNDARY-3D, respectively, when invoked with an unbounded number of processors on three sequences of length n each. We observe that in function COMPUTE-BOUNDARY-3D (lines 5–6) and in the forward pass of function COMPUTE-TRACEBACK-PATH-3D (lines 6–7) quadrants $Q_{1,1,2}$, $Q_{1,2,1}$ and $Q_{2,1,1}$ can be evaluated in parallel followed by the parallel evaluation of quadrants $Q_{1,2,2}$, $Q_{2,1,2}$ and $Q_{2,2,1}$. We also observe that in the backward pass of COMPUTE-TRACEBACK-PATH-3D (lines 8–9) no more than four recursive calls will be made and all these calls must be made sequentially. In addition to making the recursive calls in the backward pass COMPUTE-TRACEBACK-PATH-3D saves the input boundaries of all quadrants except $Q_{2,2,2}$. Thus we have the following recurrences:

$$T_{B_3}(n) \leq 4 \cdot T_{B_3}\left(\frac{n}{2}\right) + 8 \quad \text{and} \quad T_{P_3}(n) \leq 3 \cdot T_{B_3}\left(\frac{n}{2}\right) + 4 \cdot T_{P_3}\left(\frac{n}{2}\right) + \frac{21}{4} \cdot n^2 + 11$$

Now assuming for simplicity that $n = 2^q$ for some integer $q \geq 0$, and $T_{B_3}(1) = T_{P_3}(1) \leq 1$, we obtain the following by solving the recurrences above:

$$T_{B_3}(n) \leq \frac{11}{3} \cdot n^2 - \frac{8}{3} \quad \text{and} \quad T_{P_3}(n) \leq 2 \cdot (1 + 4 \log_2 n) \cdot n^2 - 1$$

Therefore, $T_\infty(n) = T_{P_3}(n) \leq 2 \cdot (1 + 4 \log_2 n) \cdot n^2 - 1$, and we obtain the following theorem:

THEOREM 3.2. *When executed with p processors parallel implementation of* COMPUTE-TRACEBACK-PATH-3D *performs $T_1(n) = \mathcal{O}\left(n^3\right)$ work and terminates in $T_p(n) \leq \frac{T_1(n)}{p} + T_\infty(n) = \mathcal{O}\left(\frac{n^3}{p} + n^2 \log n\right)$ parallel steps on three sequences of length n each.*

Assuming $p(n) = \frac{T_1(n)}{T_\infty(n)} \approx \frac{n}{2 \cdot (1 + 4 \log_2 n)}$, we have the following from Theorem 3.2.

$$T_p(n) = \begin{cases} \mathcal{O}\left(\frac{n^3}{p}\right) & \text{if } p \leq p(n), \\ \mathcal{O}\left(n^2 \log n\right) & \text{otherwise.} \end{cases} \tag{3.4}$$

4 Applications of the Cache-Oblivious Framework

In this section we describe how to apply the cache-oblivious framework described in Section 2 in order to obtain cache-oblivious algorithms for pairwise sequence alignment, median and RNA secondary structure prediction with simple pseudoknots. As described in Section 3 once a problem is mapped to the framework it also immediately implies a parallel cache-oblivious algorithm for the problem.

4.1 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 the 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 [18]. The i-th letter x_i of X is said to be in a *gap* provided (x_i, y_j) does not appear in M for any letter y_j of Y. Similar definition of gap is given for the letters of Y. Given a *gap penalty* g and a mismatch cost $s(a, b)$ for each pair of letters $(a, b) \in \Sigma$, 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 [18].

For simplicity of exposition we will assume $m = n$ for the rest of this section. All algorithms discussed in this section can be easily extended to handle sequence of unequal lengths without changing their performance bounds.

The basic sequence alignment problem can be solved using a simple dynamic program in $\mathcal{O}\left(n^2\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(\frac{n^2}{B}\right)$ cache misses [18, 27, 14] and the space complexity can be reduced to $\mathcal{O}(n)$ using Hirschberg's space-reduction technique [15]. In [6] we presented a cache-oblivious algorithm for solving the edit-distance recurrence (an instance of recurrence 2.1) which immediately implies a cache-oblivious algorithm for the basic sequence alignment problem that runs in $\mathcal{O}\left(n^2\right)$ time, uses $\mathcal{O}(n)$ space and incurs only $\mathcal{O}\left(\frac{n^2}{BM}\right)$ cache misses.

The formulation of the basic sequence alignment problem favors a large number of small gaps in the alignment while real biological processes favor the opposite. The alignment can be made more realistic (i.e., more likely to produce a small number of long gaps) by using an *affine gap penalty* [13, 3] 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 $g_i + g_e \times k$ (sometimes $g_i + g_e \times (k-1)$ is also used as the total gap cost).

In [13] 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 $\mathcal{O}(n)$ using Hirschberg's space-reduction technique [19]. However, the time and cache complexities of the algorithm remain unchanged.

In this section we use the cache-oblivious framework described in Section 2 to obtain a cache-oblivious algorithm for the pairwise alignment problem with affine gap costs that runs in $\mathcal{O}\left(n^2\right)$ time and uses $\mathcal{O}(n)$ space, but incurs only $\mathcal{O}\left(\frac{n^2}{BM}\right)$ cache misses.

Gotoh's algorithm [13] solves the following dynamic programming recurrences.

$$D(i,j) = \begin{cases} G(0,j) + g_e & \text{if } i = 0 \text{ and } j > 0 \\ \min\{D(i-1,j), G(i-1,j) + g_i\} + g_e & \text{if } i > 0 \text{ and } j > 0. \end{cases} \tag{4.5}$$

$$I(i,j) = \begin{cases} G(i,0) + g_e & \text{if } i > 0 \text{ and } j = 0 \\ \min\{I(i,j-1), G(i,j-1) + g_i\} + g_e & \text{if } i > 0 \text{ and } j > 0. \end{cases} \tag{4.6}$$

$$G(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ g_i + g_e \times j & \text{if } i = 0 \text{ and } j > 0 \\ g_i + g_e \times i & \text{if } i > 0 \text{ and } j = 0 \\ \min\{D(i,j), I(i,j), G(i-1,j-1) + s(x_i, y_j)\} & \text{if } i > 0 \text{ and } j > 0. \end{cases} \tag{4.7}$$

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

Cache-Oblivious Implementation. Recurrences 4.5 - 4.7 can be viewed as a single recurrence evaluating a single matrix $c[0:n, 0:n]$ with three fields: D, I and G. When $i = 0$ or $j = 0$ each field of

$c[i,j]$ depends only on the indices i and j, and constants g_i and g_e, and hence each such entry can be computed using a function similar to h_2 in recurrence 2.1 of Section 2. When both i and j are positive, $c[i,j]$ depends on x_i, y_j, the entries in $c[i-1:i, j-1:j] \setminus c[i,j]$ (i.e., in $D(i-1:i, j-1:j) \setminus D(i,j)$, $I(i-1:i, j-1:j) \setminus I(i,j)$ and $G(i-1:i, j-1:j) \setminus G(i,j)$,), and constants g_i and g_e. Hence, in this case $c[i,j]$ can be computed using a function similar to function f_2 in recurrence 2.1. Thus recurrences 4.5 - 4.7 completely match recurrence 2.1 in Section 2. Therefore, function COMPUTE-BOUNDARY-2D (see Figure 7 in the Appendix) can be used to compute the optimal alignment cost cache-obliviously, and function COMPUTE-TRACEBACK-PATH-2D can be used to extract the optimal alignment. Thus the following claim follows from Theorem A.1 in the Appendix.

