Bit-Parallel Computation of Local Similarity Score Matrices with Unitary Weights

Heikki Hyyrö[1] and Gonzalo Navarro[2*]

[1] Department of Computer Sciences, University of Tampere, Finland.
e-mail: heikki.hyyro@gmail.com

[2] Department of Computer Science, University of Chile.
e-mail: gnavarro@dcc.uchile.cl

Abstract. Local similarity computation between two sequences permits detecting all the relevant alignments present between subsequences thereof. A well-known dynamic programming algorithm works in time $O(mn)$, m and n being the lengths of the subsequences. The algorithm is rather slow when applied over many sequence pairs. In this paper we present the first bit-parallel computation of the score matrix, for a simplified choice of scores. If the computer word has w bits, then the resulting algorithm works in $O(mn \log \min(m, n, w)/w)$ time, achieving up to 8-fold speedups in practice. Some DNA comparison applications use precisely the simplified scores we handle, and thus our algorithm is directly applicable. In others, our method can be used as a raw filter to discard most of the strings, so the classical algorithm can be focused only on the substring pairs that can yield relevant results.

1 Introduction and Related Work

Sequence comparison is a fundamental task in Computational Biology, in order to detect relevant similarities between a pair of genetic or protein sequences [3]. Three kinds of similarities are of interest: (*i*) global similarity compares two strings as a whole, (*ii*) semiglobal (or semilocal) similarity looks for substrings of a given string that are similar to a second given string, (*iii*) local similarity looks for similar substrings of two given strings.

Similarity is usually expressed using a *score function*, which gives prizes or penalties to operations on the strings and to pairings of characters of the two strings. Usually pairing the same character in both strings involves a prize because we have found a similarity. Pairing different characters, inserting or removing characters, involves penalties. The specific values for prizes and penalties depend on the biological model used for the similarity (for example, logarithms of mutation probabilities). The similarity is then expressed as the highest possible score of a sequence of operations that align one string to the other.

Global and semiglobal similarity find applications in other areas such as text searching. Global similarity computation is then seen as a *distance computation*. The

*Partially funded by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

distance is never negative, and the smaller it is, the more similar the sequences are. Semiglobal similarity can be converted into an approximate search problem, namely to find the approximate occurrences of a short pattern inside a long text. Local similarity, on the other hand, is more specific to computational biology applications.

All these sorts of similarity computations can be easily carried out in $O(mn)$ time using dynamic programming. Given strings $A_{1...m}$ and $B_{1...n}$, the general method is to compute an $(m+1) \times (n+1)$ matrix C whose cell $C_{i,j}$ gives the maximum score/minimum distance to align/convert $A_{...i}$ to $B_{...j}$. The cells of row 0 and column 0 form initially known boundary cases, and the remaining $m \times n$ cells are computed using a recurrence. For example, for global similarity score computation we may have $C_{i,0} = -i$, $C_{0,j} = -j$, and for $i, j > 0$

$$\begin{aligned} C_{i,j} &= \max(C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i,j-1} - 1,\ C_{i-1,j} - 1) \\ \text{where } \delta(A_i, B_j) &= \text{if } A_i = B_j \text{ then } 1 \text{ else } -1 \end{aligned}$$

where we have assumed that all penalties are -1 and prizes are $+1$. More complicated score functions can be real-valued and depend on the characters involved. The maximum score for the strings A and B is $C_{m,n}$.

If we are instead computing distance, we may have $C_{i,0} = i$, $C_{0,j} = j$, and for $i, j > 0$

$$\begin{aligned} C_{i,j} &= \min(C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i,j-1} + 1,\ C_{i-1,j} + 1) \\ \text{where } \delta(A_i, B_j) &= \text{if } A_i = B_j \text{ then } 0 \text{ else } 1 \end{aligned}$$

where we have assumed that all costs are 1. The minimum distance between A and B is $C_{m,n}$.

Semiglobal similarity computation is obtained by using the above formulas except that $C_{0,j} = 0$, so that an alignment of A can start afresh at any position in B. High score/low distance at cell $C_{m,j}$ tells us that an interesting alignment ends at position j in B.

Local similarity computation needs a somewhat different arrangement and, curiously, it seems not expressible using the distance model, but just the score model. In this case we have $C_{i,0} = C_{0,j} = 0$, and for $i, j > 0$

$$\begin{aligned} C_{i,j} &= \max(0,\ C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i,j-1} - 1,\ C_{i-1,j} - 1) \\ \text{where } \delta(A_i, B_j) &= \text{if } A_i = B_j \text{ then } 1 \text{ else } -1 \end{aligned}$$

where we remark the 0 value involved in the maximum. The objective of this zero is that if an alignment in progress has given us more penalties than prizes, then it is better to start afresh from that position. Any cell value $C_{i,j}$ that is high enough indicates that similar substrings end at position i in A and j in B.

Much effort has been carried out in order to efficiently compute the *distance* matrix, both for global and semiglobal alignments. In particular, *bit-parallelism* has given the best results in practice. Bit-parallelism packs several values inside a computer word and updates them all in one shot. The bit-parallel algorithm that best "parallelizes" the matrix computation is from Myers [8], which computes semiglobal similarity and is easily adapted to compute global similarity [4, 5, 6]. Using Myers' algorithm, both similarities can be computed in $O(mn/w)$ time using a computer

word of w bits, which is the optimal bit-parallel speedup. Myers' algorithm strongly relies on the fact that consecutive cells of $C_{i,j}$ differ only by -1, 0, or 1. Several other bit-parallel algorithms exploiting the same property have been proposed [9].

