Bit-parallel and SIMD alignment algorithms for biological sequence analysis

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

AND

COLLEGE OF ENGINEERING

Dissertation

BIT-PARALLEL AND SIMD ALIGNMENT ALGORITHMS FOR BIOLOGICAL SEQUENCE ANALYSIS

by

JOSHUA LOVING

B.S., B.A., University of Hawaii at Hilo, 2011
M.S., Boston University, 2014

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2017

Approved by

First Reader

Gary Benson, PhD
Associate Professor of Biology and Computer Science

Second Reader

Michal Ziv-Ukelson, PhD
Associate Professor of Computer Science, Ben Gurion University of the Negev

Acknowledgments

To my advisor, Dr. Gary Benson - thank you for your mentorship, your encouragement, your insight and critical feedback, and the opportunity to work in your lab.

To my PhD thesis committee - thank you for your input and your helpful criticisms, they helped me shape my research into what it is now.

To my fellow labmates, Yozen and Yevgeniy - thank you for letting me bounce ideas off of you, the technical advice, and the company. Yozen, I already miss sharing a cubicle with you.

To my friends in the Bioinformatics Program, bubify, Ania, Yozen, Gracia, Beth, George, Vinay, and Rachael - thank you for being there for me. I'll never forget the long hours in the cubes working on computational biology homework.

To Dan and Nacho - thank you for all the advice as I wrote my dissertation and hunted for a job. You guys helped keep my stress levels manageable.

To the staff and faculty of the Bioinformatics Program, in particular Tom Tullius, Scott Mohr, Caroline Lyman, Johanna Vasquez, David King, and Mary Ellen Fitzpatrick - thank you for answering all the questions I have had along the way and for all the reminders to keep up with deadlines. Also, thanks for the work you do that makes the Bioinformatics Program such a welcoming and happy place to be.

To my professors at the University of Hawaii at Hilo, especially Dr. Efren Ruiz, Dr. Keith Edwards, Dr. Michael Peterson, Dr. Judith Gersting, and Karla Hayashi - thank you for encouraging me to pursue research as an undergraduate and for continuing to support me throughout my graduate school career.

To my mentor and co-mentor at Caltech, Erik Winfree and Joseph Schaeffer - thank you for giving me the opportunity to get involved with biomolecular computation. Without my time in the lab there, I might very well never have gotten involved in bioinformatics. Everything I know about Python I learned from you, Joseph.

To my siblings - the distance has been difficult during graduate school. Thank you for all the letters, pictures, and cards. I love you all.

To my parents - thank you for encouraging my inquisitiveness and giving me the strong foundation that made my educational journey possible. I love you.

To my wife, Katherine Loving - thank you for putting up with the long hours and shouldering more responsibilities as I have wrapped up my PhD. I love and appreciate you more than I could ever say.

To my son, Micah Loving - without you, this dissertation may have been submitted six months ago, but it would have meant much less. I love you son.

BIT-PARALLEL AND SIMD ALIGNMENT ALGORITHMS FOR BIOLOGICAL SEQUENCE ANALYSIS

JOSHUA LOVING

Boston University, Graduate School of Arts and Sciences

And

College of Engineering, 2017

Major Professor: Gary Benson, PhD
Associate Professor
Department of Biology
Department of Computer Science
Program in Bioinformatics

ABSTRACT

High-throughput next-generation sequencing techniques have hugely decreased the cost and increased the speed of sequencing, resulting in an explosion of sequencing data. This motivates the development of high-efficiency sequence alignment algorithms. In this thesis, I present multiple bit-parallel and Single Instruction Multiple Data (SIMD) algorithms that greatly accelerate the processing of biological sequences. The first chapter describes the BitPAl bit-parallel algorithms for global alignment with general integer scoring, which assigns integer weights for match, mismatch, and insertion/deletion. The bit-parallel approach represents individual cells in an alignment scoring matrix as bits in computer words and emulates the calculation of scores by a series of logic operations. Bit-parallelism has previously been applied to other pattern matching problems, producing fast algorithms. In timed tests, we show that BitPAl runs 7 - 25 times faster than a standard iterative algorithm.

The second part involves two approaches to alignment with substitution scoring, which assigns a potentially different substitution weight to every pair of alphabet characters, better representing the relative rates of different mutations. The first approach extends the existing BitPAl method. The second approach is a new SIMD algorithm that uses partial sums of adjacent score differences. I present a simple partial sum method as well as one that uses parallel scan for additional acceleration. Results demonstrate that these algorithms are significantly faster than existing SIMD dynamic programming algorithms.

Finally, I describe two extensions to the partial sums algorithm. The first adds support for affine gap penalty scoring. Affine gap scoring represents the biological likelihood that it is more likely for gaps to be continuous than to be distributed throughout a region by introducing a gap opening penalty and a gap extension penalty. The second extension is an algorithm that uses the partial sums method to calculate the tandem alignment of a pattern against a text sequence using a single pattern copy.

Next generation sequencing data provides a wealth of information to researchers. Extracting that information in a timely manner increases the utility and practicality of sequence analysis algorithms. This thesis presents a family of algorithms which provide alignment scores in less time than previous algorithms.

Contents

1 Introduction and Background 1

1.1 Background . . . 2

1.1.1 Global Alignment . . . 3

1.1.2 Bit-parallel Algorithms . . . 5

1.1.3 SIMD Algorithms . . . 5

1.2 Thesis Overview . . . 6

1.2.1 BitPAl, bit-parallel algorithms for global alignment with general integer scoring . . . 6

1.2.2 Bit-parallel and SIMD algorithms for global alignment with substitution scoring . . . 7

1.2.3 SIMD algorithms for global alignment with affine gapping and tandem alignment . . . 7

2 BitPAl: a bit-parallel, general integer-scoring sequence alignment algorithm 8

2.1 Introduction . . . 8

2.2 Methods . . . 10

2.2.1 Function Tables . . . 11

2.3 Algorithm . . . 13

2.3.1 BitPAl . . . 16

2.3.2 BitPAl Packed . . . 18

2.3.3 Other Tasks . . . 20

2.3.4 Backtrace to Recover Alignment 21

2.3.5 Complexity and Number of Operations 22

2.4 Experimental Results 23

2.5 Discussion 24

3 Bit-parallel and SIMD Methods for Global Alignment with Substitution Scoring 35

3.1 Introduction 35

3.2 Problem Description 36

3.3 Algorithms 40

3.3.1 Data Structures 40

3.3.2 **BitPAl Extension for BLOSUM-type scoring.** 42

3.3.3 **SIMDParSums: Method of Partial Sums of ΔH** 43

3.3.4 **Method of Partial Sums by *Striped* Scan** 46

3.4 **Complexity and Space** 48

3.4.1 BitPAl Extension Method 48

3.4.2 SIMDParSum 48

3.4.3 SIMDScan 49

3.5 Experimental Results 49

3.6 Discussion 51

4 Affine Gap and Tandem Alignment 58

4.1 Introduction 58

4.2 Affine Gap 59

4.2.1 Problem Definition, Affine Gap 59

4.2.2 Definitions and notation, Affine Gap 60

4.2.3 Algorithm: Affine Gap 61

4.3 Tandem Alignment 63

4.3.1 Problem Description, Tandem Alignment 64

4.3.2 Definitions and notation, Tandem Alignment 65

4.3.3 Algorithm, Tandem Alignment 67

4.4 Complexity and Space 69

4.5 Experimental Results 71

4.5.1 Affine Gap Results 71

4.5.2 Tandem Alignment Results 72

4.6 Discussion 73

5 Conclusion **76**

5.1 Discussion 76

5.2 Future Work 77

A Proofs for BitPAl **79**

B Proofs on Substitution Scoring **88**

C Proofs on Global Alignment with Partial Sums by Scan **99**

C.1 Introduction 99

D Proofs on Global Alignment with Affine Gap Scoring **112**

D.1 Definitions 112

D.2 Theorems and Proofs 113

E Proofs on SIMD Tandem Alignment **128**

E.1 Preliminaries 128

References **139**

List of Tables

2.1 Table of run times in minutes. Shown are averages over three trials for 25 million alignments. Needleman-Wunsch has the same runtime for all parameters, 7.056056 minutes. 25

2.2 Table of the number of operations in the main loop of BitPAl and BitPAl Packed for various alignment parameters (M, I, G). 25

3.1 Tables of run times, in minutes, for 25 million alignments. **Top:** A different pair of sequences was used for each alignment (1 to 1). **Bottom:** Each Y sequence was aligned against 100 X sequences (1 to 100). Note that both tables include run times for the SIMDParSumAffine algorithm introduced in Chapter 4. 50

4.1 Tables of run times, in minutes, for 250 thousand alignments. A different pair of sequences was used for each alignment (1 to 1). 72

List of Figures

1·1 In this figure, we represent a tandem repeat sequence as the pattern sequence (in blue), with the repeated copies below it in red. Anywhere the copy varies from the pattern, we show the base that varies. Flanking sequence is shown in green. 3

1·2 Two tandem repeats at the same locus, where one has just over 3 copies and one has just over 5. 4

2·1 Zones in the Function Table for ΔV. Zone A: All values are in $\Delta V_{\text{high}} \in \{I - G + 1, \dots, M - G\}$; Zone B: All values are in $\Delta V_{\text{low}} \in \{G, \dots,\ I - G\}$; Zone C: All values are in ΔV_{low} and values depend only on ΔH; Zone D: All values are G; Last Row: Values also apply when there is a Match;. First Column: Identity column for values in ΔV_{high}. 13

2·2 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. 27

2·3 The ΔV Function Table for the weights $M = 2,\ I = -3,\ G = -5$. Note that $\Delta V_{\text{high}}, \Delta H_{\text{high}} \in [3, 7]$; $\Delta V_{\text{low}}, \Delta H_{\text{low}} \in [-5, 2]$; $\Delta V_{\text{min}} = \Delta H_{\text{min}} = -5$; $\Delta V_{\text{max}} = \Delta H_{\text{max}} = 7$. The ΔH Function Table is the transpose of this table *i.e.*, the labels ΔH and ΔV are swapped. . . . 27

2·4 An example of the calculation of ΔV_{curr} and ΔH_{curr} values. ΔH_{prev} values come from the previous row. The Match locations and the leftmost ΔV_{curr} value are known. The ΔV_{curr} value for a particular column is found using the table in Figure 2·3. The input is the ΔH_{prev} value in the same column and the ΔV_{curr} value in the column to the left, *except*, when there is a Match, the value in the column to the left is treated as a max and, starting with Step 3, if the value in the column to the left is not assigned, it is treated as mid. $\Delta H_{prev}\dagger$is a modification of ΔH_{prev} in which all Match positions have been changed to max and all values less than mid have been changed to mid. The ΔH_{curr} value for a particular column is found using the transpose of the table in Figure 2·3. The input is the $\Delta H_{prev}\dagger$in the same column and the ΔV_{curr} value in the column to the left. 28

2·5 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. Bits erroneously set by the ADD (+) are shown in bold. Sample code is from the complete listing in Supplementary Information. 29

2·6 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. 30

2·7 Code for Zones B, C and D. 30

2·8 Top: The BitPAl Packed mapping of ΔH and ΔV values for the parameter set $M = 2, I = -3, G = -5$. Middle: Conversion from the thirteen ΔV_i vectors at left to the five "packed" vectors at right. Bottom: Example code for adding the packed representation. 31

2·9 **Running times.** Each experiment involved 25 million alignments. For BitPAl, alignment weights (M, I, G) are shown in parenthesis. All times are averages of three runs. Unit-cost BitPAl, 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, BitPAl (0, -1, -1) uses 23 bit operations. For $k = 7$, the times for BitPal and WM are nearly the same. By $k = 15$, BitPAl runs approximately twice as fast. Results for N are not shown on the graph. It was 118 to 304 times slower than BitPAl (0, -1, -1) even when optimal parameters were chosen. 32

2·10 **Running times.** Each experiment involved 25 million alignments. For BitPAl and BitPAl Packed, alignment weights (M, I, G) are shown in parenthesis. All times are averages of three runs. Variants of BitPAl and NW (shown as a horizontal line). For BitPAl, time is approximately linearly proportional to one dimension of the function table. For BitPAl packed, time is approximately linearly proportional to the area of the function tables. BitPAl packed (2, -3, -5) is approximately 7.1 times faster than NW and BitPAl (0,-1,-1) is approximately 24.9 times faster. 33

2·11 Comparison of the number of operations for BitPAl and BitPAl packed for different alignment weights (M, I, G). 34

3·1 **Left:** Actual alignment scores are shown in the corners. Δv and Δh are the differences between scores in adjacent cells and are shown along the arrows. **Middle:** Actual differences have been mapped into the variables ΔV and ΔH using Equations (3.2) with $G = -1$. ΔH values are labeled ΔH_{in} for row $i - 1$ and ΔH_{out} for the current row, i. The row i indices have been dropped for ΔV. **Right:** Substituting $\Delta H_{in}[j] = -4$, $\Delta V[j-1] = 6$, and $L[j] = 5$ into Equation (3.4) yields $\Delta V[j] = 2$. Note that $\Delta V[j-1] > L[j]$ and the sum is > 0. Substituting the same values into Equation (3.5), note that $\Delta H_{in}[j] > -L[j]$ so $\Delta H_{in}[j]$ is not used. Also the sum is > 0, so $\Delta H_{out}[j] = 0$, not 1. 39

3·2 **Three representations for Δ values.** Boxed bits represent the same value in the same column. The BitPAl Extension method uses the vector and Packed representations. The Partial Sums method used the Extended representation. 41

3·3 **Upper:** Schematic of ΔV output Function Tables for three L values, denoted $l_1 = 7, l_2 = 5$, $l_3 = 2$. Note that the L values and the relation $\Delta H = \Delta V$ set the boundaries for the marked zones in the tables. These zones contain $(\Delta H, \Delta V)$ pairs which determine the source of the best score of a cell in S, either from a horizontal gap (Zones A and B), a substitution (Zone C), or a vertical gap (Zone D). Borders between zones, indicated by dotted lines, yield ties for the best score. Each Table applies to the subset of columns j that share a common $L(j)$. **Lower:** Actual values in the l_2 and l_3 Function Tables. Note in the left table, for example, when the ΔV input falls below l_2 the output remains the same, *i.e.*, l_2 is the minimum $\Delta V[j-1]$ value. Compare with Equation (3.4). 53

3·4 An example of how values are stored across SIMD words in the *striped* format. 54

3·5 Upsweep step of SIMD prefix-sum computation. In this example, each SIMD word holds 4 values. **Step 1**: the values are *striped* across the SMID words. **Step 2**: pairs of words are added. **Step 3**: pairs at the next level of the tree are added. This process continues recursively until there is only one pair, the final SIMD word and the middle word. **Step 4**: the upsweep is continued by doing a parallel scan on the final SIMD word. 54

3·6 Downsweep step of SIMD prefix-sum computation. In this example, each SIMD word holds 4 values. The upsweep step has already been completed, the final word holds prefix-sums from the beginning for the 4 positions stored there. **Step 1**: the final word is shifted, and added to the first and second words. **Step 2**: The second word is added to the third. **Step 3**: This shows how values can be reordered (unstriped) into their original positions, but this is not actually done in each row. 55
3·7 **Comparison of algorithm run times for 25 million alignments.** Shown are averages over three trials. $|Y| = 126$. A new pair of sequences was generated for each alignment (1 to 1). 56
3·8 **Comparison of algorithm run times for 25 million alignments.** Shown are averages over three trials. $|Y| = 126$. Top: A new pair of sequences was generated for each alignment (1 to 1). Bottom: Each Y sequence was aligned against 100 newly generated X sequences (1 to 100). This models the alignment of a query sequence to a set of candidate matches and amortizes the pre-processing costs. 57

4·1 Computing ΔF, ΔE_{in} and ΔE_{out} from the S, E, and F matrices. . . 62
4·2 At the top, the wraparound dynamic programming scoring matrix (1 copy of pattern aligned to 1 copy of text) compared to the global alignment matrix for repeated pattern copies on the bottom. 64

4·3 **Comparison of algorithm run times for 25 million alignments.** Shown are averages over three trials. $|Y| = 126$. Top: A new pair of sequences was generated for each alignment (1 to 1). Bottom: Each Y sequence was aligned against 100 newly generated X sequences (1 to 100). This models the alignment of a query sequence to a set of candidate matches and amortizes the pre-processing costs. 74

4·4 **Comparison of algorithm run times for 250 thousand alignments.** Shown are averages over three trials. The pattern size used was $|Y| = 120$, compared against texts multiples (1, 2, 3, 4, and 5) of the pattern long. 75

A·1 The relationships between scores in adjacent cells in the scoring matrix: w, x, y, z are scores, h, v, u are differences: $h = y - x$, $v = w - x$, $u = z - y$. 80

C·1 Upsweep step of SIMD prefix-sum computation. In this example, each SIMD word holds 4 values. **Step 1**: the values are *striped* across the SMID words. **Step 2**: pairs of words are added. **Step 3**: pairs at the next level of the tree are added. This process continues recursively until there is only one pair, the final SIMD word and the middle word. **Step 4**: the upsweep is continued by doing a parallel scan on the final SIMD word. 100

C·2 Downsweep step of SIMD prefix-sum computation. In this example, each SIMD word holds 4 values. The upsweep step has already been completed, the final word holds prefix-sums from the beginning for the 4 positions stored there. **Step 1**: the final word is shifted, and added to the first and second words. **Step 2**: The second word is added to the third. **Step 3**: This shows how values can be reordered (unstriped) into their original positions, but this is not actually done in each row. 101

C·3 Blelloch's upsweep proceeds by summing pairs of values in a conceptual binary tree. This binary tree with n leaves has $n-1$ internal nodes, leading to $n-1$ operations. The *striped* SIMD scan arranges the pairs of values so that multiple value pairs within pairs of SIMD words can be summed at the same time. In the final word, values are shifted and added to complete the scan. Given W elements per SIMD word, there are $n/W-1$ addition operations before the scan in the final word and $2 \cdot log(W)$ operations (1 shift, 1 addition per log step in the final word scan. The larger n and W are, the larger the advantage of the *striped* SIMD scan. 103

C·4 Diagram of stripe format. Each word holds W values. To store n values, we use n/W words. Each *stripe* i consists of all values from words 0 to n/W at position i in each respective word. 104

C·5 Sums stored in ΔH_S after first step. 110

C·6 $\log(n/W)$ steps to upsweep to the final word. Pairs of words at each level are summed and stored in the "right" side of the pair. 111

C·7 The shifted last word SLW is added to words with indices that are $2^l - 1 < f$, in this case $0, 1, 3$, and 7. 111

C·8 Example of the downsweep after sums of SLW. Words are summed from left to right in steps from $\log(n/W)-2$ to 1, in this case $\log(16)-2 = 2$ to 1.**Step 2:** We start with indices $1, 3$, and 7. Words at these indices have their final values from the addition of SLW. At step 2, we will be adding words with indices $2^2 = 4$ apart. $1, 3 < 4$, so only word 7 is used and added to word 11. **Step 1:** At step one, we now consider indices $1, 3, 7$, and 11. We will add $2^1 = 2$ to each index ≥ 2. Words $3, 7$, and 11 are added to words $5, 9$, and 13. **Step 0:** We will add $2^0 = 1$ to each index ≥ 1. Words $1, 3, 5, 7, 9, 11$ and 13 are added to words $2, 4, 6, 8, 10, 12$ and 14. Every word now has its final value. . . 111

D·1 Computing ΔF, ΔE_{in} and ΔE_{out} from the S, E, and F matrices. . . 113

List of Abbreviations

BLOSUM	BLOcks SUbstitution Matrix
DNA	Deoxyribonucleic Acid
ED	Edit Distance
LCS	Longest Common Subsequence
NGS	Next Generation Sequencing
NW	Needleman-Wunsch
RNA	Ribonucleic Acid
SIMD	Single Instruction Multiple Data
TR	Tandem Repeat
VNTR	Variable Number Tandem Repeat

Chapter 1

Introduction and Background

The principle known as the central dogma of molecular biology captures the way in which genetic information, initially stored in DNA, is transformed into a protein product via the translation of messenger RNA previously transcribed from DNA. Each of these molecules are biological chain polymers that can be represented as sequences of characters corresponding to nucleic acids (DNA/RNA) or amino acids (protein). By examining these sequences, we can deepen our understanding of the processes underlying the intra- and interspecies diversification, including phylogenetic relationships, the mechanisms by which genotype can influence phenotype, and the evolution of proteins. Next generation sequencing (NGS) technologies have greatly increased in speed and decreased in cost in recent years providing researchers with an abundance of data. However, the timely processing of these data remains challenging, as the computational processing and aligning of these sequences have not improved as rapidly. This thesis addresses these challenges by presenting a family of novel sequence alignment algorithms which hold a substantial processing-time advantage over existing algorithms.

When DNA was first sequenced in the 1970s, labor and time intensive laboratory techniques were commonplace. The fields of biology and healthcare were changed forever with the first sequencing of the human genome in 2001 [38, 58]. The availability of the full human genome lead to an explosion of genomic studies and allowed the development of next-generation sequencing techniques. High-throughput NGS techniques have hugely decreased the cost and increased the speed of sequencing. Currently, a whole human genome can be sequenced for approximately a thousand dollars in a short period of time, as little as 26 hours [46]. As advances are made in biological sequencing technology and the price of sequencing continues to fall, the number of sequencing projects underway is increasing. Projects like the 1000 Genome Project [13] and the Genome Project for Food Pathogens (100,000 genome project)[1] are making available huge collections of genomic data. Consequently, the amount of available sequence data generated is growing rapidly. These large datasets, and the need to align reads across them, are highlighting the importance of fast alignment tools. Several modern aligners (Bowtie[39], SOAP 2[43], BWA[42]) focus on the problems of indexing and seed creation, building efficient indices to allow fast search through large databases. However, one avenue for optimization that has not been fully explored is the alignment algorithms at the heart of these programs, many of which use a traditional dynamic programming approach.

1.1 Background

The Benson lab focuses on the investigation of genomic features called tandem repeats (TR) and variable number of tandem repeats (VNTR)[6, 7, 23]. A tandem repeat is a locus in the genome with multiple adjacent copies of an underlying pattern.

Tandem repeats can vary in copy number across a population. For example, at a particular locus, one person may have four copies and another may have five copies. Such a locus is called a variable number of tandem repeats (VNTR). VNTRs are known to have important effects on chromatin structure [55, 56, 2, 55, 61], gene expression [60], and disease states [59, 26, 21, 11, 41, 40, 52], so their discovery and analysis are crucial to understanding genomic function. NGS data contains infor-

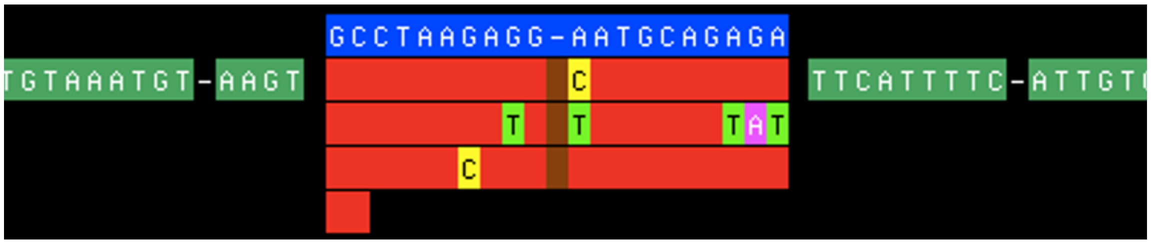


Figure 1·1: In this figure, we represent a tandem repeat sequence as the pattern sequence (in blue), with the repeated copies below it in red. Anywhere the copy varies from the pattern, we show the base that varies. Flanking sequence is shown in green.

mation on VNTRs, but current read mapping algorithms do not accurately map the reads covering these TRs because the VNTRs look like insertions or deletions (indels) or because the repeats look alike. The Benson lab has developed a computational pipeline, VNTRseek, for genome wide discovery of these VNTRs. This pipeline uses many alignments. For example, when two TRs are found that have the same pattern, the flanking sequence on each side of them must be aligned to determine whether the TRs are at the same location in the genome.

1.1.1 Global Alignment

In global sequence alignment, two sequences are aligned end-to-end, such that every base in each sequence is aligned either to a base in the other sequence or to a gap. Global alignments are useful for sequences of similar lengths expected to be generally similar, for example orthologous genes, or in our lab, the flanking sequence surrounding TRs. This differs from local alignment, in which subsequences of the sequences are aligned. A local alignment is better suited for scenarios where sequences that are dissimilar in length or sequence are expected to contain small regions of similarity.

The first global sequence alignment algorithm was the dynamic programming solution presented by Needleman and Wunsch [51]. Dynamic programming alignment

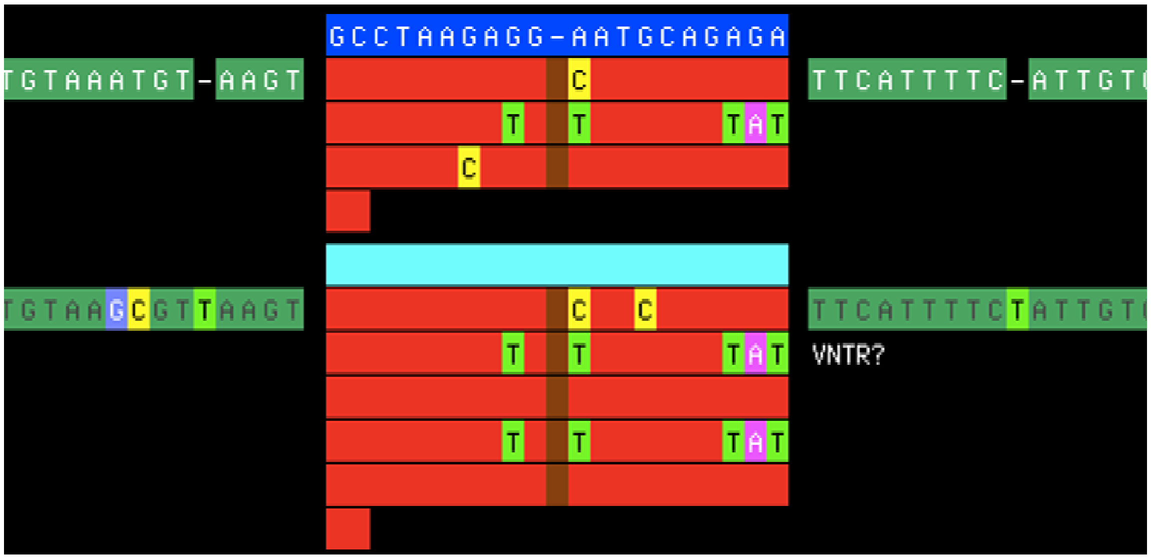


Figure 1·2: Two tandem repeats at the same locus, where one has just over 3 copies and one has just over 5.

algorithms like Needleman-Wunsch and Smith-Waterman [53] (which is used to compute local alignment) iterate cell by cell over the alignment scoring matrix, with each cell requiring many low-level machine operations. This leads to the general $O(n^2)$ efficiency of dynamic programming alignment algorithms. Since then, bit-parallel and Single Instruction Multiple Data (SIMD) algorithms have been applied to similar pattern matching problems, resulting in large decreases in run time. One of the computational differences between global and local alignment is that the global alignment score is always dependent on the surrounding alignment scores, while local alignment allows the alignment score to be reset to zero at any point in the algorithm, breaking the local dependency. Bit-parallel and SIMD algorithms have been developed for local alignment [31, 18, 16], but the acceleration of our algorithms requires the local dependencies of global alignment.

1.1.2 Bit-parallel Algorithms

Bit-parallel algorithms use computer words to represent consecutive cells in rows of the scoring matrix, so n-bit computer words can represent n sequence characters. The use of bit-vectors to represent the scoring matrix means that low-level (and fast) bit-operations can be used to compute the value of an entire bit-vector, instead of costly branching statements applied within a loop. This effectively reduces the order of magnitude of the algorithm, especially if only a few binary operations are needed. Current computer word size is 64 bits, and operations on 128 and 256 bits are now possible using SIMD instructions (Single Instruction Multiple Data). This means that significant speedups in run-time are possible over algorithms that compute scores sequentially.

Bit-parallel methods are used primarily to compute alignment scores rather than to recover the actual alignments, and they can be used as filters to quickly identify sequences that are similar enough to warrant further exploration. Until recently, efficient bit-parallel methods for pairwise sequence alignment were available only for the longest common subsequence (LCS) [28, 14, 3] and unit-cost edit distance (ED) [49, 32, 33, 29, 45] problems. However, the bit-parallel solutions for these problems were *ad hoc* and not adaptable to other common scoring schemes.

1.1.3 SIMD Algorithms

Modern processors provide Single Instruction Multiple Data (SIMD) registers and instructions that allow multiple values to be stored and processed as a unit. These have been used in the past [18], [16] to implement accelerated dynamic programming alignment algorithms. One of the weaknesses of these algorithms has been that scores stored in SIMD registers cause overflows within the small space (1 byte) provided by the most dense SIMD format, requiring recalculation with a larger storage mode (2, 4, or 8 bytes).

1.2 Thesis Overview

In this thesis, I present several bit-parallel and SIMD algorithms for sequence alignment. All of our algorithms are based on the principle of storing the difference in scores rather than the scores themselves. This is advantageous in the case of bit-parallel algorithms because it allows us to use a fixed and small number of bit-vectors. In the case of the SIMD algorithms, storing the differences ensures that our values fit within the limits of the SIMD registers. This avoids the recomputation of previous SIMD methods.

1.2.1 BitPAl, bit-parallel algorithms for global alignment with general integer scoring

Chapter 2 describes BitPAl, our bit-parallel algorithms for global alignment with general integer scoring. Integer-scoring schemes assign integer weights for match, mismatch and insertion/deletion. The bit-parallel approach represents individual cells in an alignment scoring matrix as bits in computer words and emulates the calculation of scores by a series of logic operations composed 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 fast algorithms in practice. The BitPAl method uses structural properties in the relationship between adjacent scores in the scoring matrix to construct classes of efficient algorithms, each designed for a particular set of weights. In timed tests, we show that BitPAl runs several times faster than a standard iterative algorithm.

1.2.2 Bit-parallel and SIMD algorithms for global alignment with substitution scoring

Substitution scoring assigns a potentially different substitution weight to every pair of alphabet characters, which better represents the relative rates of different mutations. Examples of substitution scoring include BLOSUM scoring, commonly used for protein sequences, and transition-transversion scoring, used for DNA sequences. Chapter 3 extends the BitPAl algorithm to use substitution scoring and introduces new SIMD algorithms for global alignment. The BitPAl approach extends the existing algorithm to the more complicated relationship between adjacent scores when given more than two possible substitution scores. The SIMD algorithm is a new approach that uses partial sums of adjacent score differences. I present a simpler partial sum method as well as one that uses parallel scan for additional acceleration. Results demonstrate that these algorithms are significantly faster than an existing SIMD dynamic programming algorithm.

1.2.3 SIMD algorithms for global alignment with affine gapping and tandem alignment

Chapter 4 implements two extensions to the partial sums algorithm. The first adds support for affine gap penalty scoring. Affine gap scoring represents the biological reality that it is more likely for gaps to be continuous than distributed throughout a region by introducing a gap opening penalty and a gap extension penalty. The second extension is an algorithm that uses the partial sums method to calculate the tandem alignment of a pattern against a text sequence using a single pattern copy.

Chapter 2

BitPAl: a bit-parallel, general integer-scoring sequence alignment algorithm

2.1 Introduction

Bit-parallel algorithms have been developed for exact and approximate string matching problems. Early examples include the algorithms of [5], which finds exact matches to a simple string pattern, and [62], which finds approximate matches to a string pattern or a regular expression, where the number of differences between the pattern and the text is at most k (counting single character substitutions and single character insertions and deletions or indels). The latter is implemented as the Unix command *agrep*. Additional k-differences examples include [63], an approach based on the Four Russians technique [4], which finds matches to "limited expressions," *i.e.*, regular expressions without Kleene closure, [49], which finds matches to simple string patterns and emulates the dynamic programming solution used in alignment, and [50], which allows arbitrary integer weights for substitution of each pair of characters, insertion of each character, and deletion of each character, and finds occurrences of regular expressions where the *sum* of the edit weights is at most k. In most k-differences algorithms, the complexity (and computing time) increases with increasing k.

Bit-parallel methods have been successfully applied to the longest common subsequence (LCS) problem [3, 14, 28], and to unit-cost edit-distance [30, 32] by modifications of [49]. These algorithms compute the alignment score, de-linking 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 bit-parallel implementations use bits to represent the cells where an increase occurs. In edit-distance scoring, 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 *ad hoc* in their approach, relying on specific properties of the underlying problems, making it difficult to directly adapt them to other alignment scoring schemes. Bergeron and Hamel [9], addressed general integer scoring, outlining construction of a distance-based algorithm from a corresponding finite automaton, but the transformation was costly in terms of bit-operations and they gave no implementation.

Below we present a bit-parallel method for similarity and distance based global alignment using general integer-scoring [8], allowing arbitrary integer weights for match, mismatch, and indel. Other approaches have been suggested by [62] and [9]. The method of [50] is more flexible in scoring and applies to both simple patterns and regular expressions, but is much slower than our method in practice. Our contribution is based on an observation of the regularity in the relationship between adjacent scores in the scoring matrix (Section 2.2.1) and the design of an efficient series of bit operations to exploit that regularity (Section 2.3). Because every distinct choice of weights requires a different program, we show how to construct a class of efficient algorithms, each designed for a particular set of weights, and provide an online C code generator for users. The complexity of our algorithms depends on the weights, not the ultimate score of the alignment. Our method works for general alphabets, but our interest derives from frequent use of DNA alignment when analyzing high-throughput sequencing data to detect genetic variation.

2.2 Methods

The problem to be solved is stated in terms of similarity scoring, but the technique applies to 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 (match), I (mismatch), and G (indel or gap), calculate the global alignment similarity score for X and Y using logic and addition operations on computer words of length w.*

We are interested in two measures of efficiency for the algorithms. The first is standard time complexity and the second is a ratio of the word size, w, and the count, p, of logic and addition operations required to process w consecutive cells in the alignment scoring matrix. The efficiency, $e = w/p$, is the average number of cells computed per operation. For example, when using 64 bit words, LCS has $e = 64/4 = 16$ ($p = 4$ operations per word [28]), and edit distance has $e = 64/15 \approx 4.2$ (an improvement from 64/16 in the method of [49, 32]; see Appendix of [45] for details). Since p is independent of w, if the word size doubles, e doubles too. Note that we are counting only logic and addition operations, not storage of values in program variables. Adding store operations would be more accurate but the number of these operations is compiler and optimization level specific.

We require that the alignment method be global or semi-global. That is, we do not restrict the initializations in the first row or column of the alignment scoring matrix or where in the last row or column the alignment score is obtained. 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 (semi-global alignment where an initial gap has no penalty).

We assume that match scores are positive or zero, $M \geq 0$, 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 (*e.g.*, Longest Common Subsequence has $G = 0,\ I = -\infty,\ M = 1$) or require more complicated structures than detailed here (*e.g.*, protein alignment using PAM or BLOSUM style amino acid substitution tables).

2.2.1 Function Tables

Let S be a recursively-defined, global similarity scoring function for two sequences X and Y computed in an alignment scoring matrix:

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

Instead of actual values of S, we store only the differences, ΔV, between a cell and the cell above, and ΔH, between a cell and the cell to its left:

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

It is an easy exercise to prove that the minimum and maximum values for ΔV and ΔH are G and $M - G$ respectively. Lemma 2.2.1 gives the recursive definitions for ΔV and ΔH in terms of M, I, and G.