CLAIM 4.1. *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 new cache-oblivious algorithm is a factor of M improvement over previous implementations [13, 19].

4.2 Median. The Median problem is the problem of obtaining an optimal alignment of three sequences using an affine gap penalty. The median sequence under the optimal alignment is also computed. Under the affine gap penalty the cost function is $w(l) = g_i + g_e \times (l-1)$, where l is the gap length and g_i and g_e are the given gap introduction and gap extension costs. Knudsen [16] presented a dynamic program to find multiple alignment of N sequences, each of length n in $\mathcal{O}\left(16.81^N n^N\right)$ time and $\mathcal{O}\left(7.442^N n^N\right)$ space. It is also mentioned in [16] that the Hirschberg memory reduction technique can be used to improve the space complexity of the algorithm by a factor of n. For the median problem, this gives an $\mathcal{O}\left(n^3\right)$ time, $\mathcal{O}\left(n^2\right)$ space algorithm that incurs $\mathcal{O}\left(n^3/B\right)$ cache-misses. An Ukkonen-based algorithm is presented in [22], which performs well especially for sequences whose (3-way) edit distance d is small. On average, it requires $\mathcal{O}\left(n + d^3\right)$ time and space to compute the alignment [22].

Knudsen's Algorithm [16] is a dynamic program over a three-dimensional matrix K. Finding an optimal alignment of three sequences (say, $X = x_1 x_2 \ldots x_n$, $Y = y_1 y_2 \ldots y_n$ and $Z = z_1 z_2 \ldots, z_n$) involves filling a table of three rows, whose entry could be either a sequence character or a gap. Each column describes how characters of three sequences are matched.

Each entry $K(i,j,k)$ is composed of 23 fields. Each field corresponds to an indel configuration d, which describes how the last characters x_i, y_j and z_k are matched. A residue configuration defines how the next three characters 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, $1 \le i \le 3$ indicates if the aligned character of sequence i is a gap or a residue, while e_4 corresponds to the aligned character of the median sequence. There are 10 residue configurations out of 16 possible ones. The recursive step calculates the value of the next entry by applying residue configurations to each indel configuration. We define $next(e,d) = d'$ if applying the residue configuration e to the indel configuration d gives the indel configuration d'. The recurrence relation of Knudsen's algorithm is:

$$K(i,j,k)_d = \begin{cases} 0 & \text{if } i=j=k=0 \text{ and } d = d_o \\ \infty & \text{if } i=j=k=0 \text{ and } d \neq d_o \\ \min_{e,d' \text{ s.t. } d=next(e,d')} \left\{ \begin{array}{c} K(i',j',k')_{d'} + G_{e,d} \\ +M_{(i',j',k')\to(i,j,k)} \end{array} \right\} & \text{otherwise.} \end{cases} \tag{4.8}$$

where d_o is the configuration where all characters are matched, $i' = i - e_1$, $j' = j - e_2$ and $k' = k - e_3$, $M_{(i',j',k')=(i,j,k)}$ is the matching cost between characters of the sequences, and $G_{e,d}$ is cost for introducing or extending the gap.

Note that both M and G do not depend on the value of $K(i,j,k)_d$ but depend on e, d and d'. Hence, the values of M and G can be pre-processed before the execution of the program. Therefore, Knudsen's algorithm runs in $\mathcal{O}\left(n^3\right)$ time and space with $\mathcal{O}\left(n^3/B\right)$ cache-misses.

Cache-Oblivious Algorithm. In order to make recurrence 4.8 match the general recurrence 2.2 given in Section 2, we shift all characters of X, Y and Z one position to the right, introduce a dummy character in front of each of those three sequences, and obtain the following recurrence by modifying recurrence 4.8.

$$c[i,j,k]_d = \begin{cases} \infty & \text{if } i=0 \text{ or } j=0 \text{ or } k=0 \\ 0 & \text{if } i=j=k=1 \text{ and } d=d_o \\ \infty & \text{if } i=j=k=1 \text{ and } d \neq d_o \\ \min_{e,d' \text{ s.t. } d=next(e,d')} \left\{ \begin{array}{c} c[i',j',k']_{d'} + G_{e,d} \\ +M_{(i',j',k')\rightarrow(i,j,k)} \end{array} \right\} & \text{otherwise.} \end{cases}$$

It is easy to see that $K(i,j,k)_d = c[i+1,j+1,k+1]_d$ for $0 \leq i,j,k \leq n$ and any d. If $i=0$ or $j=0$ or $k=0$ then $c[i,j,k]_d$ can be evaluated using a function $h_3(\ \langle\ i,\ j,\ k\ \rangle\ ) = \infty$ as in the general recurrence 2.2. Otherwise the value of $c[i,j,k]_d$ 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, $c[i,j,k]_d$ can be evaluated using a function similar to f_3 in recurrence 2.2. Therefore, the above recurrence matches the general 3-dimensional recurrence 2.2, and function Compute-Boundary-3D (see Figure 1) can be used to compute the matrix c and function Compute-Traceback-Path-3D to retrieve an optimal alignment. Hence we claim the following using Theorem 2.1.

Claim 4.2. *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.*

4.3 RNA Secondary Structure Prediction with Pseudoknots. In this section we reduce the space-complexity of Akutsu's DP algorithm for RNA secondary structure prediction with simple pseudoknots [2] from $\mathcal{O}\left(n^3\right)$ to $\mathcal{O}\left(n^2\right)$, and its cache-complexity from $\mathcal{O}\left(n^4/B\right)$ to $\mathcal{O}\left(n^4/(B\sqrt{M})\right)$.

A single-stranded RNA can be viewed as a string $X = x_1x_2\ldots x_n$ over the alphabet $\{A,U,G,C\}$ of bases. An RNA strand tends to give rise to interesting two or three dimensional structures by forming *complementary base pairs* (i.e., $\{A,U\}$ and $\{C,G\}$, and sometimes $\{G,U\}$) with itself. The shapes of these RNA macromolecules determine the properties of a cell and hence understanding these structures is essential to understanding the processes of life.

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 $i<j<k<l$ [27, 23, 2]. An *RNA secondary structure with pseudoknots* is a structure where this nesting condition is violated [23, 2] and such structures play important roles in several known RNAs [24].

Given a single-stranded RNA molecule, the *basic RNA secondary structure prediction problem* (w/o pseudoknots) asks for a secondary structure with the maximum number of base pairs [18]. This problem can be solved using dynamic programming in $\mathcal{O}\left(n^3\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(\frac{n^3}{B}\right)$ cache-misses [18, 27]. The cache-complexity of the problem can be improved to $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ [4, 6, 7] while keeping the time and space complexities unchanged.

RNA secondary structure prediction with pseudoknots is a harder problem. In [2] Akutsu presented a dynamic programming algorithm for RNA secondary structure prediction with *simple pseudoknots* (see [2] for definition) which requires $\mathcal{O}\left(n^4\right)$ time and $\mathcal{O}\left(n^3\right)$ space, and incurs $\mathcal{O}\left(\frac{n^4}{B}\right)$ cache-misses.

Though other algorithms with the same time, space and cache complexities exist [26], Akutsu's algorithm is simpler and can be easily extended to include various scoring functions. For more complex pseudoknots (e.g., *recursive pseudoknots* [2]) known algorithms have higher complexities [2, 26, 23].

We describe below the DP recurrences used in Akutsu's algorithm [2] to compute an RNA secondary structure with the maximum number of base pairs in the presence of simple pseudoknots.

