A Bit-Parallel, General Integer-Scoring Sequence Alignment Algorithm

Gary Benson[1,2,3⋆], Yozen Hernandez[1,2], and Joshua Loving[1,2]

[1] Laboratory for Biocomputing and Informatics, Boston University, Boston, MA
[2] Graduate Program in Bioinformatics, Boston University, Boston, MA
[3] Department of Computer Science, Boston University, Boston, MA
gbenson@bu.edu, yhernand@bu.edu, jloving@bu.edu

Abstract. Mapping of next-generation sequencing data and other processor-intensive sequence comparison applications have motivated a continued search for high efficiency sequence alignment algorithms. In one approach, which exploits the inherent parallelism in computer logic calculations, individual cells in an alignment scoring matrix are represented as bits in a computer word and the calculation of scores is emulated by a series of bit operations comprised of AND, OR, XOR, complement, shift, and addition. Bit-parallelism has been successfully applied to the Longest Common Subsequence (LCS) and edit-distance problems, producing solutions which are significantly faster than standard implementations. But, the intensive mental effort required to produce these solutions, which are closely tied to special properties of the problems, has limited efforts to extend bit-parallelism to more general scoring schemes. In this paper, we give the **first bit-parallel solution for general, integer-scoring global alignment**. Integer-scoring schemes, which are widely used, assign integer weights for match, mismatch, and insertion/deletion or indel. Our method depends on structural properties of the relationship between adjacent scores in the scoring matrix. We utilize these properties to construct a class of efficient algorithms, each designed for a particular set of weights, and we introduce a standard for characterizing the efficiency in terms of the average number of bit-operations per cell of the original scoring matrix.

Keywords: bit-parallelism, global sequence alignment, integer weights

1 Introduction

Sequence alignment algorithms are critical tools in the analysis of biological sequence data including DNA, RNA, and protein sequences. But, the demands placed on computational resources by high-throughput experiments such as next-generation sequencing require new, more efficient methodologies. While standard

⋆ This work was supported in part by NSF grants IIS-1017621 and DRL-1020166 (GB), NIH grant 1UL1 RR025771 (GB), and NSF IGERT grant DGE-0654108 (JL and YH).

implementations of the Smith-Waterman [11] and Needleman-Wunsch [10] algorithms calculate the score in each cell of the alignment scoring matrix sequentially, a newer technique called bit-parallelism adapts the inherent parallelism in computer logic calculations to the task of overcoming the limited dependencies between adjacent scores in order to achieve much higher efficiencies.

Bit-parallel algorithms use computer words to represent multiple adjacent cells in the scoring matrix, and bit operations to mimic the result of dynamic programming. Bit-parallel methods have been successfully applied to the longest common subsequence (LCS) [1, 3, 5] and unit-cost edit distance (Levenshtein) [6, 12, 8] problems. These algorithms focus on computing the alignment score, delinking that computation from the traceback which produces the final alignment. In the LCS scoring matrix, scores are monotonically non-decreasing in the rows and columns and the bit-parallel implementations use bits to represent the cells where an increase occurs. In the edit distance scoring matrix, adjacent scores can differ by at most one, and the binary representation stores the locations of (two of the three) possible differences, $+1, -1$, and zero. These algorithms are adhoc in their approach, relying on specific properties of the underlying problems, making it difficult to directly adapt them to other alignment scoring schemes.

Bit-parallel algorithms have also been developed for the approximate string matching problem in which a pattern and text are given and occurrences of the pattern with at most k differences are sought in the text [13, 12, 2, 9]. For example, the Wu and Manber algorithm [12] finds approximate matches to a pattern or regular expression where the number of differences between the pattern and the text is at most k. This algorithm is implemented as the Unix command *agrep*. The Navarro algorithm for approximate regular expressions [9] allows arbitrary integer weights for match, mismatch, and insertion or deletion and finds occurrences of the pattern where the *sum* of the edit weights is at most k. In these algorithms, the complexity (and computation time) increases with increasing k. By contrast, in our algorithm discussed below, the complexity depends on the edit weights, not the ultimate score of the alignment.

