HARDALIGN: A PARALLEL PAIRWISE ALIGNMENT HARDWARE

Guilherme L. Moritz, Cristiano Jory, Heitor S. Lopes, Carlos R. Erig Lima

Bioinformatics Laboratory (CPGEI)
Federal Technological University of Paraná
Av. 7 de setembro, 3165 80230-901 Curitiba (PR) – Brazil
email: hslopes@pesquisador.cnpq.br, erig@cefetpr.br

ABSTRACT

This paper describes the design and implementation of a parallel pairwise alignment hardware, implemented in a FPGA device. This system is aimed at aligning pairs of proteins, using a dynamic programming algorithm. The alignment is done in parallel thanks to a pipelined approach. All functional blocks are described in detail. Experiments were done to access the performance of the system for up to pairs of 2000-amino acids-long proteins. Hardalign was compared with a similar algorithm implemented in software and running in a PC, resulting in a 1:10 speed-up ratio. Results encourage the continuity of the work, showing that reconfigurable computing can offer interesting solutions for bioinformatics problems.

1. INTRODUCTION

In biological systems, proteins are the most abundant and functionally diverse molecules and almost all vital processes depend on these macromolecules, which are composed by amino acids chains. The common 20 different types of amino acids can be combined in a linear sequence having the necessary information for the generation of a unique tri-dimensional structure. The comparison of two protein sequences (or a group of them) is known as alignment. Alignment consists of the systematic comparison of the amino acids compounding the sequences throughout their whole extension (or only specific regions), and then generating a string that shows similar and dissimilar regions, as well as a similarity score is computed. Sequence alignment is the most important tool for discovering and representing similarities between sequences, and can unravel the evolutionary history, critical preserved motifs, details of the tertiary structure or important clues about protein function. Therefore, sequence alignment is a central topic of extensive research in computational biology [4].

However, from the computational viewpoint, the alignment of sequences (either proteins or DNA) is a very difficult task. In recent literature, many computational algorithms were proposed for sequence alignment (local/global alignment; pairwise/multiple alignment). The main difference between them is the quality of the alignment and the computational effort required. There is an exact algorithm for finding the optimal alignment between two sequences, based on dynamic programming. This algorithm, originally proposed by Needleman and Wunsch [6], was aimed at finding a similarity score between sequences, and was later explored by Waterman [7]. The drawback of this algorithm is the memory and processing time required. Given two sequences of length N, the memory complexity and time complexity of the algorithm are $O(N^2)$ and $O(N^3)$, respectively. As a consequence, pairwise alignment using dynamic programming is a problem that requires significant computational power, especially for large sequences.

On the other hand, recently, we have witnessed a pronounced growth of the hardware and software technologies for embedded systems, with many technological options arising every year. The use of open and reconfigurable structures is becoming attractive, especially due to its robustness and flexibility for easy adaptation to different project requirements. Reconfigurable devices have advantages such as: low power consumption and high speed processing, efficient tools for simulation and programming, flexibility and modular operation [2]. In special, the possibility of massive parallel processing makes reconfigurable computing (that is, systems based on reconfigurable hardware) a suitable technology to be applied to the pairwise alignment problem addressed here.

This work describes the project and implementation of a parallel pairwise alignment algorithm using reconfigurable hardware computing.

1.1 Dynamic Programming

The dynamic programming algorithm is used for aligning two amino acids sequences and computing its similarity score. This algorithm requests the construction of a (m x n) matrix, where m and n are the lengths of the two sequences to be aligned. For the construction of the matrix, the string of the first sequence is put above the matrix and the other on the left side. The first line and column of the matrix

This work was supported by the Brazilian National Research Council (CNPq) under grants no. 506479/04-8 and 305720/04-0.

depend only of the sequence above and to the left, respectively. All other cells of the matrix have to be computed recursively, depending on the values of the upper, left and diagonal cells. This data dependency imposes a serious constraint on the algorithm, not allowing its direct parallelizing [5]. The computation of the elements of the matrix takes into account a substitution matrix, which gives the "evolutionary distance" between pairs of amino acids [3]. In this work we used BLOSUM62 as the evolutionary distance matrix.

