Space and Time Optimal Parallel Sequence Alignments

Stjepan Rajko and Srinivas Aluru, *Senior Member*, *IEEE*

Abstract—We present the first space and time optimal parallel algorithm for the pairwise sequence alignment problem, a fundamental problem in computational biology. This problem can be solved sequentially in $O(mn)$ time and $O(m+n)$ space, where m and n are the lengths of the sequences to be aligned. The fastest known parallel space-optimal algorithm for pairwise sequence alignment takes optimal $O\left(\frac{m+n}{p}\right)$ space, but suboptimal $O\left(\frac{(m+n)^2}{p}\right)$ time, where p is the number of processors. On the other hand, the most space economical time-optimal parallel algorithm takes $O\left(\frac{mn}{p}\right)$ time, but $O\left(m+\frac{n}{p}\right)$ space. We close this gap by presenting an algorithm that achieves both time and space optimality, i.e. requires only $O\left(\frac{m+n}{p}\right)$ space and $O\left(\frac{mn}{p}\right)$ time. We also present an experimental evaluation of the proposed algorithm on an IBM xSeries cluster. Although presented in the context of full sequence alignments, our algorithm is applicable to other alignment problems in computational biology including local alignments and syntenic alignments. It is also a useful addition to the range of techniques available for parallel dynamic programming.

Index Terms—Computational biology, sequence alignments, space-efficient, parallel algorithms, parallel dynamic programming.

1 Introduction

Pairwise sequence alignment is an important fundamental problem in computational biology, and sequence alignments are the mainstay of molecular biology research. Sequence alignment algorithms typically use dynamic programming in which a table, or multiple tables of size $(m+1)\times(n+1)$ are filled, where m and n are the lengths of the two sequences. Several researchers have explored sequence alignment algorithms [18], [22], culminating in the solution of a variety of sequence alignment problems, including subsequence alignments, in $O(mn)$ time and space [8]. Using the technique of Hirschberg [10], developed in the context of the longest common subsequence problem, Mayers and Miller [17] presented a technique to reduce the space requirement of sequence alignment to optimal $O(m+n)$, while retaining the time complexity of $O(mn)$. Huang [13] extended this algorithm to subsequence alignments. These algorithms are very important because the lengths of biological sequences can be large enough to render algorithms that use quadratic space infeasible. An asymptotically faster sequential algorithm for sequence alignment that runs in $O(\frac{n^2}{\log^2 n})$ time in the unit-cost RAM model is given by Masek and Paterson [16]. However, this algorithm is rarely used in practice and is not expected to be faster unless the sequences are extremely large.

While space-optimal algorithms make large sequence alignment 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 computers. Edmiston et al. [6] present parallel algorithms for sequence and subsequence alignment that achieve linear speedup and can use up to $O(\min(m,n))$ processors. Lander et al. [15] discuss implementation on a data parallel computer. These algorithms store the entire dynamic programming table.

A widely studied problem that is identical to a special case of the sequence alignment 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 [3], [20], 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. Recently, Alves et al. [2] present a CGM/BSP algorithm for the string editing problem, which also uses $O\left(\frac{mn}{p}\right)$ memory per processor. From a practical standpoint, space-efficiency is important to align large sequences.

Huang [12] presented a parallel sequence alignment algorithm that uses optimal $O(\frac{m+n}{p})$ space, at the expense of increasing the runtime to $O(\frac{(m+n)^2}{p})$. However, this runtime is optimal for the special case of $m=\Theta(n)$. Aluru et al. [1] presented an algorithm that retains time optimality, but uses $O\left(m+\frac{n}{p}\right)$ space. In this paper, we present a parallel algorithm that solves the open problem of simultaneously achieving space and time optimality. The algorithm is suitable for implementation on parallel computers and we demonstrate this by presenting experimental results on an IBM xSeries cluster.

- S. Rajko is with the Department of Computer Science, Arizona State University, Tempe, AZ 85287. E-mail: stjepan.rajko@asu.edu.
- S. Aluru is with the Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011. E-mail: aluru@iastate.edu.

Manuscript received 18 May 2003; revised 31 Mar. 2004; accepted 4 June 2004.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0077-0503.

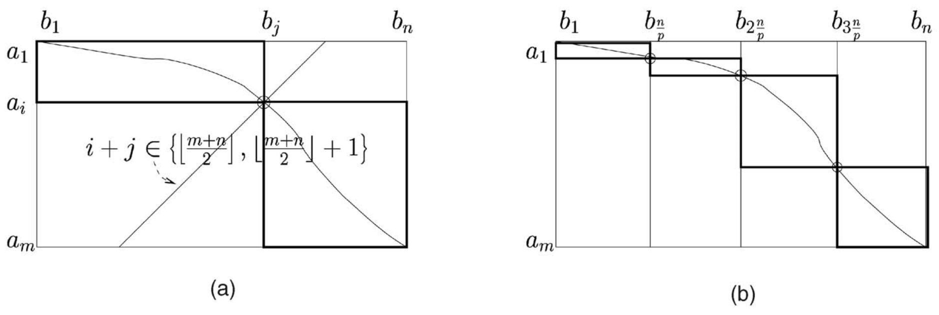


Fig. 1. Problem decomposition by finding intersection(s) with diagonal (a) or p columns ((b); for $p = 4$).

Our techniques are applicable to other types of sequence alignment problems in computational biology including semiglobal alignments [21], local alignments [21], and syntenic alignments [14]. They have potential applications to other problems, not necessarily from computational biology, whose solutions involve parallel dynamic programming.

2 Comparison with Related Work

In this section, we present a brief synopsis of the ideas underlying previous attempts at optimizing parallel runtime and space usage per processor. Sequence alignment algorithms can be visualized as finding an optimal path in a rectangular dynamic programming table from the top-left corner to the bottom right corner. An entry $[i, j]$ in the dynamic programming table depends upon three other entries $[i-1, j]$, $[i, j-1]$, and $[i-1, j-1]$. Define a k-diagonal in the table to be the cells whose row and column numbers add up to k. Sequentially, the table can be filled row-wise, column-wise, or diagonal-wise such that an entry is filled before it is needed for computing other entries. Most parallel algorithms fill the table diagonal-wise because the entries required for filling a diagonal are all contained in the previous two diagonals. As there are no intradependencies within a diagonal, each diagonal can be computed in parallel. Recently, Aluru et al. [1] show how to perform row-wise or column-wise parallel computation, despite the dependencies that exist within a row or column. This approach leads to perfect workload partitioning (row and column sizes are fixed whereas the diagonal sizes vary) and less interprocessor communication. Let $a_1, a_2, \ldots, a_m$ and $b_1, b_2, \ldots b_n$ be the two sequences to be aligned using p processors. Huang [12] presented the first and only previously known space-optimal parallel sequence alignment algorithm. This is based on the clever idea of finding the intersection of an optimal path with the $\lfloor\frac{m+n}{2}\rfloor$-diagonal or the $\left(\lfloor\frac{m+n}{2}\rfloor+1\right)$-diagonal using $O\left(\frac{m+n}{p}\right)$ parallel space. Let $[i, j]$ denote the cell where the intersection occurs. We can now decompose the problem into two subproblems, one for aligning $a_1, a_2, \ldots, a_i$ with $b_1, b_2, \ldots, b_j$ and the other for aligning $a_{i+1}, a_{i+2}, \ldots, a_m$ with $b_{j+1}, b_{j+2}, \ldots, b_n$ (illustrated in Fig. 1a). The important feature of this decomposition is that $i + j$ and $(m - i) + (n - j)$ are both approximately $\frac{m+n}{2}$. Thus, by allocating $\frac{p}{2}$ processors to each subproblem and solving recursively, space consumption of $O\left(\frac{m+n}{p}\right)$ can be guaranteed for subsequent iterations. This increases the parallel runtime to $O\left(\frac{(m+n)^2}{p}\right)$, which is still optimal when $m = \Theta(n)$. Algorithms to retain time optimality and reduce space requirement are first presented by Edmiston et al. [6] and further developed by Aluru et al. [1]. In this approach [1], the intersection of an optimal path with p equally spaced special columns is determined using $O\left(\frac{mn}{p}\right)$ parallel work, decomposing the problem into p subproblems directly. Each subproblem is solved sequentially within a processor, using Mayers and Miller's adaptation of Hirschberg's sequential, space-saving algorithm [10], [17]. As the lengths of the sequences assigned to each subproblem are $O(m)$ and $\frac{n}{p}$, optimal parallel runtime can be achieved. This approach requires $O\left(m + \frac{n}{p}\right)$ space, and is illustrated in Fig. 1b.

3 The Sequence Alignment Problem

We formalize the sequence alignment problem as follows: Suppose we are given two sequences over an alphabet Σ, $A' = a_1, a_2, \ldots, a_m$ and $B' = b_1, b_2, \ldots, b_n$ $(m \leq n)$, as well as a scoring function $f : \Sigma \times \Sigma \rightarrow \mathbb{R}$ and a gap penalty function. The goal is to find an optimal way to align the two sequences by inserting gaps ("$-$") into either or both of them, so that:

- Each character in A' (respectively, B') is aligned with either a character in B' (respectively, A') or a gap.
- The total score obtained by summing the scores as given by the scoring function over the aligned pairs of characters, and subtracting the penalties for gaps as given by the gap penalty function, is maximized.

