A Parameterized Parallel Algorithm for Efficient Biological Sequence Comparison

C. E. R. Alves
Fac. de Tecnologia e Ciências Exatas
Universidade São Judas Tadeu
São Paulo-SP-Brazil
prof.carlos_r_alves@usjt.br
http://www.usjt.br/

E. N. Cáceres *
Dept. of Computação e Estatística
Universidade Federal de Mato Grosso do Sul
Campo Grande-MS-Brazil
edson@dct.ufms.br
http://www.dct.ufms.br/∼edson

F. Dehne †
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
frank@dehne.net
http://www.dehne.net

S. W. Song ‡
Dept. de Ciência da Computação - IME
Universidade de São Paulo
São Paulo-SP-Brazil
song@ime.usp.br
http://www.ime.usp.br/∼song

Abstract

In this paper we extend and improve a previous result for computing an alignment (or string editing) between two strings A and C, with $|A| = m$ and $|C| = n$, which requires $O(p)$ communication rounds and $O(\frac{nm}{p})$ local computing time, on a distributed memory parallel computer of p processors each with $O(nm/p)$ memory. The algorithm is based on a compromise between the workload of each processor and the number of communication rounds required, expressed by a new parameter called α. The proposed algorithm is expressed in terms of this parameter. We implemented this algorithm to determine the best values for α, tuned to obtain the smallest overall parallel time. We show very promising experimental results obtained on a 64-node Beowulf machine.

1 Introduction

In Molecular Biology, the search for tools that identify, store, compare and analyze very long biosequences is becoming a major research area in Computational Biology. In particular, sequence comparison is a fundamental problem that appears in more complex problems [13], such as the search of similarities between biosequences [11, 12, 14], as well as in the solution of several other problems [10, 9, 16].

One way to identify similarities between sequences is to align them, with the insertion of spaces in the two sequences, in such way that the two sequences became of the same size. In the similarity approach, we are interested in the best alignment between two strings, and the score of such an alignment gives a measure of how much the strings are alike.

In the distance approach, we want to find the minimum number of insertions, deletions and substitutions needed to transform one sequence into the other. In other words, we want to edit one of the strings and make it equal to the other. We assign costs to elementary edit operations and seek the less expensive composition of these operations.

The notions of similarity and distance are, in most of the time, interchangeable and both can be used to infer the functionality or the aspects related with the evolutive history of the involved sequences. In either case we are looking for a numeric value that measures the degree by which the sequences are alike.

In this paper we extend and improve a previous result for computing an alignment (or string editing) between two strings A and C, with $|A| = m$ and $|C| = n$, which requires $O(p)$ communication rounds and $O(\frac{nm}{p})$ local computing time, on a distributed memory parallel computer of

*Research partially supported by CNPq, FINEP-PRONEX-SAI Proc. No. 76.97.1022.00 and FAPESP Proc. No. 1997/10982-0.
†Research partially supported by the Natural Sciences and Engineering Research Council of Canada.
‡Research partially supported by FAPESP Proc. No. 99/07390-0, CNPq Proc. No. 52.3778/96-1, 46.1230/00-3, and 521097/01-0 and CNPq/NSF Proc. No. 68.0037/99-3.

p processors each with $O(nm/p)$ memory. The proposed algorithm is based on a compromise between the workload of each processor and the number of communication rounds required, expressed by a new parameter called α. The algorithm is expressed in terms of this parameter. We implemented this algorithm to determine the best values for α, tuned to obtain the smallest overall parallel time. We show very promising experimental results obtained on a 64-node Beowulf machine. Thus in addition to showing theoretic complexity we confirm the efficiency of the proposed algorithm through implementation.

Let $A = a_1a_2 \ldots a_m$ and $C = c_1c_2 \ldots c_n$ be two strings over some alphabet I. An *alignment* between A and C is a matching of the symbols $a \in A$ and $c \in C$ in such way that if we draw lines between the matched symbols, these lines cannot cross each other. The alignment shows the similarities between the two strings. Given an alignment between two strings, we can assign a *score* to it as follows. Each column of the alignment receives a certain value depending on its contents and the total score for the alignment is the sum of the values assigned to its columns. If a column has two identical characters $r = s$, it will receive a value $p(r,s) > 0$ (a *match*). Different characters $r \neq s$ will give the column value $p(r,s) < 0$ (a *mismatch*). Finally, a space in a column receives a value $-k$, where $k \in N$. We look for the value of the best alignment (*optimal alignment*) which gives the maximum score. This maximum score is called the *similarity* between the two strings to be denoted by $sim(A,C)$ for strings A and C. There may be more than one alignment with maximum score [13].

