A Parallel Wavefront Algorithm for Efficient Biological Sequence Comparison*

C. E. R. Alves[1], E. N. Cáceres[2], F. Dehne[3], and S. W. Song[4]

[1] Universidade São Judas Tadeu, São Paulo, Brazil,
prof.carlos_r_alves@usjt.br,
http://www.usjt.br/
[2] Universidade Federal de Mato Grosso do Sul, Campo Grande, Brazil,
edson@dct.ufms.br,
http://www.dct.ufms.br/∼edson
[3] Carleton University - School of Computer Science,
Ottawa, Canada K1S 5B6,
frank@dehne.net,
http://www.dehne.net
[4] Universidade de São Paulo, São Paulo, Brazil,
song@ime.usp.br,
http://www.ime.usp.br/∼song

Abstract. In this paper we present a parallel wavefront algorithm for computing an alignment between two strings A and C, with $|A| = m$ and $|C| = n$. On a distributed memory parallel computer of p processors each with $O((m + n)/p)$ memory, the proposed algorithm requires $O(p)$ communication rounds and $O(mn/p)$ local computing time. The novelty of this algorithm is based on a compromise between the workload of each processor and the number of communication rounds required, expressed by a parameter called α. The proposed algorithm is expressed in terms of this parameter that can be tuned to obtain the best overall parallel time in a given implementation. We show very promising experimental results obtained on a 64-node Beowulf machine. A characteristic of the wavefront communication requirement is that each processor communicates with few other processors. This makes it very suitable as a potential application for grid computing.

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

* The second author was supported by CNPq, FINEP-PRONEX-SAI Proc. No. 76.97.1022.00 and FAPESP Proc. No. 1997/10982-0, the third author by the Natural Sciences and Engineering Research Council of Canada, and the fourth author by FAPESP Proc. No. 99/07390-0, CNPq Proc. No. 52.3778/96-1, 46.1230/00-3, 52.1097/01-0 and 52.2028/02-9.

biosequences [11, 12, 14], as well as in the solution of several other problems such as approximate string matching, file comparison, and text searching with errors [9, 10, 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 become equal in length. 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 similar.

In this paper we extend and improve a previous result [2] for computing an alignment between two strings A and C, with $|A| = m$ and $|C| = n$. On a distributed memory parallel computer of p processors each with $O((m+n)/p)$ memory, the algorithm proposed in this paper requires $O(p)$ communication rounds and $O(mn/p)$ local computing time. Both in [2] as in this paper the processors communicate in a *wavefront* or *systolic* manner, such that each processor communicates with few other processors. Actually each processor sends data to only two other processors. The novelty of the algorithm proposed in this paper 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 that can be tuned to obtain the best overall parallel time in a given implementation. In addition to showing theoretic complexity we confirm the efficiency of the proposed algorithm through implementation. Very promising experimental results are obtained on a 64-node Beowulf machine.

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

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].

The proposed algorithm uses the realistic BSP/CGM model. Observe that it is simple to implement as opposed to another more complex BSP/CGM algorithm proposed in [1] that is based on the Monge properties [3] of the grid DAG. A characteristic and advantage of the wavefront or systolic communication requirement used in this algorithm is that each processor communicates with few other processors. This makes it very suitable as a potential application for grid computing.

2 The Similarity Problem

We now define the similarity problem we wish to solve in this paper. Let $A = a_1a_2\ldots a_m$ and $C = c_1c_2\ldots c_n$ be two strings over some alphabet I.

A	a c t t c a − t	
C	a t t c − a c g	
Score	1 0 1 0 0 1 0 0	3

A	a c t t c a − t	
C	a − t t c a c g	
Score	1 0 1 1 1 1 0 0	5

Fig. 1. Examples of alignment

To align the two strings, we insert spaces in the two sequences in such way that they become equal in length. See Fig. 1 where each column consists of a symbol of A (or a space) and a symbol of C (or a space). 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. Fig. 1 shows two simple alignment examples where we assign a score of 1 when the aligned symbols in a column match and 0 otherwise. The alignment on the right has a higher score (5) than that on the left (3).

