# Fast and Cache-Oblivious Dynamic Programming with Local Dependencies

Philip Bille and Morten Stöckel

Technical University of Denmark, DTU Informatics, Copenhagen, Denmark

**Abstract.** String comparison such as sequence alignment, edit distance computation, longest common subsequence computation, and approximate string matching is a key task (and often computational bottleneck) in large-scale textual information retrieval. For instance, algorithms for sequence alignment are widely used in bioinformatics to compare DNA and protein sequences. These problems can all be solved using essentially the same dynamic programming scheme over a two-dimensional matrix, where each entry depends locally on at most 3 neighboring entries. We present a simple, fast, and cache-oblivious algorithm for this type of local dynamic programming suitable for comparing large-scale strings. Our algorithm outperforms the previous state-of-the-art solutions. Surprisingly, our new simple algorithm is competitive with a complicated, optimized, and tuned implementation of the best cache-aware algorithm. Additionally, our new algorithm generalizes the best known theoretical complexity trade-offs for the problem.

## 1 Introduction

Algorithms for string comparison problems such as sequence alignment, edit distance computation, longest common subsequence computation, and approximate string matching are central primitives in large-scale textual information retrieval tasks. For instance, algorithms for sequence alignment are widely used in bioinformatics for comparing DNA and protein sequences.

All of these problems can be solved using essentially the same dynamic programming scheme over a two-dimensional matrix [12]. The common feature of the dynamic programming solution is that each entry $(i, j)$ in the matrix can be computed in constant time given values of the neighboring entries $(i-1, j)$, $(i-1, j-1)$, and $(i, j-1)$ and the characters at position $i$ and $j$ in the input strings. Combined we refer to these problems as *local dynamic programming string comparison problems*.

In this paper, we revisit local dynamic programming string comparison problems for large-scale input strings. We focus on worst-case and exact solutions, however, the techniques presented are straightforward to combine with the typical heuristic or inexact solutions that filter out parts of the dynamic programming matrix which do not need to be computed. In the context of large-scale strings, I/O efficiency and space usage are key issues for obtaining a fast practical solution.

Our main result is a new simple and cache-oblivious algorithm that outperforms the previous state-of-the-art solutions. Surprisingly, our new simple algorithm is competitive with a complicated, optimized, and tuned implementation of the best cache-aware algorithm. Furthermore, our new algorithm generalizes the best known theoretical trade-offs between time, space, and I/O complexity for the problem.

### 1.1 Memory Models

The memory in modern computers is typically organized in a hierarchy of caches, main memory, and disks. The access time to memory significantly increases with each level of the hierarchy. The *external memory model* [1] abstracts this hierarchy by a simple two-level model consisting of an internal memory of size $M$ and an external memory for storing all remaining data. Data can be transferred between external and internal memory in contiguous blocks of size $B$, and all data must be in internal memory before it can be manipulated. The I/O complexity of an algorithm is the number of transfers of blocks between internal and external memory, called I/O operations (or just I/Os).

The *cache-oblivious model* [11] is an extension of the external memory model with the feature that algorithms do not use knowledge of $M$ and $B$. The model assumes an optimal offline cache replacement strategy, which can be approximated within a small constant factor by standard online cache replacements algorithms such as LRU and FIFO. These properties make cache-oblivious algorithms both I/O efficient on all levels of the memory hierarchy simultaneously and portable between hardware architectures with different memory hierarchies.

### 1.2 Previous Results

Let $X$ and $Y$ be the input strings to a local dynamic programming string comparison problem. For simplicity in the presentation, we assume that $|X| = |Y| = n$. All solutions in this paper are based on two passes over an $(n + 1) \times (n + 1)$ dynamic programming matrix (DPM). First, a *forward pass* computes the length of an optimal path from the top-left corner to the bottom-right corner, and then a *backward pass* computes the actual path by backtracking in the DPM. Finally, the path is translated to the solution to the specific string comparison problem. An overview of the complexities of the previous bounds and of our new algorithm is listed in Table 1.

The first dynamic programming solution is due to Wagner and Fischer [21]. Here, the forward pass fills in and stores all entries in the DPM using $O(n^2)$ time and space. The backward pass then uses the stored entries to efficiently backtrack in $O(n)$ time. In total the algorithm uses $O(n^2)$ time and space. The space usage of this algorithm makes it unsuitable in practice for the sizes of strings that we consider.

