ACADEMIC PRESS

Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

J. Parallel Distrib. Comput. 63 (2003) 264–272

Journal of Parallel and Distributed Computing

http://www.elsevier.com/locate/jpdc

Parallel biological sequence comparison using prefix computations

Srinivas Aluru,[a,*,1,2] Natsuhiko Futamura,[b,2] and Kishan Mehrotra[c]

[a] Department of Electrical and Computer Engineering, Iowa State University, 3218 Coover Hall, Ames, IA 50011, USA
[b] Department of Computer Science, Wright State University, Dayton, OH 45435, USA
[c] School of EECS, Syracuse University, Syracuse, NY 13244, USA

Received 7 December 1999; revised 3 October 2001; accepted 15 November 2002

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. Commonly used sequential algorithms solve the sequence comparison problems in $O(mn)$ time and $O(m+n)$ space, where m and n are the lengths of the sequences being compared. All the algorithms presented in this paper are time optimal with respect to the sequential algorithms and can use $O(\frac{n}{\log n})$ 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 $O(m+\frac{n}{p})$ space and runs in optimal time where p is the number of the processors used. We implemented the parallel space-saving algorithm and provide experimental results on an IBM SP-2 and a Pentium cluster.

Keywords: Computational biology; Sequence alignments; Parallel prefix; Space-efficient; Parallel algorithms

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 sequence data mainly consists of DNA and protein sequences, which 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 with the intention of 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 an alignment with the best possible score. Two types of comparisons are of interest: (1) aligning 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. Sequence comparison algorithms typically use dynamic programming in which a table, or multiple tables of size $(m+1)\times(n+1)$ are filled. Several researchers have explored sequence comparison algorithms [14,17], culminating in the solution of a variety of sequence comparison problems, including subsequence matching, in $O(mn)$ time and space [5]. Using the technique of Hirschberg [7], developed in the context of the longest common subsequence problem, Mayers and Miller [13] presented a technique to reduce the space requirement of sequence matching to optimal $O(m+n)$, while retaining a time complexity of $O(mn)$. Huang [9] 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.

While space-optimal algorithms make large sequence comparison feasible, the quadratic time requirement still makes it a time-consuming process. A natural approach is to reduce the time requirement with the use of parallel

*Corresponding author. Fax: 515-294-8432.
E-mail address: aluru@iastate.edu (S. Aluru).
[1] Supported by NSF CAREER under CCR-0096288 and NSF EIA-9729877.
[2] Research conducted at Iowa State University.

0743-7315/03/$ - see front matter © 2003 Elsevier Science (USA). All rights reserved.
doi:10.1016/S0743-7315(03)00010-8

computers. Edmiston et al. [3,4] 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. [11] discuss implementation on a data parallel computer. These algorithms store the entire dynamic programming table. Huang [8] presented a space-efficient algorithm for sequence alignment, that is only time optimal when the sequences have equal size.

A widely studied identical problem is string editing—finding a minimum cost sequence of operations for transforming one string into another by using insertions, deletions and substitutions of individual characters. Highly parallel algorithms for this problem have been developed for the PRAM and hypercube models of computation [1,15], using almost quadratic number of processors. While the number of processors can be scaled down by proportionately increasing the workload per processor, the corresponding algorithms are not space-efficient. From a practical standpoint, the important factors are space efficiency and the ability to use a moderate number of processors (at most a few thousand) optimally.

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 perform sequence and subsequence matching using constant and affine gap penalty functions. Our algorithms provide linear speedups while using up to $O(\frac{n}{\log n})$ processors $(m \leqslant n)$. Their primary advantages over existing approaches are their communication efficiency, uniform work allocation and ease of programming. Huang [8] presented a parallel sequence comparison algorithm that uses optimal $O(\frac{m+n}{p})$ space, at the expense of increasing the run-time to $O(\frac{(m+n)^2}{p})$. We provide an algorithm that retains time optimality and uses $O(m+\frac{n}{p})$ space. Throughout this paper, optimal time for a parallel algorithm means linear speedup with respect to the $O(mn)$ sequential algorithms. An asymptotically faster sequential algorithm for sequence comparison that runs in $O(\frac{n^2}{\log^2 n})$ time in the unit-cost RAM model is given by Masek and Paterson [12]. However, this algorithm is rarely used in practice and is not expected to be faster unless the sequences are extremely large.