A sequential algorithm to compute the similarity between two strings uses a technique called *dynamic programming*. The complexity of this algorithm is $O(nm)$. The construction of the optimal alignment can be done in sequential time $O(m + n)$ [13].

Consider $|A| = m$ and $|C| = n$. We can obtain the solution by computing all the similarities between arbitrary prefixes of the two strings starting with the shorter prefixes and use previously computed results to solve the problem for larger prefixes. There are $m + 1$ possible prefixes of A and $n + 1$ prefixes of C. Thus, we can arrange our calculations in an $(m + 1) \times (n + 1)$ matrix S where each $S(r,s)$ represents the similarity between $A[1 \ldots r]$ and $C[1 \ldots s]$.

Observe that we can compute the values of $S(r,s)$ by using the three previous elements $S(r-1,s)$, $S(r-1,s-1)$ and $S(r,s-1)$, because there are only three ways of computing an alignment between $A[1 \ldots r]$ and $C[1 \ldots s]$. We can align $A[1..r]$ with $C[1..s-1]$ and match a space with $C[s]$, or align $A[1..r-1]$ with $C[1..s-1]$ and match $A[r]$ with $B[s]$, or align $A[1..r-1]$ with $C[1..s]$ and match a space with $A[r]$.

The similarity of the alignment between strings A and C can be computed as follows:

$$S(r,s) = \max \begin{cases} S[r,s-1] - k \\ S[r-1,s-1] + p(r,s) \\ S[r-1,s] - k \end{cases}$$

An $l_1 \times l_2$ *grid DAG* (Figure 1) is a directed acyclic graph whose vertices are the l_1l_2 points of an $l_1 \times l_2$ grid, with edges from grid point (i,j) to the grid points $(i,j+1)$, $(i+1,j)$ and $(i+1,j+1)$.

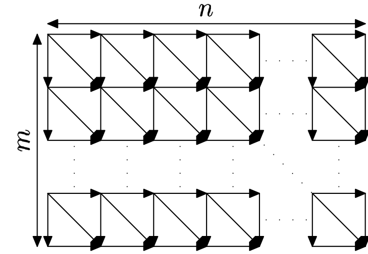

Figure 1. Grid DAG G

We associate an $(m+1) \times (n+1)$ grid dag G with the similarity problem in the natural way: the $(m+1)(n+1)$ vertices of G are in one-to-one correspondence with the $(m+1)(n+1)$ entries of the S-matrix, and the cost of an edge from vertex (t,l) to vertex (i,j) is equal to k if $t = i$ and $l = j-1$ or if $t = i-1$ and $l = j$; and to $p(i,j)$ if $t = i-1$ and $l = j-1$.

It is easy to see that the similarity problem can be viewed as computing the minimum source-sink path in a grid DAG.

One way to explore the use of parallel computation can be through the use of clusters of workstations or Fast/Gigabit Ethernet connected Unix-based Beowulf machines, with *Parallel Virtual Machine - PVM* or *Message Passing Interface - MPI* libraries. The latency in such clusters or Beowulf machines of 1Gb/s is currently less than 10 μs and programming using these resources is today a major trend in parallel and distributed computing.

Efficient parallel PRAM (*Parallel Random Access Machine*) algorithms for the dynamic programming problem have been obtained by Galil and Park [6, 7]. PRAM algorithms for the string editing problem have been proposed by Apostolico et al. [3]. A more general study of parallel algorithms for dynamic programming can be seen in [8]. PRAM algorithms, however, do not take into account communication and assume the number of available processors to be the same order of the problem size (*fine granularity*). When such theoretically efficient algorithms are implemented on real existing machines, the speedups obtained are often disappointing.