To penalize gaps, an *affine gap penalty function* is commonly used, where a maximal consecutive sequence of k gaps is given a penalty of the form $h + gk$. In other words, the first gap in such a sequence is charged $h + g$, and the rest are charged g each. When $h = 0$, the penalty function is called a *constant gap penalty function*.

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:

Sequences	Alignment	Alignment Model
ATGTCGA	A T G – – T C G A	((1,1),(2,0),(3,2),(0,3),(0,4),
AGAATCTA	A – G A A T C T A	(4,5),(5,6),(6,7),(7,8))

Fig. 2. An example alignment and its corresponding list of pairs representation.

$$f(c_1, c_2) = \begin{cases} 2, & c_1 = c_2,\ c_1, c_2 \in \Sigma \\ 0, & c_1 \neq c_2,\ c_1, c_2 \in \Sigma \end{cases}$$

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 3.

A	T	G	–	–	T	C	G	A
A	–	G	A	A	T	C	T	A
2	−3	2	−4		2	2	0	2

We model an alignment by a list of ordered pairs, where each element is either a pair of matched characters, or a match between a character and a gap. A character in a pair is represented by its position in the sequence it comes from, and a gap is represented by 0. See Fig. 2 for an example.

When aligning substrings of A' and B', we still use original positions of characters in A' and B' to model the alignment, as opposed to using their positions relative to the substrings. For example, an alignment of GT and $GAAT$ that corresponds to the alignment in Fig. 2 would be modeled by $((3,2),(0,3),(0,4),(4,5))$, and not $((1,1),(0,2),(0,3),(2,4))$.

Formally, if we let $A = a_{i'}, a_{i'+1}, \ldots, a_{i''}$ and $B = b_{j'}, b_{j'+1}, \ldots, b_{j''}$ be substrings of A' and B', we model an alignment of A and B as a list of ordered pairs $C = ((i_1, j_1), (i_2, j_2), \ldots, (i_{|C|}, j_{|C|}))$, where for all k, $i_k \in \{i', i'+1, \ldots, i''\} \bigcup \{0\}$ and $j_k \in \{j', j'+1, \ldots, j''\} \bigcup \{0\}$. $(i, j) \in C$ with both $i > 0$ and $j > 0$ means that a_i is matched with b_j. We call such elements of C to be of type 1. If i (respectively, j) is 0, then b_j (respectively, a_i) is matched with a gap, and such elements of C are of type 2 (respectively, type 3). Matching a gap with a gap is not allowed. To denote the type of the kth element of C (in this case, (i_k, j_k)) as defined above, we use $C_{type}(k)$.

By specifying positions relative to the original sequences A' and B', this modeling approach allows us to easily combine alignments of consecutive substrings, or to subdivide an alignment into subalignments. We formalize this notion in the following two observations, which we state without proof (+ denotes concatenation).

Suppose i and j are given such that $i' \leq i \leq i''$ and $j' \leq j \leq j''$, and let $A_1 = a_{i'}, a_{i'+1}, \ldots, a_i$, $A_2 = a_{i+1}, a_{i+2}, \ldots, a_{i''}$, $B_1 = b_{j'}, b_{j'+1}, \ldots, b_j$, and $B_2 = b_{j+1}, b_{j+2}, \ldots, b_{j''}$.

Observation 1. *Suppose C_1 is an alignment of A_1 and B_1, and C_2 is an alignment of A_2 and B_2. Then, $C_1 + C_2$ is an alignment of A and B.*

Observation 2. *If $C = ((i_1, j_1), (i_2, j_2), \ldots, (i_{|C|}, j_{|C|}))$ is an alignment of A and B, and k is such that $k' \leq k$ implies $i_{k'} \leq i$ and $j_{k'} \leq j$; and $k' > k$ implies $i_{k'} > i$ (or $i_{k'} = 0$) and $j_{k'} > j$ (or $j_{k'} = 0$), then $((i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k))$ is an alignment of A_1 and B_1, and $((i_{k+1}, j_{k+1}), (i_{k+2}, j_{k+2}), \ldots, (i_{|C|}, j_{|C|}))$ is an alignment of A_2 and B_2.*

Our algorithm depends on the ability to recursively subdivide the alignment problem. When using constant gap penalties, knowing some minimal information about an optimal alignment (for example, that some a_i is matched with a gap between b_j and b_{j+1}) allows us to divide the problem of finding an optimal alignment into two subproblems. In the affine gap penalty case, however, because of additional penalization of the first gap in a gap sequence, the situation is a little more complicated. Consider aligning sequences AA and BBBB, using the scoring function and gap penalty function from the above example. Then, as shown in Fig. 3, C is an optimal alignment of AA and BBBB, but C' is not, even though C'' is an optimal alignment of A and BB.

To allow subdivision in the affine gap penalty case, we define an *extended sequence alignment problem*. In addition to sequences A and B, a gap penalty function $h + gk$, and a scoring function, the extended sequence alignment problem has a $start\_type$ and an end_type, both of which must be in $\{+, -\} \times \{1, 2, 3\}$. The $\{+, -\}$ component gives us two different ways of placing a preference on alignments that start or end with matches of a particular type. The $\{1, 2, 3\}$ component specifies the type we prefer. If $start\_type = (+, t)$ (respectively, $end\_type = (+, t)$), then we allow only alignments whose first (respectively, last) element is of type t. If $start\_type = (-, t)$ (respectively, $end\_type = (-, t)$), we do not place any restrictions on the alignments, but we do modify the score of alignments whose first (respectively, last) element is of type t.

In particular, a $start\_type$ or end_type of $(-, 2)$ (respectively, $(-, 3)$) allows us to specify whether a maximal sequence of gaps of type 2 (respectively, type 3) at the start or end of the alignment should be penalized by $h + gk$ or just gk. A $start\_type$ or end_type of $(+, 1)$, $(+, 2)$, or $(+, 3)$ gives a score of $-\infty$ to any alignment whose first or last element does not match the specified type. Only alignments with a finite score (not $-\infty$) are considered to be valid alignments. For simplicity, instead of $\{+, -\} \times \{1, 2, 3\}$, we will use $\{3, 2, 1, -1, -2, -3\}$.

C				C'				C''	
A	–	–	A	A	–	A	–	A	–
B	B	B	B	B	B	B	B	B	B

Fig. 3. Counter-example to problem decomposition in affine gap penalty case: C is an optimal alignment, but C' is not, even though C'' is.

An alignment C of A and B that would have a score $sc(C)$ in the standard alignment problem would have a score $esc(C)$ for the extended alignment problem, where:

$$esc_s(C) = \begin{cases} h, & \text{If } start\_type \in \{-2,-3\} \text{ and } C_{type}(1) = start\_type \\ -\infty, & \text{If } start\_type > 0 \text{ and } C_{type}(1) \neq start\_type \\ 0, & \text{otherwise,} \end{cases}$$

$$esc_e(C) = \begin{cases} h, & \text{If } end\_type \in \{-2,-3\} \text{ and } C_{type}(|C|) = end\_type \\ -\infty, & \text{If } end\_type > 0 \text{ and } C_{type}(|C|) \neq end\_type \\ 0, & \text{otherwise,} \end{cases}$$

$$esc(C) = sc(C) + esc_s(C) + esc_e(C).$$

We will continue to use $esc(C)$ to denote the modified score of an alignment under the extended alignment problem, $sc(C)$ to denote the score the alignment would have in the corresponding standard alignment problem, and $esc_s(C)$ and $esc_e(C)$ to denote the appropriate adjustments between the two.

A solution to the extended alignment problem is an alignment C with maximal finite score. An extended alignment problem with both *start_type* and *end_type* of -1 is equivalent to the standard alignment problem. We will refer to an extended sequence alignment problem instance as a 4-tuple $(A, B, start\_type, end\_type)$, with the assumption that the scoring function and the gap penalty function are given. We will denote the set of alignments of A and B that have a finite (valid) score as $V(A, B, start\_type, end\_type)$, and denote the set of optimal alignments (solutions) as $S(A, B, start\_type, end\_type)$.

4 Parallel Optimal Sequence Alignment Algorithm

We will begin this section by showing that the extended alignment problem framework allows us to divide an alignment problem in such a way that optimal alignments to subproblems can be combined to form an optimal alignment to the original problem. Basically, if we know that the optimal alignment has a match of a certain type at a certain position, we can divide the problem into two parts around that position. By specifying the *end_type* of the left subproblem and the *start_type* of the right subproblem according to the type of match at the position, we ensure that optimal alignments to the subproblems can be combined into an optimal alignment to the original problem.

We first develop some notation that will help us refer to the position where the alignment is to be split, and then proceed to prove the results. After reading through the description of the notation which will also be used subsequently, a reader not interested in the technical details can safely skip the statement and proof of Propositions 1-4, as long as their general purpose as outlined above is understood.

