Sequence Alignment Problems in Bioinformatics: A Guided Tour on Algorithms and Complexity

Jaime M. Samaniego
Institute of Computer Science
University of the Philippines Los Baños
+63 49 536 2313
jmsamaniego@uplb.edu.ph

ABSTRACT

The field of bioinformatics relies heavily on efficient algorithms for the alignment of two or more DNA or protein sequences. This paper introduces the basic optimization problems and their variations, discusses complexity issues, and surveys some of the classic and recent alignment algorithms.[1]

Keywords

sequence alignment, dynamic programming, linear-space alignment, wavefront parallelism, progressive alignment

1. INTRODUCTION

Given two finite sequences *X* and *Y* of symbols over some fixed alphabet Σ, the basic sequence alignment problem is that of inserting a minimum number of gaps in *X* and *Y* so as to maximize the number of matches and minimize the mismatches when *X* and *Y* are aligned. In molecular biology, the sequences are often either DNA sequences over the four-letter nucleotide alphabet {A,T,C,G}, or protein sequences with a 20-letter amino acid alphabet. The total score of an alignment is based on adding scores based on matches, and penalties based on mismatches or the presence of insertion and deletion gaps (also known as *indels*). Figure 1 provides an example with a pair of short sequences, a simple scoring matrix $s:(\Sigma\cup\{\})\times(\Sigma\cup\{\})\rightarrow Z$, and two sample alignments of the pair of sequences and their corresponding alignment scores.

```
Sequence X = ATTCGA
Sequence Y = ATCTCA
```

s	**-**	**A**	**T**	**C**	**G**
-	-∞	-1	-1	-1	-1
A	-1	+2	-1	-1	-1
T	-1	-1	+2	-1	-1
C	-1	-1	-1	+2	-1
G	-1	-1	-1	-1	+2

Sub-optimal alignment
with score = 5
```
X = ATTC-GA
Y = AT-CTCA
```

Optimal alignment
with score = 8
```
X = ATTC-GA
Y = AT-CTCA
```

Figure 1. Alignment of a pair of sequences

[1] A version of this paper was presented at the 2nd Symposium on the Mathematical Aspects of Computer Science, University of the Philippines Baguio, May 2004.

Applications of sequence alignment include the comparison of two or more sequences for similarity as a basis for measuring evolutionary distance, and the search for related sequences and subsequences in publicly available databases to infer possible similarity in gene functions.

For the case of two input sequences and a simple scoring function with a negative infinity score for mismatches, this problem is essentially equivalent to the well-known *longest common subsequence* problem. While there exist polynomial-time algorithms for the case of two sequences, researchers continue to design more efficient algorithms due to the unusually large sizes of most biological sequence data (e.g., tens of thousands of symbols are sometimes present in a sequence).

For multiple sequence alignment involving $k>2$ sequences, polynomial-time algorithms are unlikely (for arbitrary *k*) and the search for fast approximation algorithms that yield good alignments is an important task.

2. DYNAMIC PROGRAMMING ALGORITHMS

The basic alignment problem for two sequences of length *m* and *n* can be solved in $O(mn)$-time and $O(mn)$-space using a straightforward application of dynamic programming. In contrast, a brute-force approach that tries to evaluate all possible alignments can easily consume exponential time and such a method is clearly undesirable for very long sequences. A key to the design of a dynamic programming algorithm is the formulation of a recurrence relation for the score function. This is followed with an analysis of the dependencies, best represented with a directed graph. A topological ordering of the nodes provides the actual algorithm where partial scores are saved in a table to avoid redundant calculations.

For a 0/1-valued scoring matrix, one can establish a basic recurrence on the scores $s[r,c]$ of the partial alignments of $X[1..r]$ and $Y[1..c]$, for $r = 0, 1, \ldots, m$, and for $c = 0, 1, \ldots, n$.

The dependencies on these partial scores can be derived from the recurrence relation and the basic pattern is shown in Figure 2.

