Parallel Sequence Alignment in Limited Space

J Alicia Grice
cricket@cse.ucsc.edu
Computer Engineering
University of California
Santa Cruz, CA 95064

Richard Hughey
rph@cse.ucsc.edu
Computer Engineering
University of California
Santa Cruz, CA 95064

Don Speck
Computer Engineering
University of California
Santa Cruz, CA 95064

Abstract

Sequence comparison with affine gap costs is a problem that is readily parallelizable on simple single-instruction, multiple-data stream (SIMD) parallel processors using only constant space per processing element. Unfortunately, the twin problem of sequence *alignment*, finding the optimal character-by-character correspondence between two sequences, is more complicated. While the innovative $O(n^2)$-time and $O(n)$-space serial algorithm has been parallelized for multiple-instruction, multiple-data stream (MIMD) computers with only a communication-time slowdown, typically $O(\log n)$, it is not suitable for hardware-efficient SIMD parallel processors with only local communication. This paper proposes several methods of computing sequence alignments with limited memory per processing element. The algorithms are also well-suited to serial implementation. The simpler algorithms feature, for an arbitrary integer L, a factor of L slowdown in exchange for reducing space requirements from $O(n)$ to $O(\sqrt[L]{n})$ per processing element. Using this result, we describe an $O(n \log n)$ parallel time algorithm that requires $O(\log n)$ space per processing element on $O(n)$ SIMD processing elements with only a mesh or linear interconnection network.

Introduction

Sequence comparison and alignment are common and compute-intensive tasks which benefit from space-efficient parallelization. A *sequence* might consist of nucleic acids in a gene fragment, amino acids in a protein, characters in a string, or lines in a file. 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. The most informative alignments have a maximum of matching and, equivalently, a minimum of gaps and mismatches. The rating returned as the comparison result measures the extent to which this can be accomplished.

Good alignments and sequence comparisons come from solutions of an appropriately chosen optimization problem. The problem formulation defines a set of edit primitives, including replace, insert, and delete, and assigns them values or costs. Optimization then maximizes the total match value (Needleman & Wunsch 1970) or minimizes the total cost of mismatches and insert and delete gaps. Selecting an appropriate cost function can result in better alignments. There are many ways to develop a cost metric including using linear hidden Markov models (Krogh *et al.* 1994).

Dynamic programming efficiently organizes sequence comparison by comparing shorter subsequences first, so their costs can be made available in a table (like Figure 1) for the next longer subsequence comparisons. Comparison normally starts from the beginning or end of the sequences, or even both (Hirschberg 1975). The final entry becomes the comparison rating. Exact sequence comparison (with or without gaps) is an $O(n^2)$-time dynamic programming algorithm: serial machines require time proportional to the square of the sequence length to solve the problem.[1] 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 not matching a_i to any character in b,and $\text{dist}(\phi, b_j)$ is the cost of not matching b_j to any character 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 $a_i = b_j$ and 2 otherwise. The recurrence can be efficiently mapped to a linear processor array in several ways, the most obvious being to assign the j-th column of the dynamic programming matrix and the j-th character of b to the j-th processor.

[1] Masek and Paterson 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 (Masek & Paterson 1983).