Lemma 2.2.1. *The values for ΔV are as shown below and the values for ΔH are computed similarly. That is, $\Delta H[i,j]$ in matrix S is equal to $V[j,i]$ in the transpose of matrix S.*

$$\underset{\forall i,j\geq 1}{\Delta V[i,j]} = \begin{cases} M - \Delta H[i-1,j] & \textit{Match, } \text{i.e.: } \textit{if } X_i = Y_j \\ I - \Delta H[i-1,j] & \textit{Mismatch, } \text{i.e.: } \textit{if} \\ & I - G \geq \begin{cases} \Delta H[i-1,j] \\ \Delta V[i,j-1] \end{cases} \\ G & \textit{Indel from above, } \text{i.e.: } \textit{if} \\ & \Delta H[i-1,j] \geq \begin{cases} I - G \\ \Delta V[i,j-1] \end{cases} \\ \Delta V[i,j-1]+ & \\ G - \Delta H[i-1,j] & \textit{Indel from left, } \text{i.e.: } \textit{if} \\ & \Delta V[i,j-1] \geq \begin{cases} I - G \\ \Delta H[i-1,j] \end{cases} \end{cases}$$

$$\left(\underset{\forall j\geq 1}{V[0,j]} = G \textit{ or } \underset{\forall j\geq 1}{V[0,j]} = 0 \right)$$

Proof. By substitution in the recursive formula for S. □

The recursion for ΔV is summarized in the Function Table in Figure 2·1. Note the value $I-G$, which frequently occurs in the recursion, and the relation $\Delta H = \Delta V$. They set the boundaries for the marked zones in the table. These zones comprise $(\Delta V, \Delta H)$ 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

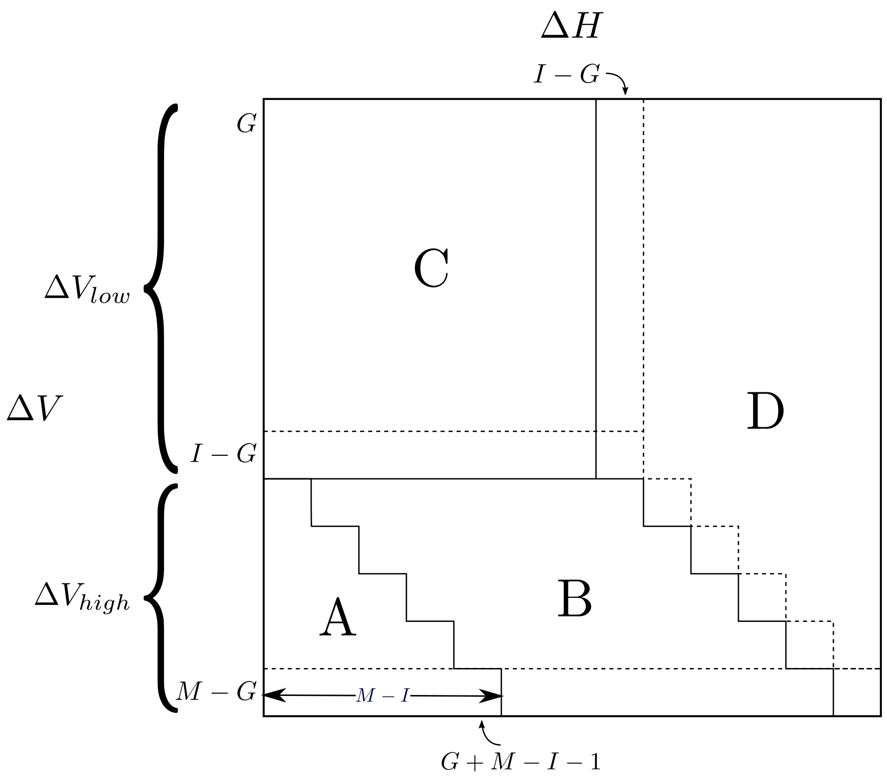


Figure 2·1: Zones in the Function Table for ΔV. Zone A: All values are in $\Delta V_{\text{high}} \in \{I-G+1, \ldots, M-G\}$; Zone B: All values are in $\Delta V_{\text{low}} \in \{G, \ldots, I-G\}$; Zone C: All values are in ΔV_{low} and values depend only on ΔH; Zone D: All values are G; Last Row: Values also apply when there is a Match;. First Column: Identity column for values in ΔV_{high}.

dotted lines, yield ties for the best score. Figure 2·2 shows how the relative size of the Zones changes with changes in I and G.

2.3 Algorithm

Definitions: min $= G$, max $= M-G$, mid $= I-G$, low $\in$ {min, …,mid}, high $\in$ {mid +1, …, max}.

For the illustrations in this chapter, we use the scoring weights:

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

which yield

$$\text{min} = -5, \text{max} = 7,\ \text{mid} = 2,$$

$$\text{low} \in \{-5, \ldots, 2\}, \text{high} \in \{3, \ldots, 7\}.$$

The ΔV Function Table for these weights is shown in Figure 2·3.

The algorithm proceeds row-by-row through the alignment matrix. For each row, the input is

- the ΔH values from the preceding row,
- the leftmost ΔV value in the current row, and
- the Match positions in the current row.

The computation first determines all the remaining ΔV values for the current row and then, using those, determines the ΔH values for the current row. A central concept is a *run of* ΔH_{min}. This is a set of consecutive positions in the preceding row for which the values of ΔH all equal min (in Figure 2·4, positions for which $\Delta H = -5$). The algorithm has the following steps (see Figure 2·4) which follow from Lemma 2.2.1.

1. Find the locations where $\Delta V = \text{max}$ (highest value in Zone A):

   Step 1A: due to a match between the characters in Sequence X and Sequence Y. These occur at match locations where $\Delta H = \text{min}$.

   Step 1B: in any run of ΔH_{min} to the right of a match location in the run.

2. Find the locations where $\Delta V = i$, for $i \in \{\text{mid}+1, \ldots, \max -1\}$ (the remaining values in Zone A). These are computed in decreasing order of i. For each i, there are two categories, those locations:

   Step 2A: due to a match or a larger preceding ΔV value. These also depend on the ΔH value.

   Step 2B: due to the value i being carried through a run of $\Delta H_{\min}$.

3. Find the locations where $\Delta V = i$, for $i \in \{\min +1, \ldots, \text{mid}\}$ (the values in Zones B and C). These are computed separately for each value i and depend on:

   Step 3A: a match or the preceding ΔV value and the ΔH value (Zone B).

   Step 3B: the ΔH value alone (Zone C).

4. Find the locations where $\Delta V = \min$ (the values in Zone D). These are:

   Step 4: all the remaining locations with undetermined ΔV values.

5. Find the current row locations where the new $\Delta H = i$ for:

   Step 5A: $i > \min$.

   Step 5B: $i = \min$.

We describe the simplest case where the length of the first sequence is less than the computer word size w. Longer sequences can be handled in "chunks," where each chunk has size w. Match positions for every row are computed prior to the calculation of the row values as is also done for the LCS and edit-distance problems. Details are given at the end.

We present two algorithms, **BitPAl** and **BitPAl Packed**. They differ in the data structures used to hold and process the ΔH and ΔV values and their computation of Steps 3, 4, and 5. Correctness theorems and proofs for the various steps are presented in Appendix A.

2.3.1 BitPAl

Data Structure for BitPAl. One computer word (sometimes called a vector) represents 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 $\{G, \ldots, M-G\} = \{-5, -4, \ldots, 6, 7\}$, and therefore 13 words each, for ΔH and ΔV. **Computing the Δ values.** To compute its output values, each cell needs to know its ΔH and ΔV input values. As in standard left to right processing, the output ΔV value from one cell becomes the input value for the cell to its right. All the input ΔH values are in the preceding row.

Zone A. Inspection of the Function Table (Figure 2·3) reveals that the output values in Zone A are interdependent, and require computing in order from high to low. For example, output $\Delta V = 5$ can be obtained in two ways from higher ΔV input values, $(\Delta V = 7, \Delta H = -3)$ and $(\Delta V = 6, \Delta H = -4)$. $\Delta V = 5$ cannot be obtained from lower ΔV input values.

The leftmost column in the table, $\Delta H_{\min}$ (-5 in the example), is an identity column. This means that for runs of $\Delta H_{\min}$, an input ΔV value yields the identical ΔV ouput for every location in the run to the right of the input. For example, if the input $\Delta V = 5$ for the leftmost position in a run, then the output ΔV for every position in the run is also 5 (see Figure 2·4 steps 1B, 2B for 4). Carrying an input value through a run of $\Delta H_{\min}$ can be accomplished with an addition (+) as seen below. Addition is similarly used to solve left-to-right dependency problems in LCS and edit-distance bit-parallel algorithms.

Note in the bottom row of the Function Table that a Match acts as an input ΔV_{max} (7 in the example), so we will treat the Match positions as having input ΔV_{max}.

Steps 1A and 1B: The locations where $\Delta V = \max$, stored in the ΔV_{max} vector, are calculated with four operations (Figure 2·5). The locations are shifted one position to the right for input to subsequent calculations. The operations are: 1) an `AND` to find max due to Matches, 2) an `ADDITION (+)` to carry max through runs of ΔH_{min} and into the position following a run (because the result will be shifted). This causes erroneous internal bit flips if there are multiple Matches in the same run, 3) an `XOR` with ΔH_{min} to complement the bits within the ΔH_{min} runs, and 4) an `XOR` with the initial ΔV_{max} to correct any erroneous bits and finish the shift by removing the locations set with Matches.

Steps 2A and 2B: Remaining ΔV_{high} vectors are calculated, in descending order from $\Delta V = \max -1$ to $\Delta V = \text{mid} + 1$ due to the dependencies as discussed above. The operations are: 1) finding the locations due to a preceding higher ΔV value using `AND` of appropriate $(\Delta V, \Delta H)$ pairs (which intersect along a common diagonal in the Function Table) and collecting them together with `OR`s, 2) shifting the initial vectors right one position for subsequent calculations, 3) carrying through runs of ΔH_{min} computed in two operations, an `ADDITION (+)` as before and an `XOR` with ΔH_{min} to complement the bits within the ΔH_{min} runs (Figure 2·6). Before the addition, those ΔH_{min} positions that have already output a ΔV_{max} value must be removed.

Steps 3A and 3B. (Figure 2·7). At this point, all the ΔV_{high} input values for Zone B have been computed (they are the outputs from Zone A), remaining output values are all ΔV_{low}. The operations are: 1) the `AND` of appropriate $(\Delta V, \Delta H)$ pairs, which intersect along a common diagonal (Zone B), 2) the `AND` of the appropriate

ΔH vector and all positions without a ΔV_{high} output (Zone C), 3) an `OR` combination of the preceding two results and 4) a shift of the locations one position to the right for subsequent calculations.

Step 4: Zone D has only one output value, ΔV_{min}. It is assigned to all remaining locations as well as the zero location if gap penalty in the first column is being used.

Step 5: After the ΔV values are computed, all inputs are available and the new ΔH vectors for the current row can be computed immediately. The Function Table for the new ΔH is the transpose of the table for ΔV, *i.e.*, the input labels are swapped. Each new ΔH vector is obtained by the `AND` of appropriate $(\Delta V, \Delta H)$ input pairs, which intersect along a common diagonal, collected together with `OR`s. Before this can proceed, though, the Match positions must be added to the previous row's ΔH_{max} vector (with `OR`) and removed from all other previous row ΔH vectors. Also, all previous row ΔH_{low} locations must be converted to ΔH_{mid}.

2.3.2 BitPAl Packed

Data structure for BitPAl packed. The number of logic operations in BitPAl scales linearly with the size of the function table. Many of these are the AND and OR operations to compute identical values along Zone B diagonals. These calculations can be performed more efficiently with a new representation. The idea is to store the input ΔH and ΔV values in such a way that they can all be added simultaneously to give the appropriate output values.

Rather than using bit-vectors to represent single ΔH or ΔV values, we use them to represent binary digits (Figure 2·8). We map the ΔV values $\{\text{min}, \dots, \text{max}\}$ one-to-one onto the positive values $\{0, \dots, \text{max} - \text{min}\}$ and store them in the vectors $\Delta V_{p0}, \Delta V_{p1}, \Delta V_{p2}$, etc. where p_i is the place holder for the ith power of 2. The mapping for ΔH is onto negative numbers *i.e.*, $\{\text{min}, \dots, \text{max}\}$ are mapped to

$\{0,\ldots,-(\max-\min)\}$ and stored in vectors $\Delta H_{p0}, \Delta H_{p1}, \Delta H_{p2}$, etc. After addition, the sums will fall in $\{-(\max-\min),\ldots,\max-\min\}$, so we use $\lceil \log_2(2(\max-\min)+1)\rceil$ bit-vectors for ΔH and ΔV. For our example, the ΔV values are mapped to $\{0,\ldots,12\}$, the ΔH values are mapped to $\{0,\ldots,-12\}$ and the sums fall within $\{-12,\ldots,12\}$, so we use 5 vectors each for ΔH and ΔV.

BitPAl Packed does not change the computation of the ΔV values in Zone A. The ΔH values are always maintained in the packed representation, but some are unpacked into the original representation for the Zone A computations. Once Steps 1 and 2 are completed, all locations without a ΔV value are set to mid, all Match locations are set to max, and the ΔV values are converted into the packed representation.

Steps 3 and 4 are computed by "adding" together the two sets of packed vectors using a series of `AND`, `OR`, and `XOR` operations (Figure 2·8) to produce the final encoded values for ΔV. Any negative values (sign bit set) are converted to min (Zone D). For Step 5, the new ΔH values are determined with a second addition. Since all input ΔH in the range [min, mid] give the same result, we first re-encode that range to mid.

Packing and unpacking. Packing ΔV vectors involves identifying the locations where the binary representation of the encoded values all have a specific bit set. For example, the binary representations for 1, 3, 5, 7, 9, and 11 all have the bit representing 2^0 set and the binary representations for 2, 3, 6, 7, 10, and 11 all have the bit representing 2^1 set. Effectively then,

$$\begin{aligned}\Delta V_{p0} &= \Delta V_1 \text{ OR } \Delta V_3 \text{ OR } \Delta V_5 \text{ OR } \Delta V_7 \text{ OR } \Delta V_9 \text{ OR } \Delta V_{11} \\ \Delta V_{p1} &= \Delta V_2 \text{ OR } \Delta V_3 \text{ OR } \Delta V_6 \text{ OR } \Delta V_7 \text{ OR } \Delta V_{10} \text{ OR } \Delta V_{11} \\ &\text{etc.}\end{aligned}$$

where ΔV_i is the vector of locations with encoded value i. However, as can be seen for these two examples, there are common terms $(\Delta V_3, \Delta V_7, \Delta V_{11})$, so combining the terms as above leads to inefficiencies.

Unpacking the ΔH vectors involves identifying locations of specific encoded values from the binary representation vectors. For example, the ΔH_{-1} locations are those (using two's complement, -1 = 11111) that have all bits set and ΔH_{-2} locations are those (using two's complement, -2 = 11110) that have all but the lowest bit set. Again, effectively:

$$\begin{aligned}
\Delta H_{-1} &= \Delta H_{p0} \,\&\, \Delta H_{p1} \,\&\, \Delta H_{p2} \,\&\, \Delta H_{p3} \,\&\, \Delta H_{p4} \\
\Delta H_{-2} &= \sim \Delta H_{p0} \,\&\, \Delta H_{p1} \,\&\, \Delta H_{p2} \,\&\, \Delta H_{p3} \,\&\, \Delta H_{p4} \\
&\qquad \text{etc.}
\end{aligned}$$

Again, there are common terms which can be combined to avoid inefficiencies. For both packing and unpacking, we use a binary tree structure in the code generator to guide creation of temporary intermediate vectors so that operations are not duplicated.

2.3.3 Other Tasks

Determining Matches. As a preprocessing step, the position of the matches are determined for each character σ in the sequence alphabet. 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.

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 in [36], this takes

$O(n + M - 2G)$ operations with a small constant:

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

where bits_i is the number of 1 bits set in ΔH_i.

For BitPAl Packed, the alignment score can similarly be computed in $O(n \cdot k)$ operations

$$S[m, n] = m * G + \sum_{i=0}^{k-1} \text{pbits}_i * 2^i.$$

where pbits is the number of 1 bits set in ΔH_{pi}, and k is the number of bit vectors in the packed representation.

Several straightforward methods can be used to efficiently find all scores in the last row or last column.

2.3.4 Backtrace to Recover Alignment

In order to recover the alignment of the two sequences, dynamic programming algorithms often store a traceback matrix which records, for each cell in the scoring matrix, which cell(s) (left, diagonal, or above) the value came from. Starting at the bottom right hand corner of the traceback matrix, the alignment can be recovered by tracing backward along the path(s) that the score came from. This method was extended in [64] to Myer's bit-parallel edit-distance algorithm, via the creation of a bit-parallel traceback matrix.

We present here an alternative method that does not require the creation of any new data structures. For each row of the alignment, it only requires the storage of the ΔH and ΔV bit-vectors storing the gap value.

Preliminaries We begin by creating two strings s_x and s_y to hold the aligned sequences. We store the final row and column in variables r and c, respectively. We set pointers p_c and p_r to the final character in each sequence X and Y.

Recursion We check the row r ΔV gap bit-vector at column c. This can be done by a SHIFT and an AND. If there is a bit set in column c, we decrement r, insert a "-" character in s_y, insert the character at p_c in s_x, and decrement p_c. If there was not a bit set in ΔV gap bit-vector column c, we check the row r ΔH gap bit-vector at column c. If there is a bit set in column c, we decrement c, insert a "-" character in s_x, insert the character at p_r in s_y, and decrement p_r. If neither ΔV or ΔH have a bit set in row r at column c, we decrement both c and r, insert the character at p_c in s_x, insert the character at p_r at s_y, and decrement both p_c and p_r.

This process continues until r and c are zero. At the end, s_x and s_y contain the reverse alignment of X and Y and the alignment can be recovered by reversing both strings. This is similar to the method of [27], although that method only uses the ΔV values and so must do more work to determine whether a horizontal gap occurred. This method has been extended to the later bit-parallel and SIMD algorithms described, allowing all of the algorithms presented to either calculate the alignment score only or calculate the alignment score and recover the alignment.

2.3.5 Complexity and Number of Operations

The time complexity of our algorithms is $O(znm/w)$ where z depends on the version. For BitPAl standard, z represents the combined size of Zones A, B, and C (the latter reduced to a single row as in Figure 2·3) 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 chapter, the number of logic and addition operations,

p, per word is 265, yielding an efficiency of $64/265 \approx 0.24$ cells per operation with 64 bit words.

For the packed version, z represents the size of Zone A, the number of distinct ΔH and ΔV values for the packing and unpacking steps, and the binary log of the number of distinct values for the addition steps:

$$z = (M - I)^2 + (M - 2G + 1) + \log_2(M - 2G + 1).$$

Unlike the standard version, the term constants are not uniform (approximately 2, 2, and 12 respectively). For the example weights used in this chapter, the number of logic and addition operations, p, per word is 166, yielding an efficiency of $64/166 \approx 0.38$ cells per operation for 64 bit words. See Figure 2·11 and Table **??** for a comparison of the number of operations required by the two algorithms for different alignment weights.

Implementation

Each unique set of weights M, I, and G requires a uniquely tailored program. To simplify usage, we have constructed a web site, http://lobstah.bu.edu/BitPAl/BitPAl.html that generates C source code for download. The website takes as input the user's alignment weights, the algorithm version (standard or packed), whether it will be used for short sequences (single word) or long sequences (multiple word), and where the final score should be found.

2.4 Experimental Results

We compared running times for several bit-parallel algorithms using different alignment weights: 1) BitPal, 2) BitPAl Packed, 2) NW – the classical [51] dynamic programming alignment algorithm, 3) LCS – the bit-parallel LCS algorithm of [28],

4) ED – our improved bit-parallel, unit-cost edit-distance algorithm from the method of [49, 32], 5) WM – the unit-cost [62] approximate pattern matching algorithm, and 6) N – the [50] general integer scoring, approximate regular expression matching algorithm. We implemented BitPAl, BitPAl Packed, 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 BitPAl, BitPAl Packed, 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. 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 Figures 2·9 and 2·10 and Table 2.1. The runtime of WM depended on k, the number of differences allowed. For $k = 7$, the runtimes for BitPal and WM are nearly the same. By $k = 15$, BitPAl runs approximately twice as fast. N was 118 to 304 times slower than BitPAl (0, -1, -1) even when optimal parameters were chosen. BitPAl Packed (2, -3, -5) is approximately 7.1 times faster than NW and BitPAl (0,-1,-1) is approximately 24.9 times faster. For parameter values other than (0, -1, -1), BitPAl Packed is faster than BitPAl and the number of operations it uses and its runtime grows more slowly, leading to BitPAl Packed being approximately 4.8 times faster than BitPAl for parameter values (4, -7, -11) with a third as many operations.

2.5 Discussion

The BitPAl and BitPAl packed algorithms outlined above can be extended in several ways. Computers now in common usage have special 128 bit SIMD registers (Single Instruction, Multiple Data). Later chapters will discuss SIMD implementations of

	Parameters (M, I, G)				
Algorithm	0, -1, -1	2, -3, 5	3, -4, -6	4, -5, -9	4, -7, -11
BitPAl	0.284000	1.903778	2.702000	5.408722	8.517500
BitPAl Packed	0.390500	0.999945	1.126500	1.475222	1.755500

Table 2.1: Table of run times in minutes. Shown are averages over three trials for 25 million alignments. Needleman-Wunsch has the same runtime for all parameters, 7.056056 minutes.

	Parameters (M, I, G)				
Algorithm	0,-1,-1	2,-3,-5	3,-4,-6	4,-5,-9	4,-7,-11
BitPAl	23	265	416	763	1059
BitPal Packed	66	166	201	279	335

Table 2.2: Table of the number of operations in the main loop of BitPAl and BitPAl Packed for various alignment parameters (M, I, G).

related algorithms. 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 done simultaneously, an ideal situation for the use of general purpose graphical processing units (GPGPU).

Extension to local alignment is also possible. This is a different class of problem in that the best final alignment score can occur in any cell of the alignment matrix. If all the cells have to be examined, then the time complexity shifts back to $O(nm)$. [31] had some success with this problem using unit cost weights and identifying *columns* in which the score of at least one cell exceeds a predefined threshold k. The BitPAl methods have already been used to accelerate software for detecting tandem repeat variants in high-throughput sequencing data [23] and are well suited to other DNA sequence comparison tasks that involve computing many alignments.

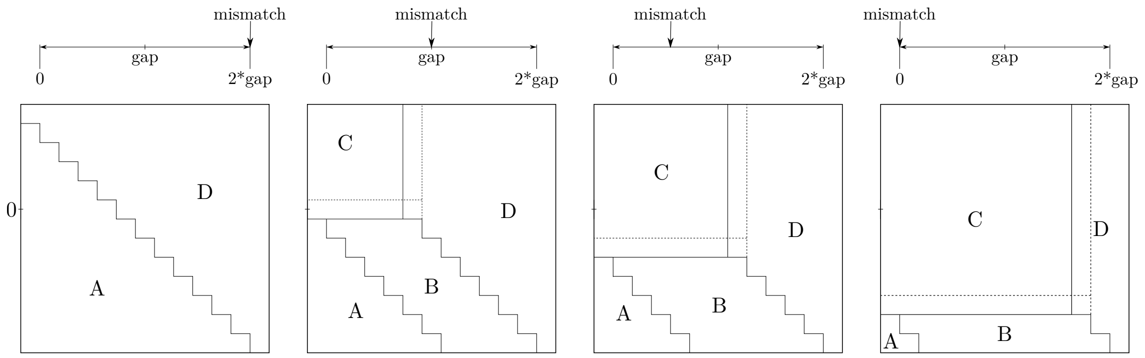


Figure 2·2: Relative size of Zones as I (mismatch penalty) decreases from $2G$ (twice gap penalty) where there is no preference for mismatches, to zero, where mismatches are free and gaps are introduced only to obtain matches.

	ΔH												
ΔV	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
-5...2	2	1	0	-1	-2	-3	-4	-5	-5	-5	-5	-5	-5
3	3	2	1	0	-1	-2	-3	-4	-5	-5	-5	-5	-5
4	4	3	2	1	0	-1	-2	-3	-4	-5	-5	-5	-5
5	5	4	3	2	1	0	-1	-2	-3	-4	-5	-5	-5
6	6	5	4	3	2	1	0	-1	-2	-3	-4	-5	-5
7 and match	7	6	5	4	3	2	1	0	-1	-2	-3	-4	-5

Figure 2·3: The ΔV Function Table for the weights $M = 2$, $I = -3$, $G = -5$. Note that $\Delta V_{\text{high}}, \Delta H_{\text{high}} \in [3, 7]$; $\Delta V_{\text{low}}, \Delta H_{\text{low}} \in [-5, 2]$; $\Delta V_{\text{min}} = \Delta H_{\text{min}} = -5$; $\Delta V_{\text{max}} = \Delta H_{\text{max}} = 7$. The ΔH Function Table is the transpose of this table *i.e.*, the labels ΔH and ΔV are swapped.

	Matches		*			*			*		*							
	ΔH_{prev}		-2	-5	-5	-5	-4	-4	3	1	-5	-5	-5	2	3	-5	-5	6
Step 1A	ΔV_{curr}	-5				7					7							
Step 1B		-5				7					7	7	7					
Step 2A (for 6)		-5				7	6				7	7	7					
Step 2A (for 5)		-5				7	6	5			7	7	7					
Step 2A (for 4)		-5	4			7	6	5			7	7	7					
Step 2B (for 4)		-5	4	4	4	7	6	5			7	7	7					
Step 3B (for 2)		-5	4	4	4	7	6	5			7	7	7			2	2	
Step 3A (for 0)		-5	4	4	4	7	6	5			7	7	7	0		2	2	
Step 3A (for -1)		-5	4	4	4	7	6	5	-1		7	7	7	0		2	2	
Step 3B (for -4)		-5	4	4	4	7	6	5	-1	-4	7	7	7	0		2	2	
Step 4	ΔV_{curr}	-5	4	4	4	7	6	5	-1	-4	7	7	7	0	-5	2	2	-5

	Matches	*			*			*		*							
	$\Delta H_{prev}\dagger$	7	2	2	7	2	2	7	2	7	2	2	2	3	2	2	6
Step 5A	ΔH_{curr}	7			-2			-3	-2	6				-2	2		-1
Step 5B		7	-5	-5	-2	-5	-5	-3	-2	6	-5	-5	-5	-2	2	-5	-1

Figure 2·4: An example of the calculation of ΔV_{curr} and ΔH_{curr} values. ΔH_{prev} values come from the previous row. The Match locations and the leftmost ΔV_{curr} value are known. The ΔV_{curr} value for a particular column is found using the table in Figure 2·3. The input is the ΔH_{prev} value in the same column and the ΔV_{curr} value in the column to the left, *except*, when there is a Match, the value in the column to the left is treated as a max and, starting with Step 3, if the value in the column to the left is not assigned, it is treated as mid. $\Delta H_{prev}\dagger$is a modification of ΔH_{prev} in which all Match positions have been changed to max and all values less than mid have been changed to mid. The ΔH_{curr} value for a particular column is found using the transpose of the table in Figure 2·3. The input is the $\Delta H_{prev}\dagger$in the same column and the ΔV_{curr} value in the column to the left.

	1	1	1	1 1	1		Matches
AND	1110	1110		111110		1110	ΔH_{min}
	0100	1000		010100		0000	ΔV_{max} (initial)
+	1110	1110		111110		1110	ΔH_{min}
	1001	0001		100**1**01		1110	
XOR	1110	1110		111110		1110	ΔH_{min}
	0111	1111		011**0**11		0000	
XOR	0100	1000		010100		0000	ΔV_{max} (initial)
	0011	0111		001111		0000	>> ΔV_{max} (final and shifted)

Example Code:

```
INITpos7 = DHneg5 & Matches;
DVpos7shift = ((INITpos7 + DHneg5) ^ DHneg5) ^ INITpos7;
```

Figure 2·5: Finding ΔV_{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 ΔH_{min} and the first position to the right of each run. Symbol >> indicates that the final ΔV_{max} values are shifted to the right one position. Bits erroneously set by the ADD (+) are shown in bold. Sample code is from the complete listing in Supplementary Information.

$$\begin{array}{llllll}
 & 1110 & & 1110 & 11101110 & \Delta H_{\min}\ \text{(remaining)} \\
+ & 1 & 1 & 1 & 1\quad \text{X} & >> \Delta V\ \text{(initial shifted)} \\
\hline
 & 0001 & 1 & 0001 & 00011110 & \\
\text{XOR} & 1110 & & 1110 & 11101110 & \Delta H_{\min}\ \text{(remaining)} \\
\hline
 & 1111 & 1 & 1111 & 11110000 & >> \Delta V\ \text{(final and shifted)}
\end{array}$$

Example Code:

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

Figure 2·6: 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.