A table of partial scores when aligning GTCCT and GCCAAT' is shown Fig. 3. The alignment score is found in the lower-right corner of the table and the actual alignment can easily be traced back in $O(m+n)$ steps from this table by finding the maximum of three neighboring values. Alternatively, an augmenting matrix of back pointers can be constructed as we evaluate all the partial scores in *O(mn)*-time.

$$s[r,c] = \begin{cases} 0, & \text{if } r = 0 \text{ or } c = 0 \\ s[r-1,c-1]+1, & \text{if } X[r] = Y[c] \\ \max(s[r-1,c], s[r,c-1]), & \text{otherwise} \end{cases}$$

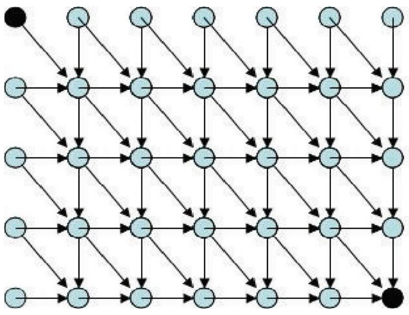

Figure 2. Dependency digraph of the recurrence relation.

We note that this algorithm is essentially that of finding a shortest path between two corner nodes in the dependency digraph with edge weights.

```
     -   G   C   C   A   A   T
-    0\  0   0   0   0   0   0
G    0  \1\  1   1   1   1   1
T    0   1|  1   1   1   1   2
C    0   1  \2\  2   2   2   2    GTCC--T
C    0   1   2  \3\-3--3\  3      G-CCAAT
T    0   1   2   3   3   3  \4\   score = 4
```

Figure 3. Partial scores and extracting the optimal alignment.

A topological ordering of the digraph can be obtained in several ways, three of which are: (a) by rows in sequence, (b) by column in sequence, or (c) by southwest-to-northeast diagonals in sequence, all summarized in Fig. 4. The first two orderings are useful for sequential algorithms based on this dynamic programming strategy, and the third is useful for a parallel algorithm based on wavefront patterns.

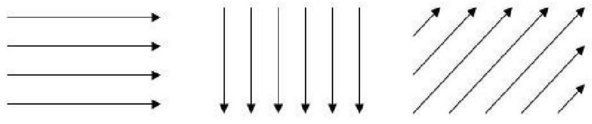

Figure 4. Possible topological orderings for the dependency digraph

This full-matrix algorithm can be modified to run in *O(min(m,n))* linear-space if we only need the final alignment score. This is accomplished by using only two rows (or two columns, whichever is shorter) in computing and saving the values in the next row. If we need the actual alignment, other space-efficient algorithms based on the classic divide-and-conquer strategy can be used and these will be discussed in the next section.

Sub-quadratic time algorithms also exist, and one approach is to compress the sequences using special data compression algorithms and perform the alignments on the compressed data. Details of this approach can be found in [7, 8] Another possibility is to restrict the computations in the matrix to a diagonal band of a certain width, which prevents long contiguous indel gaps in the alignments. A third approach is to preprocess the sequences by performing a maximal unique match (MUM) decomposition [9].

Binary-valued scoring matrices are often inadequate to properly model sequence data, particularly in the case of protein sequences. This is because some pairs of amino acids are more alike than other pairs, and hence a more general scoring matrix should be considered. Most researchers and popular software tools (e.g., BLAST or ClustalW) use either the family of PAM (percent accepted mutation) matrices or BLOSUM (blocks substitution matrix) in their computations of alignment scores. A modified recurrence relation is given below that considers such general scoring schemes.

$$s[r,c] = \begin{cases} 0, & \text{if } r = 0 \text{ and } c = 0 \\ s[r-1,0] + gap, & \text{if } c = 0 \\ s[0,c-1] + gap, & \text{if } r = 0 \\ \max, & \text{otherwise} \end{cases}$$

$$\text{where} \max = \max \begin{cases} s[r-1,c] + gap \\ s[r,c-1] + gap \\ s[r-1,c-1] + Pscore(X[r],Y[c]) \end{cases}$$

Here, *gap* is the gap penalty, *Pscore(a,a)* is the match bonus taken from the PAM or BLOSUM matrices, and *Pscore(a,b)* is the mismatch penalty for any distinct pair of symbols *a, b* in the alphabet Σ. Modifications in the dynamic programming algorithm are straightforward with similar running times and storage requirements.