In this paper, we describe, to our knowledge, the first generalization of the bit-parallel method to integer-scoring similarity and distance based global alignment. Integer-scoring schemes, which are widely used, assign integer weights for match, mismatch, and indel operations. Our new contribution is an observation of the regularity of the relationship between adjacent scores in the scoring matrix when using general integer scoring (Section 3) and the design of an efficient series of bit operations to exploit that regularity (Section 4). We show how to construct a class of efficient algorithms, each designed for a particular set of weights. The method works, as described below, for general alphabets, but our interest derives from frequent use of DNA alignment when analyzing next-generation sequencing data to detect genetic variation. The remainder of the paper is organized as follows. In Section 2 we give a formal presentation of the problem, in Section 5 we compare the performance of our algorithm with five related algorithms, and in Section 6 we discuss future work.

2 Problem Description

We state our problem in terms of similarity scoring, but the technique can be used for distance scoring as well.

Problem: *Given two sequences X and Y, of length n and m respectively, and a similarity scoring function S defined by three* **integer** *weights M, I, G (match, mismatch, indel or gap), calculate the global alignment similarity score for X and Y using bit operations with computer words of length w in time $O(nm/w)$, and more specifically, such that the actual average count of bit operations* **per cell** *of the alignment scoring matrix, p/w, is $\leq e$ for some small number e, where p is the number of operations to complete the calculation for w cells.*

We will say that an algorithm (or program) that accomplishes the task has a **per-cell bit operation cost** of at most e. For example, in the case of the LCS, the per cell bit operation cost is $p/w = 1/16$ (that is, there are $p = 4$ bit operations per word of length $w = 64$) [5]. For the edit distance problem, $p/w = 15/64 < 1/4$ (15 bit operations per word, unpublished, improved from [8, 6]). Note that in these examples we have counted only bit operations and not storage of computed values in program variables. Adding store operations is more accurate and increases the numbers here, but stores are difficult to count because they depend on specifics of the compiler and the level of optimization.

We require that the alignment method be global, but do not restrict the initializations in the first row or column of the alignment scoring matrix. Typical initializations require 1) a gap weight to be added successively to every cell (global alignment from the beginning of a sequence), and 2) a zero in every cell (global alignment where an initial gap has no penalty).

We assume that match scores are positive, $M > 0$, mismatch and gap scores are negative, $I, G < 0$ and that the use of mismatch is possible, meaning that its penalty is no worse than the penalty for two adjacent gaps, one in each sequence, $I \geq 2G$. While other weightings are possible, they either reduce to simpler problems from a bit-parallel perspective (i.e., Longest Common Subsequence has $G = 0,\ I = -\infty,\ M = 1$) or require more complicated structures than detailed here (protein alignment using PAM or BLOSUM style amino acid substitution tables).

We stress that we do not claim to have an optimal solution for any particular instance of the alignment weights. As can be seen in the cases of the LCS and edit-distance, extreme efficiency can be obtained by exploiting specific problem properties. Instead, we give a general framework for efficient bit-parallel implementation of alignment which works across a wide spectrum of weights.

3 Function Tables

Let S be a recursively-defined, similarity scoring function for computing the global alignment score between two sequences X and Y:

$$S[i,j] = \max \begin{cases} S[i-1,j-1] + M & \text{if } X_i = Y_j \\ S[i-1,j-1] + I & \text{if } X_i \neq Y_j \\ S[i-1,j] + G & \text{delete } X_i \\ S[i,j-1] + G & \text{delete } Y_j \end{cases}$$

We assume the convention that S is computed in an alignment scoring matrix. Suppose that instead of knowing the actual value in a cell $S[i,j]$ we know only the *difference*, ΔV, between that cell and the cell above, and the *difference*, ΔH, between that cell and the cell to it's left:

$$\begin{aligned} \Delta V[i,j] &= S[i,j] - S[i-1,j] \\ \Delta H[i,j] &= S[i,j] - S[i,j-1]. \end{aligned}$$

Lemma 1 defines the minimum and maximum values of ΔV and ΔH and Lemma 2 gives their recursive definitions. Proofs are omitted.

