Efficient Cache-oblivious String Algorithms for Bioinformatics *

Rezaul Alam Chowdhury[1], Hai-Son Le[2], and Vijaya Ramachandran[1]

[1] Department of Computer Sciences, University of Texas, Austin, TX 78712, USA,
{shaikat,vlr}@cs.utexas.edu

[2] Google Inc., Mountain View, CA 94043, USA,
haison3000@mail.utexas.edu

Abstract. We present theoretical and experimental results on cache-efficient and parallel algorithms for some well-studied string problems in bioinformatics: *global pairwise sequence alignment* and *median* (both with affine gap costs), and *RNA secondary structure prediction with simple pseudoknots*. For each problem we present cache-oblivious algorithms that match the best-known time complexity, match or improve the best-known space complexity, improve significantly over the cache-efficiency of earlier algorithms, and have efficient parallel implementations.
We present experimental results that show that these cache-oblivious algorithms run significantly faster than currently available software.
Our methods are applicable to several other problems 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.

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 [10]. 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 *ideal-cache model* [7] is an extension of the two-level I/O model that assumes an optimal offline cache replacement policy, and requires that algorithms remain oblivious of cache parameters M and B. A well-designed cache-oblivious algorithm is flexible and portable, and simultaneously adapts to all levels of

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

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 that are amenable to solution by dynamic programs with 'local dependencies' (see 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 also show that our framework can be parallelized with little effort, and analyze its parallel performance (in Section 3). We use this framework to develop cache-oblivious algorithms for three well-known string problems in bioinformatics: *global pairwise sequence alignment* and *median* (both with affine gap costs), and *RNA secondary structure prediction with simple pseudoknots* (Section 4). We present extensive experimental results showing that our algorithms are faster than current software for these problems (Section 5).

The results in this paper extend and generalize our earlier work in [5], where we presented a relatively simple cache-oblivious algorithm for finding the longest common subsequence (LCS) of two sequences. In [5] we also presented more involved cache-oblivious algorithms for other problems including pairwise sequence alignment with general gap costs and RNA secondary structure without pseudoknots.

We note that often in practice, biologically significant solutions are sought that may be sub-optimal under the precise optimization measure used. 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 even when such solutions are sought that are not necessarily optimal under our definitions.

2 CO Framework for a DP Class with Local Dependencies

Given $d \geq 1$ sequences $S_i = s_{i,1}s_{i,2}\ldots s_{i,n}$, $1 \leq i \leq d$, and functions $h(\cdot)$ and $f(\cdot,\cdot,\cdot)$, we consider dynamic programs that compute entries of a d-dimensional matrix $c[0:n, 0:n, \ldots, 0:n]$ as follows, where $\mathbf{i} = i_1, i_2, \ldots, i_d$ and $\mathbf{S_i}$ is the tuple $\langle\ s_{1,i_1},\ s_{2,i_2}, \ldots,\ s_{d,i_d}\ \rangle$ containing the i_j-th symbol of S_j in j-th position.

$$c[\ \mathbf{i}\ ] = \begin{cases} h\left(\ \langle\ \mathbf{i}\ \rangle\ \right) & \text{if } \exists\ i_j = 0, \\ f\left(\ \langle\ \mathbf{i}\ \rangle,\ \mathbf{S_i},\ c[i_1 - 1 : i_1, i_2 - 1 : i_2, \ldots, i_d - 1 : i_d] \setminus c[\ \mathbf{i}\ ]\ \right) & \text{otherwise.} \end{cases} \quad (2.1)$$

Function f can be arbitrary except that it is allowed to use exactly one cell from its third argument to compute the final value of $c[i_1, i_2, \ldots, i_d]$ (though it can consider all cells), and we call that specific cell the *parent cell* of $c[i_1, i_2, \ldots, i_d]$.

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

Recurrence 2.1 can be evaluated iteratively in $\mathcal{O}\left(n^d\right)$ time, $\mathcal{O}\left(n^d\right)$ space and $\mathcal{O}\left(n^d/B\right)$ cache-misses. Though space can be reduced to $\mathcal{O}\left(n^{d-1}\right)$ using

Hirschberg's technique [10], the cache-complexity remains unchanged if the traceback path must also be computed. If a traceback path is not required it is easy to reduce space requirement to $\mathcal{O}\left(n^{d-1}\right)$ even without using Hirschberg's technique, and the cache-complexity of the algorithm can be improved to $\mathcal{O}\left(n^d/(BM^{\frac{1}{d-1}})\right)$ using the cache-oblivious stencil-computation technique [8].

In Section 2.1 we present a cache-oblivious algorithm for solving the 3-dimensional version (i.e., $d = 3$) of recurrence 2.1 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. It improves over the previous best cache-miss bound by at least a factor of $\sqrt{M}$, and reduces space requirement by a factor of n when compared with the traditional iterative solution. In Sections 4.2 and 4.3 we use this algorithm to solve median of three sequences and RNA secondary structure prediction with simple pseudoknots.