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);
```

Figure 2·7: Code for Zones B, C and D.

ΔV	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
Encoded	0	1	2	3	4	5	6	7	8	9	10	11	12
ΔH	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
Encoded	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12

ΔV vectors	BitPAl
-5	1 0 0 0 0 0 0 0 0
-4	0 1 0 0 0 0 0 0 0
-3	0 0 1 0 0 0 0 0 0
-2	0 0 0 1 0 0 0 0 0
⋮	⋮
7	0 0 0 0 0 0 0 0 1
True Value	-5-4-3-2 1 2 3 5 7

ΔV Binary place value vectors	BitPAlPacked vectors
1	0 1 0 1 0 1 0 0 0
2	0 0 1 1 1 1 0 1 0
4	0 0 0 0 1 1 0 0 1
8	0 0 0 0 0 0 1 1 1
sign bit	0 0 0 0 0 0 0 0 0
True Value	-5-4-3-2 1 2 3 5 7

$\texttt{carry}_1 = \texttt{a}_1 \,\&\, \texttt{b}_1;$

$\texttt{aplusb}_2 = (\texttt{a}_2 \wedge \texttt{b}_2) \wedge \texttt{carry}_1;$

$\texttt{carry}_2 = (\texttt{a}_2 \,\&\, \texttt{b}_2) | ((\texttt{a}_2 \wedge \texttt{b}_2) \,\&\, \texttt{carry}_1));$

Figure 2·8: Top: The BitPAl Packed mapping of ΔH and ΔV values for the parameter set $M = 2, I = -3, G = -5$. Middle: Conversion from the thirteen ΔV_i vectors at left to the five "packed" vectors at right. Bottom: Example code for adding the packed representation.

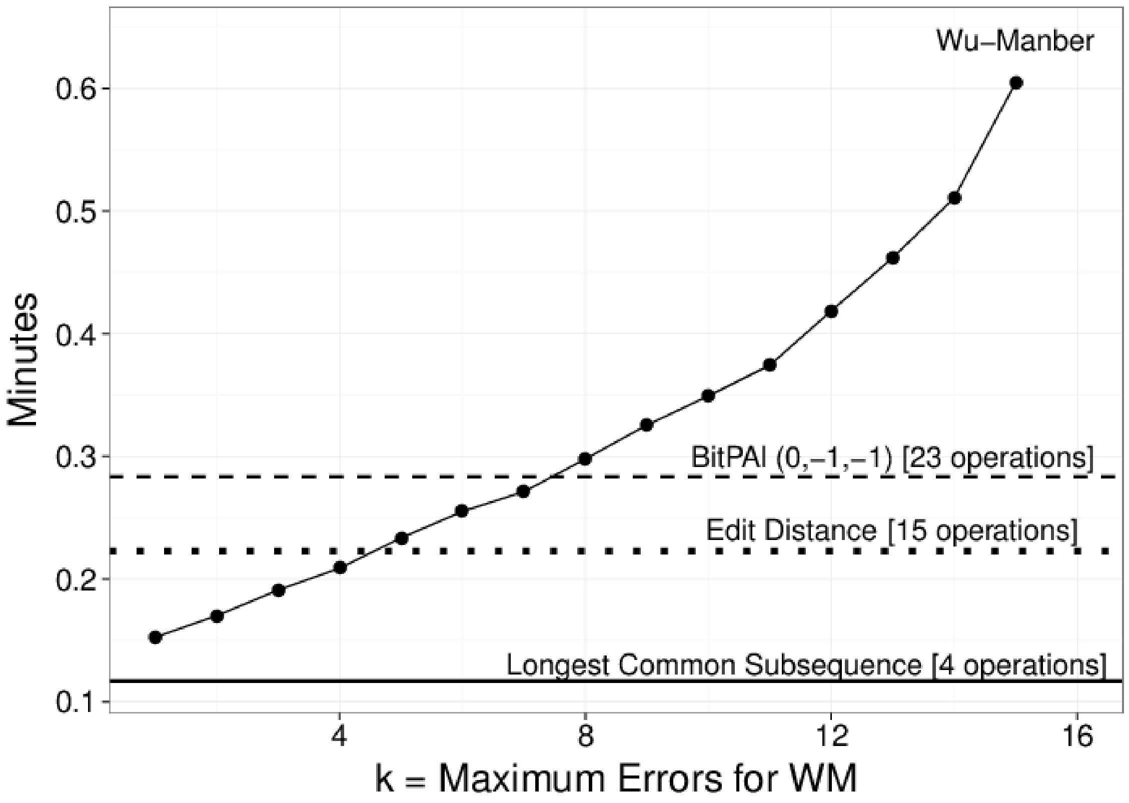


Figure 2·9: Running times. Each experiment involved 25 million alignments. For BitPAl, alignment weights (M, I, G) are shown in parenthesis. All times are averages of three runs. Unit-cost BitPAl, unit-cost WM, LCS, and ED. k is the maximum number of errors allowed for WM. k is not a parameter for the other algorithms and their times are shown as horizontal lines. LCS uses 4 bit operations per w cells, ED uses 15 bit operations, BitPAl (0, -1, -1) uses 23 bit operations. For $k = 7$, the times for BitPal and WM are nearly the same. By $k = 15$, BitPAl runs approximately twice as fast. Results for N are not shown on the graph. It was 118 to 304 times slower than BitPAl (0, -1, -1) even when optimal parameters were chosen.

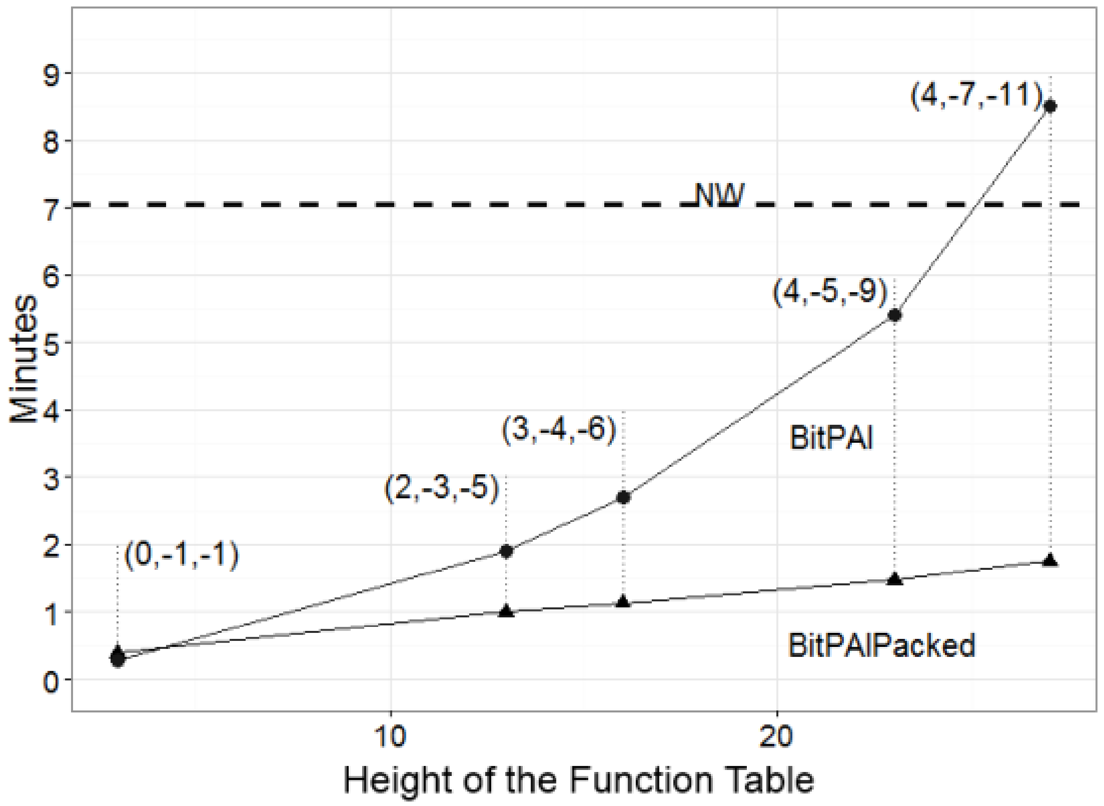


Figure 2·10: Running times. Each experiment involved 25 million alignments. For BitPAl and BitPAl Packed, alignment weights (M, I, G) are shown in parenthesis. All times are averages of three runs. Variants of BitPAl and NW (shown as a horizontal line). For BitPAl, time is approximately linearly proportional to one dimension of the function table. For BitPAl packed, time is approximately linearly proportional to the area of the function tables. BitPAl packed (2, -3, -5) is approximately 7.1 times faster than NW and BitPAl (0,-1,-1) is approximately 24.9 times faster.

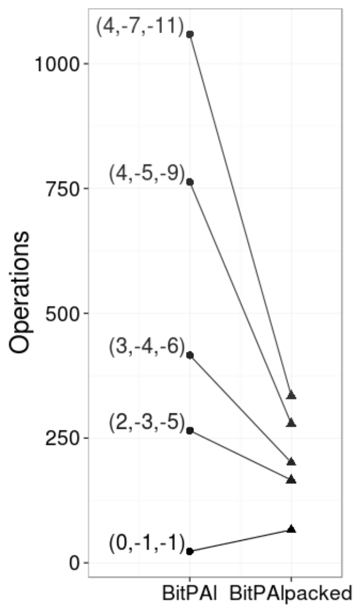


Figure 2·11: Comparison of the number of operations for BitPAl and BitPAl packed for different alignment weights (M, I, G).

Chapter 3

Bit-parallel and SIMD Methods for Global Alignment with Substitution Scoring

3.1 Introduction

In this chapter, we present bit-parallel and SIMD algorithms for similarity substitution scoring. For example, when aligning protein sequences, a BLOSUM (or PAM) protein amino acid substitution table [24, 25, 17] is commonly used. Such tables assign a weight to every pair of amino acids, (a, b). The weights are log odds scores for the substitution of a for b in reliable alignments produced from related protein sequences. Each weight is either a bonus (positive) or a penalty (negative) depending on whether the substitution is more or less likely than chance. The BLOSUM 62 table (the 62 means the table was generated from reliable alignments of protein sequences with $\geq 62\%$ identity) contains 15 distinct weights and for any given amino acid, a, there can be anywhere from four to nine different weights. Another common substitution table, used in DNA alignments, assigns different weights to *transitions*, substitutions of nucleotides with similar structures (A to G, C to T), and *transversions*, substitutions of nucleotides with different structures (A or G to C or T).

An earlier bit-parallel method by Navarro [50] allows arbitrary integer weights for substitutions as well as insertions and deletions of each character and finds occurrences, for both simple patterns and regular expressions, where the sum of the edit weights is at most k. While more flexible than our algorithms, it is much slower in practice [45]. An algorithm by Bergeron and Hamel [9] for distance scoring with arbitrary substitution weights and fixed gap weight uses an extension of the Myers [49] method, similar to our previous BitPAl method. However, no practical implementation was given.

SIMD techniques have been applied to alignment with substitution scoring, including Farrar's Needleman-Wunsch implementation [18] and Parasail (both Needleman-Wunsch and Smith-Waterman alignment with substitution scoring) [16]. These approaches use a direct dynamic programming approach to calculate the scoring matrix. They achieve their speedups via careful arrangement of storage and computation steps. However, storing the score directly means that score values can overflow the available SIMD storage space. When overflows occur, recomputation with a larger SIMD storage size is necessary. Our SIMD method instead stores and uses score differences, guaranteeing that overflows do not occur in a deleterious manner and avoiding recomputation.

The remainder of the chapter is organized as follows. In Section 3.2 we state the problem and give definitions and preliminary ideas. In Section 3.3.2, we describe an extension to our bit-parallel algorithm of Chapter 2 and in Section 3.3.3 and Section 3.3.4 we describe two new SIMD algorithms. In Section 3.5 we give results of experiments comparing our algorithms with iterative dynamic programming and SIMD accelerated dynamic programming.

3.2 Problem Description

Problem: Given two sequences X and Y, of length m and n respectively, a similarity scoring function, S, defined by a negative, integer gap weight, G, and a table of integer substitution weights, $subst(a, b)$, defined over character pairs (a, b) from the alphabet Σ, where for every pair $subst() > 2G$, calculate the global alignment score for X and Y using bit operations, addition, and max/min comparisons on computer words of length w.

We allow two types of initialization in the zero row and column of the alignment scoring matrix, 1) no penalty for an initial gap (zero in every cell), or 2) penalty for an initial gap (gap weight G added to each successive cell). We do not restrict the size of the alphabet, although the time complexity depends, in part, on the alphabet size as a result of required pre-processing of the $subst()$ table. The requirement that $subst() > 2G$ assures that every substitution is possible (*i.e.*, that it will not be precluded by two consecutive deletions).

Definitions and notation. Let S be a recursively-defined, similarity scoring function for global alignment for two sequences $X = x_1x_2 \ldots x_m$ and $Y = y_1y_2 \ldots y_n$:

$$S[i,j] = \max \begin{cases} S[i-1,j-1] + subst(x_i,y_j) & \text{substitute } x_i \text{ with } y_j \\ S[i-1,j] + G & \text{delete } x_i \\ S[i,j-1] + G & \text{delete } y_j \end{cases} \tag{3.1}$$

$$S[0,j] = j * G, \quad S[i,0] = i * G$$

where, in this case, we have defined a deletion penalty for an initial gap in the alignment.

As in Chapter 2, instead of the actual scores in S, we store only the differences between scores in adjacent cells, *i.e.*, Δv is the (vertical) difference between the scores in a cell and the cell above, and Δh is the (horizontal) difference between the scores in a cell and the cell to its left:

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

To simplify algorithmic explanation, for the remainder of this chapter we map Δv and Δh into new variables ΔV and ΔH using the formulas:

$$\Delta V = \Delta v - G, \quad \Delta H = G - \Delta h. \tag{3.2}$$

For convenience in the explanation, while referring to the current row i, we drop the first index in every term. ΔH terms for rows $i-1$ and i are labeled ΔH_{in} and ΔH_{out} respectively.

For a fixed letter x in sequence X and any position j in sequence Y, we define

$$L[j] = subst(x, y_j) - 2G.$$

As we will see below, $L[j]$ serves as a lower bound on $\Delta V[i, j-1]$ when computing $\Delta V[i,j]$ for column j. We further define maximum and minimum values for the $L[j]$ over all letters x:

$$L_{max} = \max_{x,j} L[j], \quad L_{min} = \min_{x,j} L[j]$$

Relationship between input and output values. The first two theorems give the ranges for ΔV and ΔH and formulas for computing the values of ΔV and ΔH_{out}. Proofs for all theorems are given in the Appendices.

Theorem 3.2.1. *ΔV and ΔH are integers which fall in the following ascending and* *descending ranges respectively:*

$$\begin{aligned}\Delta V &\in \{0, 1, 2, \ldots, L_{max}\}\\ \Delta H &\in \{0, -1, -2, \ldots, -L_{max}\}. \end{aligned}\tag{3.3}$$

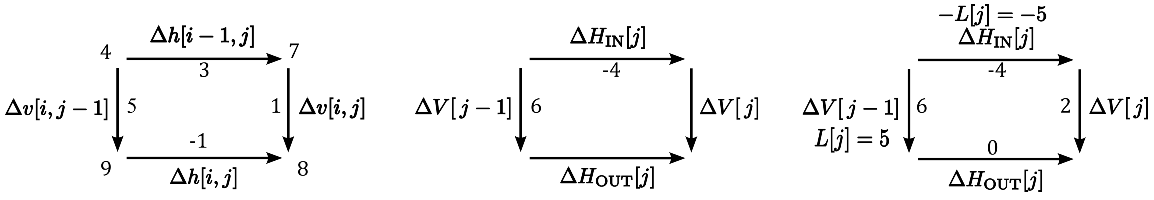


Figure 3·1: Left: Actual alignment scores are shown in the corners. Δv and Δh are the differences between scores in adjacent cells and are shown along the arrows. **Middle:** Actual differences have been mapped into the variables ΔV and ΔH using Equations (3.2) with $G = -1$. ΔH values are labeled ΔH_{in} for row $i-1$ and ΔH_{out} for the current row, i. The row i indices have been dropped for ΔV. **Right:** Substituting $\Delta H_{in}[j] = -4$, $\Delta V[j-1] = 6$, and $L[j] = 5$ into Equation (3.4) yields $\Delta V[j] = 2$. Note that $\Delta V[j-1] > L[j]$ and the sum is > 0. Substituting the same values into Equation (3.5), note that $\Delta H_{in}[j] > -L[j]$ so $\Delta H_{in}[j]$ is not used. Also the sum is > 0, so $\Delta H_{out}[j] = 0$, not 1.

Theorem 3.2.2. *$\forall j \geq 1$, ΔV and ΔH are computed by the following formulas (the meaning of the underlined parts is explained below):*

$$\Delta V[j] = \max\Big(0,\ \underline{\max\big(\Delta V[j-1], L[j]\big)} + \Delta H_{in}[j]\Big) \tag{3.4}$$

$$\Delta H_{out}[j] = \min\Big(0,\ \underline{\min\big(-L[j], \Delta H_{in}[j]\big)} + \Delta V[j-1]\Big). \tag{3.5}$$

Note that the underlined part in each sum yields a restricted range which depends on $L[j]$ and L_{max}.

$$\underline{\max\big(\Delta V[j-1], L[j]\big)} \in \{L(j), \ldots, L_{max}\} \tag{3.6}$$

$$\underline{\min\big(-L[j], \Delta H_{in}[j]\big)} \in \{-L(j), \ldots, -L_{max}\}. \tag{3.7}$$

3.3 Algorithms

Our goal is to calculate the ΔH_{out} values in row i from:

- ΔH_{in} values in row $i-1$,
- $\Delta V[0]$, the leftmost ΔV value in row i, and
- $L[j]$ values for row character x_i.

The ΔH values in row zero and the ΔV values in column zero depend on the type of global alignment, as mentioned above, but are known in advance. The L values are computed in a pre-processing step (outlined in Section 3.4) so that for any given row character x, we have the appropriate L values available.

We describe three algorithms. The first is an extension of the bit-parallel algorithms (BitPAl) presented in Chapter 2, the second is a new method based on partial sums of ΔH_{in} values, and the third modifies the second by introducing a more efficient parallel scan. The main obstacle for all is determining the missing ΔV values because of the left-to-right dependency. Once the ΔV values for the current row have been computed, determining the ΔH_{out} values is straightforward.

3.3.1 Data Structures

We use three data structures to store the ΔH and ΔV values (Figure 3·2). The first two are used in the BitPAl extension method and the third is used in the Partial Sums method:

- a vector structure where each possible value is stored in its own computer word and where each bit represents a single column in the alignment scoring matrix.
- a "packed" structure where the binary representations of the values are stored together in a set of k vectors $b_0, b_1, \ldots, b_{k-1}$ where the lth vector represents the

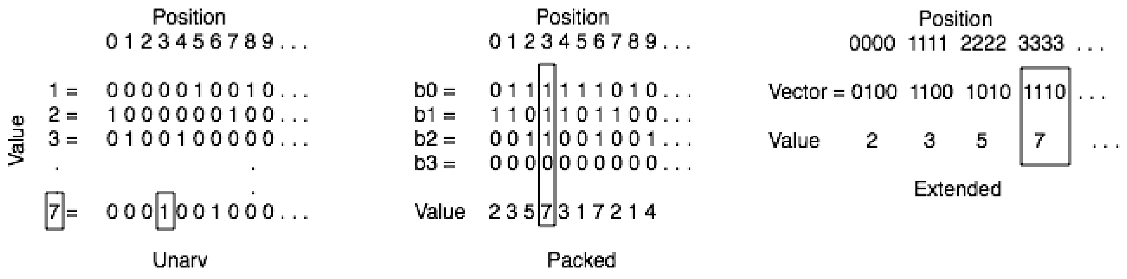


Figure 3·2: Three representations for Δ values. Boxed bits represent the same value in the same column. The BitPAl Extension method uses the vector and Packed representations. The Partial Sums method used the Extended representation.

value 2^l and where each one-bit-wide "slice" through the vectors represents a single column in the alignment scoring matrix.

- an "extended" data structure where the binary representation is stored in a block of k bits within a single computer word and each block represents a single column in the alignment scoring matrix.

The vector structure is best for finding columns containing a specific value and the packed structure is good for adding all values simultaneously, but each such addition is high cost. Conversion between the two is efficient. The extended structure adds all values very cheaply (one operation), however, it gives up the density of representation of the other two data structures. In what follows, we will assume that for the extended data structure, $k = 8$, *i.e.*, that each value is stored in a byte, in order to use efficient SIMD instructions (Single Instruction Multiple Data), but other values of k are possible.

3.3.2 BitPAl Extension for BLOSUM-type scoring.

The BitPAl algorithm of Chapter 2 applies to alignments with three weights, one each for match, M, mismatch, I, and gap, G. In effect, there are two L values,

$$\begin{aligned} L_{max} &= L(x,x) = M - 2G \text{ (for any character } x\text{)} \\ L_{min} &= L(x,y) = I - 2G \text{ (for any two different characters } x \text{ and } y\text{).} \end{aligned}$$

Theorem 3.2.2 defines a series of Function Tables for the output ΔV values, one table for each L value. Similarly defined are the Function Tables for ΔH_{out}. Figure 3·3 shows Function Tables derived from a *subst*() table with three L values. For a fixed row character x, the Function Table for each column characters y_j is determined solely by $L(j)$.

The key idea from this algorithm is to compute the columns with a single specific ΔV value all at once even if they are not contiguous. We start with $\Delta V = L_{max}$, and work down, in order, to $\Delta V = L_{min}$. Referring to the bottom left example in Figure 3·3 and assuming that $L_{max} = L_1$ and $L_{min} = L_2$, first the columns with output value $7 = L_{max}$ are determined, then using those as input, the columns with output value $6 = L_{min} + 1$ are determined. (This is Zone A2). The remaining unknown ΔV values are all $\leq 5 = L_{min}$, so, according to equation (3.4) we use L_{min} in the sum at those columns and compute all the remaining ΔV values. At this point, we have all the information required to compute the ΔH_{out} values for the current row. Full details are given in [45].

The extension for BLOSUM-type scoring assumes that there are more than two L values (bottom right example in Figure 3·3). Once the columns with output value $6 = L_2 + 1$ are determined, the remaining unknown ΔV values are all $\leq 5 = L_2$. But only for some of them can we use L_2 in the sum of equation (3.4). For those columns where $2 = L_3$ is the minimum, we need to keep computing output values in decreasing order, *i.e.*, $5, 4, 3$. (This is the remainder of Zone A3.) Finally, we use $L_3 = L_{min}$ as the input value for all remaining unknown columns and finish computing the ΔV values. Distinguishing the columns that should stop at L_2 and those that proceed to L_3 is done with a masking operation using the information stored in $L[j]$.

The weakness of this algorithm is the dependence of the time on the size of the largest Zone A. For a large Function Table, the computation cost is high. The second and third methods reduce the cost for large tables.

3.3.3 SIMDParSums: Method of Partial Sums of ΔH

The key idea in this algorithm is to initially compute the $\Delta V[j]$ values as though they were the result of a substitution or a vertical gap, *i.e.*, as though these values come directly from the row above. The initial values, which we denote $\overline{\Delta V}[j]$ are lower bounds on the true $\Delta V[j]$ since some are actually the result of a horizontal gap. Then, in a logarithmic number of rounds, we allow the existing $\overline{\Delta V}[j]$, some of which are already correct, to propagate through horizontal gaps and improve the value of other $\overline{\Delta V}[j]$. The moniker "SIMDParSums" (SIMD Partial Sums) comes from the requirement to compute sums of contiguous intervals of ΔH_{in} values for the propagation step. To make this more concrete, we state the following theorem.

Theorem 3.3.1. $\forall j > 0$*, the* $\Delta V[j]$ *values can be computed by the following recur-*

rence:

$$\Delta V[1] = \max\Big(0, \max\big(\Delta V[0], L[1]\big) + \Delta H_{in}[1]\Big)$$

$$\underset{\forall j>1}{\Delta V}[j] = \tag{3.8}$$

$$\max\begin{cases} 0 \\ L[j] + \Delta H_{in}[j] \\ L[j-1] + \Delta H_{in}[j-1] + \Delta H_{in}[j] \\ L[j-2] + \Delta H_{in}[j-2] + \Delta H_{in}[j-1] + \\ \qquad\qquad\qquad\qquad \Delta H_{in}[j] \\ \vdots \\ \max\big(\Delta V[0], L[1]\big) + \Delta H_{in}[1] + \ldots \\ \qquad\qquad + \Delta H_{in}[j-1] + \Delta H_{in}[j] \end{cases} \tag{3.9}$$

Note that in the recurrence, 0 represents a vertical gap (because ΔV is defined as $\Delta v + G$) and $L[j] + \Delta H_{in}[j]$ represents a substitution. All the remaining alternatives represent horizontal gaps arising from the left and using partial sums of the ΔH_{in}. Since we do not know the length of the horizontal gap (if any) which gives any particular $\Delta V[j]$, we need to consider all the possibilities. We do this in $\lceil \log_2 n \rceil + 1$ rounds of calculation. In the algorithm, round zero computes the initial $\overline{\Delta V}[j]$, derived as either a vertical gap or a substitution:

$$\underset{\forall j \geq 1}{\overline{\Delta V}}[j] = \max(0, L[j] + \Delta H_{in}[j]). \tag{3.10}$$

In round $i, 1 \leq i \leq \lceil \log_2(n) \rceil$ the $\overline{\Delta V}[j]$ are updated so that they contain a maximum value originating from one of the columns $j - 2^i, \ldots, j$. This requires adding a partial sum of 2^{i-1} terms of ΔH_{in}:

$$\underset{\forall j \geq 2^{i-1}}{\overline{\Delta V}[j]} = \max \begin{cases} \overline{\Delta V}[j] \\ \overline{\Delta V}[j - 2^{i-1}] + \Delta H_{in}[j - 2^{i-1} + 1] + \\ \qquad\qquad\qquad \ldots + \Delta H_{in}[j] \end{cases} \tag{3.11}$$

Note that this method avoids the SIMD overflow/underflow problems of [16, 18] because the sum of ΔH_{in} values in Equation (3.11) only affects the outcome when it is greater than or equal to $-L_{max}$. This is because in Equation 3.10, the minimum possible value for $\overline{\Delta V}$ is 0. At round i, if

$$\Delta H_{in}[j - 2^{i-1} + 1] + \ldots + \Delta H_{in}[j] < -L_{max},$$

then

$$\overline{\Delta V}[j - 2^{i-1}] + \Delta H_{in}[j - 2^{i-1} + 1] + \ldots + \Delta H_{in}[j] < 0,$$

since $\overline{\Delta V}[j - 2^{i-1}] \leq L_{max}$. Since the SIMD registers can hold everything in the range $-L_{max}, L_{max}$, this underflow doesn't affect our algorithm. Also, since all ΔH_{in} values are less than or equal to 0, in round $i + 1$, we know that if

$$\Delta H_{in}[j - 2^{i-1} + 1] + \ldots + \Delta H_{in}[j] < -L_{max},$$

then

$$\Delta H_{in}[j - 2^{i} + 1] + \ldots + \Delta H_{in}[j] < -L_{max},$$

which implies that

$$\overline{\Delta V}[j - 2^{i}] + \Delta H_{in}[j - 2^{i} + 1] + \ldots + \Delta H_{in}[j] < 0.$$

This means that a underflow in the sum of ΔH_{in} values during one round will never cause an incorrect ΔV value in a later round.

The calculation of the partial sums and the maximum calculation for all j can be performed using linear work in each round. This is formally stated in the following two Theorems:

Theorem 3.3.2. *$\forall j \in \{1, \ldots, n\}, \forall i \in \{0, \ldots, \log_2 j\}$, partial sums of length 2^i, $PS[j,i] = \Delta H_{in}[j - 2^i + 1] + \Delta H_{in}[j - 2^i + 2] + \ldots + \Delta H_{in}[j]$ can be computed in $\lceil \log_2 n \rceil$ rounds in $O(n \log n)$ time.*

Theorem 3.3.3. *$\forall j \in \{1, \ldots, n\}$, $\Delta V[j]$ can be computed in $\lceil \log_2(n) \rceil + 1$ rounds in $O(n \log n)$ time if $\forall j \in \{1, \ldots, n\}, \forall i \in \{0, \ldots, \log_2 j\}$ the partial sums $PS[j,i]$ are available.*

In our bit parallel algorithm, we store multiple ΔV values in a single word of length w. Our implementation uses SIMD instructions and word length $w = 128$ bits. With $k = 8$, this yields $W = 128/8 = 16$ values per word. The advantages of using SIMD instructions are 1) longer word length w, 2) the ability to do independent, parallel computations on values stored in consecutive bytes within each word, and 3) the availability of max and min functions that compare two values and save the best in a single operation. Theorem 3.3.4 shows that the computation time is sub-linear when using multiple values per word for a sequence of length n. Pseudocode for the SIMDParSums algorithm is shown as Algorithm 1 in Appendix B.

Theorem 3.3.4. *$\forall j \in \{1, \ldots, n)$, $\Delta V[j]$ and $\Delta H_{out}[j]$ can be computed in time $O\left(n \log(W)/W\right)$, where W is the number of values held in a word of length w.*

3.3.4 Method of Partial Sums by ***Striped*** Scan

In Section 3.3.3, SIMDParSum completes the partial sum on each of the n/W SIMD words sequentially. The sum is passed from one word to the next, resulting in $O(\log(W))$ work for each word and $O(\log(W) \cdot n/W)$ in total per row. There are two problems with this method. The first is that only a single SIMD word is being accessed, manipulated, and stored into at any given time. This reduces processor pipeline efficiency, because it creates blocks of sequentially dependent operations. The second is that each time a word is shifted, the addition involves fewer and fewer of the values in the word (wasted operations). We will introduce two techniques that can solve these problems.

The first is a striped data structure that was first described by [18] in an SIMD implementation of the Smith-Waterman algorithm [54] to prevent intra-word dependencies. It was also used in the SIMD alignment algorithm Parasail [16]. *Striping* values across words allows addition of different SIMD words to each other to have the effect of shifting without having operations wasted due to uninvolved values.

Definition 1. *'Striped' data storage - for a given data set of n elements, given a SIMD word that can store W values, we will store the data in $g = \lceil n/W \rceil$ words such that the zeroth value is in the zeroth position of the zeroth word, the first value in the zeroth position of the first word, ..., the gth value is in the first position of the zeroth word, and so on such that the kth value is in the k modulo g word in the $\lfloor k/g \rfloor$ position.*

Parallel prefix sums (also referred to as a *scan*) are frequently encountered in parallel algorithms, and efficient methods for computing them are addressed in [10]. We combine the *striped* data structure of [18] with a modified version of [10] to present an efficient *striped* SIMD scan for the partial sums calculations described in Section 3.3.3. The definitions and foundational theorems on the relationships between values remain the same as in SIMDParSum - the change is in how the partial sums of values are computed, resulting in a reduction in the number of redundant computations.

Blelloch's scan [10] proceeds in two parts - an upsweep and a downsweep, with operations modeled on an imaginary binary tree structure. We adapt Blelloch's scan to striped SIMD data. In the upsweep, pairs of values are summed and stored at each level of the tree, as shown in Figure 3·5. The down-sweep, in which the final word is distributed to the rest of the words, is shown in Figure 3·6. The details of our implementation are given in Appendix C.

3.4 Complexity and Space

3.4.1 BitPAl Extension Method

The time complexity excluding the pre-processing is

$$O\left(\frac{znm}{w}\right),$$

where z is proportional to the work for one computer word in one row of the alignment scoring matrix.

$$z = ((L_{max} - L_{min})^2)/2 + \log(2 * L_{max}).$$

The first term covers computing all ΔV values in the largest Zone A and the second term covers computing all the remaining ΔV values and the ΔH_{out} values using a manually constructed addition on the packed data structure.

3.4.2 SIMDParSum

Storing the $L[j]$ values. Pre-processing involves storing the $L[j]$ values for each possible row character x. We have tested two methods. In the first, the sequence Y is scanned one character, y_j, at a time and for each $x \in \Sigma$, we store $L(x, y_j)$ in the appropriate position of the L variables for x. The time required is $O(|\Sigma|n)$.

The second method is useful for smaller alphabets with $|\Sigma| < W$. Y is first scanned in linear time to find the columns of every character $\sigma \in \Sigma$. The column indices are stored in $|\Sigma|$ separate variables, Loc_σ. For each character $x \in \Sigma$, we determine the set of σ which share the same $L[x, \sigma]$ value and then store that value in positions determined by performing ORs of the individual Loc_σ variables. The time required is $O(|\Sigma|^2 n/W)$.

In both cases, the space required for the $L[j]$ values is $|\Sigma|n/W$. For the ΔH and ΔV values, n/W space is required for each.

Post-processing involves retrieving the alignment score from the final ΔH_{out} values and is the same method as described in [45]. The time required is $O(n)$.

The time complexity of our algorithm, excluding the pre- and post-processing, is

$$O\left(\frac{mn\log W}{W}\right).$$

m represents the number of rows that must be calculated, n/W is the number of words that are calculated in each row, and $\log W$ is proportional to the number of operations for each word.

3.4.3 SIMDScan

The time complexity for storing the L values and retrieving the alignment score remains the same as in SIMDParSum. The time complexity of the main body of the algorithm is

$$O\left(m\left[\frac{n}{W} + \log W\right]\right).$$

m represents the number of rows that must be calculated, n/W is the number of words that are calculated in each row as the scan time (except the end of the upsweep in the last word) is linear in the number of words, and $\log W$ is proportional to the number of operations done by the end of the upsweep in the final word.

3.5 Experimental Results

We compared the running times of our algorithms against the Needleman-Wunsch iterative dynamic programming algorithm [51] and Parasail, an accelerated implementation of NW using SIMD [16, 15]. While our algorithm uses a linear gap penalty, Parasail uses an affine gap penalty. This gives our algorithm a slight advan-

$\|X\|$	1	20	63	100	150
NW	0.248	3.044	11.818	18.502	24.352
Parasail	2.316	2.583	3.041	3.492	4.105
SIMDParSums	0.819	1.075	1.659	2.171	2.852
SIMDParSumScan	0.843	1.010	1.435	1.757	2.232
SIMDParSumAffine	1.048	1.286	1.811	2.243	2.855

(a) 1 to 1

$\|X\|$	1	20	63	100	150
NW	0.118	2.810	11.185	16.649	22.192
Parasail	0.409	0.813	1.703	2.482	3.564
SIMDParSums	0.027	0.293	0.867	1.413	2.088
SIMDParSumScan	0.068	0.245	0.654	0.998	1.473
SIMDParSumAffine	0.070	0.290	0.780	1.208	1.849

(b) 1 to 100

Table 3.1: Tables of run times, in minutes, for 25 million alignments. **Top:** A different pair of sequences was used for each alignment (1 to 1). **Bottom:** Each Y sequence was aligned against 100 X sequences (1 to 100). Note that both tables include run times for the SIMDParSumAffine algorithm introduced in Chapter 4.

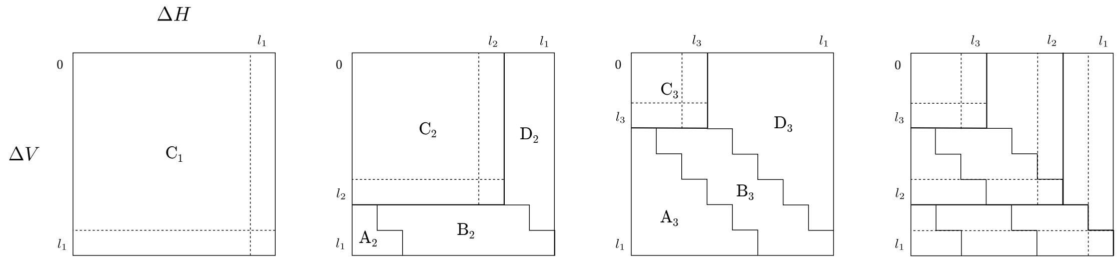
tage, since it does not have to do the additional calculations for the affine gap. For our algorithm, we used a gap penalty of -6. For Parasail we used a gap open penalty of -11, a gap extend penalty of -1, the recommended "scan" version of the algorithm with SSE 4.1 instructions, and the 16 bit wide data-structures to avoid score overflows which occurred with the 8 bit data-structures. We used the BLOSUM 62 similarity table (for a 23 character amino acid alphabet with 15 L values, $L_{max} = 23$, $L_{min} = 8$)). The algorithms are designated: 1) PartialSums (Partial Sums SIMD, BLOSUM scoring), 2) SIMDScan (Partial Sums SIMD Scan, BLOSUM scoring), 3) PARASAIL (Parasail, BLOSUM scoring), and 4) NW (Needleman-Wunsch).

For all experiments, we performed 25 million alignments, using randomly generated amino acid sequences. The length of sequence Y (along the top of the alignment scoring matrix which defines the number of columns) was 126. At this length, the SIMDParSums and SIMDParSumsScan algorithms use eight words. Five lengths were used for sequence X (along the left side of the alignment scoring matrix which defines the number of rows), $|X| = 1, 20, 63, 100, 150$. $|X| = 1$ was used to estimate the pre-processing overhead for each algorithm. The experiments were divided into two sets. In the first, a new pair of sequences was generated for each alignment (denoted "1 to 1"). In the second, each sequence Y was aligned, one at a time, against 100 newly generated X sequences (denoted "1 to 100"). The 1 to 100 experiment models the task of aligning a query sequence against a large number of candidate matches. It also amortizes the pre-processing cost for sequence Y over the 100 alignments, reducing the impact of pre-processing.

All programs were compiled with GCC using optimization level O3 and `march = native` (for SIMD commands) and run on an Intel Core i7-4710HQ CPU 2.50 - 3.5GHz CPU running Ubuntu Linux 14.04. Results are shown in Figures 3·7 and 3·8 and Table **??**. As can be seen, our new algorithms are faster than NW at all but very short sequence X lengths. In the 1 to 1 experiment, SIMDParSums and SIMDParSumsScan are 30% faster and 46% faster than Parasail, respectively. In the 1 to 100 experiment, the pre-processing costs for the SIMDParSums and SIMDParSumsScan algorithms have become insignificant and they are 41% and 59% faster than Parasail, respectively. This is a result of the fact that more of the work of the SIMDParSums and SIMDParSumsScan algorithms lie in the pre-processing step.

3.6 Discussion

We have developed a new algorithm that extends bit-parallel alignment and two new SIMD algorithms for the case of global similarity alignment with a single gap penalty and a table for variably weighted substitutions. The first algorithm is an extension of our previous work on general integer scoring bit-parallel global alignment (BitPAl [8, 45]) and the other two are new approaches based on partial sums of horizontal score differences in the alignment scoring matrix using commonly available SIMD instructions. The BitPAl extension method requires a completely different program for each different similarity table because the operations depend on the size and characteristics of the Function Table which relates the input and output alignment score differences. The SIMD programs are simpler and for different similarity tables only differ in the number of operations required in a pre-processing step. Our SIMD-ParSumScan algorithm is currently the fastest known algorithm for global alignment with substitution scoring.



ΔH (columns), ΔV (rows):

ΔV \ ΔH	0	-1	-2	-3	-4	-5	-6	-7
0	5	4	3	2	1	0	0	0
1	5	4	3	2	1	0	0	0
2	5	4	3	2	1	0	0	0
3	5	4	3	2	1	0	0	0
4	5	4	3	2	1	0	0	0
$l_2 = 5$	5	4	3	2	1	0	0	0
6	6	5	4	3	2	1	0	0
$l_1 = 7$	7	6	5	4	3	2	1	0

ΔV \ ΔH	0	-1	-2	-3	-4	-5	-6	-7
0	2	1	0	0	0	0	0	0
1	2	1	0	0	0	0	0	0
$l_3 = 2$	2	1	0	0	0	0	0	0
3	3	2	1	0	0	0	0	0
4	4	3	2	1	0	0	0	0
$l_2 = 5$	5	4	3	2	1	0	0	0
6	6	5	4	3	2	1	0	0
$l_1 = 7$	7	6	5	4	3	2	1	0

Figure 3·3: Upper: Schematic of ΔV output Function Tables for three L values, denoted $l_1 = 7, l_2 = 5$, $l_3 = 2$. Note that the L values and the relation $\Delta H = \Delta V$ set the boundaries for the marked zones in the tables. These zones contain $(\Delta H, \Delta V)$ pairs which determine the source of the best score of a cell in S, either from a horizontal gap (Zones A and B), a substitution (Zone C), or a vertical gap (Zone D). Borders between zones, indicated by dotted lines, yield ties for the best score. Each Table applies to the subset of columns j that share a common $L(j)$. **Lower:** Actual values in the l_2 and l_3 Function Tables. Note in the left table, for example, when the ΔV input falls below l_2 the output remains the same, *i.e.*, l_2 is the minimum $\Delta V[j-1]$ value. Compare with Equation (3.4).

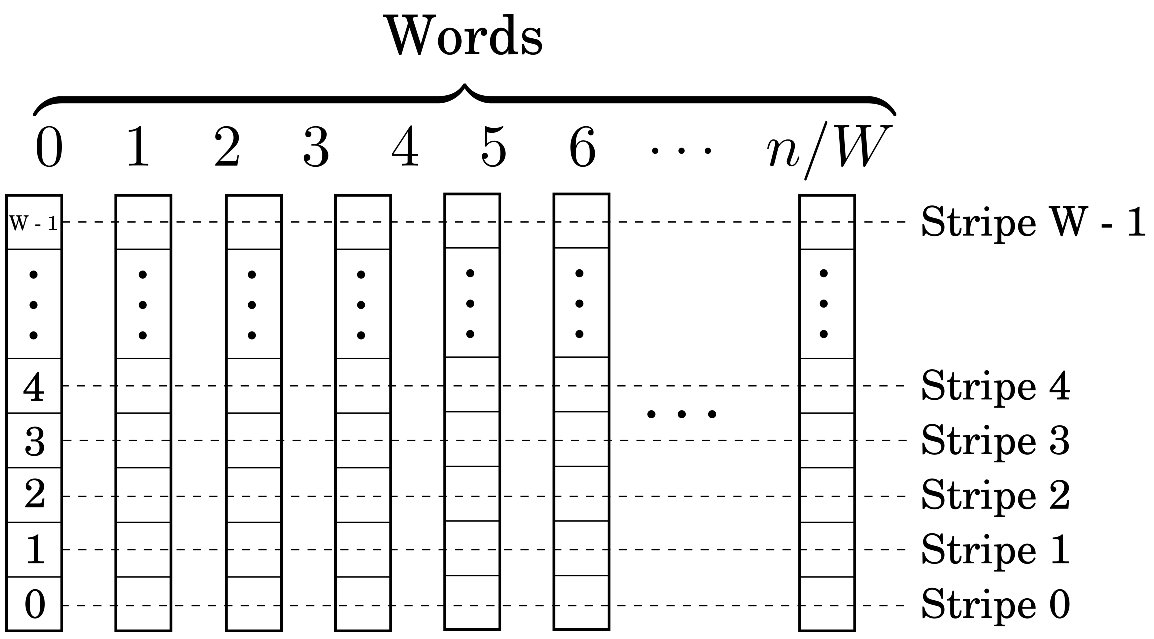


Figure 3·4: An example of how values are stored across SIMD words in the *striped* format.

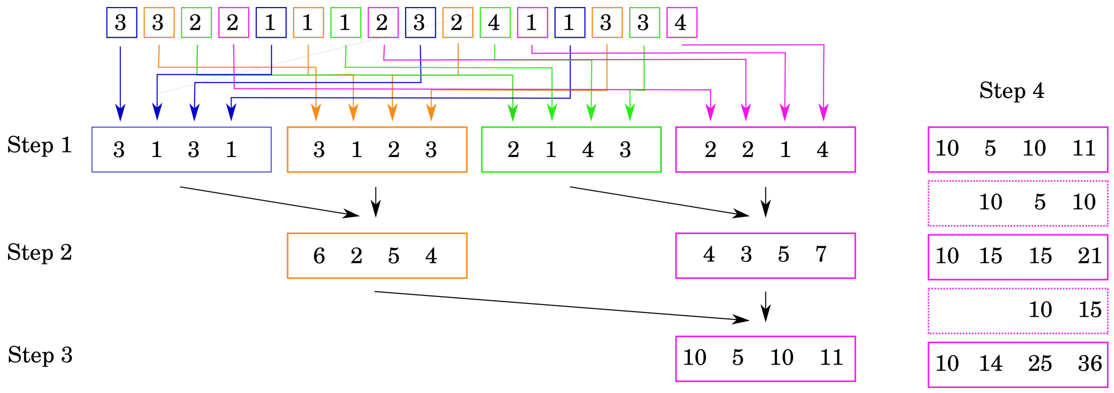


Figure 3·5: Upsweep step of SIMD prefix-sum computation. In this example, each SIMD word holds 4 values. **Step 1**: the values are *striped* across the SMID words. **Step 2**: pairs of words are added. **Step 3**: pairs at the next level of the tree are added. This process continues recursively until there is only one pair, the final SIMD word and the middle word. **Step 4**: the upsweep is continued by doing a parallel scan on the final SIMD word.

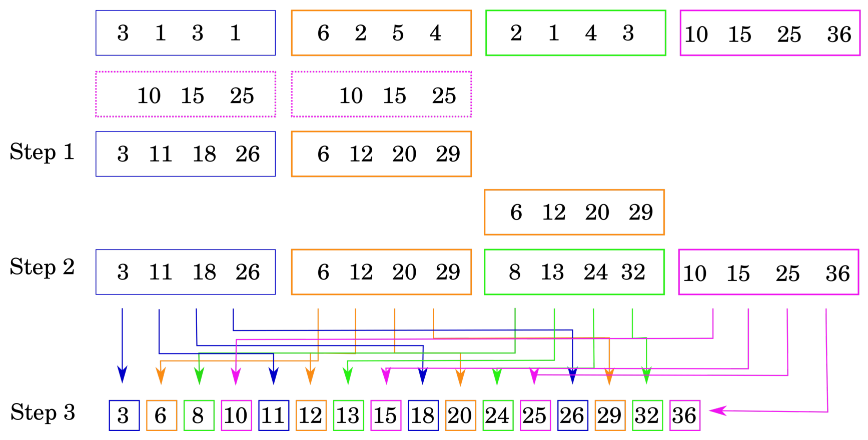


Figure 3·6: Downsweep step of SIMD prefix-sum computation. In this example, each SIMD word holds 4 values. The upsweep step has already been completed, the final word holds prefix-sums from the beginning for the 4 positions stored there. **Step 1**: the final word is shifted, and added to the first and second words. **Step 2**: The second word is added to the third. **Step 3**: This shows how values can be reordered (unstriped) into their original positions, but this is not actually done in each row.

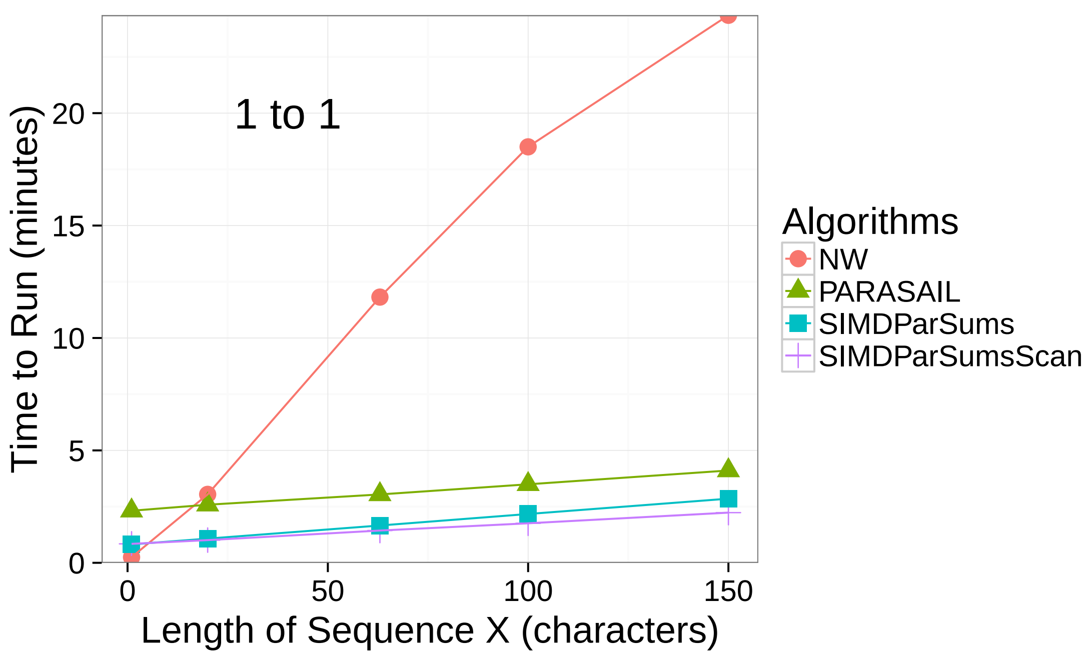
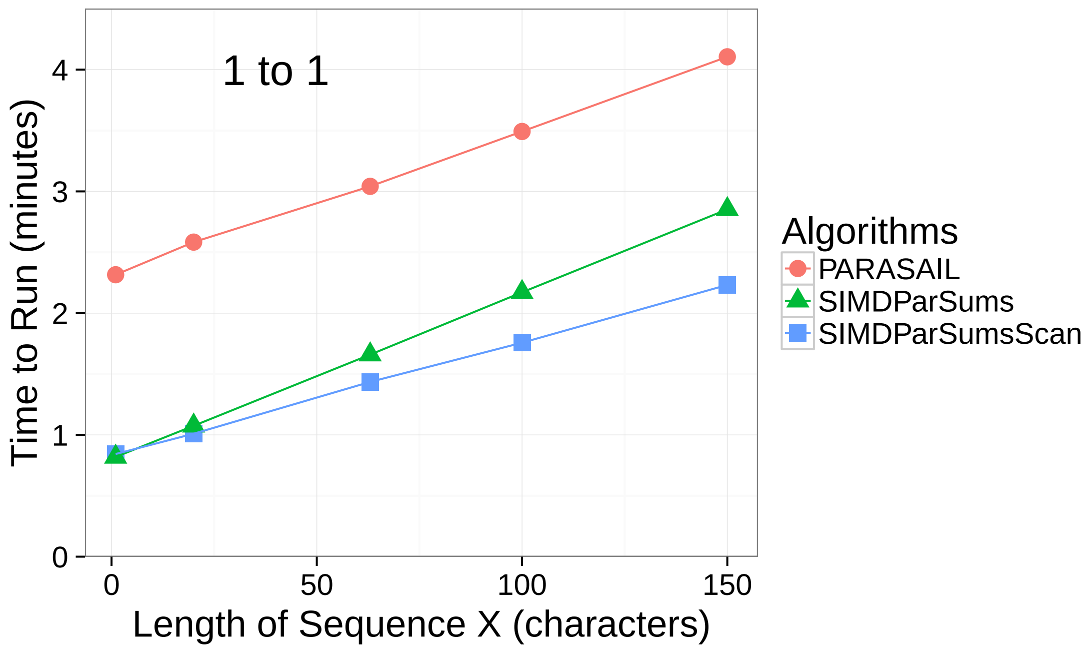


Figure 3·7: Comparison of algorithm run times for 25 million alignments. Shown are averages over three trials. $|Y| = 126$. A new pair of sequences was generated for each alignment (1 to 1).



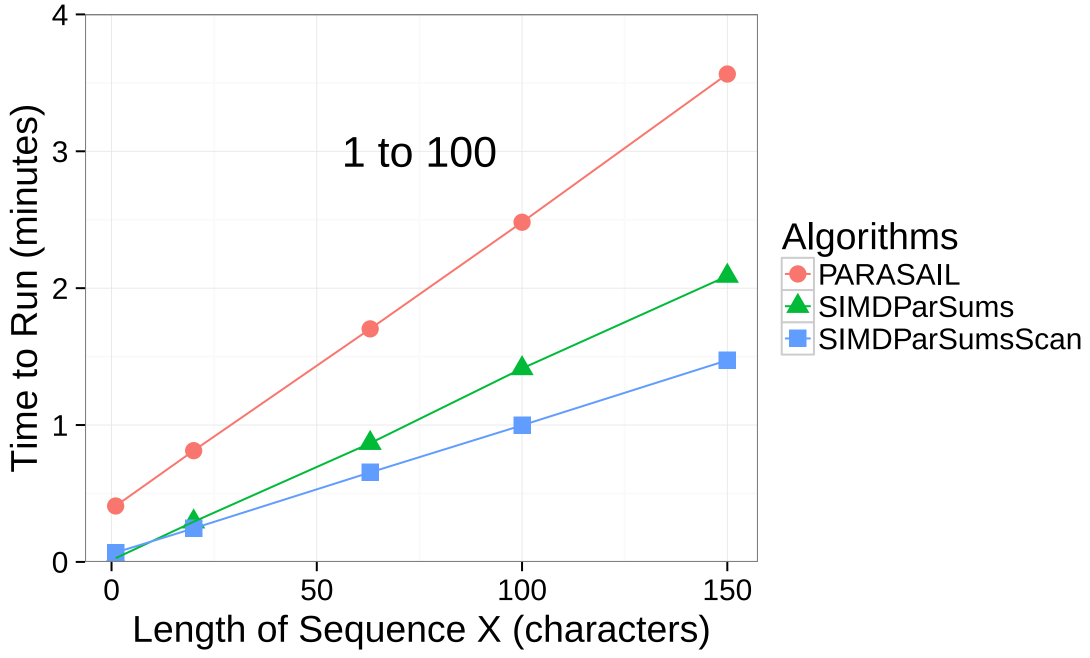


Figure 3·8: Comparison of algorithm run times for 25 million alignments. Shown are averages over three trials. $|Y| = 126$. Top: A new pair of sequences was generated for each alignment (1 to 1). Bottom: Each Y sequence was aligned against 100 newly generated X sequences (1 to 100). This models the alignment of a query sequence to a set of candidate matches and amortizes the pre-processing costs.

Chapter 4

Affine Gap and Tandem Alignment

4.1 Introduction

Our previous algorithms have all computed global alignment using a simple gap penalty. We present two extensions to this scheme. The first extension allows the computation of affine gaps and the second computes tandem alignment.

In a simple gap penalty scheme, the gap score depends only on the length of the gap. Affine gap scoring uses a gap open penalty and a gap extension penalty. This raises the cost of multiple short gaps relative to longer continuous gaps. Affine gaps reflect the biological reality that fewer, longer indels are more likely than more short indels. Protein sequence alignment typically uses an affine gap penalty scheme. Our extension computes affine gap penalties with only a few additional operations.

Repetitive sequence motifs are common in biological sequences. Thus, it is often useful to find multiple copies of a pattern in a text that possibly contains multiple copies of the pattern. Wraparound tandem alignment solves this problem efficiently by aligning a single copy of a pattern to a text [**?**]. Our method is an extension of SIMDParSum to the wraparound dynamic programming approach to tandem alignment of [19, 47].

The remainder of this chapter is organized as follows. In Section 4.2 we describe the new algorithm for alignment with affine gapping. In Section 4.3, we describe our new algorithm for tandem alignment. In Section 4.4 we give the complexity of both algorithms, and in Section 4.5 we give results of experiments comparing our algorithms with iterative dynamic programming and SIMD accelerated dynamic programming .

4.2 Affine Gap

4.2.1 Problem Definition, Affine Gap

Problem, affine gap: Given two sequences X and Y, of length m and n respectively, a similarity scoring function, S, defined by a negative, integer gap opening penalty α, a gap extension penalty β, and a table of integer substitution weights, $subst(a, b)$, defined over character pairs (a, b) from the alphabet Σ, calculate the global alignment score for X and Y using bit operations, addition, and max/min comparisons on computer words of length w.

We allow two types of initialization in the zero row and column of the alignment scoring matrix, 1) no penalty for an initial gap (zero in every cell), or 2) penalty for an initial gap ($\alpha + j \cdot \beta$ in each successive cell at j). We do not restrict the size of the alphabet, although the time complexity depends, in part, on the alphabet size as a result of required pre-processing of the $subst()$ table.

As in the SIMDParSumScan method, instead of the actual scores in S, we store only the differences between scores in adjacent cells. However, unlike previous chapters we will not exclusively use the mapped values ΔV, ΔH_{in}, and ΔH_{out} as defined in Chapter 3 Section 3.2. Instead, we will use Δv, Δh_{in}, and Δh_{out} and ΔH. That is:

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

While referring to the current row i, we drop the first index in every term. Δh terms for rows $i-1$ and i are labeled Δh_{in} and Δh_{out} respectively.

4.2.2 Definitions and notation, Affine Gap

Let S be a recursively-defined, similarity scoring function for global alignment score for two sequences $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$:

$$S[i,j] = \max \begin{cases} S[i-1,j-1] + subst(x_i, y_j) \\ F[i,j] \\ E[i,j] \end{cases}$$

$$S[0,j] = \alpha + j * \beta, \quad S[i,0] = \alpha + i * \beta$$

Note that S is now defined in terms of the additional matrices E and F, which maintain the affine gap calculations.

The affine gapping matrices E and F are defined by.

$$F[i,j] = \max \begin{cases} \alpha + \beta + S[i,j-1] \\ \beta + F[i,j-1] \end{cases}$$

$$F[i,0] = S[i,0] + \alpha$$

$$E[i,j] = \max \begin{cases} \alpha + \beta + S[i-1,j] \\ \beta + E[i-1,j] \end{cases}$$

$$E[0,j] = S[0,j] + \alpha$$

Given the above new matrices, we define several terms in addition to Δh and Δv.

$$\Delta F[j] = F[i,j] - S[i,j]$$

$$\Delta E_{in}[j] = E[i-1,j] - S[i-1,j]$$

$$\Delta E_{out}[j] = E[i,j] - S[i,j]$$

$$LB[j] = \max(score(x_i, y_j) - \Delta h_{in}[j], \beta + \max(\Delta E_{in}[j], \alpha))$$

4.2.3 Algorithm: Affine Gap

Our goal is to calculate the Δh_{out} values in row i from:

- Δh_{in} values in row $i-1$,
- $\Delta v[0]$, the leftmost Δv value in row i,
- ΔE_{in} values in row $i-1$, and
- $subst(x_i, y_j)$ substitution score values for row character x_i.

The Δh values in row zero and the Δv values in column zero depend on the type of global alignment, as mentioned above, but are known in advance, as are the ΔE_{in} values in row zero. The $subst(x_i, y_j)$ values are computed in a pre-processing step (outlined in Section 4.4) so that for any given row character x, we have the appropriate values available.

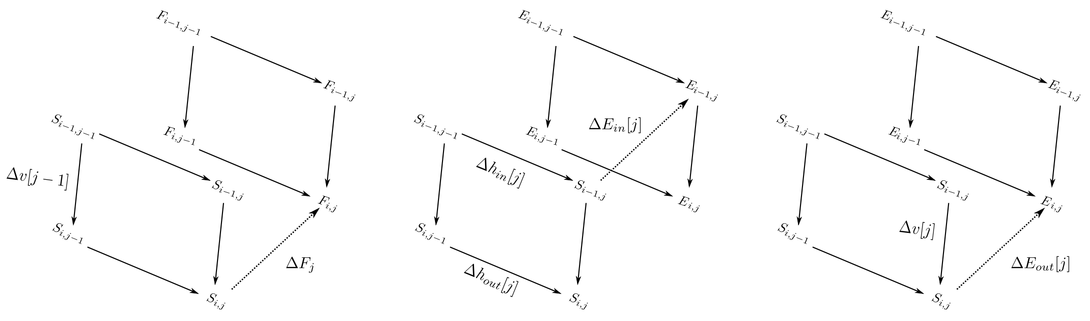


Figure 4·1: Computing ΔF, ΔE_{in} and ΔE_{out} from the S, E, and F matrices.

Method of Partial Sums, Affine Gap

In the dynamic programming algorithm for affine gap scoring, two additional matrices, E and F, must be computed. The E matrix stores the best scores possible from a continuing or newly started vertical affine gap and the F matrix stores the best scores possible from a continuing or newly started horizontal gap. At each step in the algorithm, the scores in E and F are updated and then used to compute the score in S. As shown in Figure 4·1, instead of directly using E and F, we will be considering ΔF, ΔE_{in} and ΔE_{out}.

To add support for affine gaps, we only need to add a single additional vector, ΔE that alternates between storing ΔE_{in} and ΔE_{out}. That is, ΔE_{in} at each row i is exactly ΔE_{out} of row $i-1$. In row $i-1$, ΔE_{out} can be computed from the values of Δv and ΔE_{in} in row $i-1$, as shown in the following theorem. Proofs are given in Appendix D.

Theorem 4.2.1. *Given $\Delta E_{in}[j]$ and $\Delta v[j]$, $\Delta E_{out}[j]$ can be computed by the equation*

$$\Delta E_{out}[j] = \max(\alpha, \Delta E_{in}[j]) + \beta - \Delta v[j].$$

ΔE_{in} is used to compute the LB values, as given in the definitions:

$$LB[j] = \max(subst(x_i, y_j) - \Delta h_{in}[j], \beta + \max(\Delta E_{in}[j], \alpha))$$

Due to the way that the partial sums are calculated, the value of ΔF is implicitly computed during the calculation of Δv. Theorem 4.2.2 illustrates how the value of $\Delta v[j]$ is computed taking into consideration all possible horizontal affine gaps.

Computation using SIMD words and across multiple words occurs in an analogous manner to SIMDParSumScan of Chapter 3.

Theorem 4.2.2. *$\forall j > 0$, the $\Delta v[j]$ values can be computed by the following recurrence:*

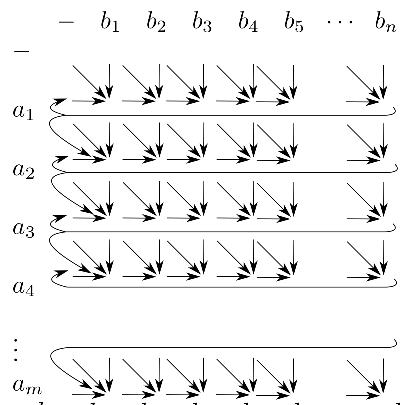
$$\Delta V[1] \quad = \quad \max\big(\Delta V[0] + \alpha, LB[1] - \beta\big) + \Delta H_{in}[1]$$

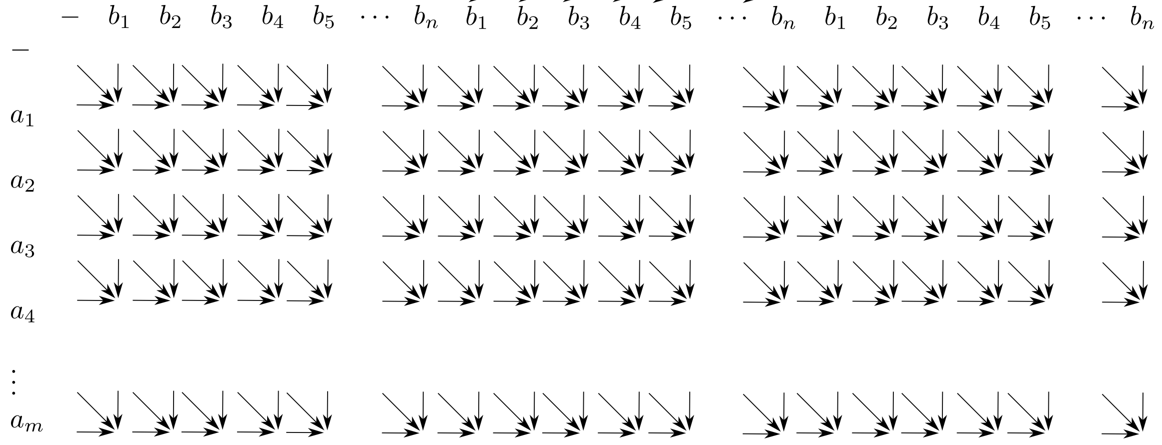
$$\underset{\forall j>1}{\Delta v}[j] \quad = \quad \max \begin{cases} LB[j] \\ LB[j-1] + \alpha + \Delta H[j] \\ LB[j-2] + \alpha + \Delta H[j-1] + \Delta H[j] \\ \vdots \\ LB[1] + \alpha + \Delta H[2] + \ldots + \Delta H[j-1] + \Delta H[j] \\ \Delta v[0] + \alpha + \Delta H[1] + \ldots + \Delta H[j-1] + \Delta H[j] \end{cases} \tag{4.1}$$

4.3 Tandem Alignment

The previously presented alignment algorithms have been designed to accommodate three common types of sequence mutations: 1) Substitutions 2) Insertions and 3) Deletions.

Tandem alignment handles a new type of mutation, tandem duplication. Tandem duplication occurs when one or more bases of DNA are duplicated in a contiguous fashion, with one or more new copies created. Tandem repeats, also known as micro- and mini-satellites are the result of tandem duplications. Tandem repeats are a





− b_1 b_2 b_3 b_4 b_5 ⋯ b_n b_1 b_2 b_3 b_4 b_5 ⋯ b_n b_1 b_2 b_3 b_4 b_5 ⋯ b_n
−
a_1
a_2
a_3
a_4
⋮
a_m

Figure 4·2: At the top, the wraparound dynamic programming scoring matrix (1 copy of pattern aligned to 1 copy of text) compared to the global alignment matrix for repeated pattern copies on the bottom.

common genomic feature, particularly in centromeres and telomeres.

Pattern copy number often varies between individuals for a given tandem repeat, leading to variable number of tandem repeats (VNTRs). VNTRs are useful in DNA fingerprinting [34] and bacterial strain identification [35, 22, 20, 44, 57]. They have also been implicated in a number of diseases including fragile-X syndrome [59], Friedreich's ataxia [11], Alzheimer's disease [52], psychiatric disorders [12, 41, 40], myotonic dystrophy [21], and Huntington's disease [26]. VNTRs are also known to have important effects on chromatin structure [55, 56, 2, 55, 61] and gene expression [60].

4.3.1 Problem Description, Tandem Alignment

Given a text sequence a and pattern sequence b, of length m and n respectively, a similarity scoring function, S, defined by a negative, integer gap weight, G, and a table of integer substitution weights, $subst(x, y)$, defined over character pairs (x, y) from the alphabet Σ calculate the global alignment score for one copy of a versus an unknown number of tandem copies of b, that is the maximum of the global alignments of a versus b, a versus bb, and so on, using bit operations, addition, and max/min comparisons on computer words of length W. This alignment score can be computed using wraparound dynamic programming with an alignment scoring matrix for one copy of a and one copy of b, with the recursion S as defined below.

We allow two types of initialization in the zero row and column of the alignment scoring matrix, 1) no penalty for an initial gap (zero in every cell), or 2) penalty for an initial gap (gap weight G added to each successive cell). We do not restrict the size of the alphabet, although the time complexity depends, in part, on the alphabet size as a result of required pre-processing of the $subst()$ table.

4.3.2 Definitions and notation, Tandem Alignment

Let S be a recursively-defined, similarity scoring function for the global wraparound alignment score of text sequence $a = a_1a_2 \dots a_m$ and pattern sequence $b = b_1b_2 \dots b_n$:

Recursion:

Initialize row zero ($1 \leq j \leq n$):

$$S[0,0] = 0$$

$$S[0,j] = S[0,0] + j \cdot G$$

Initialize column zero ($1 \leq i \leq m$):

$$S[i,0] = S[0,0] + j \cdot G$$

First pass ($i \geq 1, j \geq 1$):

$$S[i,j] = \begin{cases} \begin{cases} S[i-1,0] + subst(a_i, b_j) & \backslash\backslash\text{diagonal} \\ S[i-1,n] + subst(a_i, b_1) & \backslash\backslash\text{wraparound diagonal} \\ S[i,0] + G & \backslash\backslash\text{from left} \\ S[i-1,1] + G & \backslash\backslash\text{from above} \end{cases} & \text{if } j = 1 \\ \begin{cases} S[i-1,j-1] + subst(a_i, b_j) & \backslash\backslash\text{diagonal} \\ S[i,j-1] + G & \backslash\backslash\text{from left} \\ S[i-1,j] + G & \backslash\backslash\text{from above} \end{cases} & \text{if } j > 1 \end{cases} \tag{4.2}$$

Second pass ($i \geq 1, 1 \leq j < n$):

$$S[i,j] = \max \begin{cases} S[i,j] \\ \begin{cases} S[i,n] + G & \text{if } j = 1 \\ S[i,j-1] + G & \text{if } j > 1 \end{cases} \end{cases} \tag{4.3}$$

where, in this case, we have defined a deletion penalty for an initial gap in the alignment.

We define two sets of horizontal and vertical differences for a given row i, one based on S after the first pass and one based on S after the second pass.

For row 0, for all j, two sets of horizontal differences:

$$\Delta h_1[0,j] = \Delta h_2[0,j] = G$$

For row $i > 0$, two sets of vertical and horizontal differences:

After the first pass:

$$\Delta v_1[i,j] = S[i,j] - S[i-1,j]$$
$$\Delta h_1[i,j] = S[i,j] - S[i,j-1]$$

After the second pass:

$$\Delta v_2[i,j] = S[i,j] - S[i-1,j]$$
$$\Delta h_2[i,j] = S[i,j] - S[i,j-1]$$

To simplify algorithmic explanation, for the remainder of this chapter we map $\Delta v_1, \Delta v_2$ and $\Delta h_1, \Delta h_2$ into new variables $\Delta V_1, \Delta V_2$ and $\Delta H_1, \Delta H_2$ using the formulas:

$$\begin{aligned} \Delta V_1 &= \Delta v_1 - G, \quad & \Delta H_1 &= G - \Delta h_1 \\ \Delta V_2 &= \Delta v_2 - G, \quad & \Delta H_2 &= G - \Delta h_2 \end{aligned} \tag{4.4}$$

As with the SIMDParSum, for a fixed letter x in sequence X and any position j in sequence Y, we define

$$L[j] = subst(x, y_j) - 2G.$$

4.3.3 Algorithm, Tandem Alignment

Our goal is to calculate the ΔH_2 values in row i from:

- ΔH_2 values in row $i-1$,
- SPS_i, the sum of Δh_2 values in row $i-1$, and
- $L[j]$ values for row character x_i.

The ΔH_2 values in row zero depend on the initialization of the alignment, as mentioned above, but are known in advance. The L values are computed in a preprocessing step (outlined in Section 4.4) so that for any given row character x, we have the appropriate L values available.

Data Structures.

We use the "extended" data structure of Chapter 3 Section 3.3.1 to store the ΔH, ΔV, and L values.

Method of Tandem Alignment by Partial Sums

There are two important differences between this algorithm and the previous algorithms based on SIMDParSum. First, there are the wraparound cases: the values $\Delta V_1[i,1]$ and $\Delta V_2[i,1]$ in each row i depend on the values at position n. Due to the possibility of a wraparound, each row is computed in two passes, the first calculating ΔV_1 and the second calculating ΔV_2. Because the algorithm for calculating $\Delta V_1[i,j]$ and $\Delta V_2[i,j]$ for $j > 1$ in both passes is very similar to the SIMDParSum algorithm, we will only describe the special cases of calculating $\Delta V_1[i,1]$ and $\Delta V_2[i,1]$. Second, there is the new variable SPS_i that contains the sum of Δh_2 values. We will start by stating that SPS_i does not need to be fully recomputed for each row (which would be $O(n)$ work), it only requires a pair of operations. Additional theorems and proofs are given in Appendix E.

Lemma 4.3.1. *In row $i > 0$, given the values $\Delta V_2[i,1]$ and $\Delta V_2[i,n]$, SPS_{i+1} for row $i+1$ can be calculated as*

$$SPS_{i+1} = SPS_i + \Delta V_2[i,n] - \Delta V_2[i,1].$$

First pass:

Once we have computed SPS_i for a row, we can use it to determine the value of $\Delta V_1[i,1]$, using SPS_i to compute the diagonal wraparound.

Lemma 4.3.2. *$\Delta V_1[i,1]$ can be computed as:*

$$\Delta V_1[i,1] = \max \begin{cases} SPS_i + subst(a_i, b_1) - G & \backslash\backslash \textit{diagonal wraparound} \\ & \quad \textit{substitution} \\ subst(a_i, b_1) + \Delta H_2[i-1,1] & \backslash\backslash \textit{diagonal substitution} \\ 0 & \backslash\backslash \textit{vertical gap} \end{cases}$$

Given $\Delta V_1[i,1]$, ΔH_2 and L, the remaining values of $\Delta V_1[i,j]$ where $j > 1$ can be computed as in SIMDParSum.

Second pass:

In the second pass, we compute $\Delta V_2[i,1]$ from SPS_i, $\Delta V_1[i,1]$, and $\Delta V_1[i,n]$.

Lemma 4.3.3. *In row $i > 0$, given the values $\Delta V_1[i,1]$, $\Delta V_1[i,n]$, and SPS_i, $\Delta V_2[i,1]$ can be calculated as*

$$\Delta V_2[i,1] = \max(\Delta V_1[i,1], SPS_i + \Delta V_1[i,n] + G).$$

After the second pass, $\Delta H_2[i+1,j]$ can be computed from $\Delta V_2[i,j-1]$, $\Delta V_2[i,j]$, and $\Delta H_2[i,j]$ for all j just as ΔH_{out} was computed in SIMDParSum.

4.4 Complexity and Space

Affine:

Pre-processing involves storing the value $subst() - \beta$ for each possible row character x. We have tested two methods. In the first, the sequence Y is scanned one character, y_j, at a time and for each $x \in \Sigma$, we store $subst(x, y_j) - \beta$ in the appropriate position of the SB vector for x. The time required is $O(|\Sigma|n)$.

The second method is useful for smaller alphabets with $|\Sigma| < W$. Y is first scanned in linear time to find the columns of every character $\sigma \in \Sigma$. The column indices are stored in $|\Sigma|$ separate variables, Loc_σ. For each character $x \in \Sigma$, we determine the set of σ which share the same $subst(x, \sigma)$ value and then store that value in positions determined by performing ORs of the individual Loc_σ variables. The time required is $O(|\Sigma|^2 n/W)$.

In both cases, the space required for the $SB[j]$ values is $|\Sigma|n/W$. For the ΔH, ΔV, and ΔE vectors, n/W space is required for each.

Post-processing involves retrieving the alignment score from the final ΔH_{out} values and is the same method as described in [45]. The time required is $O(n)$.

Because calculation of ΔE requires constant time, the time complexity of the affine gap program is the same as SIMDParSumScan. Excluding the pre- and post-processing, it is

$$O\left(m\left[\frac{n}{W} + \log W\right]\right).$$

m represents the number of rows that must be calculated, n/W is the number of words that are calculated in each row as the scan time (except the end of the upsweep in the last word) is linear in the number of words, and $\log W$ is proportional to the number of operations done by the end of the upsweep in the final word.

Tandem Alignment: The preprocessing costs for the tandem alignment algorithm remain the same as in the previous algorithms.

The time complexity of the tandem alignment program is analagous to SIMD-ParSumScan. Excluding the pre- and post-processing, it is

$$O\left(m\left[\frac{n}{W}+\log W\right]\right).$$

m represents the number of rows that must be calculated, n/W is the number of words that are calculated in each row as the scan time (except the end of the upsweep in the last word) is linear in the number of words, and $\log W$ is proportional to the number of operations done by the end of the upsweep in the final word. Computing whether each wraparound occurs is done in constant time. Depending on whether or not a wraparound occurs, each row may be computed twice, but that results in a constant multiple of the number of operations and does not change the time complexity.

4.5 Experimental Results

4.5.1 Affine Gap Results

We compared the running time of our affine gap scoring algorithm against Parasail, an accelerated implementation of NW using SIMD [16, 15]. For our algorithm and Parasail we used a gap open penalty of -11 and a gap extend penalty of -1. For Parasail, we used the recommended "scan" version of the algorithm with SSE 4.1 instructions, and the 16 bit wide data-structures to avoid score overflows which occurred with the 8 bit data-structures. We used the BLOSUM 62 similarity table (for a 23 character amino acid alphabet with 15 L values, $L_{max} = 23$, $L_{min} = 8$)). The algorithms are designated: 1) SIMDAffine (Partial Sums SIMD, BLOSUM scoring with affine gaps) and 2) PARASAIL (Parasail, BLOSUM scoring).

For all experiments, we performed 25 million alignments, using randomly generated amino acid sequences. The length of sequence Y (along the top of the alignment scoring matrix which defines the number of columns) was 126. At this length, the SIMDParSumsScan and SIMDParSumsAffine algorithms use eight words. Five lengths were used for sequence X (along the left side of the alignment scoring matrix which defines the number of rows), $|X| = 1, 20, 63, 100, 150$. $|X| = 1$ was used to estimate the pre-processing overhead for each algorithm. The experiments were divided into two sets. In the first, a new pair of sequences was generated for each alignment (denoted "1 to 1"). In the second, each sequence Y was aligned, one at a time, against 100 newly generated X sequences (denoted "1 to 100"). The 1 to 100 experiment models the task of aligning a query sequence against a large number of candidate matches. It also amortizes the pre-processing cost for sequence Y over the 100 alignments, reducing the impact of pre-processing.

All programs were compiled with GCC using optimization level O3 and `march = native` (for SIMD commands) and run on an Intel Core i7-4710HQ CPU 2.50 - 3.5GHz CPU running Ubuntu Linux 14.04. Results are shown in Figure 4·3 and in Chapter 3, Table **??**. As can be seen, our new algorithms are faster than NW at all but very short sequence X lengths. In the 1 to 1 experiment, SIMDParSumsAffine is 30% faster than Parasail and 22% slower than SIMDParSumScan. In the 1 to 100 experiment, the pre-processing costs for the SIMDParSumsAffine algorithm become insignificant and it is 49% faster than Parasail and only 20% slower than SIMDParSumsScan. This is a result of the fact that more of the work of the PartialSums algorithm lies in the pre-processing step.

4.5.2 Tandem Alignment Results

We compared the running time of our tandem alignment algorithm against the WDP algorithm described in [6]. The algorithms are designated: 1) SIMDTandem (Par-

$\lvert X \rvert$	120	240	360	480	600
WDP	5.488	10.387	15.147	20.011	2.479
SIMDTandemScan	6.608	6.928	7.160	7.405	7.743

Table 4.1: Tables of run times, in minutes, for 250 thousand alignments. A different pair of sequences was used for each alignment (1 to 1).

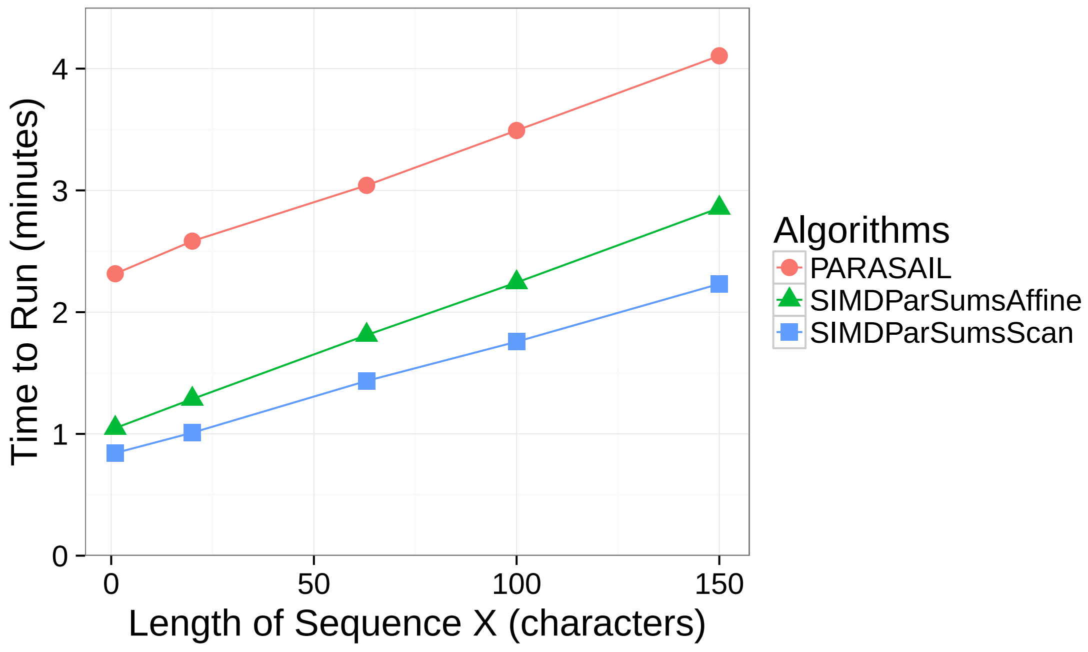
tial Sums SIMD, wraparound tandem alignment) and 2) WDP (Global wraparound dynamic programming).

For all experiments, we performed 250 thousand alignments, using randomly generated nucleic acid sequences. The length of the pattern sequence Y (along the top of the alignment scoring matrix which defines the number of columns) was 120. At this length, the PartialSums SIMD tandem alignment algorithm uses eight words. Five lengths were used for text sequence X (along the left side of the alignment scoring matrix which defines the number of rows), $|X| = 120, 240, 360, 480, 600$.

All programs were compiled with GCC using optimization level O3 and `march = native` (for SIMD commands) and run on an Intel Core i7-4710HQ CPU 2.50 - 3.5GHz CPU running Ubuntu Linux 14.04. Results are shown in Figure 4·4 and Table 4.1. As can be seen, our new algorithm is faster than WDP at all but very short sequence X lengths. SIMDtandem is up to 3 times faster than WDP.

4.6 Discussion

We have developed new algorithms for global alignment with affine gap penalties and tandem alignment. Our algorithms are faster than the standard iterative dynamic programming solution, in the case of tandem alignment, and an updated SIMD implementation of dynamic programming, in the case of global alignment. The incorporation of affine gap alignment makes our SIMD algorithm more representative of the biology of sequence mutation and more useful for protein alignment in particular. Our tandem alignment program illustrates the flexibility of our SIMD algorithm and will be useful for the tandem alignment computations done in our lab and others.



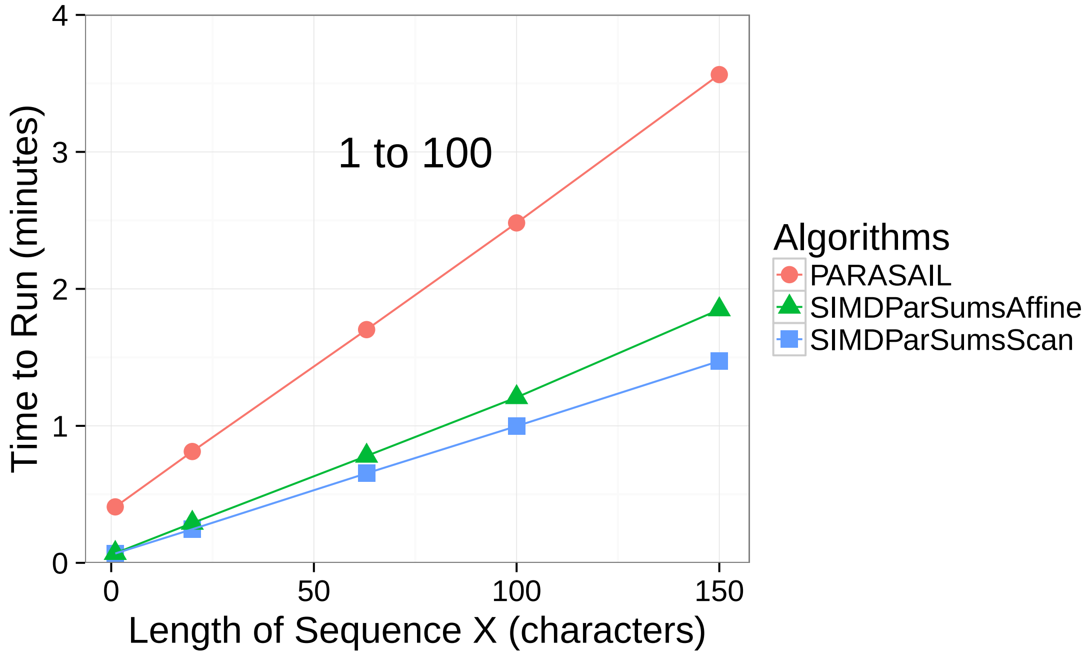


Figure 4·3: Comparison of algorithm run times for 25 million alignments. Shown are averages over three trials. $|Y| = 126$. Top: A new pair of sequences was generated for each alignment (1 to 1). Bottom: Each Y sequence was aligned against 100 newly generated X sequences (1 to 100). This models the alignment of a query sequence to a set of candidate matches and amortizes the pre-processing costs.

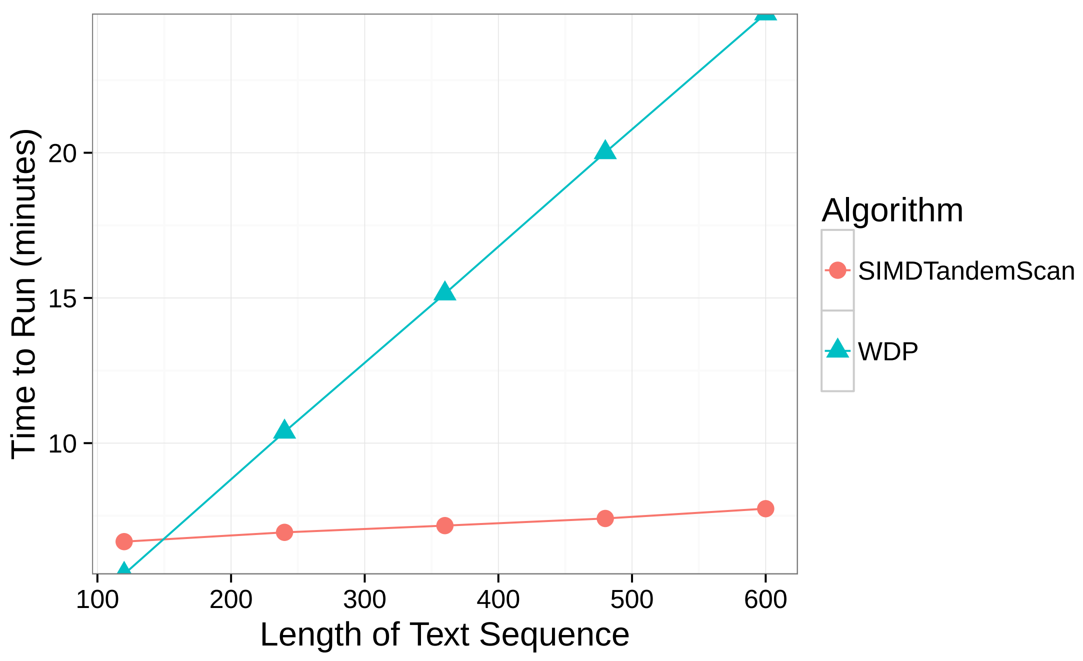


Figure 4·4: Comparison of algorithm run times for 250 thousand alignments. Shown are averages over three trials. The pattern size used was $|Y| = 120$, compared against texts multiples (1, 2, 3, 4, and 5) of the pattern long.

Chapter 5

Conclusion

5.1 Discussion

We have developed a family of bit-parallel and SIMD algorithms for global alignment. Our bit-parallel algorithms are the first bit-parallel algorithm implementations for general integer scoring global alignment, and run significantly faster than the dynamic programming approach. The BitPAl methods have already been used to accelerate software for detecting tandem repeat variants in high-throughput sequencing data [23] and are well suited to other DNA sequence comparison tasks that involve computing many alignments.

Our SIMD global alignment algorithms take a novel approach to SIMD accelerated alignment by storing the differences between scores rather than the scores themselves. The SIMD programs are simpler than the BitPAl methods and for different similarity tables only differ in the number of operations required in a preprocessing step. Our SIMDParSumsScan algorithm is currently the fastest known SIMD algorithm for global alignment with substitution scoring. We demonstrated the extensibility of the SIMDParSumsScan approach by applying it to the global alignment with affine gap penalty scoring and tandem alignment problems. The incorporation of affine gap alignment makes our SIMD algorithm more representative of the biology of sequence mutation and more useful for protein alignment in particular. Our tandem alignment program will be used to accelerate the tandem repeats finder program developed in our lab[7].

5.2 Future Work

There are further alignment problems to which we believe that our methods can be applied. Local alignment requires more information about the score than our algorithms currently store. This is a different class of problem in that 1) the score can be reset to zero in any cell and 2) the best final alignment score can occur in any cell of the alignment matrix. If all the cells have to be examined, then the time complexity shifts back to $O(nm)$. However, [31] had some success with this problem using unit cost weights and identifying *columns* in which the score of at least one cell exceeds a predefined threshold k.

Another type of alignment that could result in even faster alignment scoring is banded alignment. In banded alignment, rather than computing the entire alignment scoring matrix, only a narrow band along the diagonal is computed. Banded alignment relies on the fact that good alignments tend to lie along the diagonal of the scoring matrix. In a typical banded alignment algorithm, the band is kept centered around cells with a high score. Because our algorithm does not store the score itself, this approach may require a heuristic to estimate the location of the high score within a row.

In a similar vein to a banded alignment, k differences edit distance restricts the search space to k differences between two sequences. In the k-differences approach of [37], rather than considering rows they consider diagonals - each diagonal away from the center representing an *edit* between the two sequences. [48] extended this approach by using suffix arrays and range-minimum-queries (RMQ) for a faster algorithm. It may be possible to replace the suffix array structures and RMQ methods with a simpler bit-parallel representation of the diagonals, using a *striped* storage method to encode the interacting segments of multiple diagonals into a single SIMD vector. This could allow even greater efficiency.

Appendix A

Proofs for BitPAl

Definitions

$Max = M - G$, the largest possible value for ΔV or ΔH.

$Min = G$, the smallest possible value for ΔV or ΔH.

$Mid = I - G$, the value that marks the border between Zones A and B and Zone C.

ΔV_i and ΔH_i: bit-vectors that represent the locations of the ΔV or ΔH value $i \in [Min, Max]$.

$<< 1$: a shift of one bit toward the higher order bits in a bit-vector, with the insertion of a 0 at the lowest order bit.

$\Delta V_i^{<<}$: notation for $\Delta V_i << 1$. The shift prepares the output of one cell for input to the next.

Matches: a bit-vector representing the locations of the matches.

Block in ΔH_{min}: within ΔH_{min}, a region in which there are several contiguous bits set to the same value (either 0 or 1).

The following theorems refer to Figure A·1 which shows the relationship between values in four adjacent cells of an alignment scoring matrix.

Function Table. Theorem A.0.1 defines the function table for ΔV. The function table for ΔH is identical but transposed.

Theorem A.0.1. *Given x, y, w and z as in Figure A·1, match score $M \geq 0$, mismatch score $I < 0$ and gap (indel) score $G < 0$, ΔV input v and ΔH input h, with*

$$\Delta V \left\{ \overbrace{\begin{tikzcd} x \arrow[r, "h"] \arrow[d, "v"'] & y \arrow[d, "u"] \\ w \arrow[r] & z \end{tikzcd}}^{\Delta H} \right.$$

Figure A·1: The relationships between scores in adjacent cells in the scoring matrix: w, x, y, z are scores, h, v, u are differences: $h = y - x$, $v = w - x$, $u = z - y$.

$v, h \in \{Min, Min+1, \ldots, Max\}$, *the output* ΔV *value* u *is:*

$$u = \begin{cases} M - h, \textit{ if there is a match, for } h \in \{Min, \ldots, Max\} \\ \qquad\qquad\qquad\qquad\qquad \textit{(Match case) (1)} \\ I - h, \textit{ if } v, h \in \{Min, \ldots, Mid\} \textit{ (Zone C) (2)} \\ v - h + G, \textit{ if } v \in \{Mid+1, \ldots, Max\} \textit{ and } v > h \\ \qquad\qquad\qquad\qquad\qquad \textit{(Zones A and B) (3)} \\ G, \textit{ otherwise (Zone D) (4)} \end{cases}$$

Proof. From the similarity recurrence formula:

$$z = \max \begin{cases} x + M & \text{if match} \\ x + I & \text{if mismatch} \\ w + G & \text{horizontal gap} \\ y + G & \text{vertical gap} \end{cases}$$

Match case: Suppose that there is a match. Then

$$\begin{aligned} z &= \max(x + M, w + G, y + G) \\ &= \max(x + M, x + v + G, x + h + G) \end{aligned}$$

but, $v, h \leq M - G$. Taking the largest value creates equality in all three terms, so $z = x + M$ for all values of v, h. Substituting,

$$u = z - y = z - (x + h) = z - x - h = M - h.$$

Mismatch case (Zone C): Suppose that there is a mismatch and $z = x + I$. Then

$$x + I \geq w + G \geq x + v + G \Rightarrow Mid = I - G \geq v$$
$$x + I \geq y + G \geq x + h + G \Rightarrow Mid = I - G \geq h$$

so $h, v \in \{Min, \ldots, Mid\}$. Substituting,

$$u = z - x - h = I - h.$$

Horizontal gap (Zones A and B): Suppose z comes from a horizontal gap only. Then $z = w + G$ and

$$w + G > x + I \Rightarrow x + v + G > x + I \Rightarrow v > I - G = Mid$$
$$w + G > y + G \Rightarrow x + v + G > x + h + G \Rightarrow v > h.$$

Then $v \in \{Mid + 1, \ldots, Max\}$ and $v > h$, case (3). Substituting,

$$u = z - x - h = w + G - x - h = w - x - h + G = v - h + G.$$

Vertical gap (Zone D): Suppose z comes from a vertical gap. Then $z = y + G$ and

$$y + G \geq w + G \Rightarrow x + h \geq x + v \Rightarrow h \geq v$$
$$y + G \geq x + I \Rightarrow x + h + G \geq x + I \Rightarrow h \geq I - G = Mid.$$

Since $z = y + G$, $u = G$.

□

Output ΔV values. Theorems A.0.2–A.0.5 are used to compute the ΔV output values for the four zones of the function table. The proof for ΔH values is omitted.

Zone A.

Theorem A.0.2. *(Zone A max value.) Given the bit-vector ΔH_{min} and the bit-vector Matches, the bit-vector $\Delta V^{<<}_{max}$ ($\Delta V_{max} << 1$) can be computed using the following equation:*

$$\begin{aligned}&\Delta V_{max}^{<<} = \\ &\qquad (((\Delta H_{min} \wedge Matches) + \Delta H_{min}) \oplus \Delta H_{min}) \\ &\qquad \oplus (\Delta H_{min} \wedge Matches)\end{aligned}$$

Proof. Let *left* be the direction of the least significant bit and *right* the direction of the most significant bit. There are two ways for u to equal Max, either $h = Min$ and there is a Match or $h = Min$ and $v = Max$. Let

$$InitialV_{\max} = \Delta H_{\min} \wedge Matches.$$

Then $InitialV_{\max}$ represents all the positions where $u = Max$ because of a Match. We consider two cases: a block of consecutive 1s in $\Delta H_{\min}$ and a block of consecutive 0s.

Block of consecutive 1s: Let *Matches* contain k 1s at locations $\{p_1, p_2, \ldots, p_k\}$ within the block with p_1 the leftmost at the dth location within the block. $InitialV_{\max}$ also has 1s at these locations and nowhere else in the block. The operation $InitialV_{\max} + \Delta H_{\min}$ adds these 1s and causes a carry from p_1 to the end of the block. In the result, 1s occupy all positions left of p_1 in the block, positions $\{p_2, p_3, \ldots, p_k\}$, and the position immediately to the right of the block, if it exists. When we XOR this result with $\Delta H_{\min}$, the 1s left of p_1 are set to 0, p_1 is set to 1, the 0s between the p_is are set to 1s, and the positions $\{p_2, p_3, \ldots, p_k\}$ are set to 0. The bit to the right of the block remains 1. The final XOR of the result and $InitialV_{\max}$ sets p_1 to 0, and $\{p_2, p_3, \ldots, p_k\}$ to 1, since in $InitialV_{\max}$ those positions are all 1. The final result is a block of $n - d + 1$ 1s, starting at position $p_1 + 1$ and ending at the first position to the right of the block.

Block of consecutive 0s: Within the block, $InitialV_{\max}$ is all 0s because $\Delta H_{\min}$ was 0. Likewise $InitialV_{\max} + \Delta H_{\min}$ is all 0s, unless a carry has entered the block from the left, in which case the leftmost bit in the region is a 1. When we XOR this result with $\Delta H_{\min}$, the output is again all 0s, aside from the possible initial 1 bit. After the final XOR with $InitialV_{\max}$ the output is again all 0s, except possibly the initial bit.

Bits set to 1 now occupy all locations where $u = Max$, *i.e.*, either from $h = Min$ and there is a Match, or $h = Min$ and $v = Max$, all shifted one bit to the right.

□

Theorem A.0.3. *(Zone A Remaining Values.) Let* $u \in \{Max-1, Max-2, \ldots, Mid+1\}$ *be a* ΔV *output value in Zone A. Then the output bit-vector* $\Delta V_u^{<<}$ *can be computed by the equation*

$$\begin{aligned} &\Delta V_u^{<<} = \\ &\quad [(\Delta H_{M-u} \wedge Matches) \vee && (1)\\ &\quad (\bigvee_{\substack{k,l \mid l-k+Min=u \\ k \neq Min \\ l > Mid}} (\Delta H_k \wedge (\Delta V_l^{<<} \wedge \neg Matches)))] << 1 && (2)\\ &\quad + Remain\Delta H_{min} \oplus Remain\Delta H_{min} && (3) \end{aligned}$$

where $Remain\Delta H_{min} = \Delta H_{min} \oplus (\Delta H_{min} \wedge Matches)$.

Proof. From the function table, an output of u can be obtained in three ways: from a match, from ΔV input values $v \in \{Mid+1, Mid+1, \ldots, Max\}$, and from the propagation of u through a block of 1s in ΔH_{min}.

Formula Part 1: By Theorem A.0.1 (1), when there is a match, $u = M - h \Rightarrow h = M - u$, so the bit-vector $\Delta H_{M-u} \wedge Matches$ gives locations where the output is u due to matches.

Formula Part 2: $(\Delta V_l^{<<} \wedge \neg Matches)$ excludes locations in $\Delta V_l^{<<}$ with $Matches$, since the value of ΔV is not used if there is a match. The ΔV_l values are shifted $(\Delta V_l^{<<})$ so that they are input to the next cell. Output values of u lie on the diagonal defined by ΔH_k and ΔV_l where $l - k + Min = u$. ANDing every such pair (excluding the pair where $k = Min$ and $l = u + 1$, which will be used in the last step) and ORing the result gives a bit-vector of the locations where the u output values come from the diagonal.

Parts 1 and 2 yield the locations in $\Delta V_u^{<<}$ that must be computed before propagating through blocks of ΔH_{min}.

The $<< 1$ operation shifts these output values one bit toward the high order bit so they can act as input to the next cell.

Formula Part 3: In Part 1 of Theorem A.0.2, ΔH_{min} inputs were used to produce u output. These locations must be excluded from the propagation, since they have already produced an output. $Remain\Delta H_{\min}$ is exactly $\Delta H_{\min}$ with all such locations excluded. The operations $+\ Remain\Delta H_{\min} \oplus Remain\Delta H_{\min}$ carry out the propagation through the blocks of $\Delta H_{\min}$, as in the proof of theorem A.0.2 and results in output values shifted by one bit to the right.

□

Zones B and C.

Theorem A.0.4. *Let $u \in \{Mid, Mid-1, \ldots Min+1\}$ be a ΔV output value in Zones B or C.*

$$\begin{aligned}
\Delta V_u^{<<} = & \\
& (\Delta H_{M-u} \wedge Matches) \vee && (1)\\
& \left(\bigvee_{\substack{k,l|l-k+Min=u,\\ l>Mid}} [\Delta H_k \wedge (\Delta V_l^{<<} \wedge \neg Matches)] \right) \vee && (2)\\
& \left[\Delta H_{I-u} \wedge \left(\neg \bigvee_{l=Max}^{Mid+1} \Delta V_l^{<<} \right) \right] && (3)
\end{aligned}$$

Proof. From the function table, the ΔV output of u can be obtained in three ways: from a match, from ΔV input values $v \in \{Mid+1, Mid+2, \ldots, Max\}$ (Zone B), and from the ΔV input values $v \in \{Min, Min+1, \ldots, Mid\}$ (Zone C) .

Formula Part 1: See proof of A.0.3 Formula Part 1.

Formula Part 2: See proof of A.0.3 Formula Part 2.

Formula Part 3: The ΔV values from Min to Mid have the same outputs in the function table, given the same ΔH input value. From Theorem A.0.1 (2), since $v \in \{Min, Min+1, \ldots, Mid\}$ and $u \neq Mid = G$, then $u = I - h$ and $h = I - u$. Since Zone A has already been computed, we know the ΔV values from Max to $Mid+1$. Since the sets $\{Min, Min+1, \ldots, Mid\}$ and $\{Mid+1, Mid+2, \ldots, Max\}$ are complementary, we find the locations of the ΔV values from Min to Mid by taking the bit-wise complement of the ORed bitvectors $\Delta V_{Mid+1}, \Delta V_{Mid+2}, \ldots, \Delta V_{\max}$. ANDing them to ΔH_{I-u} gives the locations of u.

□

Zone D.

Theorem A.0.5. *(Zone D.) Suppose that the bit-vectors $\Delta V^{<<}_{max}$, $\Delta V^{<<}_{Max-1}, \ldots \Delta V^{<<}_{Min+1}$ have been computed (Zones A, B, and C). Then we can compute the bit-vector $\Delta V^{<<}_{min}$ with the equation:*

$$\Delta V^{<<}_{min} = \neg \left(\bigvee_{k=Min+1}^{max} \Delta V^{<<}_{k} \right).$$

Proof. The locations of all previously computed ΔV outputs shifted 1 bit to the right is simply $\bigvee_{k=Min+1}^{max} \Delta V^{<<}_{k}$, so the locations that have Min output shifted 1 bit to the right must be $\neg \left(\bigvee_{k=Min+1}^{max} \Delta V^{<<}_{k} \right)$. $\square$

BitPAl packed calculates the u values in Zones B and C by addition using an encoding which converts all ΔV values v into the following b values and all ΔH values h into the following c values:

$$b = v - Min$$

$$c = Min - h$$

The b values are zero or positive in the range $[0, Max - Min]$ and the c values are zero or negative in the range $[Min - Max, 0]$. The range of their sums is $[Min - Max, Max - Min]$.

Using individual bit-vectors to represent each ΔV or ΔH value results in very low information density - few bits are set compared to the overall number of bits. Instead, we use a twos complement encoding consisting of k bit vectors to store both the b and c encodings as above. The b values are stored in vectors $\Delta Vbits_{2^0}, \Delta Vbits_{2^1}, \ldots, \Delta Vbits_{2^k}$, the c values are stored in vectors $\Delta Hbits_{2^0}, \Delta Hbits_{2^1}, \ldots, \Delta Hbits_{2^k}$, and k is set to accommodate the range of sums, *i.e.*, $2^k \geq 2 * (Max - Min) + 1$ or $k = \lceil \log_2(2 * (Max - Min) + 1) \rceil$

For Zones B and C, all ΔV values $< Mid$ are treated as Mid. In this case we modify the encoding above so that

$$b = \begin{cases} Mid - Min & \text{if } v \leq Mid \\ v - Min & \text{otherwise} \end{cases}$$

The following theorem shows how to find output ΔV values using addition.

Theorem A.0.6. *(Packed Zones B and C.) Consider v, h, and u from Figure A·1 and let b and c be as in the encoding above. If $b + c > 0$, then $b + c = u - Min$, otherwise $u = Min$.*

Proof.

$\boldsymbol{b + c > 0}$:

$\boldsymbol{v > Mid}$:

$b + c = v - Min + Min - h = v - h$ and $b + c > 0 \Rightarrow v - h > 0 \Rightarrow v > h$. From Theorem A.0.1 (3), $u = v - h + G \Rightarrow u = b + c + G \Rightarrow b + c = u - G \Rightarrow b + c = u - Min$.

$\boldsymbol{v \leq Mid}$:

$b + c = Mid - Min + Min - h = Mid - h$ and $b + c > 0 \Rightarrow Mid - h > 0 \Rightarrow Mid > h$. From Theorem A.0.1 (2), $u = I - h \Rightarrow u = Mid + Min - h \Rightarrow u - Min = Mid - h \Rightarrow b + c = u - Min$.

$\boldsymbol{b + c \leq 0}$:

$\boldsymbol{v > Mid}$:

$b + c = v - Min + Min - h = v - h$ and $b + c \leq 0 \Rightarrow v - h \leq 0 \Rightarrow v \leq h$. Then by Theorem A.0.1 (4), $u = G = Min$

$\boldsymbol{v \leq Mid}$:

$b + c = Mid - Min + Min - h = Mid - h$ and $b + c \leq 0 \Rightarrow Mid - h \leq 0 \Rightarrow Mid \leq h$. Suppose that $Mid = h$. Then by Theorem A.0.1 (2), $u = I - h = I - Mid = I - (I - G) = G = Min$. Suppose that $Mid < h$. Then by Theorem A.0.1 (4), $u = G = Min$.

□

Application to Distance Based Scoring

Theorem A.0.7. *For any distance based integer scoring scheme described by alignment weights (m, i, g) with $m = 0$, $i, g > 0$, $i \leq 2g$, the global alignment bit-parallel methods described above apply to an equivalent* similarity based *integer scoring scheme, with weights (M, I, G), with $M = 0$, $I = -i$, and $G = -g$.*

Proof. Let $M = 0$, $I = -i$, and $G = -g$. Equivalence of the two scoring schemes for global alignment was established in (Smith and Waterman, 1981), Theorem 3, which states that for similarity scores M, I, G, the corresponding distance scores are $m = 0, i = M - I$, and $g = M/2 - G$. By substitution, M, I, and G will produce m, i, and g. □

The change in scoring weights is merely a remapping of the values for Max, Min, and Mid as defined above. The function table dimensions and Zones remain unchanged. Since $M \geq 0$, $k \geq 0$. Since $i, g > 0$, $i > 2k, g > k$.

Appendix B

Proofs on Substitution Scoring

Preliminaries

Theorem B.0.8. *ΔV and ΔH are integers which fall in the following ascending and descending ranges respectively:*

$$\Delta V \in \{0, 1, 2, \ldots, L_{max}\}, \qquad \Delta H \in \{0, -1, -2, \ldots, -L_{max}\}. \tag{B.1}$$

Proof. We first prove that $G \leq \Delta v \leq L_{max} + G$. Then using the transformation $\Delta V = \Delta v - G$, we obtain $0 \leq \Delta V \leq L_{max}$. The proof for Δh and ΔH is similar.

Lower bound: The recurrence for column zero is either $S[i, 0] = i * G$ (penalty for an initial gap) or $S[i, 0] = 0$ (no penalty for an initial gap). In either case, $\Delta v \geq G$. The recurrence for column $j > 0$ includes the alternative $S[i, j] = S[i - 1, j] + G$ (vertical gap). This again assures that $\Delta v \geq G$.

Upper bound: By induction. We will assume that there exists some pair x, y such that $subst(x, y) > G$. This is the typical situation because all standard substitution tables have at least one positive value. For column zero, the difference between adjacent cells is either G or 0. Since $L_{max} = \max\limits_{x,y} subst(x, y) - 2G > G - 2G > -G$ we have $L_{max} + G > 0 \geq \Delta v$.

For an arbitrary row $i > 0$ and column $j > 0$, we assume that the theorem is true for every cell above and to the left. There are three possibilities for the score $S[i, j]$, it is due to a 1) vertical gap, 2) substitution, or 3) horizontal gap. If it arises from a vertical gap, then $\Delta v = G$ which is $< L_{max} + G$. If it arises from a substitution, then $S[i, j] = S[i-1, j-1] + subst()$. Since $\Delta v = S[i, j] - S[i-1, j]$, the largest difference occurs when $S[i - 1, j] = S[i - 1, j - 1] + G$ (the minimum possible horizontal difference). Then $\Delta v = S[i - 1, j - 1] + subst() - (S[i - 1, j - 1] + G) = subst() - G \leq L_{max} + 2G - G \leq L_{max} + G$. If it arises from a horizontal gap, then $S[i, j] = S[i, j - 1] + G$. Since $\Delta v = S[i, j] - S[i - 1, j]$, the largest difference occurs when $S[i-1,j] = S[i-1,j-1] + G$ (the minimum possible horizontal difference) and $S[i,j-1] = S[i-1,j-1] + L_{max} + G$ (the maximum possible vertical difference). Then, $\Delta v \leq S[i-1,j-1] + L_{max} + G + G - (S[i-1,j-1] + G) \leq L_{max} + G$. $\square$

Theorem B.0.9. *$\forall j \geq 1$, ΔV and ΔH are computed by the following formulas:*

$$\Delta V[j] = \max\Big(0,\ \max\big(\Delta V[j-1], L[j]\big) + \Delta H_{in}[j]\Big) \tag{B.2}$$

$$\Delta H_{out}[j] = \min\Big(0,\ \min\big(-L[j], \Delta H_{in}[j]\big) + \Delta V[j-1]\Big). \tag{B.3}$$

Proof. By substitution using the recursive formula for S and the definitions of ΔV, ΔH, and $L(j)$, we get, for ΔV:

$$\underset{\forall i,j\geq 1}{\Delta V[i,j]} = \begin{cases} L[j] + \Delta H[i-1,j] & \text{Substitution, } i.e.\text{: if} \\ & L[j] \geq \begin{cases} \Delta V[i,j-1] \\ -\Delta H[i-1,j] \end{cases} \\ 0 & \text{Indel from above, } i.e.\text{: if} \\ & -\Delta H[i-1,j] \geq \begin{cases} L[j] \\ \Delta V[i,j-1] \end{cases} \\ \Delta V[i,j-1] + \Delta H[i-1,j] & \text{Indel from left, } i.e.\text{: if} \\ & \Delta V[i,j-1] \geq \begin{cases} L[j] \\ -\Delta H[i-1,j] \end{cases} \end{cases}$$

Examination of each of the cases shows that each is a maximum in it's range, yielding Equation (B.2). The case for Equation (B.3) (ΔH_{out}) is similar. $\square$

Partial Sums

Theorem B.0.10. *$\forall j > 0$, the $\Delta V[j]$ values can be computed by the following recurrence:*

$$\Delta V[1] \quad = \quad \max\Big(0, \max\big(\Delta V[0], L[1]\big) + \Delta H_{in}[1]\Big)$$

$$\underset{\forall j>1}{\Delta V[j]} \quad = \quad \max\begin{cases} 0 \\ L[j] + \Delta H_{in}[j] \\ L[j-1] + \Delta H_{in}[j-1] + \Delta H_{in}[j] \\ L[j-2] + \Delta H_{in}[j-2] + \Delta H_{in}[j-1] + \Delta H_{in}[j] \\ \vdots \\ \max\big(\Delta V[0], L[1]\big) + \Delta H_{in}[1] + \ldots + \Delta H_{in}[j-1] + \Delta H_{in}[j] \end{cases} \tag{B.4}$$

Proof. By induction on j. The base case ($j = 1$) is established by Equation (B.2). For the induction step, assume that the recurrence is true for all indices up to $j-1$, *i.e.*:

$$\Delta V[j-1] = \max\begin{cases} 0 \\ L[j-1] + \Delta H_{in}[j-1] \\ L[j-2] + \Delta H_{in}[j-2] + \Delta H_{in}[j-1] \\ \vdots \\ \max\big(\Delta V[0], L[1]\big) + \Delta H_{in}[1] + \ldots + \Delta H_{in}[j-1] \end{cases} \tag{B.5}$$

For j, we again apply Equation (B.2), which yields three alternatives, 1) a lower limit of zero, 2) the sum $L[j] + \Delta H_{in}[j]$, and 3) the sum $\Delta V[j-1] + \Delta H_{in}[j]$. The lower limit of zero is the first alternative in Equation (B.4). The sum $L[j] + \Delta H_{in}[j]$ is the second alternative in Equation (B.4). The sum $\Delta V[j-1] + \Delta H_{in}[j]$ is the maximum of the alternatives in Equation (B.5), each added to $\Delta H_{in}[j]$. These sums form the remaining alternatives in Equation (B.4). Note that we do not explicitly include the sum $0 + \Delta H_{in}[j]$ in Equation (B.4). Since all $\Delta H_{in}[j]$ are ≤ 0, this term will be zero or negative and can be discarded. □

Theorem B.0.11. *$\forall j \in \{1, \ldots, n\}, \forall i \in \{0, \ldots, \log_2 j\}$, partial sums of length 2^i,*

$$PS[j,i] = \Delta H_{in}[j - 2^i + 1] + \Delta H_{in}[j - 2^i + 2] + \ldots + \Delta H_{in}[j]$$

can be computed in $\lceil \log_2 n \rceil$ rounds in $O(n \log n)$ time.

Proof. By induction on i. We show that after round i, $PS[j,i]$ has been computed. For the base case, $i = 0$, we set $PS[j,i] = \Delta H_{in}[j]$. For the induction step, assume that after round i we have the specified partial sums. In round $i+1$, $\forall j > 2^i$ we set

$$\begin{aligned} PS[j,i+1] &= PS[j-2^i,i] + PS[j,i] \\ &= \big(\Delta H_{in}[j - 2^i - 2^i + 1] + \ldots + \Delta H_{in}[j - 2^i]\big) \\ &\qquad + \big(\Delta H_{in}[j - 2^i + 1] + \ldots + \Delta H_{in}[j]\big) \\ &= \Delta H_{in}[j - 2^{i+1} + 1] + \ldots + \Delta H_{in}[j] \end{aligned} \tag{B.6}$$

where ΔH_{in} terms with an index < 1 are omitted. All PS are computed after round $\lceil \log_2 n \rceil$. Since round $i \geq 1$ computes $n - 2^i$ additions, the number of operations per round is linear and the total time is $O(n \log n)$. □

Theorem B.0.12. *$\forall j \in \{1, \ldots, n\}$, $\Delta V[j]$ can be computed in $\lceil \log_2(n) \rceil + 1$ rounds in $O(n \log n)$ time if $\forall j \in \{1, \ldots, n\}, \forall i \in \{0, \ldots, \log_2 j\}$ the partial sums $PS[i,j]$ are available.*

In Theorem B.0.12, we show how to compute the $\Delta V[j]$ in a logarithmic number of rounds, The first round produces the best score from a vertical gap or a substitution and subsequent rounds attempt to improve the score by finding horizontal gaps starting increasing further to the left.

Proof. By induction on round i. Let $\overline{\Delta V}[j]$ be a lower bound for the final $\Delta V[j]$ value. $\overline{\Delta V}[j]$ is updated at each round and is equal to $\Delta V[j]$ after the final round. We show that after round i, for all j, $\overline{\Delta V}[j]$ is the maximum of the first $2^i + 1$ alternatives in Equation (B.4). For the base case, round 0, we compute:

$$\begin{aligned} L[1] &= \max(L[1], V[0]) \\ \underset{\forall j \geq 1}{\overline{\Delta V}}[j] &= \max(0, L[j] + \Delta H_{in}[j]). \end{aligned} \tag{B.7}$$

Equation (B.7) computes the maximum of the first two ($= 2^0 + 1$) alternatives in Equation (B.4). In round $i, 1 \leq i \leq \lceil \log_2(n) \rceil$ we compute:

$$\underset{\forall j \geq 2^{i-1}}{\overline{\Delta V}[j]} = \max \begin{cases} \overline{\Delta V}[j] \\ \overline{\Delta V}[j - 2^{i-1}] + \Delta H_{in}[j - 2^{i-1} + 1] + \ldots + \Delta H_{in}[j] \\ \qquad\qquad = \overline{\Delta V}[j - 2^{i-1}] + PS[j, i-1] \end{cases} \tag{B.8}$$

For the induction step, assume that after round i, $\overline{\Delta V}[j]$ is the maximum of the first $2^i + 1$ alternatives in Equation (B.4). That is:

$$\underset{\forall j \geq 1}{\overline{\Delta V}[j]} \quad = \quad \max \begin{cases} 0 \\ L[j] + \Delta H_{in}[j] \\ L[j-1] + \Delta H_{in}[j-1] + \Delta H_{in}[j] \\ L[j-2] + \Delta H_{in}[j-2] + \Delta H_{in}[j-1] + \Delta H_{in}[j] \\ \vdots \\ L[j - 2^i + 1] + \Delta H_{in}[j - 2^i + 1] + \ldots + \Delta H_{in}[j] \end{cases} \tag{B.9}$$

Note specifically that $\overline{\Delta V}[j - 2^i]$, <u>the lower bound 2^i positions further back from j</u>, is the maximum of its first $2^i + 1$ alternatives:

$$\underset{\forall (j-2^i) \geq 1}{\overline{\Delta V}[j - 2^i]} \quad = \quad \max \begin{cases} 0 \\ L[j - 2^i] + \Delta H_{in}[j - 2^i] \\ L[j - 2^i - 1] + \Delta H_{in}[j - 2^i - 1] + \Delta H_{in}[j - 2^i] \\ \vdots \\ L[j - 2^i - 2^i + 1] + \ldots + \Delta H_{in}[j - 2^i] \end{cases} \tag{B.10}$$

Then, in round $i+1$, Equation (B.8) looks back to $\overline{\Delta V}[j - 2^i]$ and in effect adds the sum of the 2^i terms $\Delta H_{in}[j - 2^i + 1] + \Delta H_{in}[j - 2^i + 2] + \ldots + \Delta H_{in}[j] = PS[j, i]$ to every alternative in Equation (B.10). The maximum calculation in Equation (B.8) is computed over these $2^i + 1$ sums derived from the alternatives in Equation (B.10) and the $2^i + 1$ alternatives from Equation (B.9). Because all the ΔH_{in} are ≤ 0, the first sum in the set from Equation (B.10) is ≤ 0 and can be discarded. This leaves $2^{i+1} + 1$ alternatives and these are exactly the first $2^{i+1} + 1$ alternatives in Equation (B.4).

At the end of round $i = \lceil \log_2(n) \rceil$, all $\overline{\Delta V}$ are equal to the maximum of their first $2^{\lceil \log_2(n) \rceil} + 1 \geq n + 1$ terms in Equation (B.4). Since $\Delta V[n]$ has $n+1$ terms, all $\overline{\Delta V}$ contain the final ΔV values. If the partial sums, $PS[j,i]$ for the ΔH terms are available, then each round has at most one addition and one maximum operation per index j, except round 0 which has one additional maximum operation, so the total number of operations per round is linear in n and the total time is $O(n \log n)$. $\square$

In our bit parallel approach, we store multiple ΔV values in the same word of length w. For convenience, each ΔV is allotted eight bits and there are $w/8$ values stored in each word. In the current SIMD implementation using $w = 128$ bit words, we store 16 values per word. The computation with multiple values per word requires a slight modification to the method described in Theorem B.0.12 and Theorem B.0.11. The modified method is described in Theorem B.0.13.

Theorem B.0.13. *$\forall j \in \{1, \ldots, n)$, $\Delta V[j]$ and $\Delta H_{out}[j]$ can be computed in time $O(n \log W/W))$, where W is the number of ΔV values held in a word of length w. $\forall j \in \{1, \ldots, n)$, $\Delta V[j]$ and $\Delta H_{out}[j]$ can be computed in $\lceil n/W \rceil * \left(\lceil \log_2 W \rceil + 2\right)$ rounds and $O((n * \log W)/W))$ time, where W is the number of ΔV values held in a word of length w.*

Proof. An array of words, denoted $Vval$, holds the $\Delta V[j]$ values and are numbered from 0 (containing the lowest j indices) to $\lceil n/W \rceil - 1$ (denoted $Vval_0, Vval_1$, etc.). Each word holds W values, indexed from 0 to $W-1$, and each value occupies w/W bits. Similar arrays, denoted $Hval$, $Lval$, and $PSval$, hold the ΔH, L, and partial sum values, PS, respectively. The $Vval$ words are processed in order starting with word 0. Each word is processed independently in $\lceil \log_2 W \rceil + 1$ rounds and the values are determined as described in Theorems B.0.12 and B.0.11. For the $Vval$ it suffices to show that 1) within a single word, the number of operations is linear in the *number of rounds* and 2) information can be efficiently transferred from one word to the next. Below we show how $Vval_0$ and $Vval_1$ are processed. The remaining words are processed similarly. The $\Delta H_{out}[j]$, stored in the $Hval$, are computed by a small set of instructions for each word, also shown below.

For the $\Delta V[j]$, we assume that preprocessing before computing row 1 initializes arrays $Hval$, $PSval$ and $Lval$ as follows:

