A CGM/BSP Parallel Similarity Algorithm*

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

[1] FTCE - Universidade São Judas Tadeu, São Paulo, SP - Brazil,
`prof.carlos_r_alves at usjt. br`
[2] Universidade Federal de Mato Grosso do Sul, Campo Grande, MS - Brazil,
`edson at dct. ufms. br`
[3] School of Computer Science, Carleton University, Ottawa, Canada,
`frank at dehne. net`
[4] Universidade de São Paulo, São Paulo, SP - Brazil,
`song at ime. usp. br`

Abstract. We present a CGM/BSP algorithm for computing an alignment (or string editing) between two strings A and C, with $|A| = m$ and $|C| = n$. The algorithm 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. We also present implementation results obtained on Beowulf machine with 64 nodes.

Topic of interest: Algorithms and applications in Bioinformatics

1 Introduction

Sequence comparison is a fundamental problem in Computational Biology that appears in more complex problems [12], such as the search of similarities between biosequences [10, 11, 13]. Furthermore, sequence comparison can be used in the solution of several other problems [9, 8, 15].

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. We expect that the alignment of two sequences that are similar will show the parts where they match, and different parts where spaces are inserted. 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.

The same idea can be applied when we want to find the distance between two sequences. 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. Thus, the distance is a measure of how much the strings differ.

* Partial support of E. N. Cáceres by CNPq, FINEP-PRONEX-SAI Proc. No. 76.97.1022.00 and FAPESP Proc. No. 1997/10982-0, F. Dehne by the NSERC of Canada, and S. W. Song by FAPESP Grant 99/07390-0, CNPq grants 52.3778/96-1, 46.1230/00-3, and 521097/01-0, and CNPq/NSF grant 68.0037/99-3.

Let $A = a_1 a_2 \dots a_m$ and $C = c_1 c_2 \dots c_n$ 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 [12].

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 $O(m + n)$ [12].

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 $A[1 \dots r]$ and $C[1 \dots s]$.

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 $A[1 \dots r]$ and $C[1 \dots s]$. 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 $l_1 l_2$ 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 $(i+1, j+1)$.

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.

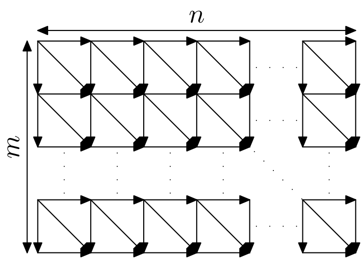

Fig. 1. Grid DAG G

One way to explore the use of parallel computing is 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 [5, 6]. PRAM algorithms for the string editing problem have been proposed by Apostolico et al. [2]. A more general study of parallel algorithms for dynamic programming can be seen in [7]. 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.

Valiant [14] introduced a simple *coarse granularity* model, called *Bulk Synchronous Parallel Model - BSP*. It gives reasonable predictions on the performance of the algorithms 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* [3]. 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 [4]. 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. To our knowledge there are no implemented results using the BSP/CGM model for sequence similarity problem. We have implemented it on a Beowulf with 64 nodes with very promising results.

2 An $O(p)$ Communication Rounds Alignment Algorithm

Let $A = \{a_1 \dots a_m\}$ and $C = \{c_1 \dots c_n\}$ be two strings on some alphabet I. We will design a parallel algorithm to compute the similarity between A and C on a *CGM/BSP* with p processors and $\frac{mn}{p}$ local memory in each processor. Using this result, we can find an optimal alignment between A and C.

The sequential algorithms that solve efficiently this problem use the technique of dynamic programming, solving an instance of the problem by taking advantage of already computed solutions for smaller instances of the same problem [12].

To solve the similarity problem on the CGM/BSP model, we divide C into p pieces, of size $\frac{n}{p}$, and each processor P_i, $1 \leq i \leq p$, receives the string A and the i-th piece of C $(c_{(i-1)\frac{n}{p}+1} \dots c_{i\frac{n}{p}})$.

Each processor P_i computes the elements $S_i(r, s)$ of the submatrix S_i, where $1 \leq r \leq m$ and $(i-1)\frac{n}{p} + 1 \leq s \leq i\frac{n}{p}$ using just three previous elements $S_i(r-1, s)$, $S_i(r-1, s-1)$ and $S_i(r, s-1)$, because there are only three ways of computing an alignment between $A[1 \dots r]$ and $C[1 \dots s]$. 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 $A[r]$.

