Parallel Similarity Search and Alignment with the Dynamic Programming Method

Adam R. Galper and Douglas L. Brutlag
Section on Medical Informatics
Department of Biochemistry
Stanford University School of Medicine
Stanford, California 94305

April 21, 1990

Abstract

We consider the problem of similarity search and alignment of biological sequence data using multiple processors operating in parallel. We review the basic dynamic programming method commonly used in sequential algorithms and adapt the theory to permit multiple processors to cooperatively solve the problem. Fast parallel algorithms exist that do not employ the basic dynamic programming method, but they require hypothetical computational architectures. We outline several practical approaches for parallelizing the basic method to run on existing architectures. Finally, we present results of our parallel implementations, written in C for the Encore Multimax, a 16-node shared-memory multiprocessor.

1. Introduction

The nearly exponential growth rate of biological sequence databases threatens to overwhelm existing computational methods for analysis and searching [4]. Current algorithms assume a serial computer architecture; multiple processors working in concert, however, can partition a task and effectively accelerate computation. Multiprocessing techniques can be readily applied to the pattern recognition tasks of molecular biology computing.

Two tasks of considerable importance are similarity search and alignment. For a query string S and a library string X, we define similarity search to be the problem of finding the substring of X most similar to S. We define alignment to be the determination of the sequence of insertions, deletions and transformations sufficient to convert one string into another.

Both problems can be solved using a well-established dynamic programming method (the *basic* method) to compute distances, often called *edit* or *evolutionary* distances, between related strings. The method was independently discovered by several researchers in a number of fields. Needleman and Wunsch first applied the method to biological sequences in 1970 [12]. The computational complexity of this exhaustive approach is $\mathcal{O}(mn)$, where m is the length of the query sequence and n is the size of the library.

Other methods have been developed which do not rely on dynamic programming, but show similar worst-case complexity. The widely-used Wilbur, Lipman and Pearson [8,13] algorithm (FASTP/FASTA) exhibits improved average time complexity by looking at k-tuples of symbols; the primary control mechanisms are hashing and the "diagonal" method.

Masek and Paterson present a general algorithm [9] to compute edit distance which runs in time $\mathcal{O}(mn/\min(\log n, m))$, with some restrictions. Dynamic programming is used, but the technique differs from that of the basic method. It is unclear whether this algorithm has been applied to biological sequence comparison.

Few researchers have looked at the problem of similarity search and alignment in a parallel context. Several theoretical results are notable. Landau and Vishkin [7] show that $m^2 + n$ processors can find all library subsequences differing from the query sequence up to a distance of k in time $\mathcal{O}(\log m + k)$. Mathies [10] has recently developed a parallel algorithm for determining edit distances which runs in $\mathcal{O}(\log m \log n)$ time, requiring mn processors.

These results rely on hypothetical models of computation; in particular, they assume the existence of a concurrent-read, concurrent-write parallel random access machine (CRCW PRAM) with p synchronous processors, each with access to a common memory. The closest thing to a PRAM in today's arsenal of computing machines is the shared-memory multiprocessor. At present, however, shared-memory machines have been designed with at most 512 processors. The most widely available, large-scale, shared-memory machine is the BBN Butterfly, with up to 508 processors.[1] Unfortunately, with biological sequences, n often exceeds one million, and the products $m^2 + n$ and mn render the aforementioned algorithms impractical.

We present here results of efforts to adapt the basic dynamic programming method for execution on an existing shared-memory architecture. Our intent is to develop an opportunistic algorithm that uses all available processors to achieve substantial

[1]The 512-processor IBM RP3 and the Butterfly architectures provide both shared-memory and message-passing paradigms.

speedup, without sacrificing the elegance and simplicity of the basic method. First, we review the method as it exists in sequential algorithms. After describing the fundamental issues of parallel design, we present the extensions to the basic method necessary to support multiple processors. Finally, we present results of several implementations and discuss the factors which limit the speedup.