```
for $k = 0$ to $n/W$
  for $h = 0$ to $W - 1$
    $Hval_k[h] = \Delta H_{in}[k * W + h + 1]$
    $Lval_k[h] = L[k * W + h + 1]$
  $PSval_k = \Delta Hval_k$
```

$Vval_0$: Round 0. This round computes the first two alternatives from Theorem B.0.10, Equation (B.4) for $j = 1 \dots W$.

$$\begin{aligned} Lval_0[0] &= \max(\Delta V[0], Lval_0[0]) \\ Sum &= Lval_0 + Hval_0 \\ Vval_0 &= \max(AllZeros, Sum) \end{aligned}$$

The re-initialization of $Lval_0[0]$ assures that if the Δv value in the first column is 0 to exclude an initial gap penalty, then the corresponding ΔV value ($\Delta V[0] = -G$) is used if it is larger than $L(1)$. The addition of the W values stored in $Lval_0$ and $Hval_0$ are done simultaneously as one operation. This is possible if the number of bits for each value is large enough to prevent overflow into the next value or, as we have done, by using an SIMD "addition with saturation" instruction which restricts each add to a single byte and, in the case of an overflow, stores the maximum (or minimum) possible value. $AllZeros$ in the max() calculation holds W values, all of which are zero. The max() calculation can be performed with a fixed number of logic instructions or with a single SIMD instruction.

$Vval_0$: Rounds $i = 1, \dots, \log W$. These rounds compute the remaining alternatives from Theorem B.0.10, Equation (B.4), following the method presented in Theorem B.0.11.

$$\begin{aligned} ShiftVval &= Vval_0 << (2^{i-1} * 8) \\ Sum &= ShiftVval + PSval_0 \\ Vval_0 &= \max(Vval_0, Sum) \\ ShiftPS &= PSval_0 << (2^{i-1} * 8) \\ PSval_0 &= PSval_0 + ShiftPS \end{aligned}$$

The first three instructions are the max() calculation of Theorem B.0.12, Equation (B.8). $ShiftVval$ is the ΔV value in the second alternative of that equation and the addition of $ShiftVval$ and $PSval_0$ is the computation of the sum in the second alternative. Note the shift is multiplied by 8 because we are using one byte to store each value. The last two instructions prepare the partial sums variable for the next round using the partial sums calculation of Theorem B.0.11, Equation (B.6).

$Vval_1$: Round 0 is identical to the one for $Vval_0$ except for the re-initialization of $Lval_1[0]$:

$$Lval_1[0] = \max(Lval_1[0], Vval_0[W-1])$$

Since the computation for $Vval_0$ is already complete, $Vval_0[W-1]$ contains its final value, $\Delta V[W]$. The equation above computes the inner max() of Theorem B.0.9, Equation (B.2), for $j = W+1$. The remaining rounds for $Vval_1$ are identical to those for $Vval_0$.

For the $\Delta H_{out}[j]$, we assume that preprocessing before computing row 1 initializes array $negLval$ which is the negative of the L values.

