FastLSA – A Fast Linear-Space Algorithm for Sequence Alignment

K. Charter[1], A. Driga[1], P. Lu[1], J. Schaeffer[1], D. Szafron[1] and I. Parsons[2]

[1]Department of Computing Science University of Alberta, Edmonton, AB, Canada, T6G 2E8 and [2]BioTools Inc., Ironwood Professional Centre, 800, 10050 - 112 Street, Edmonton, AB, Canada T5K 2J1

Abstract

For two DNA or protein sequences of length m and n, dynamic programming alignment algorithms like Needleman-Wunsch and Smith-Waterman take $O(m \times n)$ time and use $O(m \times n)$ space, so we refer to them as full matrix (FM) algorithms. This space requirement means that large sequences will not completely fit in main memory. It also means that shorter sequences will not completely reside in cache memory. Hirschberg's algorithm reduces the space requirements to $O(\min(m, n))$, but requires approximately twice the number of operations required by the FM algorithms. This paper presents the *FastLSA* algorithm that is adaptive to the amount of space available. It allows a user to trade space for operations. At one extreme, it uses linear space with approximately 1.5 times the number of operations required by the FM algorithms. At the other extreme, it uses quadratic space with no extra operations compared to the FM algorithms. However, our experiments show that in practice, due to memory caching effects, *FastLSA* and Hirschberg's algorithm are often faster than FM. This is true even though the number of operations is higher and even when there is enough main memory to hold the entire dynamic programming matrix of the FM algorithm. *FastLSA* has been incorporated into the newest and fastest commercialproducts like BioTools' ChromaTool. This paper also briefly describes how to parallelize the *FastLSA* algorithm to further improve its performance.

Introduction

Sequence alignment is one of the fundamental operations in bioinformatics since it is an essential step in two different molecular biology problems: determining the biological functions of proteins and determining the evolutionary relationship between organisms. The *primary structure* of a protein consists of a sequence of amino acids, each represented by one of 20 different letters of the alphabet. At a lower level, each amino acid is a triplet of three neucleotides represented by one of the starting letters of the names of the four neucleotides: Adenine (A), Cytosine (C), Guanine (G) and Thymine (T).

One of the consequences of this representation is that sequence alignment can be done at the neucleotide level using an alphabet of 4 characters or at the protein level, using an alphabet of 20 with a sequence 1/3 as long. Although there are $4^3 = 64$ different triplets, multiple different triplets correspond to the same amino acid so there are only 20 valid amino acids. In addition, one triplet also corresponds to the start codon and 3 triplets correspond to stop codons. The reduction from 64 different triplets to the 20 naturally occurring triplets is one reason for doing sequence matching at the amino acid level instead of at the neucleotide level. A second reason is that there are known similarities between the 20 different amino acids. Nevertheless, sequence alignment can be done either at the neucleotide level or the amino acid level, using the same sequence alignment algorithms.

To align two protein sequences, say TLDKLLKD and TDVLKAD, the sequences can be shifted right or left to align as many identical letters as possible. For example, the maximum number of identical letters (i.e. 3) can be obtained using the shift:

```
TLDKLLKD
-TDVLKAD
```

where gaps are denoted by '-'. However, by allowing gaps to be inserted into the middle of sequences, we can often obtain more identical letters (i.e. 5):

```
TLDKLLK-D    TLDKLLK-D
T-DVL-KAD    T-D-VLKAD
```

The insertion of a gap is not an abstract mathematical operation. It corresponds to specific common biological events: an *insertion mutation* in which an extra neucleotide is inserted into one sequence or a *deletion mutation* in which a neucleotide is deleted. Of course if we are matching at the protein level, a gap corresponds to the insertion or deletion of 3 neucleotides. The matching of non-identical letters also corresponds to a biological event: a *point mutation* in which a single neucleotide is replaced by another. However, some mutations have a larger biological effect than others. These effects can be more readily understood be viewing the changes at the amino acid level instead of at the neucleotide level.

Consider a *silent mutation* that causes the triplet TCT to change to the triplet TCG. Since both of these triplets code for the same amino acid (S - serine), these two triplets will have an identity match during an amino acid sequence alignment. Now consider a *neutral mutation* that causes the triplet AAA to change to the triplet AGA. This change is more significant since these triplets code for the different amino acids, (K – lysine) and (R – arginine) respectively. However, we know that these two amino acids have similar function so in sequence alignment we would like to indicate that the letters K and R are a better match than the amino acids (K – lysine) and (C – cysteine) which have very different functional properties. To accommodate such similarity matches, we do not simply count the number of identical characters when we are aligning sequences. Instead, we create a *scoring function* based on the numeric entries of a *similarity table*. For each pair of letters, the table gives a similarity score, where higher values indicate higher similarity between amino acids. The *score* of an alignment is obtained by iterating over all pairs of corresponding letters in the aligned sequences and adding up the entries in the similarity table that is indexed by each pair. An *optimal alignment* is an alignment with the highest score for a given scoring function (it might not be unique).

The similarity table for the scoring function used in this paper is given in Figure 1. This similarity table is based on the popular Dayhoff scoring matrix (MDM78 Mutation Data Matrix - 1978) [DBH1983]. The Dayoff matrix is the log-odds form of the PAM-250 (Percent Accepted Mutations) mutation matrix, where 1 PAM means there has been 1 mutation per 100 residues. 250 PAM means there has been 250 mutations per 100 residues or 2.5 mutations per residue. The PAM concept was developed by M.O. Dayhoff in the 1960's to measure the evolutionary pressure that had been placed on a protein sequence. The similarity table of Figure 1 is the default one used in the BioTools' commercial product PepTool (www.biotools.com). It has been scaled so that each entry is a non-negative integer. The table is symmetric so only the lower half is shown.

-	Name	Code	A	B	C	D	E	F	G	H	I	K	L	M	N	O	P	Q	R	S	T	V	W	X	Y	Z
A	Alanine	`GC* (*=any)`	16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
B		`N or D`	0	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
C	Cysteine	`TGT TGC`	0	0	36	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
D	Aspartic acid	`GAT GAC`	0	20	0	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
E	Glutamic acid	`GAA GAG`	8	10	0	12	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
F	Phenylalanine	`TTC TTT`	0	0	0	0	0	28	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
G	Glycine	`GG*`	0	0	0	0	0	0	16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
H	Histidine	`CAT CAC`	0	6	0	0	0	0	0	24	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
I	Isoleucine	`ATT ATC ATA`	0	0	0	0	0	12	0	0	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
K	Lysine	`AAA AAG`	0	12	0	0	0	0	0	0	0	20	-	-	-	-	-	-	-	-	-	-	-	-	-	-
L	Leucine	`TTA TTG CT*`	0	0	0	0	0	14	0	0	10	0	20	-	-	-	-	-	-	-	-	-	-	-	-	
M	Methionine	`ATG`	0	0	0	0	0	6	0	0	8	0	16	24	-	-	-	-	-	-	-	-	-	-	-	-
N	Asparagine	`AAT AAC`	0	20	0	14	10	0	0	6	0	12	0	0	20	-	-	-	-	-	-	-	-	-	-	-
O		`K`	0	12	0	0	0	0	0	0	0	20	0	0	12	20	-	-	-	-	-	-	-	-	-	-
P	Proline	`CC*`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	20	-	-	-	-	-	-	-	-	-
Q	Glutamine	`CAA CAG`	0	12	0	6	14	0	0	6	0	6	0	0	12	6	0	20	-	-	-	-	-	-	-	-
R	Arginine	`CG* AGA AGG`	0	0	0	0	0	0	0	10	0	16	0	0	0	16	0	10	20	-	-	-	-	-	-	-
S	Serine	`TC* AGT AGC`	10	8	0	0	0	0	0	0	0	0	0	0	8	0	0	0	0	20	-	-	-	-	-	-
T	Threonine	`AC*`	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	20	-	-	-	-	-
V	Valine	`GT*`	0	0	0	0	0	0	0	0	16	0	12	6	0	0	0	0	0	0	0	20	-	-	-	-
W	Tryptophane	`TGG`	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	6	0	0	0	36	-	-	-
X		`Anything`	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	20	-	-
Y	Tyrosisne	`TAT TAC`	0	0	0	0	0	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	6	28	-
Z		`E or Q`	8	12	0	6	20	0	0	6	0	6	0	0	12	6	0	20	10	0	0	0	0	6	0	20

Figure 1. The similarity scoring table for this paper (higher scores denote higher similarity).

If an amino acid in one sequence lines up with a gap in the other sequence, then a negative value, called a gap penalty is added to the score. Since insertion of multiple gaps into a particular location is common, it is more realistic to penalize subsequent gaps in the same location by a lower value. Formula 1 describes this more realistic gap penalty. If the *extensionPenalty* of Formula 1 is non-zero, then the gap penalty is said to be *affine*. In the experiments described in this paper we use an affine gap penalty given by Formula 1, where the *initialPenalty = 20* and the *extensionPenalty =10*.

Formula 1: $gapPenalty = initialPenalty + extensionPenalty \times gapSize$

Many algorithms for sequence alignment are based on dynamic programming techniques that are equivalent to the algorithms proposed by Needleman and Wunsch [NW1970] and Smith and Waterman [SW1981]. Aligning two sequences of length m and n is equivalent to finding the maximum cost path through a *dynamic program matrix* (DPM) of size $m+1$ by $n+1$, where an extra row and column is added to capture leading gaps. Given a DPM of size m by n, it takes $O(m \times n)$ time to compute the DPM cost entries, and then $O(m+n)$ time to identify the maximum-cost path in the DPM. In this paper, algorithms that are based on storing the complete matrix are called *full matrix algorithms* (FM).

Unfortunately, calculations requiring $O(m \times n)$ space can be prohibitive. For instance, aligning two sequences with 10,000 letters each requires 400Mbytes of memory, assuming each DPM entry is a single 4 byte integer. Given that we now have the capacity to sequence entire genomes, pairwise sequence comparisons involving up to four million neucleotides at a time are now desirable. $O(m \times n)$ storage of this magnitude would require $O(10^{13})$ Mbytes of memory which is beyond the range of current technology. If the storage requirements for sequence alignment could be substantially reduced, then genome-scale comparisons could be made more feasible.

Hirschberg was the first to report a way of doing the computation using linear space [Hir1975]. However, not storing the entire DPM means that some of the entries need to be recomputed to find the optimal path. It is a classic space-time tradeoff: the number of operations approximately doubles, but the space overhead drops from quadratic to linear in the length of the sequences. In fact, Hirschberg's original algorithm was designed to compute the longest common sub-strings of two strings, but Myers and Miller [MM1988] applied it to sequence alignment.