Hirschberg [14] showed how to improve the space at the cost of increasing the time for the backward pass. The key idea is to not store all values of the DPM, but instead use a divide and conquer approach in the backward pass to

| Algorithm | Forward Pass | Backward Pass | Space | I/O | CO |
|---|---|---|---|---|---|
| FullMatrix | $O\left(n^2\right)$ | $O(n)$ | $O\left(n^2\right)$ | $O\left(\frac{n^2}{B}+\frac{n}{B}+1\right)$ | Yes |
| Hirschberg | $O\left(n^2\right)$ | $O\left(n^2\right)$ | $O(n)$ | $O\left(\frac{n^2}{B}+\frac{n}{B}+1\right)$ | Yes |
| FastLSA$_k$ | $O\left(n^2\right)$ | $O\left(\frac{n^2}{k}+n\right)$ | $O(nk+D)$ | $O\left(\frac{n^2}{B}+\frac{n}{B}+1\right)$ | No |
| CO | $O\left(n^2\right)$ | $O\left(n^2\right)$ | $O(n)$ | $O\left(\frac{n^2}{BM}+\frac{n}{B}+1\right)$ | Yes |
| FastCO$_k$ | $O\left(n^2\right)$ | $O\left(\frac{n^2}{k}+n\right)$ | $O(nk)$ | $O\left(\frac{n^2k}{BM}+\frac{n}{B}+1\right)$ | Yes |

**Table 1.** Comparison of different algorithms for local dynamic programming. FullMatrix is Wagner and Fischer's algorithm [21], Hirschberg is the linear space algorithm by Hirschberg [14], FastLSA$_k$ is the algorithm by Driga et al. [9, 10] with parameter $k$, CO is the cache-oblivious algorithm by Chowdhury et al. [4,6], and FastCO$_k$ is our new algorithm with parameter $k$.

reconstruct the path. At each step in the backward pass the algorithm splits the DPM into two halves. In each half the algorithm recursively finds an optimal path. The algorithm then combines paths for each half into an optimal path for the entire DPM. The total time for the backward pass increases to $O(n^2)$. Hence, in total the algorithm uses $O(n^2)$ time and $O(n)$ space. Myers and Miller [18] popularized Hirschberg's algorithm for sequence alignment in bioinformatics, and it has since been widely used in practice.

More recently, Driga et al. [9, 10] proposed an advanced recursive algorithm, called FastLSA. The overall idea is to divide the DPM into $k^2$ submatrices, where $k$ is a tunable parameter defined by the user. The forward pass computes and stores the entries of the input boundaries of the submatrices, i.e., the row and column immediately above and to the left of the submatrix. This uses $O(n^2)$ time as the previous algorithms, but now additional $O(nk)$ space is used to store the input boundaries. The backward pass uses the stored input boundaries to speed up the computation of the optimal path by processing only the submatrices that intersect the optimal path. The submatrices are processed recursively until their size is below another user defined threshold $D$. Submatrices of size less than $D$ are processed using Wagner and Fischer's algorithm. The parameter $D$ is chosen such that space for the full matrix algorithm is sufficiently small to fit in a fast cache. With the additional stored input boundaries the time for the backward pass is improved to $O(\frac{n^2}{k}+n)$. In total the algorithm uses $O(n^2+\frac{n^2}{k}+n) = O(n^2)$ time and $O(nk+D)$ space. In addition to the basic recursive idea, the algorithm implements several advanced optimizations to improve the practical running time of the backward pass. For instance, the sizes of the submatrices in recursive calls are reduced according to the entry point of the optimal path in the submatrix and the allocation, deallocation, and caching of the additional space is handled in a non-trivial way. The resulting full algorithm is substantially more complicated than Hirschberg's algorithm.

In practice, Driga et al. [9,10] only consider strings of lengths $\leq 2000$ and in this case they showed that their solution outperforms both Wagner and Firscher's algorithm and Hirschberg's algorithm. For large strings we found that the original implementation was not competitive with the other algorithms. However, by optimizing and simplifying the implementation in the spirit of our new algorithm (see Sec. 2), we were able to obtain a fast and competitive algorithm suitable for large strings.

In terms of I/O complexity all of the above algorithms use $O(\frac{n^2}{B} + \frac{n}{B} + 1)$ I/Os. Furthermore, the algorithm by Driga et al. [9, 10] is cache-aware since it needs to know the parameters of the memory hierarchy in order to optimally select the threshold $D$.

