CABIOS

Vol. 13 no. 1 1997
Pages 45–53

Reduced space sequence alignment

J.Alicia Grice, Richard Hughey[1] and Don Speck

Abstract

Motivation: *Sequence alignment is the problem of finding the optimal character-by-character correspondence between two sequences. It can be readily solved in $O(n^2)$ time and $O(n^2)$ space on a serial machine, or in $O(n)$ time with $O(n)$ space per $O(n)$ processing elements on a parallel machine. Hirschberg's divide-and-conquer approach for finding the single best path reduces space use by a factor of n while inducing only a small constant slowdown to the serial version.*

Results: *This paper presents a family of methods for computing sequence alignments with reduced memory that are well suited to serial or parallel implementation. Unlike the divide-and-conquer approach, they can be used in the forward–backward (Baum–Welch) training of linear hidden Markov models, and they avoid data-dependent repartitioning, making them easier to parallelize. The algorithms feature, for an arbitrary integer L, a factor proportional to L slowdown in exchange for reducing space requirement from $O(n^2)$ to $O(n\sqrt[L]{n})$. A single best path member of this algorithm family matches the quadratic time and linear space of the divide-and-conquer algorithm. Experimentally, the $O(n^{1.5})$-space member of the family is 15–40% faster than the $O(n)$-space divide-and-conquer algorithm.*

Availability: *The methods will soon be incorporated in the SAM hidden Markov modeling package http://www.cse.ucsc.edu/research/compbio/sam.html.*

Contact: *wzrph@cse.ucsc.edu*

Introduction

Sequence comparison and alignment are common and compute-intensive tasks which benefit from space-efficient execution. Sequence comparison rates the difference or similarity between two sequences. For the most related sequences, one then wants to see an alignment, showing where the sequences are similar, by graphically lining up matching elements, and how they differ, shown by gaps and mismatches.

Computer Engineering, University of California, Santa Cruz, CA 95064, USA
[1]To whom correspondence should be addressed

Good alignments and sequence comparisons come from solutions of an appropriately chosen optimization problem. The problem formulation defines a set of edit primitives, including match, insert and delete, and assigns them values or costs in a distance function. Optimization then maximizes the total match value (Needleman and Wunsch, 1970) or minimizes the total cost of mismatches and insert and delete gaps.

Dynamic programming organizes sequence comparison by comparing shorter subsequences first, so their costs can be made available in a table (Figure 1) for the next longer subsequence comparisons. The final entry becomes the comparison rating. Exact sequence comparison is an $O(n^2)$ serial time dynamic programming algorithm, where n is the length of the longest sequence. Masek and Paterson (1983) describe an $O(n^2/\log n)$ algorithm for strings of equal length from a finite alphabet with restrictions on the cost function, but it has a large constant factor and is not amenable to parallelization. Distance calculation is governed by a simple recurrence. The cost of transforming a reference string b into another string a is the solution of a recurrence whose core is:

$$c_{i,j} = \min \begin{cases} c_{i-1,j-1} + \text{dist}(a_i, b_j) & \text{match} \\ c_{i-1,j} + \text{dist}(a_i, \phi) & \text{insert} \\ c_{i,j-1} + \text{dist}(\phi, b_j) & \text{delete} \end{cases}$$

where $\text{dist}(a_i, b_j)$ is the cost of matching a_i to b_i, $\text{dist}(a_i, \phi)$ is the gap cost of matching a_i with no character in b, and $\text{dist}(\phi, b_j)$ is the cost of not matching b_j to anything in a. Edit distance, the number of insertions or deletions required to change one sequence to another, can be calculated by setting $\text{dist}(a_i, \phi) = \text{dist}(\phi, b_j) = 1$, and $\text{dist}(a_i, b_j) = 0$ if $a_i = b_j$ or 2 otherwise.

Sequence comparison using affine gap penalties involves three interconnected recurrences of a similar form (Gotoh, 1982). The extra cost for starting a sequence of insertions or deletions will, for example, make the second alignment of Figure 1 preferred over the alignment with four gaps. In the most general form of sequence comparison, profiles or linear hidden Markov model (Gribskov *et al.*, 1990; Krogh *et al.*, 1994), all transition or gap costs (g) between the three states (match, insert or delete) and character distance cost are position

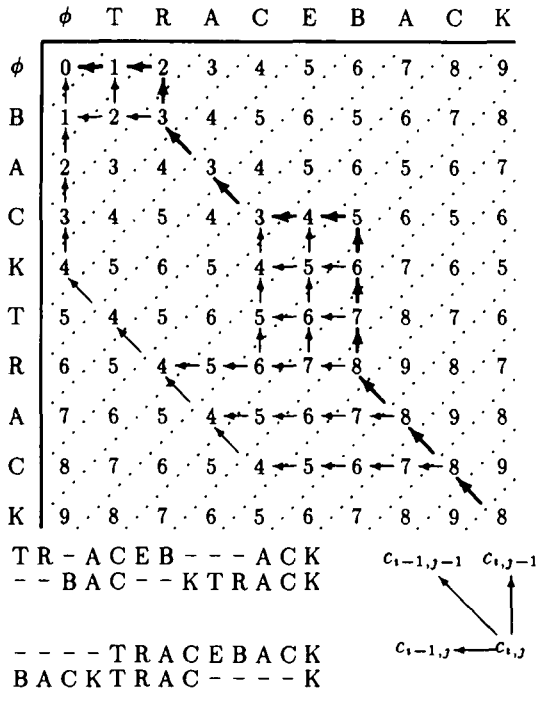

Fig. 1. Dynamic programming example to find least cost edit of 'BACKTRACK' into 'TRACEBACK' using Sellers' evolutionary distance metric (Sellers, 1974). Below the dynamic programming table are two possible alignments and an illustration of the data dependencies. Diagonals can be vectorized or computed in parallel.