In summary, there are two extremes for pairwise optimal sequence alignment:

1. full matrix, which minimizes the computational complexity, and
2. linear space, which minimizes the storage requirements.

Neither of these algorithms accommodates the real-world situation where you have more memory than needed for a linear space algorithm, but not enough to do a full matrix computation.

This paper introduces the *FastLSA* (Fast Linear-Space Alignment) algorithm. Linear-space alignment algorithms, such as Hirschberg's algorithm, do not take advantage of additional memory that might be available. Hirschbergs' algorithm achieves its linear space by recursively dividing one sequence in half, computing the corresponding optimal division point for the other sequence, and then aligning each pair of sub-sequences. *FastLSA* recursively subdivides each sequence into $k \geq 2$ pieces, and uses storage that is bounded by $k \times (m+n)$, *FastLSA* uses the additional storage to reduce the execution time. Hirschberg's algorithm re-computes approximately $m \times n$ values, while a FM algorithm has no re-computations. However, *FastLSA* with $k = 2$ re-computes only half as many values as Hirschberg's algorithm since both sequences are bisected instead of one. Higher values of k give additional savings. In the limit, where $k = min(m,n)$, *FastLSA* does no re-computations and reduces to a FM algorithm. The experimental results closely mirror the analytical results, showing that *FastLSA* out-performs Hirschberg's algorithm. In practice, *FastLSA* and Hirschberg's algorithm are often faster than the FM algorithms since the reduced space requirements result in more computations being performed in cache rather than in main memory. In summary, *FastLSA* achieves its faster running time by generalizing Hirschberg's algorithm in two important ways:

1. It subdivides the both sequences instead of only one.
2. It subdivides into k parts instead of just 2 parts.

This paper presents the *FastLSA* algorithm and analyses its space and time requirements. The paper then describes the results of some experiments that compare *FastLSA* to a FM algorithm and Hirschberg's algorithm. Next, a parallel version of *FastLSA* is described. Finally, some experimental results are provided for an implementation of the parallel version.

Dynamic Programming Algorithms

FastLSA is a dynamic programming algorithm like the FM algorithms and Hirschberg's algorithm, and it produces exactly the same optimal alignment for a given scoring function. The algorithms differ only in the time and space required.

The example sequences from the introduction can be used to illustrate the differences between these algorithms. The scoring function uses the scoring table of Figure 1 and a gap penalty defined by Formula 1. However, for simplicity of presentation, this section uses *initialPenalty =10* and *extensionPenalty = 0*, so the gaps are not really affine. In the experimental results section of this paper, true affine gaps are used, with *initialPenalty = 20* and *extensionPenalty = 10*, since they produce more biologically accurate alignments. Consider the sequences:

```
TLDKLLKD
TDVLKAD
```

The alignment:

```
TLDKLLK-D
T-D-VLKAD
```

Yields an optimal score of: ScoringTable[T,T] + gap + ST[D,D] + gap + ST[L,V] + ST[L,L] + ST[K,K] + gap + ST[D,D] = 20 + (-10) + 20 + (-10) + 12 + 20 + 20 + (-10) + 20 = 82. How is this optimal alignment obtained?

One sequence is placed along the top of the matrix and the other sequence is placed along the left side and a gap is added to the start of each sequence. Each different path from the top left corner to the bottom right corner of the matrix that goes only right, down or diagonal, represents a different alignment. Before discussing how to compute the optimal alignment, it is useful to understand how a path corresponds to an alignment. Figure 2 illustrates an alignment.

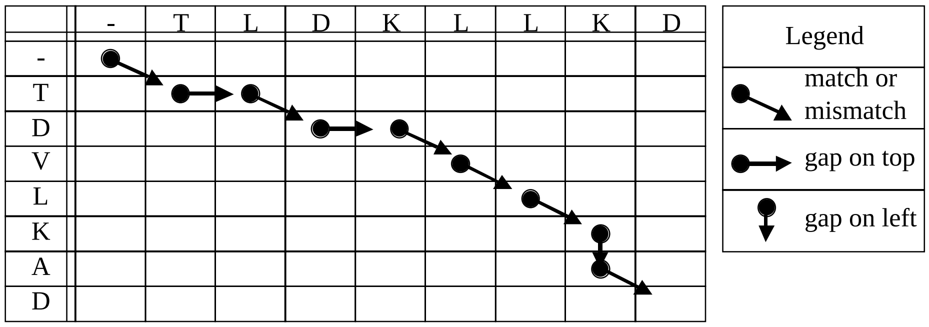


Figure 2. An alignment path.

To translate a matrix path to an alignment, follow the path from the top left to the bottom right. Every diagonal move corresponds to aligning two letters as either a match or a mismatch. For example, Figure 2 begins with the diagonal move: 1) (-\-) to (T\T). This move corresponds to the partial alignment:

```
T
T
```

A right move corresponds to the insertion of a gap in the vertical sequence. For example the right move: 2) (T\T) to (T\L) results in the partial alignment:

```
TL
T-
```

A down move corresponds to the insertion of a gap in the horizontal sequence. For example, after the diagonal move: 3) (T\L) to (D\D), the right move: 4) (D\D) to (D\K), and the three diagonal moves: 5) (D\K) to (V\L), 6) (V\L) to (L\L) and 7) (L\L) to (K\K), the down move: 8) (K\K) to (A\K) results in the partial alignment:

```
TLDKLLK-
T-D-VLKA
```

The final diagonal move 9) (A\K) to (D\D), results in the final complete alignment:

```
TLDKLLK-D
T-D-VLKAD
```

Note that move 5) (D\K) to (V\L) corresponded to a mismatch, instead of a match. Any path can be translated to an alignment, but to obtain the optimal alignment for a given scoring function, we need to identify the corresponding *optimal path*. To derive the optimal path in the matrix, each of the three algorithms can be divided into two phases, which we call *FindScore* and *FindPath*. Figure 3 shows the DPM scores for the example sequences that are computed during the *FindScore* phase. The entries with numerical subscripts form the optimal path, that is computed in the *FindPath* phase.

	-	T	L	D	K	L	L	K	D
-	0_{10}	-10	-20	-30	-40	-50	-60	-70	-80
T	-10	20_9	10_8	0	-10	-20	-30	-40	-50
D	-20	10	20	30_7	20_6	10	0	-10	-20
V	-30	0	22	20	30	32_5	22	12	2
L	-40	-10	20	22	20	50	52_4	42	32
K	-50	-20	10	20	42	40	50	72_3	62
A	-60	-30	0	10	32	42	40	62_2	72_A
D	-70	-40	-10	20	22	32	42	52_L	82_1

Figure 3. A dynamic programming matrix using the similarity table from Figure 1 and non-affine gap penalties from Formula 1 with *initialPenalty = 0* and *extensionPenalty = 10.*

In the *FindScore* phase, a 0 is placed in the upper-left corner of the matrix. Each algorithm propagates scores from the upper-left corner of the matrix to the lower-right corner of the matrix. The score that ends up in the lower-right corner is the optimal alignment score. The score of any entry is the maximum of the three scores that can be propagated from the entry on its left, the entry above it and the entry above-left. A diagonal move corresponds to a match or mismatch alignment and adds the scoring table value for the two letters being considered. A down (right) move corresponds to inserting a gap in the horizontal (vertical) sequence and adds a gap penalty. For example, the score of 20_9 in the (T\T) entry near the top left corner is the maximum of the scores from its left entry(-10 + -10 = -20), above entry (-10 + -10 = -20) and above-left entry (0 + Similarity[T, T] = 0 + 20 = 20). The score of 10_8 in the (T\L) entry is the maximum of the scores from its left entry (20 + -10 = 10), its above entry (-20 + -10 = -30) and its above-left entry (-10 + Similarity[T,L] = -10 + 0 = -10).

The FM algorithms, Hirschberg's algorithm and *FastLSA* all compute the score of the alignment in the same way. However, the FM algorithms store all of the matrix entries $(m+1)\times(n+1)$, while the other two algorithms propagate a single row of scores (m entries) as the matrix is computed, overwriting an old row of scores by a new row of scores.

In Figure 2, a path was read from the top left to the bottom right. However, the *FindPath* algorithm actually computes the optimal paths backwards. For FM algorithms, the *FindPath* phase is straightforward. Since the FM algorithms store all scores in the DPM, they can compute the path by starting at the lower right corner and computing which of the three entries (left, up and diagonal) was used to compute its score. For example, the lower right (D\D) entry is 82_1. Since its upper-left entry (A\K) has a score of 62_2 and since (62 + Similarity[D,D] = 62 + 20 = 82), an optimal path goes through its upper-left (A\K) entry. In addition, an optimal path cannot lead to its above entry (A\D) with value 72_A since $72 - 10 = 62 \neq 82$. Similarly, an optimal path cannot lead to the left entry (D\K) whose value is 52_L. Note that in general it is possible for more than one path to be optimal. However, in our example, there is a single optimal path and it is denoted by numerical subscripts as shown in Figure 3.

In the FM algorithms, the optimal path is easy to compute since the entire dynamic programming matrix is stored. However, neither Hirschberg's algorithm nor *FastLSA* stores the entire dynamic scoring matrix so the computation of the path is more complicated. In both cases, some of the DPM entries must be recomputed to find the path.

Leading, Trailing and Affine Gaps

The scoring function we described in the previous section is useful for understanding the basic concepts of sequence alignment. However, in practice, changes are necessary to make sequence alignment algorithms more representative of real biological phenomena.

Sometimes, a short sequence is aligned against a much longer sequence with the intent of matching a sub-sequence of the longer sequence. In this case, the alignment should consist of a series of leading gaps, a matching region (which might contain a few gaps) and then a series of trailing gaps. In order to find the correct match, the leading and trailing gaps should incur no gap penalties in the alignment. If gap penalties are charged for these leading and trailing gaps, the wrong alignment will be found. For example, consider the following alignment. For simplicity, neucleotides are used instead of amino acids and the simple scoring function uses 2 for a neucleotide match, 0 for a mis-match and -1 for a gap penalty. The desired alignment is:

```
AGATCTGATCGTAAGTCATTCGCATAATGCGT
----------GTACG-C---------------
```

Since there are 26 gaps, 5 matches and 1 mismatch, the score for this alignment is: $-1\times26 + 2\times5 + 0\times1 = -16$. This is the intuitive "best" alignment in which the second sequence is a subsequence of the first, with only a "few" gaps. However, the optimal alignment with respect to the scoring function is:

```
AGATCTGATCGTAAGTCATTCGCATAATGCGT
----------GTA---C----GC---------
```

since this alignment has a higher score: $-1\times26 + 2\times6 + 0\times0 = -14$. The solution to this problem is to modify the scoring function to assign leading and trailing blanks a score of zero. Now the first alignment given above has the optimal alignment. The new scoring function counts 25 leading and trailing gaps, one imbedded gap, 5 matches and 1 mismatch for a score of: $0\times25 + -1\times1 + 2\times5 + 0\times1 = 9$. The second alignment now has a lower score of: $0\times19 + -1\times7 + 2\times6 + 0\times0 = 5$. Note that the goal is not to remove all gaps in the sub-sequence, just to find the most reasonable gapped sub-sequence.