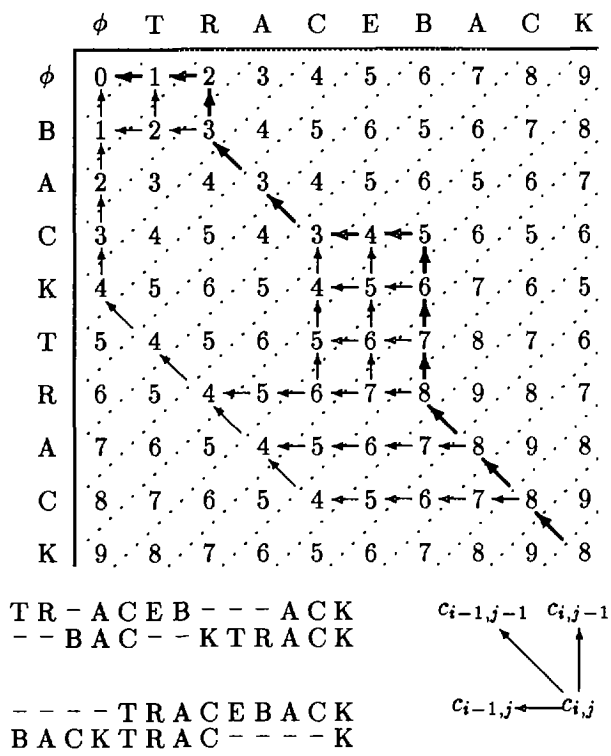


Figure 1: Dynamic programming example to find least cost edit of "BACKTRACK" into "TRACEBACK" using Sellers' evolutionary distance metric (Sellers 1974). Matches propagate the cost diagonally, insert/delete move horizontally/vertically with a cost of 1 per character. On a parallel machine with one processor per column, the dotted lines are isotimes. Below the dynamic programming table are two possible alignments and an illustration of the data dependencies.

Sequence comparison using affine gap penalties involves three interconnected recurrences of a similar form. 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 three gaps. In the most general form (a profile or linear hidden Markov model (Gribskov, Lüthy, & Eisenberg 1990; Krogh *et al.* 1994)), all transition costs between the three states (in a run of matches, insertions, or deletions) and character costs are position-dependent:

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

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

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

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 store the choices made in each minimization during the forward pass, and then use these choices to find an optimal path from $c^D_{n+1,n+1}$ to $c^D_{0,0}$ during the backward or traceback phase. The traceback begins and ends in ϕ columns. This path corresponds to the alignment of the sequence comparison. Storing these choices will require at most one byte per (i,j) cell. However, having to store one byte for each comparison requires $O(n^2)$ space, leading to the need for more space-efficient alternatives.

We have two primary implementation targets: hidden Markov modeling code executed on a general-purpose array processor, and a new parallel processor currently under development. The most computationally-intensive step of the former is a sequence of training iterations that requires an alignment-like calculation on the dynamic programming matrix. This calculation is currently constrained by the amount of local memory per processing element (64 Kbytes). The new parallel processor will have tiny amounts of local memory but, as a result of this work, be able to perform sequence alignment and even hidden Markov model training.

Related Work

Hirschberg (1975) first discovered a linear-space algorithm for sequence alignment, which was then popularized and extended by Myers and Miller (1988). In reducing space from $O(n^2)$ to $O(n)$, these algorithms introduce a small constant (about 2) 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 an 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.

Divide-and-conquer algorithms are well suited to MIMD (multiple instruction stream, multiple data stream) parallel processors such as the Cray T3D or Thinking Machines CM-5. Edmiston, Core, Saltz, and Smith (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 noted that if one considers partitioning 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 uses

a constant amount of memory per processing element. That is, if the $i + j = n$ diagonal is considered, the minimizing point in that diagonal, (i', j') will divide the problem into an upper $i' \times j'$ segment and a lower $(n - i') \times (n - j') = i' \times j'$ segment.

In related work, Huang, Hardison, and Miller applied Hirschberg's techniques to design a linear-space algorithm for local similarity which was similarly parallelized (Huang, Hardison, & Miller 1990; Huang *et al.* 1992).

Huang's parallelization of Hirschberg's algorithm, and the related parallelization of Edmiston's algorithm, is best suited to MIMD 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 re-partitioned into half-sized subsets with the first i' and j' characters of each sequence going to the first processor of one half the 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.

With the goal of $O(n/P)$ memory in each of P processing elements, copies of the sequences cannot be stored in each PE, meaning that the data must be moved though the parallel processor. In practice, because of the expense of full crossbar switches, the best that real parallel machines can do is $O(\log P)$ per message, turning the parallelization into an $O((n^2/P)\log P)$ method requiring $O(n/P)$ space per PE, or $O(n \log n)$ time and $O(1)$ space per PE, with the number of PEs proportional to the lengths of the sequences. (For some parallel processors, the communication coefficient is small, but this does not change the asymptotics.)