dependent:

$$c^M_{i,j} = \min(c^M_{i-1,j-1} + g^{M\to M}, c^I_{i-1,j-1} + g^{I\to M}, c^D_{i-1,j-1} + g^{D\to M}) + \mathrm{dist}(a_i, b_j)$$

$$c^I_{i,j} = \min(c^M_{i,j-1} + g^{M\to I}, c^I_{-1} + g^{I\to I}, c^D_{i-1,j} + g^{D\to I}) + \mathrm{dist}(a_i, \phi)$$

$$c^D_{i,j} = \min(c^M_{i,j-1} + g^{M\to D}, c^I_{i-1,j} + g^{I\to D}, c^D_{i,j-1} + g^{D\to D}) + \mathrm{dist}(\phi, b_j)$$

Local alignment (Smith and Waterman, 1981), which finds the most similar subsequences of two sequences, adds a fourth, constant term to each minimization (such as zero) representing the cost of starting the correspondence at an internal (i,j) pair, in which case highly similar regions must, for the equations above, have negative cost. To allow the correspondence to end anywhere within the dynamic programming matrix, the matrix is searched for the best final score. By negating signs, dynamic programming can be presented using similarity and maximization (more common in biology) rather than cost and minimization (more common in computer science).

An alignment based on a given cost function is the reconstruction of an optimal path through the dynamic programming matrix. The standard means of reconstructing the path is to record the decisions made in each minimization during the forward pass, and then use these choices to find an optimal path from the lower right ($c^D_{n+1,n+1}$) to upper left ($c^D_{0,0}$) during the backward or traceback phase. Storing these choices will require at most six bits per (i,j) cell. However, having to store one byte from each comparison requires $O(n^2)$ space, leading to the need for more space-efficient alternatives.

In searching for a reduced-space alignment algorithm, we have two primary implementation targets: hidden Markov modeling and a new parallel processor currently under development. The most computationally intensive step of the former is a sequence of training iterations that uses an alignment-like calculation on the dynamic programming matrix that requires a summation of all possible paths through the dynamic programming matrix, adding and multiplying probabilities rather than minimizing and adding costs (Krogh *et al.*, 1994). When training on sequences and models over $n = 2000$ elements long, the $O(n^2)$ space requirement causes virtual memory page thrashing on a serial workstation, while the MasPar parallel computer code is unable to run because of each processing element's limited memory (64 kilobytes). The algorithms presented in this paper will enable the new parallel processor, called Kestrel, to perform sequence alignment and hidden Markov model (HMM) training despite having only a tiny 256 bytes of local memory per processing element (Hirschberg *et al.*, 1996).

Related work

Hirschberg (1975) discovered the first linear-space algorithm for sequence alignment, which Myers and Miller (1988) then popularized and extended. In reducing space from $O(n^2)$ to $O(n)$, these algorithms introduce a small constant (1.8 for Myers and Miller) slowdown to the $O(n^2)$ time algorithm. The core of these algorithms is a divide-and-conquer strategy in which an optimal midpoint of an $n \times n$ alignment is computed by considering column $n/2$ as computed by both the forward cost function on the sequences and the inverse cost function on the transposed sequences. At each point along this column, the sum of the forward and reverse costs will be the minimum cost of all alignment passing through that point. Minimizing over all members of the column will produce a point through which the optimal alignment will pass. This enables the division of the problem into two subproblems of combined size $n^2/2$, which can then be solved recursively.

Edmiston *et al.* (1988) proposed an extension to Hirschberg's algorithm for use on parallel processors by dividing the problem into H segments rather than Hirschberg's original two segments. Huang (1989) further improved the parallelization by noting that if one partitions along a pair of diagonals rather than a column, the problem will be reduced to equally

sized subproblems, a critical issue in creating a load-balanced parallel algorithm. That is, if the diagonal $i+j=n$ is considered, the minimizing point in that diagonal, (i', j'), will divide the problem into an upper $i' \times j'$ segment and a lower $(n-i') \times (n-j') = j' \times i'$ segment.

Huang's parallelization of Hirschberg's algorithm, and the related parallelization of Edmiston's algorithm, are best suited to MIMD (multiple instruction stream, multiple data stream) parallel processing with shared memory or unit-time message passing systems. Although the workload is evenly partitioned in these parallelizations, after an (i', j') determination, the sequences must be repartitioned into half-sized subsets with the first i' and j' characters of each sequence going to the first processor of one-half of the processing elements (PEs) over several time steps, and the remainder going to the first processor of the other half. A simple ring network would enable this for the first partitioning, but the recursive partitionings will require a multitude of sub-rings or, in fact, a fully connected graph.

If there is only a small amount of memory in each of P processing elements, complete copies of the sequences cannot be stored in each PE, meaning that the data must be moved through the parallel processor, making repartitioning particularly costly.

Ibarra *et al.* (1992) solved many of these problems for performing sequence alignment on a one-way linear array of finite state machines, a restrictive, and hence realistic, machine model in which data can only flow from left to right. Their parallelization of Hirschberg's algorithm balances the computation by performing problem division on the larger of the two dimensions of the dynamic programming sub-matrix during any given recursive call. Also, the sequence repartitioning time overhead is eliminated by skewing the computation, moving data forward to use up new processing elements for each recursive call. Thus, their algorithm requires $O(n+m)$ processing cells, rather than $O(m)$ cells, where n and m are the sequence lengths, but requires just $O(n+m)$ time even when only left-to-right communication is available. This method is an excellent solution when sufficient processing elements are available.