2. The Dynamic Programming Method

The classical pattern matching problem asks whether S is a substring of X, given strings S and X composed of symbols in an alphabet Σ. In large databases, however, it is often desirable to retrieve substrings of X that are sufficiently like S, especially if S is not a substring of X. The approximate pattern matching problem is thus stated: Given strings S and X, find a substring T in X, such that the *distance* from S to T is no more than some value d.

The edit distance, d, is a cost function defined over a set of operations. If a and b are symbols in Σ, each editing operation a $\longrightarrow$ b has a cost metric δ, which exhibits the following properties:

1. $\delta(a,b) > 0$ for $a \neq b$, $\delta(a,b) = 0$ for $a = b$
2. $\delta(a,b) = \delta(b,a)$
3. $\delta(a,c) \leq \delta(a,b) + \delta(b,c)$

For our purposes, the basic edit operations are insertion, deletion and substitution, although additional specialized operations, like symbol transposition and phonetic transformations can be defined depending on the alphabet and problem domain. Clearly, using these edit operations, every string A can be transformed into a string B in a variety of ways; the total cost of a transformation is simply the sum of the costs of each edit operation. The edit distance is defined as the minimum total cost of a sequence of editing operations that transforms A into B.

Alternatively, the problem can be formulated as one of determining the maximum similarity between two sequences, instead of the minimum distance. Needleman-Wunsch and subsequent optimizations use this criteria. Following the computer science literature, we choose to frame the problem as one of distance minimization. [14] discusses the equivalence of minimum distance and maximum similarity criteria.

The basic dynamic programming method for finding the edit distance between any two strings $A = a_1, a_2, ..., a_m$ and $B = b_1, b_2, ..., b_n$ is to compute the $(m+1) \times (n+1)$ matrix from the recurrence relations:

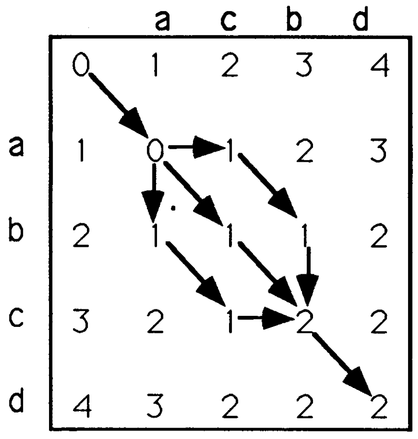

Figure 1: The edit matrix for "abcd" and "acbd". Each diagonal arc is a substitution or match, each vertical arc is a deletion and each horizontal arc represents an insertion.

$$
\begin{aligned}
d_{00} &= 0 \\
d_{ij} &= \min(\; d_{i-1,j-1} + \text{IF } a_i = b_j \text{THEN 0 ELSE } \delta(a_i \rightarrow b_j), \\
&\qquad\qquad d_{i-1,j} + \delta(a_i \rightarrow e), \\
&\qquad\qquad d_{i,j-1} + \delta(e \rightarrow a_i) \;)
\end{aligned}
$$

where e is the empty string ($a_i \rightarrow e$ represents a deletion and $e \rightarrow a_i$ an insertion). The value at d_{mn} is the edit distance between A and B. Clearly, since each d_{ij} requires constant time to compute, the algorithm runs in $\mathcal{O}(mn)$ time.

Furthermore, the sequence of editing operations along the path to the edit distance can be reconstructed by drawing a directed arc from d_{ij} to $d_{i'j'}$ if and only if the minimization step used d_{ij} to derive $d_{i'j'}$. This process is called *backtracking.* If each of the editing operations has the same cost $\delta = 1$, the edit matrix comparing the strings "abcd" and "acbd" can be depicted along with its directed dependency graph, called an edit path, as in Figure 1.

The edit path is interpreted as the alignment of the two sequences, S and X; in addition, dynamic programming and backtracking techniques can be used to determine the best substring T of X, such that T most closely resembles S, and the regions within X which show the greatest similarity to regions within S.