The mapping becomes worse on consideration of the minimal SIMD machines that can compute cost functions so efficiently. On a square mesh of processing elements, communication time for arbitrary patterns using the nearest-neighbor connections is $O(\sqrt{P})$, while on a linear array the cost is $O(P)$, producing execution times of $O(n^{1.5})$ and $O(n^2)$ on these architectures when $P = O(n)$.

With this in mind, we have turned to the consideration of parallel sequence alignment on mesh and linearly-connected processor arrays with limited amounts of memory and nearest-neighbor unit-time communication. Additionally, we would like the algorithm to be well-suited to SIMD broadcast machines, the simplest type of parallel processor, which work best when data flow is independent of the actual data, apart from its length. A typical mapping of the dynamic programming calculation is shown in Figure 2. Here, PEs are assigned to each character in sequence a while sequence b shifts through the array, one PE per time step. The final calculation of $c_{3,3}$ takes place in PE_3

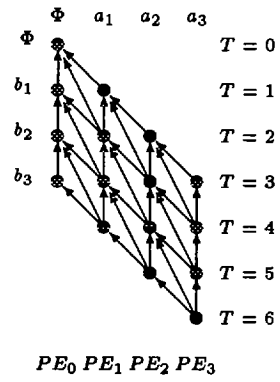


Figure 2: Mapping dynamic programming to a linear processor array.

at time step 6 using a_3 and b_3, while $c_{2,3}$ and $c_{3,2}$ are computed during time step 5.

We hold these goals for two reasons: first, the linearly-connected array is the simplest parallel hardware suitable for generating $O(P)$ speedup in the calculation of dynamic programming costs, and second because we are currently working on such hardware in our project called *Kestrel*. Kestrel is an 8-bit programmable linear processor array in which each processing element (PE) has 288 bytes of memory and an arithmetic logic unit tuned to sequence analysis. We expect 64–128 PEs to fit on a chip, and a minimum of 2048 PEs per system. A single small Kestrel board will greatly outperform general-purpose parallel computers, and perform similarly to or better than other special-purpose hardware (Lipton & Lopresti 1985; Hughey & Lopresti 1991; Chow *et al.* 1991; Gokhale & others 1991; Singh & others 1993) with significantly greater versatility (including, for example, the ability to perform sequence alignments using the algorithms described herein). Beyond our system, however, these results are applicable to any nearest-neighbor-connected array processor, such as the MasPar family of mesh-connected parallel computers, whose global communication is considerably slower than local communication (Nickolls 1990), as well as serial machines.

Algorithm

Let us define several parameters. Let n be the length of the longest of the two sequences compared and m be the length of the shorter. Let P represent the number of processing elements and M the amount of available memory per processing element (PE). For massively parallel implementations, P will be within a small constant of m. Let d be the number of diagonals in the dynamic programming matrix, where $d = n+m+1$, or if sequence lengths are equal $d = 2n+1$. To simplify the discussion, we assume without loss of generality that $n = m$. Similarly, we assume that $P = n$; if $P < n$,


```
for i ← 1 to 2n + 1
    ComputeAndSaveOutcome (diagonal[i], Trace[i]);

/* Traceback */
i ← n+1
j ← n+1
state ← delete
while (i > 0 && j > 0)
    print state (i,j) = state
    state = minstate (i,j,state)
    case state:
        delete: j = j − 1
        insert: i = i − 1
        match: i = i − 1; j = j − 1
    endcase
endwhile
```

Figure 3: Basic algorithm and memory partition

then all time and space bounds should be multiplied by the virtual processor ratio, n/P.

The basic alignment algorithm is displayed in Figure 3. It includes two parts. 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 memory used to store the minimization choices is called traceback memory, and must include $\Omega(n)$ elements per column or processing element.

