Linear space string correction algorithm using the Damerau-Levenshtein distance

Chunchun Zhao* and Sartaj Sahni

From 8th IEEE International Conference on Computational Advances in Bio and medical Sciences (ICCABS 2018)
Las Vegas, NV, USA. 18–20 October 2018

Abstract

Background: The Damerau-Levenshtein (DL) distance metric has been widely used in the biological science. It tries to identify the similar region of DNA, RNA and protein sequences by transforming one sequence to the another using the substitution, insertion, deletion and transposition operations. Lowrance and Wagner have developed an $O(mn)$ time $O(mn)$ space algorithm to find the minimum cost edit sequence between strings of length $m$ and $n$, respectively. In our previous research, we have developed algorithms that run in $O(mn)$ time using only $O(s \ast \min(m, n) + m + n)$ space, where $s$ is the size of the alphabet comprising the strings, to compute the DL distance as well as the corresponding edit sequence. These are so far the fastest and most space efficient algorithms. In this paper, we focus on the development of algorithms whose asymptotic space complexity is linear.

Results: We develop linear space algorithms to compute the Damerau-Levenshtein (DL) distance between two strings and determine the optimal trace (corresponding edit operations.) Extensive experiments conducted on three computational platforms—Xeon E5 2603, I7-x980 and Xeon E5 2695—show that, our algorithms, in addition to using less space, are much faster than earlier algorithms.

Conclusion: Besides using less space than the previously known algorithms, significant run-time improvement was seen for our new algorithms on all three of our experimental platforms. On all platforms, our linear-space cache-efficient algorithms reduced run time by as much as 56.4% and 57.4% in respect to compute the DL distance and an optimal edit sequences compared to previous algorithms. Our multi-core algorithms reduced the run time by up to 59.3% compared to the best previously known multi-core algorithms.

Keywords: Edit distance, Damerau-Levenshtein distance, Linear space, String correction

*Correspondence: zhaochunchun@gmail.com
Department of Computer and Information Science and Engineering, University of Florida, Gainesville, 32611 FL, USA

Background

Introduction

The Damerau-Levenshtein (DL) distance between two strings is the minimum number of substitutions, inserts, deletes, and transpositions of adjacent characters required to transform one string into the other. Some of the applications of the DL are spelling error correction [1–3], comparing packet traces [4], data mining and clustering [5], quantifying the similarity of biological sequences, and gene function prediction [6], analysis of B cell receptor repertoire data [7], virus detection in software [8], clustering of RNA-seq reads segments [9], DNA repeats detection [10], and codes for DNA based storage [11]. In some of these applications (e.g., spelling error correction), the strings are rather small while in others (e.g., comparing protein sequences) the strings could be tens of thousands of characters long [12], and in yet others (e.g., comparing chromosomes) they could be millions of characters long [13].

Other string edit distances used in the literature permit only a proper subset of the operations permitted by the DL distance. For example, in the Levenshtein distance [14] transpositions are not permitted, in the Hamming distance [15] only substitutions are permitted, and in the
Fig. 1 DL distance recurrence. a substitution. b insertion. c deletion. d translate A[k:i] to B[l:j] where (a_k, b_l) and (b_j, a_i) form a transposition opportunity.

Fig. 2 DL trace example [18]

Fig. 3 Traces with and without center crossings [18]. a No center crossing. b With center crossing.
Table 1 Memory usage for DL distance algorithms on Xeon4

| Algorithm          | Memory  |
|--------------------|---------|
| Lowrance and Wagner| 30100.88 MB |
| LS_DL              | 41.93 MB   |
| LS_DL2             | 8.82 MB    |
| LS_Strip           | 40.98 MB   |
| LS_Strip2          | 8.73 MB    |

Jaro distance [16], only transpositions are permitted. The correct distance metric to use depends on the application. In the applications cited above, the DL distance is used as all 4 edit operations are permitted.

Lowrance and Wagner [17] considered a generalization of DL distance to the case when substitutions, inserts, deletes, and transpositions have different costs. Through a suitable choice of weights, the weighted DL distance can be made equal to the DL distance, Levenshtein distance, Hamming distance, and Jaro distance.

Lowrance and Wagner [17] have developed an $O(mn)$ time and $O(mn)$ space algorithm to find the minimum cost edit sequence (i.e., sequence of substitutions, inserts, deletes, and transpositions) that transforms a given string of length $m$ into a given string of length $n$ provided that $2T \geq I + D$, where $T$, $I$, and $D$, are respectively, the cost of a transposition, insertion, and deletion. In the DL distance, $T = I = D = 1$ and so, $2T = I + D$. Hence, the algorithm of Lowrance and Wagner [17] may be used to compute the DL distance as well as the corresponding edit sequence in $O(mn)$ time and $O(mn)$ space. This observation has also been made in [2]. In [18] we developed algorithms that run in $O(mn)$ time using only $O(s \times \min(m,n) + m + n)$ space, where $s$ is the size of the alphabet comprising the strings, to compute the DL distance as well as the corresponding edit sequence. Since $s << m$ and $s << n$ in most applications (e.g., $s = 20$ for protein sequences), this reduction in space enables the solution of much larger instances than is possible using the algorithm of [17]. Our algorithms in [18] are much faster as well. In this paper, we develop algorithms to compute the DL distance and corresponding edit sequence using $O(m + n)$ space and $O(mn)$ time. Extensive experimentation using 3 different platforms indicates that the algorithms of this paper are also faster than those of [18]. In fact, our fastest algorithm for the DL distance is up to 56.4% faster than the fastest algorithm in [18] when run on a single core. The single core speedup to find the corresponding edit sequence is up to 57.4%. Our algorithms may be adapted to run on multicores providing a speedup of up to 59.3%.

DL dynamic programming recurrences

Let $A[1 : m] = a_1a_2\cdots a_m$ and $B[1 : n] = b_1b_2\cdots b_n$ be two strings of length $m$ and $n$, respectively. Let $H_{ij}$ be the DL distance between $A[1 : i]$ and $B[1 : j]$. So, $H_{mn}$ is the DL distance between $A$ and $B$. The dynamic programming recurrence for $H$ is given below [17, 18].

$$H_{i0} = i, \quad H_{0j} = j, \quad 0 \leq i \leq m, \quad 0 \leq j \leq n$$ (1)

When $i > 0$ and $j > 0$,

$$H_{ij} = \min \left\{ \begin{array}{rl}
H_{i-1,j-1} + c(a_i, b_j) \\
H_{ij-1} + 1 \\
H_{i-1,j} + 1 \\
H_{k-1,l-1} + (i - k - 1) + 1 + (j - l - 1)
\end{array} \right. \quad (2)$$
where \( c(a_i, b_j) \) is 1 if \( a_i \neq b_j \) and 0 otherwise, \( k = \text{lastA}[i][j] \) is the last (i.e., rightmost) occurrence of \( b_j \) in \( A \) that precedes position \( i \) of \( A \), and \( l = \text{lastB}[j][i] \) is the last occurrence of \( a_i \) in \( B \) that precedes position \( j \) of \( B \). When \( k \) or \( l \) do not exist, case 4 of Eq. 2 does not apply.

The four cases in Eq. 2 correspond to the four allowable edit operations: substitution, insertion, deletion and transposition. These cases are illustrated in Fig. 1, which depicts the possibilities for an optimal transformation of \( A[1:i] \) into \( B[1:j] \). Figure 1a illustrates the first case, which is to optimally transform \( A[1:i−1] \) into \( B[1:j−1] \) and then substitute \( b_j \) for \( a_i \). If \( a_i = b_j \), the substitution cost is 0, otherwise it is 1. Figure 1b shows the second case. Here, \( A[1:i] \) is optimally transformed into \( B[1:j−1] \) and then \( b_j \) is inserted at the end. In the third case (Fig. 1c) \( A[1:i−1] \) is optimally transformed into \( B[1:j] \) and then \( a_i \) is deleted. For the fourth and final case (Fig. 1d), assume that \( k \) and \( l \) exist. In this case, we are going to transpose \( a_k \) and \( a_i \). We first optimally transform \( A[1:k−1] \) into \( B[1:l−1] \). Since only adjacent characters may be transposed, the transposition of \( a_k \) and \( a_i \) must be preceded by a deletion of \( a_{k+1} \) through \( a_{i−1} \), which results in \( a_k \) and \( a_i \) becoming adjacent. Following the transposition, we insert \( b_{l+1} \) through \( b_{j−1} \) between the transposed \( a_k \) and \( a_i \), thereby transforming \( A[1:i] \) into \( B[1:j] \). The cost of optimally transforming \( A[1:k−1] \) into \( B[1:l−1] \) is \( H_{k−1,l−1} \). The ensuing deletions have a cost of \( i−k−1 \) as this is the number of deletions performed, the transposition \( a_k \) and \( a_i \) costs 1, and the final inserts cost \( l−k−1 \). So, the overall cost of case 4 is \( H_{k−1,l−1} + (i−k−1) + 1 + (j−l−1) \).