3. Parallel Design Issues

Amdahl's Law states that the maximum speedup achievable for an existing sequential algorithm is 1/S, where

$$\begin{aligned}
&\mathrm{S} = \text{Serial Fraction of the Algorithm}\\
&\mathrm{C} = \text{Parallelizable Fraction of the Algorithm}\\
&\mathrm{S}+\mathrm{C} = 1\\
&\text{Speedup} = \frac{T_1}{T_p} = \frac{\text{Computation Time for 1 processor}}{\text{Computation Time for p processors}} = \frac{\mathrm{S+C}}{\mathrm{S}+\frac{\mathrm{C}}{p}}
\end{aligned}$$

Thus, as $p \to \infty$, Speedup $\to (\mathrm{S}+\mathrm{C})/\mathrm{S} = 1/\mathrm{S}$. The limit imposed by Amdahl's Law must be tolerated, unless the algorithm can be modified to better exploit a parallel environment.

In designing a parallel application, close attention must be paid to the underlying architecture. Multiprocessors currently take a number of forms: shared-memory, message-passing, data-flow and systolic architectures often require entirely different approaches to the parallelization of an application.[2] In general, there are three design considerations that dominate in parallel programming: granularity, communication/synchronization, and load balance.

Granularity refers to the size of the task, or the amount of computation, each processor must handle between extraprocessor communication. Tasks are not necessarily a constant size; in message-passing architectures, extraprocessor communication is expensive and task size should be large. In shared-memory and data-parallel machines, data transmission between processors is cheap and tasks can be of a finer grain. Synchronization between processors, however, becomes crucial with smaller tasks; the overhead associated with interprocess communication can often erase any benefit parallel computation has to offer. As granularity decreases, load balance improves, in that the likelihood of a given processor to be in a waiting state decreases due to the increased number of tasks. The tradeoffs between granularity, synchronization and load balance emphasize the care with which parallel applications must be built.

In addition, parallel applications should scale well. Ideally, when p processors are working on a problem, computation time speedup is p. This is rarely the case, however, since most problems do not *decompose* well enough to permit perfect linear speedup. In addition, the computational overhead incurred by interprocessor communication and sychronization often reduces the speedup drastically.

4. Parallel Decompositions

The most interesting aspect of the dynamic programming technique is not the actual calculation of an individual d_{ij}, but rather the flow of control, driven by the availability

[2]See [?] for a discussion of parallel architectures.

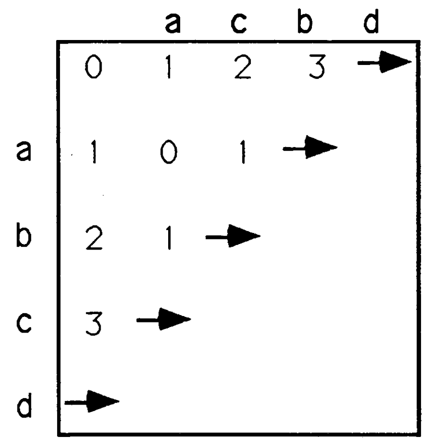

Figure 2: The arrows indicate computable d_{ij}'s. Notice how the arrows align themselves in diagonal patterns.

of data, across the entire matrix. From the above recurrence relation, a given d_{ij} can only be computed if all d_{xy}, where $x < i$ and $y < j$, have already been computed. We say d_{ij} is *computable* if $d_{i,j-1}$, $d_{i-1,j-1}$, and $d_{i-1,j}$ have already been computed. In other words, any given row of the matrix is sequentially computable only up to the i-th index, where i is the last computed index of the previous row. This is depicted schematically in Figure 2.

In many respects, dynamic programming, with its explicit data dependencies and recursion, is ideal for the data-flow model of parallel computation, in which the execution of instructions is triggered solely by the availability of operands. However, data-flow architectures only exist as prototypes and their viability has yet to be proven.