The rest of the paper is organized as follows: In Section 2, we describe the model of parallel computation used and outline some communication primitives of interest. In Section 3, we present our sequence comparison algorithm for affine gap penalty functions, and also discuss simplified algorithms for the special case of constant gap penalty functions. Our parallel space-saving algorithm is described in Section 4. The subsequence matching problem is considered in Section 5. Experimental results are presented in Section 6. Section 7 concludes the paper.

2. Preliminaries

We use the *permutation network* model of parallel computation. In this model, each processor can send and receive at most one message during a communication step. The run-time of the 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 model closely reflects the behavior of most multistage interconnection networks and Clos networks, and the programming abstraction supported by MPI. Our algorithms are described using the following well-known parallel communication primitives (p denotes the number of processors). For a detailed description and run-time analysis, see [10].

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 $s = x_0 \otimes x_1 \otimes x_2 \otimes \cdots \otimes x_{n-1}$. 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. The parallel prefix problem is to compute the n partial sums $s_0, s_1, \ldots, s_{n-1}$, where

$$s_i = x_0 \otimes x_1 \otimes x_2 \otimes \cdots \otimes x_i.$$

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 parallel sequence comparison algorithm

Let Σ be the alphabet and '—' denote the gap. A score function $f : \Sigma \times \Sigma \to \mathbb{R}$ prescribes the score for any column in the alignment that does not contain a gap. Examples of such scoring functions are the PAM [2] and BLOSUM [6] matrices widely used for protein sequences. Scores of columns involving gaps are determined by an *affine gap penalty function*: for a maximal consecutive sequence of k gaps, a penalty of $h + gk$ is applied. Thus, the first gap in a maximal sequence is charged $h + g$, while the rest of the gaps are charged g

each. When $h = 0$, the scoring function is called a *constant gap penalty function*. The score of the alignment is the sum of scores over all the columns. Affine gap penalty functions are commonly used so that a sequence of gaps is assigned less penalty than treating them as individual gaps. This is because a mutation affecting a short substring is more likely than several individual point mutations.

Consider an example of aligning the two DNA sequences (strings over the alphabet $\{A, C, G, T\}$) ATGTCGA and AGAATCTA using the simple scoring function defined as

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

and an affine gap penalty function that penalizes a maximal sequence of gaps of length k with a penalty of $2 + k$. Then, the following alignment has a total score of -2:

A	T	G	–	–	T	C	G	A
A	–	G	A	A	T	C	T	A
1	−3	1		−4	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 $m \leqslant n$. To find an optimal alignment of A and B using affine gap penalty functions, the following dynamic programming algorithm is applied [16]: The algorithm uses three tables T_1, T_2 and T_3, each of size $(m+1) \times (n+1)$. An 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 conditions: In T_1, a_i must be matched with b_j. In T_2, '–' must be matched to b_j and in T_3, a_i must be matched to '–'. The tables can be filled with the following equations (for more explanation, see [16]):

$$T_1[i,j] = f(a_i, b_j) + \max \begin{cases} T_1[i-1,j-1], \\ T_2[i-1,j-1], \\ T_3[i-1,j-1], \end{cases} \tag{1}$$

$$T_2[i,j] = \max \begin{cases} T_1[i,j-1] - (g+h), \\ T_2[i,j-1] - g, \\ T_3[i,j-1] - (g+h), \end{cases} \tag{2}$$

$$T_3[i,j] = \max \begin{cases} T_1[i-1,j] - (g+h), \\ T_2[i-1,j] - (g+h), \\ T_3[i-1,j] - g. \end{cases} \tag{3}$$

The first row and column of each table are initialized to $-\infty$, except in the following cases $(1 \leqslant i \leqslant m; 1 \leqslant j \leqslant n)$:

$$T_1[0,0] = 0,$$
$$T_2[0,j] = h + gj,$$
$$T_3[i,0] = h + gi.$$

Sequentially, the tables can be filled together 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 already known. The tables are typically filled using a row by row scan, taking $O(mn)$ time. All the parallel algorithms for sequence comparison designed so far fill the dynamic programming table diagonal by diagonal. This is because all the entries required for computing a diagonal fall only in the previous two diagonals, facilitating concurrent computation. But the size of the diagonals vary and some diagonals are too short to be concurrently computed using all processors, leading to idling of processors. Nevertheless, such an algorithm results in optimal $O(\frac{mn}{p})$ time and can use as many as $\min(m,n)$ processors [4].

