Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors

Running head: Six-fold speed-up of Smith-Waterman searches

Torbjørn Rognes and Erling Seeberg*

Institute of Medical Microbiology, University of Oslo, The National Hospital, NO-0027 Oslo, Norway

Abstract

Motivation: *Sequence database searching is among the most important and challenging tasks in bioinformatics. The ultimate choice of sequence search algorithm is that of Smith-Waterman. However, because of the computationally demanding nature of this method, heuristic programs or special-purpose hardware alternatives have been developed. Increased speed has been obtained at the cost of reduced sensitivity or very expensive hardware.*

Results: *A fast implementation of the Smith-Waterman sequence alignment algorithm using SIMD (Single-Instruction, Multiple-Data) technology is presented. This implementation is based on the MMX (MultiMedia eXtensions) and SSE (Streaming SIMD Extensions) technology that is embedded in Intel's latest microprocessors. Similar technology exists also in other modern microprocessors. Six-fold speed-up relative to the fastest previously known Smith-Waterman implementation on the same hardware was achieved by an optimised 8-way parallel processing approach. A speed of more than 150 million cell updates per second was obtained on a single Intel Pentium III 500MHz microprocessor. This is probably the fastest implementation of this algorithm on a single general-purpose microprocessor described to date.*

Availability: *Online searches with the software are available at http://dna.uio.no/search/*

Contact: *torbjorn.rognes@labmed.uio.no*

Published in Bioinformatics (2000) 16 (8), 699-706.

*) To whom correspondence should be addressed.

Introduction

The rapidly increasing amounts of genetic sequence information available represent a constant challenge to developers of hardware and software database searching and handling. The size of the GenBank/EMBL/DDBJ nucleotide database is now doubling every 15 months (Benson *et al.*, 2000). The rapid expansion of the genetic sequence information is probably exceeding the growth in computing power available at a constant cost, in spite of the fact that computing resources also have been increasing exponentially for many years. If this trend continues, increasingly longer time or increasingly more expensive computers will be needed to search the entire database.

When looking for sequences in a database similar to a given query sequence, the search programs compute an alignment score for every sequence in the database. This score represents the degree of similarity between the query and database sequence. The score is calculated from the alignment of the two sequences, and is based on a substitution score matrix and a gap penalty function. A dynamic programming algorithm for computing the optimal local alignment score was first described by Smith and Waterman (1981), and later improved by Gotoh (1982) for linear gap penalty functions.

Database searches using the optimal algorithm are unfortunately quite slow on ordinary computers, so many heuristic alternatives have been developed, such as FASTA (Pearson and Lipman, 1988) and BLAST (Altschul *et al.*, 1990; Altschul *et al.*, 1997). These methods have reduced the running time by a factor of up to 40 compared to the best known Smith-Waterman implementation, however, at the expense of sensitivity. Because of the loss of sensitivity, some distantly related sequences may not be detected in a search using the heuristic algorithms.

Due to the demand for both fast and sensitive searches, much effort has been made to produce fast implementations of the Smith-Waterman method. Several special-purpose hardware solutions have been developed with parallel processing capabilities (Hughey, 1996), such as Paracel's GeneMatcher, Compugen's Biocceleratorand TimeLogic's DeCypher. These machines are able to process more than 2 000 million matrix cells per second, and can be expanded to reach much higher speeds. However, such machines are very expensive and cannot readily be exploited by ordinary users.

A more general form of parallel processing capability is available in SIMD (Single-Instruction, Multiple-Data) computers. A SIMD computer is able to perform the same operation on several independent data sources in parallel. With the introduction of the Pentium MMX microprocessor in 1997, Intel made computing with SIMD technology available in a general-purpose microprocessor in the most widely used computer architecture – the industry standard PC. The technology is also available in the Pentium II and has been extended in the Pentium III under the name of SSE (Streaming SIMD Extensions) (Intel, 1999). The MMX/SSE instruction set include arithmetic (add, subtract, multiply, min, max, average, compare), logical (and, or, xor, not) and other instructions (shift, pack, unpack) that may operate on integer or floating point numbers. This technology is primarily designed for speeding up digital signal processing applications like sound, images and video, but seems suitable also for genetic sequence comparisons. Several other microprocessors with SIMD technology are or will be made available in the near future, as shown in table 1 (Dubey, 1998).

The Smith-Waterman algorithm has been implemented for several different SIMD computers. Sturrock and Collins (1993) implemented the Smith-Waterman algorithm for the MasPar family of parallel computers, in a program called MPsrch. This solution achieved a speed of up to 130 million matrix cells per second on a MasPar MP-1 computer with 4096 CPUs and up to 1 500 million matrix cells per second on a MasPar MP-2 with 16384 CPUs.

