Parallel Biological Sequence Comparison Using Prefix Computations

Srinivas Aluru*
Dept. of Computer Science
New Mexico State University
Las Cruces, NM
aluru@cs.nmsu.edu

Natsuhiko Futamura*
School of EECS
Syracuse University
Syracuse, NY
nfutamur@cis.syr.edu

Kishan Mehrotra
School of EECS
Syracuse University
Syracuse, NY
kishan@cis.syr.edu

Abstract

We present practical parallel algorithms using prefix computations for various problems that arise in pairwise comparison of biological sequences. We consider both constant and affine gap penalty functions, full-sequence and subsequence matching, and space-saving algorithms. The best known sequential algorithms solve these problems in $O(mn)$ time and $O(m+n)$ space, where m and n are the lengths of the two sequences. All the algorithms presented in this paper are time optimal with respect to the best known sequential algorithms and can use up to $O\left(\frac{n}{\log n}\right)$ processors where n is the length of the larger sequence. While optimal parallel algorithms for many of these problems are known, we use a simple framework and demonstrate how these problems can be solved systematically using repeated parallel prefix operations. We also present a space-saving algorithm that uses $O\left(m+\frac{n}{p}\right)$ space and runs in optimal time where p is the number of the processors used.

1 Introduction

Sequence comparison is an important tool for researchers in molecular biology in their efforts to relate the molecular structure and function to the underlying sequence. Biological sequences can be treated as strings over a fixed alphabet of *characters*. In this paper, we consider the comparison of two biological sequences. This comparison is done by aligning the two sequences, which refers to stacking one sequence above the other and matching characters from the two sequences that lie in the same position. To deal with missing characters and extraneous characters, gaps may be inserted into either sequence. A scoring mechanism is designed for each possible alignment and the goal is to find the alignment with the best possible score. Two types of comparisons are of interest: 1) aligning the full sequences and 2) finding subsequences of the sequences that result in an alignment with maximum possible score.

Let m and n be the lengths of the two sequences to be compared. Several researchers have explored sequence comparison algorithms [12, 15], culminating in our understanding of how to solve a variety of sequence comparison problems, including subsequence matching, in $O(mn)$ time and space [4]. Myers and Miller [11] presented a technique to reduce the space requirement to optimal $O(m+n)$, while retaining a time complexity of $O(mn)$. Huang [7] extended this algorithm to subsequence matching. These algorithms are very important because the lengths of biological sequences can be large enough to render algorithms that use quadratic space infeasible.

Edmiston et. al. [2] present parallel algorithms for sequence and subsequence matching that achieve linear speedup and can use up to $O(\min(m,n))$ processors. Lander et. al. [9] discuss implementation on a data parallel computer. These algorithms store the entire dynamic programming table. Huang [6] presented a space-efficient algorithm that uses only $O\left(\frac{m+n}{p}\right)$ space per processor; however, the run-time of this algorithm is $O\left(\frac{(m+n)^2}{p}\right)$.

A widely studied identical problem is string editing – finding the minimum cost sequence of operations required to turn one string into another by using insertions, deletions and substitutions of individual characters. Highly parallel algorithms for this problem have been developed for PRAMs and hypercubes [1, 13], using almost quadratic number of processors. While the number of processors can be scaled down by proportionately increasing the workload of each processor, the corresponding algorithms are not space-efficient. From a practical standpoint, space-efficiency and the ability to use a moderate number of processors optimally are important.

In this paper, we develop a simple framework for solving the various sequence comparison problems using prefix computations as the basic building block. We show how to do sequence and subsequence matching using both constant and affine gap penalty functions. Our algorithms provide linear speedups and can use up to $O\left(\frac{n}{\log n}\right)$ processors ($m \leq n$). Our space-saving algorithm retains time optimality and uses $O\left(m+\frac{n}{p}\right)$ space. Throughout this paper, optimal time for a parallel algorithm means linear speedup with respect to the best known sequential algorithm.

*Research supported by NSF CAREER under CCR-9702991.

2 Preliminaries

We use the *permutation network* as our model of parallel computation. In this model, each processor is allowed to send and receive at most one message during a communication step. The cost of the communication step is $\tau + \mu l$, where l is the length of the largest message. This corresponds to the assumption that communication corresponding to any permutation can be realized simultaneously. The permutation network model closely reflects the behavior of most multistage interconnection networks.