The scoring of alignments is made more complex by considering various ways indel gaps are penalized. In our basic algorithm, we simply penalized each gap by a constant amount. An {\it affine gap model} uses a linear penalty function that includes a constant penalty for initiating a gap and another term that is proportional to the length of the gap. Other researchers insist on using models based on *concave gap functions* (rather than linear functions) as they feel the affine gap model is still inadequate in representing the underlying mutations or other biological mechanisms that lead to these gaps. Both models basically address the biological specification that a single long gap should be penalized less than several short gaps with the same total length. Details on how the recurrence relation should be modified to consider these models can be found in [22].

Our initial description of the sequence alignment problem in which we align the entire sequence is referred to as *global alignment. Semi-global alignment* is similarly defined but the gaps located at the beginning or at the end of the alignments should not be penalized. This assumption is particularly relevant in deducing evolutionary relationships among organisms as one or both DNA sequences may have been padded with leading or trailing residues that are unrelated to the specific region of interest.

```
AGTTCACAATTGATTCG    AGTTCACA-ATT-GATTCG
AG---ACA-----TTCG    -----AGACATTCG-----
```

Figure 5. Global alignment vs. semi-global alignment

Modifications in the dynamic programming algorithm to produce semi-global alignments are fairly straightforward. To ignore trailing gaps in one of the sequences, for example, we search the last row or the last column in the matrix of partial scores for maximum values.

The search for regions of local similarity, commonly referred to as *local alignment*, is a third type where the goal is to find the best alignment between *substrings* of the two sequences. Finding the optimal local alignment is a classic combinatorial problem. A brute-force extension to our basic *O(mn)* dynamic programming algorithm wherein all pairs of possible substrings are aligned results in an impractical $O(m^3n^3)$ algorithm. However, a careful analysis and reconstruction of the recurrence relation can result in a more efficient *O(mn)* algorithm, which was first developed by Smith and Waterman in 1981 [23].

$$s[r,c] = \max\begin{cases} 0, & \textit{for aligning empty strings} \\ s[r-1,c-1] + Pscore(X[r],Y[c]), & \textit{for a match or mismatch} \\ s[r-1,c] + Pscore(X[r],-), & \textit{for a delete} \\ s[r,c-1] + Pscore(-,Y[c]), & \textit{for an insert} \end{cases}$$

This results in a table of local alignment scores where multiple maximal paths can be derived. Alternative global and semi-global alignments can also be assembled from this matrix by searching for compatible local alignment paths, which is yet another interesting combinatorial optimization problem based on the dependency digraph.

Determining whether or not a given alignment is statistically significant is a non-trivial but important task. This is particularly true for small alphabets such as the 4-letter alphabet in DNA sequences in which some of the pairs of symbols will definitely align even for randomly permuted sequences. Unfortunately, little is known about the exact tail distribution of alignment scores even with known frequencies of the symbols in the alphabet Σ.

A simple but computationally expensive approach for determining statistical significance is through a Monte Carlo simulation. The sequences are permuted and re-aligned several times under some ideal sampling strategy, and a frequency distribution of their scores is used as the basis for statistical significance. More sophisticated methods are provided in [24].