Alpern *et al.* (1995) presented several ways to speed up the Smith-Waterman algorithm including a parallel implementation utilising microparallelism by dividing the 64-bit wide Z-buffer registers of the Intel Paragon i860 processors into 4 parts. With this approach they could compare the query sequence with four different database sequences simultaneously. They achieved more than a fivefold speed-up over a conventional implementation.

Wozniak (1997) presented a way to implement the Smith-Waterman algorithm using the VIS (Visual Instruction Set) technology of Sun UltraSPARC microprocessors. This implementation reached a speed of over 18 million matrix cells per second on a 167MHz UltraSPARC microprocessor. According to Wozniak (1997), this represents a speed-up of about 2 relative to the same algorithm implemented with integer instructions on the same machine.

Both Alpern *et al.* (1995) and Wozniak (1997) seem to have compared their program to a straightforward implementation of the Smith-Waterman algorithm. However, the SWAT program (Green, 1993) and recent versions of SSEARCH (Pearson, 1991) include a non-parallel variant of the Smith-Waterman algorithm that is about twice as fast as the straightforward implementation. This is probably the best reference for speed comparisons.

In this communication, we present an implementation of the Smith-Waterman algorithm using Intel's MMX/SSE technology. It reaches a speed of more than 150 million cell updates per second on a Pentium III 500 MHz computer. To our knowledge, this is so far the fastest implementation of the Smith-Waterman algorithm on a single-microprocessor general-purpose computer. Relative to SSEARCH, it represents a speed-up of about 6 or 13, with or without the SWAT-optimisations, respectively.

System and methods

The software was written in C++ with inline assembler code and was compiled with the GNU egcs compiler. The computer was running Red Hat Linux 6.1 on a single Intel Pentium III 500 MHz microprocessor with 128MB RAM.

Algorithm and implementation

The Smith-Waterman algorithm

To compute the optimal local alignment score, the dynamic programming algorithm by Smith and Waterman (1981), as enhanced by Gotoh (1982), was used. Given a query sequence *A* of length *m*, a database sequence *B* of length *n*, a substitution score matrix Z, a gap open penalty *q* and a gap extension penalty *r*, the optimal local alignment score *t* can be computed by the following recursion relations:

$$
\begin{aligned}
e_{i,j} &= \max\left\{e_{i,\,j-1},\, h_{i-1,\,j} - q\right\} - r \\
f_{i,j} &= \max\left\{f_{i-1,\,j},\, h_{i,\,j-1} - q\right\} - r \\
h_{i,j} &= \max\left\{h_{i-1,j-1} + Z\left[A[i], B[j]\right], e_{i,j}, f_{i,j}, 0\right\} \\
t &= \max\left\{h_{i,j}\right\}
\end{aligned}
$$

Here, $e_{i,j}$ and $f_{i,j}$ represent the maximum local alignment score involving the first *i* symbols of *A* and the first *j* symbols of *B*, and ending with a gap in sequence *B* or *A*, respectively. The overall maximum local alignment score involving the first *i* symbols of *A* and the first *j*

symbols of *B*, is represented by $h_{i,j}$. The recursions should be calculated with *i* going from 1 to *m* and *j* from 1 to *n*, starting with $e_{i,j} = f_{i,j} = h_{i,j} = 0$ for all $i = 0$ or $j = 0$. The order of computation of the values in the alignment matrix is strict because the value of any cell cannot be computed before the value of all cells to the left and above it has been computed, as shown by the data interdependence graph in figure 1. A straightforward implementation of the algorithm has a running time proportional to *mn*.

The SWAT-optimisations
Green (1993) implemented an improved version of the Smith-Waterman algorithm in the SWAT program and obtained an increase of speed by a factor of about 2. In most cells in the matrix, *e* and *f* are zero, and hence do not contribute to *h*. As long as *h* is not larger than the threshold $q + r$, which is the penalty of a single symbol gap, *e* and *f* will stay at zero along a column or row in the matrix. This can save many computations, and is the basis for the enhancements used to speed up the original algorithm. It should be noted, however, that this is not effective if gap penalties are very small, as many cells will then have a value above the threshold. The SWAT-optimisations are now also implemented in the SSEARCH program (Pearson, 1991) included in Pearson's FASTA package. An alternative version of the program is called OSEARCH and uses a traditional implementation.