Equation (1) shows the Needleman-Wunsch approach for computing the elements of the dynamic programming matrix. Line 0 and column 0 are the preliminary elements needed for computing the matrix, as mentioned before, and they are obtained with the three first terms of the equation. In this equation, i and *j* are the indexes of the (*m* x *n*) matrix, *g* is the gap penalty, *A*[*i*,*j*] is the cell at coordinates i and *j*, and $S[aa_i,aa_j]$ is the value of the substitution matrix for the corresponding pair of amino acids that are in line *i* and column *j* of the matrix.

$$\begin{cases} A[0,0] = 0 \\ A[i,0] = -g.m \qquad 1 < i < m \\ A[0,j] = -g.n \qquad 1 < j < n \\ A[i,j] = \max\begin{pmatrix} A[i-1,j-1] + S[aa_i, aa_j]; \\ A[i-1,j] - g; \\ A[i,j-1] - g \end{pmatrix} \end{cases} \quad \textbf{(1)}$$

2. HARDALIGN

Fig. 1 shows a block diagram of the proposed system. HardAlign is a dedicated processing system, working as a peripheral of the NIOS II Altera embedded processor, interconnected via the Avalon bus [1]. Hardalign was especially developed for pairwise alignment of proteins.

The Hardalign block contains an arrangement of *N* matrix line processor units (MLPUs – see section 2.3) and all logic for driving the arrangement. The size of *N* is limited only by the width of the Avalon bus.

The process of pairwise alignment using dynamic programming matrix is divided in steps, as follows:

a) Reception of amino acids data.
b) Reception of system's configurations.
c) Computation of the dynamic programming matrix.
d) Backtracking in the dynamic programming matrix.
e) Transmission of results.

The critical part of the process that requires computational power is the computation of the dynamic programming matrix and for this reason it is run in parallel by the MLPUs. For this reason the computation of the dynamic programming matrix is emphasized in this work.

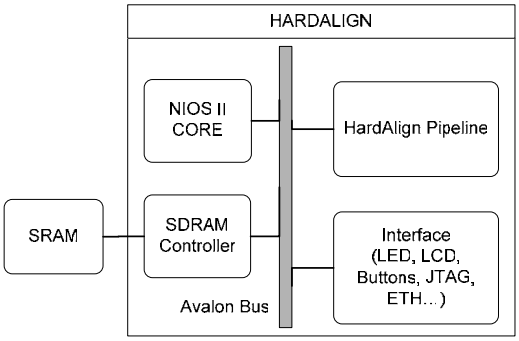

Fig. 1. Architecture of the Hardalign system.

The software running in the NIOS II processor performs steps (a) and (b) and writes data to internal registers of Hardalign. Then, the driving logic is started and NIOS II reads sequentially the results. A SDRAM memory is used for storing vectors generated by the MLPUs. Next, the backtracking procedure is started by software running in NIOS II. This algorithm uses the vectors previously generated and finds a path from the last cell of the matrix towards the first one. This sequence corresponds to the alignment of the two proteins and it is the result sent to the user, ending the process. In the next sections, the Hardalign Pipeline will be described in details.

2.1 Cell computation circuit

Each MLPU computes one cell of the dynamic programming matrix, in a single clock cycle, and it is responsible for computing a line of the matrix. Once completed, a new line is started until the matrix is fully done. Since *N* cells are computed by clock cycle, the whole matrix is concluded in *L.C/N* clock cycles, where *L* is the number of lines and *C* the number of columns of the matrix.

Fig. 2 shows the basic cell computation circuit used by each MLPU. It is responsible for computing the value of a cell, and uses the information of the adjacent cells, a substitution matrix and the Needleman-Wunsch equation [6] (section 1.1). The outputs of the circuit are the value of the cell and the information from where the cell was constructed.

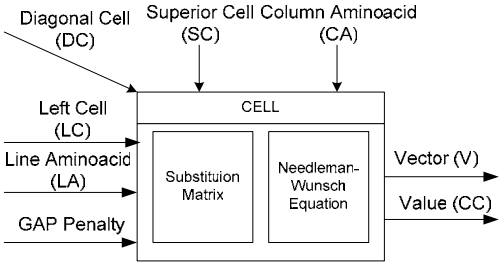

Fig. 2. Internal architecture of a cell computation circuit.

The substitution matrix is encoded as a combinatorial circuit: every possible combination of the 20 amino acids gives an evolutionary distance value as result.

This value is used in the equation to compute the final value of the cell. The circuit has a special behavior when one or both of their input amino acids correspond to a gap. This occurs in the first line and first column of the dynamic programming matrix, and known as border penalties.