Our algorithms are stated in terms of the following well-known parallel primitive operations (p denotes the number of processors). For a detailed description and run-time analysis, the reader is referred to [8].

Broadcast: In a Broadcast operation, one processor has a message of size l to be sent to all other processors. This operation takes $O((\tau + \mu l) \log p)$ time.

Reduce: Consider n data items $x_0, x_1, \ldots, x_{n-1}$ and a binary associative operator $\otimes$ that operates on these data items and produces a result of the same type. We want to compute $s = x_0 \otimes x_1 \otimes x_2 \otimes \ldots \otimes x_{n-1}$. This operation takes $O(\frac{n}{p} + (\tau + \mu) \log p)$ time.

Parallel Prefix: Consider n data items $x_0, x_1, \ldots, x_{n-1}$ and a binary associative operator $\otimes$ that operates on these data items and produces a result of the same type. We want to compute the n partial sums $s_0, s_1, \ldots, s_{n-1}$, where

$$s_i = x_0 \otimes x_1 \otimes x_2 \otimes \ldots \otimes x_i$$

This problem can be solved in $O(\frac{n}{p} + (\tau + \mu) \log p)$ time.

The algorithms presented are applicable to other models of computation as well, and often equally efficiently. This is because we use a few simple communication primitives. For example, the above operations can be done in the same time on hypercubes. Also, under the assumption that the distance between communicating processors (in the absence of network contention) can be ignored due to the large set-up costs and the moderate size of the parallel computers, the same run-time would hold for meshes as well.

3 The Basic Algorithm

Consider the problem of comparing two biological sequences. Let Σ be the alphabet and '$-$' denote the gap. A score function $f : \Sigma \cup \{\text{`}-\text{'}\} \times \Sigma \cup \{\text{`}-\text{'}\} \rightarrow I\!R$ prescribes the score for any position in the alignment. The score of the alignment is the sum of scores over all the positions. The objective of sequence comparison is to compute the alignment with the maximum possible score.

Suppose we have the simple scoring function defined as:

$$f(c_1, c_2) = \begin{cases} 1, & c_1 = c_2,\ c_1, c_2 \in \Sigma \\ 0, & c_1 \neq c_2,\ c_1, c_2 \in \Sigma \\ -1, & c_1 = \text{`}-\text{'} \text{ or } c_2 = \text{`}-\text{'} \end{cases}$$

Then, the following alignment of DNA sequences ATGACC and AGAATC has a total score of 2.

A	T	G	A	–	C	C
A	–	G	A	A	T	C
1	−1	1	1	−1	0	1

Let $A = a_1, a_2, \ldots, a_m$ and $B = b_1, b_2, \ldots, b_n$ be two sequences. Without loss of generality, we assume throughout the paper that $m \leq n$. The optimal score for aligning A and B is found by dynamic programming [14]. Construct a table T of size $(m+1) \times (n+1)$ such that $T[i,j]$ $(0 \leq i \leq m, 0 \leq j \leq n)$ records the optimal score for aligning $a_1, a_2, \ldots, a_i$ and $b_1, b_2, \ldots, b_j$. Aligning a string with the empty string is trivial; we can only insert gaps in the empty string. Therefore, $T[0,0] = 0$, and

$$T[0,j] = \sum_{k=1}^{j} f(\text{`}-\text{'}, b_k) \quad (1)$$

$$T[i,0] = \sum_{k=1}^{i} f(a_k, \text{`}-\text{'}) \quad (2)$$

$$= T[i-1,0] + f(a_i, \text{`}-\text{'}) \quad (3)$$

When aligning non-empty strings $a_1, a_2, \ldots, a_i$ and $b_1, b_2, \ldots, b_j$, there are three possibilities: 1) a_i is aligned with a gap, 2) b_j is aligned with a gap or 3) a_i is aligned with b_j. This leads to the following recurrence equation:

$$T[i,j] = \max \begin{cases} T[i-1,j] + f(a_i, \text{`}-\text{'}) \\ T[i,j-1] + f(\text{`}-\text{'}, b_j) \\ T[i-1,j-1] + f(a_i, b_j) \end{cases} \quad (4)$$

Sequentially, the matrix can be filled row by row, column by column, or diagonal by diagonal (where a diagonal represents all entries (i,j) of the table such that $i+j$ is a constant). Either way, when computing an entry of the table, the entries required in computing it are available. The table is typically filled using a row scan, taking $O(mn)$ time.