We demonstrate how to compute the tables row by row (or column by column) using a simple application of parallel prefix. Consider computing row i of T_1, T_2 and T_3, after the $(i-1)$th rows of the tables are already computed. The ith rows of T_1 and T_3 can be computed directly as they depend only on $(i-1)$th rows. After computing them, the ith row of T_2 can be computed using parallel prefix. Separating the terms that are already computed, let

$$w[j] = \max \begin{cases} T_1[i,j-1] - (g+h), \\ T_3[i,j-1] - (g+h). \end{cases} \tag{4}$$

Then,

$$T_2[i,j] = \max \begin{cases} w[j], \\ T_2[i,j-1] - g. \end{cases}$$

Let

$$\begin{aligned} x[j] &= T_2[i,j] + jg \\ &= \max \begin{cases} w[j] + jg \\ T_2[i,j-1] + (j-1)g, \end{cases} \\ &= \max \begin{cases} w[j] + jg \\ x[j-1]. \end{cases} \end{aligned}$$

Since $w[j] + jg$ is known for all j, $x[j]$'s can be computed using parallel prefix with max as the binary associative operator. Then, $T_2[i,j]$ $(1 \leqslant j \leqslant n)$ can be derived using

$$T_2[i,j] = x[j] - jg.$$

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 tables T_1, T_2 and

T_3. Distribution of sequence B is trivial because 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 $T_1[i,j]$ in Eq. (1) needs $T_1[i-1,j-1]$, $T_2[i-1,j-1]$ and $T_3[i-1,j-1]$. $w[j]$ in Eq. (4) requires $T_1[i,j-1]$ and $T_3[i,j-1]$. Those required elements are not available locally only for the extreme left column on each processor. Each processor can communicate and get these five entries from its preceding processor. The size of the messages is constant and independent of the table sizes.

Computing each row takes $O(\frac{n}{p} + (\tau+\mu)\log p)$ time. Each of the p broadcasts for broadcasting portions of sequence 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 $O(\frac{mn}{p})$.

Once the tables are filled, one of the entries $T_1[m,n]$, $T_2[m,n]$ or $T_3[m,n]$ with the maximum score gives the optimal score and an optimal alignment can be read from the table using a traceback procedure starting from that entry. If we draw links from each table element to one or more of the entries which give the maximum value in Eq. (1), (2) or (3), an optimal alignment can be expressed as a path in the three dynamic programming tables, that starts at the largest $[m,n]$ entry and ends at $T_1[0,0]$. Let us call such a path an optimal path. There may be several optimal paths corresponding to the same optimal score. Traceback procedure follows an optimal path and retrieves an optimal alignment in $O(m+n)$ time. This can be accommodated without significantly affecting the parallel runtime provided that $m+n = O(\frac{mn}{p})$. This further restricts the number of processors that can be used to $O(m)$. Thus, only $O(\min(m, \frac{n}{\log n}))$ processors can be used. Recall that $m \leqslant n$. This restriction can be improved using a parallel traceback algorithm, discussed in the next section.

In the special case of constant gap penalty functions ($h = 0$), sequence comparison can be done with only one table T, where entry $[i,j]$ stores the score for optimally aligning $a_1, a_2, \ldots, a_i$, with $b_1, b_2, \ldots, b_j$ [16]. The table is computed using

$$T[i,j] = \max \begin{cases} T[i-1,j-1] + f(a_i, b_j), \\ T[i-1,j] - g, \\ T[i,j-1] - g. \end{cases}$$

T can be computed row by row using parallel prefix in the same way as T_2 is computed in case of affine gap penalty functions. Moreover, the following simple strategy suffices to parallelize traceback: Let $\mathrm{rev}(A) = a_m a_{m-1} \ldots a_3 a_2 a_1$ and $\mathrm{rev}(B) = b_n b_{n-1} \ldots b_3 b_2 b_1$. Let $T^R[i,j]$ denote the optimal score for aligning $a_m a_{m-1} \ldots a_{i+1}$ with $b_n b_{n-1} \ldots b_{j+1}$. 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

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

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. To compute a unique optimal alignment, select the topmost (alternatively, bottommost) largest entry in each column. Each column can be processed in parallel.