We have decomposed the dynamic programming method for implementation on a shared-memory multiprocessor. Shared-memory architectures share a single global memory and offer low overhead extraprocessor communication. These machines usually have powerful processors geared for medium- to coarse-grained tasks. Process synchronization is key in a shared-memory machine; mutual-exclusion locks must be used to prevent processors from overwriting the same memory location simultaneously.

The Encore Multimax is a 16-node multiprocessor with 32 Mb of shared-memory, running a parallel version of the Unix operating system. The Multimax is configured with NS32032 32-bit CPU's, each rated at .75 MIPS. Two Multimax processors share a *cache*, which mediates rapid access to shared-memory.

We consider two parallel decompositions, differing in their perspective on the flow of computation across the matrix. For the following discussion, we assume the size

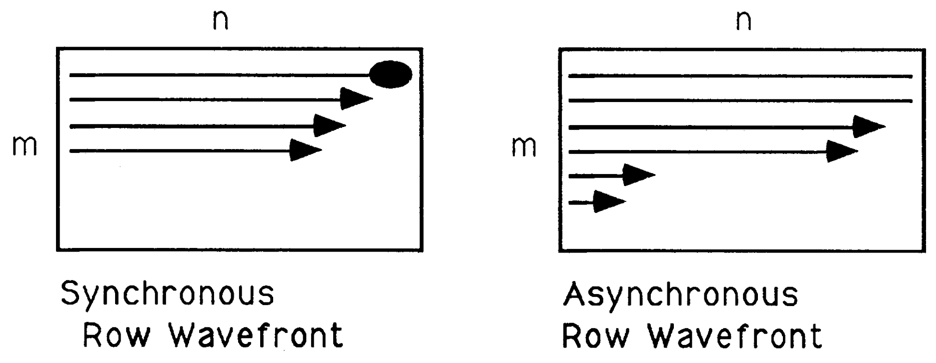

Figure 3: *Synchronous*: when a processor completes a row, it waits for all other processors to finish their rows before stepping in unison to the next block of rows. *Asynchronous*: upon completion of a row, a processor immediately grabs the next available row.

of the library (n) is much greater than the size of the query sequence (m) in an $m \times n$ edit matrix. We assume nothing about the number of processors available for a decomposition.

4.1 The Row Wavefront Approach

In the row wavefront approach, we assign one processor per row to perform ordered computations across the row. When a processor detects that the required entry in the previous row has not been computed, the processor blocks (waits). As soon as the previous row's entry is computed, the blocked processor reactivates. The resulting effect is that of leading and lagging processors, with each row one position ahead of the next row. This approach is a variation of a parallel programming paradigm known as the producer/consumer model, in which one processor feeds off the results of another. Here, each processor is a producer as well as a consumer.

We distinguish between synchronous and asynchronous wavefronts (Figure 3). Assume p processors are available for the evaluation of the entire matrix. A synchronous wavefront (sRWF) advances p rows at a time. When a processor finishes its row, it waits until all other processors have finished theirs. Then, all processors are assigned new rows at the same time. An asynchronous row wavefront (aRWF) advances continuously, by wrapping around the edit matrix. When a processor completes a row, it grabs the next available row, which is typically p rows ahead. It should be apparent that asynchronous wavefronts minimize blocking time and are thus preferred.

Assume that each entry in the edit matrix requires computation of uniform size. If p processors are available and p is less than the length of the query sequence, m, then the first p rows can be completely computed in time $n+p$. With a synchronous wavefront, each additional block of p rows requires $n+p$ time steps as well. Computation of the entire matrix requires time $\lceil m/p \rceil * (n+p)$ or $\lceil mn/p + m \rceil$. With an asynchronous wavefront, the first p rows require $n+p$ time steps, but each additional block of p rows requires time n. Computation of the entire matrix using asynchronous row wavefronts thus requires time $\lceil mn/p+p \rceil$. Asynchronous and synchronous wavefronts require the same amount of time when $p = m$, namely $n + m$ time steps.