Fig. 3 shows in details how a specific cell is computed by the circuit of Fig. 2. The arrows point back the cell from which a given cell was generated from. Numbers inside de cells are the result of applying equation (1), letters above and to the left are the two amino acids to be aligned. The current cell under computation is the bottom-right one, and all elements needed for computing it are shown.

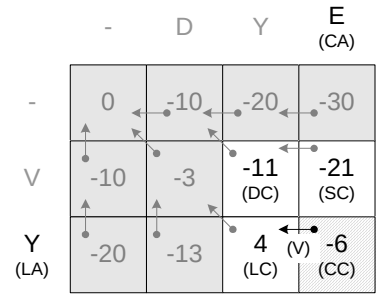

Fig. 3. Example of the of a cell computation of the dynamic programming matrix, using the cell computation circuit.

2.2 Arrangement of the Cell Computation Circuits

The first approach for the cell computation circuit arrangement is to set up them exactly as a dynamic programming matrix. However, this architecture has several disadvantages, as follows:

- Wasting of logical resources: A cell would compute a single value, remaining idle for all the rest of the process.
- Input and output bottleneck: The final score will be valid only when all of the amino acids are applied simultaneously to the cells inputs, what would be longer than the propagation time of the cells, thus characterizing a bottleneck. This limitation also appears when the output vectors are read simultaneously.
- Maximum alignment size: The maximum length of sequences to be alignment is limited by the number of cells configured in the FPGA device. Since part of the device must be reserved to other systems blocks, and the size of the matrix grows quadratically, this implementation would be useful only for short alignments.

The construction of a pipeline for computing the dynamic programming matrix is the alternative due to the limitations of the first approach. In this case, an example of cells arrangement is shown in Fig. 4.

	a	**b**	**c**	**d**
a	0	1	2	3
b	1	2	3	4
c	2	3	4	5
d	3	4	5	6

Fig. 4. Cell computation sequence for the dynamic programming matrix using a pipeline structure.

In a given clock cycle, each cell presents both V and CC (see Fig. 2) in its outputs. Those values are used for the recursive computation of the following cells in the next clock cycle. In the first clock, only the cell (*a*,*a*) can be computed. Using the value of this cell, in the next clock, it is possible to compute cells (*a*,*b*) and (*b*,*a*) at the same time. With these values, in the next clock, cells (*a*,*c*), (*b*,*b*) and (*c*,*a*) can be computed, and so on. Once computed cell (*a*,*a*), the remaining elements of the first line can be computed. In the same way, once computed cell (*b*,*a*), the second line can be computed. Therefore, as the clock cycles advance, more lines are enabled to be computed. This feature inspired the pipeline approach exploited in this work.

2.3 The MLPU entity

Except for the first line of the matrix (that is a special case), the other lines depend on the following values for the computing a cell:

a) Value of the line amino acid (constant during the computation of a line);
b) Value of the column amino acid (changes in each cycle);
c) Value of the previous cell, computed by the own entity.
d) Values of the cells of the line immediately above the one that it is computed, of the current and the previous coordinate.
e) Value of the gap penalty.

MLPU entity can be replicated to compute simultaneously several lines in parallel. All the necessary data for an entity can be read from other entities that compose the pipeline, as follows:

a) The line amino acids are constant during computation of a line of the matrix. Then, they are read in the beginning of the process.

b) The pipeline used for the first line of alignment should receive the column amino acids sequentially. Fig. 5 shows the MLPU entity in details, where an amino acid requested by a line is always requested by the pipeline that was above in the previous clock cycle.
c) The previous computed value is registered to be reused for the subsequent cells computation.
d) Besides the value immediately subsequent, the values computed in the two subsequent clock cycles are also registered. The value of these cells is also necessary for the computation of the line immediately below.
e) The gap penalty is common to all of the entities and constant during the whole process. Therefore, it is registered outside of MLPU.

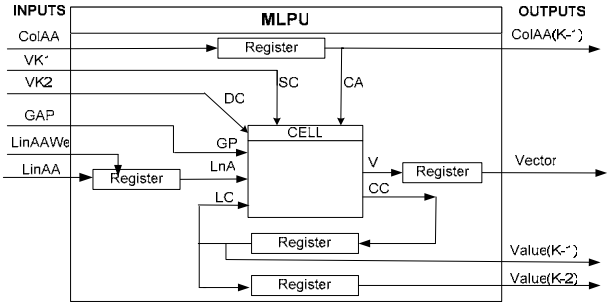

Fig. 5. Details of the MLPU entity.