In the serial case, our goal is to find a simple reduced-space algorithm that can be used with forward–backward HMM training. In the parallel case, our goal is to find a simple reduced-space algorithm that does not have the data-dependent computation (which can result in repartitioning delays on a parallel processor) of the divide-and-conquer algorithm or require the additional PEs of Ibarrra's method, and which can be efficiently implemented on linearly connected processor arrays with limited amounts of memory, nearest-neighbor unit-time communication, and broadcast instructions. These two goals can be solved with a single family of algorithms.

System and methods

The serial experimental work of this paper was performed on a Sun Microsystems UltraSparc Station Model 140 with 64 megabytes of memory. The programs were written in ANSI C and were compiled using the Gnu C compiler version 2.7.2 and optimization level 3.

The parallel results used a MasPar MP-2204 parallel computer with a DEC Alpha 3000/300X front end. Each of the 4096 processing elements has 64 kilobytes of memory. The parallel programs were written in MPL, the MasPar's parallel C language. The MPL code was compiled using MasPar's MPL compiler version 3.3.21 at its highest optimization level.

Algorithm

In the following discussion, let n be the length of the longest of the two sequences compared and m be the length of the shorter. In the linear HMM case, m will be the length of the model; studying a short motif, m may be below 100, while sequence length n may range into the thousands or tens of thousands. Let mM be the total amount of memory available, i.e. memory use is measured in the number of rows, each of length m, that can fit in memory.

Basic algorithm

The basic alignment algorithm includes two parts (Figure 2). In the first part, the $c_{i,j}$ values for each state and each (i, j) pair are calculated according to the recurrence equations described above. During this calculation, the decision bit of each minimization is saved in memory. In the second part (the traceback phase), a chain of states from $c^D_{n+1,n+1}$ to $c^D_{0,0}$ is constructed by following the path of the minimizations. The total time required by the algorithm is $2mn$, assuming that each forward and backward cell calculation requires unit time. If only the best path must be traced back (as is done in Figure 2 for simplicity), at most $mn + m + n = O(mn)$ time is

```
for i ← 1 to n
    ComputeAndSave (row[i], Trace[i]);
/* Traceback */
i← n + 1
j← n + 1
state ← delete
while (i > 0 || j > 0)
    print state (i, j) = state
    nextstate ← minstate (i, j,state)
    case state:
        delete: j← j − 1
        insert: i← i − 1
        match: i← i − 1; j← j − 1
    endcase
    state ← nextstate;
endwhile
```

Fig. 2. Basic algorithm. The subroutine call performs the dynamic programming recurrence, saving the choice made in each minimization.

required. The memory used to store the minimization choices is called traceback memory, and must include n rows of $O(m)$ elements.

Two-level algorithm

The simple alignment algorithm's limitation to $M = n$, or $O(nm)$ memory, can be overcome with checkpoints and recalculation. To recalculate efficiently the comparisons of a row in the dynamic programming matrix, the state of computation shortly before that time is needed. By appropriately selecting when to save these checkpoints of state information, which include all cost totals required to calculate future rows, alignments can be performed efficiently in limited space.

For the remainder of this discussion, we assume that M is measured by the number of checkpoints that memory can hold, and that there is a one-to-one correspondence between checkpoint and traceback storage. The space efficiency of the algorithm could be improved by noting that storing choices requires less memory than storing checkpoints (six bits as opposed to three words per column) for alignment operations other than forward–backward HMM training.

Fixed memory partition. Suppose that the available memory is divided into a space for alignment traceback calculation, M_{trace}, and a space to store checkpoints, M_{check}. Because traceback cannot commence until the final $c_{n,n}$ value of the matrix is computed, only the final block of traceback information for M_{trace} rows needs to be saved. The first M_{trace} comparison outcomes can be discarded because they can be recalculated from the initial conditions. The state of the last computation of this segment must be saved as a checkpoint.

After state has been saved, the next M_{trace} rows can be computed and another checkpoint saved. These comparison outcomes can also be discarded because they can be recomputed from the first saved checkpoint. This process is repeated until the sequences have been compared. To find an optimal alignment, a traceback is performed on the last M_{trace} rows. The previous M_{trace} comparison outcomes are recomputed from checkpoint information, and a traceback is performed on those rows. We call this a 2-level method because of the hierarchy in calculating values: checkpoints combined with simple (level-1) forward and traceback calculations (Figure 3).

The performance of the 2-level fixed partition algorithm is simple to analyze. The greatest number of rows that can be calculated with $M_{\text{check}} + M_{\text{trace}}$ memory locations is $M_{\text{trace}}(M_{\text{check}} + 1)$, which for a given amount of memory M is optimized by $M_{\text{check}} = M_{\text{trace}} = M/2$. Converting rows to n, the amount of space required for sequences of length n and m is $O(m\sqrt{n})$, while the time required for the calculation is

```
M ← size of memory; i ← 0; cycle ← 0;
SaveState (row[0], Check[cycle]);
for cycle ← 1 to ⌈n/M_trace⌉ − 1;
    for k← i to i + M_trace − 1;
        ComputeCosts (row[k]);
    SaveState (row[i + M_trace − 1], Check[cycle]);
    i← i + M_trace;

for k← i to n;
    ComputeAndSave (row[k], Trace[k mod M_trace + cycle]);
Traceback (Trace[i mod M_trace + cycle] ...
          Trace[nmod M_trace + cycle]);

endcycle ← cycle;
for cycle ← endcycle to 0
    i← i − M_trace ;
    RetrieveCheckpoint (Check[cycle], row[i − 1]);
    for k ← i to i + M_trace − 1;
        ComputeAndSave (row[k], Trace[k mod M_trace + cycle]);
    Traceback (Trace[i mod M_trace + cycle] ...
              Trace[(i + M_trace − 1) mod M_trace + cycle]);
```

Fig. 3. Two-level fixed memory partition. The subroutine calls, respectively, save a row into the row-wide memory, compute the recurrence without saving any state information, compute the recurrence saving choice information, determine the optimal path, and restore a previously stored checkpoint.

approximately $3nm = O(nm)$, assuming that each forward or backward cell calculation requires unit time, and full HMM-style traceback is used rather than the illustrated best path. Thus, with a constant factor of 1.5 slowdown, memory requirements have been reduced asymptotically by $O(\sqrt{n})$.