In the technical report [4] we present a simpler cache-oblivious algorithm that solves the 2-dimensional version (i.e., $d = 2$) of recurrence 2.1. In Section 4.1 we use this algorithm for global pairwise sequence alignment with affine gap costs.

2.1 Cache-oblivious Algorithm for Solving Recurrence 2.1 in 3D.

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 recursively 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, when the output boundaries of all subcubes are computed the problem of finding the traceback path through the entire cube is reduced to the problem of recursively finding the fragments of the path through the subcubes. 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 [10] can also be used to solve recurrence 2.1 along with a traceback path. Unlike our algorithm, however, Hirschberg's approach 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. In contrast, our algorithm always applies DP in one direction and thus is simpler to implement.

We describe below the two parts of our algorithm. The pseudocode for both parts can be found in Figure 1 of the technical report [4].

Compute-Boundary-3D. Given the input boundary of $c[i_1 : i_2, j_1 : j_2, k_1 : k_2]$ this function recursively computes its output boundary. For simplicity of exposition we assume that $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 = 2^q - 1$ for some integer $q \geq 0$.

If $q = 0$, the function can compute the output boundary directly using recurrence 2.1, otherwise it decomposes its cubic computation space Q (initially

$Q \equiv c[1:n, 1:n, 1:n]$) into 8 subcubes $Q_{i,j,k}$, $1 \le i,j,k \le 2$, where $Q_{i,j,k}$ denotes the subcube that is i-th from the left, j-th from the back and k-th from the top. It then computes the output boundary of each subcube recursively as the input boundary of the subcube becomes available during the process of computation. After all recursive calls terminate, the output boundary of Q is composed from the output boundaries of the subcubes.

Analysis. Let $I_1(n)$ be the cache-complexity of COMPUTE-BOUNDARY-3D on sequences of length n. Then $I_1(n) = \mathcal{O}\left(1 + n^2/B\right)$ if the computation can be performed entirely inside the cache. Otherwise, $I_1(n) = 8I_1\left(n/2\right) + \mathcal{O}\left(1 + n^2/B\right)$. Solving the recurrence, $I_1(n) = \mathcal{O}\left(n^3/M + n^3/(B\sqrt{M})\right)$. It is straight-forward to show that the algorithm runs in $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(n^2\right)$ space.

COMPUTE-TRACEBACK-PATH-3D. Given the input boundary of $c[i_1 : i_2, j_1 : j_2, k_1 : k_2]$ and the entry point of the traceback path on the output boundary this function recursively computes the entire path.

If $q = 0$, the traceback path can be updated directly using recurrence 2.1, 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 starts at $Q_{2,2,2}$ and updates the traceback path by calling itself recursively on the subcubes in the reverse order of the forward pass.

Analysis. Let $I_2(n)$ be the cache-complexity of COMPUTE-TRACEBACK-PATH-3D on input sequences of length n each. Then $I_2(n) = \mathcal{O}\left(1 + n^2/B\right)$ if the computation can be performed completely inside the cache. Otherwise, $I_2(n) = 4I_2\left(n/2\right) + 7I_1\left(n/2\right) + \mathcal{O}\left(1 + n^2/B\right)$ since the traceback path cannot intersect more than 4 subcubes and hence at most 4 recursive calls will be made in the backward pass. Solving the recurrence we obtain $I_2(n) = \mathcal{O}\left(n^3/M + n^3/(B\sqrt{M})\right)$. The algorithm runs in $\mathcal{O}\left(n^3\right)$ time and $\mathcal{O}\left(n^2\right)$ space as with Hirschberg's scheme.

3 Parallel Implementation of the CO Framework

The framework in Section 2 has a simple parallel implementation that for general d performs $\mathcal{O}\left(n^d\right)$ work, uses $\mathcal{O}\left(n^{d-1}\right)$ space, incurs $\mathcal{O}\left(n^d/\left(BM^{\frac{1}{d-1}}\right)\right)$ cache-misses and terminates in $\mathcal{O}\left(n^d/p + n^{\log_2(d+1)} \log n\right)$ parallel steps when run on p processors with private caches (see technical report [4] for details). This is the parallel algorithm we implemented for our experiments in Section 5.

In this section we improve the parallel time complexity of our framework to $\mathcal{O}\left(n^d/p + n\right)$ while keeping the other bounds unchanged from above. We present two different parallel implementations for the 3-dimensional case for distributed and shared caches, respectively. Implementation for general d is similar.

3.1 Distributed Caches

We consider a parallel machine with p processors with each processor having a private cache of size M and block size B.