We store the entire edit matrix, requiring $\mathcal{O}(mn)$ space, to permit the reconstruction of edit paths. In practice, this space requirement strains system resources. If only the edit distance is desired, it is easy to implement the dynamic programming method in $\mathcal{O}(m)$ space. Hirschberg [6] and Myers and Miller [11] give a recursive divide-and-conquer algorithm for producing an optimal alignment in $\mathcal{O}(n)$ space. We focus instead on the more general backtracking technique that can produce all alignments.

Clearly, the RWF approach poses problems when the library is large: Each processor requires $\mathcal{O}(n)$ space to maximize task granularity. We could divide the edit matrix into k blocks, each of size $m \times n/k$, and evaluate the blocks in order, using all p processors on each block. Boundary data would be carried over to the next block and block memory reused. However, such a strategy results in reduced parallelism, since all processors must synchronize after each block. In addition, the bookkeeping involved in transferring data between blocks detracts from the simplicity of the basic method and incurs non-negligible costs in time and space. Several papers offer strategies for reducing space consumption by constant factors [1,5].

4.2 The Diagonal Wavefront Approach

The diagonal wavefront approach computes d_{ij}'s along positive diagonal axes (sometimes called anti- or counter-diagonals). When a processor is free, it looks along the *current* diagonal and computes any of the available cells. Unlike the RWF method, in which each row must be computed in order, there is no requirement for strict ordering within the current diagonal, because, if diagonals are chosen from left to right, every entry of the current diagonal is computable (Figure 4).

Like row wavefronts, diagonal wavefronts can advance both synchronously (sDWF) and asynchronously (sDWF). In an $m \times n$ matrix, there are $n + m - 1$ diagonals, of which $n - m + 1$ are of length m. The remaining $2(m - 1)$ diagonals consist of two each of lengths $1, 2, ...m - 1$. If p processors are available, a synchronous DWF can compute a complete diagonal of length m in time $\lceil m/p \rceil$; therefore, computation of the entire matrix takes time $2\sum_{i=1}^{m-1} \lceil i/p \rceil + (n - m + 1) \lceil m/p \rceil$.

Asynchronously, the notion of a current diagonal is less useful; a processor p_i will

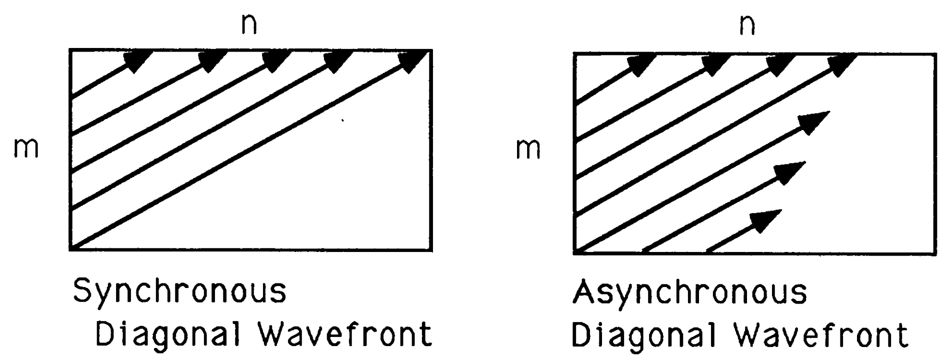
n
m
Synchronous Diagonal Wavefront
n
m
Asynchronous Diagonal Wavefront

Figure 4: *Synchronous*: Computation of diagonal entries is divided among all processors. When all processors are finished, the next diagonal is computed. *Asynchronous*: A processor is assigned an entire diagonal, much like asynchronous RWF. Each processor must check the previous diagonal to determine computability.