We are given the extended alignment problem $(A', B', -1, -1)$, where $A' = a_1, a_2, \dots, a_m$ and $B' = b_1, b_2, \dots, b_n$. Take any substrings $A = a_{i'}, a_{i'+1}, \dots, a_{i''}$ and $B = b_{j'}, b_{j'+1}, \dots, b_{j''}$ of A' and B', and an extended alignment problem (A, B, s, e).

Let $C = ((i_1, j_1), (i_2, j_2), \dots, (i_{|C|}, j_{|C|})) \in V(A, B, s, e)$. Define $C_a(i)$ for $i \in \{i', i'+1, \dots, i''\}$ as follows: If a_i is matched with some b_j in C, then $C_a(i) = j$. Otherwise, if $i \neq i'$, then let $C_a(i) = C_a(i-1)$, and if $i = i'$, then let $C_a(i) = j' - 1$. Intuitively, $j = C_a(i)$ tells us that a_i is matched either with b_j, or a gap between b_j and b_{j+1}. We can define $C_b(j)$ for $j \in \{j', j'+1, \dots, j''\}$ symmetrically.

Let $C^* = \{(i,j) | j = C_a(i) \text{ or } i = C_b(j)\}$, and for any $(i, j) \in C^*$, we define

$$C_{max}(i,j) = \max\{k | (i_k, j_k) \in \{(i,j), (i,0), (0,j)\}\},$$
$$C_{left}(i,j) = C_{type}(C_{max}(i,j)),$$
$$C_{right}(i,j) = C_{type}(C_{max}(i,j) + 1).$$

$C_{max}(i, j)$ is simply the index of the last occurrence of either a_i or b_j in the alignment. C_{left} and C_{right} are shorthand for the type of match at that and the following position.

Now, fix (i, j) so that $(i, j) \in C^*$. Let $A_1 = a_{i'} a_{i'+1} \dots a_i$, $A_2 = a_{i+1} a_{i+2} \dots a_{i''}$, $B_1 = b_{j'} b_{j'+1} \dots b_j$, and $B_2 = b_{j+1} b_{j+2} \dots b_{j''}$.

Proposition 1.

$$C'_1 \in V(A_1, B_1, s, e'), C'_2 \in V(A_2, B_2, s', e) \Rightarrow$$
$$C'_1 + C'_2 \in V(A, B, s, e).$$

Proof. By Observation 1, $C'_1 + C'_2$ is an alignment of A and B, so we only need to show that $esc(C'_1 + C'_2) \neq -\infty$. $esc_s(C'_1) \neq -\infty$ implies $esc_s(C'_1 + C'_2) \neq -\infty$ because their first elements are identical. $esc_e(C'_2) \neq -\infty$ implies $esc_e(C'_1 + C'_2) \neq -\infty$ because their last elements are identical. Hence, $esc(C'_1 + C'_2) \neq -\infty$. □

The following two Propositions hold for t that satisfies either $t = C_{left}(i, j)$, or $-t = C_{right}(i, j)$. The proofs presented will be only for the first case. Proofs for the second case are similar.

Proposition 2. *Recall that* $C = ((i_1, j_1), (i_2, j_2), \dots, (i_{|C|}, j_{|C|})) \in V(A, B, s, e)$. *Suppose* $k = C_{max}(i, j)$ *and* t *is as noted above. Then,* $C_1 = ((i_1, j_1), (i_2, j_2), \dots, (i_k, j_k)) \in V(A_1, B_1, s, t)$ *and* $C_2 = ((i_{k+1}, j_{k+1}), (i_{k+2}, j_{k+2}), \dots, (i_{|C|} j_{|C|})), \in V(A_2, B_2, -t, e)$.

Proof. By Observation 2, C_1 is an alignment of A_1 and B_1, and C_2 is an alignment of A_2 and B_2, so we only need to show that $esc(C_1) \neq -\infty$ and $esc(C_2) \neq -\infty$. Since $C \in V(A, B, s, e)$, $esc_s(C) \neq -\infty$, and $esc_e(C) \neq -\infty$, which in turn imply $esc_s(C_1) \neq -\infty$ and $esc_e(C_2) \neq -\infty$. Since $(C_1)_{type}(|C_1|) = C_{type}(k) = t$, $esc_e(C_1) \neq -\infty$. Since $-t < 0$, $esc_s(C_2) \neq -\infty$. Therefore, $esc(C_1) \neq -\infty$ and $esc(C_2) \neq -\infty$. □

Proposition 3. *Take any* C'_1, C'_2*, and* $C' = C'_1 + C'_2$ *s.t.*

$$C'_1 \in V(A_1, B_1, s, t), C'_2 \in V(A_2, B_2, -t, e), C' \in V(A, B, s, e).$$

Then, $esc(C') = esc(C'_1) + esc(C'_2)$.

Proof. By definition, $esc(C) = sc(C) + esc_s(C) + esc_e(C)$ for any alignment C. Regardless of t, we have that $esc_s(C') = esc_s(C'_1)$ because they have the same *start_type* and the same initial element. Similarly, $esc_e(C') = esc_e(C'_2)$. Also, since $t > 0$, $esc_e(C'_1) = 0$. Therefore, it suffices to show that $sc(C') = sc(C'_1) + sc(C'_2) + esc_s(C'_2)$. Also, note that if C'_1

optimal alignment C				partial balanced partition
$\mathbf{a_1} \ldots \mathbf{a_8}$	$\mathbf{a_9} \ldots \mathbf{a_{16}}$	$\ldots a_{40}$ −	$\mathbf{a_{41}} \ldots \mathbf{a_{48}}$	$(0,0,-1),(8,7,-1),(16,33,-1),$
$\mathbf{b_1} \ldots \mathbf{b_7}$	$b_8 \ldots b_{33}$	$\ldots \mathbf{b_{35}}\, \mathbf{b_{36}}$	$b_{37} \ldots b_{48}$	$(40,36,-2),(48,48,1)$

Fig. 4. An example of a partial balanced partition with $n = m = 48$ and $p = 6$.

ends with a gap and C'_2 begins with a gap (and the gaps are both in the same sequence), we require $sc(C') = sc(C'_1) + sc(C'_2) + h$. Otherwise, we require $sc(C') = sc(C'_1) + sc(C'_2)$. Consider the case when $t = 1$. Then, $esc_s(C'_2) = 0$, and $sc(C') = sc(C'_1) + sc(C'_2)$, as required. Now, consider the case when $t = 2$. If C'_2 begins with a gap in the A sequence, then $esc_s(C'_2) = h$, and $sc(C') = sc(C'_1) + sc(C'_2) + h$. Otherwise, $esc_s(C'_2) = 0$, and $sc(C') = sc(C'_1) + sc(C'_2)$. In either case, $sc(C') = sc(C'_1) + sc(C'_2) + esc_s(C'_2)$, as required. The $t = 3$ case is symmetric. □

Propositions 1-3 are concerned with the combination, division, and scoring of valid alignments. We now use them to reason about optimal alignments.

Proposition 4. *If* $C \in S(A, B, s, e)$, *then*

$$C'_1 \in S(A_1, B_1, s, t), C'_2 \in S(A_2, B_2, -t, e) \Rightarrow C'_1 + C'_2 \in S(A, B, s, e).$$

Proof. By Proposition 2, C splits up into some $C_1 \in V(A_1, B_1, s, t)$ and $C_2 \in V(A_2, B_2, -t, e)$. Using Propositions 1 and 3, it is easy to show that C_1 and C_2 are optimal, i.e., $C_1 \in S(A_1, B_1, s, t)$ and $C_2 \in S(A_2, B_2, -t, e)$. In particular, if one of them was not optimal, we could replace it with a better alignment and produce an alignment better than C, a contradiction. Similarly, Propositions 1 and 3 also tell us that any $C'_1 \in S(A_1, B_1, s, t)$ and $C'_2 \in S(A_2, B_2, -t, e)$ combine into an element of $S(A, B, s, e)$. □

Define a list $P = ((i_0, j_0, t_0), (i_1, j_1, t_1), \ldots, (i_{|P|-1}, j_{|P|-1}, t_{|P|-1}))$ to be a *a partial balanced partition* of A' and B' for p processors[1] if there is some optimal alignment C so that the following properties are satisfied:

1. For all $k \in \{0, 1, \ldots, |P| - 2\}$, $i_k \le i_{k+1}$ and $j_k \le j_{k+1}$.
2. $(i_0, j_0, t_0) = (0, 0, -1)$ and $(i_{|P|-1}, j_{|P|-1}, t_{|P|-1}) = (m, n, 1)$.
3. Either $i_{k+1} - i_k \le \frac{m}{p}$, or $j_{k+1} - j_k \le \frac{n}{p}$, or both.
4. Each element (i_k, j_k, t_k) with $0 < k < |P| - 1$ satisfies $(i_k, j_k) \in C^*$ and either $t_k = -C_{left}(i_k, j_k)$, or $t_k = C_{right}(i_k, j_k)$.