Zero-score leading gap penalties can be implemented by simply initializing the top row and left column of the DPM to zeros. The simplest way to implement trailing gap penalties of zero, is to check whether the score that is being updated is in the right column or bottom row and, if so, to use a zero gap penalty. However, this check should be factored out of the inner double loop to improve performance.

Finally, for more biologically accurate alignments, affine gaps must be supported as described in Formula 1. To support affine gaps, each entry in the DPM must be generalized to include more information than just the score. To see why this is necessary, consider a simple scoring table that assigns 3 points for a match, 0 points for a mismatch, an initial gap penalty of -3 and a gap extension penalty of -1. This makes the score for a new gap in any location, –4, and the score for each subsequent consecutive gap, –1. Figure 4 shows a DPM where four scores are maintained: a diagonal score that comes from the upper left, a vertical score that comes from above, a horizontal score that comes from the left and a max score that is the maximum of the three.

Note that in the lower right square (entry K\L), if only the maximum score was maintained in each entry then the vertical score would be the maximum entry in the above square (L\L) minus a new vertical gap penalty, 3 – (3 + 1) = -1. The correct answer should be the (non-maximum) vertical entry in the above square minus a vertical gap extension penalty, 2 – 1 = 1. Since 1 is a higher score than –1, we need this score to compute the correct maximum value in the lower right square.

	-	T	L
-	0 0	0 0	0 0
	0 0	0 0	0 0
T	0 0	3 -4	0 -4
	0 0	-4 3	-1 0
L	0 0	0 -1	6 -4
	0 0	-4 0	-4 6
L	0 0	0 -2	3 2
	0 0	-4 0	-4 3
K	0 0	0 -3	0 **1**
	0 0	-4 0	-4 **1**

Legend

diagonal entry	vertical entry
horizontal entry	maximum entry

Figure 4. Extra information must be maintained to correctly support affine gaps.

Even though the naïve approach is to replace each score by a triple of scores plus a maximum, this approach is much too expensive. Instead, it can be proven[1] that in addition to the maximum score, only one extra entry per DPM entry is needed. This extra entry contains the difference between the maximum score and vertical entry. If this difference exceeds the initial gap penalty, it can be assigned the initial gap penalty since it will never be large enough to be used in the computation of another entry. For this reason, if the initial gap penalty is smaller than 128, only a single extra byte is needed in each DPM entry. Note that a similar approach can be used to store the horizontal score, but since the DPM entries are processed row by row, the horizontal entry is only used to compute the next DPM entry. Therefore, the horizontal entry can

[1] The proof is omitted from this paper for the sake of brevity.

be kept in a temporary variable instead of stored in the DPM. This means that affine gaps can be supported by increasing the size of DPM entries from 4 bytes to 5 bytes, instead of from 4 bytes to 16 bytes.

Hirschberg’s Algorithm

Hirschberg’s algorithm uses a divide-and-conquer approach. It splits one of the two sequences in half (size *n/2*) and performs the *FindScore* computation on each half against the other original sequence (size *m*). However, the half-sequences are aligned from opposite ends.

After the two *FindScore* computations are complete, the dynamic programming matrix looks like Figure 5. The top half of this matrix is computed from the top row to the middle row, going left to right on each row. The bottom half of this matrix is computed from the bottom row to the middle row, going right to left on each row. The algorithm does not store the entire dynamic programming matrix in memory. Instead only one row in each half matrix is stored and this row is updated as the computation continues. In essence, we are using a *virtual* or *logical* dynamic programming matrix without storing it.

After the two half alignments are complete, only the middle two rows of the matrix are known. This computation determines the optimal split of the full sequence against the two half sequences. The split point is the location that maximizes the sum of the corresponding pairs of scores from the two half alignments. In Figure 5, these sums are: ($-30_1 + 20_1 = -10$), ($0_2 + 30_2 = 30$), ($22_3 + 20_3 = 42$), ($20_4 + 30_4 = 50$), ($30_5 + 40_5 = 70$), ($32_6 + 50_6 = 82$), ($22_7 + 20_7 = 42$), ($12_8 + -10_8 = 2$) and $2_9 + -40_9 = -38$). The maximum score is 82 so the optimal split point is between 32_6 and 50_6 (letters L and L). Therefore, the horizontal sequence has TLDKL as the first partial sequence and LKD as the second.

	-	T	L	D	K	L	L	K	D	-
-	0	-10	-20	-30	-40	-50	-60	-70	-80	
T	-10	20	10	0	-10	-20	-30	-40	-50	
D	-20	10	20	30	20	10	0	-10	-20	
V	-30_1	0_2	22_3	20_4	30_5	32_6	22_7	12_8	2_9	
L		20_1	30_2	20_3	30_4	40_5	50_6	20_7	-10_8	-40_9
K		-10	0	10	20	10	20	30	0	-30
A		-40	-30	-20	-10	0	10	20	10	-20
D		-50	-40	-30	-20	-10	0	10	20	-10
-		-80	-70	-60	-50	-40	-30	-20	-10	0

Figure 5. Hirschberg’s algorithm dynamic programming matrix.

Since Hirschberg’s algorithm does not store the dynamic programming matrix, no path can be directly determined. However, the optimal path can be found by solving two simpler problems, the alignment of:

```
TLDKL
TDV
```

and the alignment of:

```
LKD
LKAD
```

and putting the results together. Hirschberg's algorithm is called recursively to solve these two simpler problems. One of the sequences in each simpler problem has size *n/2*. The other sequence in each simpler problem is approximately *m/2*, depending on where the split occurred. Since the dynamic programming matrix is not stored, parts of it will need to be re-computed. Figure 6 shows a schematic picture of the recursion and the parts of the matrix that must be recomputed. The labels represent different calls to Hirschberg's algorithm. An *a* denotes a recursive call on the first portion of a sequence and a *b* denotes a recursive call on the second portion. A shaded area represents a region that bounds a portion of the optimal path and a clear area represents a region that contains no points in the optimal path. The call to *a*, computes *FindScore* for *aa*, then *FindScore* for *ab*, makes a recursive call to *aa* and then a recursive call to *ab*. Since *aa* makes other recursive calls before returning, this makes the order of the recursive calls lexicographic: (*a*, *aa*, *aaa*, …, *aab*, …, *ab*, …, *b*, *ba*, …, *bb*, …).

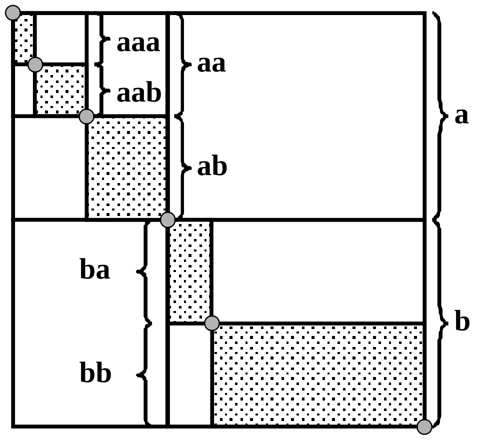


Figure 6. Recursive re-computations in Hirschberg's algorithm.

The recursion terminates when the size of the sub-problems is one, although it could be terminated sooner by using a FM algorithm when the problem size is small enough to solve in memory or in cache. Approximately *m*×*n* re-computations need to be done using Hirschberg's algorithm[MM1988].

The *FastLSA* Algorithm

The basic idea of *FastLSA* [CSS2000] is to use more available memory to reduce the number of re-computations that need to be done in Hirschberg's algorithm. This is accomplished by: 1) dividing both sequences instead of just one, 2) dividing each sequence into *k* parts instead of only two and 3) storing some specific rows and columns of the logical dynamic programming matrix (DPM) in grid lines to reduce the re-computations later.

We begin with a description of the *FastLSA* algorithm for *k = 2*, where the logical DPM is divided into four quadrants. The *FastLSA* algorithm is given in Figure 7 and a trace of the algorithm is shown in Figure 8. As shown in line 1 of Figure 7, *FastLSA* is a recursive algorithm that has two arguments. The *Grid* contains initial scores for the top row and left column of its logical DPM. The *Bounds* describes the rectangular region in the logical DPM that should be computed. *FastLSA* returns a *Path* through this target region. Figure 8a shows a *Grid*, consisting of the top row and left column of the entire logical DPM. The Grid is created and its top row and left column are filled, before the initial call to *FastLSA*. The values are based on gap penalties, such as the top row and left column of the DPM shown in Figure 3. The *Bounds* that are used for the initial call to *FastLSA* represent the entire logical DPM.

```
1.  Path FastLSA(Bounds bounds, Grid grid)
2.    if (fitsInBuffer(bounds))
3.      return FullMatrix(bounds, grid);
4.    newGrid = allocateNewGrid(bounds, grid);
5.    FindScore(bounds, grid, newGrid);
6.    newBounds = lowerRightQuadrant(bounds);
7.    path = FastLSA(newBounds, newGrid);
8.    while (!intesectsUpperLeft(bounds, path))
9.    {
10.     subBounds = pruneBounds(bounds, path);
11.     newPath = FastLSA(subBounds, newGrid);
12.     path = concatenatePaths(path, newPath);
13.   }
14.   deallocateGrid(newGrid);
15.   return path;
16. END FastLSA;
```

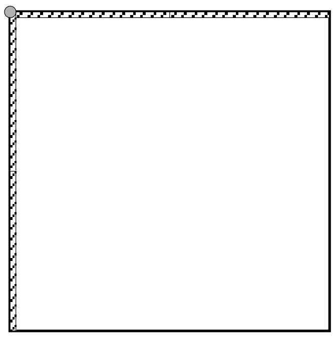
Figure 7. The *FastLSA* algorithm.

The *FastLSA* recursion terminates when the target region of the logical DPM is small enough to fit in a specific memory buffer in physical memory. In this base case, a FM algorithm is used to compute the optimal path through the target region as shown in lines 2 and 3 of Figure 7.