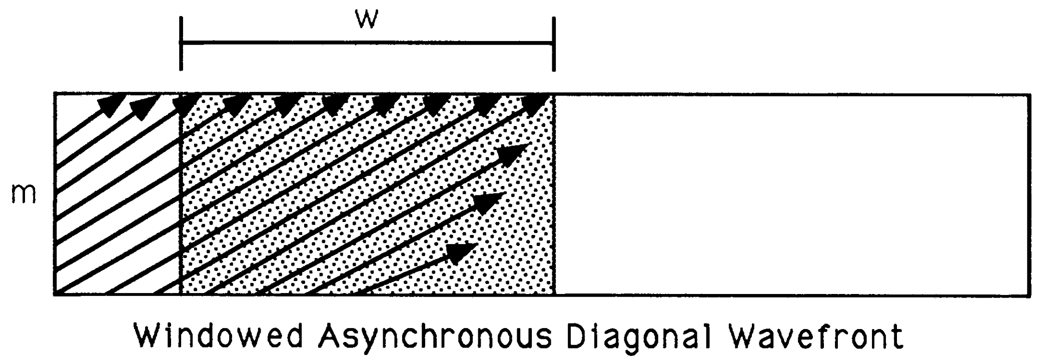

Figure 5: A sliding window of width w is a space-saving optimization. The width is the maximum cost of converting the query sequence to another sequence of length m. The window permits reconstruction of any edit path, without storing the entire edit matrix.

step to the next diagonal, because the entries on the current diagonal have already been distributed to other processors, although their values are not necessarily available. Processor p_i must then check the status of entries in the current diagonal before it can begin to compute entries in the next diagonal. Thus, every processor must check entries in the previous diagonal to ensure computability. Assigning a single processor to each diagonal results in time complexity similar to that of the aRWF approach. In practice, however, the aDWF approach generates many more tasks than aRWF ($n + m - 1$ vs. m); in addition, an aDWF task involves at most m computations, while an aRWF task is always of size n. We expect aRWF to perform better in a parallel environment.

However, the DWF methods can be readily modified to improve memory usage. By examining the cost functions for insertion, deletion and substitution, we can determine the width of an *edit window*, which slides laterally along the edit matrix and is always wide enough to permit backtracking through the edit matrix from row m to row 0 (Figure 5). The width w is simply the maximum cost of converting the query sequence to any other sequence of length m. For unitary cost functions, $w = 2m$, which corresponds to m deletions and m insertions.

For problems in which n is much larger than m, the windowed DWF method can offer substantial space savings, $\mathcal{O}(km^2)$ vs. $\mathcal{O}(mn)$, at the expense of managing the sliding window.

4.3 Discussion

The RWF and DWF approaches differ only in their method of assigning tasks to processors: a row is an RWF task, while a diagonal (or elements of a diagonal in the synchronous version) is a DWF task. The maximum number of d_{ij}'s that can be computed simultaneously in both methods is m, the length of the query sequence. Thus, the upper bound on speedup for both these methods is m. We conclude from Amdahl's Law that the inherently serial fraction **S** of the basic method is $1/m$. Thus, for large m, the basic method is highly parallelizable.

In practice, m must be greater than or equal to the number of processors if the available parallelism is to be fully exploited. We can localize the source of **S** at the start and finish of matrix computation in both approaches. Some processors may spend a good deal of time waiting for a task during evaluation of the first and last p diagonals in the DWF method, and the first and last p rows in the RWF approach. Indeed, in p time steps, p processors perform $p(p+1)/2$ computations, which is only slightly better than half of the p^2 computations possible.

5. Implementations and Results

We have implemented the synchronous and asynchronous versions of the RWF and DWF decompositions, as well as the windowed asynchronous DWF approach. All programs are written in C, using the parallel programming macro extensions developed by Argonne National Laboratory for the Encore Multimax.

For testing these decompositions, we generated query and library sequences of varying lengths. The objective of each implementation is to compute the edit distance, or d_{mn}. All cost functions for insertions, deletions and transformations are unity.