To deal with this problem, Valiant [15] introduced a simple *coarse granularity* model, called *Bulk Synchronous Parallel Model - BSP*. It gives reasonable predictions on the

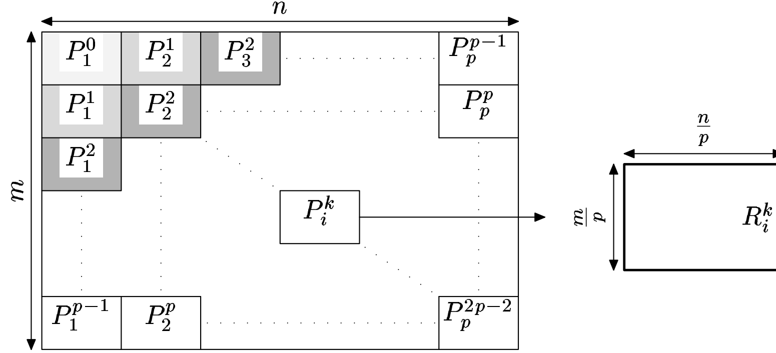

Figure 2. An $O(p)$ communication rounds scheduling with $\alpha = 1$

performance of the algorithms when implemented on existing, mainly distributed memory, parallel machines. A BSP algorithm consists of a sequence of supersteps separated by *synchronization barriers*. In a superstep, each processor executes a set of independent operations using local data available in each processor at the start of the superstep, as well as communication consisting of send and receive of messages. An *h-relation* in a superstep corresponds to sending or receiving at most h messages in each processor.

A similar model is the *Coarse Grained Multicomputers - CGM*, proposed by Dehne *et al.* [4]. In this model, p processors are connected through any interconnection network. The term *coarse granularity* comes from the fact that the problem size in each processor n/p is considerably larger than the number of processors. A CGM algorithm consists of a sequence of rounds, alternating well defined local computing and global communication. Normally, during a computing round we use the best sequential algorithm for the processing of data available locally. A CGM algorithm is a special case of a BSP algorithm where all the communication operations of one superstep are done in the h-relation. The CGM algorithms implemented on currently available multiprocessors present speedups similar to the speedups predicted in theory [5]. The CGM algorithm design goal is to minimize the number of supersteps and the amount of local computation.

We are interested in obtaining parallel algorithms that can be implemented on available parallel machines and obtain compatible execution times as predicted in the CGM model, independent of the particular type of interconnection network used. Sequence comparison is one of the basic and most used operations in computational Biology. Our work extend the results of [2], improving the time and the bounds of the string size in the implementation. We have implemented the algorithms on a Beowulf with 64 nodes with very promising results.

2 A Parameterized CGM/BSP $O(p)$ Communication Rounds Alignment Algorithm

In this section we present a parameterized $O(p)$ communication rounds parallel algorithm for computing the similarity between two strings A and B, over some alphabet I, with $|A| = m$ and $|C| = n$. We use the CGM/BSP model with p processors, where each processor has $O(\frac{mn}{p})$ local memory. This algorithm extends the result presented by Alves et al [2].

Let us first give the main idea to compute the similarity matrix S by p processors.

The string A is broadcasted to all processors, and the string C is divided into p pieces, of size $\frac{n}{p}$, and each processor P_i, $1 \leq i \leq p$, receives the i-th piece of C ($c_{(i-1)\frac{n}{p}+1} \ldots c_{i\frac{n}{p}}$).

The scheduling scheme of our previous work [2] can be illustrated in Figure 2. The notation P_i^k denotes the work of Processor P_i at round k. Thus initially P_1 starts computing at round 0. Then P_1 and P_2 can work at round 1, and so on. In other words, after computing the k-th part of the submatrix S_i (denoted S_i^k), the processor P_i sends to processor P_{i+1} the elements of the right boundary (rightmost column) of S_i^k. These elements are denoted by R_i^k. Using R_i^k, processor P_{i+1} can compute the k-th part of the submatrix S_{i+1}. After $p-1$ rounds, the processor P_p receives R_{p-1}^1 and computes the first part of the submatrix S_p. In the $2p-2$ round, the processor P_p receives R_{p-1}^p and computes the p-th part of the submatrix S_p and finishes the computation.

It is easy to see that with this scheduling, the processor P_p only initiates its work when the processor P_1 is finishing its computation, at round $p-1$. Therefore, we have a very poor load balancing.

