A MULTITHREADED PARALLEL IMPLEMENTATION OF A DYNAMIC PROGRAMMING ALGORITHM FOR SEQUENCE COMPARISON

W.S. MARTINS, J.B. DEL CUVILLO, F.J. USECHE,
K.B. THEOBALD, G.R. GAO
Department of Electrical and Computer Engineering
University of Delaware, Newark, DE 19716, USA

Abstract

This paper discusses the issues involved in implementing a dynamic programming algorithm for biological sequence comparison on a general-purpose parallel computing platform based on a fine-grain event-driven multithreaded program execution model. Fine-grain multithreading permits efficient parallelism exploitation in this application both by taking advantage of asynchronous point-to-point synchronizations and communication with low overheads and by effectively tolerating latency through the overlapping of computation and communication. We have implemented our scheme on EARTH, a fine-grain event-driven multithreaded execution and architecture model which has been ported to a number of parallel machines with off-the-shelf processors. Our experimental results show that the dynamic programming algorithm can be efficiently implemented on EARTH systems with high performance (e.g., speedup of 90 on 120 nodes), good programmability and reasonable cost.

1 Introduction

Today, one of the most powerful methods for inferring the biological function of a gene (or the protein that it encodes) is by sequence similarity searching on protein and DNA sequence databases. With the development of rapid methods for sequence comparison, discoveries based solely on sequence homology have become routine. A good introduction can be found in a book by Waterman.[1]

Although sequence comparison algorithms based on the dynamic programming method — such as Needleman-Wunsch [2] and Smith-Waterman [3] — provide optimal solutions, they are computationally expensive. Therefore, most current sequence comparison methods used in practice, e.g., BLAST [4] and FASTA,[5] are based on heuristics which are much faster, but do not produce optimal results. Speed is important given the sizes of the sequence databases currently available, but it comes at the price of getting incomplete results.

In this paper, we are interested in studying how to apply the computational power of parallel computers to speed up the process of comparing sequences,

but without having to compromise by missing some optimal results. We look at dynamic programming algorithms for sequence comparison. The first algorithm introduced for finding the optimal alignment between sequences, the Needleman-Wunsch algorithm,[2] used this technique. This algorithm had a great impact on later sequence alignment algorithms, such as the well-known Smith-Waterman method [3] and others.[6,7] Therefore, speeding up dynamic programming algorithms for finding optimal solutions to sequence comparisons is an important problem in computational biology and bioinformatics.

The dynamic programming algorithm works by computing the so-called *similarity matrix*. As will be discussed in detail in Section 2, the computation at each element in this matrix depends on the results of three other elements: its nearest west, northwest and north neighbors in the matrix. Such fine-grain data dependences present serious challenges for efficient parallel execution on current parallel computers. To meet such challenges, we exploit the power of a multithreaded execution and architecture model, such as the *EARTH* (Efficient Architecture for Running THreads) model,[8] where fine-grain parallelism can be efficiently exploited on top of a parallel machine based on off-the-shelf microprocessors. Under the EARTH model, the computation of an element (or a block of elements) of the similarity matrix can be assigned to one thread. The thread scheduling under EARTH is *event-driven*; a thread will become enabled if and only if the events on which it depends have arrived. Therefore, we can map the fine-grain data dependences into such events, and the enabling and execution of the threads in different points are performed in an asynchronous fashion. With similarity matrices above a reasonable threshold, this mapping provides ample thread parallelism to keep the processors usefully busy. Maintaining multiple enabled threads in the same processor also provides the ability to tolerate interprocessor communication and communication latencies, and to sustain high and smooth scalability.

2 Sequence Comparison Using Dynamic Programming

The first algorithm for comparing biological sequences using the dynamic programming technique was proposed by Needleman and Wunsch in 1970.[2] The algorithm consists of two parts: the calculation of the total score indicating the similarity between the two given sequences, and the identification of the alignment(s) that lead to the score. In this paper we will concentrate on the calculation of the score, since this is the most computationally expensive part.

The idea behind using dynamic programming is to build up the solution by using previous solutions for smaller subsequences. The comparison of the two sequences X and Y, using the dynamic programming algorithm, is illus-

trated in Figure 1. This algorithm finds global alignments by comparing entire sequences. The sequences are placed along the left margin (X) and on the top (Y). A *similarity matrix* is initialized with decreasing values $(0, -1, -2, -3, \ldots)$ along the first row and first column to penalize for consecutive gaps (insertions or deletions).