In other words, a partial balanced partition is simply a list of positions where an alignment problem can be split into subproblems according to Propositions 1-4. An example of a partial balanced partition is given in Fig. 4. For each subproblem, the figure outlines in bold the subsequence of bounded size as required by property 3. Having at least one subsequence of bounded size allows us to use an appropriate number of processors and the time optimal space saving algorithm in [1] to solve each subproblem in optimal space and time. This is because the algorithm in [1] uses $O(m' + \frac{n'}{p'})$ space when aligning sequences of lengths m' and n' on p' processors, as detailed in Section 2. By setting the m' length sequence to be a subsequence of bounded size, and assigning enough processors (p') to make $\frac{n'}{p'}$ either $O\left(\frac{n}{p}\right)$ or $O\left(\frac{m}{p}\right)$, the space consumption becomes

$$O\left(\frac{m+n}{p}\right).$$

Any partial balanced partition has sufficient information to achieve space and time optimal parallel alignment. The larger the partial balanced partition, the finer the size of the resulting subproblems, since each element of the partition gives us an additional position where we can split the original problem. However, finding finer partitions is computationally costly, and a coarser partition is quicker to compute. We focus on the case where the original problem is divided into at most p subproblems. We proceed to prove that a partition of that size is sufficient to solve the entire problem in optimal space and time, and then give our algorithm for finding such a partition in Section 5.

Suppose we have p processors, $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{p-1}$, to solve the sequence alignment problem, and that we also have a partial balanced partition $P = ((i_0, j_0, t_0), (i_1, j_1, t_1), \ldots, (i_{|P|-1}, j_{|P|-1}, t_{|P|-1}))$ of A' and B', such that $|P| \le p + 1$. We also assume that sequences A' and B' are distributed across the processors so that each processor holds $O(\frac{m+n}{p})$ data.

Define $A_k = a_{i_k+1} a_{i_k+2} \ldots a_{i_{k+1}}$ and $B_k = b_{j_k+1} b_{j_k+2} \ldots b_{j_{k+1}}$ for $k \in \{0, 1, \ldots, |P| - 2\}$. Also, define

$$w(k) = max\left\{\frac{|A_k|}{\frac{m}{p}}, \frac{|B_k|}{\frac{n}{p}}\right\}.$$

$w(k)$ represents the *weight* of subproblem k, and determines how many processors need to be assigned to it. Note that by property 3 of P, either $|A_k| = w(k)\frac{m}{p}$ and $|B_k| = O\left(\frac{n}{p}\right)$, or $|A_k| = O\left(\frac{m}{p}\right)$ and $|B_k| = w(k)\frac{n}{p}$. Therefore, by applying the space saving, optimal time parallel algorithm presented in [1], an optimal alignment between A_k and B_k using $\Theta(w(k))$ processors can be found in $O\left(\frac{mn}{p^2}\right)$ time and $O\left(\frac{m+n}{p}\right)$ space, as long as the longer subsequence is distributed across processors (the shorter subsequence can be replicated on all processors). A minor modification of that algorithm (the necessary details are presented in Section 5) will allow it to solve the extended alignment problem $(A_k, B_k, t_k, -t_{k+1})$, referred to as subproblem k, using asymptotically same space and time. By property 4 of P, and Proposition 4, the concatenation of optimal alignments of subproblem 0, subproblem $1, \ldots,$ subproblem $|P| - 2$, results in an optimal alignment of A' and B'.

1. For clarity of presentation, we assume that p divides both m and n.

subproblem(weight)	0(1)	1(3.25)				2(3)			3(1.5)	
work unit assignment	$\mathcal{P}_0$	$\mathcal{P}_0$	$\mathcal{P}_0$	$\mathcal{P}_1$	$\mathcal{P}_1$	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_2$	$\mathcal{P}_2$	$\mathcal{P}_3$
subproblem assignment	$\mathcal{P}_0$	$\mathcal{P}_0$, $\mathcal{P}_1$				$\mathcal{P}_1$, $\mathcal{P}_2$			$\mathcal{P}_2$, $\mathcal{P}_3$	

Fig. 5. Processor assignment for the partial balanced partition from Fig. 4. The partition consists of four subproblems, with one, four, three, and two work units, respectively. A simple strategy of assigning each processor to up to three consecutive work units yields the final processor assignment.

Many strategies can be used to determine the exact way the processors are assigned to subproblems. As a simple example, we assign $\lceil w(k) \rceil$ *work units* to each subproblem. A work unit represents a computational problem of size $O\left(\frac{mn}{p^2}\right)$, so to each subproblem we attempt to assign a number of processors proportional to its number of work units.

Note that since

$$w(k) = max\left\{\frac{|A_k|}{\frac{m}{p}}, \frac{|B_k|}{\frac{n}{p}}\right\},$$

we have $\sum_{k=0}^{|P|-2} w(k) \leq \frac{m}{\frac{m}{p}} + \frac{n}{\frac{n}{p}} = 2p$. So, $\sum_{k=0}^{|P|-2} \lceil w(k) \rceil \leq 3p$. In other words, there are no more than $3p$ work units total. If we substitute the list of subproblems by the corresponding list of work units and assign the processors in order to up to three consecutive work units, we will have a valid assignment. Note that trivially, each processor is assigned to at most three consecutive subproblems. An example is shown in Fig. 5. If P is distributed across processors so that each processor contains $O(1)$ elements, the assignment of processors to subproblems can be done in parallel using a **parallel prefix** [19]2 operation in $O(\log p)$ time and $O(1)$ space.

We distribute sequences A' and B' as follows: Each processor assigned to a subproblem receives a fraction of the longer subsequence, and one of the processors receives the entire shorter subsequence. Now, the subproblems can be solved using the space-saving parallel sequence alignment algorithm [1] and the following strategy: Since processors are assigned at most three consecutive problems, first solve subproblems $0, 3, 6, \ldots$ concurrently, then subproblems $1, 4, 7, \ldots$ concurrently, and then subproblems $2, 5, 8, \ldots$. This strategy will take $O\left(\frac{mn}{p^2}\right)$ time and $O\left(\frac{m+n}{p}\right)$ space. The processor assignment technique can be modified to limit the number of consecutive subproblems assigned to a processor to at most two. With this strategy, all the even subproblems can be solved concurrently and all the odd subproblems can be solved concurrently, reducing the number of concurrent groups from three to two.

Proposition 5. *Given p processors and a partial balanced partition P of A' and B' with $|P| \leq p+1$, an optimal alignment between A' and B' can be found in $O\left(\frac{m+n}{p}\right)$ space and $O\left(\frac{mn}{p^2}\right)$ time.*

Proof. The preceding algorithm proves this statement. Furthermore, the proof can easily be modified to show that the same is true for a partial balanced partition of any size. □

In the following section, we present a strategy to compute a partial balanced partition of size at most $p+1$ using $O\left(\frac{m+n}{p}\right)$ space and $O\left(\frac{mn}{p}\right)$ time.

5 Finding a Partial Balanced Partition in Parallel

To find a partial balanced partition, we use a technique that is similar to techniques commonly used in finding the alignment itself. We make use of three dynamic programming tables: T_1, T_2, and T_3. Each table can be regarded to be of size $(m+1) \times (n+1)$, but we will never actually store the tables completely, and we will only use portions of them when dealing with subproblems of the original problem.

Let $A = a_{i'}, a_{i'+1}, \ldots, a_{i''}$ and $B = b_{j'}, b_{j'+1}, \ldots, b_{j''}$ be substrings of A' and B', and suppose we are dealing with the subproblem (A, B, s, e). Then, we only use cells $[i, j]$ such that $i' - 1 \leq i \leq i''$ and $j' - 1 \leq j \leq j''$ in each table, with the value of the entry corresponding to the score for optimally aligning $a_{i'} a_{i+1} \ldots a_i$ with $b_{j'} b_{j'+1} \ldots b_j$, but with the following conditions: In T_1, a_i must be matched with b_j. In T_2, a gap must be matched to b_j, and in T_3, a_i must be matched to a gap. In other words, $T_k[i, j]$ contains the score of an optimal alignment to the problem $(a_{i'} a_{i'+1} \ldots a_i, b_{j'} b_{j'+1} \ldots b_j, s, k)$. We will later define more values attached to cells of each table, so the notation $[i, j]_k$ will refer to cell $[i, j]$ of table T_k, to differentiate from the value $T_k[i, j]$ itself. We will omit the subscript k when talking about cells in all three tables.

For initialization, all $[i' - 1, j' - 1]$ cells are set to $-\infty$, except if $start\_type \leq 1$, then $T_{|start\_type|}[i' - 1, j' - 1] = 0$.3 Also, if $start\_type > 0$:

$$T_2[i' - 1, j'] = \begin{cases} -(g+h), & \text{if } start\_type = 2 \\ -\infty, & \text{if } start\_type \in \{1, 3\}, \end{cases}$$

$$T_3[i', j' - 1] = \begin{cases} -(g+h), & \text{if} start\_type = 3 \\ -\infty, & \text{if } start\_type \in \{1, 2\}. \end{cases}$$

After these cells have been properly initialized, the remaining cells can be filled with the following equations (for more explanation, see [21]):

2. Given $x_1, x_2, \ldots, x_n$ and a binary associative operator $\otimes$, parallel prefix is the problem of computing $s_1, s_2, \ldots, s_n$, where $s_i = x_1 \otimes x_2 \otimes \ldots \otimes x_i$ (or equivalently, $s_i = s_{i-1} \otimes x_i$). This is a well-known primitive operation in parallel computing, and is readily available on most parallel computers. For example, the function MPI_Scan computes parallel prefix.