Our new approach intends to attribute work to the processors as soon as possible. This can be done by decreasing the size of the messages that processor P_i sends to processors P_{i+1}. Instead of message size m/p we consider sizes

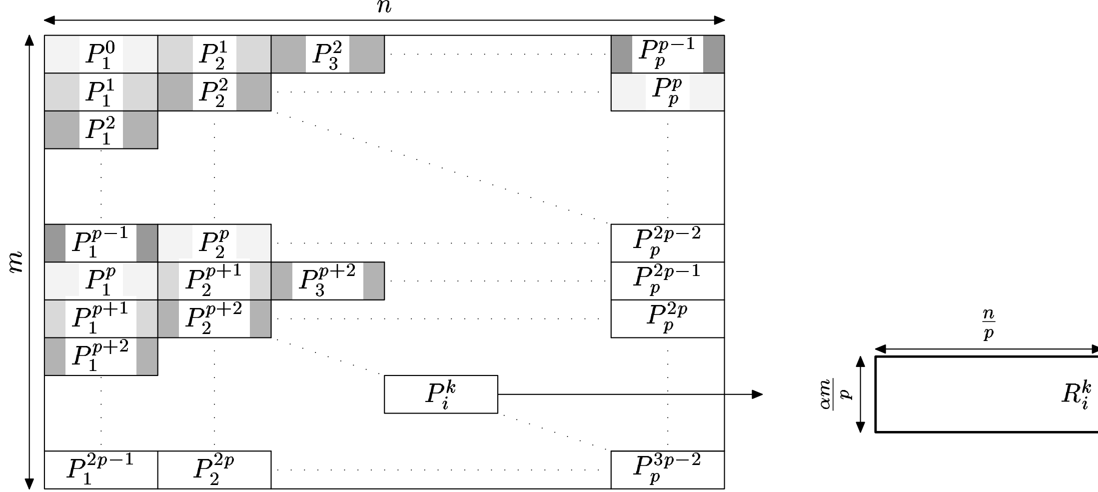

Figure 3. An $O(p)$ rounds scheduling with $\alpha = 1/2$

$\alpha\frac{m}{p}$ and explore several sizes of α. In our work, we make the assumption that the sizes of the messages $\alpha\frac{m}{p}$ divides m.

Therefore, S_i^k (the similarity submatrix computed by processor P_i at round k) represents $k\alpha\frac{m}{p}+1$ to $(k+1)\alpha\frac{m}{p}$ rows of S_i that are computed at the round k.

Algorithm 1 Similarity
Input: (1) The number p of processors; (2) The number i of the processor, where $1 \leq i \leq p$; and (3) The string A and the substring C_i of size m and $\frac{n}{p}$, respectively; (4) The constant α.
Output: $S(r,s) = \max\{S[r,s-1]-k, S[r-1,s-1]+p(r,s), S[r-1,s]-k\}$, where $(i-1)\frac{m}{\sqrt{p}}+1 \leq r \leq i\frac{m}{\sqrt{p}}$ and $(j-1)\frac{n}{p}+1 \leq s \leq j\frac{n}{p}$.

(1) **for** $1 \leq k \leq \frac{p}{\alpha}$
  (1.1) **if** $i = 1$ **then**
    (1.1.1) **for** $\alpha(k-1)\frac{m}{p}+1 \leq r \leq \alpha k\frac{m}{p}$ **and** $1 \leq s \leq \frac{n}{p}$
      compute $S(r,s)$;
    (1.1.2) send(R_i^k, P_{i+1});
  (1.2) **if** $i \neq 1$ **then**
    (1.2.1) **receive**(R_{i-1}^k, P_{i-1});
    (1.2.2) **for** $\alpha(k-1)\frac{m}{p}+1 \leq r \leq \alpha k\frac{m}{p}$ **and** $1 \leq s \leq \frac{n}{p}$
      compute $S(r,s)$;
    (1.2.3) **if** $i \neq p$ **then**
      send(R_i^k, P_{i+1});

— End of Algorithm —

Algorithm 1 works as follow: After computing S_i^k, the processor P_i sends R_i^k to processor P_{i+1}. The processor P_{i+1} receives R_i^k from P_i and computes S_{i+1}^{k+1}. After $p-2$ rounds, the processor P_p receives R_{p-1}^{p-2} and computes S_p^{p-1}. If we use $\alpha < 1$ all the processors will work simultaneously after the $(p-2)$-th round. We explore several values for α trying to find a balance between the workload of the processors and the number of rounds of the algorithms. Figure 3 shows how the algorithm works when $\alpha = 1/2$. In this case, processor P_p receives R_{p-1}^{3p-3}, computes S_p^{3p-2} and finishes the computation.

Using the schedule of Figure 3, we can see that in the first round, only processor P_1 works. In the second round, processors P_1 and P_2 work. It is easy to see that in round k, all processors P_i work, where $1 \leq i \leq k$. Since the total number of rounds is increased with smaller values of α the processors start working earlier.