Moving memory partition. The previous algorithm does not use the checkpoint memory efficiently. When memory is not holding a checkpoint, it should be available for use in traceback computation. The division between checkpoint memory and computation memory can move forward as more checkpoint memory is needed and recede after those checkpoints have been used (Figure 4). If there is enough memory to hold 32 checkpoint values, then the first 32 rows of the dynamic programming matrix can be computed and thrown away, saving the state at the last (32nd) row. After the state is saved, there are, looking forward to the traceback part of the algorithm, only 31 locations available for traceback so only 31 steps should be computed. These actions repeat until the end of the sequence. During the traceback phase of the algorithm, each iteration has the reverse effects on the available memory. With each traceback iteration, a section of the dynamic programming table is recreated from a checkpoint that is no longer taking up memory.

The 2-level algorithm with a moving partition and mM memory can compute $\Sigma_{i=1}^{M} i = M(M+1)/2$ rows and, as with the previous algorithm, execution time is approximately $3nm = O(nm)$. Thus, moving partitions have the same asymptotic performance as fixed partitions, but can

```
M ← size of memory; i ← 0; cycle ← 0;
offset ← 0;
SaveState (row[0], Mem[cycle]);
while (i + M − 1 ≤ n)
    cycle ← cycle +1;
    for k ← i to i + M − 1;
        ComputeCosts (row[k]);
    SaveState (row[i + M − 1], Mem[cycle]);
    i← i + M;
    M← M − 1;
    offset ← offset +M;

for k ← i to n;
    ComputeAndSave (row[k], Mem[k− offset]);
Traceback (Mem[i− offset] ... Mem[n−offset]);

endcycle ← cycle;
for cycle ← endcycle to 0
    offset ← offset−M;
    M← M + 1;
    i← i − M;
    RetrieveCheckpoint (Mem[cycle], row[i − 1]);
    for k ← i to i + M − 1;
        ComputeAndSave (row[k], Mem[k−offset]);
    Traceback (Mem[i− offset] ... Mem[i + M − 1− offset]);
```

Fig. 4. Two-level algorithm with moving partition.

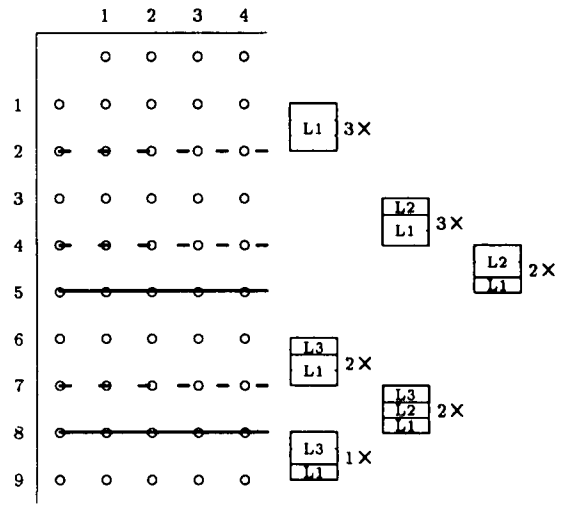

Fig. 5. Example computation of 10 rows using three memory locations and the 3-level algorithm. The solid rows are level-3 checkpoints, while the dashed are level-2 checkpoints. The level-2 checkpoints at rows 4 and 7 do not acutually need to be saved. The memory partitions correspond to the row at the base of the partition diagram. The numbers next to the memory diagrams indicate the number of times the forward calculation is redone.

calculate twice as many rows. They will be assumed from here on.

Multilevel algorithm

A multilevel version of the 2-level algorithm can extend memory use even further. The memory is dynamically divided into sections for level-3 checkpoints, level-2 checkpoints and the basic calculation. Similar to the previous algorithms, after storing a level-3 checkpoint, the 2-level algorithm is used to calculate as many rows as possible before storing the next level-3 checkpoint.

The 3-level algorithm (Figure 5) will, as it is filling its checkpoint memory, call the 2-level algorithm with a range of workspaces from M down to 1; thus, the number of rows this algorithm can compute in mM memory is:

$$\sum_{i=1}^{M} \frac{i(i+1)}{2} = \frac{(M+2)(M+1)M}{6}$$

This yields as asymptotic space requirement of $O(m\sqrt[3]{n})$. While each traceback calculation is performed exactly once, forward calculations are performed up to three times each (once at each level), giving a parallel running time of $4nm = O(nm)$.