Table 2  Cache misses for DL distance algorithms, in millions, on Xeon4

| A   | B   | LS_DL | LS_DL2 | Strip_DL | Strip_DL2 | L vs L2 | S vs S2 | L2 vs S2 |
|-----|-----|-------|--------|----------|-----------|---------|---------|----------|
| 40000 | 40000  | 265   | 14     | 6        | 1         | 94.9%   | 76.6%   | 90.4%    |
| 80000 | 80000  | 715   | 53     | 16       | 5         | 92.6%   | 66.0%   | 89.9%    |
| 120000 | 120000 | 2,180 | 121    | 42       | 11        | 94.4%   | 73.1%   | 90.7%    |
| 160000 | 160000 | 10,652| 247    | 63       | 20        | 97.7%   | 68.4%   | 91.9%    |
| 200000 | 200000 | 19,751| 397    | 147      | 32        | 98.0%   | 78.0%   | 91.9%    |
| 240000 | 240000 | 24,257| 570    | 133      | 49        | 97.7%   | 63.2%   | 91.4%    |
| 280000 | 280000 | 38,119| 781    | 188      | 66        | 98.0%   | 65.0%   | 91.6%    |
| 320000 | 320000 | 44,815| 1,021  | 242      | 86        | 97.7%   | 64.4%   | 91.6%    |
| 360000 | 360000 | 61,296| 1,290  | 1,352    | 111       | 97.9%   | 91.8%   | 91.4%    |
| 400000 | 400000 | 160,118| 1,587  | 2,407    | 136       | 99.0%   | 94.4%   | 91.5%    |

Fig. 5  Run time for DL distance algorithms on Xeon4
Table 3  Run time in $hh : mm : ss$ for DL distance algorithms on Xeon4

| A  | B  | LS_DL | LS_DL2 | Strip_DL | Strip_DL2 | L vs L2 | S vs S2 | L2 vs S2 |
|----|----|-------|--------|----------|-----------|--------|--------|--------|
| 40000 | 40000 | 0:00:17 | 0:00:09 | 0:00:13 | 0:00:08 | 47.6% | 38.5% | 15.9% |
| 80000 | 80000 | 0:01:02 | 0:00:36 | 0:00:50 | 0:00:31 | 41.3% | 38.1% | 15.4% |
| 120000 | 120000 | 0:02:40 | 0:01:22 | 0:01:52 | 0:01:09 | 48.9% | 38.3% | 15.4% |
| 160000 | 160000 | 0:05:37 | 0:02:26 | 0:03:19 | 0:02:03 | 56.6% | 38.2% | 15.7% |
| 200000 | 200000 | 0:09:38 | 0:03:48 | 0:05:14 | 0:03:12 | 60.6% | 38.7% | 15.5% |
| 240000 | 240000 | 0:13:37 | 0:05:29 | 0:07:28 | 0:04:37 | 59.8% | 38.2% | 15.7% |
| 280000 | 280000 | 0:18:34 | 0:07:28 | 0:10:10 | 0:06:17 | 59.8% | 38.2% | 15.7% |
| 320000 | 320000 | 0:24:13 | 0:09:45 | 0:13:17 | 0:08:13 | 59.7% | 38.2% | 15.8% |
| 360000 | 360000 | 0:33:10 | 0:12:21 | 0:17:22 | 0:10:24 | 62.8% | 40.1% | 15.8% |
| 400000 | 400000 | 0:37:55 | 0:15:15 | 0:20:46 | 0:12:50 | 59.8% | 38.2% | 15.8% |

The algorithm of Lowrance and Wagner [17] computes $H_{m,n}$ using a $m \times n$ array for $H$ and one-dimensional arrays of size $s$ for lastA and lastB, where $s$ is the size of the alphabet from which the strings A and B are drawn. It computes the $H_{i,j}$'s by rows beginning with row 1 and within a row, the elements are computed left-to-right by columns. While algorithm LS_DL of [18] also computes H by rows and within a row by columns left-to-right, it does this using a one-dimensional array of size $n$ for the current row being computed, a one-dimensional array of size $s$ for lastA, and an $s \times n$ array $T$ with the property that if $w$ is the last row of $H$ computed so far such that $A[w] = c$, then $T[c][\ast] = H[w−1][\ast]$. As noted in [18] when $m < n$, we may swap the strings A and B resulting in a space requirement of $O(s \min(m,n) + n)$. The time complexity of LS_DL is $O(mn)$. A cache-efficient version, Strip_DL, of LS_DL that computes $H$ by strips whose width is no larger than the cache size is also developed in [18]. This cache efficient algorithm has the same asymptotic time and space complexities as does LS_DL. But, as demonstrated in [18], Strip_DL is much faster than LS_DL.

The linear space algorithms we develop in this paper use a refined dynamic programming recurrence for $H$. We make the observation that when either $a_i = b_j$ or $\min(i−k,j−l) \geq 2$ in the fourth case of Eq. 2, then it is sufficient to consider only the first three cases. To see this, note that when $a_i = b_j$ the transposition of $a_k$ and $a_i$ done following the deletion of $a_{k+1}$ through $a_{i−1}$ in case 4 (Fig. 1d) is unnecessary as $a_k = b_j = a_i$. So, one of the first three cases has to result in a smaller value than case 4. Next, consider the case when $a_i \neq b_j$ and $\min(i−k,j−l) \geq 2$. Suppose that $2 \leq i−k \leq j−l$. The cost of transforming $A[1:i]$ to $B[1:j]$ by using an optimal transformation of $A[1:k−1]$
Table 4  CPU/cache energy in joules for DL distance algorithms on Xeon4

|       | A   | B   | LS_DL | LS_DL2 | Strip_DL | Strip_DL2 | L vs L2 | S vs S2 | L2 vs S2 |
|-------|-----|-----|-------|--------|----------|-----------|---------|---------|---------|
| 40000 | 40000| 107 | 56    | 77     | 47       | 47.3%     | 39.0%   | 17.1%   |
| 80000 | 80000| 384 | 225   | 305    | 187      | 41.5%     | 38.7%   | 16.7%   |
| 120000| 120000| 997 | 509   | 687    | 422      | 49.0%     | 38.5%   | 17.0%   |
| 160000| 160000| 2088| 907   | 1212   | 752      | 56.6%     | 38.0%   | 17.1%   |
| 200000| 200000| 5058| 2040  | 2714   | 1680     | 59.7%     | 38.1%   | 17.6%   |
| 240000| 240000| 6906| 2781  | 3711   | 2302     | 59.7%     | 38.0%   | 17.2%   |
| 280000| 280000| 9000| 3638  | 4852   | 3016     | 59.6%     | 37.9%   | 17.1%   |
| 320000| 320000| 12287| 4619 | 6366   | 3822     | 62.4%     | 40.0%   | 17.3%   |
| 360000| 360000| 14218| 5690  | 7615   | 4712     | 60.0%     | 38.1%   | 17.2%   |
| 400000| 400000| 14218| 5690  | 7615   | 4712     | 60.0%     | 38.1%   | 17.2%   |

We observe that the above refined recurrence for $H$ holds even in the weighted setting provided that $2S \leq I + D \leq 2T$, where $S$, $I$, $D$, and $T$ are, respectively, the cost of a substitution, insertion, deletion, and transposition; the cost of a substitution is $> 0$ when the characters involved are different and 0 when these are the same. This observation follows from the following.

When $a_i = b_j, S = 0$

$$H_{k-1,j-1} + (i-k-1)D + (j-l-1)I \geq H_{k-1,j} + (i-k-1)D + (j-l-1)I \geq H_{k-1,j-1}$$

Fig. 7 Cache misses for DL trace algorithms on Xeon4
### Table 5 Cache misses in millions for DL trace algorithms on Xeon4

| A   | B   | LS_Trace | LS_Trace2 | Strip_Trace | Strip_Trace2 | L vs L2 | S vs S2 | L2 vs S2 |
|-----|-----|----------|-----------|-------------|--------------|--------|--------|---------|
| 40000 | 40000 | 423      | 24        | 3           | 95.2%       | 88.1%  | 86.2%  |
| 80000 | 80000 | 1,970    | 89        | 29          | 12           | 95.5%  | 58.5%  | 86.6%   |
| 120000 | 120000 | 5,100    | 212       | 66          | 25           | 95.8%  | 61.8%  | 88.1%   |
| 160000 | 160000 | 16,350   | 403       | 115         | 44           | 97.5%  | 61.9%  | 89.1%   |
| 200000 | 200000 | 33,998   | 622       | 513         | 72           | 98.2%  | 85.9%  | 88.3%   |
| 240000 | 240000 | 42,252   | 897       | 268         | 101          | 97.9%  | 62.4%  | 88.8%   |
| 280000 | 280000 | 70,370   | 1,244     | 358         | 139          | 98.2%  | 61.3%  | 88.9%   |
| 320000 | 320000 | 91,501   | 1,576     | 453         | 181          | 98.3%  | 60.1%  | 88.5%   |
| 360000 | 360000 | 146,103  | 2,001     | 2,120       | 230          | 98.6%  | 89.1%  | 88.5%   |
| 400000 | 400000 | 221,690  | 2,435     | 6,032       | 276          | 98.9%  | 95.4%  | 88.6%   |

When $2 \leq i - k \leq j - l$

\[
H_{k-1,l-1} + (i - k - 1) D + T + (j - l - 1) I \\
\geq H_{k-1,l-1} + (i - k - 1)(2S - I) + S + (j - l - 1)I \\
= H_{k-1,l-1} + (i - k + 1)S + ((j - l) - (i - k))I \\
\quad + (i - k - 2)S \\
\geq H_{k-1,l-1} + (i - k + 1)S + ((j - l) - (i - k))I \\
\geq H_{i,j}
\]

(6)

The case when $2 \leq j - l \leq i - k$ is symmetric.