Par-Compute-Boundary-3D. This function decomposes its cubic computation space Q ($\equiv c[1:n, 1:n, 1:n]$) into $p^{\frac{3}{2}}$ subcubes of size $(n/\sqrt{p}) \times (n/\sqrt{p}) \times (n/\sqrt{p})$ each. By $Q_{i,j,k}$ $(1 \le i, j, k \le \sqrt{p})$ we denote the subcube that is i-th from the left boundary of Q, j-th from the back boundary and k-th from the top boundary. Then the computation progresses in $3\min(\sqrt{p}, n) - 2$ steps. In step t $(1 \le t \le 3\min(\sqrt{p}, n) - 2)$ output boundaries of all $Q_{i,j,k}$ with $i + j + k = t + 2$ are computed in parallel using a modified version of Compute-Boundary-3D which for each cell on the output boundary also computes the location where the traceback path from that cells hits the input boundary.

For $p \le n^2$, there are $\Theta\left(\sqrt{p}\right)$ steps of parallel subcube computations requiring $\mathcal{O}\left((n/\sqrt{p})^3\right)$ time each, and thus the entire computation terminates in $\mathcal{O}\left(\sqrt{p} \times (n/\sqrt{p})^3\right) = \mathcal{O}\left(n^3/p\right)$ parallel time. For $p > n^2$, there are $\Theta\left(n\right)$ steps of $\mathcal{O}\left(1\right)$ time each and the computation completes in $\mathcal{O}\left(n\right)$ parallel time. The parallel time complexity of the algorithm is thus $\mathcal{O}\left(n^3/p + n\right)$. It is straight-forward to show that the algorithm performs $\mathcal{O}\left(n^3\right)$ work and uses $\mathcal{O}\left(n^2\right)$ space.

Since there are $p^{3/2}$ calls to Compute-Boundary-3D on sequences of length $n/\sqrt{p}$ and each of them is executed on a single processor, total number of cache-misses incurred by all such calls is $\mathcal{O}\left(p^{3/2} \times (n/\sqrt{p})^3/(B\sqrt{M})\right) = \mathcal{O}\left(n^3/(B\sqrt{M})\right)$.

Par-Compute-Traceback-Path-3D. This function is similar to the sequential Compute-Traceback-Path-3D given in Section 2.1 except for the following differences.

<u>Forward Pass</u>: Instead of calling Compute-Boundary-3D described in Section 2.1 we call Par-Compute-Boundary-3D described in this section.

<u>Backward Pass</u>: The fragment of the traceback path inside $Q_{2,2,2}$ is extracted by calling Par-Compute-Traceback-Path-3D recursively. Now using the extra information on traceback paths computed in the forward pass we can find in constant time where the traceback path hits all other subcubes. Therefore, we can execute the remaining (at most three) recursive calls to Par-Compute-Traceback-Path-3D in parallel.

Let $T_p(n)$ denote the parallel running time of Par-Compute-Traceback-Path-3D when called with p processors. Let $T'_p(n)$ denote the same for Par-Compute-Boundary-3D. Then for $p \le n^2$, $T_p(n) \le 7 \cdot T'_p(n/2) + T_p(n/2) + T_{p/3}(n/2) + \Theta\left(n^2/p\right)$. Solving we obtain, $T_p(n) = \mathcal{O}\left(n^3/p\right)$. Therefore, for all values of p, $T_p(n) = \mathcal{O}\left(n^3/p + n\right)$. The algorithm performs $\mathcal{O}\left(n^3\right)$ work and uses $\mathcal{O}\left(n^2\right)$ space, and incurs $\mathcal{O}\left(n^3/(B\sqrt{M})\right)$ cache-misses.

3.2 Shared Caches

We consider a parallel machine with p processors sharing a single cache of size M and block size B.

The algorithms are similar to the sequential algorithms given in Section 2.1 until we reach a subproblem involving sequences of length $\Theta\left(\sqrt{p}\right)$. At that point we compute all entries of the cubic computation space and store all of them in $\Theta\left(p^{3/2}\right)$ space. We compute the entries (i.e., $1 \times 1 \times 1$ subcubes) of the cube using the parallel subcube computation method described in Section 3.1 for PAR-COMPUTE-BOUNDARY-3D, and then extract the traceback path in $\Theta\left(\sqrt{p}\right)$ time by following the parent cells through the cube sequentially.

It is not difficult to prove that the algorithms as described above perform $\mathcal{O}\left(n^3\right)$ work, use $\mathcal{O}\left(n^2 + p^{3/2}\right)$ space and terminate in $\mathcal{O}\left(n^3/p + n\right)$ parallel steps. Assuming $M = \Omega\left(p^{3/2}\right)$, the cache complexity of the algorithms can be shown to be $\mathcal{O}\left(n^3/(B\sqrt{M})\right)$. We omit the proofs from here for lack of space.