Other approaches to speed up the computation exist. Different Four-Russians techniques [7, 11] obtain $O(mn/\log(mn))$ time. The same complexity is obtained by using a Ziv-Lempel factoring [2], which generalizes to local similarity with arbitrary weights. In practice, when applicable, bit-parallel algorithms are faster.

Bit-parallel computation of *score* matrices, however, has not been attempted. Bergeron and Hamel [1] have extended Myers' scheme to handle arbitrary integer weights for substitutions, as well as a fixed weight c for insertions and deletions. Their algorithm is $O(mnc\log(c)/w)$ time. This scheme cannot be used to compute local similarity.

In general, global and semiglobal score computation can be converted into distance computation. However, local similarity is of different nature and cannot be easily mapped to a known distance computation scheme. In this paper we present a bit-parallel algorithm inspired on Myers' scheme (and more precisely on Hyyrö's version [4]), which obtains $O(mn\log\min(m,n,w)/w)$ time. The algorithm assumes that aligning two characters yields a prize of $+1$ when they are equal and a penalty of -1 otherwise, and that inserting or deleting characters has a penalty of -1.

The main obstacles to obtain the algorithm are (1) that the recurrence is more complicated than the one afforded by Myers (in particular, differences of $+2$ among contiguous cells are possible), and (2) that the zero in the maximization involves knowing absolute cell values, while the whole philosophy of Myers' scheme relies on storing differential values.

We implemented the algorithm and compared it against plain dynamic programming, which is currently the only alternative. We show that up to 8-fold speedups are obtained using our algorithm.

Our algorithm cannot replace dynamic programming because it cannot handle other prize and penalty values. On the other hand, while score computations on protein sequences are always weighted, there are many cases of score computations on DNA sequences where our simplified model is actually used [3]. It may also be feasible to use our method as a fast filter to discard most of the matrix and let the weighted dynamic programming algorithm concentrate only on the matrix areas that look promising.

2 A Bit-Parallel Design

Let us focus on the simple score function depicted in the Introduction, that is,

$$\begin{aligned} C_{i,0} &= C_{0,j} = 0 \quad \text{and, for } i,j>0, \\ C_{i,j} &= \max(0,\ C_{i-1,j-1}+\delta(A_i,B_j),\ C_{i,j-1}-1,\ C_{i-1,j}-1) \\ \text{where } \delta(A_i,B_j) &= \text{if } A_i = B_j \text{ then } 1 \text{ else } -1 \end{aligned}$$

We prove now some properties of matrix C. Note, to start, that C contains no negative values.

Lemma 1: Given the above definition of matrix C, it holds

$$\begin{aligned} C_{i,j} - C_{i-1,j-1} &\in -1, 0, +1 && \text{for any } i,j > 0 \\ C_{i,j} - C_{i,j-1} &\in -1, 0, +1, +2 && \text{for any } i \geq 0, j > 0 \\ C_{i,j} - C_{i-1,j} &\in -1, 0, +1, +2 && \text{for any } i > 0, j \geq 0 \end{aligned}$$

Proof: We proceed inductively, so we assume it proved for any (i', j') such that $j' < j$, or $j' = j$ and $i' < i$. The base cases are immediate. Now, for the inductive case, let us start with the first proposition. The option $C_{i-1,j-1} + \delta(A_i, B_j)$ in the "max" clause of the formula for $C_{i,j}$ guarantees that $C_{i,j} - C_{i-1,j-1} \geq -1$. Inductive Hypothesis tells us that $C_{i-1,j} \leq C_{i-1,j-1} + 2$ and $C_{i,j-1} \leq C_{i-1,j-1} + 2$, and thus $C_{i,j} = \max(0,\ C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i,j-1} - 1,\ C_{i-1,j} - 1) \leq \max(C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i-1,j-1} + 1,\ C_{i-1,j-1} + 1) = C_{i-1,j-1} + 1$. Here we removed the zero from the "max" clause as it is known that $C_{i-1,j-1} + 1 \geq 1 > 0$. By combining the two previous observations, we have that $-1 \leq C_{i,j} - C_{i-1,j-1} \leq 1$.

Let us now consider the second proposition. First we note that $C_{i,j} - C_{i,j-1} \geq -1$ because of the option $C_{i,j-1} - 1$ inside the "max" clause. From our Inductive Hypothesis and the above-proved first proposition we have that $C_{i,j-1} \geq C_{i-1,j-1} - 1 \geq C_{i,j} - 1 - 1 = C_{i,j} - 2$. Thus $-1 \leq C_{i,j} - C_{i,j-1} \leq 2$. The third proposition is symmetric with the second and comes out similarly. □

Given the ranges of values proved for consecutive differences, we will represent matrix C incrementally using the following *bit matrices*:

$$\begin{aligned} M_{i,j} &\equiv A_i = B_j & \qquad DP_{i,j} &\equiv C_{i,j} - C_{i-1,j-1} = +1 \\ Z_{i,j} &\equiv C_{i,j} = 0 & \qquad DZ_{i,j} &\equiv C_{i,j} - C_{i-1,j-1} = 0 \\ & & \qquad DM_{i,j} &\equiv C_{i,j} - C_{i-1,j-1} = -1 \\ HT_{i,j} &\equiv C_{i,j} - C_{i,j-1} = +2 & \qquad VT_{i,j} &\equiv C_{i,j} - C_{i-1,j} = +2 \\ HP_{i,j} &\equiv C_{i,j} - C_{i,j-1} = +1 & \qquad VP_{i,j} &\equiv C_{i,j} - C_{i-1,j} = +1 \\ HZ_{i,j} &\equiv C_{i,j} - C_{i,j-1} = 0 & \qquad VZ_{i,j} &\equiv C_{i,j} - C_{i-1,j} = 0 \\ HM_{i,j} &\equiv C_{i,j} - C_{i,j-1} = -1 & \qquad VM_{i,j} &\equiv C_{i,j} - C_{i-1,j} = -1 \end{aligned}$$