A more general score assignment for a given alignment between strings is done 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. Consider a column consisting of symbols r and s. If $r = s$ (i.e. a *match*), it will receive a value $p(r, s) > 0$. If $r \neq s$ (a *mismatch*), the column will receive a value $p(r, s) < 0$. Finally, a column with a space in it receives a value $-k$, where $k \in N$. We look for the alignment (*optimal alignment*) that 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].

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 $A[1..r]$ and $C[1..s]$, that denote the prefixes $a_1 a_2 \ldots a_r$ and $c_1 c_2 \ldots c_s$, respectively.

Observe that we can compute the values of $S(r, s)$ by using the three previous values $S(r - 1, s)$, $S(r - 1, s - 1)$ 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 score S of the alignment between strings A and C can be computed as follows:

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

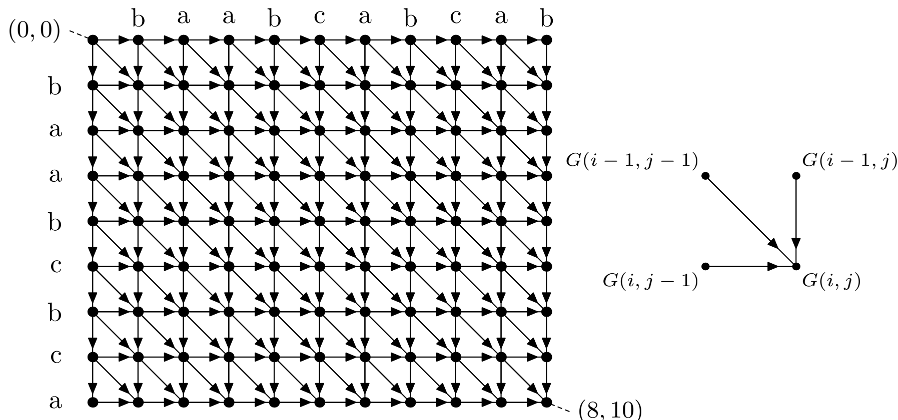

Fig. 2. Grid DAG G for A = baabcbca and B = baabcabcab.

An $l_1 \times l_2$ *grid DAG* (Fig. 2) is a directed acyclic graph whose vertices are the $l_1 l_2$ points of an $l_1 \times l_2$ grid, with edges from grid point $G(i, j)$ to the grid points $G(i, j+1)$, $G(i+1, j)$ and $G(i+1, j+1)$.

Let A and C be two strings with $|A| = m$ and $|C| = n$ symbols, respectively. 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. In Fig. 2 the problem is to find the minimum path from (0,0) to (8,10).

3 Parallel Computation

Valiant [15] introduced a simple *coarse granularity* model, called *Bulk Synchronous Parallel Model - BSP*. It gives reasonable predictions on the performance of the algorithms when implemented on existing, mainly distributed memory, parallel machines. A BSP algorithm consists of a sequence of super-steps separated by *synchronization barriers*. In a super-step, each processor executes a set of independent operations using local data available in each processor at the start of the super-step, as well as communication consisting of send and receive of messages. An *h-relation* in a super-step corresponds to sending or receiving at most h messages in each processor.

In this paper we use a similar model called 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. A CGM algorithm is a special case of a BSP algorithm where all the communication operations of one super-step 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 super-steps and the amount of local computation.

4 The Wavefront Parameterized 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 $O(mn/p)$ local memory. As will be seen later, this can be reduced to $O((m+n)/p)$.

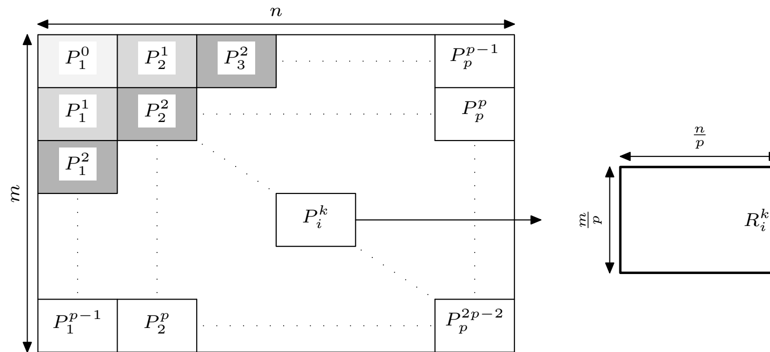

Fig. 3. An $O(p)$ communication rounds scheduling with $\alpha = 1$

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, $1 \le i \le p$, receives the i-th piece of C $(c_{(i-1)\frac{n}{p}+1} \dots c_{i\frac{n}{p}})$.