To compute the submatrix S_i, each processor P_i uses the best sequential algorithm locally. It is easy to see that processor P_i, $i > 1$, can only start computing the elements $S_i(r, s)$ after the processor P_{i-1} has computed part of the submatrix $S_{i-1}(r, s)$.

Denote by R_i^k, $1 \leq i, k \leq p$, all the elements of the right boundary (rightmost column) of the k-th part of the submatrix S_i. More precisely, $R_i^k = \{S_i(r, i\frac{n}{p}), (k-1)\frac{m}{p} + 1 \leq r \leq k\frac{m}{p}\}$.

The idea of the algorithm is the following: After computing the k-th part of the submatrix S_i, the processor P_i sends to processor P_{i+1} the elements of 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.

Using this schedule (Figure 2), 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 $1 \leq i \leq k$.

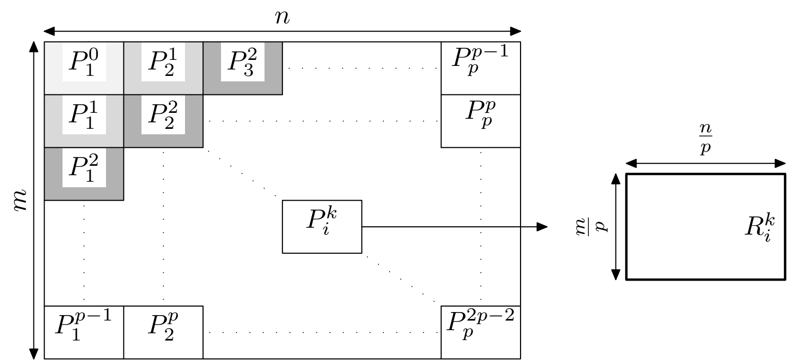

Fig. 2. An $O(p)$ communication rounds scheduling

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.
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** $1 \leq k \leq p$
  (1.1) **if** $i = 1$ **then**
    (1.1.1) **for** $(k-1)\frac{m}{p}+1 \leq r \leq k\frac{m}{p}$ **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** $(k-1)\frac{m}{p}+1 \leq r \leq k\frac{m}{p}$ **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 —

Theorem 1. *We can solve the problem using* $2p-2$ *communication rounds with* $O(\frac{mn}{p})$ *sequential computing time in each processor.*

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

Proof. Theorem 1 proves that after $2p - 2$ 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 $2p - 2$ communication rounds, $S(m, n)$ will store the similarity between the strings A and C.

3 Implementation

$m \times n$	1	2	4	8	16	32	64
512×512	0.0558	0.0377	0.0220	0.0138	0.0436	0.0459	0.0483
512×1024	0.1135	0.0787	0.0549	0.0541	0.0449	0.0527	0.0574
1024×1024	0.2246	0.1546	0.0894	0.0705	0.0629	0.0502	0.0621
1024×2048	0.4556	0.3355	0.2032	0.1208	0.1040	0.0926	0.0730
2048×2048	0.9057	0.6588	0.4005	0.2365	0.1526	0.1129	0.1040
2048×4096	1.9459	1.3030	0.7691	0.4316	0.2508	0.1633	0.1272
4096×4096	3.7533	2.5544	1.5285	0.8479	0.5058	0.4569	0.2387
4096×8192	7.5440	5.1274	2.9868	1.6403	0.8875	0.5091	0.3312
8192×8192			5.8492	3.2054	1.7659	1.0096	0.6237
8192×16384				6.3080	3.3550	1.8139	1.0178