For every pair (i_0, k_0) with $1 \leq i_0 \leq k_0 - 2 \leq n - 2$, recurrences 4.9 - 4.13 are used to compute the maximum number of base pairs in a pseudoknot whose endpoints are the i_0-th and k_0-th residues. The value computed by recurrence 4.13, i.e., $S_{pseudo}(i_0, k_0)$, is the desired value. In recurrences 4.9 and 4.10, $v(x, y) = 1$ if (x, y) form a base pair, otherwise $v(x, y) = -\infty$. We list Akutsu's recurrences [2] below.

$$S_L(i,j,k) = \begin{cases} v(a_i, a_j) & \text{if } i_0 \leq i < j \geq k, \\ v(a_i, a_j) + S_{MAX}(i-1, j+1, k) & \text{if } i_0 \leq i < j < k, \\ 0 & \text{otherwise.} \end{cases} \tag{4.9}$$

$$S_R(i,j,k) = \begin{cases} v(a_j, a_k) & \text{if } i_0 - 1 = i < j - 1 = k - 2, \\ v(a_j, a_k) + S_{MAX}(i, j+1, k-1) & \text{if } i_0 \leq i < j < k, \\ 0 & \text{otherwise.} \end{cases} \tag{4.10}$$

$$S_M(i,j,k) = \begin{cases} \max \left\{ \begin{array}{c} S_L(i-1,j,k),\ S_M(i-1,j,k), \\ S_{MAX}(i,j+1,k), \\ S_M(i,j,k-1),\ S_R(i,j,k-1) \end{array} \right\} & \text{if } i_0 \leq i < j < k, \\ 0 & \text{otherwise.} \end{cases} \tag{4.11}$$

$$S_{MAX}(i,j,k) = \max\{\ S_L(i,j,k),\ S_M(i,j,k),\ S_R(i,j,k)\ \} \tag{4.12}$$

$$S_{pseudo}(i_0, k_0) = \max_{i_0 \leq i < j < k \leq k_0} \{\ S_{MAX}(i,j,k)\ \} \tag{4.13}$$

Observe that since k_0 does not appear in recurrences 4.9 - 4.12, after we have computed all entries of S_{MAX} for a fixed i_0, we can compute all $S_{pseudo}(i_0, k_0)$ values for $k_0 \geq i_0 + 2$ using 4.13 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, computing $S_{pseudo}(i_0, k_0)$ for all valid i_0 and k_0 will require $\mathcal{O}\left(n^4\right)$ time and $\mathcal{O}\left(n^3\right)$ space and incur $\mathcal{O}\left(\frac{n^4}{B}\right)$ cache-misses.

After computing all required $S_{pseudo}(i_0, k_0)$ values, Akutsu uses the following recurrence [2] to compute the optimal score $S(1, n)$ for the entire structure in $\mathcal{O}\left(n^3\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(\frac{n^3}{B}\right)$ cache-misses.

$$S(i,j) = \max\left\{\ S_{pseudo}(i,j),\ S(i+1, j-1) + v(a_i, a_j),\ \max_{i<k\leq j}\{S(i,k-1), S(k,j)\}\ \right\} \tag{4.14}$$

The cache-complexity of evaluating recurrence 4.14 can be improved to $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ while keeping the time and space complexities unchanged using our cache-oblivious GEP framework [6, 7] (see also [4]).

Since the cost of evaluating recurrences 4.9 - 4.12 for all $n - 2$ possible values for i_0 dominates the cost of evaluating recurrence 4.14, the overall time, space and cache-complexities of the algorithm are $\mathcal{O}\left(n^4\right)$, $\mathcal{O}\left(n^3\right)$ and $\mathcal{O}\left(\frac{n^4}{B}\right)$, respectively.

Space Reduction. We now describe our space reduction result. Observe that in order to evaluate recurrence 4.13 we need to retain all $\mathcal{O}\left(n^3\right)$ values computed by recurrence 4.12. We can avoid using this extra space if we can compute all required $S_{pseudo}(i_0, k_0)$ values on the fly while evaluating recurrence

4.12. In order to achieve this goal we introduce one more recurrence (4.15), replace recurrence 4.13 for S_{pseudo} with recurrence 4.16 for S'_{pseudo}, and use S'_{pseudo} instead of S_{pseudo} for evaluating recurrence 4.14.

$$S_P(i,j,k) = \begin{cases} -\infty & \text{if } i < i_0 \text{ or } j \geq k, \\ \max\{\ S_{MAX}(i,j,k),\ S_P(i,j+1,k)\ \} & \text{if } i_0 \leq i < j < k, \\ S_P(i,j+1,k) & \text{otherwise.} \end{cases} \tag{4.15}$$

$$S'_{pseudo}(i_0,k_0) = \begin{cases} -\infty & \text{if } k_0 < i_0 + 2, \\ \max\left\{S'_{pseudo}(i_0,k_0-1),\ \max_{i_0 \leq i < k_0-1}\{S_P(i,i_0+1,k_0)\}\right\} & \text{otherwise.} \end{cases} \tag{4.16}$$

We claim that recurrence 4.16 computes exactly the same values as recurrence 4.13.

Claim 4.3. *For $1 \leq i_0 \leq k_0 - 2 \leq n-2$, $S'_{pseudo}(i_0,k_0) = S_{pseudo}(i_0,k_0)$.*

Proof. (sketch) We obtain the following by simplifying recurrence 4.15.

$$S_P(i,j,k) = \begin{cases} \max_{\max\{i+1,j\} \leq j' < k}\{\ S_{MAX}(i,j',k)\ \} & \text{if } i_0 \leq i \text{ and } j < k, \\ -\infty & \text{otherwise.} \end{cases}$$

Therefore, $\max_{i_0 \leq i < k_0-1}\{S_P(i,i_0+1,k_0)\} = \max_{i_0 \leq i < j < k_0}\{\ S_{MAX}(i,j,k_0)\ \}$. We can now evaluate $S'_{pseudo}(i_0,k_0)$ by induction on k_0. For $k_0 \geq i_0 + 2$,

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

■

Now observe that in order to evaluate recurrence 4.16 we only need the values $S_P(i,j,k)$ for $j = i_0+1$, and each entry (i,j,k) in recurrences 4.9 - 4.12 and 4.15 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 $S'_{pseudo}(i_0,k_0)$ 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.

The time and cache complexities of the algorithm remain unchanged from [2].

Cache-Oblivious Algorithm. The evaluation of recurrences 4.9 - 4.12 and 4.15 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 2.2. Therefore, for any fixed i_0 we can use the Compute-Boundary-3D function in Figure 1 to compute all entries $S_P(i,i_0+1,k)$ and consequently all relevant $S'_{pseudo}(i_0,k_0)$ values. All $S'_{pseudo}(i_0,k_0)$ values can be computed by n applications (i.e., once for each i_0) of Compute-Boundary-3D.

For any given pair (i_0,k_0) the pseudoknot with the optimal score can be traced back cache-obliviously by calling Compute-Traceback-Path-3D. Thus from Theorem 2.1 we obtain the following claim.