The algorithms developed in this paper are based on our refined recurrence for $H$.

### Methods

**DL distance algorithms**

In this section, we develop two algorithms, $LS_{DL2}$ and $Strip_{DL2}$, to compute the DL distance between two strings of length $m$ and $n$ drawn from an alphabet of size $s$. We note that when $s > m + n$, at least $s - m - n$ characters of the alphabet appear in neither $A$ nor $B$. So, these non-appearing characters may be removed from the alphabet and we can work with this reduced size alphabet. Hence, throughout this paper, we assume that $s \leq m + n$. Our algorithms, which take $O(m + n)$ space, are based on the recurrence of Eqs. 3 and 4 and are the counterparts of algorithms $LS_{DL}$ and $Strip_{DL}$ of [18] that are based on the recurrence of Eqs. 1 and 2.

**Algorithm LS_{DL2}**

Like algorithm $LS_{DL}$ of [18], $LS_{DL2}$ (Algorithm 1) computes $H$ by rows from top to bottom and within a row...
Table 6 Run time in hh : mm : ss for DL trace algorithms on Xeon4

| A      | B      | LS_Trace | LS_Trace2 | Strip_Trace | Strip_Trace2 | L vs L2 | S vs S2 | L2 vs S2 |
|--------|--------|----------|-----------|-------------|--------------|---------|---------|---------|
| 40000  | 40000  | 00:00:30 | 00:01:17  | 00:02:26    | 00:00:16     | 44.0%   | 37.8%   | 3.6%    |
| 80000  | 80000  | 00:02:29 | 00:03:44  | 00:02:33    | 00:02:14     | 47.3%   | 36.2%   | 4.0%    |
| 120000 | 120000 | 00:04:42 | 00:06:37  | 00:04:14    | 00:04:16     | 52.8%   | 36.1%   | 4.2%    |
| 160000 | 160000 | 00:08:42 | 00:10:29  | 00:09:31    | 00:10:30     | 58.1%   | 36.0%   | 4.1%    |
| 200000 | 200000 | 00:14:18 | 00:16:54  | 00:12:57    | 00:14:20     | 60.0%   | 36.0%   | 4.1%    |
| 240000 | 240000 | 00:21:08 | 00:26:21  | 00:16:37    | 00:23:15     | 62.1%   | 36.0%   | 4.1%    |
| 280000 | 280000 | 00:28:31 | 00:34:01  | 00:21:23    | 00:29:33     | 65.2%   | 36.0%   | 4.1%    |
| 320000 | 320000 | 00:37:42 | 00:41:11  | 00:26:24    | 00:36:11     | 63.4%   | 35.9%   | 4.2%    |
| 360000 | 360000 | 01:15:14 | 02:27:33  | 01:41:11    | 02:36:24     | 66.4%   | 35.9%   | 4.2%    |

by columns from left to right. For convenience, we augment $H$ with row $-1$ and column $-1$. All values on this row and on this column are $\text{maxVal}$, where $\text{maxVal}$ is a large value. Algorithm $LS\_DL2$ uses 4 one-dimensional arrays last_row_id[$1:s$], $R[-1:n]$, $R[1-1:n]$, and $FR[-1:n]$ and a few simple variables that have the following semantics when we are computing $H_{ij}$.

1. $FR[q] = H_{k-1,q-2}$ for the current $i$ in case $q \geq j$ and for the next $i$ in case $q < j$
2. $R[q] = H_{i,q}$ if $q < j$ and $H_{i-2,q}$ if $q \geq j$
3. $R1[q] = H_{i-1,q}$
4. last_row_id[$c$] = largest $k < i$ such that $A[k] = c$
5. last_col_id = largest $l < j$ such that $B[l] = A[i]$
6. $T$ is the value to use for $H_{i-2,l-1}$ should this be needed in the computation of $H_{ij}$

7. last_i2l1 = $H_{i-2,j-1}$
8. diag ... Case 1 of Eq. 4
9. left ... Case 2 of Eq. 4
10. up ... Case 3 of Eq. 4
11. transpose ... Case 4 of Eq. 4

Lines 2 and 3 of the algorithm initialize $FR$, $R1$, and $R$ so that following the swap of line 6, $R1[q] = H_{0,q}$ and $R[q] = H_{-1,q}$, $-1 \leq q \leq n$. In other words, at the start of iteration $i = 1$ of the for loop of lines 9-30, $R1$ and $R$, respectively, correspond to rows $i - 1$ (i.e., row 0) and $i - 2$ (i.e., row -1) of $H$. At this time, $FR[-1:n] = \text{maxValue}$, which corresponds to the initial situation that $k = \text{lastA}[i][b_j]$ is undefined. This will be updated as $\text{lastA}[i][b_j]$ gets defined. last_row_id[$1:s$] is initialized to $-1$ in line 4 for each character $c$ to indicate the fact that at the start of the $i = 1$ loop, $A[p], p < 1$ isn’t a character of the
Table 7 CPU/cache energy in joules for DL trace algorithms on Xeon4

| A   | B   | LS_Trace | LS_Trace2 | Strip_Trace | Strip_Trace2 | L vs L2 | S vs S2 | L2 vs S2 |
|-----|-----|----------|-----------|-------------|--------------|---------|---------|---------|
| 40000| 40000| 181.4    | 103.1     | 157.0       | 97.8         | 43.2%   | 37.7%   | 5.2%    |
| 80000| 80000| 703.1    | 413.8     | 610.7       | 389.3        | 41.1%   | 36.3%   | 5.9%    |
| 120000| 120000| 1,736.9  | 929.9     | 1,365.3     | 873.4        | 46.5%   | 36.0%   | 6.1%    |
| 160000| 160000| 3,443.8  | 1,644.4   | 2,407.9     | 1,540.3      | 52.3%   | 36.0%   | 6.3%    |
| 200000| 200000| 5,843.6  | 2,556.4   | 3,818.7     | 2,398.1      | 56.3%   | 37.2%   | 6.2%    |
| 240000| 240000| 8,665.3  | 3,659.6   | 5,403.6     | 3,430.8      | 57.8%   | 36.5%   | 6.3%    |
| 280000| 280000| 12,275.0 | 4,970.5   | 7,372.3     | 4,665.3      | 59.5%   | 36.7%   | 6.1%    |
| 320000| 320000| 16,536.9 | 6,486.5   | 9,609.6     | 6,084.6      | 60.8%   | 36.7%   | 6.2%    |
| 360000| 360000| 23,396.4 | 8,224.1   | 12,439.7    | 7,725.9      | 64.8%   | 37.9%   | 6.1%    |
| 400000| 400000| 27,551.9 | 10,229.2  | 15,167.8    | 9,616.6      | 62.9%   | 36.6%   | 6.0%    |

alphabet. last_col_id is set to -1 at the start of each iteration of the loop of lines 5–32 as at the start of this loop, no character of B has been examined and so there is no last seen occurrence of A[i] in B (for this iteration of the for loop of lines 5–32). Also, at the start of the computation of each row of H, last_id211 is set to R[0], because, by the semantics of last_id211, when we are computing H11, last_id211 = H11−2,0 = R[0]. Following this, R[0] is set to i to indicate that the cheapest way to transform A[1 : i] into B[0 : 0] is to do i deletes at a total cost of i (hence, when we are computing H11, in the loop of lines 9–30, R[q] = Hjiq, q < i), and T is set to maxVal as when we start a row computation, i is undefined. So, the initializations establish the variable semantics given above.

In lines 10–12 of the loop of lines 9–30, diag, left, and up are set to the values specified in cases 1–3 of Eq. 4. Note that from the semantics of the variables, R[j] − 1 = Hj−i,j−1, R[j − 1] = Hj−1,j, and R[1] = H1−1,j at this time. Line 13 computes the minimum of the values in the first 3 cases of Eq. 4. If A[i] = B[j] (line 14), then Hji is determined by cases 1–3 and the value of temp computed in line 13 is Hji. At this time, we need to update last_col_id as the most recently seen occurrence of A[i] is now at position j of B. Since A[i] = B[j], lastA[i + 1][j] = i. So, Hk−1,j−2 to use for the next i in case 4 is H11−2 = R1[j − 2]. This value is saved in FR[j] in line 16. Since A[i] = B[j], the value to use for H11−2,j−1 in case 4 of Eq. 4 in future iterations of the for j loop (until, of course, we encounter another j where A[i] = B[j]) becomes H11−2,j−1, which by the variable semantics is last_id211. This value is saved in T in line 17.