3. Theoretically, if $start\_type = 1$, then $T_1[i' - 1, j' - 1]$ should be $-\infty$ since in that case, this (empty) partial alignment does not satisfy the $start\_type$. However, setting it to 0 causes no computational problems and reduces the number of special cases for the neighboring cells.

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

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

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

Note that the value in each cell of each table depends only on the values of its left, upper-left, and upper neighbors. We define the *origin* of cell $[i,j]_k$ (denoted $origin([i,j]_k)$) to be the cell from which $T_k[i,j]$ was calculated. $[i'-1,j'-1]$ cells, and all cells whose value is $-\infty$, are without origin because their value is not calculated from any other cell. All nondiagonal neighbors of $[i'-1,j'-1]$ whose value is not $-\infty$ have $[i'-1,j'-1]_{|start\_type|}$ as their origin. If any cell has multiple candidates for origin, we can choose a unique origin either arbitrarily, or by giving preference in some order. Finally, when we say a cell *originates* from another cell, we will be referring to the reflexive transitive closure of origin defined above.

We can fill the tables by initializing the appropriate cells, and computing the remaining entries row by row. Because the table computation grows from $[i'-1,j'-1]$, we call this cell the *seed cell*. Note that cells in the leftmost column (respectively, the topmost row) can only originate from the cells above them (respectively, to their left), so their values are known regardless of A and B. For $i' \le i \le i''; j' \le j \le j''$,

$$T_1[i'-1,j] = T_1[i,j'-1] = -\infty,$$
$$T_3[i'-1,j] = T_2[i,j'-1] = -\infty,$$
$$T_2[i'-1,j] = \begin{cases} -g(j-j'+1), & \text{if } start\_type = -2 \\ -h-g(j-j'+1), & \text{if } start\_type \in \\ & \{2,-1,-3\} \\ -\infty & \text{otherwise,} \end{cases}$$
$$T_3[i,j'-1] = \begin{cases} -g(i-i'+1), & \text{if } start\_type = -3 \\ -h-g(i-i'+1), & \text{if } start\_type \in \\ & \{3,-1,-2\} \\ -\infty & \text{otherwise.} \end{cases}$$

Define $h'(k)$ to be h if $k = end\_type$ and $end\_type \in \{-2,-3\}$, and 0 otherwise. Once the tables are filled, if $end\_type > 0$, then $T_{end\_type}[i'',j'']$ holds the optimal score. If $end\_type < 0$, then the maximum of $T_1[i'',j'']$, $T_2[i'',j''] + h'(-2)$, and $T_3[i'',j''] + h'(-3)$ is the optimal score. The alignment itself can be extracted by using an origin traceback procedure starting from the entry said to contain the optimal score. The solution to the sequence alignment problem can therefore be viewed as a path in T from cell $[i'-1,j'-1]_{|start\_type|}$ to the appropriate $[i'',j'']$ cell, where each cell is connected to its origin.

To find the elements of a partial balanced partition, we make use of tables T_1, T_2, and T_3 to find cells where we can perform a recursive decomposition of the problem. The cells where subdivisions occur will correspond to elements of a partial balanced partition.

Recalling the definitions from Section 4, we can say that if a cell $[i,j]_k$ is on the path of the solution C through T, then either $j = C_a(i)$ or $i = C_b(j)$, implying $(i,j) \in C^*$. Therefore, by Proposition 4, if we know that the solution passes through a cell $[i,j]_k$, we can divide the original problem of finding an alignment between A and B into two parts: subproblem $(a_{i'}a_{i'+1}\ldots a_i, b_{j'}b_{j'+1}\ldots b_j, s, k)$ and subproblem $(a_{i+1}a_{i+2}\ldots a_{i''}, b_{j+1}b_{j+2}\ldots b_{j''}, -k, e)$. The first subproblem can be viewed as filling in the rectangular area of cells between $[i'-1,j'-1]$ and $[i,j]$ (inclusive), and the second as filling in the rectangular area of cells between $[i,j]$ and $[i'',j'']$ (inclusive) (although the T_1, T_2, and T_3 values will now represent scores related to the subproblems).

To find cells that lie on the solution, so that we can decompose the problem, we make use of the following idea, introduced in [10]. Let $rev(A) = a_{i''}a_{i''-1}\ldots a_{i'}$ and $rev(B) = b_{j''}\ b_{j''-1}\ldots b_{j'}$. Let $T_k^R[i,j]$ denote the score of an optimal alignment of $a_{i''}a_{i''-1}\ldots a_{i+1}$ and $b_{j''}b_{j''-1}\ldots b_{j+1}$, under the extended alignment problem with $start\_type = e$ and $end\_type = k$. The process of computing tables T_1^R, T_2^R, and T_3^R is similar to computing tables T_1, T_2, and T_3. The seed cell is now $[i'',j'']$, the optimal score is found in $[i'-1,j'-1]$, and the entire computation and its rules are reversed.

Consider any cell $[i,j]$ of the original tables T_1, T_2, and T_3, and let $A_1 = a_{i'}a_{i'+1}\ldots a_i$, $B_1 = b_{j'}b_{j'+1}\ldots b_j$, $A_2 = a_{i+1}a_{i+2}\ldots a_{i''}$, and $B_2 = b_{j+1}b_{j+2}\ldots b_{j''}$. We can construct an alignment of A and B by concatenating an alignment of A_1 and B_1 with an alignment of A_2 and B_2. $T_k[i,j]$, $k \in \{1,2,3\}$, gives us the best possible alignment of A_1 and B_1 whose last element is of type k. $T_k^R[i,j]$, $k \in \{1,2,3\}$, gives us the best possible alignment of A_2 and B_2 whose first element is of type k. Therefore, the best possible alignment of A and B that passes through $[i,j]$ has score

$$opt(i,j) = \max\begin{cases} T_{max}[i,j] + T_{max'}^R[i,j] \\ \\ T_2[i,j] + T_2^R[i,j] + h \\ \\ T_3[i,j] + T_3^R[i,j] + h, \end{cases}$$

where $T_{max}[i,j] = \max\{T_1[i,j], T_2[i,j], T_3[i,j]\}$, and $T_{max'}^R[i,j] = \max\{T_1^R[i,j], T_2^R[i,j], T_3^R[i,j]\}$. We can conclude that a solution passes through cell $[i,j]$ if $opt(i,j)$ is equal to the score of an optimal alignment. Whether this cell belongs to T_1, T_2, or T_3 is easily determined from the calculation of $opt(i,j)$.

Initially, we assume we are given p processors, with the sequences A' and B' distributed such that $\mathcal{P}_k$ is given $b_{k\frac{n}{p}+1}\ldots b_{(k+1)\frac{n}{p}}$ and $a_{k\frac{m}{p}+1}\ldots a_{(k+1)\frac{m}{p}}$. During the decomposition phase, $\mathcal{P}_k$ is considered *responsible* for columns $k\frac{n}{p}+1\ldots(k+1)\frac{n}{p}$. Define special columns of the dynamic programming table to be columns $0, \frac{n}{p}, 2\frac{n}{p}, \ldots, n$. Likewise, define special rows of the dynamic programming table to be rows $0, \frac{m}{p}, 2\frac{m}{p}, \ldots m$. To decompose the problem, we locate intersections an optimal solution makes with the special rows and special columns.

To determine the intersections that an optimal solution makes with a special row i, we compute $opt(i,j)$ for each cell in the row. An optimal alignment passes through $[i,j]$ iff

$$opt(i,j) = \max_{0 \le l \le n} (opt(i,l)),$$

since an optimal solution will have to pass through at least one cell of the row. We take the leftmost such cell as the intersection cell r_i for row i. The same technique works for subproblems, but there we only need to check $opt(i,j)$ for $j'-1 \leq j \leq j''$, since the subproblems we will be dealing with are chosen so that their optimal alignments are parts of optimal alignments to the entire problem.

As we find the intersection cell r_i by computing rows $i'-1 \dots i$ of T, and rows $i'' \dots i$ of T^R, we can also obtain intersection cells for the closest special column to the left of r_i, and the closest special column to the right of r_i. Consider adding the following two pointers to each cell $[i,j]_k$. One pointer is used only for cells that lie on special columns, and is a pointer to the uppermost cell on that special column such that $[i,j]_k$ originates from it. The second pointer points to the uppermost cell on the special column closest to the left (if one exists within the same subproblem), such that cell $[i,j]_k$ originates from it. For the leftmost column, we will set the value of the second pointer to be same as the first (otherwise, it would not be defined). We name the pointers $this([i,j]_k)$ and $prev([i,j]_k)$, and claim the following:

$$this([i,j]_k) = \begin{cases} this(origin([i,j]_k)), & \text{if } origin([i,j]_k) \text{ is the upper neighbor} \\ [i,j]_k, & \text{otherwise,} \end{cases}$$

$$prev([i,j]_k) = \begin{cases} this(origin([i,j]_k)), & \text{if } origin([i,j]_k) \text{ is on a special column to the left} \\ prev(origin([i,j]_k)), & \text{otherwise.} \end{cases}$$

For initialization, we have

$$prev([i'-1,j'-1]_{|start\_type|}) = [i'-1,j'-1]_{|start\_type|},$$
$$this([i'-1,j'-1]_{|start\_type|}) = [i'-1,j'-1]_{|start\_type|}.$$

The values of $prev$ and $this$ for the remaining cells can easily be computed as we fill in table T. Similarly, as we compute T^R, we can maintain two corresponding pointers, but the directions in the definitions become reversed (left becomes right, and up becomes down). In the case of T^R, we will call $next([i,j]_k)$ the pointer corresponding to $prev([i,j]_k)$. Consequently, we take $prev(r_i)$ as the intersection cell for the column closest to the left of r_i, and $next(r_i)$ as the intersection cell for the column closest to the right of r_i.