```
for $k = 0$ to $n/W$
  for $h = 0$ to $W - 1$
    $negLval_k[h] = L[k * W + h + 1]$
```

(B.11)

The $Hval$ variables are now used to hold the $\Delta H_{out}[j]$ (which are used subsequently as the $\Delta H_{in}[j]$ for the next row). We compute them for a generic word $Hval_k$ with the following instructions:

$$\begin{aligned} Min &= \min(Hval_k, negLval_k) \\ ShiftVval &= Vval_k << (1*8) \\ ShiftVval[0] &= Vval_{k-1}[W-1] \\ Sum &= Min + ShiftVval \\ Hval_k &= \min(AllZeros, Sum) \end{aligned} \tag{B.12}$$

The first instruction computes the inner min() of Theorem B.0.9, Equation (B.3). The second and third instructions shift the $\Delta V[j]$ values up by one in preparation for the addition, in of Equation (B.3), which is performed by the fourth instruction. Note that since we shift the $Vval_k$ up by one value, we need to insert the last value from the previous word $Vval_{k-1}$. The fifth instruction performs the outer $min()$ of Equation (B.3).

There are $\lceil n/W \rceil - 1$ words holding the $\Delta V[j]$ values and each word is processed for $\lceil \log_2 W \rceil + 1$ rounds using a fixed number of operations per round. There are $\lceil n/W \rceil - 1$ words holding the $\Delta H_{out}[j]$ values and each word is processed with a fixed number or operations. The total time is therefore $O\left(n \log W/W\right)$. $\square$

Algorithm 1 BLOSUM-type Scoring Global Alignment by Partial Sums