Parallelisation
The Smith-Waterman algorithm can be parallelised on two scales. It is fairly easy to distribute the processing of each of the database sequences on a number of independent processors in a symmetric multiprocessing (SMP) machine. On a lower scale, however, distributing the work involved within a single database sequence is a bit more complicated. Figure 1 shows the data interdependence in the alignment matrix. The final value, *h*, of any cell in the matrix cannot be computed before the value of all cells to the left and above it has been computed. But the calculations of the values of diagonally arranged cells parallel to the minor diagonal (see figure 2a) are independent and can be done simultaneously in a parallel implementation. This fact has been utilised in earlier SIMD implementations (Hughey, 1996; Wozniak, 1997).

Our approach
We have implemented the Smith-Waterman algorithm using Intel's MMX/SSE technology. Pseudo-code for our implementation is shown in figure 3. In order to get complete control over code optimisation and because of limited support for the MMX/SSE instructions in high level languages, the core of the algorithm has been written in assembly language. The main features of our implementation are:

- Vectors parallel to the query sequence
- A SWAT-like optimisation
- 8-way parallel processing with 8-bit values
- Query sequence profiles
- General code optimisations

Vectors parallel to the query sequence
Despite the loss of independence between the computation of each of the vector elements, we decided to use vectors of cells parallel to the query sequence (as shown in figure 2b), instead of vectors of cells parallel to the minor diagonal in the matrix (as shown in figure 2a). The advantage of this approach is the much-simplified and faster loading of the vector of substitution scores from memory. The disadvantage is that data dependencies within the

vector must be handled. Eight cells are processed simultaneously along each column as indicated in figure 2b. Each column represent one symbol of the database sequence.

A SWAT-like optimisation

As already indicated, we have to take into account that each element in the vector is dependent on the element above it, because of the possible introduction of gaps in the query sequence. However, as exploited by the SWAT-optimisations, most cell values are not above the $q + r$ threshold. If none of the eight cells in the vector are above that threshold, the *f*-values can simply be ignored in the computation of the *h*-values, thus removing data dependencies and greatly simplifying the computations. It is possible to check simultaneously if any of the eight cells in the vector is above the threshold. In the case that none of the cells are above the threshold, the computation of the *h*-values will be very fast. However, if any of the cell values are above the threshold, it will be necessary to go through a somewhat time-consuming process of computing the correct values for *h*, *e* and *f*.

8-way parallel processing with 8-bit values

The microprocessors provide for the SIMD instructions a set of registers (usually 64-bit wide) that can be divided into smaller units. The Pentium family of microprocessors contains several 64-bit registers that can be treated either as a single 64-bit (quadword) unit, or as two 32-bit (doubleword), four 16-bit (word), or eight 8-bit (byte) units. Operations on these units are independent. Hence, the microprocessor is able to perform up to eight independent additions or other operations simultaneously.

In order to optimise the speed of the calculations, we have chosen to divide the MMX-registers of the microprocessor into as many units as possible, i.e. eight 8-bit units. This allows eight concurrent operations to take place. Dividing the MMX-registers into eight 8-bit registers increases the number of parallel operations but limits the precision of the calculations to the range 0-255. Unless the sequences are long and very similar, this poses no problems. In the few cases where this score limit is surpassed, the use of saturation arithmetic (see below) will ensure that the overall highest score will stay at 255. For all sequences that reach a score of 255, the correct score may subsequently be recomputed by a different implementation with a larger score range (e.g. using a non-SIMD implementation).

Using Intel's MMX/SSE technology, additions and subtractions can be performed in either unsigned or signed mode. In the inner loop of the algorithm, the signed query profile scores are added to the unsigned *h*-values. Using a signed addition, the *h*-values would have been restricted to the range of 0–127. Instead, all the values in the query sequence score profile were biased by a fixed amount (e.g., 4) so that no values were negative. One signed operation was then replaced by an unsigned addition followed by an unsigned subtraction of the bias. The useful data range was hence expanded to nearly 8 bits (e.g., 0–251), at the expense of one additional instruction.

Unsigned arithmetics using the MMX technology can be performed in either a modular (also known as wrap-around) or in a saturated mode. When using 8-bit wide registers, subtracting 25 from 10 will give the result 241 (because 10-25 = 241-256) in modular mode and 0 in saturated mode. This is very useful in the inner loop calculations of the Smith-Waterman algorithm because negative results should be replaced by zero in some of the calculations. Also, because of the limited precision of a single byte value, saturated arithmetics are useful to detect potential overflow in the calculations with very high scores.