We can now present the details of the decomposition phase. At each step, we have a set of disjoint rectangular regions over T that are yet to be subdivided. Each region is considered as a separate subproblem, and has a group of processors allocated to it. This group consists exactly of the processors considered responsible for the columns the subproblem intersects (except for the leftmost column of the subproblem). The processor with the lowest ID within each group is defined as the *head* of the group. We maintain the invariant that in each step every row, as well as every column, intersect at most one active (still to be subdivided) subproblem. Furthermore, all rows (respectively, columns) that lie between two consecutive special rows (respectively, columns) that intersect an active subproblem must intersect the same subproblem. Hence, each processor is allocated to at most one subproblem within a step. Initially, we have all processors allocated to a single subproblem, ranging over the entire dynamic programming table, with $start\_type$ of -1 and end_type of -1.

Suppose we are currently decomposing some subproblem (A,B,s,e). A special row i is selected from the middle of the region as follows. If rows $k\frac{m}{p}, (k+1)\frac{m}{p}, \dots, k'\frac{m}{p}$ are the special rows going through the region, then set $i = \lceil\frac{k+k'}{2}\rceil\frac{m}{p}$. The first task is to find r_i, $prev[r_i]$, and $next[r_i]$. The relevant entries in tables T_1, T_2, and T_3, along with their associated pointers, are computed row by row. The T values can be computed using parallel prefix [1], while $this$ and $prev$ (or $next$) pointers are easily maintained along with the parallel prefix operation.

Each row is computed from the previous, so we only need $O(\frac{n}{p})$ memory at a time. The processors already have the part of sequence B' they require, while parts of sequence A' can be broadcast to all processors as they are needed. In general, the part of sequence A' that a group requires in order to subdivide its region may not lie entirely on processors inside the group. We perform the broadcasting of parts of sequence A' as follows: First, the head of each group receives a part of sequence A' from the processor that is responsible for storing it. These parts are always communicated in $\frac{m}{p}$ sized chunks to reduce the number of communications. Since we maintain that each set of rows between two special rows intersects at most one active subproblem, any processor will need to supply the part of A' it stores to at most one processor group (within a step). Because parts are required only one at a time within each group, we can communicate one required part to each group concurrently using a permutation communication. Then, the head of each group can broadcast the part to all members of the group.

Once we obtain row i of the T tables, we can similarly compute row i of the T^R tables, and compute $opt(i,j)$ for each i,j on that row. Using a **reduce** [19][4] operation with max as the operator, we find r_i, and **broadcast** [19] it along with $prev(r_i)$ and $next(r_i)$. Define the *index* of a cell to be the index of the table it belongs to. Set t_1 to be the index of $prev(r_i)$, set t_2 to be the index of r_i (in the T tables), and set t_3 to be the index of $next(r_i)$ (in the T^R tables). Recall that if a cell $[i,j]_k$ is on the path of an optimal alignment through the T tables, then $(i,j) \in C^*$. Also, the index k corresponds to $C_{left}(i,j)$. Similarly, if a cell $[i,j]_k$ is on the path of an optimal alignment through the T^R tables, then $(i,j) \in C^*$, and k corresponds to $C_{right}(i,j)$.

Using these observations and Proposition 4, the problem can be divided into three parts: the rectangular region of the table between $[i'-1,j'-1]$ and $prev(r_i)$ (with $start\_type$ s and end_type t_1), between $prev(r_i)$ and $next(r_i)$ (with $start\_type$ $-t_1$ and end_type $-t_3$), and between $next(r_i)$ and $[i'',j'']$ (with $start\_type$ t_3 and end_type e). If r_i lies on a special column, we can furthermore split the middle region into two parts around r_j, the upper-left subregion having end_type t_2 and the lower-right having $start\_type$ $-t_2$.

4. Given $x_1, x_2, \dots, x_n$ and a binary associative operator $\otimes$, reduce is the problem of computing $x_1 \otimes x_2 \otimes \dots \otimes x_n$. This is a well-known primitive operation in parallel computing, and is readily available on most parallel computers (for example, the function MPI_Reduce).

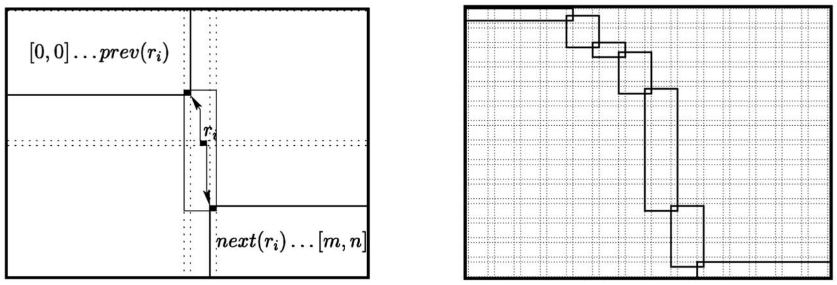


Fig. 6. First-level subdivision of the problem, and the final partial balanced partition.

The two outermost regions are to be recursively subdivided, each by the processors responsible for the columns that intersect the subproblem (with the exception of the leftmost column). Again, in each region, a special row that goes through the middle of the region is selected, and the region is then subdivided by the process described above.

There are two terminating condition for the recursion. The first is that only one processor is assigned to the group, and the other is that either there are no special rows going through the group's region, or the only special row in the region is the topmost row of the region. A depiction of the initial subdivision of the problem and the final resulting partition is shown in Fig. 6. For an outline of the decomposition algorithm, see Fig. 7. Once the decomposition is over, we will be left with regions that are either no more than $\frac{n}{p}+1$ wide, or no more than $\frac{m}{p}$ tall, corresponding to the two termination conditions.

Proposition 6. *There are $O(\log p)$ recursion levels.*

Proof. Let r_k denote the maximum number of special rows going through any active region at the kth level of recursion. We have that $r_0 = p+1$, and due to the choice of the special row for the next subdivision, $r_{k+1} = \lceil \frac{r_k}{2} \rceil$. Now, let $k' = min\{k | r_k = 1\}$. Clearly, $k' = O(\log p)$. The way we choose a special row i for the next subdivision guarantees that the row of $prev[r_i]$ is strictly above the row of $next[r_i]$. This is because we choose the leftmost candidate cell as the intersection cell, so its origin has to lie above. Hence, any region that still needs to be subdivided when $r_k = 1$ will result in one region that contains no special rows, and one region that either contains no special rows or has a special row as its topmost row. In either case, the recursion will stop at level k'. □

Proposition 7. *The recursive decomposition can be performed $O\left(\frac{mn}{p}\right)$ time and $O\left(\frac{m+n}{p}\right)$ space, as long as $p = O\left(\frac{n}{\log n}\right)$.*

Proof. For simplicity, assume p is a power of 2. Let q be the number of recursion levels in the decomposition. At recursion level r $(0 \le r < q)$, we have at most 2^r regions to be subdivided, each of height at most $\frac{m}{2^r}+1$. Hence, the combined height of all regions for a particular processor during the subdivision phase is at most $\left(\sum \frac{m}{2^r}\right) + q = O(m)$. Furthermore, because the regions always split around special rows, the number of processors storing the sections of sequence A' required for any specific region is at most $\frac{p}{2^r}$. Thus, the total number of broadcasts any processor is involved in is at most $\sum \frac{p}{2^r} = O(p)$. A row of table T can be computed in $O\left(\frac{n}{p}\right)$ time, and broadcasting of a portion of sequence A' takes $O\left(\frac{m}{p}\log p\right)$ time.[5] The total time for row computations is therefore $O\left(\frac{n}{p}\right) \times O(m) = O\left(\frac{mn}{p}\right)$, and the total time for all broadcasts is

$$O\left(\frac{m}{p}\log p\right) \times O(p) = O(m\log p) = O\left(\frac{mn}{p}\right).$$

A processor is never required to use more than $O\left(\frac{m+n}{p}\right)$ space. □

Proposition 8. *The cells of the decomposition, along with their $start\_type$ (and the element $(m,n,1)$), make up a partial balanced partition of A' and B' of size $O(p)$.*