When A[i] ≠ B[j], lines 19–26 are executed. From the semantics of last_id and last_col_id, it follows that line 19 correctly sets k and l. Lines 22 and 24, respectively, compute the cost of case 4 when j − l = 1 and i − k = 1, respectively. Note that by the semantics of FR and T, FR[j] = Hk−1,j−2 and T = H11−2,j−1. So, lines 22 and 25

Algorithm 1 Algorithm LS_DL2

1: LS_DL2(A[1 : m], B[1 : n])
2: FR[−1 : n] ← R[−1 : n] ← R[−1] ← maxVal
3: R[j] ← j, 0 ≤ j ≤ n
4: last_row_id[1 : s] ← −1
5: for i ← 1 to m do
6: swap(R, R1)
7: last_col_id ← −1
8: last_id211 ← R[0]; R[0] ← i; T ← maxVal
9: for j ← 1 to n do
10: diag ← R1[j − 1] + c(A[i], B[j])
11: left ← R[j − 1] + 1
12: up ← R1[j] + 1
13: temp ← min(diag, left, up)
14: if A[i] = B[j] then
15: last_col_id ← j // last occurrence of a
16: FR[j] ← R1[j − 1] // save H_k−1,j−2
17: T ← last_id211 // save H11−2,l−1
18: else
19: k = last_row_id[B[j]], l = last_col_id
20: if (j − l) == 1 then
21: transpose ← FR[j] + (i − k)
22: temp ← min(temp, transpose)
23: else if (i − k) == 1 then
24: transpose ← T + (j − l)
25: temp ← min(temp, transpose)
26: end if
27: end if
28: last_id211 ← R[j]
29: R[j] ← temp
30: end for
31: last_row_id[A[i]] ← i
32: end for
33: return R[n]
update \( \text{temp} \) to be \( H_{ij} \). When we reach line 29, regardless of whether \( A[i] = B[j] \) or \( A[i] \neq B[j] \), \( R[j] = H_{i-2,j} \) and \( \text{temp} = H_{ij} \).

In line 28, we set \( \text{last}_j = R[j] = H_{i-2,j} \). While this momentarily upsets the semantics of \( \text{last}_j \), the semantics are restored upon the start of next iteration of the \( \text{for} j \) loop as \( j \) increases by 1 or in line 8 if the \( \text{for} j \) loop terminates and we advance to the next iteration of the \( \text{for} i \) loop. Line 29 sets \( R[j] = H_{ij} \), which similarly upsets the semantics of \( R \) but correctly sets up the semantics for the next iteration. Finally, line 31 correctly updates \( \text{last}_\text{row} \) so as to preserve its semantics for the next iteration of the \( \text{for} i \) loop.

The correctness of \( \text{LS DL2} \) follows from the preceding discussion. The space and time complexities are readily seen to be \( O(m + n + s) = O(m + n) \) and \( O(mn + s) = O(mn) \), respectively. When \( m < n \), the space complexity may be reduced by a constant factor by swapping \( A \) and \( B \). Using the LRU cache model of [18], one may show that \( \text{LS DL2} \) has approximately \( 3mn/w \) cache misses, where \( w \) is the width of a cache line. By comparison, the number of cache misses for \( \text{LS DL} \) is \( mn(1 + 3/w) \).

**Strip algorithm Strip DL2**

As in [18], we can reduce cache misses, which in turn reduces run time, by partitioning \( H \) into \( n/q \) strips of size \( m \times q \), where \( q \) is the largest strip width for which the data needed in the computation of the strip fits into the cache. \( H \) is computed by strips from left to right and the computation of each strip is done using \( \text{LS DL2} \). To enable this computation by strips, one strip needs to pass computed

**Table 8** Run time of parallel DL distance algorithms, in hh : mm : ss, on Xeon4

| A  | B  | PP_LS_DL | PP_LS_DL2 | PP_Strip_DL | PP_Strip_DL2 | L vs L2 | S vs S2 | L2 vs S2 |
|----|----|---------|---------|------------|------------|--------|--------|--------|
| 40000| 40000| 00:00:05 | 00:00:03 | 00:00:03 | 00:00:02 | 42.4% | 38.1% | 33.8% |
| 80000| 80000| 00:00:20 | 00:00:13 | 00:00:13 | 00:00:08 | 41.8% | 38.1% | 33.6% |
| 120000| 120000| 00:00:56 | 00:00:26 | 00:00:28 | 00:00:17 | 52.8% | 38.5% | 33.8% |
| 160000| 160000| 00:01:49 | 00:00:47 | 00:00:50 | 00:00:31 | 57.2% | 38.3% | 33.9% |
| 200000| 200000| 00:02:55 | 00:01:13 | 00:01:19 | 00:00:48 | 58.3% | 38.4% | 33.9% |
| 240000| 240000| 00:04:09 | 00:01:45 | 00:01:53 | 00:01:10 | 57.9% | 38.3% | 33.5% |
| 280000| 280000| 00:05:48 | 00:02:22 | 00:02:34 | 00:01:35 | 59.1% | 38.3% | 33.4% |
| 320000| 320000| 00:07:21 | 00:03:07 | 00:03:20 | 00:02:04 | 57.5% | 38.2% | 33.8% |
| 360000| 360000| 00:10:13 | 00:03:58 | 00:04:24 | 00:02:37 | 61.2% | 40.6% | 34.2% |
| 400000| 400000| 00:11:41 | 00:04:50 | 00:05:13 | 00:03:14 | 58.6% | 38.2% | 33.3% |

**Fig. 10** Run time of parallel DL distance algorithms, in seconds, on Xeon4
values to the next using three additional one-dimensional arrays \(C, C_1, \) and \(FC\) of size \(m\) each. \(C\) records the values of \(H\) computed for the rightmost column in the strip; \(C_1\) records the values of \(H\) computed for the next to rightmost column in the strip; and \(FC\)\[i\] is the value of \(T\) (i.e., \(H[i-2][l-1]\)) at row \(i\), where \(l\) is the last column where \(B[l] = A[i]\) in the strip. We name this new algorithm as \(Strip\_DL\_2\). The space complexity of \(Strip\_DL\_2\) is \(O(m + n)\) and its time complexity is \(O(mn)\).

**DL trace algorithms**

Wagner and Fischer [19] introduced the concept of a trace to describe an edit sequence when the edit operations are limited to insert, delete, and substitute. Lowrance and Wagner [17] extended the concept of a trace to include transpositions. We reproduce here the definition and example used by us in [18]. A trace for the strings \(A = a_1 \cdots a_m\) and \(B = b_1 \cdots b_n\) is a set \(T\) of lines, where the endpoints \(u\) and \(v\) of a line \((u, v)\) denote positions in \(A\) and \(B\), respectively. A set of lines \(T\) is a trace iff:

1. For every \((u, v) \in T\), \(u \leq m\) and \(v \leq n\).
2. The lines in \(T\) have distinct \(A\) positions and distinct \(B\) positions. That is, no two lines in \(T\) have the same \(u\) or the same \(v\).

A line \((u, v)\) is balanced iff \(a_u = b_v\) and two lines \((u_1, v_1)\) and \((u_2, v_2)\) cross iff \((u_1 < u_2)\) and \((v_1 > v_2)\). It is easy to see that \(T = \{(1, 2), (3, 1), (4, 3), (5, 6)\}\) (see Fig. 2) is a trace for the strings \(A = dafac\) and \(B = fdbbec\). Line \((4, 3)\) is not balanced as \(a_4 \neq b_3\). The remaining 3 lines in the trace are balanced. The lines \((1,2)\) and \((3,1)\) cross.

Table 9 Run time of parallel DL trace algorithms, in hh : mm : ss, on Xeon4

| A      | B      | PP_LS_Trace | PP_LS_Trace2 | PP_Strip_Trace | PP_Strip_Trace2 | L vs L2 | S vs S2 | L2 vs S2 |
|--------|--------|-------------|--------------|----------------|----------------|---------|---------|----------|
| 40000  | 40000  | 00:00:10    | 00:00:06     | 00:00:07       | 00:00:05       | 40.8%   | 27.1%   | 10.4%    |
| 80000  | 80000  | 00:00:38    | 00:00:23     | 00:00:27       | 00:00:19       | 39.9%   | 30.7%   | 19.2%    |
| 120000 | 120000 | 00:01:39    | 00:01:51     | 00:00:59       | 00:00:40       | 48.3%   | 33.0%   | 22.6%    |
| 160000 | 160000 | 00:03:05    | 00:01:31     | 00:01:43       | 00:01:09       | 51.0%   | 33.5%   | 24.2%    |
| 200000 | 200000 | 00:05:03    | 00:02:21     | 00:02:43       | 00:01:47       | 53.5%   | 34.2%   | 24.0%    |
| 240000 | 240000 | 00:07:50    | 00:03:23     | 00:03:50       | 00:02:30       | 56.8%   | 34.7%   | 26.0%    |
| 280000 | 280000 | 00:11:11    | 00:04:36     | 00:05:13       | 00:03:26       | 58.8%   | 34.2%   | 25.6%    |
| 320000 | 320000 | 00:15:07    | 00:06:00     | 00:06:46       | 00:04:25       | 60.3%   | 34.8%   | 26.4%    |
| 360000 | 360000 | 00:21:25    | 00:07:37     | 00:08:44       | 00:05:38       | 64.5%   | 35.4%   | 25.9%    |
| 400000 | 400000 | 00:24:10    | 00:09:22     | 01:10:34       | 00:06:55       | 61.3%   | 34.6%   | 26.1%    |
In a trace, unbalanced lines denote a substitution operation and balanced lines denote retaining the character of A. If $a_i$ has no line attached to it, $a_i$ is to be deleted and when $b_j$ has no attached line, it is to be inserted. When two balanced lines $(u_1, v_1)$ and $(u_2, v_2)$, $u_1 < u_2$ cross, $a_{u_1} + 1 \cdots a_{u_2-1}$ are to be deleted from A making $a_{u_1}$ and $a_{u_2}$ adjacent, then $a_{u_1}$ and $a_{u_2}$ are to be transposed, and finally, $b_{v_2} + 1 \cdots b_{v_1-1}$ are to be inserted between the just transposed characters of A.