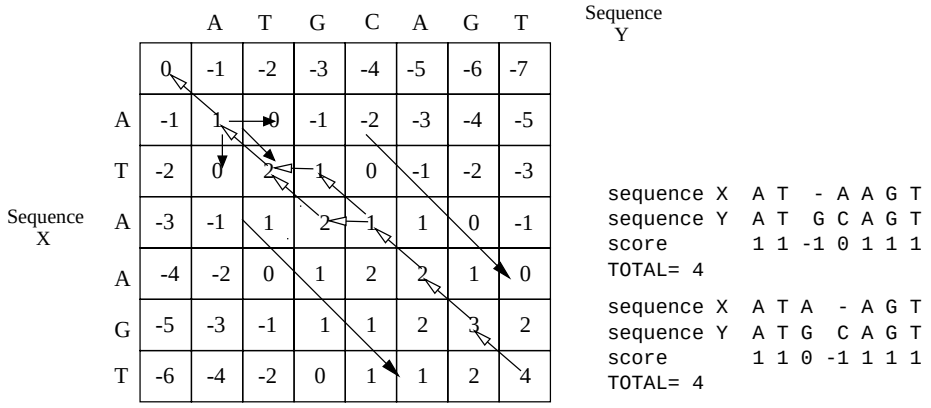


Figure 1: Similarity matrix and global alignments

The other elements of the matrix are calculated by finding the maximum value among the following three values: the left element plus gap penalty, the upper-left element plus the score of substituting the horizontal symbol for the vertical symbol, and the upper element plus the gap penalty. For the general case where $\mathrm{X} = x_1, \ldots, x_i$ and $\mathrm{Y} = y_1, \ldots, y_j$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$, the similarity matrix $SM[n, m]$ is built by applying the following recurrence equation, where gp is the gap penalty and ss is the substitution score:

$$SM[i,j] = max \begin{cases} SM[i, j-1] + gp \\ SM[i-1, j-1] + ss \\ SM[i-1, j] + gp \end{cases}$$

In our example, gp is -1, and ss is 1 if the elements match and 0 otherwise. However, other general values can be used instead.

Following this recurrence equation, the matrix is filled from top left to bottom right with entry $[i, j]$ requiring the entries $[i, j-1]$, $[i-1, j-1]$, and $[i-1, j]$. Notice that $SM[i, j]$ corresponds to the best score of the subsequences $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$. Since global alignment takes into account the entire sequences, the final score will always be found in the bottom right hand corner

of the matrix.[a] In our example, the final score 4 gives us a measure of how similar the two sequences are. Figure 1 shows the similarity matrix and the two possible alignments (arrows going up and left).

3 Parallel Computation — Challenges and Problem Formulation

A parallel version of the sequence comparison algorithm using dynamic programming must handle the data dependences presented by this method, yet it should perform as many operations as possible independently. This may present a serious challenge for efficient parallel execution on current general purpose parallel computers, i.e., MIMD (Multiple Instruction stream, Multiple Data stream) computers.

Given the data dependences presented by the algorithm, the similarity matrix can be filled row by row, column by column, or anti-diagonal by anti-diagonal (i.e., all elements (i, j) for which $i + j$ is a fixed value). The problem with the first two approaches is that most of the elements in a row or column depend on other elements in the same row (column). This means the row (or column) cannot be computed in parallel. On the other hand, the elements in an anti-diagonal depend only on previously calculated anti-diagonals. This means that parallel computation can proceed as a wave front across the similarity matrix, i.e., by computing successive anti-diagonals of the matrix simultaneously, during successive time steps.

Although it exposes parallelism, the anti-diagonal approach faces a few challenges when it comes to an efficient parallel implementation. First, the sizes of the anti-diagonals vary during the computation, which leads to unbalanced work among processors. For example, assume six processors, one per symbol (row) of sequence X, are available to compute the matrix in Figure 1. The computation would start with processor 1 calculating the element [1,1] (assuming rows and columns are numbered 0,1,2 ...). Then, in the next time step, processors 1 and 2 would calculate the elements [1,2] and [2,1] respectively, and so on. This way, processor 6, for instance, would have to wait 6 time steps before starting to work, and by the time we get to time step 8, processor 1 would already be idle. In the worst case, where X and Y are the same length, each processor would only be used half of the time on average.

Another challenge has to do with the number of elements to be computed by each processor in each time step. In the previous example we assumed that each processor would calculate one element of the matrix at a time. However,

[a] When this information is stored during the calculation of the matrix, the second part of the algorithm, identification of the alignment, can be easily computed by following the pointers from the lowest right corner to the upper left corner.

this fine-grain computation would require a very large number of processors if real biological data is to be considered. An additional problem is the high communication overheads for such an implementation, which would require data exchange among all active processors at every time step.