The simple alignment algorithm's limitation to $n = O(M)$ can be overcome by taking advantage of the fact that it is not necessary to save the outcome of all comparisons to perform the traceback. It is only necessary to have enough information to recompute the comparisons as required. To efficiently recalculate the comparisons of a diagonal in the dynamic programming matrix, the state of computation a little 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 diagonals, alignments can be performed in limited space.

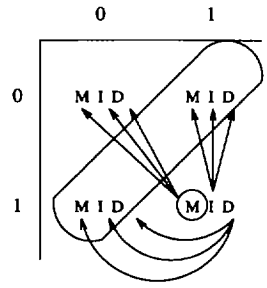


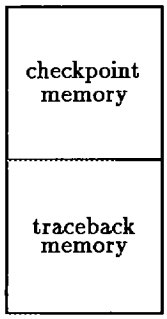
Figure 4: The circled values in the above dependency diagram must be saved to enable recalculation of diagonal 2.

Algorithm A: 2-Level Fixed Partition

To find alignments for sequences longer than M, the available memory can be 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 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. This information can be saved in four values per processing element. With affine gap costs, this corresponds to saving four values per column, as shown in Figure 4. 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. For the remainder of this discussion, we assume that M is measured by the number of checkpoints it can hold, and that there is a one-to-one correspondence between checkpoint and traceback storage. The space efficiency of the algorithms could be improved by noting that storing choices requires less memory than storing checkpoints (six bits as opposed to four words).

After state has been saved the next M_{trace} diagonals 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} diagonals. The previous M_{trace} comparison outcomes are recomputed from checkpoint information, and a traceback is performed on those diagonals. This algorithm is shown in Figure 5. We call this a two-level method because of the hierarchy in calculating values: checkpoints combined with simple (level-1) forward and traceback calculations.

The performance of algorithm A is simple to analyze. The greatest number of diagonals that can be




```
iterations ← ⌈(2n + 1)/M_trace⌉ − 1;
for cycle ← 0 to iterations − 1
    for i ← cycle * M_trace to (cycle + 1) *M_trace − 1
        ComputeCosts (diagonal[i]);
    if (cycle < iterations −1)
        SaveState (Check[cycle]);

diag ← 2n;
for cycle ← iterations to 0
    for i ← cycle * M_trace to diag
        ComputeAndSaveOutcome (diagonal[i], Trace[i]);
    Traceback (Trace[cycle * M_trace... diag]);
    diag ← cycle *M_trace − 1;
    if (cycle > 1)
        RetrieveCheckpoint (Check[cycle −2]);
```

Figure 5: Algorithm A and memory partition

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 diagonals to n, the amount of space per PE required for sequences of length n is $O(\sqrt{n})$, while the parallel time required for the calculation is approximately $3d = O(n)$, assuming each forward or backward cell calculation requires unit time. Thus, with a constant factor, 50% slowdown, memory requirements have been reduced asymptotically by $O(\sqrt{n})$.

The major problem of mapping Huang's parallelization to a linear array of processing elements is the data-dependent and arbitrary patterns of sequence movement. Algorithm A, 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 forwards 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 (the shifts will form around 10% of the cell program on Kestrel).

We do assume that there is sufficient memory *outside* the array, connected to the two end PEs, to store the complete moving sequence. Note that the basic algorithm requires sufficient memory at one side of the array to store the moving sequence. The second staging memory is simply an extension of this concept, and is available on target architectures. For example, in a general-purpose SIMD processor, the array controller's memory could be used.

Algorithm B: 2-Level Moving 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. This improvement is incorporated into Algorithm B in Figure 6. If there is enough memory to hold 32 checkpoint values, then the first 32 diagonals of the dynamic programming matrix can be computed and thrown away, saving the state at the last (32nd) diagonal. 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 is having 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.

Algorithm B, with M memory per PE, can compute $\sum_{i=1}^{M} i = M(M+1)/2$ diagonals and, as with the previous algorithm, parallel running time is approximately $3d = O(n)$. Thus, Algorithm B has similar performance as Algorithm A, but makes more effective use of its memory.