The core of the Smith-Waterman algorithm repeatedly computes the maximum of two numbers. It is therefore important to make this computation fast. The SSE instruction set includes a special instruction (pmaxub) that computes the largest of two unsigned bytes. This instruction was not included in the original MMX instruction set, but can be replaced by an

unsigned saturated subtraction (psubusb) followed by an unsigned saturated addition (paddusb).

Query sequence profiles
When the same query sequence is compared to many different database sequences, a simple speed improvement is achieved by creating a kind of score profile for the query sequence. This profile, which can be considered as a query-specific substitution score matrix, is computed only once for the entire search, and will save one memory lookup in the inner loop of the algorithm. Instead of indexing the original substitution score matrix by the query sequence symbol and the database sequence symbol, the new matrix is indexed by the query sequence position and the database sequence symbol. The score for matching symbol A (for alanine) in the database sequence with each of the symbols in the query sequence is stored sequentially in the first matrix row, followed by the scores for matching symbol B (ambiguous) in the next row, and so on. This query sequence profile is used extensively in the inner loop of the algorithm and is usually small enough to be kept in the microprocessor's first level cache.

General code optimisations
The use of conditional jumps should be avoided when it is difficult for the microprocessor to predict whether to jump or not, because mispredictions require additional time. In addition, conditional jumps based on the results of MMX/SSE operations are not straightforward on the Intel architecture because the status flags are not set by these instructions. We have hence tried to avoid conditional jumps as much as possible in the core of the algorithm.

In order to achieve the highest speed, the memory used repeatedly in the calculations should preferably be contained in the first level caches of the microprocessor. In addition to the query sequence score profile, the vectors storing the *h* and *e* values from the last column should also fit in the cache, but these are usually only about 400 bytes each for an average sequence.

The *h* and *e* vectors, each of length *m*, have been grouped into a single vector of length *m* consisting of a structure of the two elements. It is generally faster to access a single vector sequentially than to access two independent vectors sequentially. The 64-bit memory accesses used with MMX registers should preferably be placed on 8 byte boundaries, in order to be as fast as possible. We have taken this into account when aligning the data structures. Code alignment also had substantial effects on the speed.

When the computer is equipped with enough internal memory to hold the entire database, the use of memory mapped files is an effective way to read the database. The entire sequence file can then be mapped to particular address range in memory. Operating systems are usually optimised for reading sequential files in this way.

Results

The speed of the new algorithm was evaluated using a test set of 11 different amino acid query sequences. The length of the query sequences ranged from 189 to 567 amino acids, with 3807 amino acids in total. These sequences represent members of a range of well-characterised protein families. The same test set has previously been used for the evaluation of BLAST 2 (Altschul *et al.*, 1997). The SWISS-PROT (Bairoch and Apweiler, 2000) release 38 protein database containing 80 000 sequences with a total of 29 085 965 amino acid residues was searched.

The new algorithm was compared to the Smith-Waterman implementations SSEARCH and OSEARCH version 3.2t08 (Pearson, 1991). Searches with the heuristic

programs FASTA version 3.2t08 (Pearson and Lipman, 1988), NCBI BLAST version 1.4.9 (Altschul *et al.*, 1990), NCBI BLAST version 2.0.10 (Altschul *et al.*, 1997) and WU-BLAST version 2.0a19 (Gish, 1996) were also included for comparison. All searches were performed using the BLOSUM62 amino acid substitution score matrix (Henikoff and Henikoff, 1992) and with gap open and extension penalties of 10 and 1, respectively. The options of all programs were set to display the top 500 scores and no alignments. FASTA was run using both ktup=1 and ktup=2. WU-BLAST was run with the recommended postsw-option. NCBI BLAST 2.0.10 was run with the option K=500.

For each query sequence, the total CPU time of the fastest of 3 consecutive runs on a minimally loaded computer was recorded. With a database of only 29MB and 128MB of RAM, all of the database was cached in the computer's RAM; disk-reading time should hence be negligible.

Plots of search time versus query sequence length for all programs are shown in figure 4. The total time used for searching all of the query sequences was 9182s for OSEARCH, 4372s for SSEARCH, 796s for FASTA (ktup=1), 708s for SW-MMX, 267s for WU-BLAST, 228s for FASTA (ktup=2), 213s for NCBI BLAST 1.4.9 and 94s for NCBI BLAST 2.0.10.