```
procedure PARTIAL SUMS($Hval, Lval, negLval$)
    \\ Array $Hval$ holds $\Delta H_{in}$ values
    \\ Array $Lval$ holds $L$ values
    \\ Array $negLval$ holds $-L$ values
    \\ Each array contains $\lceil n/W \rceil$ words indexed from 0 to $\lceil n/W \rceil - 1$.
    \\ Each word holds $W$ values indexed from 0 to $W - 1$
    \\ $AllZeros$ holds $W$ values, each of which is zero

    \\ Initialize each partial sum variable $PSval_k$ to hold $\Delta H_{in}$ values
    for $k = 0$ to $\lceil n/W \rceil - 1$ do
        $PSval_k = Hval_k$
    end for

    \\ Process each variable $Vval_k$ to find the final $\Delta V$ values
    for $k = 0$ to $\lceil n/W \rceil - 1$ do

        \\ Round 0 for $Vval_k$
        if ($k == 0$) then
            $Lval_0[0] = \max(Lval_0[0], \Delta V[0])$
        else
            $Lval_k[0] = \max(Lval_k[0], Vval_{k-1}[W-1])$ \\look back
        end if
        $Sum = Lval_k + Hval_k$
        $Vval_k = \max(AllZeros, Sum)$

        \\ Rounds $i = 1, \ldots, \log W$ for $Vval_k$
        for $i = 1$ to $\log W$ do
            $ShiftVval = Vval_k << (2^{i-1} * 8)$ \\shift is by bytes
            $Sum = ShiftVval + PSval_k$
            $Vval_k = \max(Vval_k, Sum)$
            $ShiftPS = PSval_k << (2^{i-1} * 8)$ \\preparation for next round
            $PSval_k = PSval_k + ShiftPS$ \\preparation for next round
        end for