Lemma 1. *Given S, X, and Y as described above where match score $M > 0$, mismatch score $I < 0$ and gap (indel) score $G < 0$, the minimum and maximum differences between adjacent values in the same row (i.e., $\Delta H[i,j]$) or column (i.e., $\Delta V[i,j]$) are G and $M - G$.*

Lemma 2. *The values for ΔV are shown below and the values for ΔH are the transpose, that is $\Delta H[i,j] = \Delta V[j,i]$.*

$$\underset{\forall i,j \geq 1}{\Delta V[i,j]} = \begin{cases} \text{Score comes diagonally from a match:} \\ M - \Delta H[i-1,j] \qquad \textit{if } X_i = Y_j \\ \\ \text{Score comes diagonally from a mismatch:} \\ I - \Delta H[i-1,j] \qquad \textit{if } I - G \geq \begin{cases} \Delta H[i-1,j] \\ \Delta V[i,j-1] \end{cases} \\ \text{Score comes from the cell above:} \\ G \qquad \textit{if } \Delta H[i-1,j] \geq \begin{cases} I - G \\ \Delta V[i,j-1] \end{cases} \\ \text{Score comes from the cell to the left:} \\ \Delta V[i,j-1] + G - \Delta H[i-1,j] \qquad \textit{if } \Delta V[i,j-1] \geq \begin{cases} I - G \\ \Delta H[i-1,j] \end{cases} \end{cases}$$

$$\left(\underset{\forall j \geq 1}{V[0,j]} = G \text{ or } \underset{\forall j \geq 1}{V[0,j]} = 0 \right) \text{ and } \left(\underset{\forall i \geq 1}{H[i,0]} = G \text{ or } \underset{\forall i \geq 1}{H[i,0]} = 0 \right).$$

The recursion for the ΔV values can be summarized in a Function Table (Figure 1). Note the key value $I - G$ from the recursion and the relation $\Delta H = \Delta V$. They set the boundaries for the marked zones in the table. These zones comprise $(\Delta V, \Delta H)$ value pairs which determine how the best score of a cell in S is obtained in the absence of a match, either as an indel from the left (Zones A and B), a mismatch (Zone C), or an indel from above (Zone D). Borders between zones, indicated by dotted lines, yield ties for the best score. Figure 2 shows how the relative size of the Zones changes with changes in I and G.

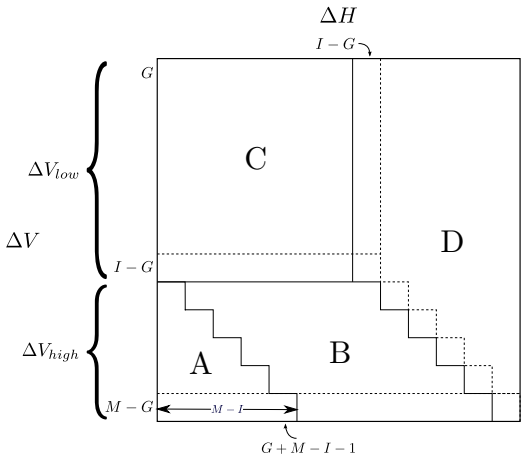

Fig. 1. Zones in the Function Table for ΔV

We use the following (see Figure 1):

Definitions:
$\Delta V_{\min} = \Delta H_{\min} = G$
$\Delta V_{\max} = \Delta H_{\max} = M - G$
$\Delta V_{low}, \Delta H_{low} \in [G, I - G]$
$\Delta V_{high}, \Delta H_{high} \in [I - G + 1, M - G]$