Chowdhury et al. [4, 6] gave a cache-oblivious algorithm that significantly improves this I/O bound. The key idea is to split the DPM into 4 submatrices and apply a simple divide and conquer approach in both passes. The forward pass computes and stores the input boundaries of the 4 submatrices similar to the algorithm by Driga et al. [9,10], however, the computation is now done recursively on each submatrix. This uses $O(n^2)$ time and $O(n)$ space. The backward pass recursively processes the submatrices that intersect the optimal path. This also uses $O(n^2)$ time and $O(n)$ space. Chowdhury et al. [4, 6] showed that the total number of I/Os incurred by the algorithm is $O(\frac{n^2}{MB} + \frac{n}{B} + 1)$. Compared to the previous results, this improves the number of I/Os in the leading quadratic term by a factor $M$. Furthermore, they also showed that this bound is optimal in the sense that any implementation of the local dynamic programming algorithm must use at least this many I/Os. In practice, the reduced number of I/Os significantly improve upon the performance of Hirschberg's algorithm on large strings. To the best of our knowledge, this algorithm is the fastest known practical solution on large strings. Furthermore, the full algorithm is nearly as simple as Hirschberg's algorithm.

The above bounds represent the best known worst-case complexities for general local dynamic programming string comparison. If we restrict the problem in terms of alphabet size or cost function or if we use the properties of a specific local dynamic programming string comparison problem better bounds are known, see e.g., [2, 3, 7, 8, 13, 15–17, 19] and also the survey [20].

### 1.3 Our Results

We present a simple new algorithm with the following complexity.

**Theorem 1.** *Let $X$ and $Y$ be strings of length $n$. Given any integer parameter $k$, $2 \leq k \leq n$, we can solve any local dynamic programming string comparison problem for $X$ and $Y$ using $O(n^2)$ time for the forward pass, $O(\frac{n^2}{k})$ time for the backward pass, and $O(nk)$ space. Furthermore, the algorithm uses $O(\frac{n^2 k}{MB} + \frac{nk}{B} + 1)$ I/Os in a cache-oblivious model.*

Theorem 1 generalizes the previous bounds. In particular, with $k = O(1)$ we match the bounds of the cache-oblivious algorithm by Chowdhury et al. [4, 6].

Furthermore, we obtain the same time-space trade-off for the backward pass as the algorithm by Driga et al. [9, 10] by choosing $k$ accordingly.

We have implemented our algorithm and our optimized and simplified version of the algorithm by Driga et al. [9, 10] with $k = 8, 16, 32$ and compared it with the previous algorithms on strings of length up to $2^{21} = 2097152$ on 3 different hardware architectures. In all our experiments, these algorithms significantly improve the current state-of-the-art cache-oblivious algorithm by Chowdhury et al. [4, 6]. Our algorithms are faster even when $k = 8$ and the performance further improves until $k = 32$. Hence, our results show that a small constant factor additional space can have a significant impact in practice. Furthermore, we found that our new simple and cache-oblivious algorithm is competitive with our optimized, cache-aware, and tuned implementation of the more complicated algorithm by Driga et al. [9, 10]. On one of the tested architectures our new algorithm was even substantially faster than the algorithm by Driga et al. [9,10].

Algorithmically, our new algorithm is a relatively simple combination of the division of the DPM into $k^2$ submatrices from Driga et al. [9,10] and the recursive and cache-oblivious approach from Chowdhury et al. [4, 6]. A similar approach has been studied for solving the problem efficiently on multicore machines [5]. Our results show that this idea can also improve performance on individual cores.

### 1.4 Basic Definitions

For simplicity, we explain our algorithms in terms of the longest common subsequence problem. All of our bounds and techniques generalize immediately to any local dynamic programming string comparison problem.

Let $X$ be a string of length $|X| = n$ of characters from an alphabet $\Sigma$. We denote the character at position $i$ in $X$ by $X[i]$ and the substrings from position $i$ to $j$ by $X[i, j]$. The substrings $X[1, j]$ and $X[i, n]$ are the *prefixes* and *suffixes* of $X$, respectively. A *subsequence* of $X$ is any string $Z$ obtained by deleting characters in $X$. Given two strings $X$ and $Y$ a *common subsequence* is a subsequence of both $X$ and $Y$. A *longest common subsequence (LCS)* of $X$ and $Y$ is a common subsequence of $X$ and $Y$ of maximal length. The *longest common subsequence problem* is to compute an LCS of $X$ and $Y$.