To reduce the usage of space for T^R, note that it is enough to compute the intersection of an optimal path with the rightmost column stored in each processor. Then, a sequential traceback can be performed in T within each processor, concurrently with and independently of other processors. Thus, in computing T^R, store only the rightmost column in each processor. Each processor adds the rightmost column of T and T^R it is assigned to and picks the topmost largest entry in that column. To find a unique optimal path by traceback, we adopt the following strategy: in case of a tie in moving from an entry $T[i,j]$ to the next, give priority to $T[i-1,j]$, $T[i-1,j-1]$, and $T[i,j-1]$, in that order. The time for the parallel traceback is $O(m + \frac{n}{p})$. This method does not work for affine gap penalty functions because the topmost entries on each column may not be part of a single optimal path.

4. A space-saving algorithm and faster traceback

In this section, we present a parallel algorithm that reduces the space required to $O(m + \frac{n}{p})$ while maintaining $O(\frac{mn}{p})$ run-time. It is developed on the ideas presented by Edmiston et al. [4] in the context of performing tracebacks with limited memory. Compared with Huang's space optimal algorithm which is not time optimal, our algorithm is time optimal but not space optimal, offering another alternative in the tradeoff between time and space. To put the various algorithms in perspective, consider aligning two sequences of length 1,000,000 using an affine gap penalty function on 100 processors. Assuming each table entry is 4 bytes, the space per processor of the naive algorithm, Huang's algorithm and our algorithm are 120 GB, 240 KB, and 12 MB, respectively. While Huang's algorithm can work with much smaller memory, the space requirement of

our algorithm is reasonable even for very large sequences. We believe it is a good practical choice given that it is simple and time optimal.

Define p special columns C_k $(0 \leqslant k \leqslant p-1)$ of a table to be the last columns of the parts of the tables allocated to each processor, except for the last processor, i.e., $C_k = (k+1) \times \frac{n}{p}$. If the intersections of an optimal path with the special columns are identified, the problem can be split into p subproblems (see Fig. 1), to be solved one per processor using the sequential space-saving algorithm [13]. The solutions of the subproblems are then concatenated to get the total alignment. 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, each subproblem can be solved sequentially in $O(\frac{mn}{p})$ time and $O(m+\frac{n}{p})$ space. Memory permitting, multiple special columns per processor can be used, resulting in smaller subproblems and decreased overall run-time. Thus, there is a memory vs. run-time tradeoff.

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. This gives a space bound of $O(m+\frac{n}{p})$. For each entry of a table, the 'value' of the entry and the $\langle$ *table number*, *row number* $\rangle$ tuple of the entry in the closest special column to the left that lies on an optimal path from $T_1[0,0]$ to the entry are computed. Call such a tuple a *pointer* to the previous special column. This essentially gives the ability to perform a traceback through special columns, without considering other columns. The values in a row of a table are computed using parallel prefix as before. The pointers for tables $T_1[i,j]$ and $T_3[i,j]$ can be copied from the entry in the previous rows of the three tables that is responsible for the value chosen by the max operator. For $T_2[i,j]$, we wish to copy the pointer from one of $T_1[i,j-1]$, $T_2[i,j-1]$, or $T_3[i,j-1]$, whichever results in the maximum value. Pointers for $T_1[i,j-1]$ and $T_3[i,j-1]$ are available, but if $T_2[i,j]$ results from $T_2[i,j-1]$, the pointer is not known because it is on the current row too. Therefore it is initially set to u (undefined), unless $j-1$ is a special column. If so, $\langle T_2, i \rangle$ is taken to be the pointer. The undefined entries can then be filled using parallel prefix and the following operation:

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

In fact, the parallel prefix for establishing the pointers can be avoided altogether. This is because the last column of the table allocated to each processor is a special column and the pointer value in an entry next to the special column is already known. Therefore, a sequential prefix computation within each processor is enough to determine the pointers.