Proof. The decomposition phase terminates with at most p rectangular regions that have overlapping corner cells. Number those regions in order (from top left to bottom right) with 0, 1, etc. For region k, construct the 3-tuple (i_k, j_k, t_k), where i_k and j_k comprise the coordinates of the upper left corner of the cell, and t_k is the $start\_type$ for the region's subproblem. Let $P = ((i_0, j_0, t_0), (i_1, j_1, t_1), \ldots, (m, n, 1))$. We can now show that P satisfies all properties of a partial balanced partition. Note that the properties are satisfied relative to any alignment C that is a concatenation of optimal alignments to the subproblems corresponding to each region. Property 1 is satisfied by the ordering of the regions. Property 2 is satisfied since t_0 has to be -1. Property 3 follows from the fact that the regions are either no more than $\frac{n}{p}+1$ wide, or no more than $\frac{m}{p}$ tall, implying that either $i_{k+1} - i_k \le \frac{m}{p} - 1$, or $j_{k+1} - j_k \le \frac{n}{p}$. Property 4 is satisfied by our choice of points and the $start\_type$ for subdivision. □

Since the region information is distributed among the processors, the corresponding partial balanced partition is

5. 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, in time proportional to the size of the largest message. The model closely reflects the behavior of Clos networks (for example, Myrinet) and most multistage interconnection networks, the BSP parallel computing model, and the programming abstraction supported by MPI. In this model, the cost for broadcast, reduce, and parallel prefix operations involving messages of length l is $O(l \log p)$. The algorithms presented are applicable to other models of computation as well, and often equally efficient. 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 runtime would hold for meshes as well [11].

```
Algorithm 1 DecomposeIteration
  $row_{mid} \leftarrow$ median special row between $row_{up}$ and $row_{down}$.
  Each processor initializes its $T$ values for $row_{up}$.
  For rows $row_{up} + 1$ through $row_{mid}$
      Processor $\mathcal{P}_i$ gets sequence $A'$ data in $\frac{m}{p}$ chunks as required and broadcasts it.
      Calculate $T$ and $prev$ values for current row using parallel prefix[1].
  Each processor initializes its $T$ values for $row_{down}$.
  For rows $row_{down} - 1$ through $row_{mid}$
      Processor $\mathcal{P}_i$ gets sequence $A'$ data in $\frac{m}{p}$ chunks as required and broadcasts it.
      Calculate $T^R$ and $next$ values for current row using parallel prefix[1].
  Compute $opt$ values for $row_{mid}$ by adding $T$ and $T^R$ values appropriately.
  $r_{mid} \leftarrow$ position of leftmost maximal value.
  $\mathcal{P}_{mid} \leftarrow$ processor responsible for $r_{mid}$.

  $\mathcal{P}_i \ldots \mathcal{P}_{mid-1}$ do next DecomposeIteration on row boundaries $row_{up}$, $prev[r_{mid}]$.
  $\mathcal{P}_{mid+1} \ldots \mathcal{P}_j$ do next DecomposeIteration on row boundaries $next[r_{mid}]$, $row_{down}$.
```

Fig. 7. An iteration of the decomposition phase on processors $\mathcal{P}_i, \mathcal{P}_{i+1}, \ldots, \mathcal{P}_j$, decomposing the region between columns $i\frac{n}{p}$ and $(j+1)\frac{n}{p}$, and rows row_{up} and row_{down}.

also distributed among processors. We can therefore reassign the regions in parallel, using $O(1)$ space and $O(\log p)$ time. Once the reassigning is completed, we proceed as outlined in Section 4.

Proposition 9. *The sequence alignment problem can be solved in $O(\frac{m+n}{p})$ space and $O(\frac{mn}{p})$ time.*

Proof. This follows directly from Propositions 7, 8, and 5.□

6 Experimental Results

We implemented the algorithm presented using C++ and MPI, and tested it on an IBM xSeries cluster. The runtime was measured for three different phases of the algorithm. In the first phase, a partial balanced partition for the sequences is computed. The redistribution of the sequences according to the subproblems resulting from the partition is performed in the second phase. In the third phase, the subproblems are solved by the processors assigned to them. Since the redistribution strategy we use assigns each processor to at most two consecutive subproblems, the third phase consists of one step in which all even subproblems are solved concurrently, and another step in which all odd subproblems are solved concurrently.

We tested the program using two types of data: 1) identical sequences, resulting in a unique optimal alignment along the diagonal from top left to the bottom right of the table and 2) sequences that use distinct characters, with a scoring function and gap penalty function such that the optimal alignment corresponds to each sequence completely aligned with gaps. The first test case results in $\Theta(\log p)$ levels of recursion in the first phase of the algorithm, but results in subproblems that are each solved by a single processor. On the other hand, the second test case is partitioned in only one subdivision, but results in two subproblems that are each assigned about half the processors. They represent the extreme cases for each of the phases of the algorithm, hence we use them to illustrate the functioning of the algorithm. We refer to the two cases as *complete match* and *complete mismatch*, respectively. (See Fig. 8.)

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 runtime and the times spent in the different phases of the algorithm for sequences $80K$ long are summarized in Table 1. The runtime spent in Phase 2 is negligible and, hence, is not shown in the table. Note that for $p = 1$, our algorithm reduces to what is basically the sequential $O(m + n)$ space, $O(mn)$ time algorithm by Mayers and Miller [17]. It can be seen that the difference in the runtimes for the first phase between the complete match and mismatch cases for the same number of processors are large when $p \geq 8$, stemming from additional decomposition phases in the complete match case. For $p \leq 4$, both input cases have only one step of decomposition. In the third phase, the runtimes are much larger for the complete mismatch case due to interprocessor communication required to solve the final subproblems. Also, note that Phase 3 runtimes reduce approximately by a factor of 4 as the number of processors doubles, since this phase takes only $O(\frac{mn}{p^2})$ time.

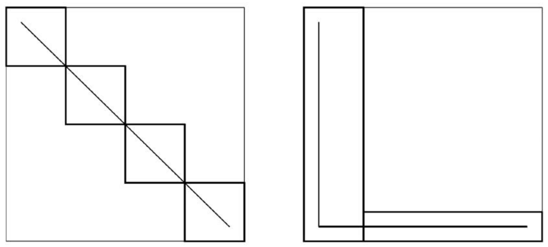


Fig. 8. Problem decomposition in complete match and complete mismatch cases.

TABLE 1
Running Time in Seconds on the xSeries Cluster for $m = n =$ 80K

	Complete match case			Complete mismatch case		
p	Ph. 1	Ph. 3	Total	Ph. 1	Ph. 3	Total
1	0	1316.3	1316.3	0	1334.6	1334.6
2	806.7	628.7	1435.4	743.3	687.8	1431.1
4	378.8	140.9	519.7	361.4	258.0	619.4
8	248.7	34.6	283.3	174.9	58.4	233.3
16	147.7	8.6	156.3	88.2	14.3	102.5
32	96.3	2.1	98.4	44.8	3.6	48.4
60	62.6	0.6	63.2	31.3	1.0	32.3

Phase 2 took less than 0.01 seconds for all of these runs.

However, the $p = 2$ cases are only about two times faster than the $p = 1$ cases, because the maximum processor workload reduces by only a factor of 2: In the complete match case, our subproblem reassignment technique assigns both of the final subproblems to the same processor, and in the complete mismatch case the largest subproblem is in fact only half the size of the original problem.

The speedups obtained as a function of the number of processors for various problem sizes are shown in Fig. 9. The serial runtime used in computing the speedups is the runtime of the best sequential algorithm [17], which is what our implementation conveniently reduces to for the special case of $p = 1$. The runtime spent in Phase 2 is negligible for all the problem sizes and number of processors used in our experiments. As the runtime of Phase 3 decreases quadratically with the number of processors while the runtime of Phase 1 decreases linearly, Phase 1 determines the runtime for larger number of processors (also evident from Table 1). This causes a superlinear benefit with the increase in number of processors beyond two. For $p = 2$, the time spent in solving the subproblems roughly halves from the serial version. But, computing the decomposition, not required for the serial algorithm, takes an equal amount of time, giving about the same runtime for both $p = 1$ and $p = 2$. This, combined with the quadratic reduction in Phase 3 for subsequent increase in number of processors, gives an interesting characteristic for the algorithm: As the number of processors increases, the speedup approaches ideal speedup, giving it a higher degree of scalability. This can be readily observed in Fig. 9. The inferior scaling when $p = 60$ and $p = 32$ in the smaller problem sizes shows that such problem sizes are too small for that many processors. This information can be used to compute the largest number of processors that can be beneficially used for a given problem size.

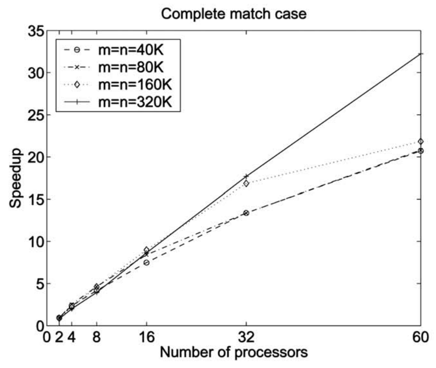
TABLE 2
Running Time in Seconds on 60 Processors for $m = n =$ 1.1M

	Complete match case			Complete mismatch case		
p	Phase 1	Phase 3	Total	Phase 1	Phase 3	Total
60	9177.4	125.2	9302.6	4622.5	239.5	4862.0

Finally, we report the runtimes from testing our algorithm on a large problem size of $m = n = 1.1$M, using 60 processors (see Table 2). The complete match case took $9,302.6$ seconds (approximately 2.58 hours), $9,177.4$ of which were taken by the first phase of the algorithm. The complete mismatch case took only $4,862.0$ seconds (approximately 1.35 hours), $4,622.5$ of which were spent in the first phase.