Here M and Z stand for "match" and "zero", respectively. D, H, and V stand for "diagonal", "horizontal", and "vertical", respectively. T, P, Z, and M stand for "plus two", "plus one", "zero", and "minus one", respectively. When a cell refers to a value out of bounds, such as $HP_{i,0}$, its value is not really important.

The above information clearly represents the cells of matrix C. For example,

$$C_{i,j} = \sum_{r=1}^{i} (2 \times VT_{r,j} + 1 \times VP_{r,j} - 1 \times VM_{r,j})$$

The next step is to derive logical properties that relate those bit matrices, so as to permit an efficient bit-parallel implementation.

$DP_{i,j} \equiv M_{i,j} \vee VT_{i,j-1} \vee HT_{i-1,j}$:
It is clear that if either $A_i = B_j$, $C_{i,j-1} = C_{i-1,j-1} + 2$, or $C_{i-1,j} = C_{i-1,j-1} + 2$, then $C_{i,j} = C_{i-1,j-1} + 1$. Moreover, if none of them hold, there is no way for $C_{i,j}$ to get the value $C_{i-1,j-1} + 1$.

$DZ_{i,j} \equiv \sim DP_{i,j} \wedge (Z_{i-1,j-1} \vee VP_{i,j-1} \vee HP_{i-1,j})$:

From the score recurrence we can easily derive the rule that $C_{i,j} = C_{i-1,j-1}$ if and only if $C_{i,j} \neq C_{i-1,j-1}+1$ and $\max(0, C_{i,j-1}-1, C_{i-1,j}-1) = C_{i-1,j-1}$. Moreover, since $0 \leq C_{i-1,j-1}$ and the condition $C_{i,j} \neq C_{i-1,j-1}+1$ implies that $C_{i,j-1} < C_{i-1,j-1}+2$ and $C_{i-1,j} < C_{i-1,j-1}+2$, it turns out that already $C_{i-1,j-1} \geq \max(0, C_{i,j-1}-1, C_{i-1,j}-1)$, so the condition $\max(0, C_{i,j-1}-1, C_{i-1,j}-1) = C_{i-1,j-1}$ can be changed into the form $C_{i-1,j-1} \in \{0,\ C_{i,j-1}-1,\ C_{i-1,j}-1\}$. This results in the above formula for $DZ_{i,j}$.

$DM_{i,j} \equiv \sim (DP_{i,j} \vee DZ_{i,j})$: As it is the only remaining choice.

$HT_{i,j} \equiv DP_{i,j} \wedge VM_{i,j-1}$:

From now on we build on $D*$ and the other bit matrices, by exhaustively examining all the choices for $C_{i,j} - C_{i-1,j-1}$ using submatrices where the lower right cell is $C_{i,j} = x$ and the upper left can thus have a value $x-1$, x or $x+1$. The lower left cell is $C_{i,j-1}$, which in this particular item must have the value $x-2$. We discard cases that are not possible according to Lemma 1 and express the remaining cases as logical conditions. We put "$\times$" in the remaining corner to signal impossible cases.

$x-1$	
$x-2$	x

x	$\times$
$x-2$	x

$x+1$	$\times$
$x-2$	x

$HP_{i,j} \equiv (DP_{i,j} \wedge VZ_{i,j-1}) \vee (DZ_{i,j} \wedge VM_{i,j-1})$:

$x-1$	
$x-1$	x

x	
$x-1$	x

$x+1$	$\times$
$x-1$	x

$HM_{i,j} \equiv VT_{i,j-1} \vee (DZ_{i,j} \wedge VP_{i,j-1}) \vee (DM_{i,j} \wedge VZ_{i,j-1})$:

$x-1$	
$x+1$	x

x	
$x+1$	x

$x+1$	
$x+1$	x

Note the simplification in the first condition since $VT_{i,j-1} \Rightarrow DP_{i,j}$.

$HZ_{i,j} \equiv \sim (HT_{i,j} \vee HP_{i,j} \vee HM_{i,j})$: As it is the only remaining choice.

$VT_{i,j} \equiv DP_{i,j} \wedge HM_{i-1,j}$:

Now we focus on the upper right corner.

$x-1$	$x-2$
	x

x	$x-2$
$\times$	x

$x+1$	$x-2$
$\times$	x

$VP_{i,j} \equiv (DP_{i,j} \wedge HZ_{i-1,j}) \vee (DZ_{i,j} \wedge HM_{i-1,j})$:

$x-1$	$x-1$
	x

x	$x-1$
	x

$x+1$	$x-1$
$\times$	x

$VM_{i,j} \equiv HT_{i-1,j} \vee (DZ_{i,j} \wedge HP_{i-1,j}) \vee (DM_{i,j} \wedge HZ_{i-1,j})$:

$x-1$	$x+1$
	x

x	$x+1$
	x

$x+1$	$x+1$
	x

$VZ_{i,j} \equiv \sim (VT_{i,j} \vee VP_{i,j} \vee VM_{i,j})$: As it is the only remaining choice.