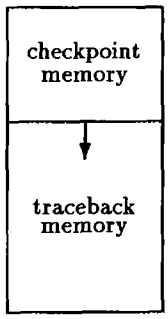
Algorithm C: 3-Level Moving Partitions

A multi-level version of Algorithm B can extend memory use even further. Algorithm C, shown in Figure 7, is a 3-level algorithm: level-3 checkpoints, level-2 (Algorithm B) checkpoints, and the basic calculation. Similar to the previous algorithms, after storing a level-3 checkpoint, Algorithm B is used to calculate as many diagonals as possible before storing the next level-3 checkpoint.

Algorithm C will, as it is filling its checkpoint memory, call Algorithm B with a range of workspaces from M down to 1, thus the number of diagonals this algorithm can compute is

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

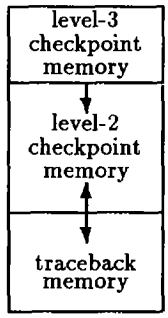
This yields an asymptotic space requirement of $O(\sqrt[3]{n})$. While each traceback calculation is performed exactly




```
M ← size of memory; diags ← 0; cycle ← 0;
while (diags + M − cycle −1 ≤ 2n + 1)
   for i ← diags to diags + M − cycle −1
      ComputeCosts (diagonal[i]);
   diags ← i + 1;
   if (not last iteration)
      SaveState (Memory[cycle]);
      M ← M − 1;
   cycle ← cycle + 1;

iterations ← cycle;
for cycle ← iterations to 0
   for i ← diags to diags + M − cycle −1
      ComputeAndSaveOutcome (diagonal[i], Memory[i]);
   Traceback (Memory[diag ... i]);
   M ← M + 1;
   diags ← diags − (M − cycle + 1);
   if (cycle > 1)
      RetrieveCheckpoint (Memory[cycle − 2]);
```

Figure 6: Algorithm B and memory partition




```
M ← size of memory; diags ← 0; cycle ← 0;
while (diags + ½M² − ½M ≤ 2n + 1)
   for i ← diags to diags + ½M² − ½M
      ComputeCosts (diagonal[i]);
   diags ← i + 1;
   if (not last iteration)
      SaveState (Memory[cycle]);
      M ← M − 1;
   cycle ← cycle + 1;

b ← 2n + 1− diags;
iterations ← cycle;
for cycle ← iterations to 0
   Algorithm B on diags to diags + b
   M ← M + 1;
   b ← ½M² − ½M;
   diags ← diags − (b + 1);
   if (cycle > 1)
      RetrieveCheckpoint (Memory[cycle − 2]);
```

Figure 7: Algorithm C and memory partition

once, forward calculations are performed up to three times each (once at each level), giving a parallel running time of $4d = O(n)$.

One additional inefficiency in the algorithms as expressed so far can be seen in Figure 8, an application of Algorithm C to calculate 15 diagonals with only three memory locations. Consider the level-2 checkpoint at diagonal 4. Processors 5–8 do not need to store any values for this checkpoint, apart from, possibly, initial conditions. Thus, some of their memory is wasted. This problem could be solved if instead of diagonal checkpoints, row checkpoints were used. While this would be fine for a serial computation, it would slow down the SIMD parallel computation. Because all values in a given row are calculated at different times, the inner core of the cell program would have to include a save instruction that was executed by only one processing element during each iteration. Thus, in exchange for slight memory inefficiency (especially slight when the fixed sequence is smaller than the moving sequence), a simpler and faster algorithm is gained.