In this paper, we are interested in the following challenging question: Can the dynamic programming algorithm be efficiently implemented on general-purpose parallel computers with high performance and efficiency, good programmability, and reasonable cost? Here we are particularly interested in parallel machines made mainly of commodity off-the-shelf microprocessors and stock hardware. The requirement of "high performance" implies good speedup and scalability, and "good programmability" and "reasonable cost" favor general-purpose parallel computer solutions as opposed to special-purpose hardware or exotic processor technology.

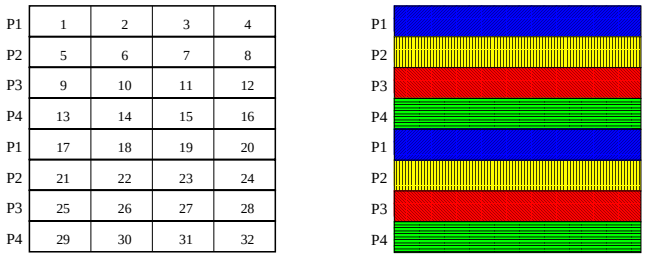
4 Parallel Implementation of the Dynamic Programming Algorithm

The previous section mentioned that computing the anti-diagonal element by element would lead to expensive communication overheads. For each element, the program would compute a single maximum, yet would have to send the result to three processors (though one of these may be the same processor).

One solution to this problem is to divide the similarity matrix into rectangular blocks, as shown in Figure 2(a). In this example, the program would compute block 1 first, followed by 2 and 5, etc. If each block has q rows and r columns, then the computation of a given block requires only the row segment immediately above the block, the column segment to its immediate left, and the element above and to the left — a total of $q + r + 1$ elements. For instance, if each block has 4 rows and 4 columns, then each block has to compute 16 maxima after receiving 9 input values. The communication-to-computation ratio drops from 3:1 to 9:16 — an 81% reduction!

Note that this blocking will decrease the maximum achievable parallelism somewhat, by introducing some sequential dependences in the code. However, given the sizes of the current problems and the parallel machines currently used, this potential loss will not be a limiting factor.

The load-balancing problem can be addressed by putting several rows of blocks (or "strips") on the same processor. Figure 2(b) illustrates this approach when four processors are used. The first and fifth strips are assigned to processor 1, the second and sixth strips are assigned to processor 2 and so on. This helps to keep all processors busy through most of the computation. For example, processor 1 initially works with the first strip, then simultaneously with the first and fifth strip, then finally only with the fifth strip. The



(a) Block Division (b) Processor Distribution

Figure 2: Partition of the similarity matrix

processor utilization rises to 75%.

4.1 The EARTH Multithreaded Architecture — Our Platform

EARTH [8,9] supports a multithreaded program execution model in which a program is viewed as a collection of threads whose execution ordering is determined by data and control dependences explicitly identified in the program. Threads, in turn, are further divided into *fibers* which are non-preemptive and scheduled according to dataflow-like firing rules, i.e., all needed data must be available before it becomes ready for execution. Programs structured using this two-level hierarchy can take advantage of both local synchronization and communication between fibers within the same thread, exploiting data locality. In addition, an effective overlapping of communication and computation is made possible by providing a pool of ready-to-run fibers from which the processor can fetch new work as soon as the current fiber ends and the necessary communication is initiated.

The EARTH model defines a common set of primitive operations required for the management, synchronization and data communication of threads. Each node in an EARTH system consists of an execution unit (EU), a synchronization unit (SU), queues linking the EU and SU, local memory, and an interface to interconnection network. While the EU merely executes fibers, i.e., does the computation, the SU is responsible for scheduling and synchronizing threads, handling remote accesses and performing dynamic load balancing.

Although designed to deal with multiple threads per node, the EARTH model does not require any support for rapid context switching (since fibers are non-preemptive) and is well-suited to running on off-the-shelf processors. EARTH systems have been implemented on a number of platforms: MANNA and PowerMANNA, IBM SP2, Sun SMP cluster and Beowulf. EARTH pro-