In general, the number of rows that can be computed with an L-level algorithm and mM memory is:

$$r_L(M) = \sum_{i=1}^{M} r_{L-1}(i)$$

This recurrence, when $r_1(M) = M$, is solved by:

$$r_L(M) = \binom{M+L-1}{L}$$
$$= \frac{(M+L-1)!}{(M-1)!L!}$$
$$= \frac{M+L-1\ldots M}{L\ldots 1}$$

At one extreme is $L = 1$, the basic algorithm that requires $O(2nm)$ time and $O(nm)$ memory. At $L = 2$, we have the 2-level algorithm, which requires $O(3nm)$ time and $O(\sqrt{nm})$ memory. For small L, $r_L(M)$ is bounded by $O(M^L/L!) = O(M^L/L^L)$, and the L-level algorithm can be calculated in $O((L+1)nm) = O(Lnm)$ time. Changing these to be in terms of n, the L-level algorithm requires $O(Lnm)$ time and $O(mL\sqrt[L]{n})$ space.

At the other extreme is $L = n$. Here, $r_n(2) = n + 1$, and two memory rows are used to calculate an alignment of any length by repeating the forward calculation from $c_{0,0}$ up to each row in turn, using $O(n^2m)$ parallel time. Consider calculating n diagonals with $M = 3$. Setting $r_L(3) = (L+2)(L+1)/2$ equal to n and solving for L gives us $L < \sqrt{2n}$. That is, with $3m$ memory locations, sequence alignment can be performed with an $O(\sqrt{n})$ time penalty. In general, for any small M, $r_L(M) = O(L^M/M!) = O(L^M/M^M)$, and there is an $O(Mn)$ space algorithm with an $O(\sqrt[M]{n})$ runtime penalty.

Thus, the equation for $r_L(M)$ provides a tableau of design points in the space–time tradeoff. Suppose $L = \log n$. The algorithm requires $O(nm \log n)$ time and

$O(m \log n \sqrt[\log n]{n}) = O(m \log n)$ space. Thus, memory requirements can be reduced by a factor of $O(n/\log n)$ with an $O(\log n)$ slowdown.

Single best path

The algorithms so far have been designed for use with the forward–backward HMM training algorithm, in which the backward, or traceback, part of the algorithm is a repeat of the full dynamic programming of the forward section based on the $c_{i,j}$ values. If only the single best path through the dynamic programming matrix is required, such as is used in aligning two sequences, or creating a multiple alignment from HMM, the checkpoint algorithms have better performance.

First, assume that diagonal, rather than row, checkpoints are used. In the case of the 2-level algorithm, there will be $n + m$ diagonals with $O(\sqrt{n+m})$ checkpoints with a spacing of $O(\sqrt{n+m})$ (with moving memory partitions, the spacing is not constant). Each diagonal has a length of at most m.

The price of diagonal checkpoints is two memory inefficiencies. First, there are $n + m$ diagonals, rather than n rows, so more checkpoint locations are required. Second, the computation of a diagonal requires values from the previous two diagonals, i.e. $(i-1, j)$ and $(i, j-1)$ refer to the previous diagonal, while $(i-1, j-1)$ refers to the diagonal before that. In the simplest case, each checkpoint can include two diagonals, and a factor of two memory penalty. For a minor savings, with increased complexity, one can save four costs and one choice, rather than six costs, for each (i, j) pair and still be able to restart the calculation (Figure 6). Here, all circled values are required in the calculation of diagonal two; the three values in the (0, 1) location will be used, with different transition costs, by (0, 2) and (1, 1) on diagonal two and (1, 2) on diagonal three with a third set of transition costs. The choice made in calculating the (1, 1) match value must be saved for traceback.

During the recalculation step, we do not need to recalculate an entire diagonal-length subsection of the dynamic programming array (a block $\sqrt{n+m}$ by m). The maximum distance of any path between a pair of diagonals is the spacing between those two diagonals, or $\sqrt{n+m}$ when $L = 2$. That is, if there is a spacing of 4 between diagonals, and the single best path is known to pass through (10, 30) on the $i + j = 40$ diagonal, then the path must hit the $i + j = 36$ diagonal somewhere between (10, 26) and (6, 30). Thus, we can recalculate only the region of interest, a $\sqrt{n} \times \sqrt{n}$ lower triangle dynamic programming matrix, ignoring anything outside this range. This calculation will be repeated, once per checkpoint, or $\sqrt{n+m}$ times. The total time for the forward pass will be nm, while the time for the recalculations will be $\sqrt{n+m}(\sqrt{n+m})^2$, giving a total time of $nm + \sqrt{n+m}(n+m) = O(nm)$.

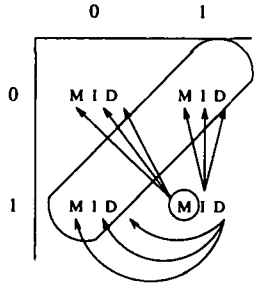

Fig. 6. The circled values in the above dependency diagram must be saved to enable recalculation of the $i + j = 2$ diagonal.

Extending this to multilevel calculation, and simplifying to $n = m$, the time to calculate the single best path through the dynamic programming matrix is the sum of the forward dynamic programming time n^2 and the backward time. At level L, there are $\sqrt[L]{n}$ checkpoints at a spacing of $n/\sqrt[L]{n}$, and the traceback path is calculated using the $L - 1$ algorithm on this $n^{(L-1)/L} \times n^{(L-1)/L}$ problem:

$$T_L(n) \leq n^2 + \sqrt[L]{n} T_{L-1}\left(n^{\frac{L-1}{L}}\right)$$

and $T_L(n) = O(n^2)$ by induction when $L \leq \log n$. Thus, when looking for the single best path, there is no time penalty with the checkpoint method when the number of levels is $O(\log n)$.