Once the table is filled, $T[m,n]$ contains the optimal score. If we draw a link from $T[i,j]$ to one of $T[i,j-1]$, $T[i-1,j]$ or $T[i-1,j-1]$ which gives the maximum value in equation (4), optimal alignment can be expressed as a path in table T, that starts at $T[m,n]$ and ends at $T[0,0]$. Let us call such a path an optimal path. Traversing an optimal path from $T[m,n]$ to $T[0,0]$ and retrieving an optimal alignment is called the Traceback procedure. This takes $O(m+n)$ time.

All the parallel algorithms for sequence comparison designed so far fill the dynamic programming table diagonal by diagonal. This is because the entries required for computing a diagonal are available from the previous two diagonals, facilitating concurrent computation of each diagonal. On the contrary, the entries in a row of the table depend not

only on the previous row but also on the row being computed. The only drawback of the diagonal based algorithm is that the size of the diagonals vary and some diagonals are too short to be concurrently computed, leading to idling of processors. Nevertheless, such an algorithm runs in optimal $O\left(\frac{mn}{p}\right)$ time and can use $\min(m,n)$ processors [2].

We present algorithms to compute the table row by row (or column by column) using parallel prefix. This leads to a uniform work distribution to all the processors at all times. Another advantage is the reduction in the number of table entries communicated per iteration from n to p. Suppose we are computing the i^{th} row using the previously computed $(i-1)^{th}$ row. We update $T[i,j]$ using equation (4). $T[i-1,j]$ and $T[i-1,j-1]$ are already computed, as they belong to the previous row. Separating precomputed terms, let

$$w[j] = \max \left\{ \begin{array}{l} T[i-1,j] + f(a_i, \text{`}-\text{'}) \\ T[i-1,j-1] + f(a_i, b_j) \end{array} \right. \quad (5)$$

$$\text{Then,} \quad T[i,j] = \max \left\{ \begin{array}{l} w[j] \\ T[i,j-1] + f(\text{`}-\text{'}, b_j) \end{array} \right. \quad (6)$$

$$\begin{aligned} \text{Let} \quad x[j] &= T[i,j] - \sum_{k=1}^{j} f(\text{`}-\text{'}, b_k) \\ &= \max \left\{ \begin{array}{l} w[j] - \sum_{k=1}^{j} f(\text{`}-\text{'}, b_k) \\ T[i,j-1] - \sum_{k=1}^{j-1} f(\text{`}-\text{'}, b_k) \end{array} \right. \\ &= \max \left\{ \begin{array}{l} w[j] - \sum_{k=1}^{j} f(\text{`}-\text{'}, b_k) \\ x[j-1] \end{array} \right. \end{aligned}$$

$$\text{Let} \quad y[j] = \sum_{k=1}^{j} f(\text{`}-\text{'}, b_k) \quad (7)$$

$$z[j] = w[j] - y[j] \quad (8)$$

The n partial sums $\sum_{k=0}^{j} f(\text{`}-\text{'}, b_k)$, as j ranges from 1 to n, can be computed using parallel prefix. As a result of this, the $z[j]$'s are known. Then,

$$x[j] = \max \left\{ \begin{array}{l} z[j] \\ x[j-1] \end{array} \right. \quad (9)$$

Since the $z[j]$'s are known, $x[j]$'s can be computed using parallel prefix with $\max$ as the binary associative operator.

Then, $T[i,j]$ $(1 \le i \le m, 1 \le j \le n)$ is derived using

$$\begin{aligned} T[i,j] &= x[j] + \sum_{k=1}^{j} f(\text{`}-\text{'}, b_k) \\ &= x[j] + y[j] \end{aligned} \quad (10)$$

Equations (5), (7), (8), (9) and (10) are used to compute a row of the table using information from the previous row. Computing equations (7) and (9) requires parallel prefix.

Typically, scoring functions assign the same score for aligning any character with a gap. Such functions are called *constant gap penalty functions*. We restrict our attention to them in order to simplify the discussion throughout the reminder of this paper. Suppose that $f(c, \text{`}-\text{'}) = f(\text{`}-\text{'}, c) = -g \quad \text{for any } c \in \Sigma$. Alternatively, we say that each gap is charged a penalty of g . For such a scoring function,

$$x[j] = \max \left\{ \begin{array}{l} w[j] + gj \\ x[j-1] \end{array} \right. \quad (11)$$

$$T[i,j] = x[j] - gj \quad (12)$$

Thus, each row of the table can be computed using equations (5), (11) and (12), using a parallel prefix for computing equation (11) alone.