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 $O(\frac{mn}{p})$ 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 element on a special column contains a pointer to the element on the previous special column in one of the three tables. It is required to establish a pointer from each element on the last special column of each table to an element on every other special column in one of the three tables following the chain of pointers leading to it. Consider the special columns $C_{p-1}, C_{p-2}, \ldots, C_0$ of all the tables. To operate on the special columns on the three tables at once, special columns with the same column numbers are combined and considered as an array of size $3(m+1)$. Let the jth element of a special column on T_i be mapped to the $((i-1)(m+1)+j)$th element of the combined special column, and pointer tuples $\langle$ *table number*, *row number* $\rangle$ stored at each special column be adjusted accordingly. Define the operator $\oplus$ such that

$$(A \oplus B)[i] = B[A[i]],$$

where A, B and $A \oplus B$ are arrays of length $3(m+1)$ representing array of pointers on special columns. Partial sums $s_{p-1}, s_{p-2}, \ldots, s_0$, where

$$s_k = C_{p-1} \oplus C_{p-2} \oplus \cdots \oplus C_{k+1}$$

can be computed using parallel prefix. As applying this operator takes $O(m)$ time, such a parallel prefix takes $O(m \log p + (\tau + \mu m)\log p)$ time. As we can take $O(\frac{mn}{p})$ time, up to $O(\frac{n}{\log n})$ processors can be utilized. $s_k[l]$ contains the intersection of an optimal path with C_k, where l corresponds to one of $T_1[m,n]$, $T_2[m,n]$ or $T_3[m,n]$ that gives the optimal score.

It remains to be described how the data required for the subproblems is moved to the respective processors. Sequence B is already distributed appropriately. To

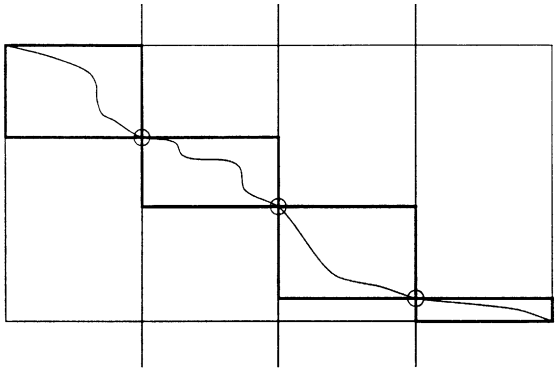

Fig. 1. Problem decomposition in parallel space-saving algorithm.

distribute sequence A, recall that A is presently distributed uniformly across all the processors. While better methods can be designed, the following suffices to prove the required time complexity. Perform p circular shift operations on 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, data movement can be avoided by storing the entire sequence A throughout the computation, without violating our space bound of $O(m + \frac{n}{p})$.

5. Subsequence matching

The subsequence matching problem between sequences A and B is to find an alignment between a subsequence of A and a subsequence of B that results in the highest possible score over all such possible alignments and subsequences [16,17]. As in the case of sequence matching, tables T_1, T_2 and T_3 are created but with the following difference: An entry $[i,j]$ in each table 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 $b_1 b_2 \ldots b_j$, with the same restriction on matching a_i and b_j as in Section 3. Alignment of empty suffixes is valid in T_1, and is assigned a score of 0. Therefore, Eq. (1) is modified in the following way, and Eqs. (2) and (3) remain the same.

$$T_1[i,j] = f(a_i, b_j) + \max \begin{cases} T_1[i-1,j-1], \\ T_2[i-1,j-1], \\ T_3[i-1,j-1], \\ 0. \end{cases}$$

The maximum score in T_1 is the optimal score. An optimal alignment is retrieved by performing a traceback from a maximum entry in T_1 until a score of 0 is reached. It is easy to extend the parallel prefix-based algorithm to the subsequence matching problem. Let

$$w[j] = \max \begin{cases} T_1[i,j-1] - (g+h), \\ T_3[i,j-1] - (g+h), \\ 0. \end{cases}$$

With this modification for computing the $w[j]$'s, the rest of the algorithm is the same as before. Each processor keeps track of the largest entry in its portion of the table. The largest entry in the whole table is computed using a simple reduction operation. It is straightforward to extend the space-saving and parallel traceback techniques presented earlier to the subsequence matching problem.