The edit sequence corresponding to the trace of Fig. 2 is delete $a_2$, transpose $a_1$ and $a_3$, substitute $b$ for $a_4$, insert $b_4 = b$ and $b_5 = e$, retain $a_5$. The cost of this edit sequence is 5.

In [18], we used a divide-and-conquer strategy similar to that used by Hirschberg [20] to determine an optimal trace in $O(mn)$ time and $O(s \min\{m,n\} + n)$ space. In [18], we made a distinction between traces that have a center crossing and those that do not. A trace has a center crossing iff it contains two lines $(u_1, v_1)$ and $(u_2, v_2)$ such that $v_2 \leq n/2$ and $v_1 > n/2$, $u_1 < u_2$, while satisfying (a) $a_i \neq a_{u_1} = b_{v_1}$, $u_1 < i < u_2$ and (b) $b_j \neq b_{v_2} = a_{u_2}$, $v_2 < j < v_1$. In words, $u_1$ is the last (i.e., rightmost)
occurrence of \( b_{v_1} \) in \( A \) that precedes position \( u_2 \) of \( A \) and \( v_2 \) is the last occurrence of \( a_{u_2} \) in \( B \) that precedes position \( v_1 \) of \( B \). (Figure 3).

In [18], we showed that the cost of an optimal trace \( T \) is given by Eq. 7 when \( T \) has no center crossing and by Eq. 8 when \( T \) has a center crossing. Hence, the cost of \( T \) is the smaller of these two costs.

\[
costNoCC(T) = \min_{i \leq j \leq m} \{H[i] + H'[i + 1]\} \tag{7}
\]

where \( H[i] \) is the cost of an optimal trace for \( A[1:i] \) and \( B[1:n/2] \) and \( H'[i + 1] \) that for an optimal trace for \( A[i + 1:m] \) and \( B[n/2 + 1:n] \). For the min, we try \( 1 \leq u_1 < m \) and for each such \( u_1 \), we set \( v_1 \) to be the smallest \( i > n/2 \) for which \( b_j = a_{u_1} \). For each \( u_1 \) we examine all characters other than \( a_{u_1} \) in the alphabet. For each such character \( c \), \( v_2 \) is set to the largest \( j \leq n/2 \) for which \( b_j = c \) and \( u_2 \) is the smallest \( i > u_1 \) for which \( a_i = c \).

Our new algorithms, \( LS\_TRACE2 \) and \( Strip\_TRACE2 \), are based on an adaptation of Eqs. 7 and 8 using Eq. 4.

**Algorithm LS\_TRACE2**

Consider the case when the optimal trace has no center crossing. Let \( R'[] \) be the value of \( R[] \) when \( LS\_DL2(B[1:n/2],A[1:m]) \) terminates and let \( R''[] \) be the value of \( R[] \) when \( LS\_DL2(B[n/2+1,n],A[m+1]) \) terminates. Let \( R', R', R'' \), and \( FR' \) be the corresponding final values for \( R1 \) and \( RF \). From Eq. 7 and \( LD\_DL2 \), we obtain

\[
costNoCC(T) = \min_{1 \leq i \leq m} \{H[i] + H'[i + 1]\}
\]

When \( T \) has a center crossing \((u_1,v_1),(u_2,v_2)\), then it follows from Eq. 4 that either \( u_1 \) and \( u_2 \) are adjacent in \( A \) or \( v_1 \) and \( v_2 \) are adjacent in \( B \) (or both). When \( v_1 \) and \( v_2 \) are adjacent in \( B \), then \( v_2 = \frac{n}{2} \) and \( v_1 = \frac{n}{2} + 1 \). Substituting into Eq. 8, we get

\[
costCC(T) = \min \{H[u_1 - 1] [v_2 - 1] + H'[u_2 + 1] [v_1 + 1] + (u_2 - u_1 - 1) + 1 + (v_1 - v_2 - 1)\}
\]

When \( u_1 \) and \( u_2 \) are adjacent in \( A \), then \( u_2 - u_1 = 1 \), \( v_2 \) is the right most occurrence of \( A[u_2] \) in \( B \) that precedes position \( \frac{n}{2} + 1 \) (i.e., \( v_2 \leq n/2 \)) and \( v_1 \) is the left most occurrence of \( A[u_1] \) in \( B \) after position \( \frac{n}{2} \) (i.e., \( v_1 \geq n/2 + 1 \)). So, we have

\[
costCC(T) = \min \{H[u_1 - 1] [v_2 - 1] + H'[u_2 + 1] [v_1 + 1] + (u_2 - u_1 - 1) + 1 + (v_1 - v_2 - 1)\}
\]

Algorithm \( LS\_TRACE2 \) (Algorithm 2) provides the pseudocode for our linear space computation of an optimal trace. It assumes that \( LS\_DL2 \) has been modified to return the arrays \( R, R1 \) and \( FR \).

**Algorithm 2 Linear space optimal trace**

1. \( LS\_TRACE2(A[1:m],B[1:n]) \)
2. if \( m \leq 1 \) or \( n \leq 1 \) then
3. Do a linear search to find an optimal trace for \( A \) and \( B \)
4. Return optimal trace
5. else
6. \((R,R1,FR) \leftarrow LS\_DL2(B[1:n/2],A[1:m])\)
7. \((R',R1',FR') \leftarrow LS\_DL2(B[n/2+1,n],A[m+1])\)
8. Compute \( costNoCC(T) \) using Eq. 9. Let \( i' \) be the \( i \) that minimizes the cost.
9. Compute \( costCC(T) \) using Eqs. 10 and 11. Let \((u'_1,u'_2)\) and \((v'_1,v'_2)\) be the indexes that minimize the cost.
10. if \( costNoCC(T) \leq costCC(T) \) then
11. \( T1 = LS\_TRACE2(A[1:i'],B[1:n/2])\)
12. \( T2 = LS\_TRACE2(A[i'+1:m],B[n/2+1:n])\)
13. Return \( T1 \cup T2 \)
14. else
15. \( T1 = LS\_TRACE2(A[1:n/2],B[1:v'_2])\)
16. \( T2 = LS\_TRACE2(A[u'_2+1:m],B[v'_1+1:n])\)
17. Return \( T1 \cup T2 \cup ((u'_1,v'_1),(u'_2,v'_2))\)
18. end if
19. end if
20. Using an analysis similar to that used by us in [18] for the analysis of \( DL\_TRACE \), we see that the time complexity of \( DL\_TRACE2 \) is \( O(mn) \). The space required is the same as for \( LS\_DL2 \). The number of cache misses is approximately twice that for \( LS\_DL2 \) when invoked with strings of size \( n \) and \( m \). Hence, the cache miss count for \( LS\_TRACE2 \) is \( \approx 6mn/w \).
| A    | B    | LS_DL  | LS_DL2 | Strip_DL | Strip_DL2 | L vs L2 | S vs S2 | L2 vs S2 |
|------|------|--------|--------|----------|-----------|--------|--------|---------|
| 40000| 40000| 0:00:14| 0:00:05| 0:00:08  | 0:00:05   | 63.2%  | 41.7%  | 9.8%    |
| 80000| 80000| 0:00:55| 0:00:21| 0:00:32  | 0:00:19   | 61.9%  | 41.5%  | 9.9%    |
| 120000| 120000| 0:02:12| 0:00:47| 0:01:13  | 0:00:42   | 64.3%  | 41.5%  | 9.7%    |
| 160000| 160000| 0:05:19| 0:01:24| 0:02:09  | 0:01:15   | 73.7%  | 41.5%  | 9.9%    |
| 200000| 200000| 0:10:16| 0:02:11| 0:03:23  | 0:01:58   | 78.7%  | 41.8%  | 9.9%    |
| 240000| 240000| 0:16:17| 0:03:09| 0:04:50  | 0:02:50   | 80.7%  | 41.5%  | 10.0%   |
| 280000| 280000| 0:24:19| 0:04:17| 0:06:36  | 0:03:51   | 82.4%  | 41.7%  | 10.1%   |
| 320000| 320000| 0:33:32| 0:05:35| 0:08:36  | 0:05:02   | 83.3%  | 41.5%  | 10.0%   |
| 360000| 360000| 0:45:50| 0:07:06| 0:10:58  | 0:06:22   | 84.5%  | 42.0%  | 10.2%   |
| 400000| 400000| 0:56:44| 0:08:44| 0:13:27  | 0:07:52   | 84.3%  | 41.6%  | 10.1%   |

**Strip trace algorithm Strip_TRACE2**

This algorithm differs from LS TRACE2 in that it uses a modified version of Strip_DL2 rather than a modified version of LS_DL2. The modified version of Strip_DL2 returns the arrays C, C1 and FC computed by Strip_DL2. The asymptotic time complexity of Strip_TRACE2 is also O(nm) and it takes the same amount of space as does Strip_DL2. The number of cache misses is approximately twice that for Strip_DL2.