7 Other Applications

In this section, we discuss other applications where the techniques developed in this paper can be used. We first consider several variants of the parallel sequence alignment problem. The problem of finding an optimal alignment between a short sequence and any subsequence of a longer sequence is known as *semiglobal alignment*. Algorithmically, the solution differs from global alignment in that a different initialization of the top row (or column) of the dynamic programming table is required and the optimal solution appears at the maximum value recorded in the bottom row (or column) [21]. These difference can be easily accommodated in our algorithm. A more important sequence alignment problem is *local alignment*, in which the highest scoring alignment between any subsequence of one sequence with any subsequence of another sequence is desired. Huang's [13] serial algorithm for this problem readily allows the application of our techniques. Perhaps the most important practical application of our techniques will be in the solution of the *syntenic alignment* problem. In



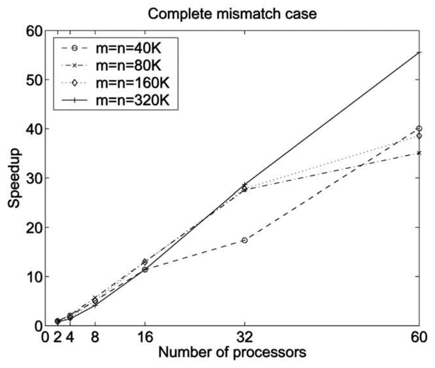


Fig. 9. Fixed-size speedups for various problem sizes.

this problem, an ordered list (of same size) of nonoverlapping subsequences is sought for each sequence such that each corresponding subsequence pair exhibits high degree of similarity. Syntenic alignments are useful in comparing syntenic regions of genomic DNA from related species (such as human and mouse) and identifying conserved exons (substrings of genes) [7]. This is a practical application involving very long sequences, making it an ideal target for applying our results. It is a straightforward exercise to extend the techniques presented in this paper to the parallel solution of the syntenic alignment problem. Our results also have potential applications to other problems involving the use of parallel dynamic programming, not necessarily from the field of computational biology.

8 Conclusions

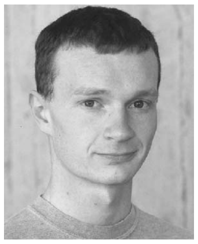
We presented the first space and time optimal parallel algorithm for computing an optimal pairwise alignment of two sequences. Our experimental results demonstrate that the algorithm is practically efficient and scalable. A number of other sequence alignment problems can be solved using the full-sequence pairwise alignment problem discussed in this paper. Such problems include semiglobal, local, and syntenic alignments [7], [13], [21]. Consequently, our result on space and time optimality extends to these problems as well. The techniques we use may also have potential applications to other problems, not necessarily from computational biology, whose solution involves parallel dynamic programming. Based on the experimental results, a pair of sequences of length one million can be aligned in a matter of few hours. We are currently studying use of this algorithm in comparative genomics, where alignments of such long sequences are required.

Acknowledgments

This research was supported by the US National Science Foundation ACI-0203782 and NSF EIA-0130861.

References

[1] S. Aluru, N. Futamura, and K. Mehrotra, "Parallel Biological Sequence Comparison Using Prefix Computations," *J. Parallel and Distributed Computing,* vol. 63, no. 3, pp. 264-272, 2003.
[2] C.E.R. Alves, E.N. Caceres, and F. Dehne, "Parallel Dynamic Programming for Solving the String Editing Problem on a CGM/BSP," *Proc. ACM Symp. Parallel Algorithms and Architectures,* pp. 275-281, 2002.
[3] A. Apostolico, M.J. Atallah, L.L. Larmore, and S. MacFaddin, "Efficient Parallel Algorithms for String Editing and Related Problems," *SIAM J. Computing,* vol. 19, no. 5, pp. 968-988, 1990.
[4] M.O. Dayhoff, R. Schwartz, and B.C. Orcutt, "A Model of Evolutionary Change in Proteins: Matrices for Detecting Distant Relationships," *Atlas of Protein Sequence and Structure,* vol. 5, M.O. Dayhoff, ed., Nat'l Biomedical Research Foundation, DC, pp. 345-358, 1978.
[5] E.W. Edmiston and R.A. Wagner, "Parallelization of the Dynamic Programming Algorithm for Comparison of Sequences," *Proc. Int'l Conf. Parallel Processing,* pp. 78-80, 1987.
[6] E.W. Edmiston, N.G. Core, J.H. Saltz, and R.M. Smith, "Parallel Processing of Biological Sequence Comparison Algorithms," *Int'l J. Parallel Programming,* vol. 17, no. 3, pp. 259-275, 1988.
[7] N. Futamura, S. Aluru, and X. Huang, "Parallel Syntenic Alignments," *Proc. Int'l Conf. High Performance Computing,* pp. 420-430, 2002.
[8] O. Gotoh, "An Improved Algorithm for Matching Biological Sequences," *J. Molecular Biology,* vol. 162, pp. 705-708, 1982.
[9] S. Henikoff and J.G. Henikoff, "Amino Acid Substitution Matrices from Protein Blocks," *Proc. Nat'l Academy of Sciences,* vol. 89, pp. 10915-10919, 1992.
[10] D.S. Hirschberg, "A Linear Space Algorithm for Computing Maximal Common Subsequences," *Comm. ACM,* vol. 18, no. 6, pp. 341-343, 1975.
[11] A. Grama, V. Kumar, and A. Gupta, *Introduction to Parallel Computing,* second ed., Reading, Mass.: Addison-Wesley Publishing, 2003.
[12] X. Huang, "A Space-Efficient Parallel Sequence Comparison Algorithm for a Message-Passing Multiprocessor," *Int'l J. Parallel Programming,* vol. 18, no. 3, pp. 223-239, 1989.
[13] X. Huang, "A Space-Efficient Algorithm for Local Similarities," *Computer Applications in the Biosciences,* vol. 6, no. 4, pp. 373-381, 1990.
[14] X. Huang and K. Chao, "A Generalized Global Alignment Algorithm," *Bioinformatics,* vol. 19, no. 2, pp. 228-233, 2003.
[15] E. Lander, J.P. Mesirov, and W. Taylor, "Protein Sequence Comparison on a Data Parallel Computer," *Proc. Int'l Conf. Parallel Processing,* pp. 257-263, 1980.
[16] W.J. Masek and M.S. Paterson, "A Faster Algorithm for Computing String Edit Distances," *J. Computer and System Sciences,* vol. 20, pp. 18-31, 1980.
[17] E.W. Mayers and W. Miller, "Optimal Alignments in Linear Space," *Computer Applications in the Biosciences,* vol. 4, no. 1, pp. 11-17, 1988.
[18] 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. Molecular Biology,* vol. 48, pp. 443-453, 1970.
[19] P. Pacheco, *Parallel Programming with MPI.* San Francisco, Calif.: Morgan Kaufmann Publishers, 1996.
[20] S. Ranka and S. Sahni, "String Editing on an SIMD Hypercube Multicomputer," *J. Parallel and Distributed Computing,* vol. 9, pp. 411-418, 1990.
[21] J. Setubal and J. Meidanis, *Introduction to Computational Molecular Biology.* Boston, Mass.: PWS Publishing Company, 1997.
[22] T.F. Smith and M.S. Waterman, "Identification of Common Molecular Subsequences," *J. Molecular Biology,* vol. 147, pp. 195-197, 1981.



Stjepan Rajko received the BS degree in computer science and mathematics in 2002, and the MS degree in computer science in 2004, both from Iowa State University. He is currently a PhD student in computer science with a concentration in arts, media, and engineering at Arizona State University. His research interests include motion analysis, bioinformatics, parallel computing, computational geometry, and robotics.

Srinivas Aluru received the BTech degree in computer science from the Indian Institute of Technology, Chennai, India, in 1989, and the MS and PhD degrees in computer science from Iowa State University in 1991 and 1994, respectively. He is an associate professor and associate chair for graduate education in the Department of Electrical and Computer Engineering at Iowa State University. He is affiliated with the Laurence H. Baker Center for Bioinformatics and Biological Statistics, the Center for Plant Genomics, and serves as associate chair for the Bioinformatics and Computational Biology Graduate Program. Earlier, he held faculty positions at New Mexico State University and Syracuse University. His research interests include parallel algorithms and applications, bioinformatics and computational biology, and combinatorial scientific computing. He cochairs the Annual Workshop in High Performance Computational Biology (http://www.hicomb.org) and has served as a guest editor of the *Journal of Parallel and Distributed Computing* for a special issue on this topic. Dr. Aluru served on the program committees of several conference and workshops on parallel processing and computational biology. He is a recipient of a US National Science Foundation CAREER award, an IBM faculty award, and the Young Engineering Faculty Research Award from Iowa State University. He has coauthored a book and more than 50 articles in peer-reviewed journals and conferences. He is a member of the ACM, and a senior member of the IEEE and the IEEE Computer Society.