Among the implementations of the Smith-Waterman algorithm, our implementation was found to be 13 times faster than OSEARCH and 6 times faster than SSEARCH. Our implementation was also slightly faster than FASTA with ktup=1. FASTA with ktup=2, NCBI BLAST 1.4.9 and WU-BLAST were all only approximately 3 times faster than ours, while BLAST 2.0.10 was about 7.5 times faster.

The algorithm performed equally well on longer and shorter sequences. The average speed was 156 million matrix cell updates per second.
The final raw scores computed by our implementation are equal to those computed by a straightforward Smith-Waterman implementation. Hence, the sensitivity and ranking of matching sequences should be equal to other Smith-Waterman programs, unless the choice of score matrix, gap penalties or the function for calculating statistical significance or expectation (Z- or E-value) is different.

Discussion

Due to the speed achieved by the presented algorithm and the low cost of Intel Pentium III-based computers, we believe it is now the most cost-effective way to perform database searches using the Smith-Waterman algorithm. A symmetric multiprocessing (SMP) computer with 8 Pentium III Xeon CPUs at 600MHz should be able to achieve a speed of about 1500 million cell updates per second. A large cluster of inexpensive computers would be a more cost-effective solution, and may reach even higher speeds.

The SIMD technology will most likely evolve further in the coming microprocessor generations. Implementations of SIMD technology in future microprocessors will probably allow even faster variations of the Smith-Waterman algorithm. The Motorola AltiVec (a.k.a. Velocity Engine) technology (Motorola, 1998), which has just been introduced in the PowerPC G4 microprocessors, includes 128-bit wide registers that can be divided into sixteen 8-bit units. A new generation of microprocessors from Intel called Willamette will also include 128-bit wide registers for SIMD processing. It would be of great interest to evaluate implementations of the presented algorithm on these processors. However, it may be even more interesting and rewarding to exploit the SIMD technology for entirely new algorithms.

For the initially highest scoring sequences in the database, FASTA and BLAST 2 computes an optimal alignment restricted to either a band or a region surrounding the most interesting part of the alignment matrix. We believe that our approach could easily be extended to alignments restricted to a band that is, preferably, a multiple of 8 cells wide. However, our approach will probably be less effective on alignments restricted to irregular

regions, as employed by BLAST 2, because much of the power of the technology lies in the repetition of simple operations.

Even the Smith-Waterman alignment algorithm is unable to identify all protein similarities based on the primary sequence alone. In addition to better algorithms, improvements in the substitution score matrix, gap penalising and the scoring system in general are also required for an optimisation of the overall sensitivity.

Acknowledgements

This work was supported by grants from the Research Council of Norway and the Norwegian Cancer Society.

References

Alpern, B., Carter, L. and Gatlin, K.S.(1995) Microparallelism and High Performance Protein Matching. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference: San Diego, California, Dec 3-8, 1995.* http://www.supercomp.org/sc95/proceedings/549_LCAR/SC95.HTM

Altschul, S.F., Gish, W., Miller, W., Myers, E.W. and Lipman, D.J. (1990) Basic local alignment search tool. *J. Mol. Biol.*, **215**, 403-410.

Altschul, S.F., Madden, T.L., Schaffer, A.A., Zhang, J., Zhang, Z., Miller, W. and Lipman, D.J. (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic. Acids. Res.*, **25**, 3389-3402.

Bairoch, A. and Apweiler, R. (2000) The SWISS-PROT protein sequence database and its supplement TrEMBL in 2000. *Nucleic. Acids. Res.*, **28**, 45-48.

Benson D.A., Karsch-Mizrachi I., Lipman D.J., Ostell J., Rapp B.A., and Wheeler D.L. (2000) GenBank. *Nucleic. Acids. Res.* **28,** 15-18.

Dubey, P. K. (1998) Architectural and design implications of mediaprocessing. http://www.research.ibm.com/people/p/pradeep/tutor.html

Gish, W. (1996) WU-BLAST. http://blast.wustl.edu/

Gotoh, O. (1982) An improved algorithm for matching biological sequences. *J. Mol. Biol.*, **162**, 705-708.

Green, P. (1993) SWAT. http://www.genome.washington.edu/uwgc/analysistools/swat.htm

Henikoff, S. and Henikoff, J.G. (1992) Amino acid substituition matrices from protein blocks. *Proc. Natl. Acad. Sci U.S.A.*, **89**, 10915-10919.

Hughey, R. (1996) Parallel hardware for sequence comparison and alignment. *Comput. Applic. Biosci.*, **12**, 473-479.

