Parallel Dynamic Programming for Solving the String Editing Problem on a CGM/BSP

C. E. R. Alves
FTCE - Universidade São Judas Tadeu
São Paulo, SP - Brazil
prof.carlos_r_alves at usjt. br

E. N. Cáceres*
Universidade Federal de Mato Grosso do Sul
Campo Grande, MS - Brazil
edson at dct. ufms. br

F. Dehne†
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
frank at dehne. net

ABSTRACT

In this paper we present a coarse-grained parallel algorithm for solving the string edit distance problem for a string A and all substrings of a string C. Our method is based on a novel CGM/BSP parallel dynamic programming technique for computing all highest scoring paths in a weighted grid graph. The algorithm requires $\log p$ rounds/supersteps and $O(\frac{n^2}{p}\log m)$ local computation, where p is the number of processors, $p^2 \leq m \leq n$. To our knowledge, this is the first efficient CGM/BSP algorithm for the alignment of all substrings of C with A. Furthermore, the CGM/BSP parallel dynamic programming technique presented is of interest in its own right and we expect it to lead to other parallel dynamic programming methods for the CGM/BSP.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms

Keywords

Parallel algorithms, string editing, dynamic programming, CGM, BSP

*Partially supported by CNPq and FINEP-PRONEX-SAI Proc. No. 76.97.1022.00.
†Partially supported by the Natural Sciences and Engineering Research Council of Canada.

SPAA'02, August 10-13, 2002, Winnipeg, Manitoba, Canada.

1. INTRODUCTION

Molecular Biology is an important field of application for parallel computing. Sequence comparison is among the fundamental tools in Computational Molecular Biology and is used to solve more complex problems [14], including the computation of similarities between biosequences [11, 13, 15]. Beside such Molecular Biology applications, sequence comparison is also used in several other applications [8, 9, 17]. The notions of similarity and distance are, in most cases, interchangeable and both of them can be used to infer the functionality or the aspects related to the evolutive history of the evolved sequences. In either case we are looking for a numeric value that measures the degree by which the sequences are alike or different.

We now give a formal definition of the *string editing* problem. Let A be a string with $|A|$ symbols on some fixed size alphabet Σ. In this string we can do the following edit operations: deletion, insertion and substitution. Each edit operation is assigned a non negative real number representing the cost of the operation: $D(x)$ for deletion of a symbol x; $I(x)$ for insertion of a symbol x and $T(x, y)$ for the exchange of the symbol x with the symbol y. An *edit sequence* σ is a sequence of editing operations and its cost is the sum of the costs of its operations. Let A and C be two strings with $|A| = m$ and $|C| = n$ symbols, respectively, with $m < n$. The *string editing problem* for input strings A and C consists of finding an edit sequence σ' of minimum cost that transforms A into C.

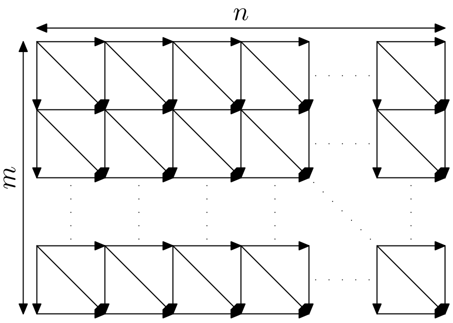

Figure 1: Grid DAG G

The cost of σ' is the *edit distance* from A to C. Let $E(i, j)$ be the minimum cost of transforming the prefix of