Let p be the number of processors with id's ranging from 0 to $p-1$, and for simplicity, assume m and n are multiples of p. Processor i is responsible for computing columns $i\frac{n}{p}+1$ through $(i+1)\frac{n}{p}$ of the table T. Distribution of sequence B is trivial since b_j is needed only in computing column j. Therefore, processor i is given $b_{i\frac{n}{p}+1} \ldots b_{(i+1)\frac{n}{p}}$.

Each a_i is needed by all the processors at the same time when row i is being computed. We distribute sequence A among all the processors to reduce storage. Processor i stores $a_{i\frac{m}{P}+1} \ldots a_{(i+1)\frac{m}{P}}$, and broadcasts it to all processors when row $i\frac{m}{P}$ is about to be computed. If there is enough space, each processor can store a copy of A and broadcasting is eliminated.

Computing $w[j]$ in equation (5) requires $T[i-1,j-1]$, which may not be available locally. Each processor k can communicate and get such $T[i-1,j-1]$ from processor $k-1$. This process can be done fast because each processor has only one such entry to get from another processor. Another way to solve this problem is to make use of the communication performed for parallel prefix to compute $y[j]$ and $x[j]$. When parallel prefix is performed for computing $y[j]$ and $x[j]$ in row $i-1$, each processor k receives $y[j-1]$ and $x[j-1]$ where column j belongs to itself and column $j-1$ belongs to processor $k-1$. We can use them to compute $T[i-1,j-1]$, and $w[j]$ in the next row can be computed locally.

Initializing the first row of the table is done by parallel prefix using equation (1). For the computation of row i, $T[i,0]$ should be initialized and stored by processor 0 using equation (3).

Computing each row takes $O(\frac{n}{p} + (\tau+\mu)\log p)$ time. Each of the p broadcasts for broadcasting portions of string A takes $O(\frac{m}{p} + (\tau + \mu\frac{m}{p})\log p)$ time. The whole computation takes $O(\frac{mn}{p} + \tau(m+p)\log p + \mu m \log p)$ time, which is optimal for $p = O(\frac{n}{\log n})$. The space required on each processor is $O(\frac{mn}{p})$.

So far we have focused on computing the table, which only yields the score of an optimal alignment. To retrieve

an optimal alignment, we must perform the traceback procedure. The sequential traceback procedure takes $O(m+n)$ time and can be accommodated without significantly affecting the parallel run-time provided that $m+n = O\left(\frac{mn}{p}\right)$. This further restricts the number of processors that can be used to $O(m)$. Thus, we can use only $O\left(\min\left(m, \frac{n}{\log n}\right)\right)$ processors. Recall that $m \leq n$.

We can increase the number of processors that can be used by parallelizing the traceback procedure, modifying Hirschberg's technique [5]: Let $rev(A)$ (resp., $rev(B)$) denote the reverse of the sequence A (resp., B), i.e. $rev(A) = a_m a_{m-1} \ldots a_3 a_2 a_1$ and $rev(B) = b_n b_{n-1} \ldots b_3 b_2 b_1$. Let $T^R[i,j]$ denote the optimal score for aligning $a_m a_{m-1} \ldots a_{i+1}$ with $b_n b_{n-1} \ldots b_{j+1}$. We can compute T^R in a manner similar to T, using parallel prefix. Consider any column j. An optimal alignment passes through $T[k,j]$ iff

$$T[k,j] + T^R[k,j] = \max_{0 \leq i \leq m} \; (T[i,j] + T^R[i,j])$$

i.e., add column j of the tables T and T^R and each largest entry in the resulting column is on an optimal path. Each column can be processed in parallel. To compute a unique optimal alignment, we can select the topmost (alternatively, bottommost) largest entry in each column.

Computing T^R takes the same amount of time and space as computing T. Each processor has $\frac{n}{p}$ columns of both the tables. By adding the corresponding columns from both the tables, each processor can produce its share of the optimal path in $O\left(\frac{mn}{p}\right)$ time. This increases the run-time by about a factor of 3 and the space required by a factor of 2.