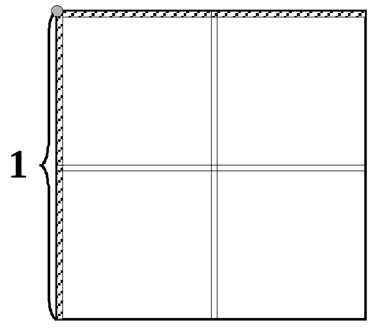
In the non-base case, *FastLSA* allocates a new *Grid* to store some of the scores that will be computed during the *FindScore* phase of the algorithm. It then copies the argument *Grid* values into a new *Grid* as shown in line 4. This situation is illustrated in Figure 8b. The argument *Grid* is shown as the shaded top row and left column of the DPM. The new *Grid* is shown as a non-shaded middle row and middle column of the DPM. Space has been allocated for the new *Grid* and the values have been initialized to the values from the old *Grid*. However, the proper scores reflecting the middle row and column of the DPM have not yet been computed.

The *FindScore* phase of *FastLSA* is similar to the *FindScore* phase of the FM and Hirschberg's algorithm. As shown in Figure 3, the *FindScore* phase of the FM algorithm computes scores and stores them in a physical DPM. In Hirschberg's algorithm, one row vector of scores is propagated for each of two target regions, as shown in Figure 6. The *FindScore* phase of *FastLSA* is a compromise between these approaches. As the scores are propagated through the logical DPM, only scores that lie on the *Grid* are stored. The other difference is that *FindScore* is only applied to three of the four quadrants of the logical DPM. Line 5 shows the call to *FindScore* and Figure 8c shows the result. *FindScore* has been applied to the three quadrants with dotted texture (1A, 1B and 1C) and the new *Grid* contains the generated scores for the middle row and column.

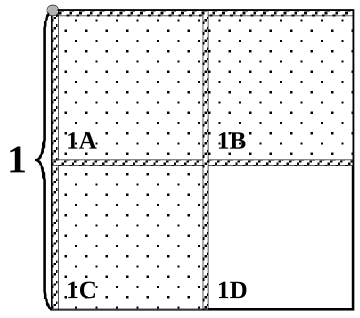
Line 6 constructs a new *Bounds* which represents the lower right quadrant, 1D of Figure 8c. The new *Grid* and new *Bounds* are used as arguments to the second call of *FastLSA* on line 7. Figure 8d shows the situation in this second call to *FastLSA*, after line 5 has been executed, so that the call to *FindScore* has already been completed. The dotted texture in three of the quadrants (2A, 2B and 2C) shows that scores have been computed once in these quadrants, but not stored. The shading of the Grid lines denotes scores that have been computed and stored.



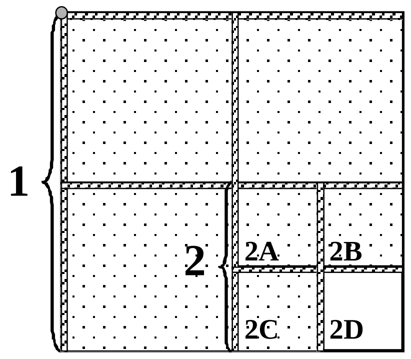
a) Filling the initial grid lines before the first call to *FastLSA*.



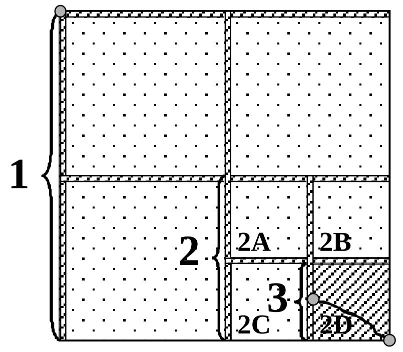
b) First call to *FastLSA* after line 4.



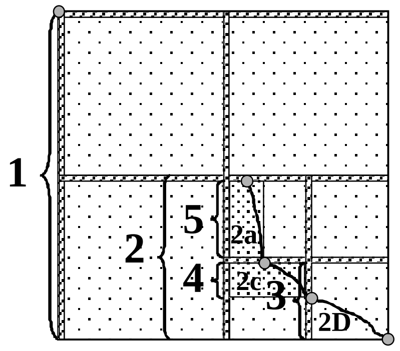
c) After *FindScore* fills grid on line 5.



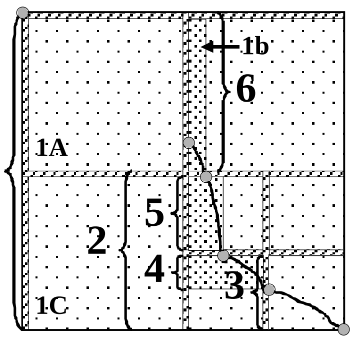
d) The second call to *FastLSA* line 5 of Figure 7.



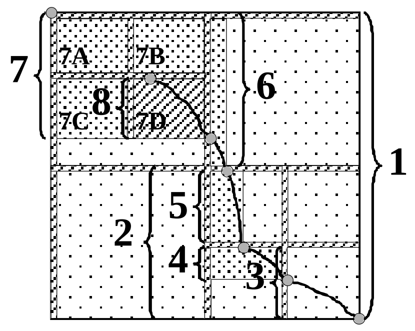
e) Base case call uses FM



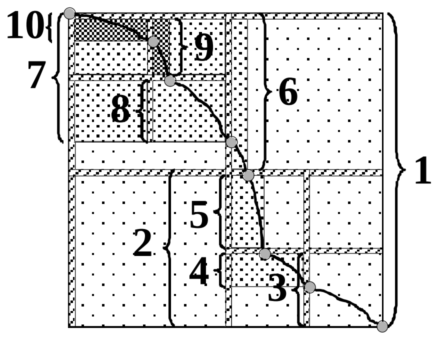
f) Unwinding *FastLSA* after



g) A non-base call to *FastLSA*



h) Final calls to *FastLSA*



i) The *FastLSA* call graph

Figure 8 Tracing the *FastLSA* algorithm.

A new *Bounds* is created on line 6, that represents quadrant 2D. A third call is then made to *FastLSA* on line 7. We assume that for this call, the termination condition holds, so a FM algorithm is called to return a *Path*. This *Path* is shown in Figure 8e. The FM algorithm has its own *FindScore* phase in which scores are computed for the entire quadrant (2D) and saved as they are computed. Therefore this region is shaded to show that the scores are both computed and stored, like the *Grid* lines.

After the third call to *FastLSA* returns on line 7, the while loop on lines 8-13 is responsible for extending the returned *Path* backwards from quadrant 2D, through quadrants 2A, 2B and 2C of Figure 8e. There are actually three possibilities, depending on the location of the upper left endpoint of the *Path* in 2D. If this endpoint is on the corner of 2D, then the path must only go through 2A. If this endpoint is on the left side of 2D (but not the corner), then the path must go through 2C and possibly through 2A, but not through 2B. If this endpoint is on the top side of 2D (but not the corner), then the path must go through 2B and possibly through 2A, but not through 2C. In Figure 8e, the endpoint is on the left side of 2D. Note that we do not have to call *FastLSA* with the entire quadrant 2C as the bounds. Instead, we can prune the part of 2C that is below the y coordinate of the endpoint as shown in Figure 8f. This pruned sub-bounds is computed in line 10. Figure 8f shows the while loop of Figure 7 being executed twice so there are two more calls to *FastLSA*, the fourth call on region 2c and the fifth call on region 2a. We have used the notation 2c for the sub-region of 2C to indicate that it is not the entire quadrant. The while loop of Figure 7 terminates when a returned *Path* of a recursive call to *FastLSA* intersects the *Bounds* of the calling copy of *FastLSA*. In Figure 8f, the fifth call to *FastLSA* returns a *Path* that intersects the bounds of the second *FastLSA* call, which is its caller so the while loop terminates.

Figure 9 shows the complete call graph for the trace of *FastLSA* illustrated in Figure 8. For example, at the stage depicted by Figure 8f, call 1 made call 2 at line 7. Then call 2 made three more calls that already returned: call 3 (at line 7), call 4 (at line 11) and call 5 (at line 11). The condition of the while loop at line 8 is now false, so call 2 de-allocated its grid lines and returned the path shown in Figure 8f. At this point of the trace, quadrant 2D is now shown in dotted texture, instead of shading. This represents the fact that it is no longer stored in memory. We assume that call 4 on region 2c and call 5 on region 2a were both base cases that used a FM algorithm, so they did not make recursive calls to *FastLSA*. However, these regions are drawn with a slightly darker dotted texture in Figure 8f to show that they have been re-computed once. They are the only regions so far that have been computed more than once.

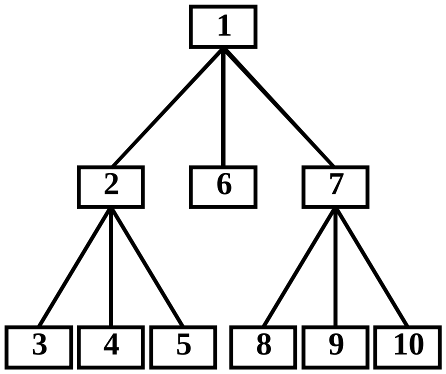


Figure 9. A call graph for *FastLSA*. All leaf nodes call a FM algorithm.

Figure 8g shows the situation where call 1 reaches line 11 and makes a recursive call (call 6) to *FastLSA*. We assume that call 6 used the base case and returned a *Path*. Region 1b is shown in slightly darker dotted texture to illustrate that its scores have been computed twice, just like regions 2c and 2a.

Figure 8h shows how call 1 makes another recursive call (call 7) at line 11 of the *FastLSA* algorithm, but this time the region is too large to use the base case. Instead, *FindScore* is called on three of the quadrants (7A, 7B and 7C) and the slightly darker dotted texture is used to show that they have been computed a second time. Next, a recursive call (call 8) is made on quadrant 7D that uses the base case of the algorithm.

Figure 8i shows the final two calls to *FastLSA* (calls 9 and 10). These are made at line 11 of the algorithm from call 1. Note that regions 9 and 10 use very dark dotted texture. This indicates that these regions have actually been computed three times. In summary, Figure 9 shows that the leaf nodes: 3, 4, 5, 6, 8, 9, 10 use a FM algorithm. Each non-leaf node (1, 2 and 7) makes recursive calls to *FastLSA*.