The scheduling scheme can be illustrated in Fig. 3. 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, P_1, P_2 and P_3 at round 2, and so on. In other words, after computing the k-th part of the sub-matrix S_i (denoted S_i^k), 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 sub-matrix S_{i+1}. After $p-1$ rounds, processor P_p receives R_{p-1}^1 and computes the first part of the sub-matrix S_p. In the $2p-2$ round, processor P_p receives R_{p-1}^p and computes the p-th part of the sub-matrix S_p and finishes the computation.

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

In the following we attempt to assign 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 $\frac{m}{p}$ we consider sizes $\alpha\frac{m}{p}$ 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 sub-matrix 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, processor P_i sends R_i^k to processor P_{i+1}. Processor P_{i+1} receives R_i^k from P_i and computes S_{i+1}^{k+1}. After $p-2$ rounds, 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. Fig. 4 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 Fig. 4, 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.*

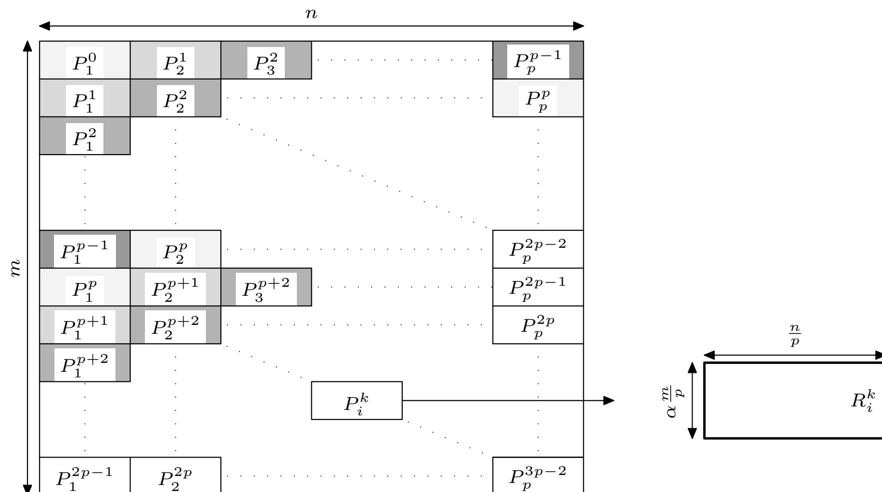

Fig. 4. An $O(p)$ rounds scheduling with $\alpha = 1/2$

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 $\frac{mn}{p}$ sub-matrix 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 sub-matrix 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.*

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.

5 Implementation Results

We have implemented the proposed 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.

5.1 Parameter Tuning

$8K \times 16K$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
$\alpha = 2$	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

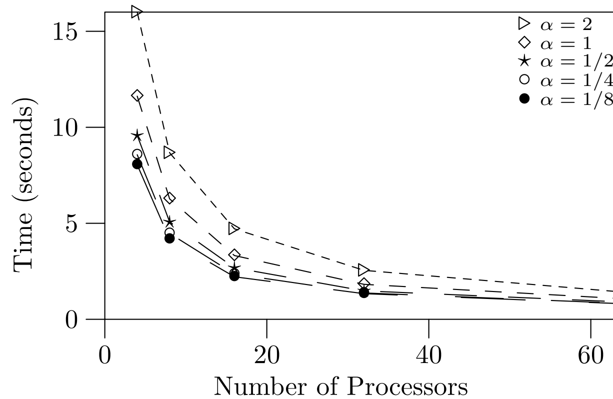

Fig. 5. Curves of the observed times for $m = 8196$ and $n = 16384$

The table shows running times for string sizes $m = 8K$ and $n = 16K$ where $K = 1024$. The same times are shown in Fig. 5. 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 sub-matrix S_i^k), we have an increase on the total time. The best times are obtained for α between 1/4 and 1/8.