To reduce the usage of space, note that it is sufficient to compute the intersection of an optimal path with the rightmost column stored in each processor. We can then perform a sequential traceback within each processor, concurrently with and independently of other processors. Thus, in computing T^R, we only store entries in columns $\frac{n}{p}, \frac{2n}{p}, \frac{3n}{p}, \ldots, n$ and throw away the remaining entries. We still need to store the most recently computed row of T^R in order to compute the next row. We need only $O\left(m + \frac{n}{p}\right)$ space per processor to compute T^R. To find a unique optimal path, we adopt the following strategy: in case of a tie in moving from one entry to the next during traceback, we give priority to 1) the entry just above the current entry in the same column 2) the entry just above the current entry in the previous column and 3) the entry in the same row as the current entry but in the previous column, in that order. The time for the parallel traceback is $O\left(m + \frac{n}{p}\right)$.

Though we can use up to $O\left(\frac{n}{\log n}\right)$ processors in this manner, it may be sufficient in practice to be able to use $O\left(\min(m, \frac{n}{\log n})\right)$ processors. If so, we can save time and space by computing just T and using sequential traceback.

4 Affine Gap Penalty Functions

The gap penalty functions used so far compute the penalty of a sequence of gaps to be equal to the sum of the penalties of the individual gaps. The scoring functions used in practice are often *affine gap penalty functions*. For a maximal consecutive sequence of k gaps, an affine gap penalty function assigns a penalty of $h + gk$, where h is the gap sequence starting penalty. Thus, the first gap is charged $h+g$, while the rest of the gaps are charged g each.

To find the optimal alignment using affine gap penalty functions, we consider the sequential algorithm presented in [14]. It uses three tables T_1, T_2 and T_3, each of size $(m+1) \times (n+1)$. As before, entry $[i,j]$ in each table corresponds to the score for optimally aligning $a_1, a_2, \ldots, a_i$ with $b_1, b_2, \ldots, b_j$, but with the following exception: In filling T_1, we consider alignments where a_i is matched with b_j. For T_2, we consider alignments where '$-$' is matched to b_j and for T_3, we consider alignments where a_i is matched to '$-$'. It is easy to see that the tables can be filled with the following equations (for more explanation, see [14]):

$$\begin{aligned}
T_1[i,j] &= f(a_i, b_j) + \max \left\{ \begin{array}{l} T_1[i-1,j-1] \\ T_2[i-1,j-1] \\ T_3[i-1,j-1] \end{array} \right. \\
T_2[i,j] &= \max \left\{ \begin{array}{l} T_1[i,j-1] - (g+h) \\ T_2[i,j-1] - g \\ T_3[i,j-1] - (g+h) \end{array} \right. \\
T_3[i,j] &= \max \left\{ \begin{array}{l} T_1[i-1,j] - (g+h) \\ T_2[i-1,j] - (g+h) \\ T_3[i-1,j] - g \end{array} \right.
\end{aligned}$$

We compute all the three tables together, row by row. The i^{th} rows of T_1 and T_3 can be computed directly as they depend only on $(i-1)^{th}$ rows. After computing them, the i^{th} row of T_2 can be computed using parallel prefix. Separating precomputed terms, let

$$w[j] = \max \left\{ \begin{array}{l} T_1[i,j-1] - (g+h) \\ T_3[i,j-1] - (g+h) \end{array} \right. \qquad (13)$$

$$\text{Then} \qquad T_2[i,j] = \max \left\{ \begin{array}{l} w[j] \\ T_2[i,j-1] - g \end{array} \right.$$

This is same as (6) by treating $f(\text{`}-\text{'}, b_j) = -g$ for any b_j. So, the i^{th} row of T_2 can be computed in the same way using parallel prefix: Use equations (13), (11) and (12), except that T should be changed to T_2 in equation (12).

Initialization and computation of the tables is done in the same manner as the basic algorithm, except that rows i

of the three tables are filled before filling row $i+1$. The same run-time analysis holds as well. Thus, we can use at most $O\left(\frac{n}{\log n}\right)$ processors to compute the tables in optimal parallel time. If we perform a sequential traceback procedure and wish to be time-optimal, we can use only $O\left(\min\left(m, \frac{n}{\log n}\right)\right)$ processors.

We can attempt to increase the number of processors that can be used by computing T_1^R, T_2^R, T_3^R, corresponding to aligning $rev(A)$ and $rev(B)$ using the affine gap penalty function. By properly combining all the tables, we can compute a table where a largest entry in each column lies on an optimal path. However, picking the topmost largest entry in each column may not give us an optimal path but only parts from several optimal paths. However, a different method can be used to utilize $O\left(\frac{n}{\log n}\right)$ processors. This method is presented in the next section.