Intel (1999) Intel Architecture Software Developer's manual; Volume 2: Instruction Set Reference. http://developer.intel.com/design/pentiumii/manuals/243191.htm

Motorola (1998) AltiVec Technology Programming Environments Manual. http://www.mot.com/SPS/PowerPC/teksupport/teklibrary/manuals/altivec_pem.pdf

Pearson, W.R. (1991) Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms. *Genomics*, **11**, 635-650.

Pearson, W.R. and Lipman, D.J. (1988) Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci U. S. A.*, **85**, 2444-2448.

Smith, T.F. and Waterman, M.S. (1981) Identification of common molecular subsequences. *J. Mol. Biol.*, **147**, 195-197.

Sturrock S.S. and Collins J.F. (1993) MPsrch V1.3 User Guide. Biocomputing Research Unit, University of Edinburgh, UK.

Wozniak, A. (1997) Using video-oriented instructions to speed up sequence comparison. *Comput. Appl. Biosci.*, **13**, 145-150.

Tables

Table 1 – Microprocessors with SIMD technology

Manufacturer	Microprocessor	Name of technology
AMD	K6/K6-2/K6-III	MMX / 3DNow!
	Athlon	Extended MMX / 3DNow!
Chromatics	MPact	
Compaq (Digital)	Alpha	MVI (Motion Video Instruction)
HP	PA-RISC	MAX(-2) (Multimedia Acceleration eXtensions)
HP / Intel	Itanium (Merced)	SSE ?
Intel	Pentium MMX / II	MMX (MultiMedia eXtensions)
	Pentium III	SSE (Streaming SIMD Extensions)
MicroUnity	MediaProcessor	
Motorola	PowerPC G4	Velocity Engine (AltiVec)
Philips	TriMedia	
SGI	MIPS	MDMX (MIPS Digital Media eXtensions)
Sun	SPARC	VIS (Visual Instruction Set)

Figure legends

Figure 1

Computational dependencies in the Smith-Waterman alignment matrix.

Figure 2

Vector arrangements in SIMD implementations of the Smith-Waterman algorithm.

a) Traditional approach with vectors parallel to the minor diagonal
b) New approach with vectors parallel to the query sequence

Figure 3

Pseudocode for the new approach.

The MAX operation returns a vector with the pairwise maximum of the elements of the two arguments. The LSHIFT and RSHIFT operations shifts the elements of a vector the specified number of times to the left or right. The OR operation returns the bitwise or of the vector elements. All vector subtractions and additions are saturated and unsigned. The query sequence is assumed to be padded to a multiple of 8 bytes.

Figure 4

Plots with a comparison of the search time versus query length for (a) Smith-Waterman implementations. and (b) heuristic search algorithms. Our implementation is included in both plots for reference. The query sequences have accession numbers P00762, P01008, P01111, P02232, P03435, P05013, P07327, P10318, P10635, P14942 and P25705 in SWISS-PROT.

Figures

Figure 1

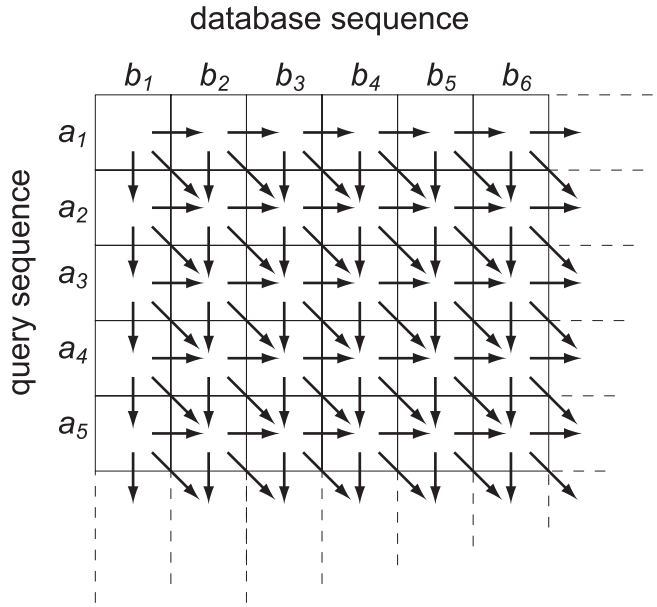
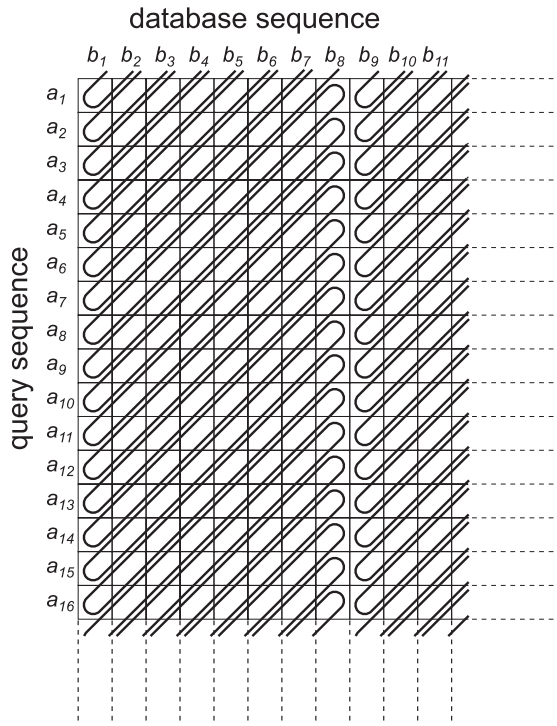