4 Some Applications of the Cache-oblivious Framework

4.1 Pairwise Global Sequence Alignment with Affine Gap Penalty.

Given two strings $X = x_1x_2\ldots x_m$ and $Y = y_1y_2\ldots y_n$ over a finite alphabet Σ, an *alignment* of X and Y is a matching M of sets $\{1,2,\ldots,m\}$ and $\{1,2,\ldots,n\}$ such that if $(i,j),(i',j') \in M$ and $i < i'$ hold then $j < j'$ must also hold [12]. The i-th letter of X or Y is said to be in a *gap* if it does not appear in any pair in M. Given a *gap penalty* g and a mismatch cost $s(a,b)$ for each pair $a,b \in \Sigma$, the *basic (global) pairwise sequence alignment problem* asks for a matching M_{opt} for which $(m + n - |M_{opt}|) \times g + \sum_{(a,b)\in M_{opt}} s(a,b)$ is minimized [12].

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

The formulation of the basic sequence alignment problem favors a large number of small gaps while real biological processes favor the opposite. The alignment can be made more realistic by using an *affine gap penalty* [9, 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 $g_i + g_e \times k$.

In [9] 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 $\mathcal{O}\left(n^2/B\right)$ cache misses. Gotoh's algorithm solves the following DP recurrences.

$$D(i,j) = \begin{cases} G(0,j) + g_e & \text{if } i = 0 \text{ and } j > 0 \\ \min\{D(i-1,j), G(i-1,j) + g_i\} + g_e & \text{if } i > 0 \text{ and } j > 0. \end{cases} \tag{4.2}$$

$$I(i,j) = \begin{cases} G(i,0) + g_e & \text{if } i > 0 \text{ and } j = 0 \\ \min\{I(i,j-1), G(i,j-1) + g_i\} + g_e & \text{if } i > 0 \text{ and } j > 0. \end{cases} \tag{4.3}$$

$$G(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ g_i + g_e \times j & \text{if } i = 0 \text{ and } j > 0 \\ g_i + g_e \times i & \text{if } i > 0 \text{ and } j = 0 \\ \min\{D(i,j), I(i,j), G(i-1,j-1) + s(x_i,y_j)\} & \text{if } i > 0 \text{ and } j > 0. \end{cases} \tag{4.4}$$

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

Cache-oblivious Implementation. Recurrences 4.2 - 4.4 can be viewed as a single recurrence evaluating a single matrix $c[0:n, 0:n]$ with three fields: D, I and G. This recurrence matches recurrence 2.1 with $d = 2$ (see technical report [4] for explanation), and thus can be solved using our cache-oblivious framework in $\mathcal{O}\left(n^2\right)$ time, $\mathcal{O}\left(n\right)$ space, and only $\mathcal{O}\left(n^2/(BM)\right)$ cache misses.

4.2 Median of Three Sequences.

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. Knudsen [11] 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. For the median problem, this gives an $\mathcal{O}\left(n^3\right)$ time and space algorithm that incurs $\mathcal{O}\left(n^3/B\right)$ cache-misses. An Ukkonen-based algorithm is presented in [15], which performs well especially for sequences whose (3-way) edit distance δ is small. On average, it requires $\mathcal{O}\left(n+\delta^3\right)$ time and space [15].

Knudsen's Algorithm [11] for three sequences (say, $X = x_1x_2\ldots x_n$, $Y = y_1y_2\ldots y_n$ and $Z = z_1z_2\ldots, z_n$) is a dynamic program over a three-dimensional matrix K. Each entry $K(i, j, k)$ is composed of 23 fields. Each field corresponds to an indel configuration q, 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 $\nu(e, q) = q'$ if applying the residue configuration e to the indel configuration q gives the indel configuration q'. The recurrence relation used by Knudsen's algorithm is:

$$K(i,j,k)_q = \begin{cases} 0 & \text{if } i=j=k=0 \wedge q=q_o \\ \infty & \text{if } i=j=k=0 \wedge q \neq q_o \\ \min_{e,q':q=\nu(e,q')} \left\{ \begin{array}{c} K(i',j',k')_{q'} + G_{e,q} \\ +M_{(i',j',k')\to(i,j,k)} \end{array} \right\} & \text{otherwise.} \end{cases} \tag{4.5}$$

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

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

Cache-oblivious Algorithm. In the technical report [4] we show how to reduce recurrence 4.5 to an instance of the general recurrence 2.1 with $d = 3$ using a simple transformation. Therefore, function Compute-Boundary-3D (see Section 2.1) can be used to compute the matrix K and function Compute-Traceback-Path-3D to retrieve an optimal alignment and the median 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.

4.3 RNA Secondary Structure Prediction with Simple Pseudoknots.

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 structures by forming *complementary base pairs* with itself. An *RNA secondary structure* (w/o pseudoknots) is a planar graph with the nesting condition: if $\{x_i, x_j\}$ and $\{x_k, x_l\}$ form base pairs and $i < j$, $k < l$ and $i < k$ hold then either

$i < k < l < j$ or $i < j < k < l$ [19, 16, 2]. An *RNA secondary structure with pseudoknots* is a structure where this nesting condition is violated [16, 2].