5 A Space-Saving Algorithm

All the algorithms presented so far require $O(\frac{mn}{p})$ space per processor, which limits the lengths of the sequences that can be compared. Here, we show how to reduce the space-requirement to $O\left(m + \frac{n}{p}\right)$ while maintaining $O\left(\frac{mn}{p}\right)$ run-time. Our algorithm develops on the ideas presented by Edmiston et. al. [2] in the context of performing tracebacks with limited amounts of memory. Huang [6] presented an algorithm that uses only optimal $O\left(\frac{m+n}{p}\right)$ space, at the expense of increasing the run-time to $O\left(\frac{(m+n)^2}{p}\right)$. Our algorithm offers another alternative in the tradeoff between time and space where we retain time-optimality and attempt to reduce the space requirement as much as possible.

To put the various algorithms in practical perspective, DNA sequences can range from a few thousand base pairs (characters for our purpose) to three billion base pairs (for the human DNA) long. Suppose we are comparing two sequences of length about $100,000$ and we use 100 processors. Assuming that each table entry is 4 bytes, the naive algorithm requires about 400 MB of space per processor, too large to fit in the main memory of most current systems. Huang's algorithm requires only about 8 KB! Our algorithm uses about 0.4 MB. While Huang's algorithm can work with very limited amounts of memory, the space-requirement of our algorithm can be reasonably satisfied even for very large sequences. We believe that it is a good practical choice given that it is simple and time-optimal.

Let us define $p-1$ special columns C_k $(1 \leq k \leq p-1)$ of a table to be the last columns of the parts of the table allocated to each processor, except the last processor. i.e., $C_k = k \times \frac{n}{p}$. If we identify the intersection of an optimal path with the special columns, we can split the problem into p subproblems and solve them sequentially on each processor (see Figure 1), using the sequential space-saving algorithm [11]. The solutions of the subproblems are then concatenated to get the total alignment. Note that each subproblem receives exactly $\frac{1}{p}^{th}$ of sequence B but an undetermined portion of sequence A. Since the total length of the sequence A is m, the sequential run-time of computing each subproblem is $O\left(\frac{mn}{p}\right)$.

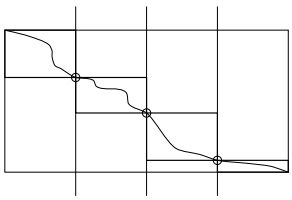

Figure 1. Parallel space-saving algorithm

It remains to describe how the intersection of an optimal path with the special columns can be computed. We only store information on the special columns of a table. In addition, we store the most recently computed row of a table in order to compute the next row using parallel prefix. For constant gap penalty functions, we can compute the forward and reverse tables and identify the topmost largest entries along the special columns to split the problem into subproblems. As noted before, such a method does not work for affine gap penalty functions.

A more general approach, that works for either type of penalty functions, is as follows: For each entry $T[i,j]$ of a table, compute both the 'value' of the entry and the row number of the entry in the closest special column to the left that lies on an optimal path from $T[0,0]$ to $T[i,j]$. Let us call such a row number a pointer to the previous special column. This essentially gives us the ability to jump from special column to special column and perform a traceback in the entire table, without considering other columns. Suppose we are computing a row of a table. The values in the row are computed using parallel prefix as before. The pointers to previous special column can be computed by another parallel prefix. Recall that an entry $T[i,j]$ may originate from $T[i-1,j]$, $T[i-1,j-1]$, or $T[i,j-1]$. If the entry results from an entry in the previous row ($T[i-1,j]$ or $T[i-1,j-1]$), we simply copy the pointer of the entry. If the entry results from $T[i,j-1]$, we wish to copy the pointer of $T[i,j-1]$ but it may not yet be available. Therefore it is initially set to u (undefined), unless $j-1$ is a special column. If so, the row number i is taken to be the pointer. We can then fill the undefined entries using parallel prefix and the following operation:

$$x \oplus y = \begin{cases} u, & x = u, y = u \\ y, & y \neq u \\ x, & x \neq u, y = u \end{cases}$$

After computing the current row, we discard the previous row except for entries intersecting with the special columns. This gives us a space bound of $O\left(m + \frac{n}{p}\right)$. It is possible to combine the two parallel prefix operations into a single parallel prefix operation using a slightly more complex operator. This is useful in practice because it slashes the communications costs in half. The details are omitted in the interest of brevity.