FastLSA does fewer re-computations than Hirschberg's algorithm at the cost of extra storage. This tradeoff can be further exploited by dividing the matrix into *k* divisions vertically and horizontally instead of only 2 divisions. Figure 10 shows the recursive nature of *FastLSA* for $k = 4$. Call 7 is expanded to show one more level of the recursion. The order of calls that are shown is 1 to 7, then 7.1 to 7.6. The calls 1-6 and 7.1-7.6 may either use a FM algorithm or make recursive calls of their own. Only the shaded regions are re-computed. Region 1 is shown with cross-hatches since it is special. Each of the shaded regions is re-computed exactly once if it uses a FM algorithm. If a shaded region uses a recursive call to *FastLSA*, portions of it will not be re-computed and other portions may be re-computed more than once, as illustrated previously in Figure 8. If a FM algorithm can be applied to region 1, it is computed only once, with no re-computations. Otherwise, portions of it are computed once with no re-computations and other portions have re-computations.

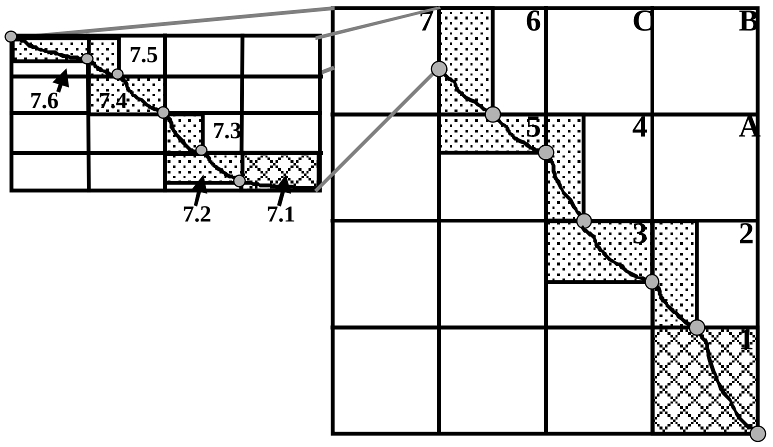


Figure 10. The *FastLSA* algorithm for k = 4.

There is one more generalization that can be made to the *FastLSA* algorithm. Instead of dividing each sequence into the same number of parts, *k*, one sequence can be divided into *k* parts and the other sequence can be divided into a different number of parts, *l*. This approach is especially important when aligning two sequences of very different lengths.

Space Analysis for *FastLSA*

The FM algorithms use $m \times n$ space. Hirschberg's algorithm only uses $2 \times max(m,n)$ space. For *FastLSA*, we will assume that the DPM is divided into *k* parts on each axis. The row and column caches for the first invocation occupy $(k-1) \times (m+n)$ space since there are $(k-1)$ rows of length *n* and $(k-1)$ columns of length *m*. *FastLSA* needs additional storage when it is called recursively for the lower-right block which has size $(n/k) \times (m/k)$. Each time a recursive call is made, the extra storage required is divided by *k* for both the rows and the columns. After recursing *q* times, *FastLSA* will require $(n/k^q) \times (m/k^q)$ storage which will be less than the amount of memory, *L*, allocated for calling a FM algorithm so the recursion will terminate. If we define: $u = m/k^q$, $v = n/k^q$, then the amount of memory for the last call to FM is: $S(u,v) = u \times v < L$.

The total memory required can be expressed by a recurrence relation:

Formula 1: $S(m,n,k) = (k-1) \times (m+n) + S(m/k,n/k,k)$

We can solve this recurrence relation:

$$\begin{aligned} S(m,n,k) &= (k-1)\times(u+v)\times k^q + S(u\times k^{q-1},v\times k^{q-1},k) \\ &= (k-1)\times(u+v)\times k^q + (k-1)\times(u+v)\times k^{q-1} + S(u\times k^{q-2},v\times k^{q-2},k) \\ &= (k-1)\times(u+v)\times\sum_{i=1}^{q} k^i + S(u\times k^0,v\times k^0,k) \\ &= (k-1)\times(u+v)\times\sum_{i=1}^{q} k^i + u\times v \\ &= (k-1)\times(u+v)\times k\times (k^q-1)/(k-1) + u\times v \\ &= (u+v)\times k\times(k^q-1) + u\times v \end{aligned}$$

The memory needed for the recursion (without the call to FM) is:

Formula 2: $S(m,n,k) - u\times v = (u+v)\times k\times(k^q-1)$

For $m,n >> 1$, we have $k^q >> 1$ so the amount of memory needed for the recursion is:

Formula 3: $S(m,n,k) - u\times v \approx (u+v)\times k\times k^q = k\times(m+n)$

Time Analysis for *FastLSA*

The FM algorithms, Hirschberg’s algorithm and *FastLSA* must compute all matrix values at least once. The difference between the algorithms is in the space required and the cost of finding the path. The FM algorithms do not have to re-compute any values to find the path since the dynamic programming scores are all available in memory. They only need to access $O(m+n)$ values. On average, Hirschberg’s algorithm re-computes the entire matrix once [MM1988]. Hence the cost of using linear storage is the additional work that is done, O($m\times n$). By using additional storage, *FastLSA* falls in between the extremes of the FM algorithms with zero additional values recomputed, and Hirschberg, with O($m\times n$)values recomputed.

For a given k, there are k^2 regions and *FastLSA* must first compute boundary information for all regions except the lower right one. The number of operations to compute these boundaries is:

Formula 4: $B(m,n,k) = m\times n – m\times n/k^2 = m\times n(k^2-1)/k^2$

Another way to view this expression is that there are (k^2-1) regions with $m\times n/k^2$ operations on each region.

FastLSA must then perform re-computations on some regions by making recursive calls on them. Let $r(k)$ be the number of regions that require recursive calls. At least one call must be made since the lower right region remains. However, the solution path starts in the lower right corner and must reach the upper left corner. This means that the minimum number of recursive calls is actually k, and this situation occurs when the solution path follows the diagonal. For example, this would be the path [1, 3, 5, 7] in the right side of Figure 10. Although the solution path can move off the diagonal, it can never go down or to the right. This means that the maximum number of regions that require recursive calls is $2\times k-1$. For example, this would correspond to paths: [1, 2, 3, 4, 5, 6, 7] or [1, 2, A, B, C, 6, 7] in the right side of Figure 10. Therefore, we have bounds on the number of regions that require re-computations:

Formula 5: $k \le r(k) \le 2\times k-1$

If we let $T(m,n,k)$ be the number of computations required for our initial matrix of size m by n that is sub-divided into k by k regions then:

Formula 6: $T(m,n,k) = m\times n(k^2-1)/k^2 + r(k)\times R(m/k,n/k,k)$

where $R(m/k,n/k,k)$ is the number of re-computations that must be performed on each of the $r(k)$ regions that include the solution path. This formula is almost a recurrence relation. However, there are two important issues. First, $r(k)$ may vary on different recursive calls to *FastLSA* since it depends on the solution path. For example, in Figure 10, the first level of recursion on the right side has $r(k) = 7$, since the path is *1, 2, 3, 4, 5, 6, 7*. However, the second level of recursion on the left side has $r(k) = 6$, since the path is *7.1, 7.2, 7.3, 7.4, 7.5, 7.6*.

The second issue is more subtle. The number of initial operations at the highest level is $m\times n(k^2-1)/k^2$. This expression is based on computing boundary information for all (k^2-1) regions of size m/k by n/k except the lower right region. However, in general, at the next level of recursion, the size of each region will be less than m/k^2 by n/k^2, as illustrated on the left side of Figure 10. To turn Formula 6 into a recurrence relation, we can do two things. Firstly, we can overestimate the size of each region to be m/k^2 by n/k^2, so that the number of operations at each level is: $R(m/k,n/k,k) = m\times n(k^2-1)/k^2$. Secondly, we must regard $r(k)$ as a recursion-level independent estimator for the number of regions that must be recomputed. If we do this, we obtain a recurrence relation that bounds the number of operations that must be performed in *FastLSA*:

Formula 7: $T(m,n,k) = m\times n(k^2-1)/k^2 + r(k)\times T(m/k,n/k,k)$

To solve this recurrence relation, we recall that after recursing q times, *FastLSA* will require $(n/k^q)\times(m/k^q)$ storage which will be less than the amount of memory, L, allocated for calling an FM algorithm and the recursion will terminate. If we again define: $u = m/k^q$, $v = n/k^q$, then the amount of time (number of operations) for the last call to the FM algorithm is: $T(u,v,k) = u\times v < L$. We can solve the recurrence relation:

$$\begin{aligned}
T(m,n,k) &= m\times n(k^2-1)/k^2 + r(k)\times T(m/k,n/k,k)\\
&= [(k^2-1)/k^2]\times u\times v\times k^{2q} + r(k)\times T(u\times k^{q-1},v\times k^{q-1},k)\\
&= [(k^2-1)/k^2]\times u\times v\times k^{2q} + r(k)\{[(k^2-1)/k^2]\times u\times v\times k^{2q-2}+r(k)\times T(u\times k^{q-2},v\times k^{q-2},k)\}\\
&= [(k^2-1)/k^2]\times u\times v\times\sum_{i=1}^{q}\left[r(k)^{q-i}k^{2i}\right] + T(u\times k^0,v\times k^0,k)\\
&= [(k^2-1)/k^2]\times u\times v\times r(k)^q \sum_{i=1}^{q}\left[\frac{k^2}{r(k)}\right]^i + T(u,v,k)\\
&= [(k^2-1)/k^2]\times u\times v\times r(k)^q[k^2/r(k)]\{[k^2/r(k)]^q-1\}/[k^2/r(k)-1] + u\times v\\
&= (k^2-1)\times u\times v\times[k^{2q}-r(k)^q]/[k^2-r(k)] + u\times v\\
&= m\times n\times(k^2-1)\times[1-r(k)^q/k^{2q}]/[k^2-r(k)] + u\times v
\end{aligned}$$

The solution is:

Formula 8: $T(m,n,k) = m \times n \times (k^2-1) \times [1-r(k)^q/k^{2q}]/[k^2-r(k)] + u \times v$

From Formula 5, in the worst case, $r(k)=2k-1$, so Formula 8 becomes:

$$\begin{aligned} T_w(m,n,k) &= m \times n \times (k^2-1) \times [1-(2k-1)^q/k^{2q}]/[k^2-(2k-1)] + u \times v \\ &= m \times n \times (k-1) \times (k+1) \times [1-(2k-1)^q/k^{2q}]/[(k-1)^2] + u \times v \\ &= m \times n \times (k+1) \times [1-(2k-1)^q/k^{2q}]/[(k-1)] + u \times v \\ &\approx m \times n \times (k+1)/(k-1) \end{aligned}$$

since for $m,n >> 1$, we have $k^q >> 1$, so $[(2k-1)^q/k^{2q}] << 1$ and $u \times v << m \times n$. The worst case number of re-computations, $R_w(m,n,k)$, is:

Formula 10: $R_w(m,n,k) = T_w(m,n,k) – m \times n = m \times n \times (k+1)/(k-1) – m \times n = 2m \times n/(k-1)$

Another interesting approximation is the number of computations if only the diagonal regions are re-computed. In this case, from Formula 5, $r(k)=k$, so Formula 8 becomes:

$$\begin{aligned} T_D(m,n,k) &= m \times n \times (k^2-1) \times [1-k^q/k^{2q}]/[k^2-k)] + u \times v \\ &= m \times n \times (k-1) \times (k+1) \times [1-k^q/k^{2q}]/[k(k-1)] + u \times v \\ &= m \times n \times (k+1) \times [1-k^q/k^{2q}]/k + u \times v \\ &\approx m \times n \times (k+1)/k \end{aligned}$$

The diagonal case number of re-computations, $R_D(m,n,k)$, is:

Formula 11: $R_D(m,n,k) = T_D(m,n,k) – m \times n = m \times n \times (k+1)/k – m \times n = m \times n/k$

Note that the diagonal case is not the best case. In fact, the best case is illustrated by Figure 11, where each re-computation consists of only a single partial row or single partial column. Since the best case is not on the diagonal, it does not result from a minimum value for $r(k)$. Instead, it results from the case where Formula 4 drastically over-estimates the number of boundary computations that must be done in each region. This best case number of re-computations, $R_B(m,n,k)$, is:

Formula 12: $R_B(m,n,k) = m+n-(m+n)/k$

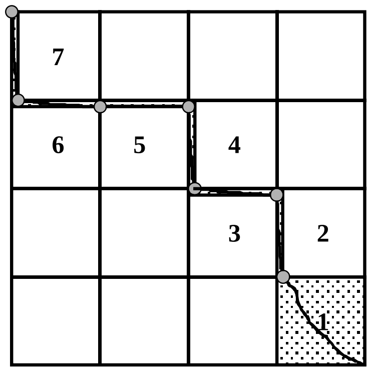


Figure 11. The best case re-computations for the *FastLSA* algorithm with k = 4.

As we will see in the next Section, the empirical number of re-computations is close to the diagonal value, $R_D(m,n,k) = m \times n/k$. For $k = 2$ this number is $m \times n/2$, which is half the number of re-computations done by Hirschberg's algorithm. Even if the worst case did occur in practice, *FastLSA* can reduce the number of re-computations to the $m \times n$ value obtained using Hirschberg's algorithm by selecting $k=3$ in Formula 10. From Formula 3, the space cost of this choice is only $3 \times (m+n)$ which adds little space over the $2 \times min(m,n)$ units used by Hirschberg's algorithm. In general, a value of k can be selected to use all of the available memory so that *FastLSA* reduces the re-computation time to its minimum value at a linear space cost.

Figure 12 shows the number of re-computations for the FM algorithms, Hirschberg's algorithm and *FastLSA*.

	Full Matrix	Hirschberg	*FastLSA* (worst)	*FastLSA* (expected)
Space	m×n	2×min(m,n)	k×(m+n)	k×(m+n)
Re-computations	[0]×(m×n)	[1]×(m×n)	[2/(k-1)]×(m×n)	[1/k]×(m×n)

Figure 12. Space and Time requirements of the FM, Hirschberg's and *FastLSA*.

Experimental Results

We compared the empirical performance of the FM algorithm, Hirschberg's algorithm, and *FastLSA* using a common software and hardware base. The commercial ChromaTool sequence analysis suite developed by BioTools, Inc. (www.biotools.com) uses an implementation of *FastLSA*. For completeness, we implemented an FM algorithm and Hirschberg's algorithm within the same BioTools framework. All algorithms have been tuned for performance including the removal of a number of error checking code segments. Also, all algorithms share the same input/output code, the same scoring table (Figure 1) and an affine gap penalty given by Formula 1 (*initialPenalty = 20* and *extensionPenalty = 10*).

The experiments were performed on a 800 MHz Pentium III (Coppermine) with 16 Kbytes of Level 1 data cache, 256 Kbytes of Level 2 cache (clocked at 800 MHz), 133 MHz front side bus (FSB), 512 MB of main memory and Red Hat Linux 6.1 with the Linux 2.2.16 kernel. Although there are two CPUs, our application is single-threaded. We also performed these experiments on an SGI Origin 2400 machine with 400MHz processors and obtained similar results. The configuration of the SGI machine is described further in the Section on parallelizing *FastLSA*.

We mimic a typical sequence search that takes a new query amino acid or DNA sequence and pairwise aligns it with each sequence in a database. High alignment scores between the query sequence and a specific database sequence are flagged for further consideration by the biologist. Given that these pairwise alignments produce optimal matches for the selected scoring function, the speed of these pairwise alignments is the most important consideration.

For our experiments, we use the well-known Swiss-Prot protein database[ABH994]. We randomly selected 5 sequences of lengths 100, 200, 500, 800, 1000, and 2000 amino acids, plus or minus 5% in length, from the Swiss-Prot database to serve as our query sequences. The results of our first experiment are shown in Figure 13 and Figure 14. The X-axis represents the nominal length of the query sequences. There are three data points for each nominal length: the leftmost data point always represents the FM algorithm, the middle data point is for Hirschberg's algorithm, and the rightmost data point is for *FastLSA*. Since the Y-axis measures time, lower data points represent higher performance.

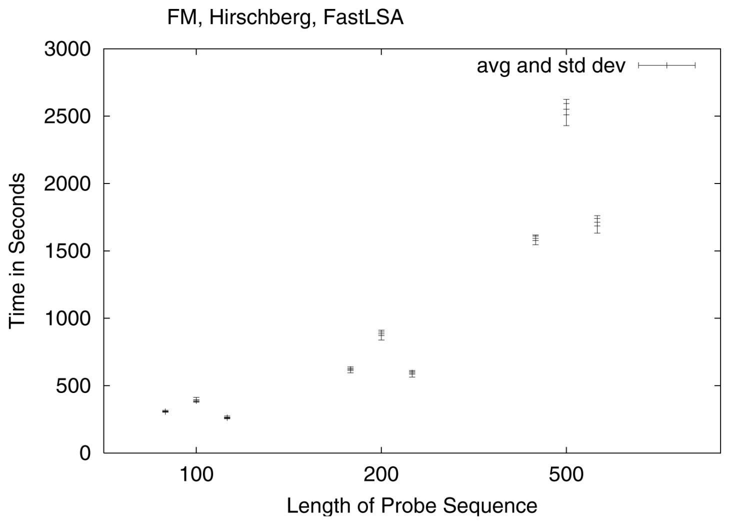


Figure 13. Searching the SwissPort database with FM, Hirschberg and *FastLSA*.

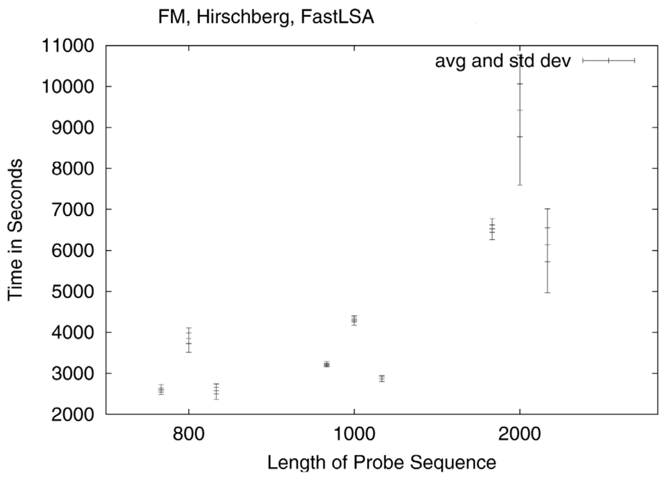


Figure 14. Searching the SwissPort database with FM, Hirschberg and *FastLSA*.

Since 5 sequences, each of the same nominal length, are used as the query sequences for the experiment, there are a total of 30 query sequences from 6 categories based on length. The diamond represents the average time for 5 query sequences of similar length. The bars closest to the diamond represent one standard deviation of the 5 data points and the bars farthest from the diamond represent the minimum and maximum times. Even though no other users were using the CPU during the experiments, background tasks sometimes cause fluctuations in timings. To mitigate these effects, each query sequence was aligned against the entire Swiss-Prot database (which includes the original query sequence) 3 times and the average of the times was taken. However, the times obtained did not show significant variations across the 3 runs. All of the algorithms used the same query sequences and the same version of the Swiss-Prot database. We used Formula 13 to pick a value of *k*, based on the lengths of the two sequences. This formula has been empirically determined to obtain good results.

Formula 13: $k = truncate(\log_{10}(m)) + truncate(\log_{10}(n)) + 3$

We used a heuristic function from the ChromaTool code for picking the buffer size of the recursion-terminating call to our FM code. Figure 15 shows the results of the experiment as a table. For example, for the 5 query sequences of nominal length 100, it took an average of 307 seconds to align a query sequence against the entire Swiss-Prot database using FM. For the 5 query sequences of nominal length 100, the shortest time was 303 seconds, the longest time was 317 seconds, and the standard deviation for the 5 query sequences was 3 seconds. Similarly, for Hirschberg's algorithm, the average time to align a query sequence of nominal length 100 was 389 seconds with a standard deviation of 7 seconds. Finally, for *FastLSA*, the average time to align a query sequences of nominal length 100 was 262 seconds with a standard deviation of 4 seconds.

query length	FM (sec$\times 10^3$)	Hirschberg (sec$\times 10^3$)	*FastLSA* (sec$\times 10^3$)
100	0.307 ± 0.003	0.389 ± 0.007	0.262 ± 0.004
200	0.621 ± 0.008	0.885 ± 0.014	0.595 ± 0.009
500	1.594 ± 0.016	2.551 ± 0.042	1.713 ± 0.028
800	2.594 ± 0.049	3.853 ± 0.129	2.580 ± 0.081
1000	3.216 ± 0.026	4.305 ± 0.048	2.882 ± 0.030
2000	6.531 ± 0.091	9.418 ± 0.642	6.136 ± 0.415

Figure 15. Searching the SwissPort database with FM, Hirschberg and *FastLSA*.