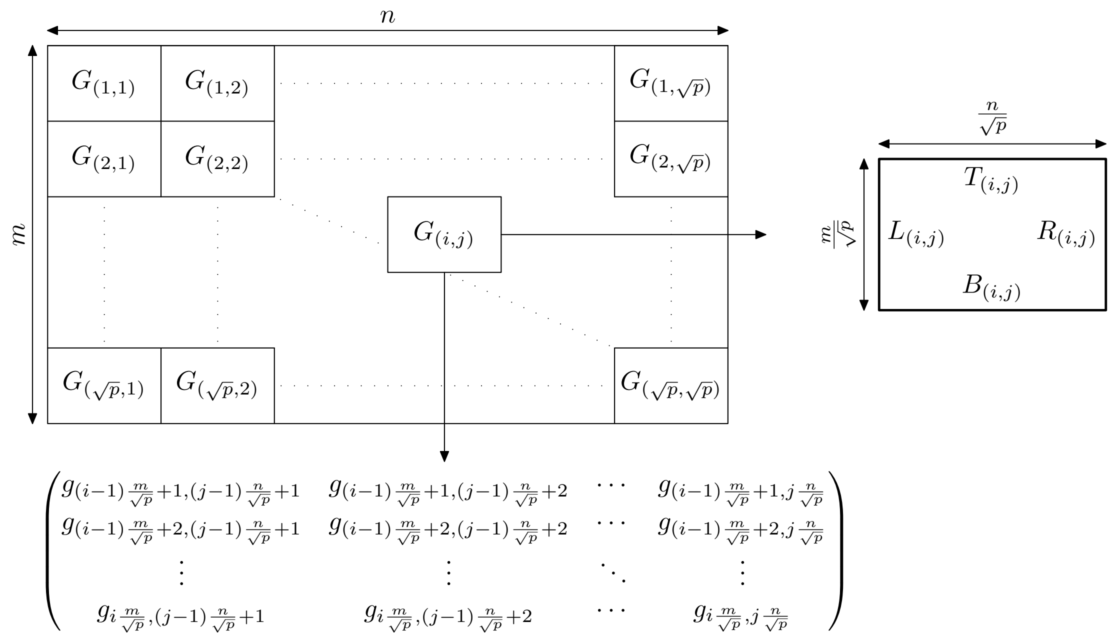

Figure 2: Processor $P_{(i,j)}$ Stores the Submatrix $G_{(i,j)}$

A of length i into the prefix of C of length j, $0 \leq i \leq m, 0 \leq j \leq n$. It follows that $E(i,j) = \min\{E(i-1,j-1)+T(x_i,y_j), E(i-1,j)+D(x_i), E(i,j-1)+I(y_j)\}$ for all $i,j, 1 \leq i \leq m, 1 \leq j \leq n$.

It is easy to see that the string editing problem can be modeled by a grid graph [1, 12] (Figure 1). An (m,n) *grid graph* $G = (V,E)$ is a directed acyclic weighted graph whose vertices are $(m+1) \times (n+1)$ points of the grid, with rows $0 \dots m$ and columns $0 \dots n$. Vertex (i,j) has a directed edge to $(i+1,j)$, $(i+1,j+1)$, and $(i,j+1)$, provided that these endpoints are within the boundaries of the grid [12].

In [14] the authors describe how to obtain the similarity (alignment) between two strings by using string editing. Assuming a similarity score that satisfies the triangle inequality, the similarity problem can be solved by computing the largest source-sink path in the weighted directed acyclic graph G (grid dag) that corresponds to the edit sequence which transforms A into C.

The standard sequential algorithms for the string editing problem are based on *dynamic programming*. The complexity of these algorithms is $O(mn)$ time. Given the similarity matrix, the construction of the optimal alignment can be done in $O(m+n)$ sequential time [14]. Parallel dynamic programming is a well studied topic. Efficient parallel PRAM algorithms for dynamic programming have been presented by Galil and Park [5, 6]. PRAM algorithms for the string editing problem have been proposed by Apostolico et al [1]. A general study of parallel algorithms for dynamic programming can be found in [7].

In this paper we study parallel dynamic programming for the string editing problem using the BSP [16] *Coarse Grained Multicomputer* (CGM) [3, 4] model. A CGM consists of a set of p processors $P_1, \dots, P_p$ with $O(N/p)$ local memory per processor, where N is the space needed by the sequential algorithm. Each processor is connected by a router that can send messages in a point-to-point fashion. A CGM algorithm consists of alternating local computation and global communication rounds separated by a barrier synchronization. A round is equivalent to a superstep in the BSP model. Each communication round consists of routing a single h-relation with $h = O(N/p)$. We require that all information sent from a given processor to another processor in one communication round be packed into one long message, thereby minimizing the message overhead. In the CGM model, the communication cost is modeled by the number of communication rounds. The main advantage of BSP/CGM algorithms is that they map very well to standard parallel hardware, in particular Beowulf type processor clusters [4].

The main concern is on the communication requirements. Our goal is to minimize the number of communication rounds.

We present a CGM/BSP algorithm for solving the string edit distance problem for a string A and all substrings of a string C via parallel dynamic programming. An $O(n^2 \log m)$ sequential algorithm was presented in [12] (to solve the *all approximate repeats in strings* problem). This problem also arises in the common substring alignment problem [10]. The method requires $\log p$ rounds/supersteps and $O(\frac{n^2}{p} \log m)$ local computation. To our knowledge, this is the first efficient CGM/BSP algorithm for this problem.

Furthermore, the CGM/BSP parallel dynamic programming technique presented is of interest in its own right. We expect that our result will lead to other parallel dynamic programming methods for the CGM/BSP.