The space requirements of the single best path algorithm are also reduced. Here, the space for the 1-level algorithm, $S_1(n)$, is n^2. The 2-level algorithm requires space for the $\sqrt{n}$ diagonal checkpoints, $n\sqrt{n}$ memory locations, as well as space for computing a single $\sqrt{n} \times \sqrt{n}$ block using the next lower level. Thus, $S_2(n) = n\sqrt{n} + S_1(\sqrt{n}) = n\sqrt{n} + n = O(n\sqrt{n})$. Writing this as a recurrence,

$$S_L(n) \leq n\sqrt[L]{n} + S_{L-1}\left(n^{\frac{L-1}{L}}\right)$$

and $S_L(n) = O(n\sqrt[L]{n})$ by induction when $L \geq 1$.

Applying these results to $\log n$ levels, this combined checkpointing and divide-and-conquer algorithm requires $O(n)$ space and $O(nm)$ time to find the single best path, matching the asymptotic performance of Hirschberg's technique.

Parallel algorithm

Dynamic programming is readily parallelizable: values along the diagonals of the dynamic programming matrix can be calculated simultaneously, leading to the typical mapping of Figure 7. Here, m PEs are assigned to the characters in sequence a while the n characters of sequence b shift through the array, one PE per time step. The final calculation of $c_{3,3}$ takes place in PE_3 at time step 6 using a_3 and b_3, while $c_{2,3}$ and $c_{3,2}$ are computed during time step 5.

As with single best path alignment, the parallel algorithms must checkpoint diagonals (Figure 8).

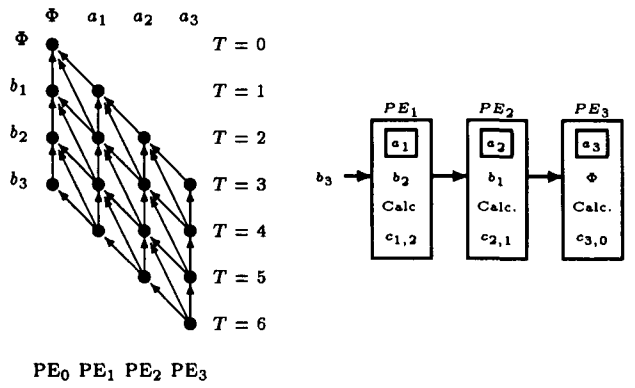

Fig. 7. Mapping dynamic programming to a linear processor array. Arrows indicate dependencies between calculations; the b_3 value will travel from PE_0 at time 3 to PE_3 at time 6. On the right, the array is shown at time step 2 (PE_0 is not shown).

The 2-level algorithm for a parallel processor requires $O(m)$ PEs, $O(\sqrt{n})$ space per PE, and $2(m+n)+m+n = O(n)$ time. Unlike the serial case, in the parallel case, the traceback phase requires asymptotically the same amount of time as the forward phase.

The major problem of mapping the divide-and-conquer algorithm, or the single best path algorithm above, to a linear array of PEs are the data-dependent and arbitrary patterns of sequence movement. The 2-level algorithm, on the other hand, only requires linear shifts of the moving sequence through the array: initially, one sequence is shifted entirely through the array as a complete forward calculation is performed and several checkpoints are saved. For each of the segments of the traceback calculation, the sequence is first shifted backwards through the array to the start of the previous segment, then forward for the recalculation. Because each processing element always computes the same column, the data movement is entirely regular and data independent. Additionally, the number of backward sequence shifts needed to start a new segment is proportional to the number of diagonals in that segment: the asymptotics of the algorithm do not change.

```
M ← size of memory; i ← 0; cycle ← 0;
offset ← 0;
SaveState (diag[0], Mem[cycle]);
while (i + M − 1 ≤ 2n + 1)
    cycle ← cycle +1;
    for k ← i to i + M − 1;
        ComputeCosts (diag[k]);
    SaveState (diag[i + M − 1], Mem[cycle]);
    i← i + M;
    M← M − 1;
    offset ← offset +M;

for k ← i to 2n + 1;
    ComputeAndSave (diag[k], Mem[k− offset]);
Traceback (Mem[i− offset] ... Mem[2n + 1−offset]);

endcycle ← cycle;
for cycle ← endcycle to 0
    offset ← offset−M;
    M← M + 1;
    i← i − M;
    RetrieveCheckpoint (Mem[i], diag[i − 1]);
    ShiftCharactersBackward (M)
    for k ← i to i + M − 1;
        ComputeAndSave (diag[k], Mem[k−offset]);
    Traceback (Mem[i− offset] ... Mem[i + M − 1−offset]);
```

Fig. 8. The parallel version of the 2-level algorithm with moving partition requires calculation on the diagonals.

On a linear processor array, the $\log n$-level algorithm requires $O(m)$ PEs, $O(\log n)$ space per PE, and $O(n \log n)$ time. The $\sqrt{n}$-level algorithm requires three memory elements per PE and $O(n^{1.5})$ time. The per-PE memory requirements of the simple algorithm, as well as the 2-level, 3-level and $\log(n)$-level algorithms, are shown in Figure 9.

Implementation

We implemented the 2-level alignment algorithm, the divide-and-conquer algorithm and, for comparison, the basic $O(n^2)$ space algorithm. We implemented both row and diagonal versions of the checkpoint algorithm, with and without the restricted traceback.

To test the parallel variants, we implemented the 2-level and the basic algorithm on a MasPar parallel computer.