Claim 4.4. *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.*

As mentioned before, all $S'_{pseudo}(i_0, k_0)$ values can be computed by n applications of COMPUTE-BOUNDARY-3D, and we observe that all these n function calls can be made in parallel. We know from Section 3.2 that COMPUTE-BOUNDARY-3D executes $T_{B_3(n)} = \mathcal{O}\left(n^2\right)$ parallel steps when called with an unbounded number of processors on an RNA sequence of length n. Since COMPUTE-BOUNDARY-3D performs $\mathcal{O}\left(n^3\right)$ work, n parallel applications of this function will perform $\mathcal{O}\left(n^4\right)$ work and execute $\mathcal{O}\left(\frac{n^4}{p} + n^2\right)$ parallel steps when executed with p processors. After computing all $S'_{pseudo}(i_0, k_0)$ values a pseudoknot with the optimal score is determined using recurrence 4.14 which can be solved in $\mathcal{O}\left(n^3\right)$ work and $\mathcal{O}\left(\frac{n^3}{p} + n \log^2 n\right)$ parallel steps using our cache-oblivious parallel GEP framework [8]. Finally the optimal pseudoknot thus determined can be traced back in a single call of COMPUTE-TRACEBACK-PATH-3D which according to Theorem 3.2 performs $\mathcal{O}\left(n^3\right)$ work and executes $\mathcal{O}\left(\frac{n^3}{p} + n^2 \log n\right)$ parallel steps when called with p processors. Therefore, we can claim the following.

CLAIM 4.5. *Given an RNA sequence of length n and p processors, a secondary structure that has the maximum number of base pairs in the presence of simple pseudoknots can be computed in $\mathcal{O}\left(n^4\right)$ work and $\mathcal{O}\left(\frac{n^4}{p} + n^2 \log n\right)$ parallel steps.*

Extensions. In [2] 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 [2], 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.

5 Experimental Results

Model	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 (4 dual-core)	2.2 GHz	64 KB (2-way, $B = 64$ B)	1 MB (8-way, $B = 64$ B)	32 GB

Table 1: Machines used for experiments.

We ran our experiments on the machines listed in Table 1. All machines were running Ubuntu Linux 5.10. All our algorithms were implemented in C++ using a uniform programming style, and implementations of some algorithms we collected for comparing against our algorithms were written in C. We compiled all C++ code using *g++* 3.3.4 and all C code using *gcc* 3.3.4. Optimization parameter -O3 was used in both cases. Each machine was exclusively used for experiments (i.e., no other programs were running on them).

We describe our experimental results below.

5.1 Pairwise Global Sequence Alignment with Affine Gap Penalty.

We performed experimental evaluation of both the sequential and the parallel implementations of our cache-oblivious algorithm.

Sequential Performance. We performed experimental evaluation of the algorithms in Table 2 on Intel P4 Xeon and AMD Opteron 250. On both machines only one processor was used.

Algorithm	Comments	Time	Space	Cache Misses
PA-CO	our cache-oblivious algorithm (see Section 4.1)	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n^2}{BM}\right)$
PA-LS	our implementation of linear-space variant of Gotoh's algorithm [19]	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n^2}{B}\right)$
PA-FASTA	linear-space implementation of Gotoh's algorithm [19] available in *fasta2* package [20]	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n^2}{B}\right)$

Table 2: Pairwise sequence alignment algorithms used in our experiments.

Both PA-LS and PA-FASTA used 32 bit integer costs while PA-CO was implemented to use both 32 bit integer costs and single-precision floats with SSE ("Streaming SIMD Extension") instructions. In order to reduce the overhead of recursion in PA-CO, instead of stopping the recursion at $r = 1$ in COMPUTE-BOUNDARY-2D and COMPUTE-TRACEBACK-PATH-2D, we stopped at $r = 256$, and solved the subproblem using the traditional iterative method.

In most cases PA-FASTA ran about 20%-30% slower than the integer version of PA-CO on AMD Opteron and up to 10% slower on Intel Xeon. We summarize our results below.

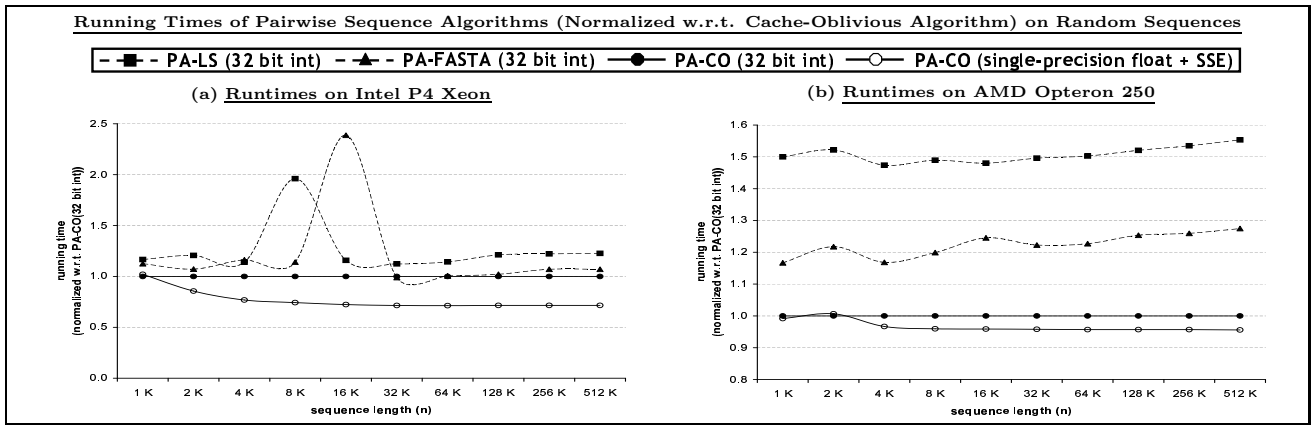

Figure 2: Comparison of running times of two implementations (one with 32 bit integer costs and the other one is SSE-enabled with single-precision float costs) of our cache-oblivious pairwise alignment algorithm (PA-CO) with the linear-space implementation of Gotoh's algorithm available in FASTA [20] (PA-FASTA), and our linear-space implementation of Gotoh's algorithm (PA-LS). All running times are normalized w.r.t. that of the 32 bit integer implementation of PA-CO. Figure (a) plots running times on Intel P4 Xeon and Figure (b) on AMD Opteron 250. Each data point is the average of 3 independent runs on randomly generated strings over $\{ A, T, G, C \}$.

Random Sequences. We ran all three implementations on randomly generated equal-length string-pairs over $\{ A, C, G, T \}$ both on Intel P4 Xeon (see Figure 2(a)) and AMD Opteron 250 (see Figure 2(b)). We varied string lengths from 1 K to 512 K. Both PA-FASTA and PA-LS always ran slower than both versions of PA-CO on AMD Opteron (PA-FASTA ran around 27% slower and PA-LS about 55% slower than the 32 bit integer version of PA-CO for sequences of length 512 K) and generally the relative performance of PA-CO improved over both PA-FASTA and PA-LS as the length of the sequences increased. The SSE-enabled single-precision float version of PA-CO ran slightly ($\approx 5\%$) faster than its integer version. The trends were almost similar on Intel Xeon except that the improvements of the integer version of PA-CO over PA-FASTA and PA-LS were more modest, and the SSE-enabled float version of PA-CO ran around 35% faster than its integer version. 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-LS and PA-FASTA).

Sequence pairs with lengths (10^6)	Running times of pairwise alignment algorithms on *CFTR DNA Sequences* [25] (on AMD Opteron with 32 bit integer costs)		
	PA-FASTA (t_1)	PA-CO (t_2)	ratio ($\frac{t_1}{t_2}$)
human/baboon (1.80/1.51)	$20h\ 34m$	$17h\ 23m$	1.18
human/chimp (1.80/1.32)	$19h\ 51m$	$15h\ 25m$	1.29
baboon/chimp (1.51/1.32)	$16h\ 43m$	$12h\ 43m$	1.31
human/rat (1.80/1.50)	$24h\ \ 1m$	$18h\ 16m$	1.31
rat/mouse (1.50/1.49)	$16h\ 49m$	$13h\ 55m$	1.21

Table 3: Comparison of running times (on AMD Opteron 250) of our cache-oblivious pairwise alignment algorithm (PA-CO in col 3) with the linear-space implementation of Gotoh's algorithm available in FASTA [20] (PA-FASTA in col 2) on CFTR DNA sequences [25]. Column 4 gives the ratio of the running time of PA-FASTA to that of PA-CO. Both algorithms use 32 bit integer costs. Each number in columns 2 and 3 is the time for a single run.