2. A CGM ALGORITHM FOR COMPUTING ALL HIGHEST SCORING PATHS IN $O(\log P)$ ROUNDS

In this section we present a parallel algorithm for computing all highest scoring paths (AHSP) in weighted (m,n) grid graphs using a CGM with p processors and $\frac{mn}{p}$ local memory per processor. Using this method, we can find an optimal alignment between A and C.

We divide the grid graph G into p subgrids $G_{(i,j)}$, $1 \leq i,j \leq \sqrt{p}$, of size $(\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}})$ and each processor $P_{(i,j)}$ stores subgrid $G_{(i,j)}$ (Figure 2).

Let the left boundary, L, of G be the set of points in the leftmost column. The right, top and bottom boundaries, R, T and B, respectively, are defined analogously. The boundary of G is the union of its left, right, top, and bottom boundaries ($L \cup R \cup T \cup B$).

Let $DIST_{G_{(i,j)}}$ be a $\frac{m+n-1}{\sqrt{p}} \times \frac{m+n-1}{\sqrt{p}}$ matrix containing the lengths of all shortest paths that begin at the left ($L_{(i,j)}$) or top ($T_{(i,j)}$) boundary of $G_{(i,j)}$, and end at the right ($R_{(i,j)}$) or bottom ($B_{(i,j)}$) boundary of $G_{(i,j)}$. The matrix $DIST_{G_{(i,j)}}$ consists of four submatrices $L_{(i,j)} \times R_{(i,j)}$ (storing all the shortest paths that begin at the left boundary and end at the right boundary of $G_{(i,j)}$), $L_{(i,j)} \times B_{(i,j)}$ (storing all the shortest paths that begin at the left boundary and end at the bottom boundary of $G_{(i,j)}$), $T_{(i,j)} \times R_{(i,j)}$ (storing all the shortest paths that begin at the top boundary and end at the right boundary of $G_{(i,j)}$) and $T_{(i,j)} \times B_{(i,j)}$ (storing all the shortest paths that begin at the left boundary and end at the bottom boundary of $G_{(i,j)}$). Using the algorithm of Schmidt [12], each processor can compute all distances of the paths from the left and top boundaries to the right and bottom boundaries in $G_{(i,j)}$ in time $O(\frac{mn}{p} \log \frac{m}{\sqrt{p}})$.

The general strategy of our CGM/BSP algorithm is as follows: In the general step of the algorithm, several processors collaborate to join previously calculated subgrids. At the beginning of each step, each subgrid has a distance matrix distributed among a group of processors. Two neighbor grids are joined by the processors that hold the two distance matrices, resulting in a new distance matrix distributed among these processors. Each step of the algorithm reduces by a factor of 1/2 the number of subgrids remaining to be merged.

2.1 Joining Grids

We will now show a sequential algorithm to join two adjacent grids with a common horizontal boundary. The case of a common vertical boundary is analogous. In the next subsection we will show how the distance matrices of two grids of size $l \times k$, each stored in q processors, can be used by the $2q$ processors to build the distance matrix of the $(2l-1) \times k$ size merged grid. This procedure takes time $O((l+k)^2/q)$ (provided that q is small compared to l and k) and a constant number of communication rounds. Each round transfers $O((l+k)^2/q)$ data from/to each processor and the local memory required by each processor is $O((l+k)^2/q)$.

For simplicity, we will refer the upper grid as G_u, with boundaries L_u, T_u, B_u and R_u, and the lower grid as G_l, with boundaries L_l, T_l, B_l and R_l. We will refer to the distance matrices for the upper, lower and final grids as $DIST_u$, $DIST_l$ and $DIST_{ul}$, respectively. It is important to note that the size of the resulting distance matrix can be different from the total size of the two original distance matrices. However, when four grids are joined in a 2×2 configuration the sizes add up precisely.

Let $t = l + k - 1$. Each initial distance matrix is stored as a $t \times t$ matrix evenly distributed among q processors (the -1 in the definition of t accounts for the top left and bottom right corners). The q processors that store $DIST_u$ ($P_{u1}, P_{u2}, \ldots, P_{uq}$) will store consecutive *columns* of $DIST_u$. The q processors that store $DIST_l$ ($P_{l1}, P_{l2}, \ldots, P_{lq}$) will store consecutive *rows* of $DIST_l$. Note that the distance matrices are actually banded matrices and that a great portion of these matrices will not be involved in the joining operation.