3 A Bit-Parallel Algorithm

Up to now we have focused on how to compute the C matrix without regard for which should be the output of the algorithm. As explained, computational biologists are interested in matrix positions where the local score exceeds some threshold k. Those positions are then subject of further analysis.

Hence our algorithm will receive two strings A and B, as well as a threshold value k, and will point out all the positions (i, j) of matrix C where the score of the local alignment between $A_{...i}$ and $B_{...j}$ is at least k, that is, where $C_{i,j} \geq k$.

The idea of the bit-parallel algorithm is to process C column by column (just like the standard dynamic programming algorithm). However, the bit-parallel algorithm will process all the column in one shot, not row by row. In this section we assume $m \leq w$, that is, we can pack all bits of a column $G_j = G_{1...m,j}$ in a single computer word, for any matrix G. Note that row zero is not represented. When needed, the ith bit of vector G_j will be written as $G_j(i) = G_{i,j}$.

Therefore, our computation will proceed with column bit vectors DP_j, DM_j, DZ_j, and so on, for $j = 0 \ldots n$, each packed in a computer word. After step j of the algorithm, the vectors will hold the bits corresponding to column j of the matrix.

We will use the usual C instructions to handle bits: "&" as the bitwise-and, "|" as the bitwise-or, "^" as the bitwise-xor, "$\sim$" as the bitwise-not, and "<<" to shift all the bits one position to the left and enter a zero at the rightmost position. Sometimes we will treat bit vectors as integers and perform arithmetic operations on them.

In a precomputation step, explained in Section 3.1, the "match" matrix M is built in a suitable way for bit-parallel processing. The boundary conditions of matrix C are handled by giving the proper values to Z_0 and $V*_0$ vectors, namely $VP_0 = VM_0 = VT_0 = 0$ and $Z_0 = VZ_0 = 2^m - 1$. Then we process the characters of B (matrix columns) one by one. Each step j computes the bit vectors for column j from the vectors of column $j - 1$. First, the diagonal vectors $D*_j$ as well as the horizontal vector HP_j are computed. Vector HP_j is computed already at this stage as we use it in computing DZ_j. This part is complex and is explained in Section 3.2. Then the rest of the horizontal and vertical vectors $H*_j$ and $V*_j$ are easy to compute, as explained in Section 3.3. Finally, in Section 3.4, we show how to find and report high enough scores in column j, and how the same mechanism handles also computing vector Z_j. The way this last part is done is again slightly complicated and uses a technique that is rather different from all the rest.

3.1 Computing Matrix M

Matrix M is represented as a table indexed by alphabet characters. $M[c]$ is a bit vector such that $M[c](i) = 1$ iff $A_i = c$. This table is precomputed before filling matrix C. This way the cell value $M_{i,j}$ is actually represented by $M[B_j](i)$.

Matrix M is precomputed in $O(m + |\Sigma|)$ time, where Σ is the alphabet of A and B, as follows. First initialize $M[c] \leftarrow 0$ for every $c \in \Sigma$ and then traverse string A character-wise, setting bit $M[A_i](i) \leftarrow 1$.

3.2 Computing Vectors $D*_j$ and HP_j

Let us start with DP_j. As seen in Section 2, $DP_{i,j} \equiv M_{i,j} \vee VT_{i,j-1} \vee HT_{i-1,j}$. Since we are computing all the values at column j in one shot, component $HT_{i-1,j}$ is troublesome because it is not yet computed ($M_{i,j} = M[B_j](i)$ is known so it is not problematic). Let us expand $HT_{i-1,j}$ using its definition:

$$DP_{i,j} \equiv M_{i,j} \vee VT_{i,j-1} \vee (DP_{i-1,j} \wedge VM_{i-1,j-1})$$

where now the problematic value belongs to the same DP column. Let us express this recurrence in vector form. We define temporary vectors $X(i) \equiv M[B_j](i) \vee VT_{j-1}(i)$ and $Y(i) \equiv VM_{j-1}(i)$. Then the recurrence for vector DP_j is

$$DP_j(i) \equiv X(i) \vee (DP_j(i-1) \wedge Y(i-1))$$

This particular kind of circular dependency has already been solved by Myers [8] in his simpler formulation for edit distance computation. Following Hyyrö's explanation [4, 10], we unroll $DP_j(i-1)$ to obtain

$$DP_j(i) \equiv X(i) \vee (X(i-1) \wedge Y(i-1)) \vee (DP_j(i-2) \wedge Y(i-1) \wedge Y(i-2))$$

and unrolling repeatedly we obtain

$$DP_j(i) \equiv \vee_{r=0}^{i} \left(X(i-r) \wedge \left(\wedge_{s=i-r}^{i-1} Y(s)\right)\right)$$

that is, any bit set in X before position i can propagate through a sequence of bits set in Y that reach position $i-1$, so as to set position i in DP_j. Myers [8] has shown that the above formula can be computed using bit-parallelism as follows:

$X \leftarrow M[B_j] \mid VT_{j-1}$
$Y \leftarrow VM_{j-1}$
$DP_j \leftarrow ((Y + (X \,\&\, Y)) \,\hat{}\, Y) \mid X$