Huang [9] presented a sequential, space-saving, subsequence matching algorithm that results in faster run-times when the subsequences resulting in optimal alignment are much shorter than the sequences. The algorithm consists of (1) computing the three tables as mentioned above but only keeping track of a largest entry and its position ($[i,j]$) to identify the end of an optimal subsequence alignment, (2) running a sequence comparison algorithm on $a_i a_{i-1} \ldots a_1$ and $b_j b_{j-1} \ldots b_1$ to locate a largest entry, which corresponds to the beginning ($[k,l]$) of a subsequence alignment that ends at $[i,j]$, and (3) running a space-saving sequence comparison algorithm on $a_k a_{k+1} \ldots a_i$ and $b_l b_{l+1} \ldots b_j$ with the ability to do traceback. Each of these components can be parallelized using the algorithms already presented in this paper.

6. Experimental results

We implemented the parallel space-saving algorithm for sequence comparison using affine gap penalty functions. The program is written in C and MPI and run on an IBM SP-2 with 160 MHZ thin nodes and a Pentium cluster with 333 MHZ processors connected by Myrinet, capable of supporting communication rates of up to 1.28 Gb/s. The parallel space-saving algorithm consists of a parallel phase, followed by a serial phase: In the parallel phase, the tables are computed in parallel, storing entries only on the special columns. This is followed by a traceback procedure to split the problem into p subproblems. In the serial phase, the subproblems are solved independently on each processor. The time spent in the parallel phase depends only on the size of the tables. Even though the time spent in the serial phase is bounded by $O(\frac{mn}{p})$, the actual time spent depends upon how evenly the problem splits into subproblems, which in turn depends upon the structure of the optimal alignment. To bring out the differences in the serial phase, we tested the program using two types of data: (1) identical sequences, resulting in a unique optimal alignment along the diagonal, and (2) sequences that use distinct characters, with a gap penalty of 0 and mismatch score of -1 so that optimal alignment corresponds to each sequence completely aligned with gaps. The former results in exactly balanced subproblems and is the best case. In the latter case, one processor receives a subproblem of size $m \times \frac{n}{p}$ and all others receive a subproblem of size $1 \times \frac{n}{p}$. This represents the worst case. We refer to the two cases as *complete match* and *complete mismatch*, respectively (see Fig. 2).

The program is run for the complete match and complete mismatch cases using sequences of the same length, varying the problem size and number of processors. The total run-time and the time spent in the parallel and serial phases for a particular problem size on both parallel computers are summarized in Tables 1 and 2. As expected, the run-times spent in the parallel phase are approximately the same for both the complete match and the complete mismatch cases. The

speedups obtained over running the program on one processor for various problem sizes (fixed-size speedup) are shown in Figs. 3 and 4.

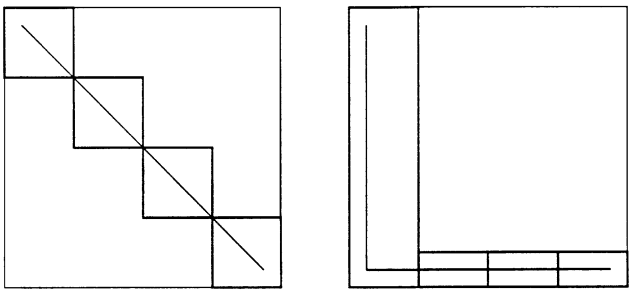

Fig. 2. Problem decomposition in complete match and complete mismatch cases.

For reasonably large problem sizes, near perfect scaling is observed. Notice the super linear speed up in the complete match case, which is more readily observable in case of the SP-2. This is due to the fact that increase in the number of special columns reduces total work. Based on an approximate calculation (ignoring lower order terms, etc.), our parallel algorithm requires computing 4 mn entries per table in the parallel phase. Note that parallel prefix on a row of a table requires processing each table entry twice, and only one of the three tables needs parallel prefix. In the complete match case, each processor computes $\frac{6\,mn}{p^2}$ entries in the serial phase. In the complete mismatch case, the processor with the heaviest load computes $\frac{6\,mn}{p}$ entries in the serial phase. Thus, the total work done by our

Table 1
Running time in seconds on SP-2 for $m = n = 65536$

Number of processors	Complete match case			Complete mismatch case		
	Parallel phase	Sequential phase	Total time	Parallel phase	Sequential phase	Total time
1	—	—	41701	—	—	43845
2	10377	4971	15348	10494	10323	20817
4	5258	1238	6496	5691	5115	10806
8	2582	309	2891	2733	2402	5135
16	1504	79	1583	1519	1103	2622
32	943	14	957	921	473	1394