Theorem 1 *Algorithm 1 uses $(1+1/\alpha)p-2$ communication rounds with $O(\frac{mn}{p})$ sequential computing time in each processor.*

Proof. Processor P_1 sends R_1^k to processor P_2 after computing the k-th block of $\alpha\frac{m}{p}$ lines of the $\frac{mn}{p}$ submatrix S_1. After $p/\alpha - 1$ communication rounds, processor P_1 finishes its work. Similarly, processor P_2 finishes its work after p/α communication rounds. Then, after $p/\alpha - 2 + i$ communication rounds, processor P_i finishes its work. Since we have p processors, after $(1+1/\alpha)p-2$ communication rounds, all the p processors have finished their work.

Each processor uses a sequential algorithm to compute the similarity submatrix S_i. Thus this algorithm takes $O(\frac{mn}{p})$ computing time. ⋆

Theorem 2 *At the end of Algorithm 1, $S(m,n)$ will store the score of the similarity between the strings A and C.*

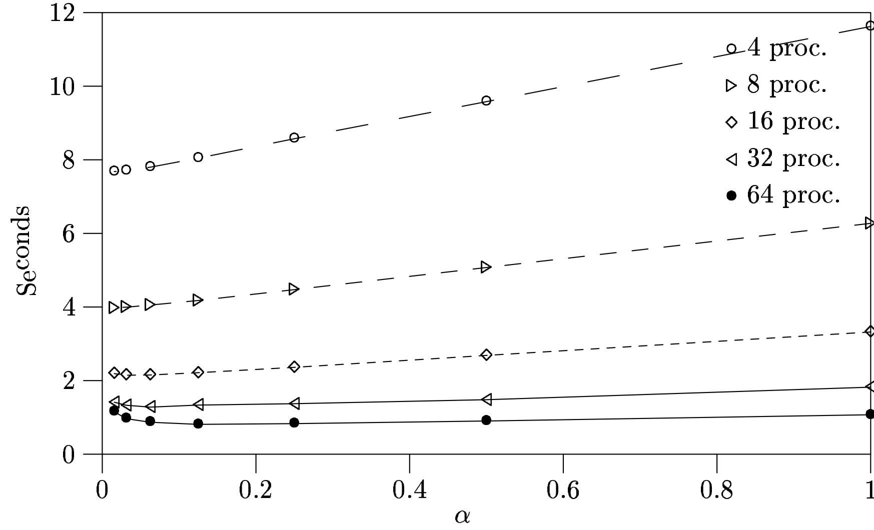

Figure 4. Results of a 8Kx16K matrix for several values of α

Proof. Theorem 1 proves that after $(1+1/\alpha)p-2$ communication rounds, processor P_p finishes its work. Since we are essentially computing the similarity sequentially in each processor and sending the boundaries to the right processor, the correctness of the algorithm comes naturally from the correctness of the sequential algorithm. Then, after $(1+1/\alpha)p-2$ communication rounds, $S(m,n)$ will store the similarity between the strings A and C. ⋆

3 Implementation

$4K \times 8K$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$
$\alpha=2$	3.9790	2.2244	1.2223	0.7007	0.4281
$\alpha=1$	2.9637	1.6294	0.8986	0.5373	0.3426
$\alpha=1/2$	2.5599	1.3295	0.7320	0.4353	0.3112
$\alpha=1/4$	2.1977	1.1891	0.6680	0.4199	0.2938
$\alpha=1/8$	2.0660	1.1224	0.6452	0.4067	0.3367
$\alpha=1/16$	2.0197	1.0857	0.6310	0.4298	0.3637
$\alpha=1/32$	1.9956	1.0841	0.6493	0.4632	0.4852
$\alpha=1/64$	1.9840	1.0964	0.6996	0.5668	

$8K \times 16K$	$p=4$	$p=8$	$p=16$	$p=32$	$p=64$
$\alpha=2$	16.000	8.6738	4.7117	2.5494	1.4172
$\alpha=1$	11.622	6.2732	3.3209	1.8213	1.0718
$\alpha=1/2$	9.5802	5.0730	2.6848	1.4811	0.9023
$\alpha=1/4$	8.5727	4.4721	2.3604	1.3726	0.8306
$\alpha=1/8$	8.0455	4.1770	2.2151	1.3349	0.8107
$\alpha=1/16$	7.7996	4.0530	2.1522	1.2794	0.8681
$\alpha=1/32$	7.7079	3.9948	2.1469	1.3295	0.9656
$\alpha=1/64$	7.6800	3.9857	2.1891	1.4127	1.1525

We have implemented the $O(p)$ rounds similarity algorithm on a Beowulf with 64 nodes. Each node has 256 MB of RAM memory in addition to 256 MB for swap. The nodes are connected through a 100 MB interconnection network.