Figure 2

a)



b)

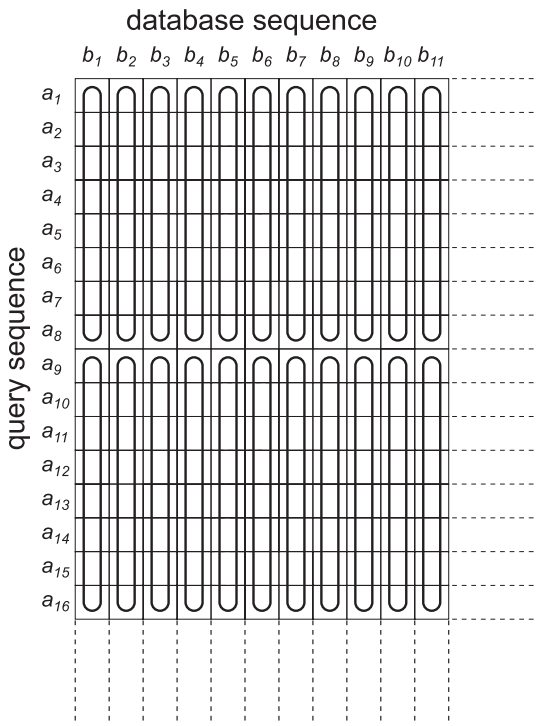
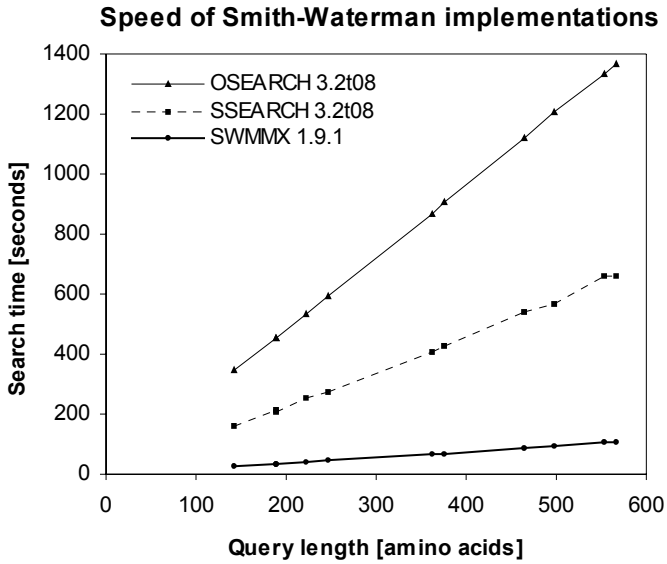