Observations:
Zone A – All value are in V_{high}
Zone B – All values are in V_{low}
Zone C – All values are in V_{low}. Values depend only on ΔH.
Zone D – All values are G
Last Row – Values from this row also apply when there is a Match.
First Column – Identity column for values in V_{high}

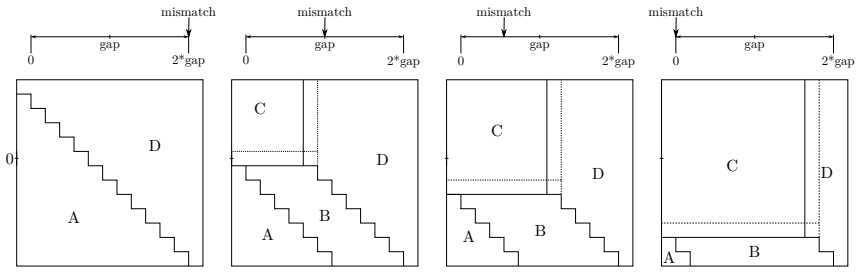

Fig. 2. Relative size of Zones as I (mismatch penalty) decreases from $2G$ (twice gap penalty) where there is no preference for mismatches, to zero, where mismatches are free and gaps are introduced only to obtain matches.

4 Bit-Parallel Alignment

Our goal is to develop an algorithm for calculating the ΔH values in row i from:

- the ΔH values in row $i-1$,
- the initial ΔV value in row i
- the Match positions in row i.

What follows is a description of the simplest case where the length of the first sequence, n, is less than the computer word size w. Longer sequences can be handled in "chunks," where each chunk has size w. The Match positions for every row are computed prior to the calculation of the row values as is done for the LCS and edit-distance problems. Details are given at the end. For the remainder of the paper, we will use the following set of scoring weights for illustration:

$$M = 2,\ I = -3,\ G = -5.$$

The ΔV Function Table for these weights is shown in Figure 3.

Representation of ΔH and ΔV values. In the bit-parallel framework, we use one computer word (sometimes referred to later as a vector) to represent each possible value of ΔH and ΔV. Bit i in a word refers to column i in the alignment scoring matrix. With the weights used for illustration, there are 13 values, each, for ΔH and ΔV.

Algorithm Outline. We first calculate all ΔV values for row i and then use them to calculate the ΔH values. Close inspection of the Function Table (Figure 3) reveals that the values in Zone A, which are all in ΔV_{high}, are interdependent, and require computing in order from high to low. The other complication is the identity column ΔH_{min} which requires carrying a ΔV input value through runs of ΔH_{min}. Values in Zones B, C, and D, which are all in ΔV_{low}, can be computed once the values from Zone A are obtained. The main work is combining outputs from $(\Delta V, \Delta H)$ pairs which intersect along the same diagonal. Once all ΔV values are available, the ΔH values can be computed in any order.

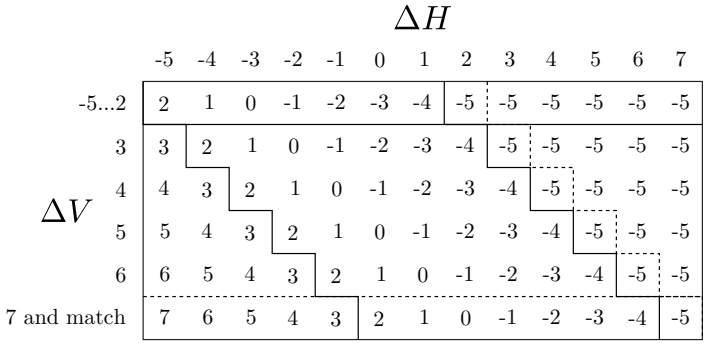

Fig. 3. The ΔV Function Table for the weights $M = 2,\ I = -3,\ G = -5$. Note that $\Delta V_{high}, \Delta H_{high} \in [3, 7]$; $\Delta V_{low}, \Delta H_{low} \in [-5, 2]$; $\Delta V_{\min} = \Delta H_{\min} = -5$; $\Delta V_{\max} = \Delta H_{\max} = 7$.