Table 2
Running time in seconds on Pentium cluster for $m = n = 65536$

Number of processors	Complete match case			Complete mismatch case		
	Parallel phase	Sequential phase	Total time	Parallel phase	Sequential phase	Total time
1	—	—	25487	—	—	25227
2	8887	1815	10702	9148	3499	12647
4	5768	433	6201	5748	1634	7382
8	3154	233	3387	2981	786	3767
16	1617	58	1675	1554	393	1947
32	829	8	837	807	197	1004

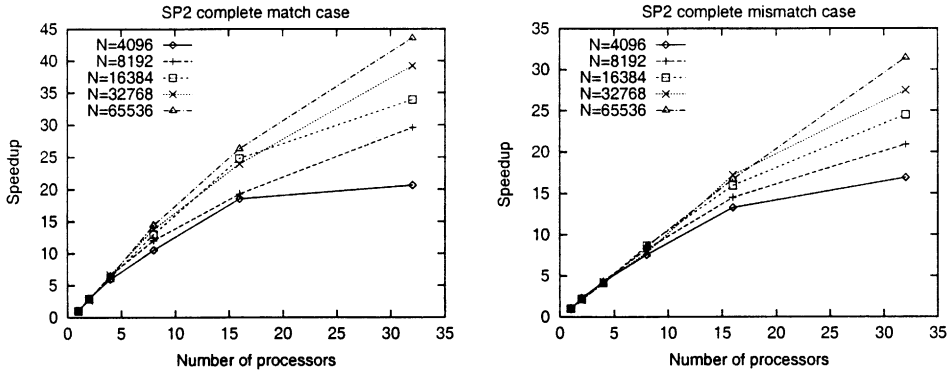

Fig. 3. Fixed-size speedups on IBM SP-2 for various problem sizes. N denotes the length of both input sequences.

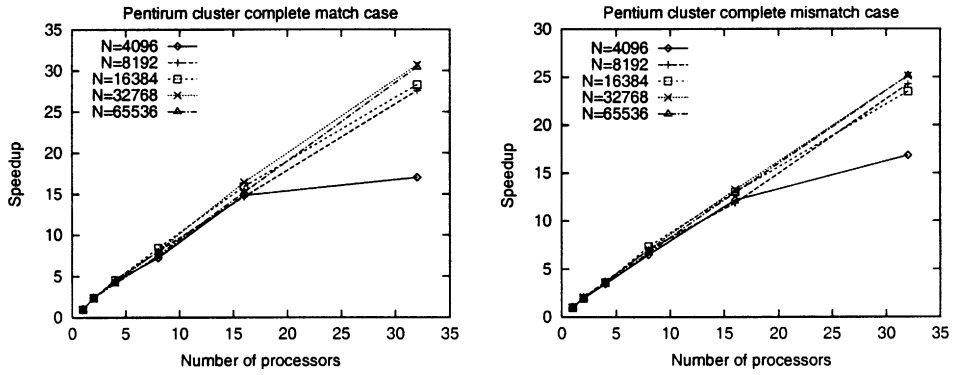

Fig. 4. Fixed-size speedups on Pentium cluster for various problem sizes. N denotes the length of both input sequences.

parallel algorithm is $4\,mn + \frac{6\,mn}{p}$ in the case of complete match and $10\,mn$ in the case of complete mismatch including the idle time on processors (taking work to be the product of parallel run-time multiplied by the number of processors). The superlinear speedup is more evident in the case of SP-2 because it has better communication latency with respect to the time for unit computation when compared with the Myrinet Pentium cluster.

7. Conclusions

In this paper we used prefix computation to derive parallel algorithms 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. The scalability of our algorithms is demonstrated by experimental validation on an IBM SP-2 and a Pentium cluster. Our methodology, discussed in the context of the sequence comparison problems, is general and can be used to parallelize other dynamic-programming problems. Developing a space and time-optimal parallel algorithm for sequence comparison is still an open problem.

Acknowledgments

We gratefully acknowledge the reviewers for their valuable suggestions, and the Maui High-Performance Computing Center for granting access to their IBM SP-2.