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

We list below the DP recurrences used in Akutsu's algorithm [2]. For every pair (i_0, k_0) with $1 \leq i_0 \leq k_0 - 2 \leq n - 2$, recurrences 4.6 - 4.10 compute the maximum number of base pairs in a pseudoknot with endpoints at the i_0-th and k_0-th residues. The value computed by recurrence 4.10, i.e., $S_{pseudo}(i_0, k_0)$, is the desired value. In recurrences 4.6 and 4.7, $v(x, y) = 1$ if (x, y) form a base pair, otherwise $v(x, y) = -\infty$. All uninitialized entries are assumed to have value 0.

$$S_L(i,j,k) = \begin{cases} v(a_i, a_j) & \text{if } i_0 \leq i < j \geq k, \\ v(a_i, a_j) + S_{MAX}(i-1, j+1, k) & \text{if } i_0 \leq i < j < k. \end{cases} \tag{4.6}$$

$$S_R(i,j,k) = \begin{cases} v(a_j, a_k) & \text{if } i_0 - 1 = i < j - 1 = k - 2, \\ v(a_j, a_k) + S_{MAX}(i, j+1, k-1) & \text{if } i_0 \leq i < j < k. \end{cases} \tag{4.7}$$

$$S_M(i,j,k) = \max \left\{ \begin{array}{c} S_L(i-1,j,k),\ S_M(i-1,j,k), \\ S_{MAX}(i,j+1,k), \\ S_M(i,j,k-1),\ S_R(i,j,k-1) \end{array} \right\} \text{ if } i_0 \leq i < j < k. \tag{4.8}$$

$$S_{MAX}(i,j,k) = \max \{\ S_L(i,j,k),\ S_M(i,j,k),\ S_R(i,j,k)\ \} \tag{4.9}$$

$$S_{pseudo}(i_0, k_0) = \max_{i_0 \leq i < j < k \leq k_0} \{\ S_{MAX}(i,j,k)\ \} \tag{4.10}$$

After computing all entries of S_{MAX} for a fixed i_0, all $S_{pseudo}(i_0, k_0)$ values for $k_0 \geq i_0 + 2$ can be computed using 4.10 in $\mathcal{O}\left(n^3\right)$ time and space and $\mathcal{O}\left(n^3/B\right)$ cache-misses. Since there are $n - 2$ possible values for i_0, all $S_{pseudo}(i_0, k_0)$ can be computed in $\mathcal{O}\left(n^4\right)$ time, $\mathcal{O}\left(n^3\right)$ space and $\mathcal{O}\left(n^4/B\right)$ cache-misses.

Finally, the following recurrence computes 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 $\mathcal{O}\left(n^3/B\right)$ cache-misses [2].

$$S(i,j) = \max \left\{ \begin{array}{c} S_{pseudo}(i,j),\ S(i+1, j-1) + v(a_i, a_j), \\ \max_{i < k \leq j} \{S(i, k-1), S(k, j)\} \end{array} \right\} \tag{4.11}$$

Recurrence 4.11 can be evaluated in only $\mathcal{O}\left(n^3/(B\sqrt{M})\right)$ cache-misses and $\mathcal{O}\left(n^2\right)$ space without changing the other bounds using our GEP framework [5].

Space Reduction. We now describe our space reduction result. Observe that evaluating recurrence 4.10 requires retaining all $\mathcal{O}\left(n^3\right)$ values computed by recurrence 4.9. We avoid using this extra space by computing all required $S_{pseudo}(i_0, k_0)$ values on the fly while evaluating recurrence 4.9. We achieve this goal by introducing recurrence 4.12, replacing recurrence 4.10 with recurrence 4.13 for S'_{pseudo}, and using S'_{pseudo} instead of S_{pseudo} for evaluating recurrence 4.11. All uninitialized entries in recurrences 4.12 and 4.13 are assumed to have value $-\infty$.

$$S_P(i,j,k) = \begin{cases} \max \{\ S_{MAX}(i,j,k),\ S_P(i,j+1,k)\ \} & \text{if } i_0 \leq i < j < k, \\ S_P(i, j+1, k) & \text{if } i_0 \leq i \geq j < k. \end{cases} \tag{4.12}$$

$$S'_{pseudo}(i_0,k_0) = \max \left\{ \begin{array}{c} S'_{pseudo}(i_0,k_0-1), \\ \max_{i_0 \le i < k_0-1} \{S_P(i,i_0+1,k_0)\} \end{array} \right\} \text{ if } k_0 \ge i_0+2. \quad (4.13)$$

We prove in the technical report [4] that for $1 \le i_0 \le k_0 - 2 \le n - 2$, $S'_{pseudo}(i_0,k_0) = S_{pseudo}(i_0,k_0)$.