4.1 Computing ΔV

We present the following without theorems or proofs for compactness.

Zone A – Dependencies. To compute its output value, each cell needs to know its ΔH and ΔV input values. The ΔH values are already known, as is the input ΔV value for the first cell. As in standard left to right processing, the output ΔV value from one cell becomes the input value for the cell to its right. The arrangement of the ΔV values in Zone A of Figure 3 indicates a chain of dependency:

$$Matches \rightarrow 7 \rightarrow 6 \rightarrow \ldots \rightarrow 3.$$

Additionally, the identity column $\Delta H_{\min}$ $(= -5)$ indicates that a ΔV_{high} value that is fed into a ΔH run of -5s will yield the identical output for every cell in the run. Therefore, to know where the 7s are (for ΔV) first requires knowing where the Matches are and then which of the 7s from Matches carry through runs of -5; to know where the 6s are first requires knowing where the Matches and 7s are and then which of those 6s carry through runs of -5, etc. The "carry through runs of $\Delta H_{\min}$" is really the only obstacle, but can be accomplished with an addition (`+`) as seen below. Addition is also used to solve similar left to right dependency problems in the LCS and edit-distance bit-parallel algorithms.

Zone A – Finding $\Delta V_{\max}$. The vector is calculated with four operations (Figure 4). The result stores locations of $\Delta V_{\max}$ shifted one position to the right for input to subsequent calculations. The operations are 1) an `AND` to find the 7s from Matches, 2) an addition (`+`) to carry through adjacent runs of $\Delta H_{\min}$ and into the position following a run (and causing erroneous internal bit flips if there are multiple Matches in the same run), 3) an `XOR` to complement the bits within the runs, and 4) an `XOR` to correct any erroneous bits and accomplish the shift by removing the leading 1 in a run.

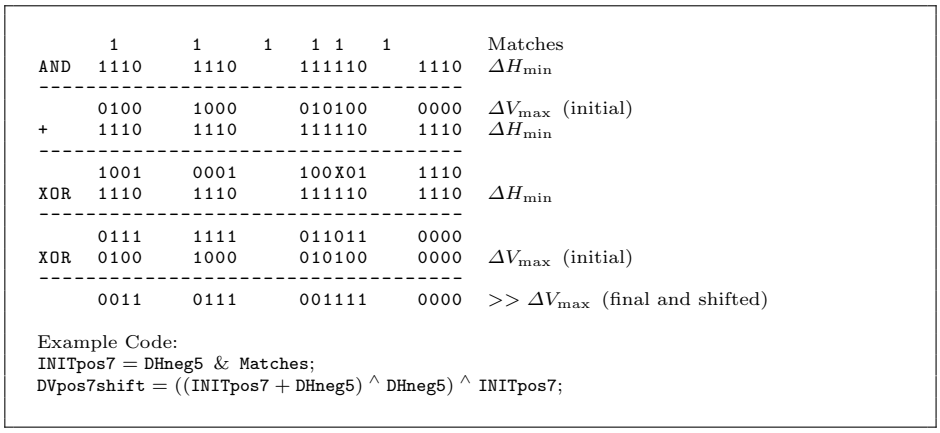

Fig. 4. Finding $\Delta V_{\max}$. Each line represents a computer word with low order bit, corresponding to the first position in a sequence, on the left. 1s are shown explicitly, 0s are only shown to fill runs of $\Delta H_{\min}$ and the first position to the right of each run. Symbol >> indicates that the final $\Delta V_{\max}$ values are shifted to the right one position. Erroneous bit set by the ADD (+) is marked X.