Additionally, we tested the windowed version of the asynchronous DWF methods with real DNA query and library sequences. For this test, the objective is to determine the optimal alignments given unitary cost functions. We searched for the ten best alignments of human cytochrome cl mRNA with sequences in the bovine cytochrome library of GenBank.

Each implementation was run with 1, 2, 4, 8, 10, and 12 processors. Speedup results are shown in Figures 6 and 7. Computation time refers to the evaluation of the entire matrix, excluding sequence loading, memory allocation, and process creation times.

The data indicate that row wavefronts perform consistently well, yielding sublinear, but still substantial, speedups. The windowed aDWF method scales well, too, although its absolute computation time reflects the cost of managing the sliding window.

The results of the cytochrome search and alignment are shown in Figure ??.

6. Summary

We have presented several parallel decompositions of the basic dynamic programming method for execution on a shared-memory multiprocessor. Shared-memory is particularly appropriate for this task, because of the highly localized dependencies and the need for backtracking across substantial portions of the matrix.

The basic method has been implemented on other architectures, including the 1024-element AMT DAP (John Collins, personal communication to D.B.); each processor contains 4K of local memory and must make explicit requests for memory belonging to neighboring processors. The DAP is a SIMD (single-instruction, multiple-data stream) machine, which means each processor performs the same, synchronized operations, but on different data streams.

Carriero and Gelernter [2,3] have parallelized the Needleman-Wunsch algorithm in a fashion similar to our methods, using the Linda model for parallel programming.

Sequence Lengths	p	Sequential Time	sRWF	sDWF	aRWF	aDWF	Windowed aDWF	
m = 22	1	67.87 secs	73.29 secs	72.73 secs	71.83 secs	105.40 secs	120.77 secs	
n = 30,000	2		1.85	1.62	1.85	1.83	1.89	SPEEDUP
	4		3.23	2.66	3.35	3.15	3.39	
	8		6.73	3.32	5.87	5.03	6.61	
	10		6.17	3.15	6.54	5.25	8.32	
	12		8.51	2.21	9.53	5.13	8.94	
m = 118	1	185.85 secs	198.41 secs	198.52 secs	199.12 secs	296.63 secs	336.38 secs	
n = 14,908	2		1.77	1.81	1.93	1.93	1.92	SPEEDUP
	4		3.26	2.89	3.65	3.44	3.62	
	8		7.26	5.10	6.70	6.73	7.56	
	10		8.43	5.14	8.85	7.76	9.03	
	12		10.29	3.21	9.79	8.28	8.55	

Figure 6: The computation time speedups for various problem sizes and processor configurations. *sRWF*: synchronous row wavefront; *sDWF*: synchronous diagonal wavefront; *aRWF*: asynchronous row wavefront; *aDWF*: asynchronous diagonal wavefront.