3. LINEAR-SPACE ALIGNMENT AND WAVEFRONT PARALLELISM

Recall that the full-matrix dynamic programming algorithms require quadratic space to store the partial scores and to retrieve the optimal alignment. When aligning very long DNA or protein equences, a more space-efficient algorithm is clearly desirable. Hirschberg in 1975 presented the first linear-space algorithm for the closely-related *longest common subsequence* problem \cite{waterman, chao1}. It uses the classic divide-and-conquer strategy to reduce storage requirements from *O(mn)* to *O(min{m,n})*, with only a moderate increase in running time.

The basic idea is to bisect one of the sequences, say *X*, find the best alignment scores of the two halves of *X* with the other sequence *Y*, and recursively align two pairs of smaller subsequences. This divide-and-conquer algorithm is presented in more detail in Fig. 6, along with a graphical representation in Fig. 7.

```
algorithm align ( sequence x, int m, sequence y, int n ) {
   if (m+n is small enough) {
      align_directly(x, m, y, n) using dynamic programming;
   }
   else {
      split x equally into x1 and x2;
      find the best score of aligning x1 and y;
      find the best score of aligning reverse(x2) and reverse(y);
      find the column j that has the maximum
         sum of the scores in the middle row;
      recursively align ( x1, m/2, y[1..j], j );
      recursively align ( x2, m/2, y[j..n], n-j );
   }
}
```

Figure 6. Hirschberg's divide-and-conquer algorithm

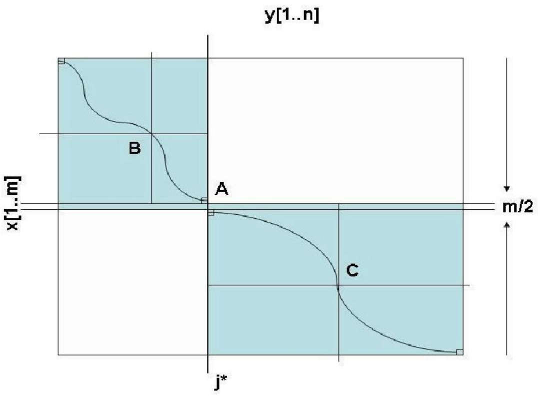

Figure 7. Graphical illustration of Hirschberg's algorithm

Note that the total cost of the two sub-problems is half the cost of the original problem ($T = mn$). Continuing recursively, we then get a fourth of the original cost, then an eighth, then a sixteenth, etc., i.e., $T + T/2 + T/4 + T/8 + \ldots = 2T$. It follows we need twice the time to come up with the optimal alignment as compared to the classic full-matrix algorithm, but this divide-and-conquer variant uses only linear storage.

Another linear-space algorithm known as FastLSA (Fast Linear Space Alignment) [10] generalizes both the full-matrix and Hirschberg's algorithm. An important feature of fastLSA is that it is adaptive to the amount of available storage. By setting a parameter that indicates the amount of available memory, it can behave like a full-matrix quadratic-space algorithm, or a linear-space algorithm. An added advantage is that localization of the computations avoids excessive memory swaps and claims of reduced running times have been empirically observed [10].

Recall that Hirschberg's method recursively divides the problem in half, and saving the row information. In FastLSA, both input sequences are divided, i.e., the matrix is bisected both row-wise and column-wise, and both row and column boundary information are saved. A double split allows the computation of an optimal alignment while recalculating fewer values, as demonstrated in Fig. 8a.