References

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

[2] M.O. Dayhoff, R. Schwartz, B.C. Orcutt, A model of evolutionary change in proteins: matrices for detecting distant relationships, in: M.O. Dayhoff (Ed.), Atlas of Protein Sequence and Structure, Vol. 5, National Biomedical Research Foundation, DC, 1978, pp. 345–358.

[3] E.W. Edmiston, N.G. Core, J.H. Saltz, R.M. Smith, Parallel processing of biological sequence comparison algorithms, Internat. J. Parallel Programming 17 (3) (1988) 259–275.

[4] E.W. Edmiston, R.A. Wagner, Parallelization of the dynamic programming algorithm for comparison of sequences, Proceedings of the International Conference on Parallel Processing, 1987, pp. 78–80.

[5] O. Gotoh, An improved algorithm for matching biological sequences, J. Molec. Biol. 162 (1982) 705–708.

[6] S. Henikoff, J.G. Henikoff, Amino acid substitution matrices from protein blocks, Proc. Nat. Acad. Sci. 89 (1992) 10915–10919.

[7] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Commun. ACM 18 (6) (1975) 341–343.

[8] X. Huang, A space-efficient parallel sequence comparison algorithm for a message-passing multiprocessor, Internat. J. Parallel Programming 18 (3) (1989) 223–239.

[9] X. Huaug, A space-efficient algorithm for local similarities, Comput. Appl. Biosci. 6 (4) (1990) 373–381.

[10] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[11] E. Lander, J.P. Mesirov, W. Taylor, Protein sequence comparison on a data parallel computer, Proceedings of the International Conference on Parallel Processing, 1988, pp. 257–263.

[12] W.J. Masek, M.S. Paterson, A faster algorithm for computing string edit distances, J. Comput. System Sci. 20 (1980) 18–31.

[13] E.W. Mayers, W. Miller, Optimal alignments in linear space, Comput. Appl. Biosci. 4 (1) (1988) 11–17.

[14] S.B. Needleman, C.D. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, J. Molec. Biol. 48 (1970) 443–453.

[15] S. Ranka, S. Sahni, String editing on an SIMD hypercube multicomputer, J. Parallel Distrib. Comput. 9 (1990) 411–418.

[16] J. Setubal, J. Meidanis, Introduction to Computational Molecular Biology, PWS Publishing Company, Boston, MA, 1997.

[17] T.F. Smith, M.S. Waterman, Identification of common molecular subsequences, J. Molec. Biol. 147 (1981) 195–197.

Srinivas Aluru is an Associate Professor in the Department of Electrical and Computer Engineering and the Laurence H. Baker Center for Bioinformatics and Biological Statistics at Iowa State University. Previously, he held faculty positions at New Mexico State University (1996–1999) and Syracuse University (1994–1996). He received his B. Tech degree in Computer Science from Indian Institute

of Technology, Madras in 1989, and his M.S. and Ph.D. degrees in Computer Science from Iowa State university in 1991 and 1994, respectively. His research interests include parallel algorithms and applications, computational biology and scientific computing. He is a recipient of the NSF CAREER award, an IBM Faculty Award and a Young Engineering Faculty Research Award from Iowa State University. He is a member of ACM, and a senior member of IEEE and the IEEE Computer Society.

Natsuhiko Futamura received his B.S. degree in Mathematics from Waseda University, Tokyo in 1992. He received his M.S. and Ph.D. degrees in Computer Science from Syracuse University in 1996 and 2002, respectively. He is currently an Assistant Professor in the Department of Computer Science at Wright State University. His research interests include parallel and distributed computing and computational biology.

Kishan G. Mehrotra received his M.Sc. degree from Lucknow University, India in 1962 and MS and Ph.D. from the University of Wisconsin, Madison, in 1968 and 1970, respectively, in statistics. Currently he is a Professor of Computer and Information Science at Syracuse University, Syracuse, NY. Since November 1998 he is the Director of the Computer and Information Science Program in the Department of Electrical Engineering and Computer Science, Syracuse University. Mehrotra's research interests include algorithms, computer performance evaluation, neural networks, and genetic algorithms. He is coauthor of "Elements of Neural Networks" and more than 100 review articles.