Figure 3 illustrates which parts of the old matrices are copied and which parts are used to build the new matrix. The copied parts will require some redistribution at the end of the step. We will concentrate on calculating the $t \times t$ submatrix of $DIST_{ul}$. The shaded areas illustrate the regions where no paths exist. The submatrices effectively involved in the calculations have a thicker border.

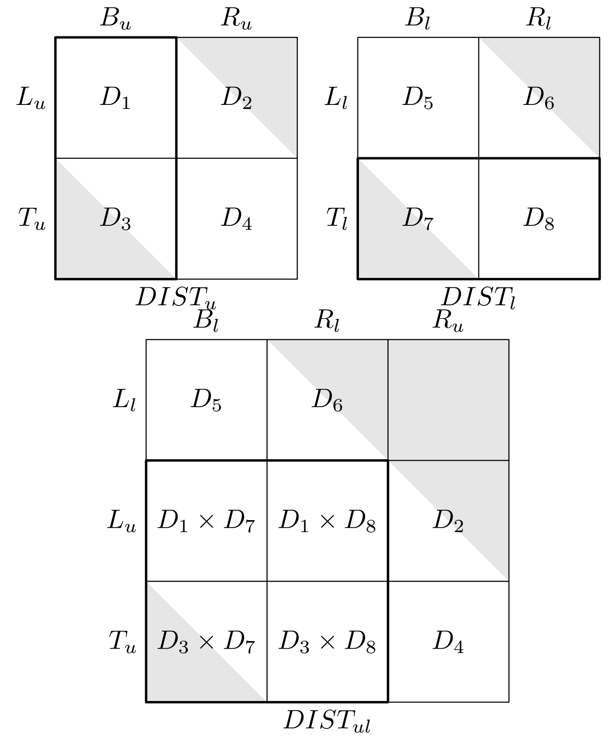

Figure 3: Matrices $DIST_u$, $DIST_l$, and $DIST_{ul}$.

The existence of the unused (shaded) parts in the distance matrices has an impact on some constants in the paper, but is not important to our results. Therefore we will ignore it for simplicity.

To define indices to the interesting part of $DIST_{ul}$ let us concentrate on the paths from $L_u \cup T_u$ to $B_l \cup R_l$. All these paths cross the common boundary $B_u = T_l$. Let S (*sources*) be the sequence $s_i, i = 1, 2, \ldots, t$ of points of $L_u \cup T_u$ beginning at the lower left corner of L_u and ending at the top right corner. Let D (*destinations*) be the sequence $d_i, i = 1, 2, \ldots, t$ of points of $B_l \cup R_l$ starting at the lower left corner of G_l and ending at the top right corner. Let M be the (*middle*) sequence $m_i, i = 1, 2, \ldots, k$ of points of $B_u = T_l$, taken from left to right (Figure 4). We will denote by $m(i,j)$ the index k of the leftmost point m_k that belongs to an optimum path between s_i and d_j. If there is no such path, then $m(i,j) = -1$ (this value will be used as a sentinel, with no other meaning).

The determination of a single $m(i,j)$ involves a search through the entire sequence M to find the x that minimizes $DIST_u(i,x) + DIST_l(x,j)$, but we can use previously calculated results to restrict this search, using the following

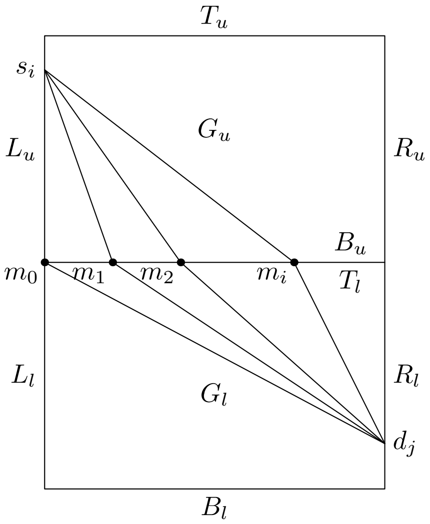

Figure 4: Merging G_u and G_l

Monge properties [1, 12]:

PROPERTY 1. *If $i_1 < i_2$ then $m(i_1, j) \leq m(i_2, j)$ for all valid j.*

PROPERTY 2. *If $j_1 < j_2$ then $m(i, j_1) \leq m(i, j_2)$ for all valid i.*

Basically, these properties imply that two optimum leftmost paths that share a common extremity cannot cross. The proof is based on the fact that we can take two crossing paths and exchange parts of them to build even better paths, or build a path that is more to the left.