Figure 3

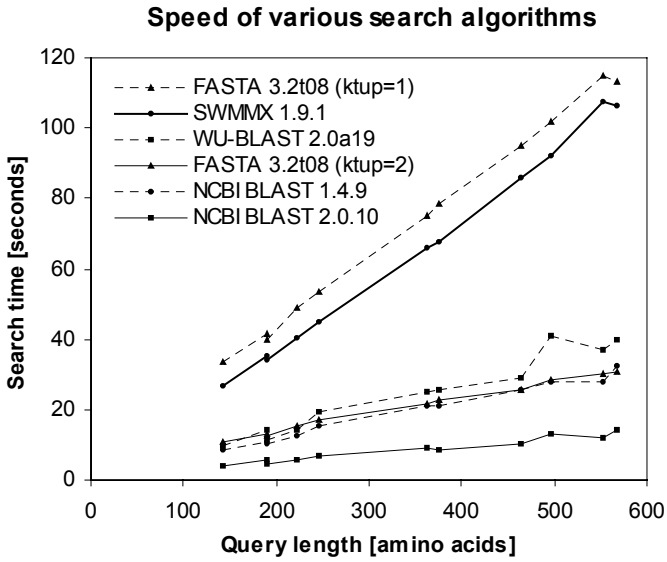
Pseudocode	Comments
`FUNCTION SWMMX(MM, DSEQ, q, r, m, n)`	MM is a query-specific score matrix DSEQ is the database sequence q and r are gap open and extension penalties m and n are query and database sequence lengths
`CHAR c`	One database sequence symbol (c)
`INTEGER i,j`	Loop indices (i,j)
`CONST INTEGER y = m/8`	Number of vectors along query sequence (y)
`VECTOR8 H,X,E,F,T1,T2,SCORE,HH[y],EE[y]`	Vectors (H,X,E,F,T1,T2) and arrays (HH,EE)
`CONST VECTOR8 BASE = [4, 4, 4, 4, 4, 4, 4, 4]`	Score base vector (constant)
`CONST VECTOR8 QQ = [q, q, q, q, q, q, q, q]`	Gap open penalty vector (constant)
`CONST VECTOR8 RR = [r, r, r, r, r, r, r, r]`	Gap extension penalty vector (constant)
`FOR i=0 TO y-1 DO` `{`	
`HH[i] = [0, 0, 0, 0, 0, 0, 0, 0]`	Initialise HH-array of H-values from previous column
`EE[i] = [0, 0, 0, 0, 0, 0, 0, 0]`	Initialise EE-array of E-values from previous column
`}`	
`SCORE = [0, 0, 0, 0, 0, 0, 0, 0]`	Initialise score vector
`FOR j = 0 TO n-1 DO`	For each symbol in the database sequence...
`{`	
`X = [0, 0, 0, 0, 0, 0, 0, 0]`	Initialise X-vector for 1. round
`F = [0, 0, 0, 0, 0, 0, 0, 0]`	Initialise F-vector for 1. round
`c = DSEQ[j]`	Get one database symbol
`FOR i = 0 TO y-1 DO`	For each vector of 8 matrix cells along query sequence...
`{`	
`H = HH[i]`	Load previous H-vector from HH-array
`E = EE[i]`	Load previous E-vector from EE-array
`T1 = (H RSHIFT 7)`	Save previous H[7] for use below
`H = (H LSHIFT 1) OR X`	Shift H-vector and OR with H[7] from previous round
`X = T1`	Save old H[7] in X for next round
`H = ( H + MM[c][i] ) - BASE`	Add score profile vector to H and subtract base
`H = MAX(H , E)`	Check if score with database gap is better
`F = (H LSHIFT 1) OR (F RSHIFT 7)`	Calculate initial F-vector by shifting H and previous F
`F = F - QQ - RR`	Subtract single gap penalty
`IF (any element of F > 0)`	Check if vertical gaps are possible
`{`	Compute correct F-vector if necessary
`T2 = F`	T2 is initial F-vector
`WHILE (any element of T2 > 0)`	Repeat while any element of T2 is nonzero...
`{`	
`T2 = (T2 LSHIFT 1) - RR`	Shift and subtract gap extension penalty
`F = MAX(F , T2)`	Update F if new score is higher
`}`	
`H = MAX(H , F)`	Update H if vertical gap is better
`F = MAX(H , F + QQ)`	Update F for use in next round
`}`	
`ELSE`	
`{`	
`F = H`	Update F for use in next round
`}`	
`HH[i] = H`	Store H-vector in HH-array
`EE[i] = MAX(H - QQ, E) - RR`	Store E-vector in EE-array
`SCORE = MAX(SCORE, H)`	Update Score with new H-vector if it is better
`}`	
`}`	
`RETURN MAX(SCORE[0], SCORE[1], ..., SCORE[7])`	Return largest element in score vector

Figure 4

a)


Speed of Smith-Waterman implementations
OSEARCH 3.2t08
SSEARCH 3.2t08
SWMMX 1.9.1
Search time [seconds]
Query length [amino acids]
0
200
400
600
800
1000
1200
1400
0
100
200
300
400
500
600

b)


Speed of various search algorithms
FASTA 3.2t08 (ktup=1)
SWMMX 1.9.1
WU-BLAST 2.0a19
FASTA 3.2t08 (ktup=2)
NCBI BLAST 1.4.9
NCBI BLAST 2.0.10
Search time [seconds]
Query length [amino acids]
0
20
40
60
80
100
120
0
100
200
300
400
500
600