Let us now consider DZ. From Section 2 we have

$$DZ_{i,j} \equiv \sim DP_{i,j} \wedge (Z_{i-1,j-1} \vee VP_{i,j-1} \vee HP_{i-1,j})$$

where this time the problem arises with $HP_{i-1,j}$. But it turns out that vector HP_j can be computed once the vector DP_j is known. In Section 2 we gave the formula

$$HP_{i,j} \equiv (DP_{i,j} \wedge VZ_{i,j-1}) \vee (DZ_{i,j} \wedge VM_{i,j-1})$$

for it. If we look at the situation where the condition $DZ_{i,j} \wedge VM_{i,j-1}$ is true, we can have $C_{i,j} = x$ only if $C_{i-1,j} = x+1$, that is, only if $HP_{i-1,j}$ is true. Also, $DP_{i,j}$ must obviously be false. Hence, $DZ_{i,j} \wedge VM_{i,j-1} \Rightarrow HP_{i-1,j} \wedge VM_{i,j-1} \wedge \sim DP_{i,j}$. Moreover, it is straighforward to see that the condition $DZ_{i,j} \wedge VM_{i,j-1}$ is true whenever $HP_{i-1,j} \wedge VM_{i,j-1} \wedge \sim DP_{i,j}$ is true, and thus we have the following alternative formula for $HP_{i,j}$:

$$HP_{i,j} \equiv (DP_{i,j} \wedge VZ_{i,j-1}) \vee (HP_{i-1,j} \wedge VM_{i,j-1} \wedge \sim DP_{i,j})$$

The circular dependency on HP_j can be solved in similar fashion as in the case of computing vector DP_j. In this case, defining temporary vectors X and Y such that

$X(i) \equiv DP_j(i) \wedge VZ_{j-1}(i)$ and $Y(i) \equiv VM_{j-1}(i+1) \wedge \sim DP_j(i+1)$, the preceding formula for $HP_{i,j}$ gets the vector form

$$HP_j(i) \;\equiv\; X(i) \;\vee\; (HP_j(i-1) \;\wedge\; Y(i-1))$$

which is identical to the previous circular dependency for computing DP_j. We get immediately the following bit-parallel formula for computing HP_j:

$X \leftarrow DP_j \;\&\; VZ_{j-1}$
$Y \leftarrow (VM_{j-1} \;\&\; \sim DP_j) >> 1$
$HP_j \leftarrow ((Y + (X \;\&\; Y)) \,\hat{}\, Y) \mid X$

Once vector HP_j is available, computing the vector DZ_j becomes easy: a straightforward conversion of its formula leads into the following bit-parallel code.

$DZ_j \leftarrow \sim DP_j \;\&\; (((Z_{j-1} << 1) \mid 1) \mid VP_{j-1} \mid (HP_j << 1))$

where, after the shift of Z_{j-1} we have introduced a "1" at its lowest bit to reflect the fact that $C_{0,j-1} = 0$ (that is, $Z_{0,j-1} = 1$) for any j (recall that row zero of Z is not represented). Similarly, $HP_{0,j} = 0$ because $C_{0,j} - C_{0,j-1} = 0 \neq 1$, so we leave the new rightmost bit in zero after shifting HP_j. Finally, we have the following simple bit-parallel formula for DM_j.

$DM_j \leftarrow \sim (DP_j \mid DZ_j)$

3.3 Computing Other Vectors $H*_j$ and $V*_j$

Once DP_j, HP_j, DM_j, and DZ_j corresponding to the current column j are computed, the rest flows easily by following the formulas used in Section 2. Again, when we shift a bit vector to the left, we add or not a "1" bit at the rightmost position depending on which is the value of that vector at the unrepresented row zero.

$HT_j \leftarrow DP_j \;\&\; VM_{j-1}$
$HM_j \leftarrow VT_{j-1} \mid (DZ_j \;\&\; VP_{j-1}) \mid (DM_j \;\&\; VZ_{j-1})$
$HZ_j \leftarrow \sim (HT_j \mid HP_j \mid HM_j)$
$VT_j \leftarrow DP_j \;\&\; (HM_j << 1)$
$VP_j \leftarrow (DP_j \;\&\; ((HZ_j << 1) \mid 1)) \mid (DZ_j \;\&\; (HM_j << 1))$
$VM_j \leftarrow (HT_j << 1) \mid (DZ_j \;\&\; (HP_j << 1)) \mid (DM_j \;\&\; ((HZ_j << 1) \mid 1))$
$VZ_j \leftarrow \sim (VT_j \mid VP_j \mid VM_j)$

3.4 Keeping Scores and Computing Vector Z_j

Once the bit vectors for column j have been computed, we check whether some cell values in column j of matrix C exceed the matching threshold k. At the same time it is also convenient to check which cells have the value zero and record those positions into vector Z_j. Unfortunately the differential information of the bit vectors does not allow us to make this in any simple and fast way. The naive approach would be to use the difference information between adjacent cell values to compute and check the cell values $C_{1\ldots m,j}$. This would take $O(m)$ time per column, making the overall run time $O(mn)$, the same as with classical dynamic programming.

On the other hand, as shown by Myers [8], a single value $C_{i,1\ldots n}$ can be tracked in constant time per column by using the horizontal vectors $H*_j$. The problem is that we need to track all the rows i, falling again to $O(m)$ time per column.

Our approach is to set up multiple *witnesses* into a single bit vector, and then scan the column in parallel with the witnesses. Each witness will be associated with some i and keep track of the cell values $C_{i,1\ldots n}$, that is, the cell values on row i of C. A somewhat similar method was used in [5, 6] as part of an approximate string matching algorithm.