**Platforms and test data**

The single-core algorithms were implemented using C and the multi-core ones using C and OpenMP. The relative performance of these algorithms was measured on the following platforms:

1. Intel Xeon CPU E5-2603 v2 Quad-Core processor 1.8GHz with 10MB cache.
2. Intel I7-x980 Six-Core processor 3.33GHz with 12MB LLC cache.
3. Intel Xeon CPU E5-2695 v2 2x12-Core processors 2.40GHz with 30MB cache.

For convenience, we will, at times, refer to these platforms as Xeon4, Xeon6, and Xeon24 (i.e., the number of cores is appended to the name Xeon).

All codes were compiled using the gcc compiler with the O2 option. On our Xeon4 platform, the benchmarking included a comparison of memory, cache misses, run time, and energy consumption. The cache miss count and the energy consumption was measured using the "perf" [21] software through the RAPL interface. For the Xeon6...
and Xeon24 platforms only the run time was benchmarked.

For test data, we used randomly generated protein sequences as well as real protein sequences obtained from the Protein Data Bank [22] and DNA/RNA/protein sequences from the National Center for Biotechnology Information (NCBI) database [23]. The results for our randomly generated protein sequences were comparable to those for similarly sized sequences used from the two databases [22] and [23]. So, we present only the results for the random data sets here.

**Xeon E5-2603 (Xeon4)**

**DL distance algorithms**

Table 1 gives the memory required to process random protein sequences of length 400,000 using each of the single-core DL scoring algorithms considered in this paper. LS_DL takes 4.75 times the memory taken by LS_DL2 and LS_Strip takes 4.69 times the memory taken by LS_Strip2.

Figure 4 and Table 2 give the number of cache misses on our Xeon4 platform for randomly generated sequences of size between 40,000 and 400,000. The column of Table 2 labeled LvsL2 gives the percentage reduction in cache misses achieved by LS_DL2 relative to LS_DL, that labeled SvsS2 gives this percentage for Strip_DL2 relative to Strip_DL, and that labeled L2vsS2 gives this percentage for Strip_DL2 relative to LS_DL2. Strip_DL2 has the fewest cache misses. Strip_DL2 reduces cache misses by up to 91.9% relative to LS_DL2 and by up to 94.4% relative to Strip_DL.

---

**Table 12** Run time of parallel DL distance algorithms, in hh : mm : ss, on Xeon6

| A      | B      | PP_LS_DL | PP_LS_DL2 | PP_Strip_DL | PP_Strip_DL2 | L vs L2  | S vs S2   | L2 vs S2 |
|--------|--------|----------|-----------|-------------|--------------|----------|-----------|----------|
| 40000  | 40000  | 0:00:03  | 0:00:01   | 0:00:03     | 0:00:01      | 54.6%    | 64.1%     | 24.4%    |
| 80000  | 80000  | 0:00:11  | 0:00:05   | 0:00:06     | 0:00:04      | 54.2%    | 37.4%     | 27.2%    |
| 120000 | 120000 | 0:00:32  | 0:00:11   | 0:00:13     | 0:00:08      | 66.1%    | 39.4%     | 28.1%    |
| 160000 | 160000 | 0:01:18  | 0:00:21   | 0:00:23     | 0:00:14      | 72.8%    | 36.4%     | 32.5%    |
| 200000 | 200000 | 0:02:34  | 0:00:30   | 0:00:36     | 0:00:23      | 80.3%    | 35.9%     | 24.4%    |
| 240000 | 240000 | 0:03:47  | 0:00:44   | 0:00:52     | 0:00:31      | 80.8%    | 39.8%     | 28.5%    |
| 280000 | 280000 | 0:05:25  | 0:01:02   | 0:01:09     | 0:00:43      | 80.9%    | 38.4%     | 31.0%    |
| 320000 | 320000 | 0:06:59  | 0:01:20   | 0:01:32     | 0:00:56      | 80.9%    | 39.2%     | 29.6%    |
| 360000 | 360000 | 0:09:23  | 0:01:39   | 0:01:56     | 0:01:10      | 82.4%    | 39.6%     | 29.2%    |
| 400000 | 400000 | 0:11:03  | 0:02:01   | 0:02:23     | 0:01:29      | 81.7%    | 37.8%     | 26.7%    |

---
Figure 5 and Table 3 give the run times on our Xeon4 platform for our random data set. In the figure, the time is in seconds while in the table, the time is given using the format $hh:mm:ss$. The table also gives the percentage reduction in run time.

As can be seen, on our Xeon4 platform, Strip DL2 is the fastest followed by LS DL2, Strip DL, and LS DL. Strip DL2 reduces run time by up to 15.9% relative to LS DL2 and by up to 40.1% relative to Strip DL.

Figure 6 and Table 4 give the CPU and cache energy consumed, in joules, on our Xeon4 platform. On our data sets, Strip DL2 required up to 17.6% less CPU and cache energy than LS DL2 and up to 40.0% less than Strip DL. It is interesting to note that the energy reduction is comparable to the reduction in run time suggesting a close relationship between run time and energy consumption for this application.

**DL trace algorithms**

Figure 7 and Table 5 give the number of cache misses for the trace algorithms on our Xeon4 platform for randomly generated sequences of size between 40,000 and 400,000. The column of Table 5 labeled L vs L2 gives the percentage reduction in cache misses achieved by LS_TRACE2 relative to LS_TRACE, that labeled S vs S2 gives this percentage Strip_TRACE2 relative to Strip_TRACE, and that labeled L2 vs S2 gives this percentage Strip_TRACE2 relative to LS_TRACE. Strip_TRACE2 has the fewest cache misses.

![Fig. 15 Run time of parallel DL trace algorithms, in seconds, on Xeon6](image-url)
Table 14 Run time in \textit{hh : mm : ss} for DL distance algorithms on Xeon24

| A    | B    | LS_DL | LS_DL2 | Strip_DL | Strip_DL2 | L vs L2 | S vs S2 | L2 vs S2 |
|------|------|-------|--------|----------|-----------|---------|---------|---------|
| 40000| 40000| 0:00:14| 0:00:06| 0:00:10 | 0:00:05  | 58.0%   | 50.6%   | 13.8%   |
| 80000| 80000| 0:00:53| 0:00:22| 0:00:41 | 0:00:20  | 58.1%   | 51.6%   | 11.9%   |
| 120000| 120000| 0:02:01| 0:00:57| 0:01:31 | 0:00:49  | 53.1%   | 46.5%   | 13.5%   |
| 160000| 160000| 0:03:34| 0:01:28| 0:02:42 | 0:01:18  | 58.8%   | 51.9%   | 11.6%   |
| 200000| 200000| 0:05:57| 0:02:19| 0:04:15 | 0:02:01  | 62.1%   | 52.4%   | 12.9%   |
| 240000| 240000| 0:08:05| 0:03:16| 0:06:05 | 0:02:56  | 59.6%   | 51.8%   | 10.4%   |
| 280000| 280000| 0:11:32| 0:04:27| 0:08:20 | 0:03:59  | 61.5%   | 52.1%   | 10.2%   |
| 320000| 320000| 0:15:55| 0:07:24| 0:10:58 | 0:05:13  | 63.1%   | 52.4%   | 10.9%   |
| 360000| 360000| 0:26:41| 0:07:24| 0:15:11 | 0:06:38  | 72.2%   | 56.4%   | 10.5%   |
| 400000| 400000| 0:30:11| 0:09:02| 0:17:15 | 0:08:06  | 70.1%   | 53.0%   | 10.3%   |

misses. \textit{Strip\_Trace2} reduces cache misses by up to 89.1% relative to \textit{LS\_Trace2} and by up to 95.4% relative to \textit{Strip\_Trace}.

Figure 8 and Table 6 give the run times on our Xeon4 platform for our random data set. The table also gives the percentage reduction in run time. \textit{Strip\_Trace2} is the fastest followed by \textit{LS\_Trace2}, \textit{Strip\_Trace}, and \textit{LS\_Trace} (in this order). \textit{Strip\_Trace2} reduces run time by up to 4.3% relative to \textit{LS\_Trace2} and by up to 37.8% relative to \textit{Strip\_Trace}.