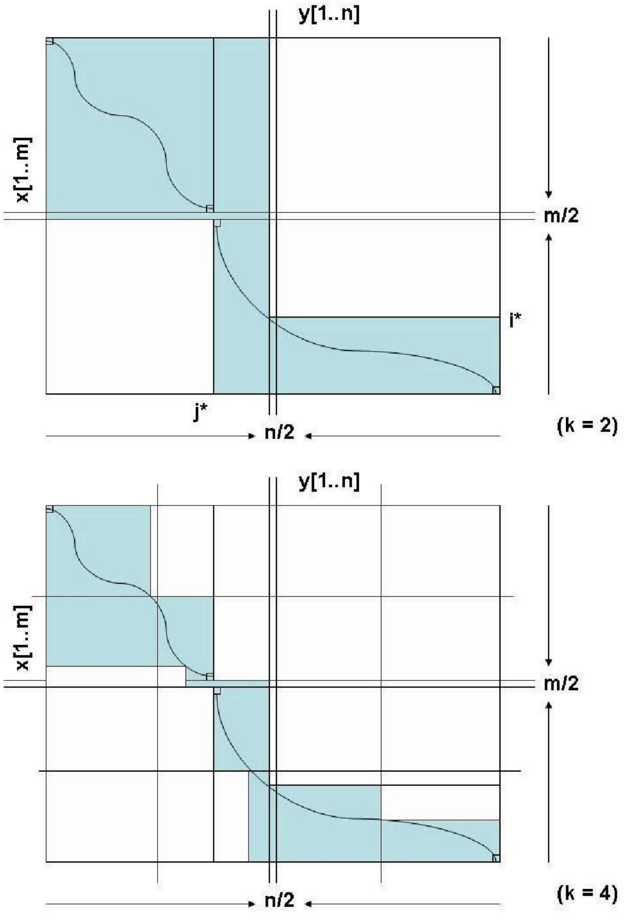

Figure 8. Fast LSA with (a) binary double splits, and (b) k>2 subproblems

Partitioning the sequences into $k>2$ subproblems (Fig. 8b) can yield even better performance, where an optimal choice of k is based on available cache or main memory sizes. A careful analysis shows the storage requirements $S(m,n,k)$, with a $k \times k$ grid cache, is expressed by the inequality $S(m,n,k) \leq k \times (m+n) + BaseCaseBuffer$. Worst-case running times for FastLSA remain quadratic at $T(m,n,k) = m \times n \times (k+1)/(k-1)$. Note that this upper bound decreases as the value of k increases. Details, as well as extensions to a parallel implementation, are in [10].

These algorithms can be extended when p processors (e.g., Beowulf clusters) are available [15]. A common theme associated with the basic dynamic programming recurrence is to use the wavefront pattern [3] in Fig. 9. In this digraph, each node represents a k×k contiguous sub-block (where $k = n/p$) of the dynamic programming matrix that can be assigned to one of the processors.

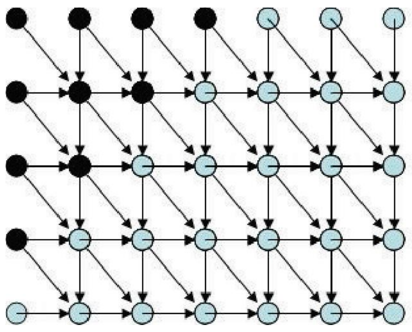

Figure 9. Wavefront computation

Under a message-passing model of cluster computing, the computation of p sub-blocks along a wavefront diagonal will be done in parallel. After computing each block, only the bottom row vector, right-most column vector, and corner value need to be passed along to the neighboring blocks below and to the right.

4. MULTIPLE SEQUENCE ALIGNMENT

The simultaneous alignment of several sequences is significantly a much harder problem than pairwise alignment. Extending the dynamic programming algorithm to 3 or more dimensions is computationally impractical in terms of both running time and storage. The general problem is in fact NP-hard [11, 6, 12] and thus polynomial-time algorithms are unlikely to be found. Jiang [13]further showed that even the simpler *longest common subsequence* problem for n sequences cannot be approximated with ratio n^{δ} for some $\delta > 0$ in polynomial time, unless **P = NP**. The development of fast approximation algorithms that yield reasonably good alignments for several long sequences will therefore continue to be the focus of many research groups.

Several heuristic methods for multiple sequence alignment are employed in the widely-used bioinformatics packages (e.g., ClustalW). These are often based on greedy-type heuristics and commonly referred to as *progressive alignment*. Many of the more powerful iterative multiple alignment methods are still under testing and have yet to be incorporated in the more popular packages.