Now observe that in order to evaluate recurrence 4.13 we only need the values $S_P(i,j,k)$ for $j = i_0 + 1$, and each entry (i,j,k) in recurrences 4.6 - 4.9 and 4.12 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.

Cache-oblivious Algorithm. The evaluation of recurrences 4.6 - 4.9 and 4.12 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 2.1 for $d = 3$. Therefore, for any fixed i_0 we can use Compute-Boundary-3D from Section 2.1 to compute all entries $S_P(i, i_0+1, k)$ and consequently all relevant $S'_{pseudo}(i_0,k_0)$. All $S'_{pseudo}(i_0,k_0)$ values can be computed by n applications (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. Therefore, the required RNA secondary structure can be computed cache-obliviously in $\mathcal{O}\left(n^4\right)$ time, $\mathcal{O}\left(n^2\right)$ space and $\mathcal{O}\left(n^4/(B\sqrt{M})\right)$ cache misses. Using an explanation similar to that in Section 4.3 of the technical report [4] we can show that the parallel algorithm in Section 3 can solve the problem in $\mathcal{O}\left(n^4\right)$ work and $\mathcal{O}\left(n^4/p+n\right)$ parallel steps while keeping the other bounds unchanged.

Extensions. Our cache-oblivious framework also applies to several extensions of the basic DP for simple pseudoknots [2]. See our technical report [4] for details.

5 Experimental Results

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

Table 1. Machines for experiments. All block sizes: 64 bytes. OS: Ubuntu Linux 5.10.

We ran our experiments on the machines listed in Table 1. We used the *Cachegrind* profiler [17] for simulating cache effects. All our algorithms were implemented in C++ (compiled with *g++* 3.3.4) while some software packages we used for comparison were written in C (compiled with *gcc* 3.3.4). Optimization parameter -O3 was used in all cases. Each machine was exclusively used for experiments.

We now describe our experimental results on pairwise sequence alignment and the median problem. Results on RNA secondary structure prediction can be found in the technical report [4].

5.1 Pairwise Global Sequence Alignment with Affine Gap Penalty.

We performed experimental evaluation of the algorithms in Table 2.

Algorithm	Description	Time	Space	Cache Misses
PA-CO	our algorithm (see Section 4.1)	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(n\right)$	$\mathcal{O}\left(n^2/(BM)\right)$
PA-LS	linear-space Gotoh [13] (our code)	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(n\right)$	$\mathcal{O}\left(n^2/B\right)$
PA-FASTA	linear-space Gotoh [13] (from *fasta2* [14])	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(n\right)$	$\mathcal{O}\left(n^2/B\right)$

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

Sequential Performance. For random sequences both PA-FASTA and PA-LS always ran slower than PA-CO on AMD Opteron (see Figure 1), e.g., PA-FASTA ran around 27% slower and PA-LS about 55% slower than PA-CO for sequences of length 512 K. Relative performance of PA-CO improved over PA-FASTA and PA-LS as sequence length increased. The trends were similar on Intel Xeon except that improvements of PA-CO over PA-FASTA/PA-LS were more modest.

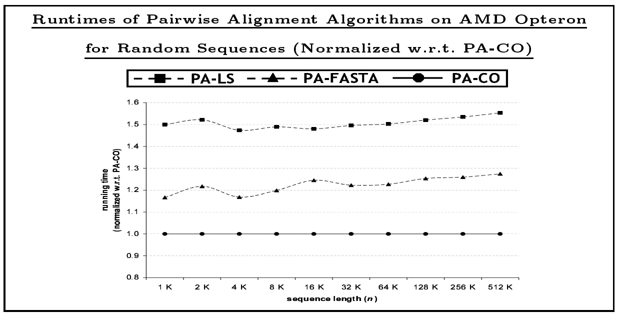

Fig. 1. Comparison of running times of pairwise sequence alignment algorithms (see Table 2) on AMD Opteron 250. All running times are normalized w.r.t. PA-CO. Each data point is the average of 3 independent runs on randomly generated strings over $\{\ A, T, G, C\ \}$.

For real-world CFTR DNA sequences [18] PA-FASTA ran around 20%-30% slower than PA-CO on AMD Opteron (see Table 3).

Runtimes of PA-FASTA and PA-CO on *CFTR DNA Sequences* [18] (on Opteron)			
Sequence pairs with lengths (10^6)	PA-FASTA (t_1)	PA-CO (t_2)	ratio (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 runtimes (on Opteron 250) of PA-CO with PA-FASTA (see Table 2) on CFTR DNA sequences [18]. Column 4 gives the ratio of runtime of PA-FASTA to that of PA-CO. Each number in cols 2 and 3 is the time for a single run.