Figure 9 and Table 7 give the CPU and cache energy consumed, in joules, \textit{Strip\_Trace2} required up to 6.3% less CPU and cache energy than \textit{LS\_Trace2} and up to 37.9% less than \textit{Strip\_Trace}.

\textbf{Parallel algorithms}

Figure 10 and Table 8 give the run times for our parallel DL distance algorithms on our Xeon4 platform. \textit{PP\_LS\_DL2} is up to 61.2% faster than \textit{PP\_LS\_DL} and \textit{PP\_Strip\_DL2} is up to 40.6% faster than \textit{PP\_Strip\_DL}. Also, \textit{PP\_LS\_DL2} and \textit{PP\_Strip\_DL2} achieve a speedup of up to 3.15 and 3.98 compared to the corresponding single-core algorithms on a four-core machine, respectively.

Figure 11 and Table 9 give the run times for our parallel DL trace algorithms on our Xeon4 platform. \textit{PP\_LS\_Trace2} is up to 64.5% faster than \textit{PP\_LS\_Trace} and \textit{PP\_Strip\_Trace2} is up to 35.4% faster than \textit{PP\_Strip\_Trace}. Also, \textit{PP\_LS\_Trace2} and \textit{PP\_Strip\_Trace2} achieves a speedup up to 2.94 and 3.83 compared to the corresponding single-core algorithms, respectively.

\textbf{17-x980 (Xeon6)}

\textbf{DL distance algorithms}

Figure 12 and Table 10 give the run times of our single-core distance algorithms on our Xeon6 platform. As can
be seen, Strip\textsubscript{DL}2 is the fastest followed by LS\textsubscript{DL}2, Strip\textsubscript{DL}, and LS\textsubscript{DL} (in this order). Strip\textsubscript{DL}2 reduces run time by up to 10.2% relative to LS\textsubscript{DL}2 and by up to 42.0% relative to Strip\textsubscript{DL}.

**DL trace algorithms**

Figure 13 and Table 11 give the run times of our single-core trace algorithms on our Xeon6 platform. Strip\textsubscript{Trace2} is the fastest followed by LS\textsubscript{Trace2}, Strip\textsubscript{Trace}, and LS\textsubscript{Trace} (in this order). Strip\textsubscript{Trace2} reduces run time by up to 3.9% relative to LS\textsubscript{Trace2} and by up to 37.6% relative to Strip\textsubscript{Trace}.

**Parallel algorithms**

Figure 14 and Table 12 give the run times for our parallel DL distance algorithms on our Xeon6 platform. PP\textsubscript{LS\_DL}2 is up to 82.4% faster than PP\textsubscript{LS\_DL} and PP\textsubscript{Strip\_DL}2 is up to 39.8% faster than PP\textsubscript{Strip\_DL}. Also, PP\textsubscript{LS\_DL}2 and PP\textsubscript{Strip\_DL}2 achieves a speedup up to 4.32 and 5.44 compared to the corresponding single core algorithms on a six core machine, respectively.

Figure 15 and Table 13 give the run times for our parallel DL trace algorithms on our Xeon6 platform. PP\textsubscript{LS\_Trace2} is up to 74.7% faster than PP\textsubscript{LS\_Trace} and PP\textsubscript{Strip\_Trace2} is up to 41.8% faster than PP\textsubscript{Strip\_Trace}. Also, PP\textsubscript{LS\_Trace2} and PP\textsubscript{Strip\_Trace2} achieves a speedup up to 4.32 and 5.30 compared to the corresponding single-core algorithms, respectively.

**Xeon E5-2695 (Xeon24)**

**DL distance algorithms**

Figure 16 and Table 14 give the run times of our single-core distance algorithms on our Xeon24 platform. Strip\textsubscript{DL}2 is the fastest followed by LS\textsubscript{DL2}, Strip\textsubscript{DL},

### Table 15 Run time in hh:mm:ss for DL trace algorithms on Xeon24

| A    | B    | LS\_Trace | LS\_Trace2 | Strip\_Trace | Strip\_Trace2 | L vs L2 | S vs S2 | L2 vs S2 |
|------|------|-----------|-----------|-------------|--------------|--------|--------|---------|
| 40000| 40000| 00:00:25  | 00:00:12  | 00:01:21    | 00:00:11     | 52.7%  | 49.7%  | 9.4%    |
| 80000| 80000| 00:01:36  | 00:00:40  | 00:01:23    | 00:00:37     | 58.5%  | 55.4%  | 7.1%    |
| 12000| 12000| 00:03:38  | 00:01:28  | 00:03:04    | 00:01:22     | 59.5%  | 55.5%  | 7.0%    |
| 16000| 16000| 00:06:26  | 00:02:36  | 00:05:27    | 00:02:26     | 59.5%  | 55.4%  | 6.6%    |
| 20000| 20000| 00:10:26  | 00:04:08  | 00:08:54    | 00:03:47     | 60.3%  | 57.4%  | 8.4%    |
| 24000| 24000| 00:14:44  | 00:05:51  | 00:12:14    | 00:05:30     | 60.3%  | 55.0%  | 6.0%    |
| 28000| 28000| 00:20:22  | 00:07:57  | 00:16:39    | 00:07:28     | 61.0%  | 55.2%  | 6.1%    |
| 32000| 32000| 00:27:39  | 00:10:26  | 00:21:51    | 00:09:48     | 62.3%  | 55.1%  | 6.1%    |
| 36000| 36000| 00:41:56  | 00:13:10  | 00:28:55    | 00:12:21     | 68.6%  | 57.3%  | 6.2%    |
| 40000| 40000| 00:50:01  | 00:16:42  | 00:34:16    | 00:15:43     | 66.6%  | 54.1%  | 5.9%    |
and $LS\_DL$ (in this order). $Strip\_DL2$ reduces run time by up to 13.8% relative to $LS\_DL2$ and by up to 56.4% relative to $Strip\_DL$.

**DL trace algorithms**

Figure 17 and Table 15 give the run times of our single-core trace algorithms on our Xeon24 platform. $Strip\_Trace2$ is the fastest followed by $LS\_Trace2$, $Strip\_Trace$, and $LS\_Trace$ (in this order). $Strip\_Trace2$ reduces run time by up to 9.4% relative to $LS\_Trace2$ and by up to 57.4% relative to $Strip\_Trace$.

**Parallel algorithms**

Figure 18 and Table 16 give the run times for our parallel DL distance algorithms on our Xeon24 platform. $PP\_LS\_DL2$ is up to 68.7% faster than $PP\_LS\_DL$ and $PP\_Strip\_DL2$ is up to 54.6% faster than $PP\_Strip\_DL$. Also, $PP\_LS\_DL2$ and $PP\_Strip\_DL2$ achieves a speedup up to 11.2 and 21.36 compared to the corresponding single core algorithms on a twelve-four core machine, respectively.

Figure 19 and Table 17 give the run times for our parallel DL trace algorithms on our Xeon24 platform. $PP\_LS\_Trace2$ is up to 58.3% faster than $PP\_LS\_Trace$ and $PP\_Strip\_Trace2$ is up to 59.3% faster than $PP\_Strip\_Trace$. Also, $PP\_LS\_Trace2$ and $PP\_Strip\_Trace2$ achieves a speedup up to 9.65 and 16.4 compared to the corresponding single-core algorithms, respectively.

### Table 16 Run time of parallel DL distance algorithms, in $hh:mm:ss$, on Xeon24

| A       | B       | PP\_LS\_DL | PP\_LS\_DL2 | PP\_Strip\_DL | PP\_Strip\_DL2 | L vs L2 | S vs S2 | L2 vs S2 |
|---------|---------|------------|-------------|---------------|----------------|--------|--------|---------|
| 400000  | 400000  | 00:00:01   | 00:00:01    | 00:00:01      | 00:00:00       | 30.9%  | 48.6%  | 41.6%   |
| 80000   | 80000   | 00:00:04   | 00:00:02    | 00:00:02      | 00:00:01       | 41.8%  | 47.2%  | 51.4%   |
| 120000  | 120000  | 00:00:08   | 00:00:06    | 00:00:04      | 00:00:02       | 30.2%  | 47.4%  | 57.4%   |
| 160000  | 160000  | 00:00:18   | 00:00:09    | 00:00:08      | 00:00:04       | 48.1%  | 54.6%  | 60.3%   |
| 200000  | 200000  | 00:00:28   | 00:00:13    | 00:00:11      | 00:00:06       | 53.8%  | 47.6%  | 54.2%   |
| 240000  | 240000  | 00:00:37   | 00:00:19    | 00:00:16      | 00:00:09       | 48.4%  | 47.3%  | 54.8%   |
| 280000  | 280000  | 00:00:59   | 00:00:24    | 00:00:23      | 00:00:12       | 59.8%  | 47.2%  | 50.0%   |
| 320000  | 320000  | 00:01:33   | 00:00:32    | 00:00:30      | 00:00:15       | 65.0%  | 50.3%  | 54.8%   |
| 360000  | 360000  | 00:02:01   | 00:00:40    | 00:00:40      | 00:00:19       | 66.9%  | 53.1%  | 53.0%   |
| 400000  | 400000  | 00:02:44   | 00:00:51    | 00:00:45      | 00:00:23       | 68.7%  | 48.7%  | 55.3%   |
Discussion
We have developed linear space algorithms to compute the DL distance between two strings and also to determine an optimal trace (edit sequence). Besides using less space than the space efficient algorithms of [18], these algorithms are also faster. Significant run-time improvement (relative to known algorithms) was seen for our new algorithms on all three of our experimental platforms.