Hence, if we know $m(i_1, j)$ and $m(i_2, j)$ for certain $i_1 < i_2$ and j, for all i between i_1 and i_2 we can search for $m(i, j)$ between $m(i_1, j)$ and $m(i_2, j)$. Furthermore, if for a certain j we have $m(i, j)$ for all i in a sequence $1 < i_1 < i_2 < \ldots < i_r < k$ we can calculate $m(i, j)$ for one value of i in each interval of the sequence using only one sweep through M in $O(k + r)$ time and doubling the number of known paths. In this operation, that we will call *sweep*, we use several rows of $DIST_u$ and the jth column of $DIST_l$. A similar procedure can be used to calculate $m(i, j)$ for several values of j.

These properties lead to the following sequential algorithm to obtain the new distance matrix. This algorithm will be the base for our parallel algorithm. It is based on a recursive version presented in [1]. We will calculate and use all $m(i, j)$, calculating the distance between s_i and d_j in the process. At each step, we will begin with some *marked* points in the sequences S and D, such that if s_i and d_j are *marked* then $m(i, j)$ is already known. The intervals between marked points contain points not yet used in the computations. At each step, we pick the middle point of each interval and mark it, calculating all the required paths and crossing points. We begin with only the extremities of S and D being marked.

ALGORITHM 1. *Sequential Merge of $DIST_u$ and $DIST_l$*
Input: Two distance matrices $DIST_u$ and $DIST_l$.
Output: $DIST_{ul}$

(1) $m(1, 1) = 1$, $m(1, t) = 1$, $m(t, 1) = -1$, $m(t, t) = k$;
(2) Mark s_1, s_t, d_1, d_t;
(3) **DO**
  (3.1) Take the middle point of each of the remaining intervals in S and calculate the paths to all marked points in D;
  (3.2) Take the middle point of each of the remaining intervals in D and calculate the paths to all marked points in S;
  (3.3) Take the already used middle points in S and D and calculate the paths between them. Mark all these points;
  WHILE there are unmarked points

— End of Algorithm —

THEOREM 1. *Algorithm 1 requires $\Theta(kt + t^2)$ (sequential) time.*

PROOF. Step 3.1 makes one sweep for each marked point in D. This requires time $\Theta(kr + r^2)$ where r is the number of marked points in D. Step 3.2 also requires time $\Theta(kr + r^2)$. Step 3.3 can be done in time $\Theta(kr + r^2)$ with a sweep for each (now) marked point in D. It is not hard to see that the loop is executed for $\Theta(\log t)$ iterations. The value of r approximately doubles at each iteration, from 2 to t. Hence, it follows that the total time for this algorithm is $\Theta(kt + t^2)$. $\square$

2.2 Parallelizing the Join Operation

We will now show how Algorithm 1 can be modified to compute $DIST_{ul}$ on a CGM. The natural way to parallelize this algorithm is to make each processor determine a different part of $DIST_{ul}$, but the division of $DIST_u$ and $DIST_l$ to accomplish this is data dependent. Our solution is based on a dynamic scheduling of blocks of $DIST_{ul}$ to the processors.

The CGM version of Algorithm 1 has three main phases: determination of the subproblems and of the used parts of $DIST_u$ and $DIST_l$, determination of the scheduling of the subproblems, and solution of the subproblems.

We will now adapt Algorithm 1 to calculate all distances between points in an interval S' of S (from i_1 to i_2) and points in an interval D' of D (from j_1 to j_2). Both intervals have size t', so $i_2 = i_1 + t' - 1$ and $j_2 = j_1 + t' - 1$. Assume that $m(i_1, j)$ and $m(i_2, j)$ are already calculated for all $j, j_1 \leq j \leq j_2$, and $m(i, j_1)$ and $m(i, j_2)$ are already calculated for all $i, i_1 \leq i \leq i_2$. In other words, we already know the solution for the *borders* of this subproblem. The most important difference between this subproblem and the entire problem is that only a part of each matrix $DIST_u$ or $DIST_l$ is used. The shapes of these parts are irregular as shown in Figure 5. Hence, in order to make a sweep for one point in S' (or D') we sweep a *segment* of one row of $DIST_u$ (a *segment* of one column of $DIST_l$). The running time of the sweep is determined by the size of this segment and is therefore variable, and so is the total running time of the subproblem. The sizes of all the necessary segments can be calculated in time $O(t)$, and we know how many times the algorithm will sweep each segment. Thus, the total running time of the subproblem can be estimated.