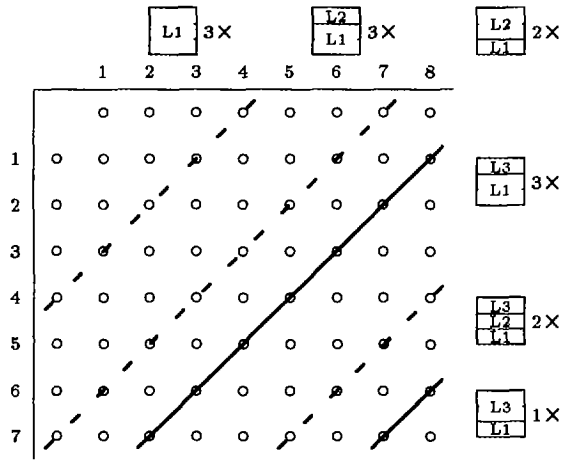


Figure 8: The computation of fifteen diagonals using three memory locations and the 3-level algorithm C. The solid diagonals are level-3 checkpoints, while the dashed lines are level-2 checkpoints. The level-2 checkpoints at diagonals 7 and 12 do not actually need to be saved. If the initial conditions are to be stored, rather than recomputed, an additional level-3 checkpoint is required for diagonal 0. The numbers next to the memory diagrams indicate the number of times the forward calculation is redone.

Algorithm D: Fully-Recursive Partitioning

The previous sections discussed base case 1-level alignment, 2-level alignment with fixed (Algorithm A) and moving (Algorithm B) partitions of M into M_{check} and M_{trace}, and 3-level alignment with moving partitions (Algorithm C). In this section, we consider arbitrary L-level recursion, in particular the case in which the number of levels is equal to the amount of memory.

Having seen 2- and 3-level algorithms, a similar 4-level algorithm will be able to calculate

$$\sum_{i=1}^{M} \frac{(i+2)(i+1)i}{6} = \frac{(M+3)(M+2)(M+1)(M)}{24} = O(M^4)$$

diagonals. Additionally, each forward calculation will be repeated at most 4 times, leading to $5d$ cell program calculations, on consideration of the traceback phase.

In general, the number of diagonals that can be computed with a level-L algorithm and M memory is

$$d_L(M) = \sum_{i=1}^{M} d_{L-1}(i).$$

This recurrence, when $d_1(M) = M$, is solved by:

$$\begin{aligned} d_L(M) &= \binom{M+L-1}{L} \\ &= \frac{(M+L-1)!}{(M-1)!L!} \\ &= \frac{M+L-1\ldots M}{L\ldots 1}. \end{aligned}$$

Thus, for a given number of levels L, $O(M^L/L^L)$ diagonals can be calculated in $O(Ld)$ time. Changing these to be in terms of n, the L-level algorithm will requires $O(Ln)$ parallel time and $O(L\sqrt[L]{n})$ memory per PE.

At one extreme is $L = 1$, the basic algorithm that requires $O(n)$ parallel time and $O(n)$ memory per PE. At the other extreme is $L = n$. Here, a single memory location is used to calculate an alignment of any length by repeating the forward calculation from $c_{0,0}$ up to each diagonal in turn, using $O(n^2)$ parallel time.

Now suppose that $\log n$ levels are used. The algorithm requires $O(n \log n)$ time and $O(\log n\ ^{\log}\sqrt[\log]{n}) = O(\log n)$ space per PE. Thus, memory requirements on a linearly-connected parallel processor can be reduced from $O(n)$ per PE to $O(\log n)$ per PE with an $O(\log n)$ slowdown.

The per-PE memory requirements of the simple algorithm, as well as algorithms B, C, and D, are shown in Figure 9.

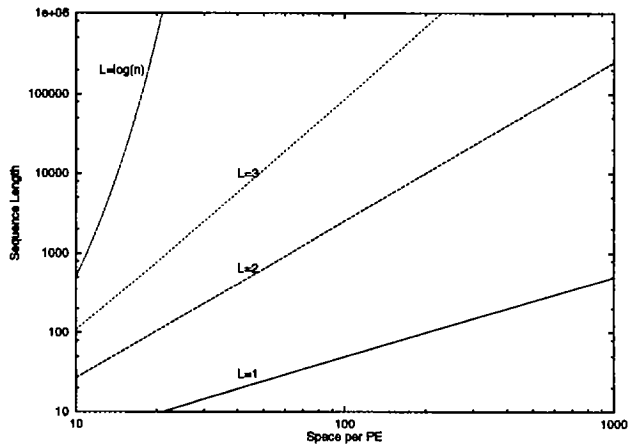


Figure 9: Alignable sequence length as a function of PE memory for the algorithms. For fewer than n PEs, multiply space requirements by the virtual processor ratio, n/P.