Let $X$ and $Y$ be strings of length $n$. The standard dynamic programming solution fills in an $n + 1 \times n + 1$ DPM $C$ according to the following recurrence.

$$C[i, j] = \begin{cases} 0 & \text{if } j = 0 \vee i = 0, \\ C[i-1, j-1] + 1 & \text{if } i, j > 0 \wedge X[i] = Y[j], \\ \max \begin{cases} C[i, j-1] \\ C[i-1, j] \end{cases} & \text{if } i, j > 0 \wedge X[i] \neq Y[j] \end{cases} \tag{1}$$

The entry $C[i, j]$ is the length of the LCS between prefixes $X[1, i]$ and $Y[1, j]$ and hence the length of LCS of $X$ and $Y$ is $C[n, n]$. Note that each entry $C[i, j]$ depends only on the values in $C[i - 1, j]$, $C[i, j - 1]$, $C[i - 1, j - 1]$ and the characters of $X[i]$ and $Y[j]$. Hence, we can fill in the entries in a top-down left-to-right order. The *LCS path* is the path in $C$ obtained by backtracking from

| | | s | u | r | g | e | r | y |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| s | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| u | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |
| r | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 |
| v | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 |
| e | 0 | 1 | 2 | 3 | 3 | 4 | 4 | 4 |
| y | 0 | 1 | 2 | 3 | 3 | 4 | 4 | 5 |

(a)

| | | s | u | r | g | e | r | y |
|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| s | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| u | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 |
| r | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 |
| v | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 3 |
| e | 0 | 1 | 2 | 3 | 3 | 4 | 4 | 4 |
| y | 0 | 1 | 2 | 3 | 3 | 4 | 4 | 5 |

(b)

**Fig. 1.** Computing the LCS of `survey` and `surgery`. (a) The dynamic programming matrix. (b) The LCS path. Each diagonal edge corresponds to a character of the LCS. The resulting LCS is `surey`.

$C[n, n]$ to $C[0, 0]$. Each diagonal edge in the LCS path corresponds to a character of the LCS. See Fig. 1 for an example.

## 2 A New Algorithm for LCS

We now present our new algorithm for LCS. We describe our algorithm in the same basic framework as Chowdhury et al. [4, 6].

Let $X$ and $Y$ be strings of length $n$ and let $k$ be the parameter for the algorithm. For simplicity, we assume that $n$ and $k$ are powers of 2. Our algorithm repeatedly uses a simple recursive procedure for computing the output boundary of the submatrix given the input boundary. We explain this procedure first and then give the full algorithm.

### 2.1 Computing Boundaries

Given the input boundary of a DPM for two strings of length $n$ we can compute the output boundary by computing the entries in the DPM in the standard top-down left-to-right order. This uses $O(n^2)$ time and $O(n)$ space, since we compute each entry in constant time and we only need to store the last two rows of the DPM during the algorithm. However, the number of I/Os incurred is $O(\frac{n^2}{B})$. The following simple recursive algorithm, similar to the one presented in Chowdhury et al. [4,6], improves this bound to $O(\frac{n^2}{MB})$ I/Os. If $n = 1$ we compute the output boundary directly using recurrence 1. Otherwise, we split the matrix into $k^2$ submatrices of size $\frac{n}{k} \times \frac{n}{k}$. We recursively compute the output boundaries for each submatrix by processing them in a top-down left-to-right order. At each recursive call to process a submatrix, the input boundary consists of the output boundary of the submatrix immediately above, to the left, and above-left. Hence with this ordering, the input boundary is available for each recursive call.

The algorithm uses $O(n^2)$ time and $O(n)$ space as before, i.e., we only need to store the boundaries of the submatrices that are currently being processed.

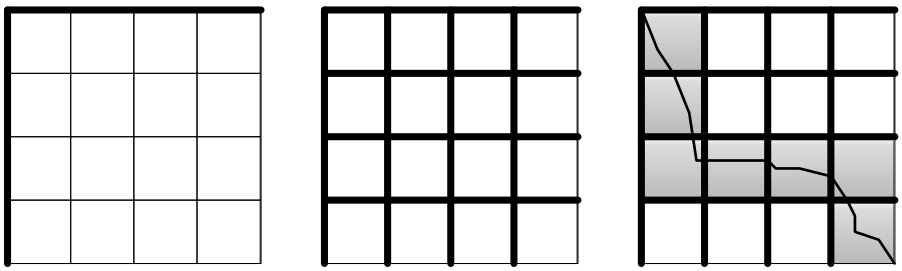


**Fig. 2.** States of the algorithm for $k = 4$. (a) A partition of the DPM into $4 \times 4$ submatrices. Thick lines indicate the initially stored boundaries. (b) After the forward pass. (c) After the backward pass. Only the shaded submatrices intersecting the LCS path are processed.