Real-World Sequences. We ran PA-CO (with integer costs) and PA-FASTA on CFTR DNA sequences of lengths between 1.3 million to 1.8 million [25] on AMD Opteron, and PA-FASTA ran 20%-30% slower than PA-CO on these sequences (see Table 3). 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.

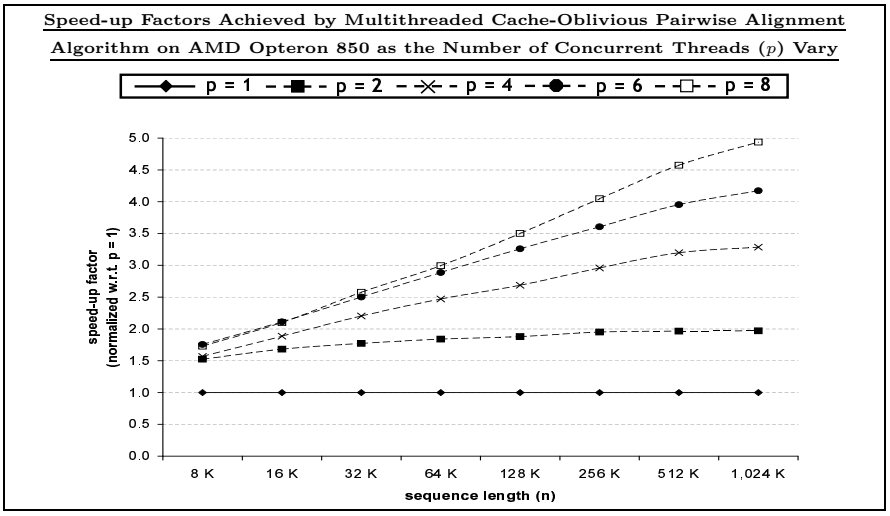

Figure 3: Speed-up factors (w.r.t. unthreaded code) achieved by multithreaded cache-oblivious pairwise alignment algorithm on 8-processor Opteron 850 as number of threads (p) vary. Sequences were randomly generated strings over $\{ A, T, G, C \}$.

Parallel Performance. In Figure 3 we plot the speed-up factors achieved by the parallel (multithreaded) implementation of PA-CO (with 32 bit integer costs) w.r.t. its sequential (unthreaded) implementation on an 8-processor AMD opteron 850. The number of concurrent threads (i.e., the number of available processors p) was varied from 1 to 8, and the length of the sequences was varied from 8 K to 1024 K. The experimental results show that PA-CO achieves reasonable speed-up as the number of processors increases, and for a fixed number of processors the speed-up factor improves as the length of the sequence grows. For example, with 8 processors the algorithm achieves a speed-up factor of 1.7 when $n = 8$ K, but achieves a speed-up factor of about 5 when $n = 1024$ K. These results follow theoretical predictions since as equation 3.3 in Section 3 predicts the running time of PA-CO improves

significantly as long as $p \leq p(n) \approx \frac{n^{\log_2(4/3)}}{2\cdot(4+\log_2 n)}$, and as the following table shows the value of $p(n)$ does not grow beyond 8 until n exceeds 1024 K.

n	1 K	2 K	4 K	8 K	16 K	32 K	64 K	128 K	256 K	512 K	1024 K	2048 K	4096 K
$p(n)$	0.63	0.79	0.99	1.24	1.56	1.97	2.49	3.17	4.03	5.14	6.57	8.41	10.78

Table 4: $p(n)$ values (see equation 3.3 in Section 3) for parallel COMPUTE-TRACEBACK-PATH-2D.

5.2 Median. We evaluated sequential and parallel performance of our cache-oblivious median algorithm.

Sequential Performance. We performed experimental evaluations of the algorithms in Table 5 (all using 32 bit integer costs) on Intel P4 Xeon, AMD Opteron 250 and SUN Blade. Only a single processor was used on each machine. We used $g_i = 3$, $g_e = 1$ and a mismatch cost of 1 in all experiments.

Algorithm	Comments	Time	Space	Cache Misses
MED-CO	our cache-oblivious algorithm (see Section 4.2)	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$
MED-Knudsen	Knudsen's implementation of his algorithm [17]	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}\left(\frac{n^3}{B}\right)$
MED-H	our implementation of MED-Knudsen using Hirschberg's space-reduction technique	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}\left(\frac{n^3}{B}\right)$
MED-ukk.alloc	Powell's implementation [21] of an $\mathcal{O}(d^3)$-space Ukkonen-based algorithm (d = 3-way edit distance of sequences)	$\mathcal{O}(n + d^3)$ (avg.)	$\mathcal{O}(n + d^3)$	$\mathcal{O}\left(\frac{d^3}{B}\right)$
MED-ukk.checkp	Powell's implementation [21] of an $\mathcal{O}(d^2)$-space Ukkonen-based algorithm (d = 3-way edit distance of sequences)	$\mathcal{O}(n \log d + d^3)$ (avg.)	$\mathcal{O}(n + d^2)$	$\mathcal{O}\left(\frac{d^3}{B}\right)$

Table 5: Median algorithms used in our experiments.

In order to reduce the overhead of recursion in MED-CO, instead of stopping the recursion when $r = 1$ COMPUTE-BOUNDARY-3D and COMPUTE-TRACEBACK-PATH-3D, we stopped the recursion at $r = 64$ on both Intel Xeon and AMD Opteron, and solved the problem by calling a sub-routine similar to Knudsen's algorithm at that point.

Overall MED-Knudsen ran about 1.5-2.5 times slower than MED-CO on both Intel Xeon and AMD Opteron. However, on all machines MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp could not be run for sequences longer than 640 due to their high space overhead. 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 Intel Xeon (see Figure 4(a)) and AMD Opteron (see Figure 4(b)). Due to lack of memory space MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp crashed on both machines for sequences longer than 384, 256 and 640, respectively.

On Intel Xeon, MED-CO was the fastest. It ran at least 1.45 times faster than MED-Knudsen and at least 1.25 times faster than MED-H. 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 upto 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 1.4, 2.5, 3.4 and 4.2 times faster than MED-H, MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp, respectively.

Real-World Sequences. We ran all algorithms (except MED-H) on triplets of 16S bacterial rDNA sequences from the *Pseudanabaena group* [10] (see Table 6). All experiments were run on Intel Xeon.