Practical Considerations

The previous sections have, purposely, simplified several aspects of the algorithms in the hope of presenting the broader picture. There are several practical considerations that deserve a brief mention.

The discussion generally assumed that the number of processors was equal to the length of the sequences. As long as the number of processors is within a small constant factor of the length of the sequences, these algorithms can still be applied. In this case, a virtual processor ratio greater than one is used; a single PE will compute values of two or more columns. In the case of two characters per PE, this will halve the amount of memory available for alignment.

There is a second drain on available memory, especially when protein sequences are being compared. The match cost tables, as well as transition costs in the case of an linear hidden Markov model (HMM), must be stored in each processing element. Fortunately, since our systolic mapping of choice keeps one sequence fixed, this will only be a single column of the cost table, or 20 numbers.

A final reduction to available memory will occur on the Kestrel architecture when calculation modulo-256 is not appropriate, in particular, in subsequence alignment and HMM training. In this case, multiple-precision arithmetic must be used, usually with two bytes per dynamic programming table entry. In spite of the need for multiple-precision $c_{i,j}$ values, individual costs, such as for the 20 amino acids, can still be stored with lower precision. HMM training on the MasPar, our second application target, will always use that machine's native 32-bit numbers.

A memory savings can be had by noting that only six bits of information, corresponding to the two minimizations performed in the calculation for each of the three states, needs to be saved to perform traceback. To realize this savings, the program loop must be unrolled four times for all possible positionings of the first

of the six bits within a byte. (Theoretically, five 3-way decisions could be packed into a byte, rather than just four.)

Another point of interest is when a sequence or model does not closely match the array length, such as if it has $P/2$ or $1.5P$ characters, where P is the number of processors. The MasPar features direct communication to any of the PEs in the array, so that placing and retrieving data from any PE has little cost. In the HMM code, if a model is of length L, $\lfloor P/L \rfloor$ copies of the model are placed in the array and used simultaneously. While the Kestrel array allows direct, fast input to any one PE in the array, output from the PEs is slower. There are three solutions. First, most sequence comparison problems can be extended to include effective no-operation costs or pad characters, which can be used to ensure that the flow through the "excess" processing elements does not affect results. Second, data can be fed directly into any PE via the broadcast mechanism, with results retrieved from the end PE as usual. Third, the data movement can be folded, so that data enters and exits the same end of the array. For example, if the stored sequence is of length $1.5P$, the first PE could store the first and last character, the second PE the second and next-to-last character, and so on, with the moving character sequence reversing direction three quarters of the way through the array.

Finally, in several places, this paper makes the simplifying assumption that the two sequences are of equal length. This assumption is really only used in the creation of Figure 9. The number of levels required by the algorithms, given that one sequence has been successfully placed in the array, is entirely determined by the amount of memory available per logical PE (ie, half the total memory if the virtual processor ratio is 2) and the length of the moving sequence. For sequences of length n and m the fully-recursive algorithm will require $O(n)$ PEs each with $O(\log m)$ memory with a $O(\log m)$ slowdown.

Conclusions

This paper has considered sequence alignment on SIMD parallel processors with nearest-neighbor interconnections and limited local memory. The simplest algorithm squares the length of sequence that can be aligned in a given amount of memory with only a 50% slowdown in execution time. Multilevel variants of the algorithm can further reduce memory requirements to $O(\log n)$ space per processing element and $O(n \log n)$ parallel execution time. Our algorithms do not match the constant space per processing element of Huang's parallelization of Hirschberg's algorithm. However, unlike those algorithms, our algorithms require no problem-dependent partitioning or data movement, and as such are appropriate for SIMD parallel computers, 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.

Two current projects at the University of California, Santa Cruz, will use these methods: the *SAM* sequence alignment and modeling software system, and the *Kestrel* sequence analysis co-processor.