Let $I_1(n, k)$ denote the number of I/Os incurred by the algorithm on strings of length $n$ with parameter $k$. If $n$ is sufficiently small such that the recursive computation is done within internal memory, the algorithm only incurs I/Os to read and write the input strings and to read and write the boundaries. The length of the input strings and boundaries is $O(n + nk) = O(nk)$ and hence the number of I/Os is $O(\frac{nk}{B} + 1)$. Otherwise, the algorithm additionally creates $k^2$ subproblems of strings of length $\frac{n}{k}$ each. Thus, the total number of I/Os is given by the following recurrence.

$$I_1(n,k) = \begin{cases} O(\frac{nk}{B} + 1) & \text{if } n \leq \alpha_1 M, \\ k^2 I_1(\frac{n}{k}, k) + O(\frac{nk}{B} + 1) & \text{otherwise.} \end{cases} \tag{2}$$

Here, $\alpha_1$ is a suitable constant such that all computations of strings of length $\alpha_1 M$ are done entirely within memory. It follows that the total number of I/Os is $I_1(n, k) = O(\frac{n^2 k}{MB} + \frac{nk}{B} + 1)$.

### 2.2 Computing the LCS

We now present the full algorithm to compute the LCS in $C$. The algorithm is recursive and works as follows. If $n = 1$ we simply compute an optimal path directly using recurrence (1). Otherwise, we proceed in the following steps (see Fig 2).

*Step 1: Forward Pass* Partition $C$ into $k^2$ square submatrices of size $\frac{n}{k} \times \frac{n}{k}$. Compute and store the input boundaries of each submatrix in a top-down left-to-right order. We compute the boundaries using the algorithm from Sect. 2.1.

*Step 2: Backward pass* Compute an optimal LCS path through the submatrices from the bottom-right to the top-left. At each step we recursively find an optimal

path through a submatrix $C'$ given the input boundary (computed in step 1) and a point on the optimal path on the output boundary. Depending on the exit point on the input boundary of the computed optimal LCS path through $C'$ we continue in the submatrix above, to the left, or above-left of $C'$ using the exit point as the point on the output boundary in the next step.

*Step 3: Output LCS* Finally, concatenate the path through the submatrices to form an optimal LCS path in $C$ and output the corresponding LCS.

### 2.3 Analysis

First consider the time complexity of the algorithm. Step 1 (the forward pass) uses $O(n^2)$ time. In step 2 (the backward pass), we only process the submatrices that are intersected by the optimal path. Since any path from $(n, n)$ to $(0, 0)$ can intersect at most $2k-1$ submatrices, step 2 uses $O((2k-1)\cdot\frac{n^2}{k^2}+n) = O(\frac{n^2}{k}+n)$ time. Finally, step 3 concatenates the pieces of the path and outputs the LCS in $O(n)$ time. In total, the algorithm uses $O(n^2)$ time.

Next consider the space used by the algorithm. Let $S(n, k)$ denote the space for a subproblem of size $n$ with parameter $k$. The stored input boundaries use $O(nk)$ space and the recursive call uses $S(n/k, k)$ space. Hence, the total space $S(n, k)$ is given by the recurrence

$$S(n,k) = \begin{cases} O(1) & \text{if } n = O(1), \\ S(n/k,k) + O(nk) & \text{otherwise.} \end{cases}$$

It follows that the space used by the algorithm is $S(n, k) = O(nk)$.

Next consider the I/O complexity. Let $I_2(n, k)$ denote I/O complexity of the algorithm on strings of length $n$ with parameter $k$. If $n$ is sufficiently small such that the recursive computation is done within internal memory, the algorithm incurs $O(\frac{nk}{B}+1)$ I/Os by similar arguments as in the analysis above. Otherwise, the algorithm additionally does $k^2 - 1$ boundary computations in step 1 on subproblems of size $\frac{n}{k}$ and recursively creates $2k - 1$ subproblems of strings of length $\frac{n}{k}$. Hence, the total number of I/Os is given by

$$I_2(n,k) = \begin{cases} O(\frac{nk}{B}+1) & \text{if } n \leq \alpha_2 M, \\ (k^2-1)I_1\left(\frac{n}{k},k\right) + (2k-1)I_2\left(\frac{n}{k},k\right) + O\left(\frac{nk}{B}+1\right) & \text{otherwise.} \end{cases} \tag{3}$$

Here, $\alpha_2$ is a suitable constant such that computation is done entirely in memory. It follows that $I_2(n, k) = O(\frac{n^2k}{MB} + \frac{nk}{B} + 1)$.

In summary, our algorithm for LCS uses $O(n^2)$ time for the forward pass, $O(\frac{n^2}{k} + n)$ time for the backward pass, $O(nk)$ space, and $O(\frac{n^2k}{MB} + \frac{nk}{B} + 1)$ I/Os. Since the algorithm only uses the local dependencies of LCS these bounds hold for any local dynamic programming string comparison problem. Hence, this completes the proof of Theorem 1.

# 3 Experimental Results

## 3.1 Setup

We have compared the following algorithms.

FullMatrix Wagner and Fischer's [21] original algorithm in our own implementation.