Zone A – Others. Remaining ΔV_{high} vectors are calculated, in descending order as discussed above. First, initial vectors are computed by `AND` of appropriate $(\Delta V, \Delta H)$ pairs (which intersect along a common diagonal in the Function Table) and collected together with `OR`s. Second, the intial vectors are shifted right one position for subsequent calculations. Third, the carry through runs of $\Delta H_{\min}$ is computed in two operations (Figure 5), an addition (`+`) as before and an `XOR` to complement the bits within the runs. Before the carry operation, those $\Delta H_{\min}$ positions that have already output a $\Delta V_{\max}$ value must be removed. Note that initial ΔV values, when shifted to the right, can only occur at the leftmost position of a $\Delta H_{\min}$ run, and not at the single bit between adjacent runs.

Zones B and C and D. (Figure 6). At this point, all the ΔV_{high} input values for Zone B have been computed, remaining output values are all ΔV_{low}, and Zone C output depends only on ΔH values. Each output vector is an `OR` combination of 1) Zone B – the `AND` of appropriate $(\Delta V, \Delta H)$ pairs, which intersect along a common diagonal, and 2) Zone C – the `AND` of the appropriate ΔH vector and all positions without a ΔV_{high} output. The result is shifted one position to the right for subsequent calculations. Zone D has only one output value, ΔV_{min}. It is assigned to all remaining positions as well as the first position if gap penalty in the first column is being used.

4.2 Computing ΔH

After the ΔV values are computed, there are no longer any dependencies. All the new vectors for ΔH can be immediately computed. Again, each vector is an `OR` combination of the `AND` of appropriate pairs of ΔH and ΔV values.

```
      1110     1110 11101110  ΔH_min (remaining)
+     1     1  1     1  X      >> ΔV (initial shifted)
--------------------------
      0001 1   0001 00011110
XOR   1110     1110 11101110  ΔH_min (remaining)
--------------------------
      1111 1   1111 11110000  >> ΔV (final and shifted)
```

Example Code:

```
RemainDHneg5 = DHneg5 ^ (DVpos7shift >> 1);
INITpos3s = (DHneg1 & DVpos7shiftorMatch)|(DHneg2 & DVpos6shiftNotMatch)|
            (DHneg3 & DVpos5shiftNotMatch)|(DHneg4 & DVpos4shiftNotMatch);
DVpos3shift = ((INITpos3s << 1) + RemainDHneg5) ^ RemainDHneg5;
DVpos3shiftNotMatch = DVpos3shift & NotMatches;
```

Fig. 5. Carry through runs of $\Delta H_{\min}$ for remaining values in ΔV_{high}. Symbol `X` marks a single position between runs which cannot be 1 in the initial shifted values.

4.3 Other Tasks

Determining Matches. Prior to computing any ΔH or ΔV, the position of the matches are determined for each character σ in the sequence alphabet Σ. A bit vector $Match_\sigma$ records those positions in sequence X where σ occurs. Filling all the $Match_\sigma$ simultaneously can be accomplished efficiently in a single pass through X. For row i of the ΔH and ΔV calculations, the $Match$ vector for character Y_i is used.

Decoding the Alignment Score. The score in the last column of the last row of the alignment scoring matrix can be obtained by calculating the score in the zero column ($= m * G$) and then adding the number of 1 bits in each of the ΔH vectors multiplied by the value of the vector. Using the method described

Example Code Zones B and C:

```
DVnot7to3shiftorMatch = ~ (DVpos7shiftorMatch|DVpos6shift|DVpos5shift|DVpos4shift|
                DVpos3shift);
DVpos2shift = ((DHzero & DVpos7shiftorMatch)|(DHneg1 & DVpos6shiftNotMatch)|
                (DHneg2 & DVpos5shiftNotMatch)|(DHneg3 & DVpos4shiftNotMatch)|
                (DHneg4 & DVpos3shiftNotMatch)|(DHneg5 & DVnot7to3shiftorMatch)) << 1;
```