Progressive alignment uses a simple greedy technique which works as follows: Two of the closest sequences are aligned together, and the resulting alignment replaces the original pair. This process is repeated until all sequences have been pairwise aligned. One way of determining which sequence is to be aligned next is based on the construction of a phylogenetic tree, which in turn is based on the graph-theoretic notion of Steiner trees [11]. This is itself an NP-hard problem but several good approximation algorithms exist [21]. A more sophisticated variation of the progressive alignment algorithm that allows user-specified constraints on where certain sequence positions should appear relative to others makes this method relatively powerful [16].

A more powerful (but often slower) class of methods is based on the iterative improvement of multiple alignments. This includes the use of Tabu search [20], and not surprisingly for the application -- genetic algorithms [2,18]. A more extensive survey of such methods can be found in [17].

Tabu search involves the iterative improvement of a multiple alignment by perturbing some sections of the solution, but uses a set of restricting rules to avoid local traps and allow a more extensive exploration of the solution space.

Genetic algorithms are based on the iterative improvement of a population of feasible solutions using the principles of natural selection. Within each iteration, each solution is evaluated, and some of the best ones are probabilistically selected as parents for the next generation of feasible solutions. A recombination operator is applied to certain pairs of parent solutions to produce new child solutions which hopefully will inherit some of the better alignment building blocks of their parents. Mutations that change some aspect of the solutions, when applied with a small but significant probability, can prevent premature convergence and allow further exploration of the solution space. After several generations, the best multiple alignment encountered serves as the final solution.

One specific implementation of a genetic algorithm for multiple sequence alignment is SAGA [18], and it involves several additional problem-dependent operators to improve its performance. This includes a two-phase crossover operation (in which only the better of the two children produced is retained), gap insertion, block-shuffling, block searching, and sub-optimal rearrangement.

A parallel implementation is but natural for genetic algorithms due to the need to maintain multiple feasible solutions. In the island model of parallel genetic algorithms [2], the population is partitioned and each group assigned to a processor. A genetic algorithm can run fairly independently in each processor but each subpopulation are also occasionally allowed to exchange some of the better solutions with its neighboring processors.

5. DISCUSSION

It is interesting to note that the island model of parallel genetic algorithms in a way mirrors the way in which modern biologists, physicists, mathematicians, statisticians and computing scientists co-evolve as they identify domain-specific problems, search for potential solution methods from neighboring disciplines and integrate these with their own strategies.

These efforts should yield further collaboration among different disciplines in several avenues. In basic computer science education for example, we have applied the principles of multiple sequence alignment in the automated detection of plagiarism in student programming assignments with good initial results. Just knowing that instructors have automated tools that can potentially check on hundreds of assignments is a great deterrent to plagiarism and can therefore improve students' efforts in programming. Given the fundamental nature of sequence comparison problems, such methods can also be useful for computer virus detection and taxonomy, as well as for the automated classification of web documents.

The automated discovery of short but powerful protein subsequences called *motifs* by mining megabytes of sequence databases is another one of the advanced applications of sequence alignment. Besides the obvious impact on medical and agricultural applications, having better knowledge of just how biology works at the molecular level might eventually lead us to practical DNA-based computers for solving future computational challenges.

6. ACKNOWLEDGEMENTS

We wish to thank PCASTRD for its support of this study. We also wish to thank Eliezer Albacea, Ramil Mauleon, Leo Namuco, Arian Jacildo, Aurea Perez, Joseph Hermocilla and Dean Berris for interesting discussions related to these problems.

7.REFERENCES

[1] T. Akutsu. Optimization problems and metaheuristics in bioinformatics. *Fifth Metaheuristic International Conference* (2003).

[2] L.A. Anbarasu, P. Narayanasamy, V. Sundararajan. Multiple molecular sequence alignment by island parallel genetic algorithm. *Current Science* (2000) 78:7.