Triplets 1–5 in Table 6 were formed by choosing three sequences of length less than 500 from the group at random, while triplet 6 was formed manually for reasons to be explained in the next paragraph. On triplets 1–5, MED-Knudsen ran around 35–50% slower and MED-ukk.checkp upto 3.2 times slower

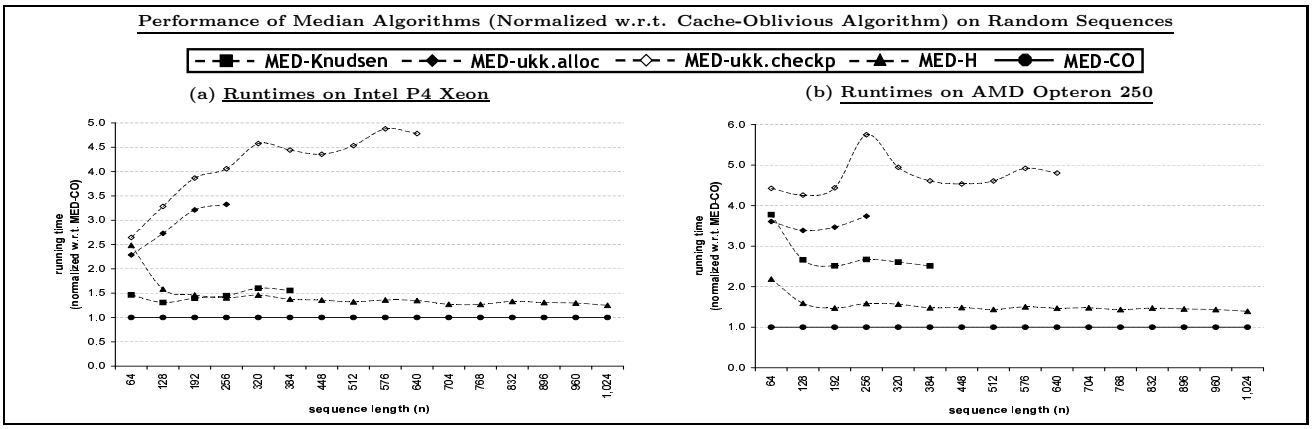

Figure 4: Comparison of performance of our cache-oblivious median algorithm (MED-CO) with Knudsen's implementation of his algorithm [16] available in [17] (MED-Knudsen), our Hirschberg-style implementation of Knudsen's algorithm (MED-H), and Powell's implementation of two algorithms [22] available in [21] (MED-ukk.alloc and MED-ukk.checkp). Figures (a) and (b) plot running times of the algorithms on Intel P4 Xeon and AMD Opteron 250, respectively. figure for MED-CO. Due to their high space overhead MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp could not be run for sequences longer than 640 on any machine. Each data point is the average of 3 independent runs on random strings over $\{ A, T, G, C \}$.

	Running times (in sec) on Intel Xeon for random triples of *16S Bacterial rDNA Sequences from the Pseudanabaena Group* [10] (runtime w.r.t. MED-CO)					
Triplet	Lengths	Alignment Cost	MED-Knudsen	MED-ukk.alloc	MED-ukk.checkp	MED-CO
1	367 387 388	299	722 (1.48)	512 (1.05)	601 (1.23)	487 (1.00)
2	378 388 403	324	752 (1.42)	– (–)	769 (1.45)	529 (1.00)
3	342 367 389	339	611 (1.35)	– (–)	863 (1.91)	451 (1.00)
4	342 370 474	432	764 (1.44)	– (–)	1,701 (3.20)	531 (1.00)
5	370 388 447	336	– (–)	– (–)	824 (1.49)	553 (1.00)
6	367 388 389	260	695 (1.42)	330 (0.67)	380 (0.77)	491 (1.00)

Table 6: Comparison of running times (on Intel Xeon) of four algorithms on 16S bacterial rDNA sequences from the Pseudanabaena group [10]: our cache-oblivious median algorithm (PA-MED in col 7), Knudsen's implementation of his algorithm [17] (MED-Knudsen in col 4), and Powell's implementation of two Ukkonen-based algorithms [21] (MED-ukk.alloc and MED-ukk.checkp in cols 5 and 6, respectively). Triplets 1–5 were formed by choosing random sequences of length less than 500 from the group while triplet 6 was formed by choosing sequences manually in order to keep the alignment cost small. 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.

than MED-CO. Running time of MED-ukk.checkp w.r.t. MED-CO degraded as the alignment cost increased. MED-ukk.alloc which requires space cubic in the alignment cost could not be run on triplets with alignment cost larger than 299, that is, on triplets 2–5. On triplet 5 MED-Knudsen also crashed due to its high space requirement which is cubic in the sequence length.

The sequences in triplet 6 were chosen manually in order to keep their alignment cost small. We used this triplet in order to verify the theoretical prediction that MED-ukk.alloc and MED-ukk.checkp would run faster on triplets with small alignment costs since unlike MED-CO whose running time is cubic in the sequence length, running times of those two algorithms are cubic in the 3-way edit distance

of the input sequences (see Table 5). The alignment cost of triplet 6 is 260, and as Table 5 shows both MED-ukk.alloc and MED-ukk.checkp, indeed, ran faster than MED-CO on this triplet.

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) when the alignment cost is moderately large.

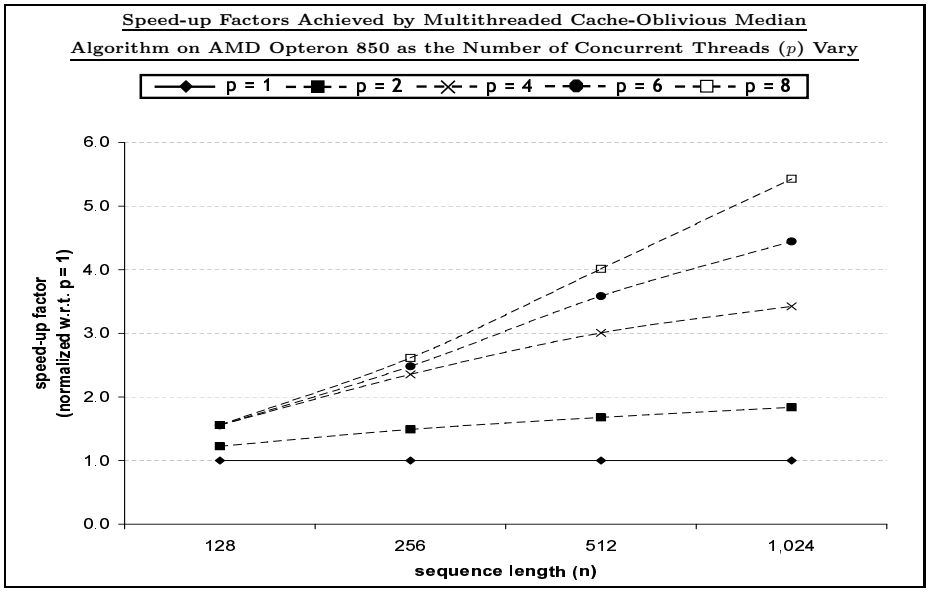

Figure 5: Speed-up factors (w.r.t. unthreaded code) achieved by multithreaded cache-oblivious median algorithm on 8-processor Opteron 850 as number of threads (p) vary. Sequences were randomly generated strings over $\{ A, T, G, C \}$.

Parallel Performance. Figure 5 plots the speed-up factors achieved by multithreaded (parallel) MED-CO (with 32 bit integer costs) w.r.t. its sequential implementation on an 8-processor AMD opteron 850. The number of concurrent threads (i.e., the number of available processors p) was varied from 1 to 8, and the sequences lengths from 2^7 (128) to 2^{10} (1024). Experimental results show that MED-CO speeds up better than the cache-oblivious pairwise alignment algorithm PA-CO in Section 5.1 as the number of processors grows. For example, with 8 processors MED-CO achieves a speed-up factor of around 5.5 when the sequence length is only 1024, while PA-CO requires sequences of length 1024 K in order to reach a speed-up factor of 5 (see Figure 3). As explained below this empirical observation follows theoretical prediction. We know from equation 3.4 in Section 3 that MED-CO speeds up well with the number of processor p as long as $p \leq p(n) \approx \frac{n}{2 \cdot (1+4 \log_2 n)}$, and the following table shows that $p(n)$ values for MED-CO grows a lot faster than $p(n)$ values for PA-CO (see Table 4 in Section 5.1).

n	32	64	128	256	512	1024
$p(n)$	0.76	1.28	2.21	3.88	6.92	12.49

Table 7: $p(n)$ values (equation 3.4 in Section 3) for parallel COMPUTE-TRACEBACK-PATH-3D.