Based on the complexity analysis, one would expect FM to be the fastest algorithm in all cases. After all, as indicated in Figure 12, FM does not require any re-computation to recover the path of the optimal alignment. In contrast, both Hirschberg's algorithm and the *FastLSA* reduce their storage costs at the expense of re-computation. In fact, from Figure 12, FM does 0 re-compuations, Hirschberg's algorithm does $m \times n$ recompuations and *FastLSA* does $(m \times n)/8$ re-computations (for $k = 8$). For example, if the query sequence has size 100 and the database sequences range in size from 100 to 5,000, FM does 0 re-computations, *FastLSA* does 1250 to 62,500 re-computations, and Hirschberg's algorithms does 10,000 to 500,000 re-computations. Since *FastLSA* makes fewer re-computations than Hirschberg's algorithm, it is not surprising that it is consistently faster. However, why is *FastLSA* faster than FM for query sequences of length 100 and 200, slower than FM for sequences of size 500 and then faster again for longer sequences?

It is an inescapable fact of contemporary computer systems is that in practice, the cache behavior of an algorithm can have a substantial impact on its performance. Each query sequence of size 100 was aligned against the entire Swiss-Prot database, which contains sequences ranging from less than 100 amino acids to over 5,000 amino acids as shown in Figure 16. This means that the DPM ranged in size from $100 \times 100 \times 4$ bytes = 40Kbytes to $100 \times 5{,}000 \times 4$ bytes = 2Mbytes. Since the secondary cache has only 256Kbytes, the FM DPM would not fit in secondary cache and a large number of main memory accesses were made. In contrast, the memory requirements for *FastLSA* are much smaller. From Formula 3, *FastLSA* with $k=8$ requires only

8×(100+1000)×16bytes[2] = 140.8Kbytes for the grid vectors. This easily fits into the 256Kbyte secondary cache. Since a main memory access is more than 10 times slower than an access to secondary cache, the fact that the FM DPM did not fit into cache is sufficient to account for the faster *FastLSA* performance. Hirschberg's algorithm also fits into the secondary cache. However, since it does so many more re-computations than *FastLSA*, it cannot overtake the FM algorithm.

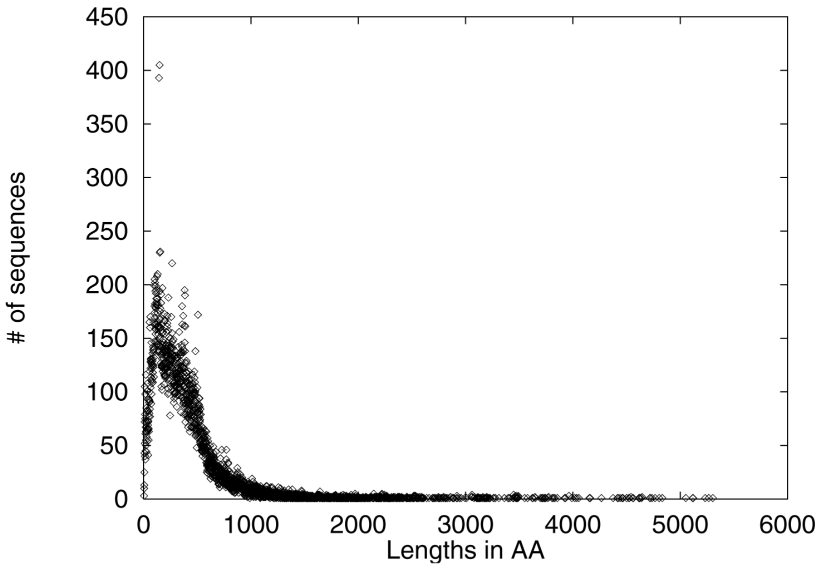


Figure 16. The distribution of sequence lengths in the Swiss-Prot database.

For larger problems, even Hirschberg's algorithm and *FastLSA* require more storage than is available in cache. For example, when the nominal query sequence length reaches 500, the performance of *FastLSA* and FM converge since both algorithms must access main memory. Figure 17 shows the memory required for *FastLSA* with a query sequence of size 500, as the target database sequences range from 100 to 5,000 bytes. Formula 13 was used to pick *k* and Formula 3 was used to estimate the amount of storage used for the grid vectors. In addition to this space for the grid vectors, space must also be allocated for the buffer used in the recursion-ending base case. This space can be reduced as small as necessary to stay in secondary cache, but eventually even the grid vector space is too large to fit in cache.

For example, when the 500 length query is compared to a database sequence of length 2000, *FastLSA* requires: 8×(500+2000)×16bytes = 320Kbytes for the grid vectors. Since this is larger than the 256Kbyte secondary cache, main memory must be used. This makes re-computations expensive and FM becomes the fastest algorithm. Note that using 5 bytes per entry for affine gaps instead of 16 bytes per entry, will allow the *FastLSA* algorithm to stay in cache for longer query sequences. However, this will just move the crossing point from sequences of size 500 to sequences of size 2,000. It will not eliminate this crossing. As the query sequence gets even longer (800, 1,000 and 2,000), *FastLSA* and FM have comparable performance. The re-computations done by *FastLSA* are offset by the fact that *FastLSA* has a smaller memory footprint and therefore has fewer (but still some) cache misses.

[2] The version of *FastLSA* available for these timings used 4 entries of size 4 bytes for a total of 16 bytes for each score to support affine gaps. As described previously in this paper, this size can be reduced to 5 bytes per entry.

Database sequence length	100	500	1,000	2,000	5,000
Memory used	67Kbytes	112Kbytes	192Kbytes	320Kbytes	704Kbytes

Figure 17. Memory requirements for *FastLSA* using a 500 length query for different size database entries.

However, Figures 13 and 14 do not present the whole story. There is a sequence length for which FM will exhaust main memory and have to use disk. This is a disastrous situation for the algorithm since disk access time is more than a million times greater than memory access time. At this point, *FastLSA* and Hirschberg will again dominate FM and by a significantly larger margin. For the main memory configuration used in our experiments (512Mbytes) this does not occur using the Swiss-Prot database, even with a query sequence of size 5,000 since the longest database target sequences have length 5,000. In this case, a pairwise comparison will only require 5,000×5,000×4 = 100Mbytes. However, for DNA sequences this bound can easily be surpassed.

To illustrate the advantages of *FastLSA* for problems larger than the protein sequences found in Swiss-Prot, we experimented with longer DNA sequences. From the GeneBank database[NCBI2001], we selected mouse DNA sequences of nominal lengths 10 kbp, 20 kbp, 30 kbp, and 50 kbp. We randomly selected 21 sequences between 9,950 bp and 10,050 bp, 18 sequences between 19,600 bp and 20,400 bp, 13 sequences between 29,700 bp and 30,300 bp, and 8 sequences between 49,600 bp and 50,400 bp. For each nominal length, five randomly chosen query sequences were aligned against the other sequences and then the time was normalized to the time for a single pairwise alignment. Figure 18 shows the results with sequence length on the x-axis and pairwise alignment time on the y-axis.

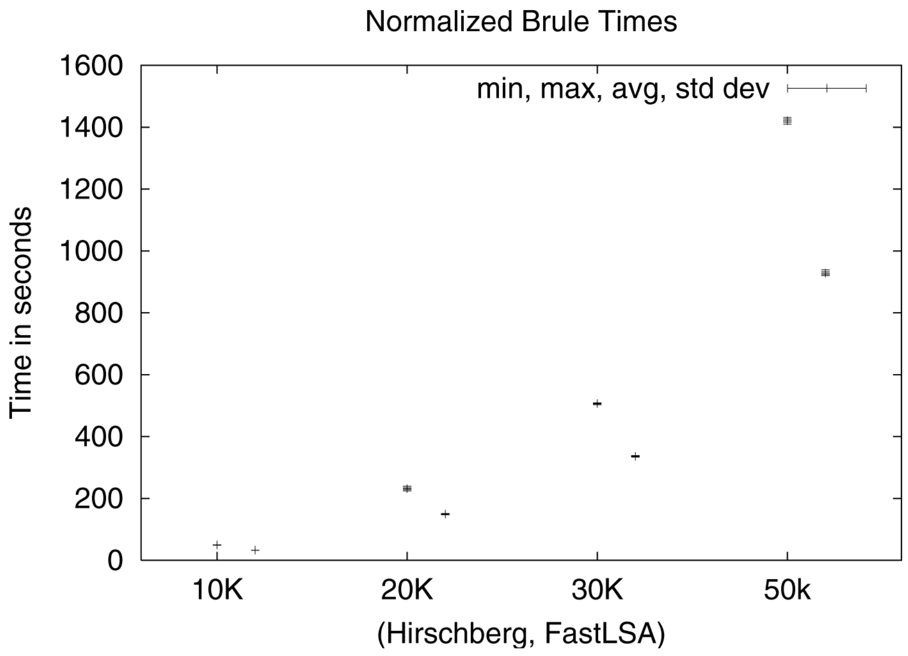


Figure 18. DNA alignment times for *FastLSA* and Hirschberg.

Even for the 10 kbp data point, FM required more main memory storage than is physically available on our computer. The DPM requires approximately $10{,}000 \times 10{,}000 \times 4 = 400$ Mbytes of memory. With kernel and other system overheads, our experience is that just under 400 Mbytes of main memory is free on an otherwise idle computer with 512Mbytes of main memory. Consequently, FM caused the operating system to page to disk and the run-times are prohibitively long for all of the data points in Figure 18. For example, for the smallest 10 kbp data point, FM required five times longer than *FastLSA* to align two sequences. Therefore, we do not consider FM any further.

In contrast to FM, both *FastLSA* and Hirschberg avoid paging to disk even for pairwise alignments of sequences of length 50 kbp. In Figure 18, the leftmost data point for a given nominal length is Hirschberg. The rightmost data point is *FastLSA*. Once again, lower values are better. For all data points, *FastLSA* is faster than Hirschberg. By Formula 13, k = 11 for all the datapoints in Figure 18, therefore we would expect the performance gap between *FastLSA* and Hirschberg to remain constant. In fact, *FastLSA*'s time is consistently about 2/3 of Hirschberg's time.

From Figure 18, we conclude that *FastLSA* is the fastest algorithm in practice for pairwise alignment of sequences longer than 10 kbps. FM is impractical for long sequences that do not fit in main memory due to paging effects. Hirschberg's algorithm does not use available memory effectively and is generally slower than *FastLSA*

From Figure 13 and Figure 14 we conclude that for shorter sequences, the choice of the best algorithm depends on various cache effects. However, *FastLSA* is always better than Hirshberg's algorithm. We can explain this result by computing the ratio of computations done by the two algorithms. Figure 12 gives the number of re-computations for each of the algorithms. Since each algorithm does $m\times n$ initial computations followed by its re-compuations, for $k=11$, Hirschberg's algorithm does $2\times m\times n$ computations and *FastLSA* does $m\times n + (1/11) \times m\times n = 1.09\times m\times n$ computations. The expected ratio of pairwise alignment times of *FastLSA* to Hirschberg's algorithm should be 1.09 / 2 = .55. In practice, from Figure 15, we obtain an average ratio of 0.67. The difference is due to the overhead of maintaining the grid in the *FastLSA* algorithm. However, the bottom line is that *FastLSA* always outperforms Hirschberg's algorithm.