[3] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, K. Tan}, Generating parallel programs from the wavefront design pattern. (2002). www.cs.ualberta.ca/~systems/papers/HIPS02.pdf.

[4] K.-M. Chao, R.C. Hardison, W. Miller , Recent developments in linear-space alignment methods: a survey. *J. of Comp. Biol.* (1994). 271-291.

[5] K.-M. Chao, Dynamic programming strategies for analyzing biomolecular sequences. (2003). www.csie.ntu.edu.tw/~kmchao/seq2003/dp.pdf.

[6] P. Crescenzi, V. Kann, A compendium of NP optimization problems. www.nada.kth.se/~viggo/wwwcompendium/node167.html

[7] X. Cheng, S. Kwong, M. Li, A compression algorithm for DNA sequences and its applications in genome comparison. http://www.ncbi.nlm.nih.gov/pubmed/11072342

[8] M. Crochemore, G.M. Landau, M. Ziv-Ukelson, A sub-quadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM J. Comput. 32* (6): 1654-1673 (2003).

[9] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, Alignment of whole genomes. *Nucleic Acids Research* (1999) 27:11, 2369-2376. www.tigr.org/~salzberg/MUMmer.pdf.

[10] A. Driga, P. Lu, J. Schaeffer, D. Szafron, K. Charter, I. Parsons}, FastLSA: A fast, linear-space, parallel and sequential algorithm for sequence alignment. (2003) www.cs.ualberta.ca/duane/pdf/2003icpp.pdf.

[11] M.R. Garey, D.S. Johnson , Computers and intractability: a guide to the theory of NP-completeness. Freeman, New York, (1979).

[12] T. Jiang, P. Kearney, M. Li, Some open problems in computational molecular biology. *J. of Algorithms* 34, (2000) 194-201.

[13] T. Jiang, M. Li, On the approximation of shortest common supersequences and longest common subsequences. LNCS 820 (1994).

[14] W. Just, G. Della Vedova, Multiple sequence alignment as a facility location problem. www.statistica.unimib.it/utenti/dellavedova/.

[15] K. Michalickova, M. Dharsee, C.W.V. Hogue, Sequence analysis on a 216-processor Beowulf cluster. *Proc. 4th Annual Linux Showcase Conf.* (2000) www.blueprint.org/proteinfolding/distributedfolding/docs/moblast.pdf.

[16] G. Myers, S. Selznick, Z. Zhang, W. Miller, Progressive multiple alignments with constraints. *J. of Computational Biology* 3, 4 (1996), 563-572

[17] C. Notredame , Recent progresses in multiple sequence alignment: a survey. *Pharmacogenomics* (2002) 3:1, 131-144.

[18] C. Notredame, D.G. Higgins, SAGA: Sequence alignment by genetic algorithm. *Nucleic Acids Research* (1996) 24:8, 1515-1524. igs-server.cnrs-mrs.fr/~cnotred/Publications/Ps_pdf/saga_paper.pdf.

[19] C. Notredame, D.G. Higgins, J. Heringa, T-Coffee: A novel method for multiple sequence alignments. *J. of Molecular Biology* (2000) 302, 205-217.

[20] T. Riaz, Y. Wang, K.-B. Li, Multiple sequence alignment using Tabu search. *2nd Asia-Pacific Bioinformatics Conference*.

[21] J.M. Samaniego, Evolutionary methods for the Steiner problem in networks: an experimental evaluation. *Proc. Symposium on Mathematical Aspects of Computer Science*, UP Baguio, (1997), 99-108.

[22] W.-K. Sung, Lecture notes on sequence comparison, Combinatorial methods in Bioinformatics (2003).

[23] M. Waterman, Introduction to computational biology - maps, sequences and genomes. Chapman & Hall, London (1995).

[24] Y.-K. Yu, T. Hwa, Statistical significance of probabilistic sequence alignment and related local hidden Markov models. *J. of Computational Biology* 8:3 (2001), 249-282.