Let MW_j be a length-m bit vector that holds the multiple witnesses at column j and let Q denote the number of bits taken by each witness. Then MW_j can hold $r = \lfloor m/Q \rfloor$ witnesses. Let $MW_j\{i\}$ denote a witness that has its first bit in position i of MW_j. $MW_j\{i\}$ occupies the bits $MW_j(i \ldots i+Q-1)$ and keeps track of the cell values on row i of C. The first witness is always $MW_j\{1\}$, and the rest are spread evenly into MW_j. This can be done in such manner that the largest empty gap after the region of any witness is $\lceil (m-rQ)/r \rceil$. Let us define $Q' = Q + \lceil (m-rQ)/r \rceil$, that is, Q' gives the maximum distance between the first bit of a witness and the first bit of the next witness or, for the last witness, the position after the last bit of the whole vector.

Assume that $C_{i,j} = x$ and the witness $MW_j\{i\}$ exists. For reasons that become clear below, we record the value x into $MW_j\{i\}$ in the form $2^{Q-1} - x$. To guarantee that the witnesses can represent all possible score values from zero to $\min(m, n)$, the parameter Q is determined as the minimum number for which $2^{Q-1} \geq \min(m, n)$, that is, $Q = \lceil \log_2 \min(m, n) \rceil + 1$. Figure 1 exemplifies (vectors S, E, K will be introduced soon).

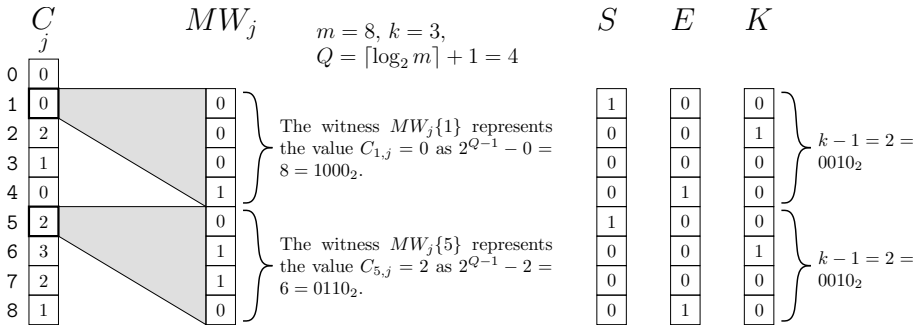

Figure 1: Example of usage of MW, S, E, and K vectors.

With these conventions the witnesses have the following properties:

(1) The Qth bit of $MW_j\{i\}$ is set if and only if $C_{i,j} = 0$.

(2) Adding some value x to $C_{i,j}$ corresponds to subtracting x from $MW_j\{i\}$, and vice versa.

(3) If we add $k-1$ to $MW_j\{i\}$, then the Qth bit of $MW_j\{i\}$ is set if and only if $C_{i,j} < k$.

The witnesses are initialized to $MW_0\{i\} = 2^{Q-1}$ since all values in column 0 of C are zero. After that the witness values are computed by using the horizontal vectors. For

example, if $MW_{j-1}\{i\} = x$ and the ith bit of HT_j is set, then $MW_j\{i\} = MW_{j-1}\{i\} - 2 = x - 2$ (note that we subtracted the $+2$ due to property (2)). When MW_{j-1} and the horizontal vectors $H*_j$ are available, all witnesses $MW_j\{1\} \ldots MW_j\{r\}$ may be computed in bit-parallel fashion. To achieve this, we use a "start" bit mask S with bits set in those locations that correspond to the first bits of witnesses. Then, the whole witness vector MW_j may be computed as:

$$MW_j \leftarrow MW_{j-1} - 2(HT_j \;\&\; S) - (HP_j \;\&\; S) + (HM_j \;\&\; S)$$

Once MW_j and the vertical vectors $V*_j$ are available, all cell values in column j of C can be scanned in bit-parallel manner. First we copy MW_j into an auxiliary vector X. At this stage each witness $MW_j\{i\}$ copied into X represents the value $C_{i,j}$. Then each witness MW_j is updated $Q' - 1$ times. First to represent the value $C_{i+1,j}$, then the value $C_{i+2,j}$, and so on until the value $C_{i+Q'-1,j}$. After $Q' - 1$ iterations, all cells of column j have been scanned (some possibly twice if $Q' \neq Q$). At each stage of the scan we check the current witness values for matches or zeros. For this we use an "end" bit mask $E \leftarrow S << (Q-1)$ that has a bit set in those positions that correspond to the last bits of the witnesses. In addition we use a bit mask K that holds the value $k - 1$ at each witness location.

When the witnesses $MW_j\{i\}$ in X represent the cells $C_{i+h,j}$, the vector $((X + K) \;\&\; E) >> (Q - 1 - h)$ has bits set in those positions u where $C_{u,j} < k$, and the vector $(X \;\&\; E) >> (Q - 1 - h)$ has bits set in those positions u where $C_{u,j} = 0$.

Our strategy for checking matches is to record during the scan whether column j contains any matches or not. These may then be checked more carefully, if desired, but if all matching locations are recorded exactly, the run time becomes again $O(mn)$ in the worst case.