Conclusion
On all platforms, the linear-space cache-efficient algorithms Strip_DL2 and Strip_TRACE2 were the best-performing single-core algorithms to determine the DL distance and optimal trace, respectively. Strip_DL2 reduced run time by as much as 56.4% relative to the Strip_DL and Strip_TRACE2 reduced run time by as much as 57.4% relative to the Strip_Trace. Our multi-core algorithms reduced the run time by up to 59.3% compared to the best previously known multi-core algorithms.

The linear space string correction algorithm developed in this paper requires $2S \leq I + D \leq 2T$, where $S$, $I$, $D$ and $T$ are, respectively, the cost of a substitution, insertion, deletion and transposition. As noted earlier, this requirement is met by the DL distance as for this metric, $S = I = D = T = 1$. When only $I + D \leq 2T$ is satisfied, the more general algorithm ([18]) that runs in $O(mn)$ time and uses $O(s \ast \min(m,n) + m + n)$ space, where $s$ is the size of alphabet comprising the strings, may be used.

Table 17 Run time of parallel DL trace algorithms, in hh:mm:ss, on Xeon24

| A     | B     | PP_LS_Trace | PP_LS_Trace2 | PP_Strip_Trace | PP_Strip_Trace2 | L vs L2 | S vs S2 | L2 vs S2 |
|-------|-------|-------------|--------------|----------------|----------------|---------|---------|---------|
| 40000 | 40000 | 0:00:03     | 0:00:02      | 0:00:02        | 0:00:01        | 38.0%   | 59.3%   | 53.3%   |
| 80000 | 80000 | 0:00:11     | 0:00:06      | 0:00:07        | 0:00:04        | 40.3%   | 49.2%   | 42.5%   |
| 120000| 120000| 0:00:21     | 0:00:12      | 0:00:15        | 0:00:09        | 42.3%   | 44.1%   | 29.2%   |
| 160000| 160000| 0:00:35     | 0:00:19      | 0:00:24        | 0:00:12        | 45.8%   | 50.2%   | 36.5%   |
| 200000| 200000| 0:00:53     | 0:00:26      | 0:00:41        | 0:00:22        | 50.6%   | 45.2%   | 14.2%   |
| 240000| 240000| 0:01:13     | 0:00:36      | 0:00:53        | 0:00:26        | 50.4%   | 51.7%   | 29.4%   |
| 280000| 280000| 0:01:52     | 0:00:51      | 0:01:10        | 0:00:30        | 54.4%   | 56.7%   | 40.4%   |
| 320000| 320000| 0:02:30     | 0:01:09      | 0:01:26        | 0:00:41        | 54.0%   | 52.3%   | 40.6%   |
| 360000| 360000| 0:03:20     | 0:01:23      | 0:01:40        | 0:00:48        | 58.2%   | 51.4%   | 41.9%   |
| 400000| 400000| 0:04:14     | 0:01:46      | 0:02:08        | 0:00:58        | 58.3%   | 55.1%   | 45.5%   |
Abbreviations
CPU: Central Processing Unit; DL: Damerau-Levenshtein; DNA: Deoxyribonucleic acid; LLC: Last level cache; LRU: Least recently used; NCBI: National Center for Biotechnology Information; PDB: Protein Data Bank; RNA: RiboNucleic acid

Acknowledgments
The authors also would like to thank the anonymous reviewers for their valuable comments and suggestions.

About this supplement
This article has been published as part of BMC Bioinformatics Volume 21 Supplement 1: 2020: Selected articles from the 8th IEEE International Conference on Computational Advances in Bio and medical Sciences (ICCABS 2018): bioinformatics. The full contents of the supplement are available online at https://bmcbioinformatics.biomedcentral.com/articles/supplements/volume-21-supplement-1.

Authors’ contributions
CZ and SS developed the new linear space cache efficient DL distance algorithms, did theoretical analysis and the experimental results analysis, and wrote the manuscript. CZ programmed the algorithms and ran the benchmark tests. All authors have read and approved the final manuscript.

Funding
Publication of this supplement was funded, in part, by the National Science Foundation under award NSF 1447711. Publication costs were funded by the University of Florida.

Availability of data and materials
Data sharing is not applicable to this article as all test data are randomly generated protein sequences.

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Received: 20 October 2019   Accepted: 30 October 2019
Published: 9 December 2020

References
1. Brill E, Moore RC. An improved error model for noisy channel spelling correction. In: Proceedings of the 38th Annual Meeting on Association for Computational Linguistics. Stroudsburg: Association for Computational Linguistics; 2000. p. 286–93.
2. Bard GV. Spelling-error tolerant, order-independent pass-phrases via the Damerau-Levenshtein string-edit distance metric. In: Proceedings of the Fifth Australasian Symposium on ACSW Frontiers - Volume 68. ACSW '07. Darlinghurst: Australian Computer Society, Inc.; 2007. p. 117–24.
3. Li M, Zhang Y, Zhu M, Zhou M. Exploring distributional similarity based models for query spelling correction. In: Proceedings of the 21st International Conference on Computational Linguistics and the 44th Annual Meeting of the Association for Computational Linguistics. ACL-44. Stroudsburg: Association for Computational Linguistics; 2006. p. 1025–32.
4. Cai X, Zhang KC, Joshi B, Johnson A. Touching from a distance: Website fingerprinting attacks and defenses. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. New York: ACM; 2012. p. 605–16.
5. Faloutsos C, Megaloukokomou V. On data mining, compression, and kolmogorov complexity. Data Min Knowl Discov. 2007;15(1):3–20.
6. Majecki KA, Dunin-Horkawicz S, Steczkiwicz K, Muszewska A, Novotny M, Ginalska K, Bujnicki JM. The RNase H-like superfamily: new members, comparative structural analysis and evolutionary classification. In: Nucleic Acids Research, vol. 42. Oxford, United Kingdom: Oxford University Press; 2014. p. 4160–4179.
7. Bischof J, Ibrahim SM. bcRep: R package for comprehensive analysis of B cell receptor repertoire data. PLoS ONE. 2016;11(8):1–15.
8. Pomorova O, Savenko O, Lysenko S, Nicheporuk A. Information and Communication Technologies in Education, Research, and Industrial Applications. In: 12th Internal conference. Kyiv: Springer; 2016. http://ceur-ws.org/Vol-1614/.
9. Biswas AK, Gao JX. PROSS2Clust: patched ma-seq read segments’ structure-oriented clustering. J Bioinforma Comput Biol. 2016;14(05):1650027.
10. Pop PG. In-memory dedicated dot-plot analysis for DNA repeats detection. In: Intelligent Computer Communication and Processing (ICCP), 2016 IEEE 12th International Conference On. Washington, DC: IEEE; 2016. p. 317–20.
11. Gabrys R, Yaakobi E, Milenkovic O. Codes in the damerau distance for deletion and adjacent transposition correction. IEEE Trans Inf Theory. 2018;64(4):2550–70.
12. Largest Protein Sequence: Titin. https://en.wikipedia.org/wiki/Titin. Accessed 15 Feb 2019.
13. Largest Chromosome. https://en.wikipedia.org/wiki/Chromosome_1. Accessed 115 Feb 2019.
14. Levenshtein VI. Binary codes capable of correcting deletions, insertions, and reversals. Sov Phys Dokl. 1966;10(8):707–10.
15. Robinson DJS. An Introduction to Abstract Algebra. Berlin: Walter de Gruyter; 2003, pp. 255–7.
16. Jaro MA. Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida. J Am Stat Assoc. 1989;84(406):414–20.
17. Lowrance R, Wagner RA. An extension of the string-to-string correction problem. J ACM. 1975;22(2):177–83.
18. Zhao C, Sahni S. String correction using the damerau-levenshtein distance. BMC Bioinforma. 20:277. https://doi.org/10.1186/s12859-019-2819-0.
19. Wagner RA, Fischer MJ. The string-to-string correction problem. J ACM. 1974;21(1):168–73.
20. Hirschberg DS. A linear space algorithm for computing longest common subsequences. Commun ACM. 1975;18(6):341–3.
21. Perf Tool. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed 15 Feb 2019.
22. Protein Data Bank. http://www.rcsb.org/pdb/home/home.do. Accessed 15 Feb 2019.
23. NCBI Database. http://www.ncbi.nlm.nih.gov/gquery. Accessed 15 Feb 2019.

Publisher’s Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.