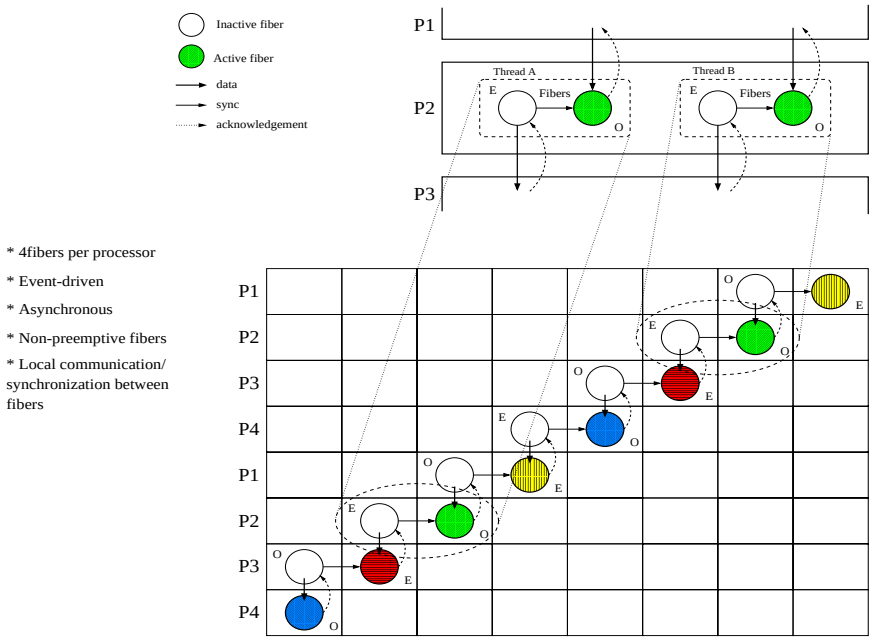


Figure 3: Computation of the similarity matrix on EARTH

grams are written using the programming language Threaded-C.[8,9] This is an extension of the ANSI-C programming language which, by incorporating EARTH operations, allows the user to indicate parallelism explicitly.

4.2 Our Multithreaded Implementation

Our multithreaded implementation follows the description given at the beginning of this section. Generally speaking, it assigns the computation of each strip to a thread, having 2 independent threads per node. However, in order to better overlap computation and communication, blocks on a strip are actually calculated by two fibers within a thread. These fibers are repeatedly instantiated to compute one block at a time, and only one of the two fibers of each thread can be active at a particular time.

The decision of having two alternating fibers within each thread was based on the following reasoning. It would be a waste of resources if we had one separate fiber for each block, in each strip, since only one block can be calculated at a time. Having just one fiber for all blocks is also not a good idea because

this fiber would get delayed due to the synchronization signal coming from the fiber immediately below. This signal acknowledges the receipt of data — without it the fiber, re-instantiated, would be allowed to overwrite the previous data. Thus, with just one fiber, computation would not be allowed to proceed until this acknowledgment signal is received. With the addition of an extra fiber we can further overlap computation and communication since one of the fibers can wait for the acknowledgment while the other starts working on the following block. (This double-buffering and acknowledgment scheme is used with other parallel applications on EARTH.[8,10])

A snapshot of the computation of the similarity matrix using our multi-threaded implementation is illustrated in Figure 3. A thread is assigned to each horizontal strip and the actual computation is done by fibers labeled E(ven) and O(dd). The figure shows the computation of the main anti-diagonal of the matrix. The arrows indicate data and synchronization signals. For example, processor 2 sends data (downward arrows) to processor 3 and receives data from processor 1 — i.e., fibers E of strips 2 and 6 send data to fibers E of strips 3 and 7, and fibers O of strips 1 and 5 send data to fibers O of strips 2 and 6. Fibers within a same thread, that is, associated with the same strip, send only a synchronization signal (horizontal arrows) since they share data local to the thread to which they belong. Finally, dotted upward arrows acknowledge the receipt of data so that the fiber receiving this signal can be re-instantiated to calculate another block of the same strip.

During the initialization phase, each thread grabs a piece of the input sequence X. This piece is all a thread needs from sequence X so the whole sequence need not be stored. Moreover, after computing a block, each fiber sends to the fiber beneath a piece of the sequence Y being compared. By doing so, we minimize the initialization delay that occurs when the nodes are reading sequence X from the server. Furthermore, since subsequent pieces of sequence Y can be stored in the same memory area, the demands for space are considerably reduced.

5 Results

The experiments in this study are based on the EARTH implementation for the MANNA parallel machine. Our experiments were run using both a 20-node MANNA and SEMi, an accurate simulator of the MANNA.[8,9] The difference in the clock cycle counts between the simulator and the real MANNA have been measured and were less than 3% for the same Threaded-C code. The simulator allows one to test different configurations for the EARTH system. In this paper we consider only the most conservative of them, one in which each

node contains only a single off-the-shelf microprocessor, and the operations of the EU and SU must be performed therefore on the same processor.