SAM is a suite of programs for using and training linear hidden Markov models (Krogh *et al.* 1994; Hughey & Krogh 1995). The SAM distribution includes a parallel version that runs on the MasPar family of SIMD mesh-connected computers. SAM's core expectation-maximization 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 possible alignment, could not be used to improve memory performance. When implemented, the 2-level algorithm B will square our current sequence length limit of about 2000.

As mentioned previously, the Kestrel processing elements will each have 288 bytes of local memory. The 3-level Algorithm C will enable sequence alignment of over 30,000 nucleotides. Indeed, the discovery of these algorithms produced major architectural simplifications from our original design with 2048 bytes of memory per PE, illustrating the importance of developing algorithms and applications in concert. A prototype system is expected in early 1997.

An overview of the *Kestrel* project and the *SAM* sequence alignment and modeling software system is located at the University California, Santa Cruz, computational biology group's World-Wide Web page at `http://www.cse.ucsc.edu/research/compbio`, and related papers are available from `ftp.cse.ucsc.edu`, in the `protein`, `dna`, and `rna` subdirectories of `pub`.

Acknowledgments

We especially thank Kevin Karplus for his many useful discussions and the anonymous reviewers for their helpful comments. Alicia Grice was supported by a Patricia Roberts Harris Fellowship. The Kestrel project has been supported in part by funds granted by the Natural Sciences Division of the University of California, Santa Cruz.

References

Chow, E.; Hunkapiller, T.; Peterson, J.; and Waterman, M. S. 1991. Biological information signal processor. In Valero, M., et al., eds., *Proc. Int. Conf. Application Specific Array Processors*, 144–160. IEEE Computer Society.

Edmiston, E. W.; Core, N. G.; Saltz, J. H.; and Smith, R. M. 1988. Parallel processing of biological sequence comparison algorithms. *International Journal of Parallel Programming* 17(3):259–275.

Gokhale, M., et al. 1991. Building and using a highly parallel programmable logic array. *Computer* 24(1):81–89.

Gribskov, M.; Lüthy, R.; and Eisenberg, D. 1990. Profile analysis. *Methods in Enzymology* 183:146–159.

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

Huang, X.; Miller, W.; Schwartz, S.; and Hardison, R. C. 1992. Parallelization of a local similarity algorithm. *CABIOS* 8(2):155–165.

Huang, X.; Hardison, R. C.; and Miller, W. 1990. A space-efficient algorithm for local similarities. *CABIOS* 6(4):373–381.

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

Hughey, R., and Krogh, A. 1995. SAM: Sequence alignment and modeling software system. Technical Report UCSC-CRL-95-7, University of California, Santa Cruz, CA.

Hughey, R., and Lopresti, D. P. 1991. B-SYS: A 470-processor programmable systolic array. In Wu, C., ed., *Proc. Int. Conf. Parallel Processing*, volume 1, 580–583. CRC Press.

Krogh, A.; Brown, M.; Mian, I. S.; Sjölander, K.; and Haussler, D. 1994. Hidden Markov models in computational biology: Applications to protein modeling. *J. Mol. Biol.* 235:1501–1531.

Lipton, R. J., and Lopresti, D. 1985. A systolic array for rapid string comparison. In Fuchs, H., ed., *1986 Chapel Hill Conference on VLSI*. Rockville, MD: Computer Science Press. 363–376.

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.* Reading, MA: Addison-Wesley. 337–349.

Myers, E. W., and Miller, W. 1988. Optimal alignments in linear space. *CABIOS* 4(1):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.

Nickolls, J. R. 1990. The design of the Maspar MP-1: A cost effective massively parallel computer. In *Proc. COMPCON Spring 1990*, 25–28. IEEE Computer Society.

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

Singh, R., et al. 1993. A scalable systolic multiprocessor system for bioseqeunce similarity analysis. In Snyder, L., ed., *Symposium on Integrated Systems*, 169–181. University of Washington.