2.4 Basic structures of the Pipeline entity

The Pipeline entity, shown in Fig. 6, is arranged to compute several lines in a parallel way. In this figure, an arrangement with 3 MLPUs is shown, therefore allowing the simultaneous computation to 3 lines of the dynamic programming matrix. The inputs of this entity are the gap penalty, the line amino acids and the column amino acids, in a sequential way.

Each MLPU entity has four outputs: two cell values, an amino acid value and a vector. Only the vector is necessary for the backtracking procedure later performed. The other outputs are important for the computation of the subsequent cells. In fact, only the vectors are defined as outputs, and the other signals are internal values of the pipeline. These vectors are defined like an output bus (vector bus) of 2.*N* bits, when *N* is the pipelines number. Initially, just the first MLPU contains valid values in their internal registers, and only some bits of the vector bus are valid. After some cycles the own MLPU apply valid values to the subsequent MLPU.

Table 1 shows an example of the computation of the dynamic programming matrix, when it is shown the evolution of signal throughout the MLPUs. In the first clock cycle, the first MLPU registers the used as input of *ColumnAA* and *LineAA*. Since the two values are gaps, after the gate propagation of the cell, the MLPU output value is *0d* and the output vector is *00b*. In the next clock cycle, the first MLPU output vector is registered in the second MLPU. *LinAAWe* of the second MLPU should be in high level, and all the other ones in low level. Then, the second MLPU will register the line amino acid "V" and, after the end of the gate propagation, the output vector presents *10b* as value.

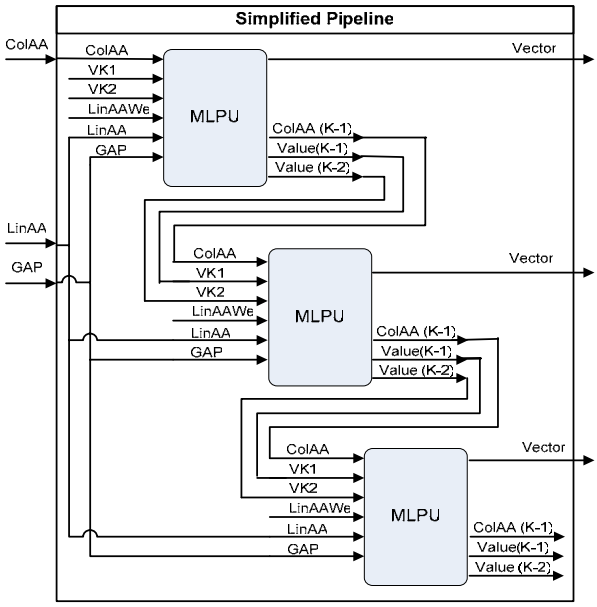

Fig. 6. Details of a possible Pipeline entity.

Values of the column amino acids travel through all of the MLPUs while the value of the line amino acid is kept fixed, and so the computation of all cells is done.

Table 1. Contents of the line and column registers, and respective computed vectors.

		Clock cycle							
	MLPU	0	1	2	3	4	5	6	7
ColAA	1	X	-	D	Y	E	X	X	X
	2	X	X	-	D	Y	E	X	X
	3	X	X	X	-	D	Y	E	X
LinAA	1	X	-	-	-	-	-	-	-
	2	X	X	V	V	V	V	V	V
	3	X	X	X	Y	Y	Y	Y	Y
Vector	1	XX	XX	00	10	10	10	XX	XX
	2	XX	XX	XX	01	11	11	10	XX
	3	XX	XX	XX	XX	01	01	11	10

2.5 Extended structure of the MLPU entity

Some additional logic is necessary to handle *LinAAWe* pins of each MLPU in a sequential way, during the first 3 clock cycles, and to maintain them inactive for the other cycles. This logic is supplied by the entity PipelineControl.