The match checking is done by using an auxiliary vector Y that is initialized by setting $Y \leftarrow E$. When $MW_j\{i\}$ represents $C_{i+h,j}$, we set $Y \leftarrow Y \;\&\; (X + K)$. There is at least one match in column j if and only if $Y \neq E$ after the Q' iterations (consisting of the initial stage and $Q' - 1$ update stages). The zero vector Z_j is computed by initializing it to zero and setting $Z \leftarrow Z \mid ((X \;\&\; E) >> (Q - 1 - h))$ when $MW_j\{i\}$ represents $C_{i+h,j}$.

computes MW_j, then it updates the witnesses in the auxiliary vector X to go through all cell values in column j. It also records matching columns as well as computes the vector Z_j during the scan. The first stage is handled separately, and for this reason for example the vector Z_j is directly given a non-zero value.

Figure 2 gives the complete algorithm. Note that, by carefully choosing the update order, we manage to keep only one copy of each vector.

3.5 Analysis

Up to now we have assumed that $m \leq w$. In this case computing the M table takes $O(m+|\Sigma|)$ time, and the rest of the algorithm in Figure 2 clearly runs in time $O(nQ')$. Since $Q' < 2Q$ and $Q = \lceil \log_2 \min(m,n) \rceil + 1$, we have that $nQ' = O(n \log \min(m,n))$ and the total running time is $O(|\Sigma| + m + n \log \min(m,n))$.

If $m > w$, the length-m bit vectors can be simulated in $O(\lceil m/w \rceil)$ time by using $\lceil m/w \rceil$ vectors of length w (details are omitted for lack of space). This results in the time $O(m + \lceil m/w \rceil |\Sigma|)$ for computing the M table, and the run time of the rest of the

LocalScores $(A_{1\ldots m},\ B_{1\ldots n},\ k)$

1. **For** $c \in \Sigma$ **Do** $M[c] \leftarrow 0$
2. **For** $i \in 1 \ldots m$ **Do** $M[A_i] \leftarrow M[A_i] \mid 2^{i-1}$
3. $VP, VM, VT \leftarrow 0,\ VZ, Z \leftarrow 2^m - 1$
4. $Q \leftarrow \lceil \log_2 m \rceil + 1$
5. $r \leftarrow \lfloor m/Q \rfloor$
6. $S \leftarrow$ distribute evenly r witnesses and mark their first bit
7. $MW, E \leftarrow S << (Q-1)$
8. $K \leftarrow S \times (k-1)$
9. $Q' \leftarrow Q + \lceil (m - rQ)/r \rceil$
10. **For** $j \in 1 \ldots n$ **Do**
11. $\quad X \leftarrow M[B_j] \mid VT$
12. $\quad DP \leftarrow ((VM + (X \ \&\ VM)) \wedge VM) \mid X$
13. $\quad X \leftarrow DP \ \&\ VZ$
14. $\quad Y \leftarrow (VM \ \&\ \sim DP) >> 1$
15. $\quad HP \leftarrow ((Y + (X \ \&\ Y)) \wedge Y) \mid X$
16. $\quad DZ \leftarrow \sim DP \ \&\ (((Z << 1) \mid 1) \mid VP \mid (HP << 1))$
17. $\quad DM \leftarrow \sim (DP \mid DZ)$
18. $\quad HT \leftarrow DP \ \&\ VM$
19. $\quad HM \leftarrow VT \mid (DZ \ \&\ VP) \mid (DM \ \&\ VZ)$
20. $\quad HZ \leftarrow \sim (HT \mid HP \mid HM)$
21. $\quad VT \leftarrow DP \ \&\ (HM << 1)$
22. $\quad VP \leftarrow (DP \ \&\ ((HZ << 1) \mid 1)) \mid (DZ \ \&\ (HM << 1))$
23. $\quad VM \leftarrow (HT << 1) \mid (DZ \ \&\ (HP << 1)) \mid (DM \ \&\ ((HZ << 1) \mid 1))$
24. $\quad VZ \leftarrow \sim (VT \mid VP \mid VM)$
25. $\quad MW \leftarrow MW - 2(HT \ \&\ S) - (HP \ \&\ S) + (HM \ \&\ S)$
26. $\quad X \leftarrow MW$
27. $\quad Y \leftarrow E$
28. $\quad Z \leftarrow 0$
29. $\quad$ **For** $h \in 0 \ldots Q' - 1$ **Do**
30. $\quad\quad Z \leftarrow Z \mid ((X \ \&\ E) >> (Q - 1 - h))$
31. $\quad\quad Y \leftarrow Y \ \&\ (X + K)$
32. $\quad\quad X \leftarrow X - 2((VT_j >> h) \ \&\ S) - ((VP_j >> h) \ \&\ S) + ((VM_j >> h) \ \&\ S)$
33. $\quad$ **If** $Y \neq E$ **Then** `Record match at column` j

Figure 2: Complete bit-parallel algorithm to compute local similarity. Some optimizations have been discarded for clarity.

algorithm is multiplied by a factor of $O(\lceil m/w \rceil)$, which yields $O(mn \log \min(m,n)/w)$, taking the alphabet size as a constant for simplicity.

The run time $O(mn \log \min(m,n,w)/w)$ mentioned in the beginning of the paper is finally achieved by observing that the values in different length-w segments of the bit vectors may be stored using delta encoding. If $C_{\lceil hw+w/2 \rceil, j} = x$ for some $h \geq 0$, we know from Lemma 1 that the values in the corresponding length-w section, $C_{hw+1 \ldots (h+1)w, j}$, must be in the range $x - w + 1 \ldots x + w$. Thus if the witnesses for section $C_{hw+1 \ldots (h+1)w, j}$ represent the values of form $C_{\lceil hw+w/2 \rceil, j} - C_{i,j}$, we may use the value $Q = \lceil \log_2 \min(m, n, 2w) \rceil + 1$ in storing the witnesses. Here we use the value $2w$ instead of w in order to ensure that the sum $X + K$ in match checking does not cause an overflow. Note that this scheme requires some modifications in the process of checking for zero values and/or matches. For example the values in K must be adjusted depending on the current base-value x.