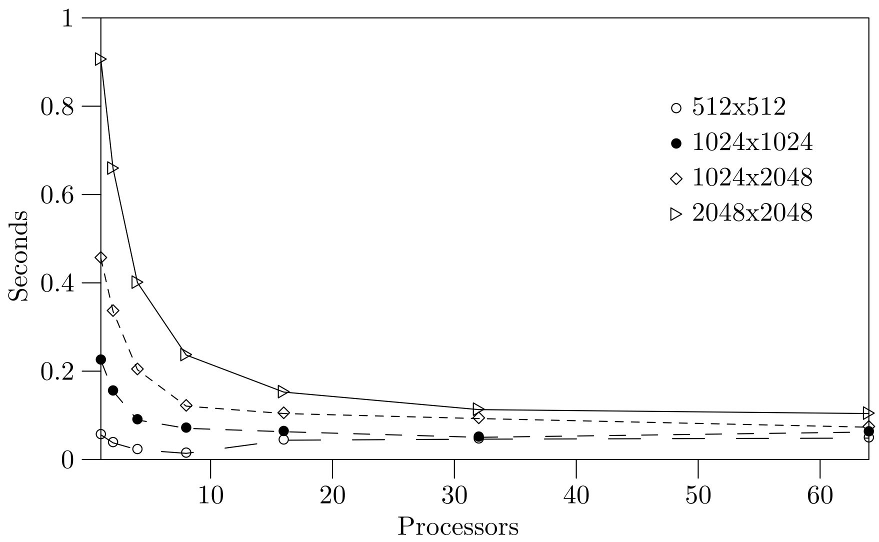

Fig. 3. Curves of the observed times

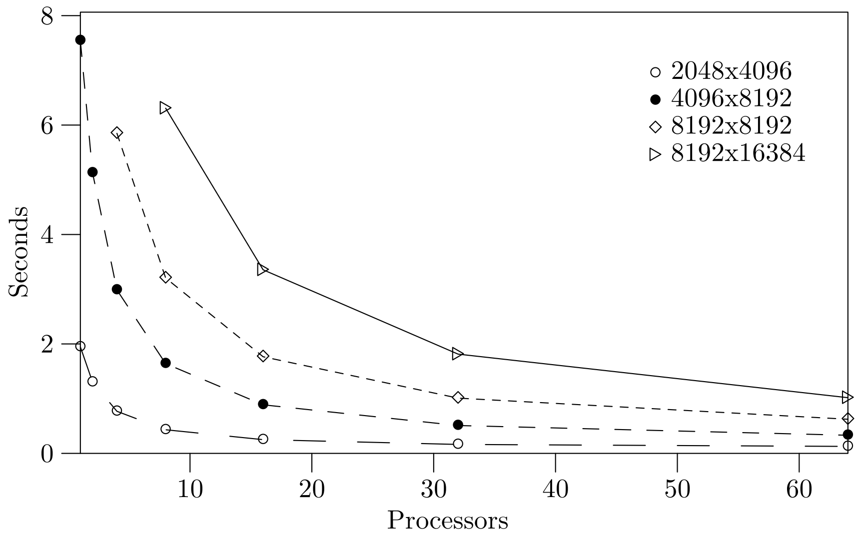

Fig. 4. Curves of the observed times

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

The obtained times show that with *small* sequences, the communication time is significant when compared to the computation time with more than 8 and 16 processors, respectively (512×512 and 512×1024). When we apply the algorithm to sequences greater than 8192, using one or two processors, the main memory is not enough to solve the problem. The utilization of swap gives us meaningless resulting times. This would not occur if the nodes have more main memory. Thus we have suppressed these times.

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 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 fixed block size of $\frac{m}{p} \times \frac{n}{p}$. We are currently working with an *adaptative choice of the optimal block size* to further decrease the running time of the algorithm.

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 [12]. 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 [2] 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.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 SPAA'02 - 14th ACM Symposium on Parallel Algorithms and Architectures*, page accepted. ACM PRESS, 2002.
2. 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.
3. 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.
4. F. Dehne (Ed.). Coarse grained parallel algorithms. *Special Issue of Algorithmica*, 24(3/4):173–176, 1999.
5. Z. Galil and K. Park. Parallel dynamic programming. Technical Report CUCS-040-91, Columbia University-Computer Science Dept., 1991.
6. Z. Galil and K. Park. Dynamic programming with convexity, concavity ans sparsity. *Theoretical Computer Science*, pages 49–76, 1992.
7. M. Gengler. An introduction to parallel dynamic programming. *Lecture Notes in Computre Science*, 1054:87–114, 1996.
8. P.A. Hall and G.R. Dowling. Approximate string matching. *Comput. Surveys*, (12):381–402, 1980.
9. J.W. Hunt and T. Szymansky. An algorithm for differential file comparison. *Comm. ACM*, (20):350–353, 1977.
10. 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.
11. P.H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, (1):359–373, 1980.
12. J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.
13. T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *J. Mol. Bio.*, (147):195–197, 1981.
14. L. Valiant. A bridging model for parallel computation. *Communication of the ACM*, 33(8):103–111, 1990.
15. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. ACM*, (35):83–91, 1992.