Hirschberg Hirschberg's [14] linear space divide and conquer algorithm in our own implementation.

CO The cache-oblivious algorithm by Chowdhury et al. [4, 6]. We have tested the original implementation of the algorithm.

FastLSA$_k$ The FastLSA algorithm by Driga et al. [9, 10] with parameter $k$. We used an optimized version of the original implementation of the algorithm. The optimization improves and simplifies parameter passing in the recursion, and the allocation and deallocation of the auxiliary arrays. In our experiments, we report the results for $k = 8, 16, 32$, since larger values of $k$ did not further improve the performance of the algorithm. Furthermore, we have tuned the threshold parameter $D$ for each of the tested hardware architectures.

FastCO$_k$ An implementation of our new algorithm with parameter $k$. As with FastLSA, we report the results for $k = 8, 16, 32$. In our experiments we found that the choice of $k$ did not affect the forward pass. For simplicity, we therefore fixed $k = 2$ for the forward and only varied $k$ in the backward pass.

We compared the algorithms on the following 3 architectures.

Intel i7 2.66GHz. 32KB L1, 256KB L2, 8MB L3 cache. 4GB memory
AMD X2 - 2.5GHz. 64KB L1, 512KB L2 cache. 4GB memory
Intel M - 1.6GHz. 2MB L2 cache. 1GB memory

All algorithms were implemented in C/C++ and compiled using the `gcc` 3.4 compiler. We tested the performance of the algorithms on strings of lengths $n = 2^i$, for $i = 16, 17, 18, 19, 20, 21$, i.e., the largest strings are slighty larger than 2 million. The strings are DNA strings taken from the standardized text collection of Pizza&Chili Corpus [1]. We have experimented with other types of strings but found only very small differences in performance. This is likely due to the small difference between worst-case and best-case performance. For brevity, we therefore focus DNA strings from the standardized text collection in our experiments. Additionally, we have used Cachegrind[2] to simulate the algorithms on a standard memory hierarchy with 64KB L1 and 512KB L2 caches.

## 3.2 Results

The results of the running time and the Cachegrind experiments are listed in Tables 2 and 3. Results for FullMatrix are not reported since they were either

[1] `pizzachili.dcc.uchile.cl` or `pizzachili.di.unipi.it`.
[2] `valgrind.org/info/tools.html#cachegrind`

Intel i7

| $n$ | HB | CO | $\text{FLSA}_8$ | $\text{FLSA}_{16}$ | $\text{FLSA}_{32}$ | $\text{FCO}_8$ | $\text{FCO}_{16}$ | $\text{FCO}_{32}$ |
|---|---|---|---|---|---|---|---|---|
| $2^{16}$ | $0.016h$ | $0.012h$ | $0.009h$ | $0.009h$ | **0.008h** | $0.009h$ | $0.009h$ | **0.008**$h$ |
| $2^{17}$ | $0.063h$ | $0.049h$ | $0.0387h$ | $0.036h$ | **0.033h** | $0.036h$ | $0.035h$ | $0.034h$ |
| $2^{18}$ | $0.251h$ | $0.194h$ | $0.150h$ | $0.144h$ | **0.132h** | $0.143h$ | $0.136h$ | $0.134h$ |
| $2^{19}$ | $1.003h$ | $0.775h$ | $0.584h$ | $0.559h$ | **0.529h** | $0.565h$ | $0.539h$ | $0.530h$ |
| $2^{20}$ | $4.059h$ | $3.129h$ | $2.320h$ | $2.290h$ | $2.127h$ | $2.238h$ | $2.258h$ | **2.100h** |
| $2^{21}$ | $16.105h$ | $12.297h$ | $9.544h$ | $9.022h$ | $8.741h$ | $9.036h$ | $8.611h$ | **8.355h** |

AMD X2

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $2^{16}$ | $0.017h$ | $0.009h$ | $0.010h$ | $0.010h$ | $0.010h$ | **0.007h** | **0.007h** | **0.007h** |
| $2^{17}$ | $0.069h$ | $0.037h$ | $0.041h$ | $0.039h$ | $0.038h$ | $0.028h$ | $0.027h$ | **0.026h** |
| $2^{18}$ | $0.278h$ | $0.149h$ | $0.169h$ | $0.159h$ | $0.156h$ | $0.114h$ | $0.108h$ | **0.104h** |
| $2^{19}$ | $1.123h$ | $0.597h$ | $0.685h$ | $0.640h$ | $0.624h$ | $0.455h$ | $0.430h$ | **0.418h** |
| $2^{20}$ | $4.474h$ | $2.389h$ | $2.752h$ | $2.574h$ | $2.498h$ | $1.846h$ | $1.721h$ | **1.671h** |
| $2^{21}$ | $17.949h$ | $9.442h$ | $11.007h$ | $10.337h$ | $9.950h$ | $7.278h$ | $6.873h$ | **6.685h** |