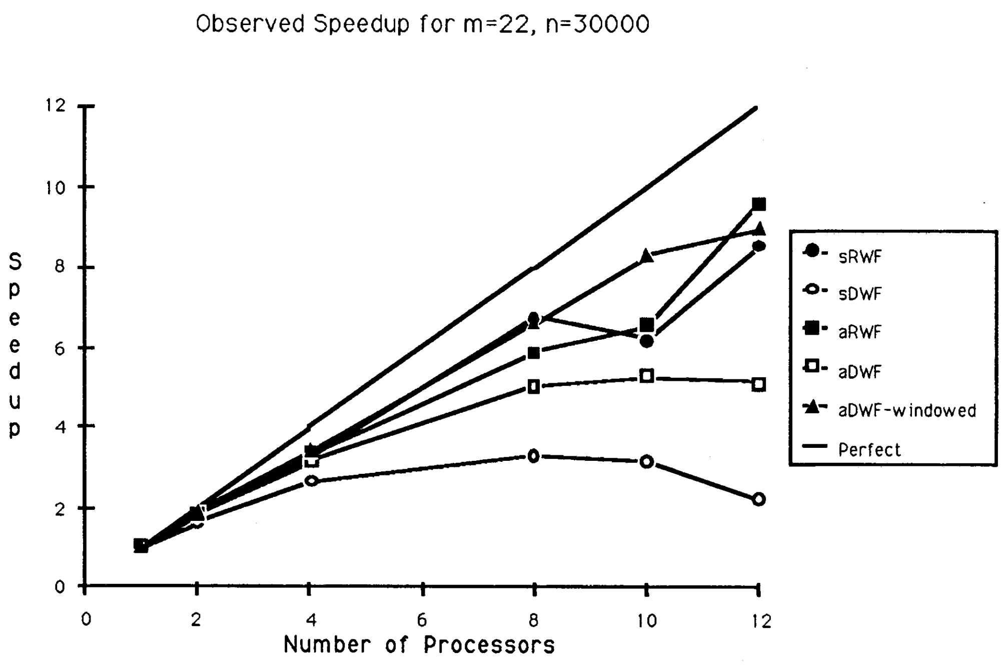

Figure 7: Plot of Speedup versus number of processors for all approaches.

The Linda implementation captures the dataflow aspect of the basic method conceptually, but implementations of the Linda constructs typically degrade performance. Linda is, by design, language- and architecture-independent, and suffers because of this.

Our results indicate that substantial speedup can be achieved with up to m processors using relatively simple and practical parallel modifications to the basic method. By far, the most limiting factor in our implementations is memory; the more we have, the better off we are, but the problem remains of how to handle the increasing size of biological sequence libraries. GenBank currently contains $26,000,000$ nucleotides; a herpes virus query sequence of $177,000$ bases results in an edit matrix on the order of 5×10^9 in size. As we partition and fragment the matrix, we slowly lose the identifiable benefits of parallelism, as we must do extra work to put the pieces back together.

Although it departs from the simplicity of the basic method, Masek and Paterson's optimal sequential algorithm [9] displays great potential for efficient parallelization, because it partitions the matrix into coarse-grained, independent submatrices and minimizes the time costs of rebuilding the whole. We are currently continuing our study of parallel decompositions of sequence comparison algorithms, with a focus on the current optimal algorithms and on massively-parallel machines which offer shared-memory paradigms.

Acknowledgments. We thank Thierry Barsalou for his TEXpertise and Lyn Dupre for her editorial comments. We thank the Stanford Computer Systems Laboratory and Anoop Gupta for the use of the Encore Multimax. Adam Galper is supported by the National Library of Medicine under NIH/NLM Training Grant NIH LM007033 Dr. Brutlag is supported by NLM grant LM04957.

References

[1] S. Altschul and B. Erickson. Optimal sequence alignments using affine gap costs. *Bulletin of Mathematical Biology*, 48:606–616, 1986.

[2] N. Carriero and D. Gelernter. Applications experience with linda. *Proceedings of the ACM Symposium on Parallel Programming*, July 1988.

[3] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACMg*, pages 444–458, April 1989.

[4] C. DeLisi. Computers in molecular biology: Current applications and emerging trends. *Science*, 240:47–52, 1988.

[5] O. Gotoh. Pattern matching of biological sequences with limited storage. *CABIOS*, 3:17–20, 1987.

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

[7] Gad M. Landau and U. Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. *Eighteenth Annual Symposium on Theory of Computing*, pages 220–230, 1986.

[8] D.J. Lipman and W.R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227:1435–1441, 1985.

[9] W.J. Masek and M.S. Paterson. *How to compute string-edit distances quickly in Time Warps, String Edits and Macromolecules.* Addison-Wesley Publishing Company, Reading, MA, 1983.

[10] T.R. Mathies. A fast parallel algorithm to determine edit distance. Technical Report CMU-CS-88-130, Computer Science Department, Carnegie Mellon University, April 1988.

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

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

[13] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85:2444–2448, 1988.

[14] T.F. Smith M.S. Waterman and W.M. Fitch. Comparative biosequence metrics. *Journal of Molecular Evolution*, 18:38–46, 1981.