A sequential traceback procedure along the special columns can be used to split the problem into p subproblems in $O(p)$ time. This does not significantly affect the run-time of $O\left(\frac{mn}{p}\right)$ provided $p^2 = O(mn)$. While this is a reasonable assumption in practice, time-optimality can be retained even if this is not true. The idea is to parallelize the traceback procedure itself using parallel prefix. Each special column contains pointers to the previous special column. It is required to establish pointers from the last special column to every other special column by following the chain of pointers leading to it. This can again be done using a parallel prefix within the time bounds required.

It remains to describe how the data required for the subproblems is moved to the respective processors. Sequence B is already distributed appropriately. To distribute sequence A, recall that A is presently distributed uniformly across all the processors. While better methods can be designed, a simple circular shift of the sequence among all the processors suffices to prove the required time complexity. Perform p circular shift operations on the sequence A such that the entire sequence passes through each processor. Each processor retains as much of sequence A as it needs. Assuming a linear array can be embedded in the processor topology, this requires only $O(\tau p + m)$ time. If there is sufficient memory on each processor to store the entire sequence A through out the computation, none of this data movement is necessary. Notice that this does not violate our space bound of $O\left(m + \frac{n}{p}\right)$. Finally, by defining all columns to be special columns, we can obtain a non-space saving algorithm for affine gap penalty functions that can use $O\left(\frac{n}{\log n}\right)$ processors.

6 Subsequence Matching

In the subsequence matching problem, we are interested in an alignment between a subsequence of A and a subsequence of B that results in the highest possible score. The problem can be solved using an approach similar to sequence matching [14, 15]. For simplicity, assume a constant gap penalty function. As in the case of sequence matching, a table T of size $(m+1) \times (n+1)$ is created but with the following difference: Entry $T[i, j]$ is used to store the highest score of an alignment between a suffix of $a_1 a_2 \ldots a_i$ and a suffix of $b_1 b_2 \ldots b_j$. Alignment of empty suffixes is a valid alignment, resulting in a score of 0. Thus, if no non-empty suffixes of $a_1 a_2 \ldots a_i$ and $b_1 b_2 \ldots b_j$ can be aligned with a positive score, a 0 should be placed in $T[i, j]$. Therefore, all the scores in the table are non-negative. The table can be filled with the following equation:

$$T[i,j] = \max \begin{cases} T[i-1,j] + f(a_i, \text{`}-\text{'}) \\ T[i,j-1] + f(\text{`}-\text{'}, b_j) \\ T[i-1,j-1] + f(a_i, b_j) \\ 0 \end{cases}$$

Once the table is filled, the maximum entry in the table gives the score for the subsequence matching problem. An optimal alignment corresponding to this score is retrieved by performing a traceback from a maximum entry in the table until a score of 0 is reached.

It is very easy to extend the parallel prefix based algorithm to the subsequence matching problem. Let

$$w[j] = \max \begin{cases} T[i-1,j] + f(a_i, \text{`}-\text{'}) \\ T[i-1,j-1] + f(a_i, b_j) \\ 0 \end{cases}$$

As before, computing the $w[j]$'s requires information only from the previous row. With this modification for computing the $w[j]$'s, the rest of the algorithm is the same as before. Affine gap penalty functions can be taken care of in a similar manner as well. The equations for computing T_1, T_2 and T_3 in Section 4 need to be modified to turn any non-negative entry into a zero entry. Each processor can scan its portion of the table in $O\left(\frac{mn}{p}\right)$ time to identify the largest entry in its portion of the table. The largest entry in the whole table can be computed using a simple reduction operation. Optimal alignment can be read off from the table starting from the largest entry. As before, if a sequential traceback procedure is used, the number of processors that can be used is limited by the length of the optimal alignment. Since the length of the optimal alignment can be as high as $m + n$ in the worst case, only $O\left(\min\left(m, \frac{n}{\log n}\right)\right)$ processors can be used, the $\frac{n}{\log n}$ limit stems from the need to perform parallel prefix on n items.