Intel M

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $2^{16}$ | $0.027h$ | $0.021h$ | $0.0140h$ | $0.013h$ | - | $0.015h$ | **0.012h** | - |
| $2^{17}$ | $0.108h$ | $0.083h$ | $0.0571h$ | $0.053h$ | - | $0.061h$ | **0.050h** | - |
| $2^{18}$ | $0.438h$ | $0.334h$ | $0.227h$ | $0.218h$ | - | $0.234h$ | **0.200h** | - |
| $2^{19}$ | $1.800h$ | $1.337h$ | $0.945h$ | $0.889h$ | - | $0.928h$ | **0.852h** | - |
| $2^{20}$ | $7.170h$ | $5.325h$ | $3.814h$ | $3.575h$ | - | $3.697h$ | **3.481h** | - |
| $2^{21}$ | $28.999h$ | $20.601h$ | $15.283h$ | $14.994h$ | - | $14.730h$ | **14.529h** | - |

**Table 2.** Performance results for HIRSCHBERG (HB), CO, $\text{FASTLSA}_k$ ($\text{FLSA}_k$), $\text{FASTCO}_k$ ($\text{FCO}_k$) on Intel i7, AMD X2, and Intel M. The fastest running times for each row is in boldface.

infeasible or far from competitive in all our experiments. Furthermore, for Intel M we only report result for FASTLSA and FASTCO with parameter $k$ up to 16 due to the small memory of this machine.

From Table 2 we see that $\text{FASTLSA}_{32}$ or $\text{FASTCO}_{32}$ significantly outperform the current state-of-the-art cache-oblivious algorithm CO. Compared with HIRSCHBERG, $\text{FASTLSA}_{32}$ and $\text{FASTCO}_{32}$ are a about a factor 1.8 to 2.6 faster. Surprisingly, our simple cache-oblivious $\text{FASTCO}_k$ is competitive with our optimized and cache-aware implementation of the more complicated $\text{FASTLSA}_k$. For the AMD X2 architecture, FASTCO is even significantly faster than FASTLSA. We outperform previous results even when $k = 8$ and our performance further improves until $k = 32$.

Our Cachegrind experiments listed in Table 3 show that $\text{FASTCO}_k$ executes a similar number of instructions as $\text{FASTLSA}_k$ and far less than both HIRSCHBERG and CO. Furthermore, $\text{FASTCO}_k$ incurs at least a factor 2000 less cache misses in both L1 and L2 cache compared to $\text{FASTLSA}_k$. Note that this corresponds well with the difference in the theoretical I/O complexity between the algorithms. The fewest number of cache misses are incurred by CO closely followed by $\text{FASTCO}_k$.

Instructions Executed $\times 10^9$

| $n$ | HB | CO | $FCO_8$ | $FCO_{16}$ | $FCO_{32}$ | $FLSA_8$ | $FLSA_{16}$ | $FLSA_{32}$ |
|---|---|---|---|---|---|---|---|---|
| $2^{16}$ | 177.30 | 123.84 | 77.122 | 72.823 | **70.670** | 84.463 | 79.043 | 76.584 |
| $2^{17}$ | 708.87 | 494.68 | 308.06 | 290.91 | **282.53** | 337.68 | 315.95 | 306.07 |
| $2^{18}$ | 2,835.4 | 1,978.2 | 1,221.2 | 1,163.4 | **1,129.3** | 1,352.9 | 1,263.7 | 1,223.8 |
| $2^{19}$ | 11,339 | 7,907.8 | 4,914.2 | 4,646.4 | **4,514.6** | 5,416.1 | 5,057 | 4,894.5 |

L1 cache misses $\times 10^6$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $2^{16}$ | 1,090 | **0.855** | 1.682 | 1.980 | 2.070 | 1,293 | 1,162 | 1,154 |
| $2^{17}$ | 4,639 | **1.952** | 3.823 | 4.566 | 7.49 | 5,156 | 4,834 | 4,626 |
| $2^{18}$ | 18,866 | **5.916** | 11.23 | 12.29 | 17.41 | 20,640 | 19,4654 | 18,789 |
| $2^{19}$ | 76,654 | **19.85** | 37.27 | 39.59 | 46.55 | 83,201 | 77,630 | 75,355 |