The complete problem can be divided in several subproblems and solved in parallel by $2q$ processors. To do this, let us divide S and D into $2q$ equal segments of size $t' = \frac{t-1}{2q} + 1$ (we suppose, for simplicity, that t' is an integer) that overlap only in the extremities. This will lead to $4q^2$ subproblems

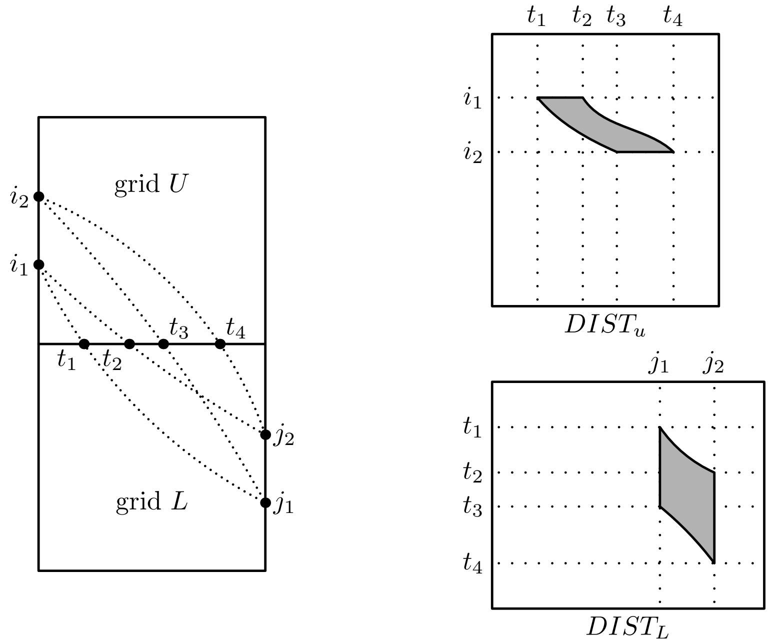

Figure 5: Data required to compute a block of $DIST_{ul}$

which will be distributed among the $2q$ processors. In fact, some of these subproblems, involving T_u and B_l, will be empty. We will omit this for now, because this will not be significant to the asymptotic performance. For now, we will work with $4q^2$ subproblems, instead of the more natural quantity q^2, in order to distribute the workload more evenly. At the beginning, the CGM/BSP algorithm for the entire problem calculates the $m(i, j)$ for all (i, j) that belong to any subproblem boundary. That implies calculating $2q+1$ equally spaced rows and $2q+1$ equally spaced columns of $DIST_{ul}$ and associated $m(i, j)$.

LEMMA 1. *All values of $DIST_{ul}(i, j)$ with $i-1$ or $j-1$ multiple of $t'-1$ and the corresponding values of $m(i, j)$ can be calculated in parallel by the $2q$ processors that store $DIST_u$ and $DIST_l$ in time $O(k \log \frac{t}{q} + qt)$ and space $O(\frac{t^2}{q})$, with two communication rounds, where $O(qt)$ data is sent/received by each processor.*

PROOF. Processors $P_{l1}, P_{l2}, \ldots, P_{lq}$ (that store $DIST_l$ distributed by rows) will calculate the $2q+1$ rows. Each one knows the lengths of the paths from an interval of points in M to all points in D. Hence, they need to receive from $P_{u1}, P_{u2}, \ldots, P_{uq}$ the length of the paths from the chosen $2q+1$ points of S (a *sample* of S) to the same interval in M. They also send information to allow $P_{u1}, P_{u2}, \ldots, P_{uq}$ to calculate the $2q+1$ required columns of $DIST_{ul}$. For this communication, each processor sends/receives $O(k)$ data.

With these data, each processor from $P_{l1}, P_{l2}, \ldots, P_{lq}$ calculates the paths from the sample of S to D, using a variation of Algorithm 1. Since $t \gg 2q+1$, the running time of this step will be dominated by the last $\log(\frac{t}{2q})$ iterations of the loop, when all points in the sample of S are marked and several in D are not. Each iteration will make $2q+1$ sweeps of a segment of size $\frac{k}{q}$. The total running time is $O(k \log(t/q))$.

Each processor among $P_{l1}, P_{l2}, \ldots, P_{lq}$ now has versions of all the paths (with lengths and crossing points) from the sample of S to D, each one considering only crossings at a certain interval of M. To calculate the better paths, D is partitioned into q equal intervals, and each processor receives all q versions of paths to all points in a certain interval ($2q+1$ paths to t/q points). Each processor sends/receives $O(qt)$ data and spends $O(qt)$ time in a naïve search, or $O(t)$ time using Monge properties to avoid searching through all versions of a path. We will omit the details for the latter due to lack of space. This concludes the procedure. ☐

Each processor among $P_{l1}, P_{l2}, \ldots, P_{lq}$ now stores information about $4q$ of the $4q^2$ subproblems: the $m(k, x)$, $m(k+t'-1, x)$ frontiers of each subproblem, as previously described, that define the part of $DIST_l$ that is used by each subproblem. $P_{u1}, P_{u2}, \ldots, P_{uq}$ will contain information about the $m(x, k)$ and $m(x, k+t'-1)$ borders of the subproblems that define the usage of $DIST_u$. These processors will not contain the actual data necessary to process these subproblems, just the information about their borders. This information is sent to the proper processors, so each one will identify which of its data is used in which subproblem. Besides that, this information will be used to estimate the time/space requirement for each subproblem.