The above tables show running times for string sizes $m = 4K$, $n = 8K$ and $m = 8K$, $n = 16K$ where $K = 1024$. They show that with *very small* α, the communication time is significant when compared to the computation time. We have analyzed the behavior of α to estimate the optimal block size. The observed times show that when $\alpha\frac{m}{p}$ decreases from 16 to 8 (the number of lines of the submatrix S_i^k), we have an increase on the total time.

The Figure 4 depicts the obtained results of the $O(p)$ communication rounds algorithm applied to strings A ($|A| = 4096$) and C ($|C| = 8192$), with $\alpha = \{1, 1/2, 1/4, \ldots, 1/64\}$.

In general, the implementation of the CGM/BSP algorithm shows that the theoretical results are confirmed in the implementation.

4 Conclusion

We have presented a parameterized CGM/BSP parallel algorithm with $O(p)$ communication rounds to compute the score of the similarity between two strings. In this paper we have worked with a variable block size of $\alpha\frac{m}{p} \times \frac{n}{p}$. We have studied the behavior of the block size. Once we have studied how the algorithm works with different block sizes, we intend to explore this result and try an *adaptative choice of the optimal block size*.

The alignment between the two strings can be obtained with $O(p)$ communication rounds backtracking from the lower right corner of the grid graph in $O(m+n)$ time [13]. For this, $S(r,s)$ for all points of the grid graph must be

stored during the computation (requiring $O(mn)$ space). A slightly different algorithm which uses only $O(\min\{m,n\})$ space is also being implemented on the CGM/BSP model.

Using the Monge properties [3] of the grid DAG, Alves *et al.* [1] have proposed an $O(\log p)$ communication rounds CGM/BSP dynamic programming algorithm for solving the string editing problem between a string A and all substrings of a string C. We are working with the implementation details on this problem. Furthermore, we intend to explore the above ideas to solve the multiple alignment problem.

References

[1] C. Alves, E. N. Cáceres, F. Dehne, and S. W. Song. Parallel dynamic programming for solving the string editing problem on a cgm/bsp. In *Proceedings SPAA'02 - 14th ACM Symposium on Parallel Algorithms and Architectures*. ACM PRESS, accepted, 2002.

[2] C. Alves, E. N. Cáceres, F. Dehne, and S. W. Song. A cgm/bsp parallel similarity algorithm. submitted, 2002.

[3] A. Apostolico, L. L. M.J. Atallah, and S. Macfaddin. Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput.*, 19(5):968–988, 1990.

[4] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proc. ACM 9th Annual Computational Geometry*, pages 298–307, 1993.

[5] F. Dehne (Editor). Coarse grained parallel algorithms. *Special Issue of Algorithmica*, 24(3/4):173–176, 1999.

[6] Z. Galil and K. Park. Parallel dynamic programming. Technical Report CUCS-040-91, Columbia University-Computer Science Dept., 1991.

[7] Z. Galil and K. Park. Dynamic programming with convexity, concavity ans sparsity. *Theoretical Computer Science*, pages 49–76, 1992.

[8] M. Gengler. An introduction to parallel dynamic programming. *Lecture Notes in Computre Science*, 1054:87–114, 1996.

[9] P. Hall and G. Dowling. Approximate string matching. *Comput. Surveys*, (12):381–402, 1980.

[10] J. Hunt and T. Szymansky. An algorithm for differential file comparison. *Comm. ACM*, (20):350–353, 1977.

[11] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Bio.*, (48):443–453, 1970.

[12] P. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, (1):359–373, 1980.

[13] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[14] T. Smith and M. Waterman. Identification of common molecular subsequences. *J. Mol. Bio.*, (147):195–197, 1981.

[15] L. Valiant. A bridging model for parallel computation. *Communication of the ACM*, 33(8):103–111, 1990.

[16] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. ACM*, (35):83–91, 1992.