If more processors need to be used, we can easily convert the subsequence matching problem into the sequence matching problem and use any of the methods described earlier. Given sequences A and B, we compute the necessary dynamic programming tables to identify the position of a largest entry. While doing so, it is not necessary to store the table as we can keep track of a largest entry occurred so far and the position at which it occurred using a single variable on each processor. Therefore, the only memory required is that needed for storing the previous row and for computing the current row, which is only $O\left(\frac{n}{p}\right)$. A

largest entry in the table corresponds to the end of an optimal subsequence matching. Let $[i, j]$ denote the position of a largest entry and let max denote the largest entry. We can run a sequence comparison algorithm on the sequences $a_i a_{i-1} \ldots a_1$ and $b_j b_{j-1} \ldots b_1$ until an entry records max. This entry corresponds to the starting point for an optimal subsequence that ends at $[i, j]$. Again, note that we need only $O\left(\frac{n}{p}\right)$ memory to compute the table as we are interested only in the entry recording max. Let the entry be $[k, l]$. Therefore, the subsequences from the sequences A and B that result in an alignment with the highest score are $a_k a_{k+1} \ldots a_i$ and $b_l b_{l+1} \ldots b_j$. We can align them using any of the sequence matching algorithms described earlier.

The method consists of three phases: Running a subsequence matching algorithm between $a_1 a_2 \ldots a_m$ and $b_1 b_2 \ldots b_n$, running a sequence matching algorithm between $a_i a_{i-1} \ldots a_1$ and $b_j b_{j-1} \ldots b_1$, and running a sequence matching algorithm between $a_k a_{k+1} \ldots a_i$ and $b_l b_{l+1} \ldots b_j$. We need to find the alignment only in the third phase. It remains to discuss how data is distributed for the second and third phases. To prove time-optimality, it is not necessary to redistribute the data to balance the computation in the second and third phases across processors. We can simply partition the table to be computed such that a processor is responsible for computing the columns of the table that correspond to the subsequence of the B it is originally assigned to. It is easy to verify that while some processors may idle and some others may have less columns to compute during the second and third phases, the work done by each processor in any of the phases is only $O\left(\frac{mn}{p}\right)$. In practice, one may wish to redistribute the necessary subsequences of the sequences such that all processors have a uniform work load during the second and third phases.

7 Conclusion

In this paper we use prefix computation to obtain optimum time complexity for solving various problems related to sequence comparisons; the methods reduce the space complexity as well. In the proposed algorithms all processors are assigned equal amount of work. Furthermore, our algorithms communicate fewer number of table entries. Consequently, they are expected to give better performance in real applications than the previous optimal time parallel algorithms. Our methodology, discussed in the context of the sequence comparison problems, is general and can be used to parallelize other dynamic-programming problems.

Acknowledgements

The authors gratefully acknowledge valuable suggestions from the anonymous referees, and in particular, for pointing out an erroneous algorithm in an earlier draft of this paper.

References

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

[2] E. Edmiston, N. Core, J. Saltz, and R. Smith. Parallel processing of biological sequence comparison algorithms. *International Journal of Parallel Programming*, 17(3):259–275, 1988.

[3] E. Edmiston and R. Wagner. Parallelization of the dynamic programming algorithm for comparison of sequences. *Proceedings of the 1987 International Conference on Parallel Processing*, pages 78–80, 1987.

[4] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[5] D. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[6] X. Huang. A space-efficient parallel sequence comparison algorithm for a message-passing multiprocessor. *International Journal of Parallel Programming*, 18(3):223–239, 1989.

[7] X. Huang. A space-efficient algorithm for local similarities. *Computer Applications in the Biosciences*, 6(4):373–381, 1990.

[8] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to parallel computing*. The Benjamin/Cummings Publishing Company, Inc, Redwood City, CA, 1994.

[9] E. Lander, J. Mesirov, and W. Taylor. Protain sequence comparison on a data parallel computer. *Proceedings of the 1988 International Conference on Parallel Processing*, pages 257–263, 1988.

[10] E. Myers. An overview of sequence comparison algorithm in molecular biology. Technical report, University of Arizona, 1991.

[11] E. Myers and W.Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

[12] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[13] S. Ranka and S. Sahni. String editing on an SIMD hypercube multicomputer. *Journal of Parallel and Distributed Computing*, 9:411–418, 1990.

[14] J. Setubal and J. Meidanis. *Introduction to computational molecular biology*. PWS Publishing Company, Boston, MA, 1997.

[15] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[16] T. Yap, O. Frieder, and R. Martino. Parallel computation in biological sequence analysis. *IEEE Transactions on parallel and Distributed Systems*, 9(3):283–293, 1998.