2.3 Scheduling Subproblems to Processors

As commented earlier, to solve each subproblem, Algorithm 1 sometimes sweeps a segment of a row from $DIST_l$ and sometimes sweeps a segment of a column from $DIST_u$. The estimated running time of the subproblem can be divided in an estimation based on the used parts of $DIST_l$ and other based on the used parts of $DIST_u$. This running time can thus be estimated by the efforts of two processors. The same will apply to the total memory necessary to store the needed parts of $DIST_u$ and $DIST_l$. Each processor then takes $O(t')$ time to work on their estimations for one subproblem, $O(4qt') = O(t)$ to work on $4q$ subproblems. Then all processors send its time/space estimation to a single processor, say P_{u1}, that receives and processes $O(q^2)$ data.

Now, we need to distribute the $4q^2$ subproblems to the $2q$ processors. The objective of this distribution is to minimize the completion time by balancing the load on all processors. This is a special case of the well known, NP-hard, multiprocessor scheduling problem. In our case, we have an additional restriction on the space required by all subproblems assigned to a single processor because we have to perform the entire distribution in a single communication round. In the following, we present a solution which ensures that the claimed time and space bounds are met.

LEMMA 2. *The $4q^2$ subproblems can be scheduled among $2q$ processors in time $O(q^2 \log q)$, resulting in $O(\frac{t^2}{q})$ space and time requirements for each processor.*

PROOF. Since the subproblems overlap only at the borders, the total space required is $Space = O(t^2)$. In the worst case, one subproblem will require entire columns and rows, or $Space/2q$. The total running time of all subproblems is more difficult to calculate, because the sweeps along segments of rows or columns take different times. Let us consider the case of $DIST_l$: If we get the $2q$ subproblems that are vertically aligned in $DIST_l$, then they all use the same sweeping pattern for their segments of columns of $DIST_l$. As all these segments add up to size t, the *total* running time of these sweeps will be basically the same as the running time for these sweeps in the sequential algorithm. The final conclusion is that $Time$, the sum of the running times of all the subproblems, is approximately equal to the running time of the sequential algorithm, which is $O(t^2)$. In the worst case, one subproblem will run in $Time/2q$.

To guarantee that each processor will spend $O(t^2/q)$ time and need $O(t^2/q)$ space, we calculate a *cost* for each processor, equal to the sum of its time and space requirements. The sum of all these costs is $Cost = O(t^2)$. The problem is to distribute the subproblems among all processors. The *local cost* at a processor is the sum of the costs of all subproblems assigned to it. We will try to minimize the maximum (among all processors) local cost. Since the maximum cost that a subproblem can have is $Cost/2q$, the maximum local cost cannot be greater than the minimum local cost plus $Cost/2q$, or it would be possible to reassign subproblems in a better way. Hence, the optimum solution has cost less than $Cost/q$ (twice the cost of the best possible solution). Using a list scheduling heuristic, allocating in a greedy way the most costly subproblems first, we obtain a solution that has cost $4/3$ of the optimum solution [2]. Hence, the maximum local cost will be less than or equal to $4Cost/3q = O(t^2/q)$. Since this cost is the sum of the space and time requirements, all processors will require $O(t^2/q)$ time and $O(t^2/q)$ space to solve its subproblems. This scheduling is calculated by P_{u1} in time $O(q^2 \log q)$, where this time is dominated by the required sorting of the subproblems. □

Once the assignment is established, processor P_{u1} broadcasts this information to all other processors ($O(q^3)$ data sent). Each processor then sends its data to the proper processors and receive the data of the subproblems assigned to it. This is a complicated communication step that will require considerable bookkeeping, but the total data sent/received is $O(kt/q)$ per processor. Finally, each processor solves its subproblems, generating a $t' \times t'$ submatrix of $DIST_{ul}$ for each subproblem. The space required by the data from $DIST_l$ and $DIST_u$ can be discarded as the subproblems are solved. Once the subproblems are solved, a last communication step redistributes the submatrices of $DIST_{ul}$ in a way that will be adequate to join the new grid with its neighbor (to the left or to the right).