Compared to the best bit-parallel complexity for global and semiglobal similarity (actually, for distance computation), $O(mn/w)$, we have a logarithmic penalty factor because of the use of local similarity. At this point it should be clear that we can compute global and semiglobal scores (rather than distances) within the same $O(mn/w)$ complexity, just by removing the use of vector Z and checking the score only at a single cell or a single row. This removes the need for the witnesses and their logarithmic penalty.

4 Experimental Results

We implemented the $O(mn \log \min(m,n,w)/w)$ version of our algorithm and compared it to the plain dynamic programming algorithm. Both algorithms were programmed in C, and we tried to make both implementations as efficient as possible. The test computer was a 64-bit Sparc Ultra 2 with 128 MB RAM, and the codes were compiled with GCC 3.3.1 with optimization switched on. The test strings were randomly selected DNA sequences from the genome of S. cerevisiae (baker's yeast). The test contained two different types of scenarios. In the first we tested with short patterns and a long text. This test involved the matching thresholds $k = 1$ and $k = m-1$ to see what kind of effect the value of k has. In the second we tested aligning patterns and texts that have the same length, and this time we used only $k = m - 1$. The results are shown in Fig. 3 (left and right, respectively). In them our algorithm is observed to be 1.2 - 8.5 times faster than the basic dynamic programming algorithm when $w = 64$.

5 Conclusions

We have presented the first bit-parallel algorithm to compute local similarity score between two strings, which has many practical applications in computational biology. While dynamic programming, the only existing algorithm, takes time $O(mn)$ (m and n being the lengths of the strings), our algorithm needs time $O(mn \log \min(m,n,w)/w)$ using a computer word of w bits. Our experiments show up to 8-fold speedups.

Our algorithm cannot replace dynamic programming because it cannot handle prize and penalty values other than ± 1. However, it can be used as a fast filter

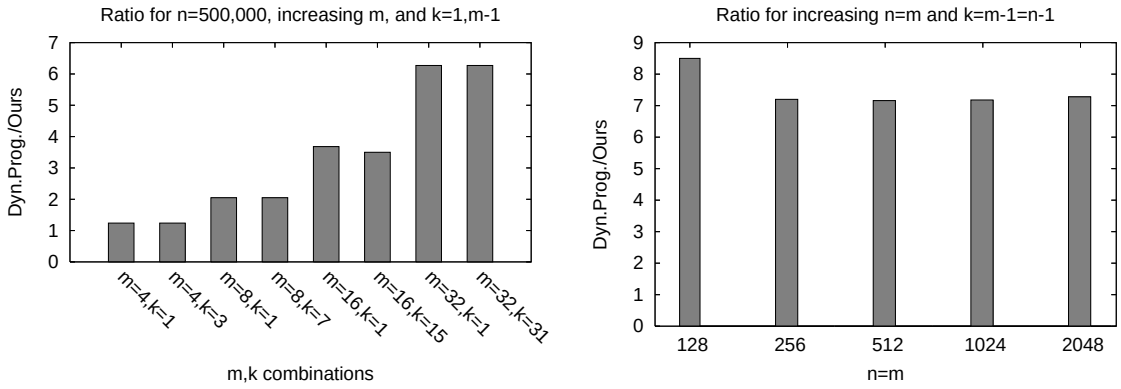

Figure 3: Speedup factor of our bit-parallel algorithm over the basic dynamic programming algorithm. On the left, aligning long against short strings. On the right, aligning strings of the same length.

to discard most of the areas and let the dynamic programming algorithm concentrate only on the areas that look promising. Moreover, there are some DNA-related applications where they use precisely those ± 1 penalties.

As future research issues, the most immediate is to investigate whether it is possible to "pack" the logical conditions describing the differences across the diverse directions in a different way that makes the overall formula faster to compute. Longer-term goals are accomodating other cost functions apart from the unitary-cost one, and trying to obtain optimal speedup, removing the term $O(\log \min(m, n, w))$ from the cost formula.

References

[1] A. Bergeron and S. Hamel. Vector algorithms for approximate string matching. *International Journal of Foundations of Computer Science*, 13(1):53–65, 2002.

[2] M. Crochemore, G. Landau, and M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted scoring matrices. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, pages 679–688, 2002.

[3] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[4] H. Hyyrö. Explaining and extending the bit-parallel approximate string matching algorithm of Myers. Technical Report A-2001-10, Dept. of Computer and Information Sciences, University of Tampere, Tampere, Finland, 2001.

[5] H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proc. 13th Combinatorial Pattern Matching (CPM'02)*, LNCS 2373, pages 203–224, 2002.

[6] H. Hyyrö and G. Navarro. Bit-parallel witnesses and their applications to approximate string matching. *Algorithmica*, 41(3):203–231, 2005.

[7] W. Masek and M. Paterson. A faster algorithm for computing string edit distances. *J. of Computer and System Sciences*, 20:18–31, 1980.

[8] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.

[9] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[10] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical online search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[11] S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.