L2 cache misses $\times 10^6$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $2^{16}$ | 604.6 | **0.345** | 1.038 | 1.373 | 1.711 | 1,151 | 1,146 | 1,150 |
| $2^{17}$ | 3,207 | **0.594** | 1.949 | 2.49 | 5.2 | 4,575 | 4,579 | 4,581 |
| $2^{18}$ | 16,517 | **1.312** | 4.385 | 5.067 | 9.373 | 18,741 | 18,303 | 18,290 |
| $2^{19}$ | 71,629 | **3.416** | 11.689 | 15.12 | 18.792 | 82,131 | 75,219 | 73,117 |

**Table 3.** Cachegrind results for HIRSCHBERG (HB), CO, FASTLSA$_k$ (FLSA$_k$), FASTCO$_k$ (FCO$_k$) on a 64K L1 and 512K L2 cache hierarchy. Lowest instruction/miss count shown in boldface. We could only test inputs up to $n = 2^{19}$ due to the overhead of the Cachegrind simulation.

The difference between the number of cache misses incurred by FASTCO$_k$ and FASTLSA$_k$ is much larger than the difference in their running time. The main reason for this is because the number of cache misses incurred relative to the total number of instructions executed is low (around 4% for FASTLSA$_k$). Ultimately, the simple FASTCO$_k$ simultaneously achieves good cache performance and a low instruction count making it competitive with current state-of-the-art algorithms.

## Acknowledgments

We would like to thank the authors of Driga et al. [9, 10] and Chowdhury et al. [4, 6] for providing us with the source code of their algorithms.

## References

1. Aggarwal, A., Vitter, J.S.: The Input/Output complexity of sorting and related problems. Commun. ACM 31(9), 1116–1127 (1988)
2. Bille, P.: Faster approximate string matching for short patterns. Theory Comput. Syst. (2011), to appear
3. Bille, P., Farach-Colton, M.: Fast and compact regular expression matching. Theoret. Comput. Sci. 409(3), 486 – 496 (2008)

4. Chowdhury, R.A., Ramachandran, V.: Cache-oblivious dynamic programming. In: Proc. 17th Symp. on Discrete Algorithms. pp. 591–600 (2006)
5. Chowdhury, R.A., Ramachandran, V.: Cache-efficient dynamic programming algorithms for multicores. In: Proc. 20th Symp. on Parallelism in Algorithms and Architectures. pp. 207–216 (2008), http://doi.acm.org/10.1145/1378533.1378574
6. Chowdhury, R.A., Le, H.S., Ramachandran, V.: Cache-oblivious dynamic programming for bioinformatics. Trans. Comput. Biol. and Bioinformatics 7, 495–510 (2010)
7. Cole, R., Hariharan, R.: Approximate string matching: A simpler faster algorithm. SIAM J. Comput. 31(6), 1761–1782 (2002)
8. Crochemore, M., Landau, G.M., Ziv-Ukelson, M.: A subquadratic sequence alignment algorithm for unrestricted scoring matrices. SIAM J. Comput. 32(6), 1654–1673 (2003)
9. Driga, A., Lu, P., Schaeffer, J., Szafron, D., Charter, K., Parsons, I.: FastLSA: A fast, linear-space, parallel and sequential algorithm for sequence alignment. In: Proc. Intl. Conf. on Parallel Processing. pp. 48–57 (2005)
10. Driga, A., Lu, P., Schaeffer, J., Szafron, D., Charter, K., Parsons, I.: FastLSA: A fast, linear-space, parallel and sequential algorithm for sequence alignment. Algorithmica 45, 337–375 (2006)
11. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. 40th Symp. Foundations of Computer Science. pp. 285 – 297 (1999)
12. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge (1997)
13. Hermelin, D., Landau, G.M., Landau, S., Weimann, O.: A unified algorithm for accelerating edit-distance computation via text-compression. In: Proc. 26th Symp. Theoretical Aspects of Computer Science. Leibniz International Proceedings in Informatics (LIPIcs), vol. 3, pp. 529–540 (2009)
14. Hirschberg, D.S.: A linear space algorithm for computing maximal common subsequences. Commun. ACM 18(6), 341–343 (1975)
15. Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest common subsequences. Commun. ACM 20, 350–353 (1977)
16. Landau, G.M., Vishkin, U.: Fast parallel and serial approximate string matching. J. Algorithms 10, 157–169 (1989)
17. Masek, W., Paterson, M.: A faster algorithm for computing string edit distances. J. Comput. System Sci. 20, 18–31 (1980)
18. Myers, E.W., Miller, W.: Optimal alignments in linear space. Comput. Appl. Biosci. 4(1), 11–17 (1988)
19. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. J. ACM 46(3), 395–415 (1999)
20. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. 33(1), 31–88 (2001)
21. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. J. ACM 21, 168–173 (1974)