THEOREM 2. *If a $(2l-1)\times k$ (or $l\times(2k-1)$) grid has two $l\times k$ halves, and the distance matrix of each half is distributed in the local memories of a different set of q processors, then it is possible to calculate the distance matrix of the full grid in parallel with the $2q$ processors given in time $O((l+k)^2/q)$ and a constant number of communication steps. The local memories and the total data sent/received in each step by one processor is $O((l+k)^2/q)$.*

The theorem follows from the algorithm described above. The total communication required consists of the following steps: (1) Distribution of the samples of S and D to allow the processors to start calculating the boundaries of the subproblems. (2) Distribution of the tentative lengths and crossing points of the paths determined in the previous step. Each processor concentrates candidates for certain paths. (3) Distribution of the results of the searches for the best paths, defining the boundaries of the subproblems. The estimated size and time requirements of the subproblems are sent to one processor. (4) One processor sends the assignment of subproblems to all processors. (5) The data for the subproblems are distributed among processors, following the pre-determined assignment. (6) The results are redistributed among the processors.

2.4 Overall Analysis

Given an $n\times m$ grid and p processors, this grid is initially divided into p smaller grids. To simplify the exposition, we assume the number of processors to be an even power of two. Each subgrid is then processed by one processor to obtain its distance matrix. The division of the grid must aim for the best overall performance of the algorithm. To our knowledge, the best sequential algorithm for the problem requires $O(nm \log(\min(n,m)))$ time [12]. This algorithm was proposed to build a more complex structure that would support several kinds of queries and take $O(nm \log(\min(n,m)))$ space. However it can easily be adapted to use only $O((n+m)^2)$ space in our case where we are interested only in the boundary to boundary distances. These results make it tempting to divide the grid in strips to minimize the logarithmic factor but this is a very small gain and the local memory required would be prohibitive. As previously stated, we will divide the grid into a $\sqrt{p}\times\sqrt{p}$ configuration to ensure that the local memory required for the distance matrix will be $O((n+m)^2/p)$.

This leads to the following conclusion:

THEOREM 3. *The distance matrix of an $n\times m$ with $n > m$ grid can be calculated in parallel, on a CGM with $p < \sqrt{m}$ processors, in time $O(\frac{n^2}{p}\log m)$ with $O(\log p)$ communication rounds and $O(\frac{nm}{p})$ local memory.*

PROOF. Based on the previous discussion. The algorithm requires $O(\frac{mn}{p}\log\frac{m}{\sqrt{p}})$ time to calculate the distance matrices of the p subgrids. To build the final distance matrix it requires $\log p$ merging steps where each merging step requires a constant number of communication rounds. The $p < \sqrt{m}$ bound is sufficient to ensure that all communication rounds involve $O(\frac{nm}{p})$ data per processor. The processing time of the merging steps is $O(\frac{(n+m)^2}{p}) = O(n^2/p)$ per step, resulting in a total of $O(\frac{n^2}{p}\log p)$. Thus the whole algorithm runs in $O(\frac{n^2}{p}\log m)$ time. $\square$

In the final distance matrix we find the scores of all the alignments of substrings of C with A (lengths of paths from top to bottom of the grid), among other results of the same kind.

3. CONCLUSION

In this paper, we present an efficient algorithm to compute the edit distance between a string A and all substrings of a string C on the CGM model. The algorithm requires $\log p$ rounds/supersteps and $O(\frac{n^2}{p}\log m)$ local computation. Thus it presents linear speedup and the number of communication rounds is independent of the problem size.

4. ACKNOWLEDGMENTS

The authors wish to thank the referees for their helpful comments.

5. ADDITIONAL AUTHORS

S. W. Song (Universidade de São Paulo, São Paulo, SP - Brazil, email: `song at ime. usp. br`. Partially supported by FAPESP grant 98/06138-2, CNPq grant 52.3778/96-1 and 46.1230/00-3, and CNPq/NSF Collaborative Research Program grant 68.0037/99-3).

6. REFERENCES

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

[2] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.*, 17:416–426, 1969.

[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 Computer 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] G. M. Landau and M. Ziv-Ukelson. On the Common Substring Alignment Problem. *Journal of Algorithms*, 41:338–359, 2001.

[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] J. Schmidt. All highest scoring paths in weighted graphs and their application to finding all approximate repeats in strings. *SIAM J. Computing*, 27(4):972–992, 1998.

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

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

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

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

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