Cache Performance. Though PA-FASTA causes fewer cache-misses than PA-CO when the input fits into the cache, it incurs significantly more misses than PA-CO as the input size grows beyond cache size (Figure 2). On AMD Opteron PA-FASTA incurs upto 300 times more L1 misses and 2500 times more L2 misses than PA-CO while on Intel Xeon the figures are 10 and 1000, respectively.

Parallel Performance. Our experimental results in Figure 3(a) 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 with sequence length. For example, with 8 processors PA-CO achieves a speed-up factor of 1.7 when $n = 8$ K, and about 5 when $n = 1024$ K.

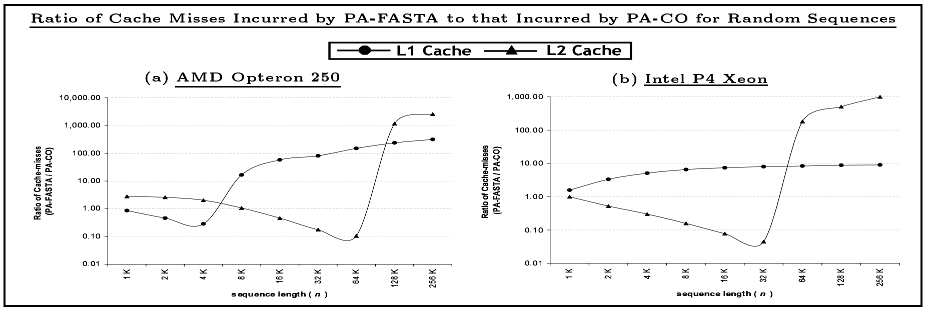

Fig. 2. Ratio of cache-misses incurred by PA-FASTA to that incurred by PA-CO (see Table 2) for both L1 and L2 caches. Data was obtained using Cachegrind [17].

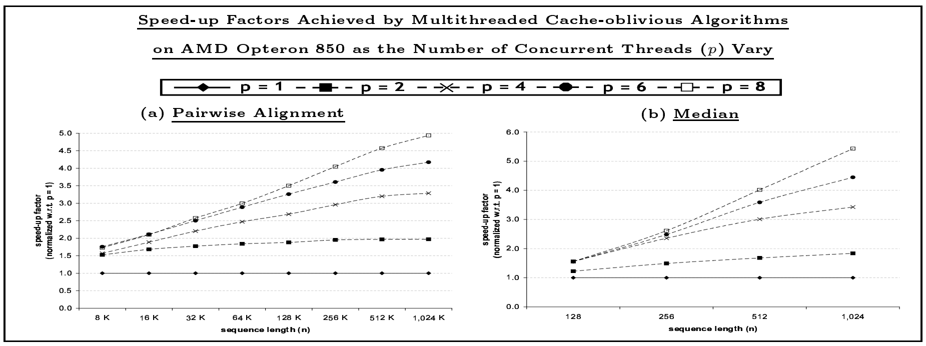

Fig. 3. Speed-up factors (w.r.t. unthreaded code) achieved by multithreaded cache-oblivious pairwise alignment and median algorithms on 8-processor Opteron 850 as number of threads (p) vary. Sequences were randomly generated over $\{ A, T, G, C \}$.

5.2 Median of Three Sequences.

We evaluated the algorithms in Table 4. We used $g_i = 3$, $g_e = 1$ and a mismatch cost of 1 in all experiments.

Algorithm	Description	Time	Space	Cache Miss
MED-CO	our algorithm (see Section 4.2)	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$
MED-Knudsen	Knudsen [11] (Knudsen's code)	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(n^3/B\right)$
MED-H	n^2-space Knudsen (our code)	$\mathcal{O}\left(n^3\right)$	$\mathcal{O}\left(n^2\right)$	$\mathcal{O}\left(n^3/B\right)$
MED-ukk.alloc	Powell [15] (Powell's code)	$\mathcal{O}\left(n+\delta^3\right)$	$\mathcal{O}\left(n+\delta^3\right)$	$\mathcal{O}\left(\delta^3/B\right)$
MED-ukk.checkp	Powell [15] (Powell's code)	$\mathcal{O}\begin{pmatrix} n\log\delta \\ +\delta^3 \end{pmatrix}$	$\mathcal{O}\left(n+\delta^2\right)$	$\mathcal{O}\left(\delta^3/B\right)$

Table 4. Median algorithms in experiments. Here, δ = 3-way edit distance of sequences.

Sequential Performance. For random sequences MED-CO ran at least 1.45 times faster than MED-Knudsen and at least 1.25 times faster than MED-H on Intel Xeon (see Figure 4(a)). MED-ukk.alloc and MED-ukk.checkp 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 (see Figure 4(b)). None of MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp could be run for sequences longer than 640 due to their high space overhead on either machine.