In the typical case of comparing query sequences against a database, such as Swiss-Prot, with sequences of various lengths, *FastLSA* is usually faster than FM due to good caching for short sequences and no paging for longer sequences. However, for a very narrow range of intermediate length sequences when all three algorithms are out of cache, but none of the algorithms exhibit paging, FM and *FastLSA* have very similar performance. This is illustrated, by the 500 and 800 query lengths in Figure 15. For the other data points in Figure 13 and Figure 14, and all of the data points in Figure 18, *FastLSA* dominates.

Parallelizing *FastLSA*

Despite the good performance results reported in the previous section, the theoretical time complexity of *FastLSA* is still quadratic. Consequently, we have parallelized *FastLSA* to further improve performance in practice. Parallel *FastLSA* has two major components:

(1) a parallel implementation of *FullMatrix)* (Figure 7, line 3), which is the base case for the recursion, and

(2) a parallel implementation of *FindScore)* (Figure 7, line 5), which computes the grid

values.

It should be noted that parallel *FastLSA* still uses linear space. In the parallel FM algorithm, the dynamic programming matrix is divided into $R \times C$ equally sized smaller matrices, called tiles. The tiles are laid out in a grid-like pattern with R rows and C columns. The parallel processing starts with one processor computing the entries of the top-left tile (Figure 19, label 1). After this tile is processed, there is enough information available to start computing the entries in its neighboring tiles to the right and below (Figure 19, label 2). The tiles are processed in parallel according to a wavefront data-dependency pattern from top-left to bottom-right. As in the sequential algorithm, after the scores are computed, one processor builds an optimal path, which extends from the bottom-right corner of the DPM to its left or upper boundary.

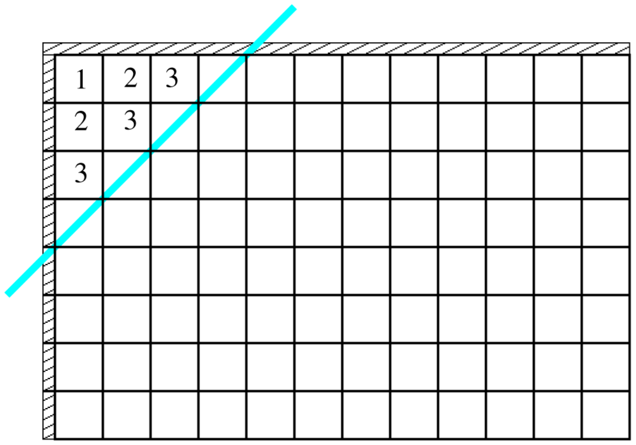


Figure 19. Wavefront parallelism.

In the parallel *FindScore*, the logical DPM is already split into k^2 smaller matrices (sub-matrices) by the grid itself. To increase the amount of parallel work, each of the k^2-1 sub-matrices in the grid are further sub-divided into $R \times C$ equally sized tiles. This new grid has $(k^2-1) \times R \times C$ tiles which are processed in parallel using a strategy similar to that used in the parallel version of the FM algorithm. Details about the parallel *FastLSA* algorithm can be found in [Dri2001].

We have experimented with parallel *FastLSA* on an SGI Origin 2400 shared-memory multiprocessor with 64 processors (400 MHz MIPS R12000), each with 128 KB of primary data cache and a unified 8 MB secondary cache. We used the *sproc* thread library under Irix 6.5.

Unlike the previous experiments that used the BioTools code framework, these experiments used an independent, non-commercial implementation of the algorithms. No degenerate neucleotides were used so all letters were from the set: {A, C, T, G}. A simple scoring function was used whose scoring function assigns 2 for a match and –1 for a mis-match. A non-affine gap penalty of –2 was used.

The data points shown in Figure 20 are a subset of a more extensive set of experiments [Dri2001]. All times are the averages of 5 runs. For relatively short sequences such as Alfa-Globin, the parallel algorithm scales well up to 8 processors. As the number of processors increases, the granularity of the work decreases. This leads to a situation where the processors spend more time trying to get a tile of work than actually computing. Although parallel *FastLSA* computes more than 200 million DPM entries for the alignment of the Alfa-Globin sequences

(10kbs), this is not enough computation to ensure sufficient work for 16 processors, since the real time on 16 processors is only 1.64 seconds. However, as the problem size increases, so does the speedup. For the largest problem, the TCR (T-cell receptor alpha/delta region), both sequences are over 300 kbps in length. Since this corresponds to over 90 billion DPM entries, a reasonable speedup of 17.21 is obtained for 32 processors.

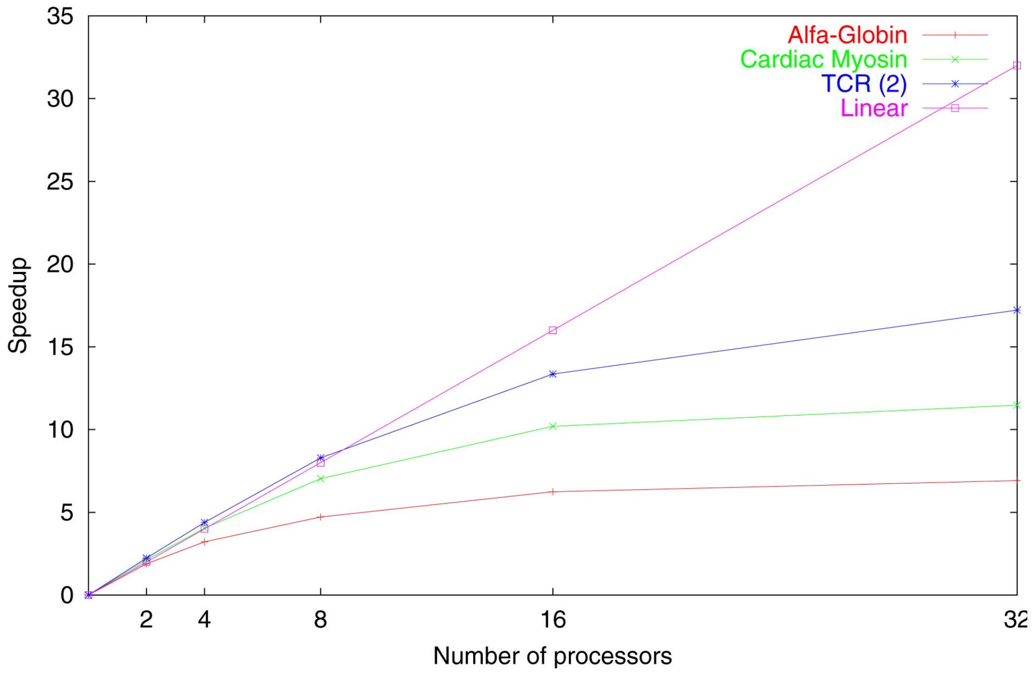


Figure 20. DNA alignment times for *FastLSA* and Hirschberg.

We believe that Parallel *FastLSA* can be tuned to obtain even better speedups for the alignment of medium to large sequences on 16 and 32 processors. However, linear speedups cannot be reached for a large number of processors because of the wavefront dependency between the tiles.

Conclusion

In this paper we have described a new algorithm for optimal pairwise alignment of protein or DNA sequences called *FastLSA*. LikeHirschberg's algorithm *FastLSA* is linear in space. However, unlike Hirschberg's algorithm, *FastLSA* can be allocated as much space as is available in your cache for small sequences or in your main memory for large sequences. *FastLSA* uses this extra space to reduce the number of computations below the number required for Hirschberg's algorithm. We have presented a complexity analysis for *FastLSA* that shows how many fewer computations are performed. Fewer computations means that in theory, *FastLSA* will align sequences faster than Hirschberg's algorithm. We have also provided experimental results that show that *FastLSA* always outperforms Hirschberg's algorithm in practice.

If a full matrix algorithm like Needleman-Wunsch is used for large sequences, the large dynamic programming matrix is too large to fit in main memory. In this case, disk access makes these algorithms much slower than linear space algorithms. However, even with small sequences that fit in main memory, full matrix algorithms are slower than linear space ones, since the faster speed of cache memory provides a significant speed advantage. The bottom line is that *FastLSA* should be used whenever an optimal pairwise alignment is needed.

Acknowledgements

We would like to acknowledge Scott Fortin at BioTools for several helpful discussions. This research was partially funded by research grants from the Protein Engineering Network of Centres of Excellence (PENCE), the National Science and Engineering Research Council (NSERC) and Alberta Informatics Circle of Research Excellence (ICORE).

References

[ABH1994] Appel, R.D., Bairoch, A. and Hochstrasser, D.F., A new generation of information retrieval tools for biologists: the example of the ExPASy WWW server (http://ca.expasy.org/sprot/), Trends Biochem. Sci. 19:258-260 , 1994.
[CSS2000] Charter, K., Schaeffer, J., Szafron, D., Sequence Alignment using *FastLSA*, Proceedings of The 2000 International Conference on Mathematics and Engineering Techniques in Medicine and Biological Sciences (METMBS'2000), June 2000, Las Vegas, Nevada, pp. 239-245.
[DBH1983] Dayhoff, MO., Barker, WC., and Hunt, LT., Establishing homologies in protein sequences. Methods in Enzymology. 91:524-45, 1983.
[Dri2001] Driga, A., Experiences with Sequential and Parallel *FastLSA* Sequence Alignment, M.Sc. thesis, University of Alberta, 2001.
[Hir1975] Hirschberg, D., “A linear space algorithm for computing maximal common subexpressions”, Communications of the ACM, 18(6):341-343, 1975.
[MM1988] Myers, E. and Miller, W., “Optimal alignments in linear space”, CABIOS, Vol. 4, pp. 11-17, 1988.
[NCBI2001] http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=Nucleotide
[NW1970] Needleman, S. and Wunsch, C, “A general method applicable to the search for similarities in the amino acid sequences of two proteins”, Journal of Molecular Biology, Vol. 48, pp. 443-453, 1970.
[SW1981] Smith, T. and Waterman, M., “Identification of common molecular sequences”, Journal of Molecular Biology, Vol. 147, pp. 195-197, 1981.