The left graph of Figure 10 shows the execution times, normalized to the basic algorithm, for the serial code, comparing the 2-level algorithm with row checkpoints with and without restrictions [the restricted 2-level row-based algorithm does not achieve the same asymptotic $O(nm)$ time, but can work well in practice], diagonal checkpoints with and without restriction, and Hirschberg's method. The runs were done on an unloaded workstation, and slowdown in CPU time and wall clock time closely correlated up to 2000-long sequences. The 2-level diagonal-restricted algorithm per-

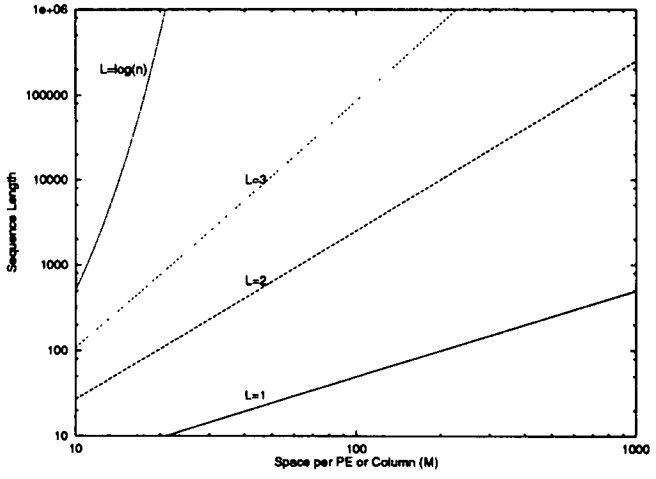

Fig. 9. Approximate alignable sequence length ($n = m$) as a function of M for the algorithms. With fewer than m PEs, multiply the space requirements by the virtual processor ratio, m/P.

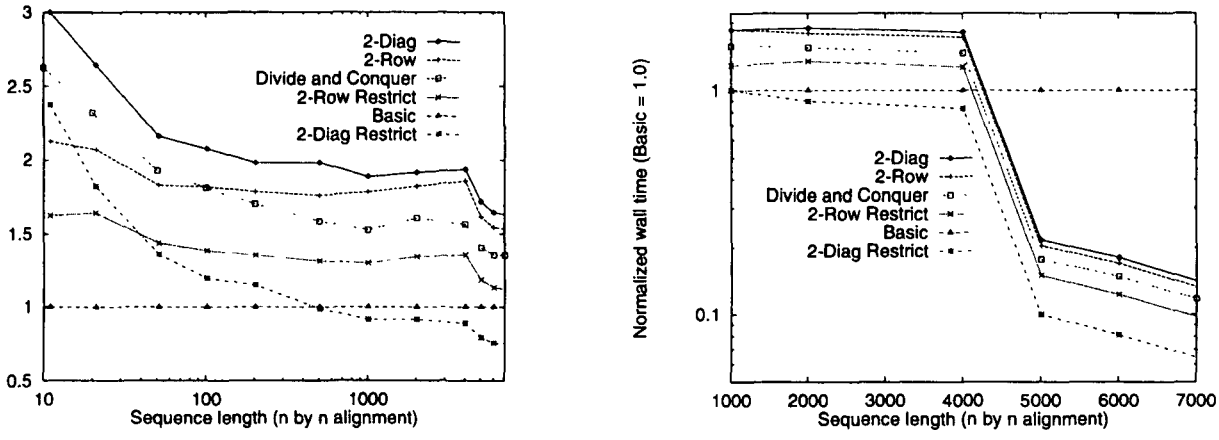

Fig. 10. Normalized CPU time (left) and real time (right) over the basic alignment algorithm for the three reduced-space alignment algorithms on a workstation. The horizontal axis is logarithmic on the left and linear on the right, while the vertical is linear on the left and logarithmic on the right.

forms ~60% faster than the divide-and-conquer algorithm, and even requires less CPU time than the standard algorithm for long sequences. Two aspects of the diagonal algorithm contribute to its performance. First, there is much less function call overhead in comparison to the divide-and-conquer approach. Second, calculating along the diagonals, the compiler is able to extract more parallelism from the computation. For this reason, a comparison of the row-based divide-and-conquer with the row-based restricted alignment algorithm (which performs 15–20% faster) is more equitable.

The prime advantage of the space-saving algorithms is seen for sequences over 2000 characters in real-time measurement, which includes the cost of virtual memory. The results in the right graph of Figure 10 are dramatic: in terms of real time, the normalized execution time moves from 0.9–1.9 at length 2000 to 0.06–0.14 at length 7000. That is, because of memory efficiency, these algorithms are 7–16 times faster than the basic algorithm. As with the shorter sequences, the unrestricted algorithms are ~20% slower than the divide-and-conquer algorithm, while the 2-level row-restricted algorithm is ~16% and the 2-level diagonal-restricted algorithm is ~45% faster than the divide-and-conquer algorithm. For the 7000 × 7000 alignments, the wall clock times are 8 min for the basic algorithm, 68 s for the 2-level diagonals, 64 s for 2-level rows, 56 s for divide and conquer, 47 s for 2-level rows with restricted alignment, and 31 s for 2-level diagonals with restricted alignment.

As with any recursive algorithm, the divide-and-conquer algorithm could be slightly speeded up, without an asymptotic change in space requirements, by solving, for some k, all $k \times k$ subproblems using the basic, non-recursive algorithm, in which case it would begin to take on the flavor of the checkpoint algorithms. We did not do this. Similarly, the multilevel algorithm could be tuned to fit in cache. All performance numbers in this paper are for using the minimum amount of memory required to perform the algorithm. Performance can be increased if more memory is available.

Figure 11 shows execution time, normalized to the basic algorithm, as a function of sequence length for performing an $n \times n$ sequence alignment on the 4096-PE MasPar. The slowdown is a factor of 1.85, partly because this implementation uses the memory-saving technique of Figure 6. Saving two diagonals would be faster. For the 4000 × 4000 alignments, the execution time was 3.0 s for the basic algorithm and 5.4 s for the 2-level algorithm, seven times faster than the serial 2-level unrestricted algorithm.