5.2 Quadratic versus Linear Space Implementation

We can further improve our results by exploring a linear space implementation, by storing a vector instead of the entire matrix. In the usual quadratic space implementation, each processor uses $O(mn/p)$ space, while in the linear space implementation each processor requires only $O((m+n)/p)$ space. The results are impressive, as shown in Figures 6 and 7. With less demand on the swap of disk space, we get an almost 50% improvement. We have used $\alpha = 1$.

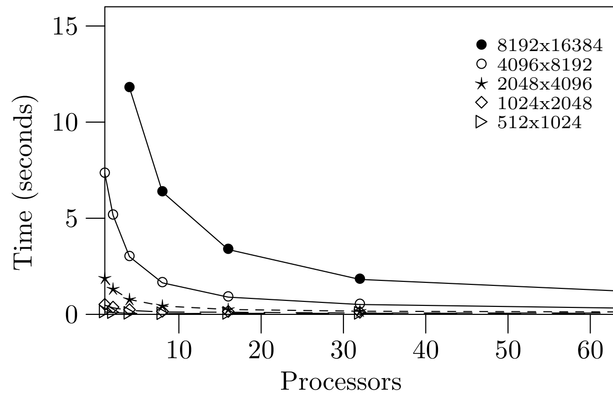

Fig. 6. Curves of the observed times - quadratic space

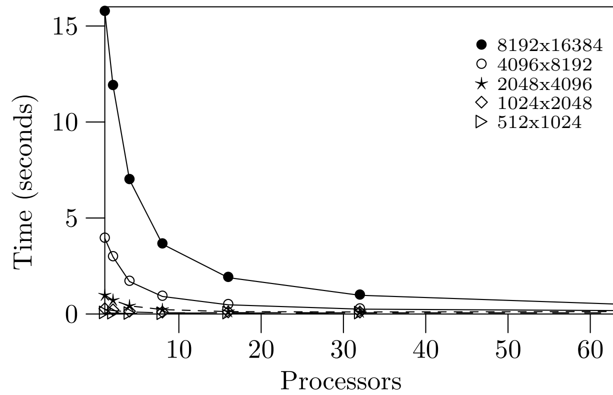

Fig. 7. Curves of the observed times - linear space

6 Conclusion

We have presented a parameterized BSP/CGM parallel algorithm to compute the score of the similarity between two strings. On a distributed memory parallel computer of p processors each with $O((m+n)/p)$ memory, the proposed algorithm requires $O(p)$ communication rounds and $O(nm/p)$ local computing time. The novelty of this 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. We have worked with a variable block size of $\alpha \frac{m}{p} \times \frac{n}{p}$ and studied the behavior of the block size. We show how this parameter can be tuned to obtain the best overall parallel time in a given implementation. Very promising experimental results are shown.

As a final observation notice that a characteristic of the wavefront communication requirement is that each processor communicates with few other processors. This makes it very suitable as a potential application for grid computing.

References

1. C. E. R. 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 of the 14th Symposium on Parallel Algorithms and Architectures (ACM-SPAA)*, pages 275–281. ACM Press, 2002.
2. C. E. R. Alves, E. N. Cáceres, F. Dehne, and S. W. Song. A CGM/BSP parallel similarity algorithm. In *Proceedings of the I Brazilian Workshop on Bioinformatics*, pages 1–8, october 2002.
3. A. Apostolico, L. L. Larmore 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 and 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. A. Hall and G. R. Dowling. Approximate string matching. *Comput. Surveys*, (12):381–402, 1980.
10. J. W. Hunt and T. Szymansky. An algorithm for differential file comparison. *Comm. ACM*, (20):350–353, 1977.
11. S. B. Needleman and C. D. 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. H. 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. F. Smith and M. S. 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.