The pipeline is generalized to accomplish alignments with number of lines greater than the number of MLPUs. After finishing the computation of a line, a new round is

started, for the next *N* lines of the matrix. To do so, it is enough to access the next *N* line amino acids and provide again the column amino acids. However, notice that, by this time, the first MLPU is not in charge of computing the horizontal border penalties anymore. The computation depends on the values of the upper line that was computed by the last MLPU of the arrangement in the previous processing round. Considering that they are not synchronized in time with the current data among MLPUs, it is necessary a memory for to store these values with same size of the number of columns of the alignment.

Fig. 7 shows the complete structure of the Pipeline entity. The block PipeControl implements the logic to control the *LinAAWe* line of each MLPU and the memory storage to synchronize data.

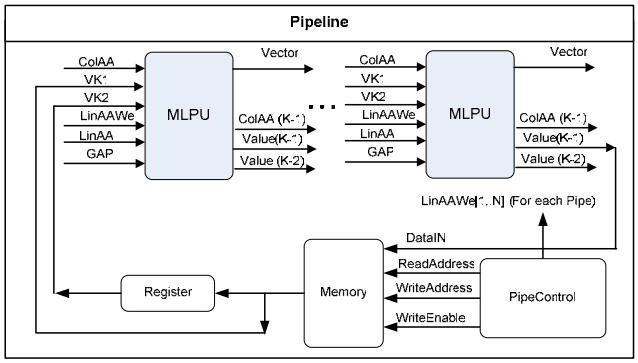

Fig. 7. Complete structure of the Pipeline entity.

2.6 Configuration layer

Each amino acid is encoded with 5 bits (20 types of valid amino acids, plus the gap), then we used triplets to represent them, so as to avoid waste of memory. An entity, denominated Pipeline Driver (Fig. 8), decodes these amino acids and applies them in the correct order to the pipeline input. The Pipeline Driver obeys the line amino acids limits, applies a new processing round whenever necessary and interrupts NIOS II, signaling the end of the process.

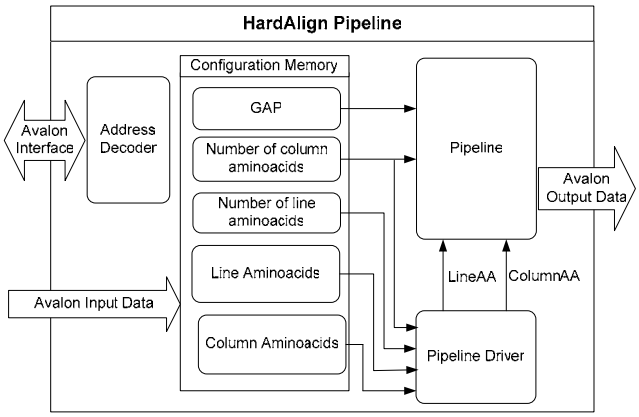

Fig. 8. A complete view of Hardalign pipeline.

The data necessary for correct operation of Pipeline (that is, gap penalty, number of line amino acids, number of column amino acids, value of the line amino acids and value of the column amino acids) are stored in a memory into the HardAlign Pipeline. An address decoder manages data input and output.

3. RESULTS

The synthesis of Hardalign was done in an Stratix II EP2S60F672C5ES device (Altera), with a 40 MHz clock rate. For compiling the circuitry described in VHDL we used SOPC builder (System on a Programmable Chip Builder) software (to integrate the several modules with the NIOS II core), and for the physical synthesis and simulations, Quartus II (Altera).

Two main experiments were done: analysis of resources allocation as function of number of MLPUs and analysis of the performance for different sizes of dynamic programming matrix, compared with a software implementation in a PC. For this last experiment, we used the same method implemented in hardware, but now implemented in C language, and run in a PC with Athlon XP 1600+ processor, 512Mb de RAM DDR 266 and Windows XP Operational System. An additional estimation is presented for the case when Hardalign is modified for using DMA (Direct Memory Access). This further modification will speed up data transfer rate, thus improving dramatically the performance.

3.1 Resources demand

The demand for LUTs and internal registers grows up linearly as function of the number of MLPUs. This experiment takes into account only the Pipeline logic, excluding the Avalon bus, NIOS II core, additional memories and pipeline drivers. The maximum frequency operation of pipeline is not affected by the number of MLPUs used. Table 2 shows the resources necessary for 5 to 64 MLPUs.

Table 2. Resources allocation.