Example Code Zone D:

```
DVneg5shift = all_ones ^ (DVpos7shift|DVpos6shift|DVpos5shift|DVpos4shift|DVpos3shift|
                DVpos2shift|DVpos1shift|DVzeroshift|DVneg1shift|DVneg2shift|DVneg3shift|
                DVneg4shift);
```

Fig. 6. Code for Zones B and C and D.

in [7], this takes $O(n + M - 2G)$ operations with a small constant:

$$S[m, n] = m * G + \sum_{i=G}^{M-G} \text{bits}_i * i.$$

Several methods can be used to efficiently find all scores in the last row. Discussion of these is omitted due to space limitations.

4.4 Complexity and Number of Operations

The time complexity of our algorithm is $O(znm/w)$ where z depends on the combined size of the Zones A, B, and C (the latter is reduced to a single row as in Figure 3) in the Function Table. This in turn depends on the alignment weights M, I, and G:

$$z = \frac{(M - 2G + 1)^2 - (I - 2G)^2}{2}$$

and the constant hidden in the big O notation is approximately 4 (dominated by two operations per cell of Zones A, B, and C for ΔV and separately for ΔH). For the example weights used in this paper, the number of bit operations per w cells of the scoring matrix is 265 yielding a per-cell bit operation cost of $265/64 \approx 4.2$.

5 Experimental Results

We compared running times for several related bit-parallel algorithms: 1) BHL – our new algorithm with 5 sets of alignment weights to show the effect of the weights on the running time, 2) NW – the classical Needleman-Wunch [10] dynamic programming alignment algorithm, 3) LCS – the bit-parallel LCS algorithm of [5], 4) ED – a bit-parallel, unit-cost edit-distance algorithm, improved from [8, 6], 5) WM – the unit-cost Wu-Manber approximate pattern matching algorithm [12], and 6) N – the Navarro, general integer scoring, approximate regular expression matching algorithm [9]. We implemented BHL, NW, LCS, ED, and WM. N was graciously provided by Gonzalo Navarro.

For all experiments, we used human DNA and ran 100 pattern sequences against 250,000 text sequences for a total of 25 million alignments. (Pattern and text distinctions are irrelevant for BHL, NW, LCS, and ED.) All sequences were 63 characters long. For WM we varied k, the maximum number of allowed errors, from 1 to 15. For N, we varied k from 1 to 12. (Additionally, we selected the smallest values for *tables* and *mask length*, internal parameters to the N method, for which the program could be successfully run. For $k = 1$, we used $tables = 8$ and $mask\ length = 4$. For $k = 2$ through 5, $tables = 9$ and $mask\ length = 5$. For $k = 6$ through 12, $tables = 13$ and $mask\ length = 7$.)

All programs were compiled with GCC using optimization level O3 and were run on an Intel Core 2 Duo E8400 3.0 GHz CPU running Ubuntu Linux 12.10. Results are shown in Figure 7.