    end for

    \\ Compute variables $Hval_k$, the output $\Delta H$ values
    for $k = 0$ to $\lceil n/W \rceil - 1$ do
        $Min = \min(Hval_k, negLval_k)$
        $ShiftVval = Vval_k << (1 * 8)$ \\shift is by bytes
        if ($k == 0$) then
            $ShiftVval[0] = \Delta V[0]$ \\value in column zero
        else
            $ShiftVval[0] = Vval_{k-1}[W-1]$ \\look back
        end if
        $Sum = Min + ShiftVval$
        $Hval_k = \min(AllZeros, Sum)$
    end for
end procedure
```

Appendix C

Proofs on Global Alignment with Partial Sums by Scan

C.1 Introduction

We will show that the Partial Sum computations described in Appendix B can also be done in $O(n/W + \log(W))$ time by *striping* our values across computer words [18] and performing a *scan* (parallel-prefix type sum or other prefix operation) across the words [10]. The definitions and foundational theorems on the relationships between values remain the same as in Appendix B - the change is in how the partial sums of values are computed, resulting in a reduction in the number of redundant computations.

The SIMD scan algorithm requires that data be distributed across SIMD words so that multiple parts of the scan can be done in a single SIMD operation.

Definition 2. *'Striped' data storage - for a given data set of n elements, given a SIMD word that can store W values, we will store the data in $g = \lceil n/W \rceil$ words such that the zeroth value is in the zeroth position of the zeroth word, the first value in the zeroth position of the first word, ..., the gth value is in the first position of the zeroth word, and so on such that the kth value is in the (k modulo g) word in the k/g position.*

The striped data structure was first described by [18] and was used in the SIMD alignment algorithm Parasail [16]. Our original SIMDParSum completes a scan on each of the n/W SIMD words sequentially. The sum is passed from one word to the

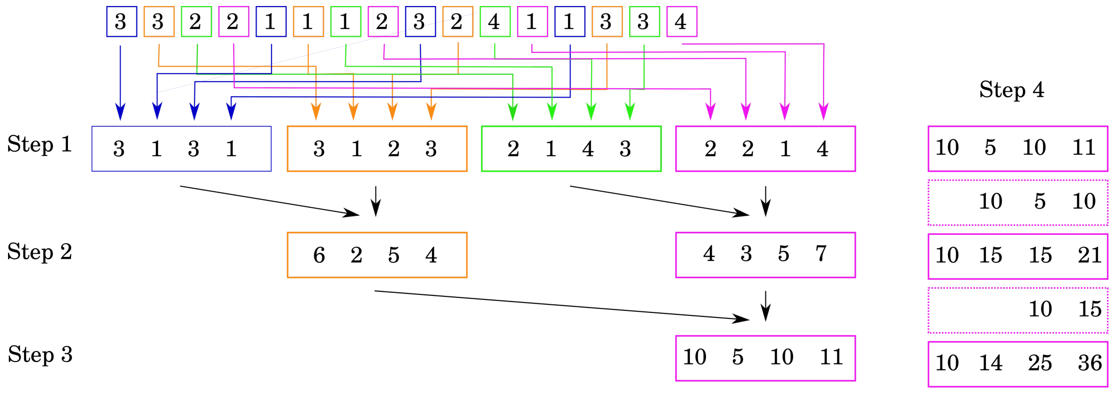


Figure C·1: Upsweep step of SIMD prefix-sum computation. In this example, each SIMD word holds 4 values. **Step 1**: the values are *striped* across the SMID words. **Step 2**: pairs of words are added. **Step 3**: pairs at the next level of the tree are added. This process continues recursively until there is only one pair, the final SIMD word and the middle word. **Step 4**: the upsweep is continued by doing a parallel scan on the final SIMD word.

next, resulting in $O(\log(W))$ work for each word and $O(\log(W) \cdot n/W)$ in total per row. There are two problems with this method. The first is that only a single SIMD word is being accessed, manipulated, and stored into at any given time. This reduces processor pipeline efficiency, because it creates blocks of sequentially dependent operations. The second is that each time a word is shifted, the addition involves fewer and fewer of the values in the word (wasted operations). By *striping* values across words, we can instead add different SIMD words to each other to have the effect of shifting without having operations wasted due to uninvolved values.

Theorem C.1.1. *$\forall j \in \{1, \ldots, n\}$, $\Delta V[j]$ can be computed in $O\left(\frac{n}{W} + \log W\right)$ time given $\Delta H_{in}[j]$ and $L[j]$ stored in a* striped *manner in $\frac{n}{W}$ words.*

Proof. For this proof, we will suppose that n/W, the number of words is a power of 2. It is trivial to extend the proof to situations where this is not true.

As in SIMDParSum, for a value $j \in \{1, 2, \ldots, n\}$ we will calculate $\Delta V[j]$ by the recursion given in Appendix A:

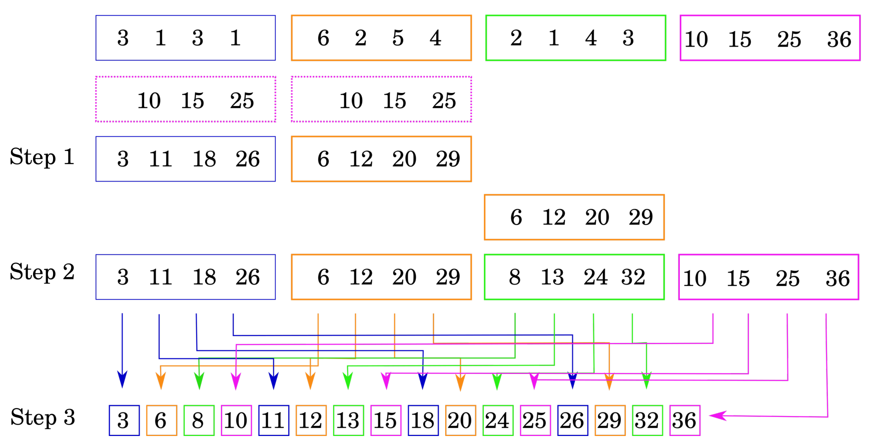


Figure C·2: Downsweep step of SIMD prefix-sum computation. In this example, each SIMD word holds 4 values. The upsweep step has already been completed, the final word holds prefix-sums from the beginning for the 4 positions stored there. **Step 1**: the final word is shifted, and added to the first and second words. **Step 2**: The second word is added to the third. **Step 3**: This shows how values can be reordered (unstriped) into their original positions, but this is not actually done in each row.

$$\Delta V[1] = \max\Big(0, \max\big(\Delta V[0], L[1]\big) + \Delta H_{in}[1]\Big)$$

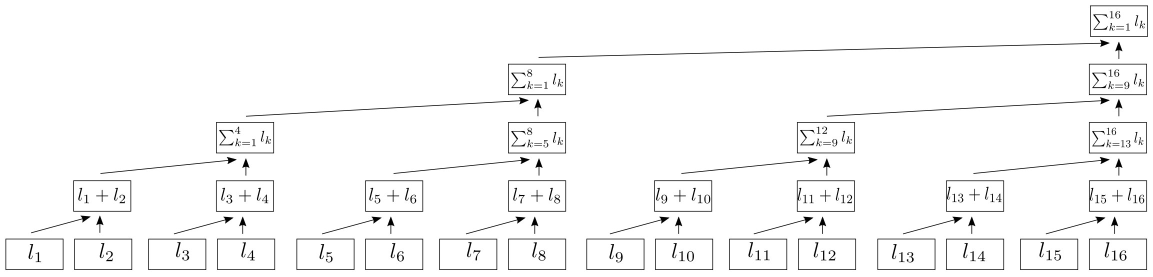
$$\underset{\forall j>1}{\Delta V}[j] = \max\begin{cases} 0 \\ L[j] + \Delta H_{in}[j] \\ L[j-1] + \Delta H_{in}[j-1] + \Delta H_{in}[j] \\ L[j-2] + \Delta H_{in}[j-2] + \Delta H_{in}[j-1] \\ \qquad\qquad\qquad\qquad + \Delta H_{in}[j] \\ \vdots \\ \max\big(\Delta V[0], L[1]\big) + \Delta H_{in}[1] + \dots \\ \qquad\qquad\qquad\qquad + \Delta H_{in}[j-1] + \Delta H_{in}[j] \end{cases} \tag{C.1}$$

Note that the major work involved in this recurrence is the sum of ΔH_{in} values. We will follow the outline of the efficient prefix-sum algorithm given by Blelloch [10], and apply the method outlined there for computing the above recurrence. Blelloch's algorithm consists of two parts: an upsweep (or reduce) and a downsweep.

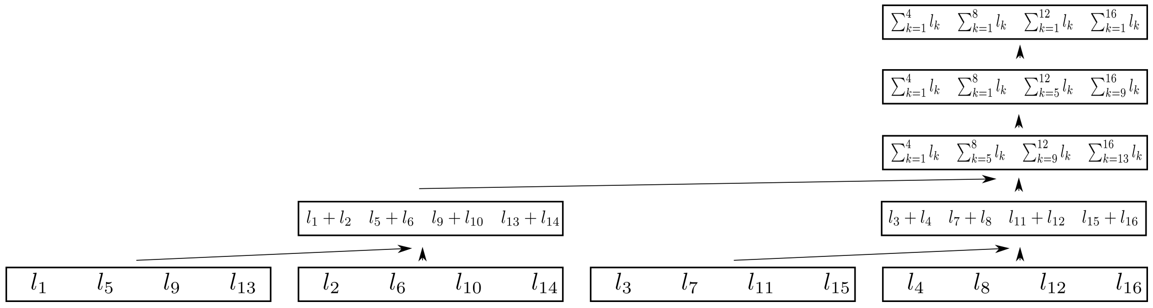
Blelloch's upsweep is done in $O(n/p + log(p))$ time where n is the number of values, p is the number of processors. Rather than parallelizing across processors, we are parallelizing across SIMD vectors. By *striping* the data, each sum step we do on the upsweep will apply to all W values in an SIMD word, resulting in $O(n/W)$ work to get the upsweep into the final word. At that point, the upsweep continues in the final word via shifts and adds within the word, as in the scan of our SIMDParSum completing the upsweep in time $log(W)$. At each step in our scan, we are computing both the sum of ΔH_{in} values as well as the maximum of $L + \sum \Delta H_{in}$.

The upsweep proceeds as follows, using the sum of ΔH_{in} to illustrate:

We store ΔH_{in} in *striped* format in an array we call ΔH_S with n/W striped words,



(a) Blelloch's Upsweep


$\sum_{k=1}^{4} l_k$ $\sum_{k=1}^{8} l_k$ $\sum_{k=1}^{12} l_k$ $\sum_{k=1}^{16} l_k$
$\sum_{k=1}^{4} l_k$ $\sum_{k=1}^{8} l_k$ $\sum_{k=5}^{12} l_k$ $\sum_{k=9}^{16} l_k$
$\sum_{k=1}^{4} l_k$ $\sum_{k=5}^{8} l_k$ $\sum_{k=9}^{12} l_k$ $\sum_{k=13}^{16} l_k$
$l_1 + l_2$ $l_5 + l_6$ $l_9 + l_{10}$ $l_{13} + l_{14}$
$l_3 + l_4$ $l_7 + l_8$ $l_{11} + l_{12}$ $l_{15} + l_{16}$
l_1 l_5 l_9 l_{13}
l_2 l_6 l_{10} l_{14}
l_3 l_7 l_{11} l_{15}
l_4 l_8 l_{12} l_{16}

(b) *Striped* SIMD upsweep

Figure C·3: Blelloch's upsweep proceeds by summing pairs of values in a conceptual binary tree. This binary tree with n leaves has $n - 1$ internal nodes, leading to $n - 1$ operations. The *striped* SIMD scan arranges the pairs of values so that multiple value pairs within pairs of SIMD words can be summed at the same time. In the final word, values are shifted and added to complete the scan. Given W elements per SIMD word, there are $n/W - 1$ addition operations before the scan in the final word and $2 \cdot log(W)$ operations (1 shift, 1 addition per log step in the final word scan. The larger n and W are, the larger the advantage of the *striped* SIMD scan.

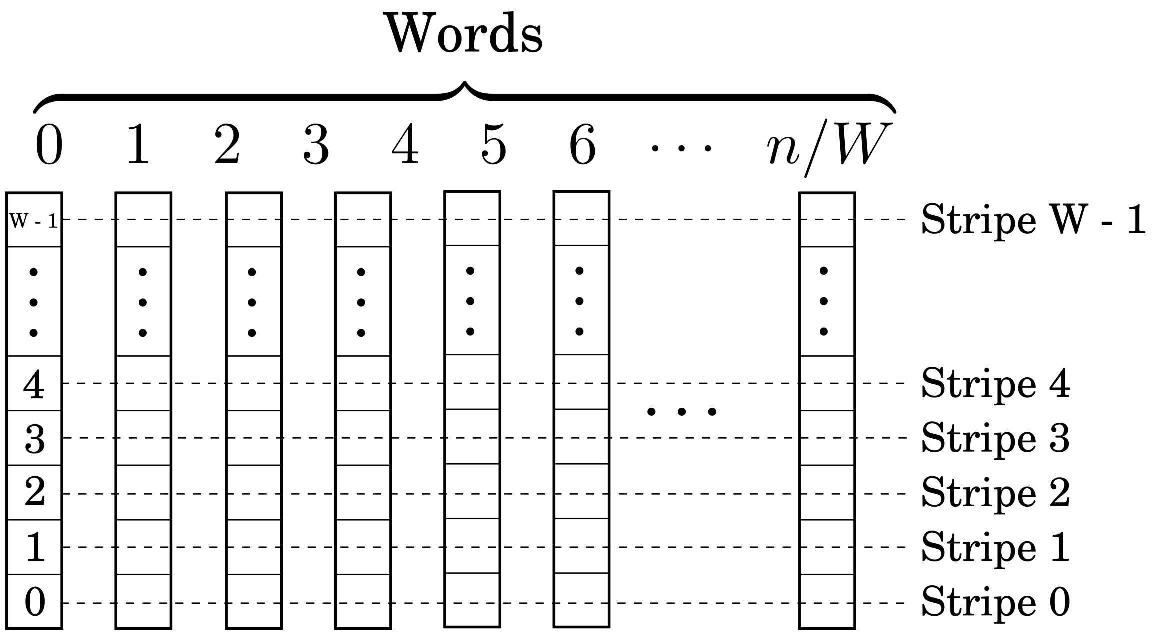


Figure C·4: Diagram of stripe format. Each word holds W values. To store n values, we use n/W words. Each *stripe* i consists of all values from words 0 to n/W at position i in each respective word.

as in Figure C·4. We indicate the zeroth value in the zeroth word by $\Delta H_S[0][0]$. Thus,

$$\begin{aligned}\Delta H_S[0][0] &= \Delta H_{in}[1]\\ \Delta H_S[1][0] &= \Delta H_{in}[2]\\ &\vdots\\ \Delta H_S[n/W-1][0] &= \Delta H_{in}[n/W]\\ \Delta H_S[0][1] &= \Delta H_{in}[n/W+1]\\ &\vdots \quad .\end{aligned}$$

There are $\log(n/W)$ steps, one for each level of the tree.

At step k, pairs of words are summed (with indices i and $i-2^{k-1}$ for i modulo $2^k = 2^k - 1$) and the result stored in the higher indexed word of the pair, *i.e.*,

$$\Delta H_S[i] = SIMDAdd(\Delta H_S[i-2^{k-1}], \Delta H_S[i]).$$

The words at i and $i-2^{k-1}$ were previously stored into at round $k-1$, so step k

produces sums of 2^k values. As a result, each word with index i where $(i+1)$ modulo $2^l = 0$ for some l contains sums of 2^l values after step l.

At step 1, each pair of words, i and $i - 2^0 = i - 1$, where i modulo $2 = 2 - 1 = 1$, is summed and stored in the higher index of the pair, *i.e.*,

$$\begin{aligned}
\Delta H_S[1] &= SIMDAdd(\Delta H_S[0], \Delta H_S[1]) \\
\Delta H_S[3] &= SIMDAdd(\Delta H_S[2], \Delta H_S[3]) \\
\Delta H_S[5] &= SIMDAdd(\Delta H_S[4], \Delta H_S[5]) \\
&\vdots
\end{aligned} \tag{C.2}$$

Figure C·5 shows the results of this set of sums, note that $\Delta H_S[0][0]$ is unchanged.

At step 2, each pair of words, i and $i - 2^1$, where i modulo $2^2 = 2^2 - 1 = 3$, is summed, *i.e.*,

$$\begin{aligned}
\Delta H_S[3] &= SIMDAdd(\Delta H_S[1], \Delta H_S[3]) \\
\Delta H_S[7] &= SIMDAdd(\Delta H_S[5], \Delta H_S[7]) \\
&\vdots \\
\Delta H_S[i] &= SIMDAdd(\Delta H_S[i], \Delta H_S[i-2])
\end{aligned}$$

Note that the sum in step 2 makes use of results from step 1, so that the sums produced by step 2 are sums of four ΔH_{in} values. For example,

$$\Delta H_S[3][0] = \sum_{i=0}^{3} \Delta H_{in}[i].$$

Continuing through $\log(n/W)$ steps, as in Figure C·6, the resulting final word has sums of $2^{\log(n/W)} = n/W$ values, meaning that the final word contains, in each position i, the sum of all of the values in stripe i from every word. Let $f = n/W - 1$ be the index of the final word. Then

$$\begin{aligned}
\Delta H_S[f][0] &= \sum_{i=1}^{n/W} \Delta H_{in}[i] \\
\Delta H_S[f][1] &= \sum_{i=n/W+1}^{2n/W} \Delta H_{in}[i] \\
&\vdots \\
\Delta H_S[f][k] &= \sum_{i=(k-1)n/W+1}^{kn/W} \Delta H_{in}[i] \\
&.
\end{aligned}$$

In order to have the sums in the final word start with the zeroth value rather than the beginning of the particular stripe, we shift and sum in the final word. A shift and add is done for each of $\log W$ rounds, *i.e.*, in round $i \in \{1, 2, ..., \log(W)\}$,

$$\Delta H_S[f] = SIMDAdd(\Delta H_S[f], \Delta H_S[f] << 2^i).$$

At step 1,

$$\Delta H_S[f] = SIMDAdd(\Delta H_S[f], \Delta H_S[f] << 1),$$

which is equivalent to

$$\begin{aligned} \Delta H_S[f][0] &= \Delta H_S[f][0] \\ \Delta H_S[f][1] &= \Delta H_S[f][0] + \Delta H_S[f][1] \\ &\vdots \\ \Delta H_S[f][W-1] &= \Delta H_S[f][W-2] + \Delta H_S[f][W-1]. \end{aligned}$$

From above, before the shift and add $\Delta H_S[f][0] = \sum_{i=1}^{n/W} \Delta H_{in}[i]$ and $\Delta H_S[f][1] = \sum_{i=n/W+1}^{2n/W} \Delta H_{in}[i]$, so

$$\begin{aligned} \Delta H_S[f][1] &= \sum_{i=1}^{n/W} \Delta H_{in}[i] + \sum_{i=n/W+1}^{2n/W} \Delta H_{in}[i] \\ &= \sum_{i=1}^{2n/W} \Delta H_{in}[i] \end{aligned}$$

Likewise,

$$\begin{aligned} \Delta H_S[f][2] &= \sum_{i=n/W+1}^{2n/W} \Delta H_{in}[i] + \sum_{i=2n/W+1}^{3n/W} \Delta H_{in}[i] \\ &= \sum_{i=n/W+1}^{3n/W} \Delta H_{in}[i] \end{aligned}$$

Progressing through the $\log(W)$ shifts and adds, the final values in $\Delta H_S[f]$ are

$$\Delta H_S[f][0] = \sum_{i=1}^{n/W} \Delta H_{in}[i]$$
$$\Delta H_S[f][1] = \sum_{i=1}^{2n/W} \Delta H_{in}[i]$$
$$\vdots$$
$$\Delta H_S[f][k] = \sum_{i=1} kn/W \Delta H_{in}[i].$$

Because our upsweep in the final word computes sums from the beginning, some of the downsweep work has already been done. Traversing the tree to spread the values in the downsweep requires only a single set of operations at each node, resulting in $O(n/W)$ work.

First, the values in the final word are shifted and saved in a temporary variable SLW. SLW has in its first position 0, in its second position the sum of all the ΔH_{in} values in Stripe 0, in its third position the sum of all the ΔH_{in} values in Stripe 0 and Stripe 1, and so on. This is shown in Figure C·7. Recall that in the upsweep, every word with index $k < n/W - 1$ such that $k + 1 = 2^l$ for some l, has values that are sums to the beginning of the stripe. Each word with such an index k has SLW added to it: $\Delta H_S[k] = SIMDAdd(\Delta H_S[k], SLW)$ - this extends the sum from the beginning of the stripe to the beginning of the values.

The downsweep then proceeds in steps from $\log(n/W) - 2$ to 1. In our example, shown in Figure C·8, the steps are $\log(16) - 2 = 2$, 1, and 0. The goal is to propagate sums from words with indices $k < n/W - 1$ where $k + 1 = 2^l$ from left to right. These are the words that just had SLW added to them.

In our initial step $j = \log(n/W) - 2$, we establish a set of words with indices $k < n/W - 1$ such that $k + 1 = 2^l$. We will term this set I. Each word with index $k \in I$ where $k \geq 2^j$ is added to words with indices $k + 2^j$. Recall that at index k, the sum in each position x run from $\Delta H_{in}[1]$ to $\Delta H_{in}[k + x * (n/W)]$. At index $k + 2^j$, sums are from $\Delta H_{in}[k + x * (n/W) + 1]$ to $\Delta H_{in}[k + 2^j + x * (n/W)]$, so if we add $\Delta H_S[k]$ to $\Delta H_S[k + 2^j]$, the sum at $k + 2^j$ will now be from $\Delta H_{in}[1]$ to $\Delta H_{in}[k + 2^j + x * (n/W)]$. Note that these values are the sums from the beginning.

After performing the addition, we will add the indices $k + 2^j$ for $k|k \geq 2^j, k \in I$ to the set I and set $j = j - 1$.

For each consecutive step j, we recursively repeat this process, with I growing to include by Step 0 all indices with odd values. In our final step, Step 0, we are adding pairs of values, and all words have their final values. Because we are adding into each word one time, this process is $O(n/W)$ work.

Thus, total time for the algorithm is $O(n/W + \log(W) + n/W)$

$\square$

$$
\begin{array}{rlcr}
\Delta H_S[0][0] = \Delta H_{in}[1] & , \quad \Delta H_S[0][1] = \Delta H_{in}[n/W+1] & & , \quad \Delta H_S[0][W-1] = \Delta H_{in}[n/W+1] \\
\Delta H_S[1][0] = \Delta H_{in}[1] + \Delta H_{in}[2] & , \quad \Delta H_S[1][1] = \Delta H_{in}[n/W+1]+ & & , \quad \Delta H_S[1][W-1] = \Delta H_{in}[n/W+1]+ \\
 & \qquad\qquad \Delta H_{in}[n/W+2] & \ldots & \Delta H_{in}[n/W+2] \\
\Delta H_S[2][0] = \Delta H_{in}[3] & , \quad \Delta H_S[2][1] = \Delta H_{in}[n/W+3] & \ldots & , \quad \Delta H_S[2][W-1] = \Delta H_{in}[n/W+3] \\
\Delta H_S[3][0] = \Delta H_{in}[3] + \Delta H_{in}[4] & , \quad \Delta H_S[3][1] = \Delta H_{in}[n/W+3]+ & \ldots & , \quad \Delta H_S[3][1] = \Delta H_{in}[n/W+3]+ \\
 & \qquad\qquad \Delta H_{in}[n/W+4] & \ldots & \Delta H_{in}[n/W+4] \\
\vdots & \qquad \vdots & & \vdots \\
\Delta H_S[n/W-2][0] = \Delta H_{in}[n/W-1] & \Delta H_S[n/W-2][1] = \Delta H_{in}[2n/W-1] & & \Delta H_S[n/W-2][W-1] = \Delta H_{in}[2n/W-1] \\
\Delta H_S[n/W-1][0] = \Delta H_{in}[n/W-1]+ & \Delta H_S[n/W-1][1] = \Delta H_{in}[2n/W-1]+ & & \Delta H_S[n/W-1][W-1] = \Delta H_{in}[2n/W-1]+ \\
\Delta H_{in}[n/W] & \qquad\qquad \Delta H_{in}[2n/W] & & \Delta H_{in}[2n/W]
\end{array}
$$

Figure C·5: Sums stored in ΔH_S after first step.

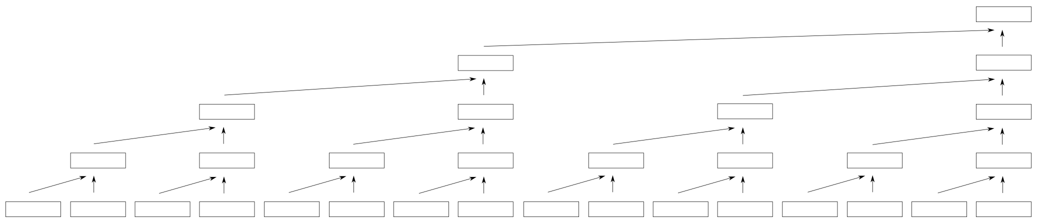


Figure C·6: $\log(n/W)$ steps to upsweep to the final word. Pairs of words at each level are summed and stored in the "right" side of the pair.

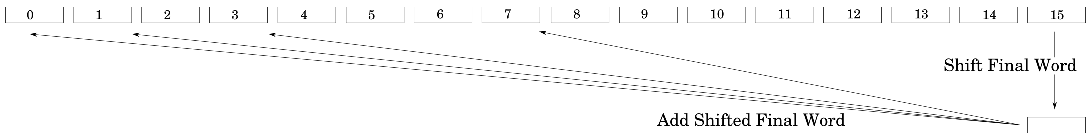


Figure C·7: The shifted last word SLW is added to words with indices that are $2^l - 1 < f$, in this case $0, 1, 3$, and 7.

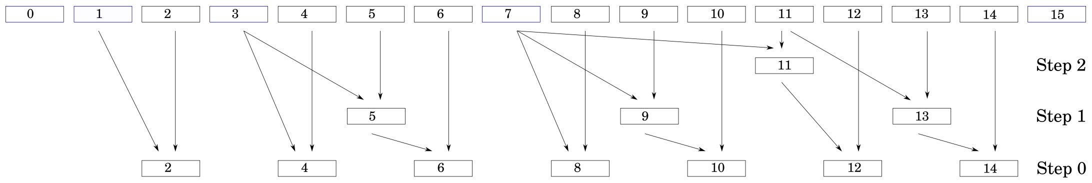


Figure C·8: Example of the downsweep after sums of SLW. Words are summed from left to right in steps from $\log(n/W) - 2$ to 1, in this case $\log(16) - 2 = 2$ to 1.**Step 2:** We start with indices $1, 3$, and 7. Words at these indices have their final values from the addition of SLW. At step 2, we will be adding words with indices $2^2 = 4$ apart. $1, 3 < 4$, so only word 7 is used and added to word 11. **Step 1:** At step one, we now consider indices $1, 3, 7$, and 11. We will add $2^1 = 2$ to each index ≥ 2. Words $3, 7$, and 11 are added to words $5, 9$, and 13. **Step 0:** We will add $2^0 = 1$ to each index ≥ 1. Words $1, 3, 5, 7, 9, 11$ and 13 are added to words $2, 4, 6, 8, 10, 12$ and 14. Every word now has its final value.

Appendix D

Proofs on Global Alignment with Affine Gap Scoring

D.1 Definitions

Consider the dynamic programming algorithm for global alignment with affine gap scoring. The scoring matrix S is defined by

$$S_{i,j} = \max \begin{cases} F_{i,j} \\ score(x_i, y_j) + S_{i-1,j-1} \\ E_{i,j} \end{cases}$$

$$S_{0,j} = \alpha + j \cdot \beta, j \in [0, \dots, n]$$

$$S_{i,0} = \alpha + i \cdot \beta, i \in [0, \dots, m]$$

The affine gapping matrices E and F are defined by

$$F_{i,j} = \max \begin{cases} \alpha + \beta + S_{i,j-1} \\ \beta + F_{i,j-1} \end{cases}$$

$$F_{i,0} = S_{i,0} + \alpha$$

$$E_{i,j} = \max \begin{cases} \alpha + \beta + S_{i-1,j} \\ \beta + E_{i-1,j} \end{cases}$$

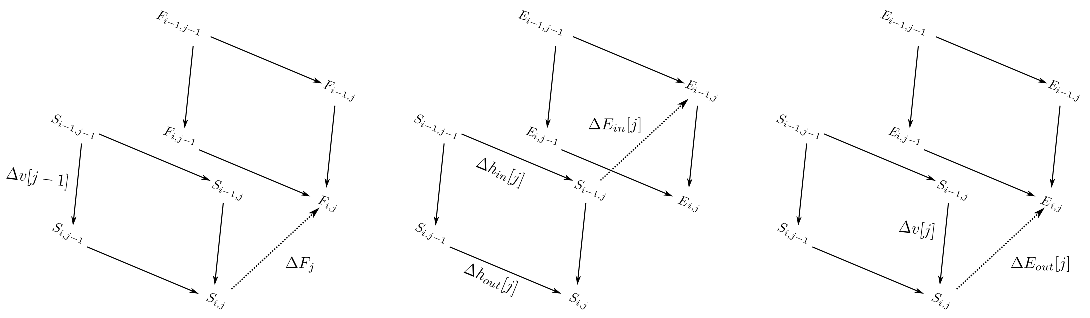


Figure D·1: Computing ΔF, ΔE_{in} and ΔE_{out} from the S, E, and F matrices.

$$E_{0,j} = S_{0,j} + \alpha$$

where E represents a vertical gap and F represents a horizontal gap. Given the recurrences above, we define several terms.

$$\begin{aligned}
\Delta v[j] =& S_{i,j} - S_{i-1,j} \\
\Delta h_{in}[j] =& S_{i-1,j} - S_{i-1,j-1} \\
\Delta h_{out}[j] =& S_{i,j} - S_{i,j-1} \\
\Delta H[j] =& \beta - \Delta h_{in}[j] \\
\Delta H_{out}[j] =& \beta - \Delta h_{out}[j] \\
\Delta F[j] =& F_{i,j} - S_{i,j} \\
\Delta E_{in}[j] =& E_{i-1,j} - S_{i-1,j} \\
\Delta E_{out}[j] =& E_{i,j} - S_{i,j}
\end{aligned}$$

D.2 Theorems and Proofs

Theorem D.2.1. *Suppose that for a pair i, j we have $\Delta v[j-1]$, $\Delta h_{in}[j]$, $\Delta F[j-1]$, and $LB[j]$. Then we can compute $\Delta v[j]$ with the equation*

$$\Delta v[j] = \max \begin{cases} \Delta v[j-1] + \max(\Delta F[j-1], \alpha) + \Delta H[j] \\ score(x_i, y_j) - \Delta h_{in}[j] \\ \max(\Delta E_{in}[j], \alpha) + \beta. \end{cases}$$

Proof. From our definition, we know that $\Delta v[j] = S_{i,j} - S_{i-1,j}$.

From the definition of S,

$$S_{i,j} = \max \begin{cases} F_{i,j} \\ score(x_i, y_j) + S_{i-1,j-1} \\ E_{i,j} \end{cases}$$

.

Thus,

$$\Delta v[j] = S_{i,j} - S_{i-1,j} = \max \begin{cases} F_{i,j} - S_{i-1,j} & \text{(D.1)} \\ score(x_i, y_j) + S_{i-1,j-1} - S_{i-1,j} & \text{(D.2)} \\ E_{i,j} - S_{i-1,j} & \text{(D.3)} \end{cases}$$

We will consider these three cases separately.

Case (1): From the definition of $F_{i,j}$,

$$\begin{aligned}
F_{i,j} - S_{i-1,j} &= \max(\alpha + \beta + S_{i,j-1}, \beta + F_{i,j-1}) - S_{i-1,j} \\
&= \max(\alpha + \beta + S_{i,j-1} - S_{i-1,j}, \beta + F_{i,j-1} - S_{i-1,j}) \\
&= \max(\alpha + \beta + S_{i,j-1} \underline{- S_{i-1,j-1} + S_{i-1,j-1}} - S_{i-1,j}, \\
&\qquad\qquad \beta + F_{i,j-1} - S_{i-1,j}) \\
&= \max(\alpha + \beta - \Delta h_{in}[j] + \Delta v[j-1], \\
&\qquad \beta + F_{i,j-1} \underline{- S_{i,j-1} + S_{i,j-1}} - S_{i-1,j}) \\
&= \max(\alpha + \beta - \Delta h_{in}[j] + \Delta v[j-1], \\
&\qquad\qquad \beta + \Delta F[j-1] + S_{i,j-1} - S_{i-1,j}) \\
&= \max(\alpha + \beta - \Delta h_{in}[j] + \Delta v[j-1], \\
&\qquad \beta + \Delta F[j-1] - \Delta h_{in}[j] + \Delta v[j-1]) \\
&= \max(\Delta F[j-1], \alpha) + \Delta v[j-1] - \Delta h_{in}[j] + \beta \\
&= \max(\Delta F[j-1], \alpha) + \Delta v[j-1] + \Delta H[j]
\end{aligned}$$

where the underlined terms show the addition of zero.
Case (2): By definition, $\Delta h_{in}[j] = S_{i-1,j} - S_{i-1,j-1}$, so

$$\begin{aligned}
&score(x_i, y_j) + S_{i-1,j-1} - S_{i-1,j} \\
&\qquad = score(x_i, y_j) - \Delta h_{in}[j].
\end{aligned}$$

Case (3): From the definition of $E_{i,j}$,

$$\begin{aligned}
E_{i,j} - S_{i-1,j} &= \max(\alpha + \beta + S_{i-1,j}, \beta + E_{i-1,j}) - S_{i-1,j} \\
&= \max(\alpha + \beta + S_{i-1,j} - S_{i-1,j}, \beta + E_{i-1,j} - S_{i-1,j}) \\
&= \max(\alpha + \beta, \Delta E_{in}[j] + \beta) \\
&= \max(\Delta E_{in}[j], \alpha) + \beta
\end{aligned}$$

Recombining, we get

$$\Delta v[j] = \max \begin{cases} \Delta v[j-1] + \max(\Delta F[j-1], \alpha) + \Delta H[j] \\ score(x_i, y_j) - \Delta h_{in}[j] \\ \max(\Delta E_{in}[j], \alpha) + \beta. \end{cases}$$

□

For the discussion that follows, we define a new term $LB[j]$:

$$\forall j > 0, LB[j] = \max(score(x_i, y_j) - \Delta h_{in}[j], \beta + \max(\Delta E_{in}[j], \alpha)).$$

Note that $LB[j]$ is equal to the maximum of the second and third terms in the formula for Theorem D.2.1, *i.e.*, the maximum value for $\Delta v[j]$ from a substitution or vertical gap score, the two possible values originating from the row above. As such, $LB[j]$ is a lower bound to $\Delta v[j]$ and Theorem D.2.1 can be rewritten as:

$$\Delta v[j] = \max \begin{cases} \Delta v[j-1] + \max(\Delta F[j-1], \alpha) + \Delta H[j] \\ LB[j]. \end{cases} \tag{D.4}$$

Theorem D.2.2. *Given $\Delta E_{in}[j]$ and $\Delta v[j]$, $\Delta E_{out}[j]$ can be computed by the equation*

$$\Delta E_{out}[j] = \max(\Delta E_{in}[j], \alpha) + \beta - \Delta v[j]$$

.

Proof. From the definitions:

$$\begin{aligned}
\Delta E_{out}[j] &= E_{i,j} - S_{i,j} \\
&= \max(\alpha + \beta + S_{i-1,j}, \beta + E_{i-1,j}) - S_{i,j} \\
&= \max(\alpha + \beta + S_{i-1,j} - S_{i,j}, \beta + E_{i-1,j} - S_{i,j}) \\
&= \max(\alpha + \beta - \Delta v[j], \beta + E_{i-1,j} - S_{i,j}) \\
&= \max(\alpha + \beta - \Delta v[j], \beta + E_{i-1,j} \underline{- S_{i-1,j} + S_{i-1,j}} - S_{i,j}) \\
&= \max(\alpha + \beta - \Delta v[j], \beta + \Delta E_{in}[j] - \Delta v[j]) \\
&= \max(\Delta E_{in}[j], \alpha) + \beta - \Delta v[j]
\end{aligned}$$

□

Theorem D.2.3. *Given $\Delta v[j-1], \Delta v[j], \Delta H[j]$ and $\Delta F[j-1]$, $\Delta F[j]$ can be computed by the equation*

$$\Delta F[j] = \Delta v[j-1] - \Delta v[j] + \Delta H[j] + \max(\Delta F[j-1], \alpha)$$

Proof. From the definition, $\Delta F[j] = F_{i,j} - S_{i,j}$.

$$\begin{aligned}
\Delta F[j] &= F_{i,j} - S_{i,j} \\
&= F_{i,j} \underline{- S_{i-1,j} + S_{i-1,j}} - S_{i,j} \\
&= F_{i,j} - S_{i-1,j} - \Delta v[j] \\
&= F_{i,j} \underline{- S_{i-1,j-1} + S_{i-1,j-1}} - S_{i-1,j} - \Delta v[j] \\
&= F_{i,j} - S_{i-1,j-1} - \Delta h_{in}[j] - \Delta v[j] \\
&= F_{i,j} \underline{- S_{i,j-1} + S_{i,j-1}} - S_{i-1,j-1} - \Delta h_{in}[j] - \Delta v[j] \\
&= F_{i,j} - S_{i,j-1} + \Delta v[j-1] - \Delta h_{in}[j] - \Delta v[j] \\
&= \max(\beta + F_{i,j-1}, \alpha + \beta + S_{i,j-1}) - S_{i,j-1} + \Delta v[j-1] \\
&\qquad\qquad\qquad\qquad - \Delta h_{in}[j] - \Delta v[j] \\
&= \max(\beta + F_{i,j-1} - S_{i,j-1}, \alpha + \beta + S_{i,j-1} - S_{i,j-1}) \\
&\qquad\qquad\qquad\qquad + \Delta v[j-1] - \Delta h_{in}[j] - \Delta v[j] \\
&= \max(\beta + \Delta F[j-1], \alpha + \beta) \\
&\qquad\qquad\qquad\qquad + \Delta v[j-1] - \Delta h_{in}[j] - \Delta v[j] \\
&= \max(\Delta F[j-1], \alpha) + \Delta v[j-1] + \beta - \Delta h_{in}[j] - \Delta v[j] \\
&= \max(\Delta F[j-1], \alpha) + \Delta v[j-1] + \Delta H[j] - \Delta v[j] \\
&= \Delta v[j-1] - \Delta v[j] + \Delta H[j] + \max(\Delta F[j-1], \alpha)
\end{aligned}$$

□

Theorem D.2.4. *$\forall j > 0$, the $\Delta v[j]$ values can be computed by the following recurrence:*

$$\Delta v[1] = \max\left(\Delta v[0] + \alpha + \Delta H[1], LB[1]\right)$$

$$\underset{\forall j>1}{\Delta v}[j] = \max \begin{cases} LB[j] \\ \max\limits_{i=1}^{j-1} \left(LB[i] + \alpha + \sum\limits_{k=i+1}^{j} \Delta H[k] \right) \\ \Delta v[0] + \alpha + \Delta H[1] + \Delta H[2] \\ \qquad\qquad + \ldots + \Delta H[j-1] + \Delta H[j] \end{cases} \tag{D.5}$$

Proof. By induction on j.

Base case ($j = 1$): This is established by equation D.4 following Theorem D.2.1 because $\Delta F[0] = F_{i,0} - S_{i,0} = \alpha$.

Induction step: Assume that the recurrence is true for all indices up to $j-1$, *i.e.*:

$$\Delta v[j-1] = \max \begin{cases} LB[j-1] \\ \max\limits_{i=1}^{j-2} \left(LB[i] + \alpha + \sum\limits_{k=i+1}^{j-1} \Delta H[k] \right) \\ \Delta v[0] + \alpha + \Delta H[1] + \Delta H[2] \\ \qquad\qquad + \ldots + \Delta H[j-1] \end{cases}$$

Note in particular that in the latter two alternatives, $S_{i,j-1}$ comes from a horizontal gap. That means that $S_{i,j-1}$ was set equal to $F_{i,j-1}$, and in particular, that $\Delta F[j-1] = 0$. Substituting the formula above into equation D.4, produces

$$\Delta v[j] = \max \begin{cases} LB[j] \\ \\ LB[j-1] + max(\Delta F[j-1], \alpha) + \Delta H[j] \\ \\ \max\limits_{i=1}^{j-2} \left(LB[i] + \alpha + \sum\limits_{k=i+1}^{j-1} \Delta H[k] \right) \\ \qquad\qquad + max(\Delta F[j-1], \alpha) + \Delta H[j] \\ \\ \Delta v[0] + \alpha + \Delta H[1] + \Delta H[2] + \ldots + \Delta H[j-1] \\ \qquad\qquad + max(\Delta F[j-1], \alpha) + \Delta H[j] \end{cases}$$

In the second alternative, the maximum score, $S_{i,j}$, comes from a gap which starts in column $j-1$ and derives from the score $S_{i,j-1}$ which derives from a score in the preceding row (hence the term $LB[j-1]$). In this event, $S_{i,j-1}+\alpha+\beta \geq F_{i,j-1}+\beta$ else the gap would have started further to the left. Then $\alpha \geq F_{i,j-1} - S_{i,j-1} \geq \Delta F[j-1]$

and so $max(\Delta F[j-1], \alpha) = \alpha$.

In each of the cases in the third and fourth alternatives, the maximum score, $S_{i,j}$, extends a gap which was also chosen as the best score for column $j-1$. As stated above, for these gaps, $F[j-1] = 0$, and so $max(\Delta F[j-1], \alpha) = 0$.

Replacing the term $max(\Delta F[j-1], \alpha)$ with α in the second alternative and with 0 in the third and fourth alternatives yields

$$\Delta v[j] = \max \begin{cases} LB[j] \\ LB[j-1] + \alpha + \Delta H[j] \\ \max_{i=1}^{j-2} \left(LB[i] + \alpha + \sum_{k=i+1}^{j-1} \Delta H[k] \right) + \Delta H[j] \\ \Delta v[0] + \alpha + \Delta H[1] + \Delta H[2] + \ldots + \Delta H[j-1] \\ \qquad + \Delta H[j] \end{cases}$$

$$= \max \begin{cases} LB[j] \\ \max_{i=1}^{j-1} \left(LB[i] + \alpha + \sum_{k=i+1}^{j} \Delta H[k] \right) \\ \Delta v[0] + \alpha + \Delta H[1] + \Delta H[2] \\ \qquad + \ldots + \Delta H[j-1] + \Delta H[j] \end{cases}$$

□

Theorem D.2.5. *$\forall j \in \{1, \ldots, n\}, \forall k \in \{0, \ldots, \log_2 j\}$, partial sums of length 2^k, $PS[j,k] = \Delta H_{in}[j - 2^k + 1] + \Delta H_{in}[j - 2^k + 2] + \ldots + \Delta H[j]$ can be computed in $\lceil \log_2 n \rceil$ rounds in $O(n \log n)$ time.*

Proof. See proof in Appendix A, Theorem 6.4. □

Theorem D.2.6. *$\forall j \in \{1, \ldots, n\}$, the $\Delta v[j]$ can be computed in $\lceil \log_2(n) \rceil + 1$ rounds in $O(n \log n)$ time if $\forall j \in \{1, \ldots, n\}$, $\forall k \in \{0, \ldots, \log_2 j\}$ the partial sums $PS[k,j]$ are available.*

Proof. By induction on round k. Let $VHG[j]$ be a lower bound for the value of $\Delta v[j]$ obtained through a horizontal gap. $VHG[j]$ is updated at each round and gives us $\Delta v[j]$ after the final round which selects the maximum of $VHG[j]$ and $LB[j]$.

We define $VHG[j]$ during round $k = 0 \cdots \log_2 j$ by
for $k = 0$:

$$\begin{aligned} &\text{for } j = 1 : \\ &\quad VHG[1] = \Delta v[0] + \alpha + \Delta H[1] && \text{(D.6)} \\ &\quad \forall j > 1 : \\ &\quad VHG[j] = LB[j-1] + \alpha + \Delta H[j] && \text{(D.7)} \end{aligned}$$

for $k > 0$:

$$\forall j > 2^{k-1} :$$

$$VHG[j] = \max \begin{cases} VHG[j] \\ VHG[j-2^{k-1}] + \Delta H[j-2^{k-1}+1] + \ldots + \Delta H[j] \\ \qquad = VHG[j-2^{k-1}] + PS[j, k-1] \end{cases} \tag{D.8}$$

Consider the expansion of Equation (D.5):

$$\underset{\forall j \geq 1}{\Delta v}[j] = \max \begin{cases} LB[j] \\ LB[j-1] + \alpha + \Delta H[j] \\ LB[j-2] + \alpha + \Delta H[j-1] + \Delta H[j] \\ \vdots \\ LB[1] + \alpha + \Delta H[2] + \Delta H[3] + \cdots + \Delta H[j] \\ \Delta v[0] + \alpha + \Delta H[1] + \Delta H[2] \\ \qquad + \ldots + \Delta H[j-1] + \Delta H[j] \end{cases} \tag{D.9}$$

We show that after round k, for all j, $VHG[j]$ is the maximum of the second through the $2^k + 1$ alternatives in Equation (D.9).

Base case: Round 0 computes

$$\begin{aligned} VHG[1] &= \Delta v[0] + \alpha + \Delta H[1] \\ \forall j > 1, VHG[j] &= LB[j-1] + \alpha + \Delta H[j]. \end{aligned} \tag{D.10}$$

Equation (D.10) computes the second ($= 2^0 + 1$) alternative in Equation (D.9).

For the induction step, assume that after round $k-1$, for $0 \le k-1 < \lceil \log_2(n) \rceil$, $VHG[j]$ is the maximum of the second through $2^{k-1}+1$ alternatives in Equation (D.9). That is:

$$\underset{\forall j>1}{VHG[j]} = \max \begin{cases} LB[j-1] + \alpha + \Delta H[j] \\ LB[j-2] + \alpha + \Delta H[j-1] \\ \qquad\qquad\qquad + \Delta H[j] \\ \vdots \\ LB[j-2^{k-1}+1] + \alpha + \Delta H[j-2^{k-1}+2] \\ \qquad\qquad\qquad + \ldots + \Delta H[j-1] + \Delta H[j] \end{cases} \tag{D.11}$$

Note specifically that $VHG[j-2^{k-1}]$ is the maximum of its second through $2^{k-1}+1$ alternatives, for $j - 2^{k-1} > 1$:

$$\underset{\forall j|(j-2^{k-1})>1}{VHG[j-2^{k-1}]} = \max \begin{cases} LB[j-2^{k-1}-1] + \alpha + \Delta H[j-2^{k-1}] \\ LB[j-2^{k-1}-2] + \alpha + \Delta H[j-2^{k-1}-1] \\ \qquad\qquad\qquad + \Delta H[j-2^{k}] \\ \vdots \\ LB[j-2^{k-1}-2^{k-1}+1] + \alpha \\ \qquad\qquad + \Delta H[j-2^{k-1}-2^{k-1}+2] \\ \qquad\qquad + \ldots + \Delta H[j-2^{k-1}-1] \\ \qquad\qquad + \Delta H[j-2^{k-1}] \end{cases} \tag{D.12}$$

Then, round k computes:

$$\forall j > 2^{k-1} : \tag{D.13}$$

$$VHG[j] = \max \begin{cases} VHG[j] \\ VHG[j - 2^{k-1}] + \Delta H[j - 2^{k-1} + 1] + \ldots + \Delta H[j] \\ \qquad\qquad = VHG[j - 2^{k-1}] + PS[j, k-1] \end{cases} \tag{D.14}$$

Round k, Equation (D.8), in effect adds the sum of the 2^{k-1} terms $\Delta H[j - 2^{k-1} + 1] + \Delta H[j - 2^{k-1} + 2] + \ldots + \Delta H_{in}[j] = PS[j, k-1]$ to every alternative in Equation (D.12). The maximum calculation in Equation (D.8) is computed over these 2^{k-1} alternatives as well as the 2^{k-1} alternatives from Equation (D.11). This leaves 2^k alternatives and these are exactly the second through $2^k + 1$ alternatives in Equation (D.5). At the end of round $i = \lceil \log_2(n) \rceil$, all VHG are equal to the maximum of their second through $2^{\lceil \log_2(n) \rceil} + 1 \geq n + 1$ terms in Equation (D.5). Since $\Delta v[n]$ only has $n + 1$ terms, with a single additional maximization to set

$$VHG[j] = \max\big(LB[j], VHG[j]\big), \forall j > 0,$$

$VHG[j]$ will be exactly $\Delta v[j]$ as computed in Equation D.5.

If the partial sums, $PS[j, k]$ for the ΔH terms are available, then each round has at most one addition and one maximum operation per index j, except the final round which has one additional maximum operation, so the total number of operations per round is linear in n and the total time is $O(n \log n)$. □

Theorem D.2.7. $\forall j \in \{1, \ldots, n\}$, *$\Delta H_{out}[j]$ can be computed in $O(n)$ time, given $\Delta H[j]$ and $\Delta v[j]$ for all j.*

Proof. From the definitions,

$$\begin{aligned}
\Delta H_{out}[j] &= \beta - \Delta h_{out}[j] \\
&= \beta - S_{i,j} + S_{i,j-1} \\
&= \beta - S_{i,j} + \underline{S_{i-1,j} - S_{i-1,j}} + S_{i,j-1} \\
&= \beta - \Delta v[j] - S_{i-1,j} + S_{i,j-1} \\
&= \beta - \Delta v[j] - S_{i-1,j} + \underline{S_{i-1,j-1} - S_{i-1,j-1}} + S_{i,j-1} \\
&= \beta - \Delta v[j] - \Delta h_{in}[j] + \Delta v[j-1] \\
&= \Delta H[j] - \Delta v[j] + \Delta v[j-1]
\end{aligned}$$

So

$$\Delta H_{out}[j] = \Delta H[j] - \Delta v[j] + \Delta v[j-1]. \tag{D.15}$$

Since we have all values $\Delta H[j]$ and $\Delta v[j]$, the $\Delta H_{out}[j]$ can be computed immediately. These computations are done for each $j \in [1, \ldots, n]$, resulting in $O(n)$ time. □

For a fixed letter x, define $SB[j] = subst(x, y_j) - beta$

Theorem D.2.8. *$\forall j \in \{1, \ldots, n\}$, $\Delta V[j]$ and $\Delta H_{out}[j]$ can be computed in $\lceil n/W \rceil * (\lceil \log_2 W \rceil + 2)$ rounds and $O((n * \log W)/W))$ time, where W is the number of ΔV values held in a computer word of length w.*

Proof. Unlike previous proofs, this one will be illustrated using pseudocode. An array of words, denoted $Vval$, holds the $\Delta V[j]$ values and are numbered from 0 (containing the lowest j indices) to $\lceil n/W \rceil - 1$ (denoted $Vval_0, Vval_1$, etc.). Each word holds W values, indexed from 0 to $W-1$, and each value occupies w/W bits. Similar arrays, denoted $Eval$, $Hval$, $SBval$, and $PSval$, hold the ΔE, ΔH, SB, and partial sum values, PS, respectively. An array named all_{alpha} has α as it's value at every location. The $Vval$ words are processed in order starting with word 0. Each word is processed independently in $\lceil \log_2 W \rceil + 1$ rounds and the values are determined as described in Theorems D.2.6 and D.2.5. For the $Vval$ it suffices to show that 1) within a single word, the number of operations is linear in the *number of rounds* and 2) information can be efficiently transferred from one word to the next. Below we show how $Vval_0$ and $Vval_1$ are processed. The remaining words are processed similarly to $Vval_1$. The $\Delta H_{out}[j]$, stored in the $Hval$, and $\Delta E_{out}[j]$ are computed by a small set of instructions for each word, also shown below. We assume that preprocessing before computing row 1 initializes arrays $Hval$, $PSval$, $Eval$, and $SBval$ as follows:

```
for $k = 0$ to $n/W$
    for $h = 0$ to $W - 1$
        $Hval_k[h] = \Delta H_{in}[k * W + h + 1]$
        $SBval_k[h] = SB[k * W + h + 1]$
    $PSval_k = \Delta Hval_k$
    $Eval_k = \alpha + \beta$
```

$Vval_0$: Round 0. This round computes the first alternative, LB from Theorem D.2.4, Equation (D.5) for $j = 0 \ldots W$.

$$\begin{aligned} SBval_0[0] &= \max(\Delta V[0], SBval_0[0]) \\ DVbar_0 &= SBval_0 + Hval_0 \\ DVbar_0 &= \max(Eval_0, DVbar_0) \end{aligned}$$

The re-initialization of $SBval_0[0]$ assures that if $\Delta v[0] = 0$ to exclude an initial gap penalty, then the corresponding ΔV value ($\Delta V[0] = -G$) is used if it is larger than $SB[0]$.

$Vval_0$: Round 1. This round computes the second alternative from Theorem D.2.4, Equation (D.5) for $j = 1 \ldots W$.

$$\begin{aligned} Shifted &= (DVbar_0 << 8) \\ Vval_0 &= Shifted + all_{alpha} + Hval_0 \end{aligned}$$

The values in $DVbar_0$ are shifted 8 bits higher (1 byte) to put the $j - 1$ value in the j position. Each addition of the W values stored in $Shifted$, all_{alpha}, and $Hval_0$ is done simultaneously as one operation. This is possible if the number of bits for each value is large enough to prevent overflow into the next value or, as we have done, by using an SIMD "addition with saturation" instruction which restricts each add to a single byte and, in the case of an overflow, stores the maximum (or minimum) possible value.

$Vval_0$: Rounds $i = 2, \ldots, \log W$. These rounds compute the remaining alternatives from Theorem D.2.4, Equation (D.5), following the method presented in Theorem D.2.6.

$$\begin{aligned}
ShiftVval &= Vval_0 << (2^{i-1} * 8)\\
ShiftVval &= ShiftVval \text{ OR } (all_{alpha} >> 2^{k-i} * 8)\\
Sum &= ShiftVval + PSval_0\\
Vval_0 &= \max(Vval_0, Sum)\\
ShiftPS &= PSval_0 << (2^{i-1} * 8)\\
PSval_0 &= PSval_0 + ShiftPS
\end{aligned}$$

The first, third, and fourth instructions are the max() calculation of Theorem D.2.6, Equation (D.8). The second instruction is to ensure that the values at the beginning of the word aren't affected by the max calculations. The values at the beginning of the word should not be changing, since the values that we wish to compare are the ones that are being shifted over, and we don't want to compare the zeros that we are shifting in to anything. The max() calculation can be performed with a single SIMD instruction. $ShiftVval$ is the ΔV value in the second alternative of Equation (D.8) and the addition of $ShiftVval$ and $PSval_0$ is the computation of the sum in the second alternative. Note the shift is multiplied by 8 because we are using one byte to store each value. The last two instructions prepare the partial sums variable for the next round using the partial sums calculation of Theorem D.2.5.

$Vval_0$: Round $\log W + 1$. This round computes the maximum of 1) the first alternative, LB, from Theorem D.2.4, Equation (D.5) and 2) the maximum of the remaining alternatives, calculated above. Before computing the max of 1) and 2), 2) is saved as $DVLmax$ in order to allow carrying over into the next word.

$Vval_1$: Round 0 is identical to the one for $Vval_0$ except the re-initialization of $SBval_1[0]$ is not done:

$$\begin{aligned}
DVbar_0 &= SBval_0 + Hval_0\\
DVbar_0 &= \max(Eval_0, DVbar_0)
\end{aligned}$$

$Vval_1$: Rounds $1 \ldots \log W + 1$ are identical to the ones for $Vval_0$, however there is an added step between round $\log W$ and round $\log W + 1$.