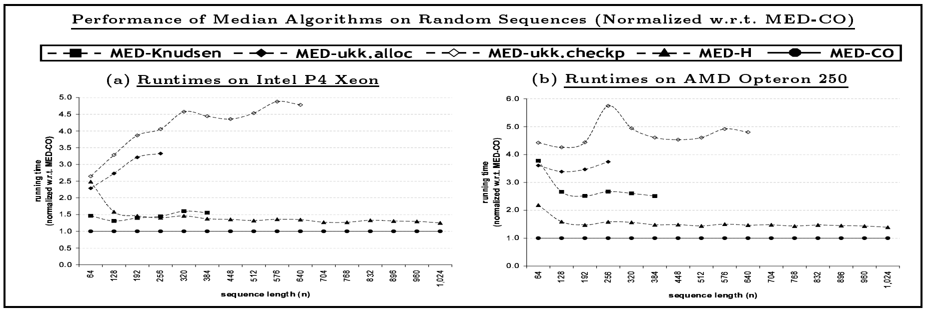

Fig. 4. Comparison of median algorithms (see Table 4). Figures (a) and (b) plot running times on Intel P4 Xeon and AMD Opteron 250, respectively. MED-Knudsen, MED-ukk.alloc and MED-ukk.checkp could not be run for sequences longer than 640. Each data point is the average of 3 independent runs on random strings over $\{ A, T, G, C \}$.

For real-world 16S bacterial rDNA sequences from the *Pseudanabaena group* [6] MED-Knudsen ran around 35–50% slower and MED-ukk.checkp upto 3.2 times slower than MED-CO on Intel Xeon. (see Table 5). Running time of MED-ukk.checkp w.r.t. MED-CO degraded as the alignment cost increased. MED-ukk.alloc could not be run on sequences with alignment cost larger than 299. However, for small alignment costs both MED-ukk.alloc and MED-ukk.checkp ran faster than MED-CO (see triplet 6 in Table 5).

	Running times (in sec) on Intel Xeon for random triples of *16S Bacterial rDNA Sequences from the Pseudanabaena Group* [6] (runtime w.r.t. MED-CO)					
#	Lengths	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 5. Triplets 1–5 were formed by choosing random sequences of length less than 500 from the Pseudanabaena group [6] while triplet 6 was chosen manually in order to keep the alignment cost small. Columns 4–7 give time for a single run with the ratio of that running time to the running time of MED-CO given within parentheses. A '–' denotes that the corresponding algorithm could not be run due to high space overhead.

Parallel Performance. Figure 3(b) shows multithreaded MED-CO achieving reasonable speed-up as the number of processors grow and reaching upto a speed-up factor of 5.5 with 8 processors.

Acknowledgement. We thank Mike Brudno for providing us with the CFTR DNA sequences, and David Zhao for median software packages.

References

1. A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *CACM*, 31:1116–1127, 1988.
2. T. Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Appl. Math.*, 104:45-62, 2000.
3. S. Altschul and B. Erickson. Optimal sequence alignment using affine gap costs. *Bulletin of Math. Biol.*, 48:603–616, 1986.

4. R. Chowdhury, H. Le, and V. Ramachandran. Efficient cache-oblivious string algorithms for Bioinformatics. CS TR-07-03, UT Austin, 2007.
5. R. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. *Proc. SODA*, pp. 591–600, 2006.
6. 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.
7. M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *Proc. FOCS*, pp. 285–297, 1999.
8. M. Frigo and V. Strumpen. Cache-oblivious stencil computations. *Proc. ICS*, pp. 361–366, 2005.
9. O. Gotoh. An improved algorithm for matching biological sequences. *JMB*, 162:705–708, 1982.
10. D. Hirschberg. A linear space algorithm for computing maximal common subsequences. *CACM*, 18(6):341–343, 1975.
11. B. Knudsen. Optimal multiple parsimony alignment with affine gap cost using a phylogenetic tree. *Proc. WABI*, pp. 433–446, 2003. S/W package: `multalign.tar`.
12. J. Kleinberg and E. Tardos. *Algorithm Design.* Addison-Wesely, 2005.
13. E. Myers and W. Miller. Optimal alignments in linear space. *CABIOS*, 4(1):11–17, 1988
14. W. Pearson and D. Lipman. Improved tools for biological sequence comparison. *Proc. NAS-USA*, 85:2444–2448, 1988.
15. D. Powell, L. Allison and T. Dix. Fast, optimal alignment of three sequences using linear gap cost. *J. Theo. Biol.*, 207(3):325–336, 2000. S/W package: `align3str_checkp.tar.gz`.
16. E. Rivas and S. Eddy. A dynamic programming algorithm for RNA structure prediction including pseudoknots. *JMB*, 285(5):2053-68, 1999.
17. J. Seward and N. Nethercote. Valgrind. http://valgrind.kde.org/index.html
18. J. Thomas et al. Comparative analyses of multi-species sequences from targeted genomic regions. *Nature*, 424:788–793, 2003.
19. M. Waterman. *Introduction to Computational Biology.* Chapman & Hall, 1995.