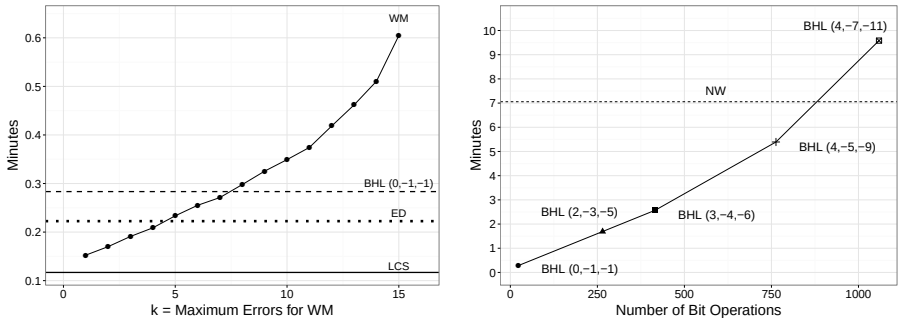

Fig. 7. Running times. For BHL, alignment weights (M, I, G) are shown in parenthesis. All times are averages of three runs. **Left.** Unit-cost BHL, unit-cost WM, LCS, and ED. k is the maximum number of errors allowed for WM. k is not a parameter for the other algorithms and their times are shown as horizontal lines. LCS uses 4 bit operations per w cells, ED uses 15 bit operations, BHL (0, -1, -1) uses 23 bit operations. For $k = 7$, the times for BHL and WM are nearly the same. By $k = 15$, BHL is approximately twice as fast. Results for N are not shown on the graph due to the much longer running time. For N, we ran 2 sequences against 250,000 and multiplied by 50 to find comparable times. For values of k from 1 to 12, the performance of N varied from 20 to 50 times slower than BHL (2, -3, -5). **Right.** Variants of BHL and NW. For BHL, time is approximately linearly proportional to the number of bit operations (and z) as explained in Section 4.4. For NW, the number of bit operations is not available. Its time is shown as a horizontal line. BHL (2, -3, -5) is approximately 4.2 times faster than NW and BHL (0,-1,-1) is approximately 24.9 times faster.

6 Discussion

The algorithm outlined above can be extended in several ways. Computers now in common usage have a word size of $w = 64$ bits. A straightforward extension is to use the 128 bit SIMD registers (Single Instruction, Multiple Data). This essentially halves the number of operations per cell (with the addition of several bookkeeping operations) and doubles the speed of computation. Details of the method will be given in the journal version of this abstract. Another extension is due to the unexploited parallelism of the operations. There are no dependencies on prior computations after the ΔV vectors in Zone A are computed. This means that all the computations in Zones B, C, and D for ΔV and all the subsequent computations for ΔH can be computed simultaneously, an ideal situation for the use of general purpose graphical processing units (GPGPU). CUDA programming (Compute Unified Device Architecture) for this method will be presented in a separate paper.

Because a different program has to be created for each unique set of weights for M, I, and G, adoption of this algorithm would require a complete understanding of the program structure. To simplify usage, we are constructing a web site that will generate C computer code given the weights as input.

This method has already been used to accelerate software for detecting tandem repeat variants in next-generation sequencing reads [4] and is well suited to other DNA sequence comparison tasks that involve computing many alignments.

References

1. L. Allison and T. I. Dix. A bit-string longest-common-subsequence algorithm. *Information Processing Letters*, 23(5):305–310, 1986.
2. Anne Bergeron and Sylvie Hamel. Vector algorithms for approximate string matching. *International Journal of Foundations of Computer Science*, 13(01):53–65, February 2002.
3. M. Crochemore, C.S. Iliopoulos, Y.J. Pinzon, and J.F. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80(6):279–285, 2001.
4. Y. Gelfand, J. Loving, Y. Hernandez, and G. Benson. VNTRseek – A Computational Pipeline to Detect Tandem Repeat Variants in Next-Generation Sequencing Data: Analysis of the 454 Watson Genome. In *Proc. of RECOMB-seq: The Third Annual RECOMB Satellite Workshop on Massively Parallel Sequencing*, 2013. To appear.
5. H. Hyyrö. Bit-parallel LCS-length computation revisited. In *Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA 2004)*, 2004.
6. H. Hyyrö, K. Fredriksson, and G. Navarro. Increased bit-parallelism for approximate and multiple string matching. *Journal of Experimental Algorithmics (JEA)*, 10:2–6, 2005.
7. B.W. Kernighan and D.M. Ritchie. *The C programming language*. Prentice Hall, 2nd edition edition, 1988.
8. G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415, 1999.
9. Gonzalo Navarro. Approximate regular expression searching with arbitrary integer weights. *Nordic Journal of Computing*, 11(4):356–373, December 2004.
10. S. Needleman and C. Wunch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.
11. T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.
12. S. Wu and U. Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.
13. Sun Wu, U. Manber, and G. Myers. A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, January 1996.