5.3 RNA Secondary Structure Prediction with Pseudoknots. We implemented the algorithms in Table 8 (all are sequential and use 32 bit integer costs) 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.

Algorithm	Comments	Time	Space	Cache Misses
RNA-CO	our cache-oblivious algorithm (see Section 4.3)	$\mathcal{O}(n^4)$	$\mathcal{O}(n^2)$	$\mathcal{O}\left(\frac{n^4}{B\sqrt{M}}\right)$
RNA-CS	Akutsu's original cubic-space algorithm [2]	$\mathcal{O}(n^4)$	$\mathcal{O}(n^3)$	$\mathcal{O}\left(\frac{n^4}{B}\right)$
RBA-QS	Our iterative quadratic-space version of Akutsu's algorithm (see Section 4.3)	$\mathcal{O}(n^4)$	$\mathcal{O}(n^2)$	$\mathcal{O}\left(\frac{n^4}{B}\right)$

Table 8: Algorithms for RNA secondary structure prediction used in our experiments.

In order to reduce the overhead of recursion in RNA-CO, instead of executing line 1 of the algorithm for $r = 1$ (see Figure 1), we stopped as soon as we reached $r \leq 64$ and solved the problem directly using our iterative quadratic-space variant RNA-QS.

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.

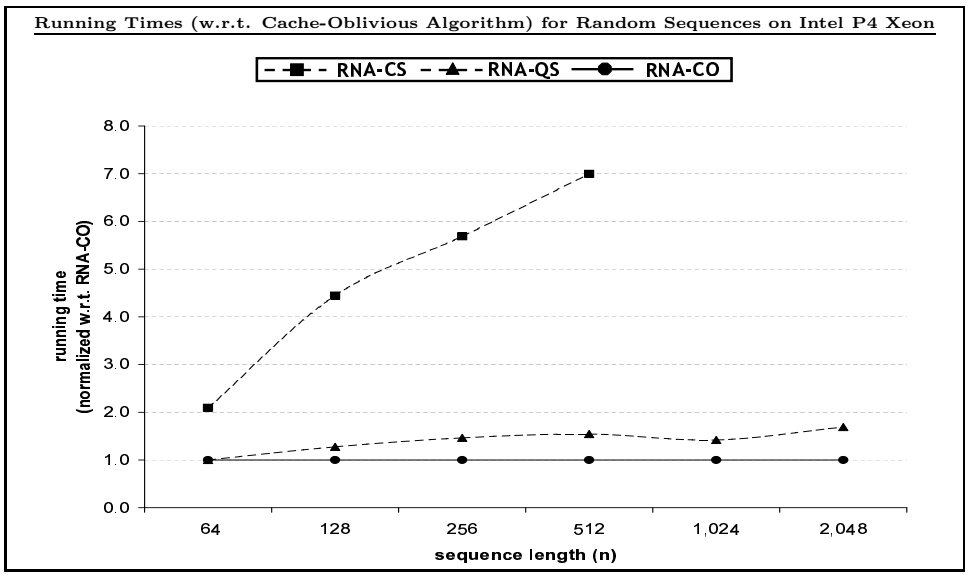

Figure 6: Comparison of running times of three algorithms for RNA secondary structure prediction with simple pseudoknots on Intel P4 Xeon: our cache-oblivious algorithm (RNA-CO), Akutsu's algorithm [2] (RNA-CS) and our quadratic-space version of Akutsu's algorithm (RNA-QS). All running times are normalized w.r.t. RNA-CO. Due to its high space overhead RNA-CS could not be run for sequences longer than 512. Each data point is the average of 3 independent runs on random strings over $\{ A, U, G, C \}$.

Random Sequences. We ran all three algorithms on randomly generated string-pairs over $\{ A, U, G, C \}$ (see Figure 6). 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 [9]. 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 5.3).

		Runtimes of algorithms for RNA secondary structure prediction with simple pseudoknots on Intel Xeon for *Bacterial (Spirochaetes) 16S rRNA Sequences* [9]		
Organism	Length (n)	Quadratic Space (RNA-QS)	Cache-Oblivious (RNA-CO)	$\frac{\text{RNA-QS}}{\text{RNA-CO}}$
Brevinema andersonii	1443	$2h\ 14m$	$1h\ 22m$	1.64
Borrelia burgdorferi	1530	$2h\ 48m$	$1h\ 44m$	1.62
Borrelia burgdorferi	1537	$2h\ 48m$	$1h\ 45m$	1.60
Borrelia hermsii	1523	$2h\ 43m$	$1h\ 41m$	1.61
Brachyspira hyodysenteriae	1463	$2h\ 21m$	$1h\ 27m$	1.61
Cristispira CP1	1491	$2h\ 30m$	$1h\ 33m$	1.62
Leptonema illini	1526	$2h\ 45m$	$1h\ 42m$	1.61
Leptospira interrogans	1508	$2h\ 37m$	$1h\ 38m$	1.61
Spirochaeta aurantia	1520	$2h\ 42m$	$1h\ 41m$	1.60
Treponema pallidum (rRNA A)	1549	$2h\ 54m$	$1h\ 48m$	1.60
Average	1509	$2h\ 38m$	$1h\ 38m$	1.61

Table 9: Comparison of running times of two algorithms for RNA secondary structure prediction with simple pseudoknots on Intel P4 Xeon: our cache-oblivious algorithm (RNA-CO), and our quadratic-space version of Akutsu's algorithm (RNA-QS). Inputs were Bacterial (Spirochaetes) 16S rRNA sequences [9] with an average length of 1509. Akutsu's cubic space algorithm could not be run because these sequences are too long for it. Each number in columns 2 and 3 represents time for a single run.

5.4 Discussion. We implemented all algorithms in C++ using a uniform programming style. However, the results in Figure 2 suggest that our C++ implementation of the linear-space variant of Gotoh's algorithm (denoted PA-LS) runs slower than the C implementation of the same algorithm in FASTA (denoted PA-FASTA). This might have happened because either our code is not as optimized as the FASTA code or the optimization quality of the $g++$ and the *gcc* compilers are not the same. In either case there is room for improving the performance of our implementations even further. Therefore, we expect that our cache-oblivious algorithms can be implemented to run even faster.

The method used to implement the base case of the cache oblivious algorithms also matters. In order to reduce the overhead of recursion the base case is implemented using a nonrecursive algorithm (typically the traditional iterative dynamic programming algorithm). If the traceback path is not required the traditional iterative DP can be easily implemented to use a factor of n less space than the case when the traceback path is required (see [5]). That's why the base case of RNA-CO (an implementation of our RNA secondary structure prediction algorithm which does not compute a traceback path) was implemented using a quadratic-space algorithm while that of MED-CO (our median algorithm that computes a traceback path) was implemented using a cubic-space algorithm. Though the size of the base case is a constant and thus does not affect asymptotic running times, execution of the base case can flood the smaller cache levels if it uses a space-intensive algorithm and thus can increase the running time of the algorithm by degrading its cache performance. We believe this is one of the reasons why the performance MED-CO relative to its counterparts was not as impressive as that of RNA-CO. However, we also implemented the base case of MED-CO using a Hirschberg-style quadratic-

space implementation of the standard cubic-space DP, and our experimental results (not included in this paper) show that the space-reduced base case does not improve the running time of MED-CO. We believe this happens because for small problem sizes such as the base case the overhead of the Hirschberg-based implementation dominates its running time and consequently it fails to improve over the running time of the simple iterative DP.