Number of MLPUs	LUTs	Registers
5	1384	399
8	2189	589
16	4228	1094
32	8287	2103
64	16392	4120

3.2 Performace comparison

In this experiment, Hardalign used 8 MLPUs and was run at 40 MHz clock. The performance was tested for several

protein sizes to be aligned. Table 3 shows the computation time for pairs of 20- to 2000-amino acids-long sequences.

The resolution of the PC timer is 1 ms, precluding to measure the processing time for 20 *x* 20 and 50 *x* 50 matrices. In this table, we observed a 1:10 ratio, approximately. Data of this table is also shown in Fig. 9.

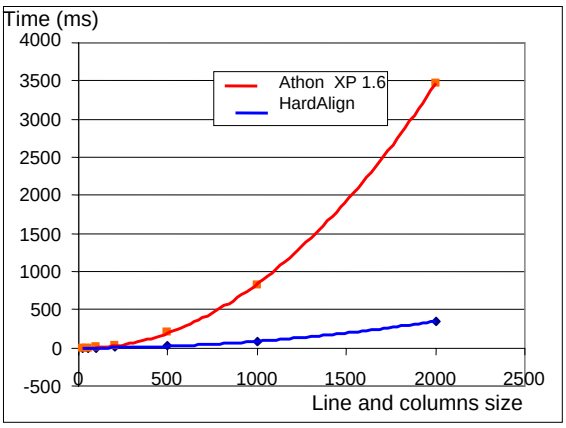

Fig. 9. Performance comparison between the software and hardware approaches shown in Table 3.

3.3 DMA performance

A further improvement of Hardalign is the use of a DMA controller module to improve the transfer rate between the SRAM and the Hardalign Pipeline (see Fig. 1). This comparison is shown in Table 3. Data was obtained by simulation. An improvement of about 28 times in performance can be observed comparing this approach and the previous one.

Table 3. Performance comparison.

Matrix size (lines *x* columns)	Hardalign (ms)	Hardalign with DMA (ms)*	Athlon XP (ms)
20 *x* 20	0.074	0.001	-
50 *x* 50	0.280	0.008	-
100 *x* 100	0.949	0.031	10
200 *x* 200	3.548	0.125	30
500 *x* 500	22.123	0.781	200
1000 *x* 1000	87.618	3.125	831
2000 *x* 2000	350.207	12.500	3465

* estimated

4. CONCLUSIONS

Pairwise sequence comparison by alignment is an important problem and still an open issue in computational biology, especially when dealing with large sequences. We have proposed a methodology for parallelizing a pairwise sequence alignment algorithm using a reconfigurable hardware computing. The performance analysis reveals that the improvement by using the hardware approach achieves around 1000% of speed-up, when compared with the conventional software processing. If we consider the use of DMA in Hardalign, this ratio, the hardware implementation can be something around 280 faster.

Reconfigurable logic for local processing allows a dramatic minimization of the processing time, when compared with a traditional software approach, currently in use. Such performance is possible thanks to the parallel processing and reduced computation time, inherent to a FPGA hardware implementation. The use of reconfigurable logic for this class of application was essential, and it enabled an efficient solution, satisfying the project requirements and constraints.

We believe that the proposed algorithm is a useful contribution to both reconfigurable systems and bioinformatics. In the next future we intend to do more extensive experiments and extend the system to the alignment of multiple sequences. In the same way, it is expected that this technology can be a feasible alternative for practical problems that require high computational power and real-time response, not only in bioinformatics, but also in combinatorial optimization problems.

5. REFERENCES

[1] Altera Corporation, *Nios II Processor Reference Handbook*, 2005.

[2] K. Compton, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp.171–210, 2002.

[3] S. Henikoff and J.G. Henikoff, "Amino acid substitution matrices from protein blocks," *PNAS*, vol. 89, pp. 10915-10919, 1992.

[4] A.R. Leach, *Molecular Modelling: Principles and Applications*, 2nd ed., Prentice-Hall, Dorset, 2001.

[5] H.S. Lopes and G.L. Moritz, "A distributed approach for a multiple sequence alignment algorithm using parallel virtual machine," In: *Proceedings of 27th Int. Conf. of IEEE EMBS*, Shanghai, China, 2005.

[6] S.B. Needleman and C.D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, pp. 433-443, 1970.

[7] M.S. Waterman, T.F. Smith, and W.A. Beyer, "Some biological sequence metrics," *Advances in Mathematics*, vol. 20, pp. 367-387, 1976.