```
$for\ j\ in\ 0 \ldots W :$
    $Carry[j] = DVLmax[W-1]$
$Sum = Carry + PSval_1$
$Vval_1 = max(Sum, Vval_1)$
```

Since the computation for $Vval_0$ is already complete, $Vval_0[W-1]$ contains its final value, $\Delta V[W]$. This allows the computations of Theorem D.2.4 Equation D.5 all the way back to $\Delta v[0]$ to be done with a single computation rather than re-computing each alternative.

For the $\Delta H_{out}[j]$, the $Hval$ variables are now used to hold the $\Delta H_{out}[j]$ (which are used subsequently as the $\Delta H_{in}[j]$ for the next row). We compute them for a generic word $Hval_k$ with the following instructions:

$$\begin{aligned} Hval_k &= Hval_k - Vval_k \\ ShiftVval &= Vval_k << (1 * 8) \quad \text{[i.e., one byte]} \\ ShiftVval[0] &= Vval_{k-1}[W-1] \\ Hval_k &= Hval_k + ShiftVval_k \end{aligned} \tag{D.16}$$

The first instruction computes the subtraction of $\Delta H[j]$ and $\Delta v[j]$ as shown in Theorem D.2.7 Equation (D.15). The second and third instructions shift the $\Delta V[j]$ values up by one in preparation for the addition of $\Delta v[j-1]$ in Theorem D.2.7 Equation (D.15), which is performed by the fourth instruction. Note that since $Vval_k$ is shifted up by one value, the last value from the previous word, $Vval_{k-1}[W-1]$, is inserted at the beginning.

There are $\lceil n/W \rceil - 1$ words holding the $\Delta V[j]$ values and each word is processed for $\lceil \log_2 W \rceil + 1$ rounds using a fixed number of operations per round. There are $\lceil n/W \rceil - 1$ words holding the $\Delta H_{out}[j]$ values and each word is computed with a fixed number of operations. The total time is therefore $O\left(n \log W/W\right)$. $\square$

Appendix E

Proofs on SIMD Tandem Alignment

E.1 Preliminaries

Consider the global Wraparound Dynamic Programming algorithm (WDP) for tandem alignment of a pattern sequence of length n against a text of length m.

Definitions

Σ is the alphabet of characters over which the alignment is performed. G is the gap penalty. *subst* is a table of substitution scores where $subst(a, b) > 2G$ is the substitution score from a to b with $a, b \in \Sigma$. W is the dynamic programming scoring matrix for the wraparound dynamic programming algorithm.

Recursion: Initialize row zero ($1 \leq j \leq n$):

$$S[0,0] = 0$$

$$S[0,j] = S[0,0] + j \cdot G$$

Initialize column zero ($1 \leq i \leq m$):

$$S[i,0] = S[0,0] + j \cdot G$$

First pass ($i \geq 1, j \geq 1$):

$$S[i,j] = \begin{cases} \begin{cases} S[i-1,0] + subst(a_i, b_j) & \backslash\backslash\text{diagonal} \\ S[i-1,n] + subst(a_i, b_j) & \backslash\backslash\text{wraparound diagonal} \\ S[i,0] + G & \backslash\backslash\text{from left} \\ S[i-1,1] + G & \backslash\backslash\text{from above} \end{cases} \\ \text{if } j = 1 \\ \begin{cases} S[i-1,j-1] + subst(a_i, b_j) & \backslash\backslash\text{diagonal} \\ S[i,j-1] + G & \backslash\backslash\text{from left} \\ S[i-1,j] + G & \backslash\backslash\text{from above} \end{cases} \\ \text{if } j > 1 \end{cases} \tag{E.1}$$

Second pass ($i \geq 1, 1 \leq j < n$):

$$S[i,j] = \max \begin{cases} S[i,j] \\ \begin{cases} S[i,n] + G & \text{if } j = 1 \\ S[i,j-1] + G & \text{if } j > 1 \end{cases} \end{cases} \tag{E.2}$$

Definition 3. *We define two sets of horizontal and vertical differences for a given row i, one based on W after the first pass and one based on W after the second pass.*

For row 0, for all j, two sets of horizontal differences:

$$\Delta h_1[0,j] = \Delta h_2[0,j] = G$$

For row $i > 0$, two sets of vertical and horizontal differences:

After the first pass:

$$\begin{aligned}
\Delta v_1[i,j] &= S[i,j] - S[i-1,j] \\
\Delta h_1[i,j] &= S[i,j] - S[i,j-1] \\
\Delta V_1[i,j] &= \Delta v_1[i,j] - G \\
\Delta H_1[i,j] &= G - \Delta h_1[i,j]
\end{aligned}$$

After the second pass:

$$\Delta v_2[i,j] = S[i,j] - S[i-1,j]$$

$$\Delta h_2[i,j] = S[i,j] - S[i,j-1]$$

$$\Delta V_2[i,j] = \Delta v_2[i,j] - G$$

$$\Delta H_2[i,j] = G - \Delta h_2[i,j]$$

Sums

Definition 4. *For a row $i > 0$, SPS_i (Second Pass Sum) is the sum of the Δh_2 values in row $i-1$ (the previous row) for $j = 2 \ldots n$, i.e. $SPS_i = \sum_{j=2}^{n} \Delta h_2[i-1,j] = S[i-1,n] - S[i-1,1]$.*

Lemma E.1.1. *In row $i > 0$, given the values $\Delta v_1[i,1]$, $\Delta v_1[i,n]$, and SPS_i, $\Delta v_2[i,1]$ can be calculated as*

$$\Delta v_2[i,1] = \max(\Delta v_1[i,1], SPS_i + \Delta v_1[i,n] + G).$$

Proof. In row i, $SPS_i = S[i-1,n] - S[i-1,1]$. After the first pass , $\Delta v_1[i,1] = S[i,1] - S[i-1,1]$, and $\Delta v_1[i,n] = S[i,n] - S[i-1,n]$. In the second pass, $S[i,1] = \max(S[i,1], S[i,n] + G)$. Thus, $\Delta v_2[i,1] = \max(S[i,1], S[i,n] + G) - S[i-1,1]$.

$$\begin{aligned}
\Delta v_2[i,1] &= \max(S[i,1], S[i,n] + G) - S[i-1,1] \\
&= \max(S[i,1] - S[i-1,1], S[i,n] + G - S[i-1,1]) \\
&= \max(\Delta v_1[i,1], S[i,n] + G - S[i-1,1]) \\
&= \max(\Delta v_1[i,1], S[i,n] - S[i-1,n] + S[i-1,n] + G - S[i-1,1]) \\
&= \max(\Delta v_1[i,1], S[i,n] - S[i-1,n] + G + S[i-1,n] - S[i-1,1]) \\
&= \max(\Delta v_1[i,1], \Delta v_1[i,n] + G + S[i-1,n] - S[i-1,1]) \\
&= \max(\Delta v_1[i,1], \Delta v_1[i,n] + G + SPS_i)
\end{aligned}$$

□

Lemma E.1.2. *In row $i > 0$, given the values $\Delta v_2[i,1]$ and $\Delta v_2[i,n]$, SPS_{i+1} for row $i+1$ can be calculated as*

$$SPS_{i+1} = SPS_i + \Delta v_2[i,n] - \Delta v_2[i,1].$$

Proof. In row i, $SPS_i = S[i-1,n] - S[i-1,1]$ and in row $i+1$ $SPS_{i+1} = S[i,n] - S[i,1]$. After the second pass $S[i,1] = S[i-1,1] + \Delta v_2[i,1]$ and $S[i,n] = S[i-1,n] + \Delta v_2[i,n]$. Thus,

$$\begin{aligned}
SPS_{i+1} &= S[i,n] - S[i,1] \\
&= S[i-1,n] + \Delta v_2[i,n] - (S[i-1,1] + \Delta v_2[i,1]) \\
&= S[i-1,n] - S[i-1,1] + \Delta v_2[i,n] - \Delta v_2[i,1] \\
&= SPS_i + \Delta v_2[i,n] - \Delta v_2[i,1]
\end{aligned}$$

□

Lemma E.1.3. *$\Delta v_1[i,1]$ can be computed as:*

$$\Delta v_1[i,1] = \max \begin{cases} SPS_i + subst(a_i, b_1) & \backslash\backslash \textit{diagonal wraparound} \\ & \quad \textit{substitution} \\ subst(a_i, b_1) - \Delta h_2[i-1,1] & \backslash\backslash \textit{diagonal substitution} \\ G & \backslash\backslash \textit{vertical gap} \end{cases}$$

Proof. By the definition,

$$\Delta v_1[i,1] = S[i,1] - S[i-1,1]$$

Note that $S[i,1] = \max(S[i-1,1] + G, S[i-1,0] + subst(a_i,b_1), S[i-1,n] + subst(a_i,b_1))$

Suppose $S[i,1] = \boldsymbol{S[i-1,1]+G}$. Then $S[i,1]-S[i-1,1] = S[i-1,1]+G-S[i-1,1] = G$.

Suppose $S[i,1] = \boldsymbol{S[i-1,0]+subst(a_i,b_1)}$. $S[i-1,0] = S[i-1,1]-\Delta h_2[i-1,1]$, so

$$\begin{aligned} S[i,1] - S[i-1,1] & \\ &= S[i-1,0] + subst(a_i,b_1) - S[i-1,1] \\ &= S[i-1,1] - \Delta h_2[i-1,1] + subst(a_i,b_1) - S[i-1,1] \\ &= -\Delta h_2[i-1,1] + subst(a_i,b_1) \end{aligned}$$

Suppose $S[i,1] = \boldsymbol{S[i-1,n]+subst(a_i,b_1)}$. Since $S[i-1,n]-S[i-1,1]$ is exactly SPS_i,

$$\begin{aligned} S[i,1] - S[i-1,1] & \\ &= S[i-1,n] + subst(a_i,b_1) - S[i-1,1] \\ &= SPS_i + subst(a_i,b_1) \end{aligned}$$

Hence,

$$\Delta v_1[i,1] = \max \begin{cases} SPS_i + subst(a_i, b_1) \\ subst(a_i, b_1) - \Delta h_2[i-1,1] \\ G \end{cases}$$

□

Definition 5. $M = \max\limits_{a,b\in\Sigma}(subst(a,b))$

Theorem E.1.4. *For all $i > 0$,*

- $SPS_i \leq -G$
- $G \leq \Delta v_1[i,1] \leq M - G$
- $G \leq \Delta v_2[i,1] \leq M - G.$

Proof. By induction.

Base case:

$\boldsymbol{SPS_1}$ **:** At row 1, SPS_1 is $(n-1)\cdot G$. Since $G < 0$, $SPS_1 < 0 \leq -G$.

$\boldsymbol{\Delta v_1}$ **:** In row 1, $\Delta h_2[0,j] = G$ for all j due to the initialization. From Lemma E.1.3,

$$\Delta v_1[1,1] = \max \begin{cases} SPS_1 + subst(a_1, b_1) \\ subst(a_1, b_1) - \Delta h_2[1-1,1] \\ G \end{cases}$$

Since $SPS_1 < -G = -\Delta h_2[0,1]$, the middle term is larger than the first term, so $\Delta v_1[1,1] = \max(subst(a_1,b_1) - G, G)$. The maximum value of $subst(a_1,b_1)$ is M, so $G \leq \Delta v_1[1,1] \leq M - G$.

$\boldsymbol{\Delta v_2[1,1]}$ **:** By Lemma E.1.1, $\Delta v_2[1,1] = \max(\Delta v_1[1,1], SPS_1 + \Delta v_1[1,n] + G)$. Note that $\Delta v_2[1,1] \geq \Delta v_1[1,1]$ so $\Delta v_2[1,1] \geq G$

If $\Delta v_2[1,1] = \Delta v_1[1,1]$, then $G \leq \Delta v_2[1,1] \leq M - G$.

Suppose instead that $\Delta v_2[1,1] = SPS_1 + \Delta v_1[1,n] + G$, then

$$\Delta v_2[1,1] \leq -G + \Delta v_1[1,n] + G$$
$$\Delta v_2[1,1] \leq \Delta v_1[1,n]$$

By Theorem 6.1 of Appendix A of *Affine Gap Paper*, $0 \leq \Delta V_1[1,n] \leq M - 2G$ which implies that $G \leq \Delta v_1[1,n] \leq M - G$. Thus, $\Delta v_2[1,1] \leq M - G$

Hence, $G \leq \Delta v_2[1,1] \leq M - G$.

Induction step: We assume that $SPS_i \leq -G$, $G \leq \Delta v_1[i,1] \leq M - G$, and $G \leq \Delta v_2[i,1] \leq M - G$ for all i up to $k-1$. We have three conditions to prove:

Condition 1: $SPS_k \leq -G$:

We know that after the second pass in row $k-1$,

$$S[k-1,1] = \max(S[k-1,1], S[k-1,n] + G)$$
$$\Rightarrow S[k-1,1] \geq S[k-1,n] + G.$$

$SPS_k = S[k-1,n] - S[k-1,1]$, so $SPS_k \leq S[k-1,n] - (S[k-1,n] + G) \Rightarrow SPS_k \leq -G$.

Condition 2: $G \leq \Delta v_1[k,1] \leq M - G$ From Lemma E.1.3,

$$\Delta v_1[k,1] = \max \begin{cases} SPS_k + subst(a_i, b_n) \\ subst(a_k, b_1) - \Delta h_2[k-1,1] \\ G \end{cases}$$

we know that $\Delta v_1[k,1] \geq G$.

Consider $SPS_k + subst(a_k, b_n)$. $SPS_i \leq -G$ and $subst(a_k, b_n) \leq M$, so $SPS_k + subst(a_k, b_n) \leq M - G$. Similarly, because $\Delta h_2[k-1,1] \geq G$, $subst(a_k, b_1) - \Delta h_2[k-1,1] \leq M - G$.

Hence, $G \leq \Delta v_1[k,1] \leq M - G$.

Condition 3: $G \leq \Delta v_2[k,1] \leq M - G$

By Lemma E.1.1, $\Delta v_2[k,1] = \max(\Delta v_1[k,1], SPS_k + \Delta v_1[k,n] + G)$. Note that $\Delta v_2[k,1] \geq \Delta v_1[k,1]$ so $\Delta v_2[k,1] \geq G$.

If $\Delta v_2[k,1] = \Delta v_1[k,1]$, then $G \leq \Delta v_2[k,1] \leq M - G$.

Suppose instead that $\Delta v_2[k,1] = SPS_k + \Delta v_1[k,n] + G$, then

$$\Delta v_2[k,1] \leq -G + \Delta v_1[k,n] + G$$
$$\Delta v_2[k,1] \leq \Delta v_1[k,n]$$

By Theorem 6.1 of Appendix A of *Affine Gap Paper*, $0 \leq \Delta V_1[k,n] \leq M - 2G$ which implies that $G \leq \Delta v_1[k,n] \leq M - G$. Thus, $\Delta v_2[k,1] \leq M - G$

Hence, $G \leq \Delta v_2[k,1] \leq M - G$. □

Lemma E.1.5. $0 \leq \Delta V_1[i,1] \leq M - 2G$ *and* $0 \leq \Delta V_2[i,1] \leq M - 2G$

Proof. From Theorem E.1.4, $G \leq \Delta v_1[i,1] \leq M - G$ and $G \leq \Delta v_2[i,1] \leq M - G$. Subtracting G from both sides of each equation gives us exactly

$$0 \leq \Delta V_1[i,1] \leq M - 2G$$

and

$$0 \leq \Delta v_2[i,1] \leq M - 2G.$$

□

Lemma E.1.6. $\forall j \in \{1,\ldots,n\}, \forall i \in \{0,\ldots,\log_2 j\}$, *partial sums of length* 2^i,

$$PS_1[j,i] = \Delta H_1[j - 2^i + 1] + \Delta H_1[j - 2^i + 2] + \ldots + \Delta H_1[j]$$

and

$$PS_2[j,i] = \Delta H_2[j - 2^i + 1] + \Delta H_2[j - 2^i + 2] + \ldots + \Delta H_2[j]$$

can each be computed in $\lceil \log_2 n \rceil$ *rounds in* $O(n \log n)$ *time.*

Proof. See proof of Theorem B.0.11. □

Theorem E.1.7. $\forall j \in \{2,\ldots,n\}$, $\Delta V_1[i,j]$ *can be computed in* $\lceil \log_2(n) \rceil + 1$ *rounds in* $O(n/W \log W)$ *time if* $\forall j \in \{1,\ldots,n\}, \forall i \in \{0,\ldots,\log_2 j\}$ *the partial sums* $PS_2[i,j]$ *are available.*

Proof. By Lemma 6.3, $\Delta v_1[i, 1]$ can be computed in constant operations. Given ΔH_2 values of row $i - 1$, $\Delta V_1[j]$ of row i where $j > 1$ can be computed exactly as in Theorem B.0.12, in $O(n/W \log W)$ time. Hence, $\Delta v_1[i, j]$ can be computed in $O(n/W \log W)$ time. □

Theorem E.1.8. *$\forall j \in \{2, \ldots, n\}$, $\Delta V_2[j]$ can be computed in $\lceil \log_2(n) \rceil + 1$ rounds in $O(n \log n)$ time if $\forall j \in \{1, \ldots, n\}, \forall i \in \{0, \ldots, \log_2 j\}$ the partial sums $PS_1[i, j]$ are available.*

Proof. By Lemma 6.1, $\Delta v_2[i, 1]$ can be computed in constant operations. Given ΔH_1 values of row i, $\Delta V_2[j]$ of row i where $j > 1$ can be computed exactly as in Theorem B.0.12, in $O(n/W \log W)$ time. Hence, $\Delta v_2[i, j]$ can be computed in $O(n/W \log W)$ time. □

Theorem E.1.9. *$\forall j \in \{1, \ldots, n\}$, $\Delta V_1[j]$, $\Delta H_1[j]$, $\Delta V_2[j]$, and $\Delta H_2[j]$ can be computed in time $O(n \log W/W))$, where W is the number of ΔV values held in a word of length w.*

Proof. Note that ΔV_1 and ΔH_1 and ΔV_2 and ΔH_2 can be computed in a manner analogous to ΔV and ΔH_{out} of Theorem B.0.13. This results in ~ 2 times the work, for time $O(2n \log W/W) \Rightarrow O(n \log W/W)$. □

List of Journal Abbreviations

ACM	Association for Computing Machinery
Am. J. Med. Genet.	American Journal of Medical Genetics
AWOCA	Australasian Workshop on Combinatorial Algorithms
B Neuropsychiatr. Genet.	Part B: Neuropsychiatric Genetics
BICoB	Bioinformatics and Computational Biology
Dement. Geriatr. Cogn. Disord.	Dementia and Geriatric Cognitive Disorders
Dokl. Akad. Nauk	Doklady Akademii Nauk
FEMS	Federation of European Microbiological Societies
Genes Dev.	Genes & Development
Genome Announc.	Geneome Announcements
ICCMSE	International Conference of Computational Methods in Sciences and Engineering
Int. J. Found. Comput. Sci.	International Journal of Foundations in Computer Science
JACM	Journal of the Association for Computing Machinery
J. Clin. Microbiol.	Journal of Clinical Microbiology
J. Mol. Biol.	Journal of Molecular Biology
MIT	Massachusetts Institute of Technology
Natl. Biomed. Res. Found	National Biomedical Research Foundation
Nucleic Acids Res.	Nucleic Acids Research
PNAS	Proceedings of the National Academy of Sciences
Psychol. Med	Psychological Medicine

RECOMB	Research in Computational Molecular Biology

References

[1] Allard, M. W., Muruvanda, T., Strain, E., Timme, R., Luo, Y., Wang, C., Keys, C. E., Payne, J., Cooper, T., Luong, K., Song, Y., Chin, C. S., Korlach, J., Roberts, R. J., Evans, P., Musser, S. M., and Brown, E. W. (2013). Fully assembled genome sequence for Salmonella enterica subsp. enterica Serovar Javiana CFSAN001992. *Genome Announc*, **1**(2), e0008113.

[2] Alleman, M., Sidorenko, L., McGinnis, K., Seshadri, V., Dorweiler, J. E., White, J., Sikkink, K., and Chandler, V. L. (2006). An RNA-dependent RNA polymerase is required for paramutation in maize. *Nature*, **442**, 295–298.

[3] Allison, L. and Dix, T. I. (1986). A bit-string longest-common-subsequence algorithm. *Information Processing Letters*, **23**(5), 305–310.

[4] Arlazarov, V., Dinic, E., Kronrod, M., and Faradzev, I. (1970). On economical construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk.*, **194**(11). (in Russian), English Translation in Soviet Math Dokl.

[5] Baeza-Yates, R. and Gonnet, G. H. (1992). A new approach to text searching. *Commun. ACM*, **35**(10), 74–82.

[6] Benson, G. (1997). Sequence Alignment with Tandem Duplication. *J. Computational Biology*, **4**, 351–367.

[7] Benson, G. (1999). Tandem repeats finder: a program to analyze DNA sequences. *Nucleic Acids Research*, **27**, 573–580. (doi:10.1093/nar/27.2.573).

[8] Benson, G., Hernandez, Y., and Loving, J. (2013). A bit-parallel, general integer-scoring sequence alignment algorithm. In J. Fischer and P. Sanders, editors, *Combinatorial Pattern Matching*, volume 7922 of *Lecture Notes in Computer Science*, pages 50–61. Springer Berlin Heidelberg.

[9] Bergeron, A. and Hamel, S. (2002). Vector algorithms for approximate string matching. *Int. J. Found. Comput. Sci.*, **13**(1), 53–66.

[10] Blelloch, G. E. (1990). *Vector models for data-parallel computing*, volume 356. MIT press Cambridge.

[11] Campuzano, V., Montermini, L., Molto, M., Pianese, L., and Cossee, M. (1996). Friedreich's Ataxia: Autosomal Recessive Disease Caused by an Intronic GAA Triplet Repeat Expansion. *Science*, **271**, 1423–1427.

[12] Clarke, H., Flint, J., Attwood, A., and Munafo, M. (2010). Association of the 5-httlpr genotype and unipolar depression: a meta-analysis. *Psychol. Med.*, **40**, 1767–1778.

[13] Consortium, . G. P. (2010). A map of human genome variation from population-scale sequencing. *Nature*, **467**(7319), 1061–1073. (doi:10.1038/nature09534).

[14] Crochemore, M., Iliopoulos, C., Pinzon, Y., and Reid, J. (2001). A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, **80**(6), 279–285.

[15] Daily, J. (2015a). Parasail github repository. https://github.com/jeffdaily/parasail.

[16] Daily, J. (2015b). *Scalable Parallel Methods for Analyzing Metagenomic Data at Extreme Scale*. Ph.D. thesis, Washington State University.

[17] Dayhoff, M., Schwartz, R., and Orcutt, B. (1978). A model of evolutionary change in proteins. In M. Dayhoff, editor, *Atlas of protein sequence and structure*, vol. 5 suppl. 3, pages 345–352. Natl. Biomed. Res. Found.

[18] Farrar, M. (2007). Striped smith–waterman speeds database searches six times over other simd implementations. *Bioinformatics*, **23**(2), 156–161.

[19] Fischetti, V., Landau, G., Schmidt, J., and Sellers, P. (1992). Identifying periodic occurrences of a template with applications to a protein structure. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Proc. 3rd annual Symp. on Combinatorial Pattern Matching, Lecture Notes in Computer Science*, volume 644, pages 111–120. Springer-Verlag.

[20] Frothingham, R. and Meeker-O'Connell, W. A. (1998). Genetic diversity in the mycobacterium tuberculosis complex based on variable numbers of tandem dna repeats. *Microbiology*, **144**(5), 1189–1196.

[21] Fu, Y.-H., Pizzuti, A., Fenwick Jr., R., King, J., Rajnarayan, S., Dunne, P., Dubel, J., Nasser, G., Ashizawa, T., DeJong, P., Wieringa, B., Korneluk, R., Perryman, M., Epstein, H., and Caskey, C. (1992). An Unstable Triplet Repeat in a Gene Related to Myotonic Muscular Dystrophy. *Science*, **255**, 1256–1258.

[22] Gascoyne-Binzi, D., Barlow, R., Frothingham, R., Robinson, G., Collyns, T., Gelletlie, R., and Hawkey, P. (2001). Rapid identification of laboratory contamination with *Mycobacterium tuberculosis* using variable number tandem repeat analysis. *J Clin Microbiol*, **39**, 69–74.

[23] Gelfand, Y., Loving, J., Hernandez, Y., and Benson, G. (2013). VNTRseek – A Computational Pipeline to Detect Tandem Repeat Variants in Next-Generation Sequencing Data: Analysis of the 454 Watson Genome. In *RECOMB-seq 2013: The Third Annual RECOMB Satellite Workshop on Massively Parallel Sequencing, Book of Abstracts*, page 31.

[24] Henikoff, S. and Henikoff, J. (1991). Automated assembly of protein blocks for database searching. *Nucleic Acids Res.*, **19**, 6565–6572.

[25] Henikoff, S. and Henikoff, J. (1992). Amino Acid Substitution Matrices from Protein Blocks. *PNAS*, **89**, 10915–10919.

[26] Huntington's disease collaborative research group (1993). A novel gene containing a trinucleotide repeat that is expanded and unstable on Huntington's disease chromosomes. *Cell*, **72**, 971–983.

[27] Hyyrö, H. (2004a). A note on bit-parallel alignment computation. In *Proc. Prague Stringology Conference 2004 (PSC2004)*.

[28] Hyyrö, H. (2004b). Bit-parallel LCS-length computation revisited. In *Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA 2004)*, pages 16–27.

[29] Hyyrö, H. and Navarro, G. (2002). Faster Bit-Parallel Approximate String Matching. In A. Apostolico and M. Takeda, editors, *Combinatorial Pattern Matching*, volume 2373 of *Lecture Notes in Computer Science*, pages 119–120. Springer Berlin / Heidelberg.

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

[31] Hyyrö, H. and Navarro, G. (2006). Bit-parallel computation of local similarity score matrices with unitary weights. *Int. J. Found. Comput. Sci.*, **17**(6), 1325–1344.

[32] Hyyrö, H., Fredriksson, K., and Navarro, G. (2005). Increased bit-parallelism for approximate and multiple string matching. *Journal of Experimental Algorithmics*, **10**, 2–6.

[33] Hyyrö, H. and Navarro, G (2005). Bit-Parallel Computation of Local Similarity Score Matrices with Unitary Weights. In *Proc. Prague Stringology Conference '05 (PSC 2005)*.

[34] Jobling, M. A. and Gill, P. (2004). Encoded evidence: Dna in forensic analysis. *Nature Reviews Genetics*, **5**(10), 739–751.

[35] Keim, P., Pearson, T., and Okinaka, R. (2008). Microbial forensics: Dna fingerprinting of bacillus anthracis (anthrax). *Analytical Chemistry*, **80**(13), 4791–4800. PMID: 18609763.

[36] Kernighan, B. and Ritchie, D. (1988). *The C programming language*. Prentice Hall, 2nd edition edition.

[37] Landau, G. and Vishkin, U. (1986). Introducing Efficient Parallelism Into Approximate String Matching. *Proc. of the 18th Ann. ACM Symp. on Theory of Computing*, pages 220–230.

[38] Lander, E. S., Linton, L. M., Birren, B., Nusbaum, C., Zody, M. C., Baldwin, J., Devon, K., Dewar, K., Doyle, M., FitzHugh, W., Funke, R., Gage, D., Harris, K., Heaford, A., Howland, J., Kann, L., Lehoczky, J., LeVine, R., McEwan, P., McKernan, K., Meldrim, J., Mesirov, J. P., Miranda, C., Morris, W., Naylor, J., Raymond, C., Rosetti, M., Santos, R., Sheridan, A., Sougnez, C., Stange-Thomann, Y., Stojanovic, N., Subramanian, A., Wyman, D., Rogers, J., Sulston, J., Ainscough, R., Beck, S., Bentley, D., Burton, J., Clee, C., Carter, N., Coulson, A., Deadman, R., Deloukas, P., Dunham, A., Dunham, I., Durbin, R., French, L., Grafham, D., Gregory, S., Hubbard, T., Humphray, S., Hunt, A., Jones, M., Lloyd, C., McMurray, A., Matthews, L., Mercer, S., Milne, S., Mullikin, J. C., Mungall, A., Plumb, R., Ross, M., Shownkeen, R., Sims, S., Waterston, R. H., Wilson, R. K., Hillier, L. W., McPherson, J. D., Marra, M. A., Mardis, E. R., Fulton, L. A., Chinwalla, A. T., Pepin, K. H., Gish, W. R., Chissoe, S. L., Wendl, M. C., Delehaunty, K. D., Miner, T. L., Delehaunty, A., Kramer, J. B., Cook, L. L., Fulton, R. S., Johnson, D. L., Minx, P. J., Clifton, S. W., Hawkins, T., Branscomb, E., Predki, P., Richardson, P., Wenning, S., Slezak, T., Doggett, N., Cheng, J. F., Olsen, A., Lucas, S., Elkin, C., Uberbacher, E., Frazier, M., Gibbs, R. A., Muzny, D. M., Scherer, S. E., Bouck, J. B., Sodergren, E. J., Worley, K. C., Rives, C. M., Gorrell, J. H., Metzker, M. L., Naylor, S. L., Kucherlapati, R. S., Nelson, D. L., Weinstock, G. M., Sakaki, Y., Fujiyama, A., Hattori, M., Yada, T., Toyoda, A., Itoh, T., Kawagoe, C., Watanabe, H., Totoki, Y., Taylor, T., Weissenbach, J., Heilig, R., Saurin, W., Artiguenave, F., Brottier, P., Bruls, T., Pelletier, E., Robert, C., Wincker, P., Smith, D. R., Doucette-Stamm, L., Rubenfield, M., Weinstock, K., Lee, H. M., Dubois, J., Rosenthal, A., Platzer, M., Nyakatura, G., Taudien, S., Rump, A., Yang, H., Yu, J., Wang, J., Huang, G., Gu, J., Hood, L., Rowen, L., Madan, A., Qin, S., Davis, R. W., Federspiel, N. A., Abola, A. P., Proctor, M. J., Myers, R. M., Schmutz, J., Dickson, M., Grimwood, J., Cox, D. R., Olson, M. V., Kaul, R., Raymond, C., Shimizu, N., Kawasaki, K., Minoshima, S., Evans, G. A., Athanasiou, M., Schultz, R., Roe, B. A., Chen, F., Pan, H., Ramser, J., Lehrach, H., Reinhardt, R., McCombie, W. R., de la Bastide, M., Dedhia, N., Blocker, H., Hornischer, K., Nordsiek, G., Agarwala, R., Aravind, L., Bailey, J. A., Bateman, A., Batzoglou, S., Birney, E., Bork, P., Brown, D. G.,

Burge, C. B., Cerutti, L., Chen, H. C., Church, D., Clamp, M., Copley, R. R., Doerks, T., Eddy, S. R., Eichler, E. E., Furey, T. S., Galagan, J., Gilbert, J. G., Harmon, C., Hayashizaki, Y., Haussler, D., Hermjakob, H., Hokamp, K., Jang, W., Johnson, L. S., Jones, T. A., Kasif, S., Kaspryzk, A., Kennedy, S., Kent, W. J., Kitts, P., Koonin, E. V., Korf, I., Kulp, D., Lancet, D., Lowe, T. M., McLysaght, A., Mikkelsen, T., Moran, J. V., Mulder, N., Pollara, V. J., Ponting, C. P., Schuler, G., Schultz, J., Slater, G., Smit, A. F., Stupka, E., Szustakowki, J., Thierry-Mieg, D., Thierry-Mieg, J., Wagner, L., Wallis, J., Wheeler, R., Williams, A., Wolf, Y. I., Wolfe, K. H., Yang, S. P., Yeh, R. F., Collins, F., Guyer, M. S., Peterson, J., Felsenfeld, A., Wetterstrand, K. A., Patrinos, A., Morgan, M. J., de Jong, P., Catanese, J. J., Osoegawa, K., Shizuya, H., Choi, S., Chen, Y. J., and Szustakowki, J. (2001). Initial sequencing and analysis of the human genome. *Nature*, **409**(6822), 860–921.

[39] Langmead, B. and Salzberg, S. L. (2012). Fast gapped-read alignment with Bowtie 2. *Nat Meth*, **9**(4), 357–359.

[40] Lasky-Su, J. A., Faraone, S. V., Glatt, S. J., and Tsuang, M. T. (2005). Meta-analysis of the association between two polymorphisms in the serotonin transporter gene and affective disorders. *Am. J. Med. Genet. B Neuropsychiatr. Genet.*, **133B**, 110–115.

[41] Lesch, K. P., Bengel, D., Heils, A., Sabol, S. Z., Greenberg, B. D., Petri, S., Benjamin, J., Muller, C. R., Hamer, D. H., and Murphy, D. L. (1996). Association of anxiety-related traits with a polymorphism in the serotonin transporter gene regulatory region. *Science*, **274**, 1527–1531.

[42] Li, H. and Durbin, R. (2009). Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, **25**(14), 1754–1760.

[43] Li, R., Yu, C., Li, Y., Lam, T.-W. W., Yiu, S.-M. M., Kristiansen, K., and Wang, J. (2009). SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, **25**(15), 1966–1967.

[44] Lindstedt, B.-A. (2005). Multiple-locus variable number tandem repeats analysis for genetic fingerprinting of pathogenic bacteria. *Electrophoresis*, **26**(13), 2567–2582.

[45] Loving, J., Hernandez, Y., and Benson, G. (2014). Bitpal: A bit-parallel, general integer-scoring sequence alignment algorithm. *Bioinformatics*, **30**(22), 3166–3173.

[46] Miller, N. A., Farrow, E. G., Gibson, M., Willig, L. K., Twist, G., Yoo, B., Marrs, T., Corder, S., Krivohlavek, L., Walter, A., Petrikin, J. E., Saunders, C. J., Thiffault, I., Soden, S. E., Smith, L. D., Dinwiddie, D. L., Herd, S., Cakici, J. A., Catreux, S., Ruehle, M., and Kingsmore, S. F. (2015). A 26-hour system of highly sensitive

whole genome sequencing for emergency management of genetic diseases. *Genome Medicine*, **7**(1), 100.

[47] Miller, W. and Myers, E. (1989). Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, **51**, 5–37.

[48] Miranda, R. and Ayala-Rincon, M. (2005). A modification of the landau- vishkin algorithm computing longest common extensions via suffix arrays. In *Proc. of BSB 2005*, volume 3594 of *Lecture Notes in Computer Science*, pages 1611–3349. Springer.

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

[50] Navarro, G. (2004). Approximate regular expression searching with arbitrary integer weights. *Nordic Journal of Computing*, **11**(4), 356–373.

[51] Needleman, S. and Wunsch, C. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, **48**, 443–453.

[52] Pritchard, A. L., Pritchard, C. W., Bentham, P., and Lendon, C. L. (2007). Role of serotonin transporter polymorphisms in the behavioural and psychological symptoms in probable Alzheimer disease patients. *Dement Geriatr Cogn Disord*, **24**, 201–206.

[53] Smith, T. and Waterman, M. (1981a). Comparison of Biosequences. *Advances in Applied Mathematics*, **2**, 482–489.

[54] Smith, T. and Waterman, M. (1981b). Identification of common molecular subsequences. *Journal of Molecular Biology*, **147**(1), 195–197.

[55] Stam, M., Belele, C., Dorweiler, J. E., and Chandler, V. L. (2002). Differential chromatin structure within a tandem array 100 kb upstream of the maize *b1* locus is associated with paramutation. *Genes Dev.*, **16**, 1906–1918.

[56] Teixeira, F. K. and Colot, V. (2010). Repeat elements and the *Arabidopsis* DNA methylation landscape. *Heredity*, **105**, 14–23.

[57] Van Belkum, A. (2007). Tracing isolates of bacterial species by multilocus variable number of tandem repeat analysis (mlva). *FEMS Immunology & Medical Microbiology*, **49**(1), 22–27.

[58] Venter, J. C., Adams, M. D., Myers, E. W., Li, P. W., Mural, R. J., Sutton, G. G., Smith, H. O., Yandell, M., Evans, C. A., Holt, R. A., Gocayne, J. D., Amanatides, P., Ballew, R. M., Huson, D. H., Wortman, J. R., Zhang, Q., Kodira,

C. D., Zheng, X. H., Chen, L., Skupski, M., Subramanian, G., Thomas, P. D., Zhang, J., Gabor Miklos, G. L., Nelson, C., Broder, S., Clark, A. G., Nadeau, J., McKusick, V. A., Zinder, N., Levine, A. J., Roberts, R. J., Simon, M., Slayman, C., Hunkapiller, M., Bolanos, R., Delcher, A., Dew, I., Fasulo, D., Flanigan, M., Florea, L., Halpern, A., Hannenhalli, S., Kravitz, S., Levy, S., Mobarry, C., Reinert, K., Remington, K., Abu-Threideh, J., Beasley, E., Biddick, K., Bonazzi, V., Brandon, R., Cargill, M., Chandramouliswaran, I., Charlab, R., Chaturvedi, K., Deng, Z., Di Francesco, V., Dunn, P., Eilbeck, K., Evangelista, C., Gabrielian, A. E., Gan, W., Ge, W., Gong, F., Gu, Z., Guan, P., Heiman, T. J., Higgins, M. E., Ji, R. R., Ke, Z., Ketchum, K. A., Lai, Z., Lei, Y., Li, Z., Li, J., Liang, Y., Lin, X., Lu, F., Merkulov, G. V., Milshina, N., Moore, H. M., Naik, A. K., Narayan, V. A., Neelam, B., Nusskern, D., Rusch, D. B., Salzberg, S., Shao, W., Shue, B., Sun, J., Wang, Z., Wang, A., Wang, X., Wang, J., Wei, M., Wides, R., Xiao, C., Yan, C., Yao, A., Ye, J., Zhan, M., Zhang, W., Zhang, H., Zhao, Q., Zheng, L., Zhong, F., Zhong, W., Zhu, S., Zhao, S., Gilbert, D., Baumhueter, S., Spier, G., Carter, C., Cravchik, A., Woodage, T., Ali, F., An, H., Awe, A., Baldwin, D., Baden, H., Barnstead, M., Barrow, I., Beeson, K., Busam, D., Carver, A., Center, A., Cheng, M. L., Curry, L., Danaher, S., Davenport, L., Desilets, R., Dietz, S., Dodson, K., Doup, L., Ferriera, S., Garg, N., Gluecksmann, A., Hart, B., Haynes, J., Haynes, C., Heiner, C., Hladun, S., Hostin, D., Houck, J., Howland, T., Ibegwam, C., Johnson, J., Kalush, F., Kline, L., Koduru, S., Love, A., Mann, F., May, D., McCawley, S., McIntosh, T., McMullen, I., Moy, M., Moy, L., Murphy, B., Nelson, K., Pfannkoch, C., Pratts, E., Puri, V., Qureshi, H., Reardon, M., Rodriguez, R., Rogers, Y. H., Romblad, D., Ruhfel, B., Scott, R., Sitter, C., Smallwood, M., Stewart, E., Strong, R., Suh, E., Thomas, R., Tint, N. N., Tse, S., Vech, C., Wang, G., Wetter, J., Williams, S., Williams, M., Windsor, S., Winn-Deen, E., Wolfe, K., Zaveri, J., Zaveri, K., Abril, J. F., Guigo, R., Campbell, M. J., Sjolander, K. V., Karlak, B., Kejariwal, A., Mi, H., Lazareva, B., Hatton, T., Narechania, A., Diemer, K., Muruganujan, A., Guo, N., Sato, S., Bafna, V., Istrail, S., Lippert, R., Schwartz, R., Walenz, B., Yooseph, S., Allen, D., Basu, A., Baxendale, J., Blick, L., Caminha, M., Carnes-Stine, J., Caulk, P., Chiang, Y. H., Coyne, M., Dahlke, C., Mays, A., Dombroski, M., Donnelly, M., Ely, D., Esparham, S., Fosler, C., Gire, H., Glanowski, S., Glasser, K., Glodek, A., Gorokhov, M., Graham, K., Gropman, B., Harris, M., Heil, J., Henderson, S., Hoover, J., Jennings, D., Jordan, C., Jordan, J., Kasha, J., Kagan, L., Kraft, C., Levitsky, A., Lewis, M., Liu, X., Lopez, J., Ma, D., Majoros, W., McDaniel, J., Murphy, S., Newman, M., Nguyen, T., Nguyen, N., Nodell, M., Pan, S., Peck, J., Peterson, M., Rowe, W., Sanders, R., Scott, J., Simpson, M., Smith, T., Sprague, A., Stockwell, T., Turner, R., Venter, E., Wang, M., Wen, M., Wu, D., Wu, M., Xia, A., Zandieh, A., and Zhu, X. (2001). The sequence of the human genome. *Science*, **291**(5507), 1304–1351.

[59] Verkerk, A., Pieretti, M., Sutcliffe, J., Fu, Y., Kuhl, D., Pizzuti, A., Reiner, O., Richards, S., Victoria, M., Zhang, F., Eussen, B., van Ommen, G., Blonden, A., Riggins, G., Chastain, J., Kunst, C., Galjaard, H., Caskey, C., Nelson, D., Oostra, B., and Warren, S. (1991). Identification of a gene (FMR-1) containing a CGG repeat coincident with a breakpoint cluster region exhibiting length variation in fragile X syndrome. *Cell*, **65**, 905–914.

[60] Vinces, M. D., Legendre, M., Caldara, M., Hagihara, M., and Verstrepen, K. J. (2009). Unstable tandem repeats in promoters confer transcriptional evolvability. *Science*, **324**, 1213–1216.

[61] Walker, E. L. (1998). Paramutation of the *r1* locus of maize is associated with increased cytosine methylation. *Genetics*, **148**, 1973–1981.

[62] Wu, S. and Manber, U. (1992). Fast text searching: allowing errors. *Communications of the ACM*, **35**(10), 83–91.

[63] Wu, S., Manber, U., and Myers, G. (1996). A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, **15**(1), 50–67.

[64] Yu, Y., Baba, K., E, H., and Murakami, K. (2006). Bit-parallel computation for string alignment. In *Selected Papers from the International Conference of Computational Methods in Sciences and Engineering 2006 (ICCMSE 2006), Lecture Series on Computer and Computational Sciences*, volume 7, pages 589–593. VSP/Brill.

Curriculum Vitae

[redacted]