We believe that in addition to being faster and more cache-efficient our cache-oblivious algorithms are simpler to implement than Hirschberg's space-reduction technique. As explained in Section 2.1, complicated recurrence relations and recurrence relations with multiple fields increase the difficulties of implementing Hirschberg's algorithm, but they do not complicate the implementation of our algorithms.

One of the major advantages of our cache-oblivious algorithms is that they are parallelizable with very little extra effort. In contrast, iterative algorithms such as MED-Knudsen and MED-ukk.alloc (see Section 5.2) are not parallelizable without substantial modifications. Though algorithms obtained using Hirschberg-style space-reduction techniques (e.g., PA-FASTA and PA-LS in Section 5.1, and MED-H and MED-ukk.checkp in Section 5.2) can be parallelized, the computation-space is divided only into two subproblems and substantial amount of sequential computation is performed before the next level of recursive subdivision. The sequential computation along with the fact that the decomposition is often unbalanced drastically reduces the amount of parallelism. Moreover, as described in the previous paragraph, it is not always easy to apply Hirschberg's technique.

6 Conclusion

In this paper we presented a general cache-oblivious framework for a class of widely encountered dynamic programming problems, 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 show that our algorithms are faster, both theoretically and experimentally, than the previous algorithms for these problems. Our algorithms can be parallelized with very little effort and they show good parallel performance in practice. Our framework can also 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.

Some interesting open research problems on this topic still exist especially designing cache-efficient algorithms for solving other classes of DP that also occur frequently in practice.

Acknowledgement. We thank Mike Brudno for providing us with 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. *Communications of the ACM*, 31:1116–1127, 1988.

[2] T. Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Applied Mathematics*, 104:45-62, 2000.

[3] S. Altschul and B. Erickson. Optimal Sequence Alignment Using Affine Gap Costs. *Bulletin of Mathematical Biology*, 48:603–616, 1986.

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

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

[6] R. Chowdhury and V. Ramachandran. Cache-Oblivious Dynamic Programming. In *Proc. of the 17th ACM-SIAM Symposium on Discrete Algorithms*, pp. 591–600, 2006.

[7] R. Chowdhury and V. Ramachandran. The Cache-Oblivious Gaussian Elimination Paradigm: Theoretical Framework and Experimental Evaluation. Technical Report TR-06-04, Department of Computer Sciences, University of Texas at Austin, March 2006. url: http://www.cs.utexas.edu/ftp/pub/techreports/tr06-04.pdf.

[8] R. Chowdhury and V. Ramachandran. The Cache-Oblivious Gaussian Elimination Paradigm: Theoretical Framework, Parallelization and Experimental Evaluation. Manuscript.

[9] 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 Bioinformatics*, 3:2. [Correction: *BioMed Central Bioinformatics*. 3:15.], 2002. url:http://www.rna.icmb.utexas.edu/

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

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

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

[13] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

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

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

[16] B. Knudsen. Optimal Multiple Parsimony Alignment with Affine Gap Cost Using a Phylogenetic Tree. In *Proc. of Workshop on Algorithms in Bioinformatics*, pp. 433–446, 2003.

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

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

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

[20] W. Pearson and D. Lipman. Improved tools for biological sequence comparison. In *Proc. of the National Academy of Sciences of the USA*, 85:2444–2448, 1988.

[21] D. Powell. *Software package* `align3str_checkp.tar.gz`.

[22] D. Powell, L. Allison and T. Dix. Fast, optimal alignment of three sequences using linear gap cost. *Journal of Theoretical Biology*, 207(3):325–336, 2000.

[23] E. Rivas and S. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *Journal of Molecular Biology*, 285(5):2053-68, 1999.

[24] E. ten Dam, K. Pleij, and D. Draper. Structural and functional aspects of RNA pseudoknots. *Biochemistry*, 31:11665-11676, 1992.

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

[26] Y. Uemura, A. Hasegawa, S. Kobayashi, and T. Yokomori. Grammatically modeling and predicting RNA secondary structures. In *Proc. of Genome Informatics Workshop VI*, Tokyo, 67–76, 1995.

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

Appendix

A Cache-Oblivious Algorithm for Evaluating the 2D Recurrence 2.1 in Section 2

Function A.1. Compute-Boundary-2D($X,\ Y,\ L,\ T$)

Input. See the input description of Compute-Traceback-Path-2D.

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. ***for*** $j \leftarrow 1$ ***to*** 2 ***do***
5. ***for*** $i \leftarrow 1$ ***to*** 2 ***do***
6. $\langle\ R_{ij},\ D_{ij}\ \rangle \leftarrow$ Compute-Boundary-2D($X_i,\ Y_j,\ L'_{ij},\ T'_{ij}$)
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-2D Ends

Function A.2. Compute-Traceback-Path-2D($X,\ Y,\ L,\ T,\ P$)

Input. Here $r = |X| = |Y| = 2^t$ for some nonnegative integer $t \le p$, 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 stored in L ($\equiv Q[0, 0:r]$) and T ($\equiv Q[0:r, 0]$), respectively. The 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 2.1
3. ***else*** { *For $i, j \in [1, 2]$, the left, right, top and bottom boundaries of quadrant Q_{ij} are denoted by L_{ijk}, R_{ijk}, T_{ijk} and D_{ijk}, 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, respectively, where $i, j \in [1, 2]$
{ $L_{2,j} \equiv R_{1,j}$ *and* $T_{i,2} \equiv D_{i,1}$ *for* $i, j \in [1, 2]$ }

Forward Pass (Compute Boundaries):

5. ***for*** $j \leftarrow 1$ ***to*** 2 ***do***
6. ***for*** $i \leftarrow 1$ ***to*** 2 ***do***
7. ***if*** $\langle i, j\rangle \neq \langle 2, 2\rangle$ ***then***
8. $\langle\ R_{ijk},\ D_{ijk}\ \rangle \leftarrow$ Compute-Boundary-2D($X_i,\ Y_j,\ L'_{ijk},\ T'_{ijk}$)
{ *if* $L_{ij} \equiv Q[i', j_1 : j_2]$ *then* $L'_{ij} \equiv Q[i', j_1 - 1 : j_2]$*; similarly for* B_{ij} *and* T_{ij}. }

Backward Pass (Compute Traceback Path):

9. ***for*** $j \leftarrow 2$ ***downto*** 1 ***do***
10. ***for*** $i \leftarrow 2$ ***downto*** 1 ***do***
11. $P \leftarrow$ Compute-Traceback-Path-2D($X_i,\ Y_j,\ L'_{ij},\ T'_{ijk},\ P$)
12. ***return*** P

Compute-Traceback-Path-2D Ends

Figure 7: Cache-oblivious algorithm for evaluating recurrence 2.1 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-2D 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 $P = \langle (n, n)\rangle$.

Analysis. The following theorem can be proved by analyzing the time, space and cache-complexities of the functions given in Figure 7. The analyses are simpler than those given in Section 2.1 for our algorithm for solving the general 3-dimensional recurrence 2.2, and hence are omitted.

THEOREM A.1. *Given two sequences X and Y of length n each any recurrence relation of the same form as recurrence 2.1 can be solved and a traceback path can be computed 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.*