Discussion

This paper has considered sequence alignment on workstations and parallel processors. The simplest algorithm reduces the amount of memory required for sequence alignment by an $O(\sqrt{n})$ factor with a 20–30% slowdown in execution time for short sequences and, because of the memory efficiency, a factor of 10 or more speedup for long sequences. Multilevel variants of the algorithm can further

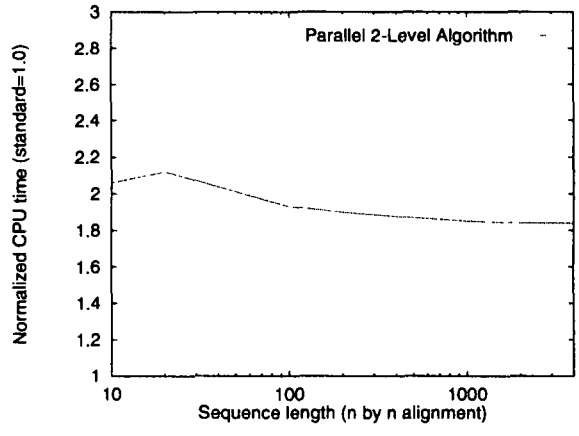

Fig. 11. Normalized CPU time over basic alignment algorithm for the 2-level parallel algorithm.

reduce memory requirements to $O(m \log n)$ space and $O(nm \log n)$ execution time, or to $O(m)$ space and $O(nm\sqrt{n})$ time. When only the single best path is required, there is no time penalty for the family of checkpoint algorithms, and the $\log n$-level algorithm only requires $O(m)$ space, matching the linear space and quadratic time of Hirschberg's algorithm.

Unlike the divide-and-conquer approach, our multiple path algorithms require no problem-dependent partitioning or data movement, and as such are especially appropriate for simple parallel computers with broadcast instructions, in particular linear or mesh-connected processor arrays. Additionally, because of their close relation to the basic algorithm, they are more appropriate for extending the capabilities of existing serial and parallel applications.

One of the most intriguing aspects of this family of algorithms is the breadth of choice. For a given sequence length, the number of levels, and hence the performance penalty, can be chosen according to the amount of available memory in a processing element or cache. Experimental evaluation will be able to optimize the method for any architecture's memory hierarchy.

The algorithms are particularly appropriate to HMM systems such as SAM (Hughey and Krogh, 1996). SAM's core forward–backward (Baum–Welch) training phase requires information about all possible alignments of each sequence to the model, rather than just the best alignment. Thus, Hirschberg's algorithm and its successors, which only locate the best (or several best) possible alignment, could not be used to improve memory performance.

Although this paper has only considered global alignment, the method applies equally well to similarly formed recurrences, such as local alignment and k-best alignment.

An overview of the Kestrel project and the SAM sequence alignment and modeling software system is located at the University of California, Santa Cruz, computational biology group's World-Wide Web page at http://www.cse.ucsc.edu/research/compbio.

Acknowledgements

We especially thank Richard Durbin for his suggestion of the serial single best path, and Zheng Zhang and Webb Miller who suggested the line of analysis for the single best path algorithm. We thank Kevin Karplus for his many useful discussions and the ISMB-95 reviewers for their helpful comments on an early version of this paper. Alicia Grice was supported in part by a Patricia Roberts Harris Fellowship. This work was supported in part by National Science Foundation grants MIP-9423985, BIR-9408579 and CDA-9115268.

References

Edmiston,E.W., Core,N.G., Saltz,J.H. and Smith,R.M. (1988) Parallel processing of biological sequence comparison algorithms. *Int. J. Parallel Program.*, **17**, 259–275.

Gotoh,O. (1982) An improved algorithm for matching biological sequences. *J. Mol. Biol.*, **162**, 705–708.

Gribskov,M., Lüthy,R. and Eisenberg,D. (1990) Profile analysis. *Methods Enzymol.*, **183**, 146–159.

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

Hirschberg,J.D., Hughey,R., Karplus,K. and Speck,D. (1996) Kestrel: A programmable array for sequence analysis. In *Proceedings of An International Conference on Application Specific Array Processors*. IEEE CS, Los Alamitos, CA, pp. 24–34.

Huang,X. (1989) A space-efficient parallel sequence comparison algorithm for a message-passing multiprocessor. *Int. J. Parallel Program.* **18**, 223–239.

Hughey,R. and Krogh,A. (1996) Hidden Markov models for sequence analysis: extension and analysis of the basic method. *Comput. Applic. Biosci.*, **12**, 95–107.

Ibarra,O.H., Jiang,T. and Wang,H. (1992) String editing on a one-way linear array of finite-state machines. *IEEE Trans. Comput.*, **41**, 112–118.

Krogh,A., Brown,M., Mian,I.S., Sjölander,K. and Haussler,D. (1984) Hidden Markov models in computational biology: Applications to protein modeling. *J. Mol. Biol.*, **235**, 1501–1531.

Masek,W.J. and Paterson,M.S. (1983) How to compute string-edit distances quickly. In *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, Reading, MA, pp. 337–349.

Myers,E.W. and Miller,W. (1988) Optimal alignments in linear space. *Comput. Applic. Biosci.*, **4**, 11–17.

Needleman,S.B. and Wunsch,C.D. (1970) A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. Mol. Biol.*, **48**, 443–453.

Sellers,P.H. (1974) On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, **26**, 787–793.

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

Received on July 8, 1996; accepted on September 9, 1996