The proposed multithreaded dynamic-programming algorithm for sequence comparison has been implemented under the EARTH experimental platform. The results are reported for sequences ranging from 512 to 10K elements. Figure 4 shows absolute speedups for various problem sizes (sizes list the length of the X and Y sequences). Absolute speedup compares the running time of the parallel code to the best sequential program, so that the results account for multithreading overheads.

The main experimental results include: [b]

- **Speedup:** The multithreaded implementation has achieved impressive speedup: up to 90 on a 120-node platform.
- **Scalability:** A good scalability is obtained from 4 up to 120 nodes.
- **Processor Efficiency:** Very good efficiency has been achieved; on 8 nodes, processors are utilized 99% as much as on one node running sequential code. Even on 120 nodes, processors achieve 75% utilization on the largest problem size.
- **Programmability:** The good performance is achieved using a straightforward partitioning method, without requiring expensive compiler support for optimal partitioning algorithms.
- **Reasonable cost:** This is achieved on an EARTH platform based on off-the-shelf general-purpose microprocessor technology.

6 Related Work

Different parallel implementations of pair-wise sequence comparison algorithms using dynamic programming techniques range from the exploitation of instruction level parallelism in uniprocessor machines [11,12] to SIMD [13,14] (Single Instruction stream, Multiple Data stream) and MIMD implementations.[15,16] Related problems such as sequence alignment and database search have also made extensive use of parallel hardware, from special-purpose VLSI to reconfigurable hardware and programmable co-processor designs.[14,17] Some work has been reported in the field of hardware accelerators [18] and some basic software platforms

[b] The authors are not aware of any published work reporting results, for a parallel MIMD implementation of the dynamic programming algorithm, for the range of input data and number of processors reported in this paper.

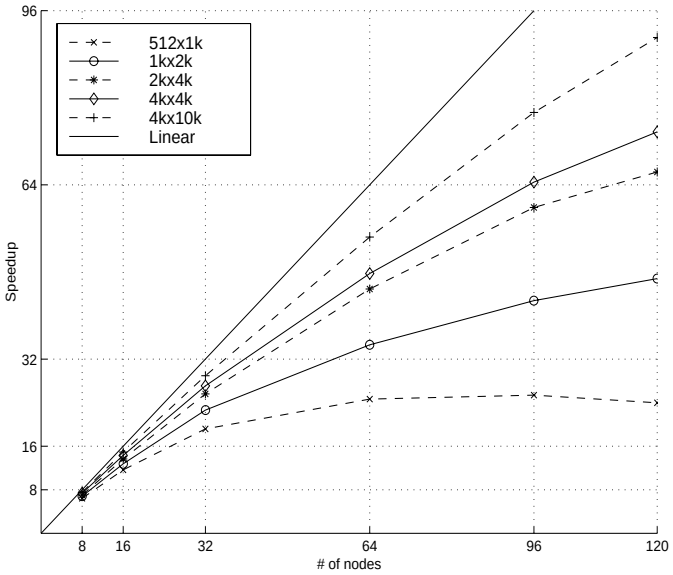


Figure 4: Speed-Up

for parallel computers have been developed which provide a general framework for sequence similarity searching.[19] None of these implementations have made use of fine-grain multithreading, and in this paper we have shown that this greatly improves the performance of the dynamic programming algorithm.

7 Conclusion

The aim of this study is to apply the computational power of parallel computers to speed up the process of comparing sequences, but without having to compromise with incomplete results (e.g., missing some optimal results). We looked at the dynamic programming algorithm and presented a multithreaded parallel implementation under the EARTH model — a fine-grain multithreaded execution and architecture model.

The implementation uses straightforward data partitioning but takes advantage of the special features of the EARTH multithreading model. Fine-grain threads ("fibers") in the code are synchronized strictly according to which data they need, and local data is shared so that data locality can be exploited. The result is that the current implementation of the dynamic programming

on EARTH runs completly asynchronously and is able to effectively overlap communication and computation.

Acknowledgements

We thank GMD First (Berlin) for research collaboration and for providing us with the MANNA machine used in this study. The authors also acknowledge support from the Delaware Biotechnology Institute (DBI). The work on EARTH is partially supported by DARPA, NSA, and NASA through the HTMT project; NSF (grants CISE-9726388, MIPS-9707125, EIA-9972853, and CCR-9808522); and DARPA through the DIVA project.

The authors would like to thank the current and former members of the ACAPS group at McGill University, and the CAPSL group at the University of Delaware, for their insights, ideas, encouragement and help. Also, we would like to thank Jean-François Tomb and his group members at DuPont for introducing us to the problem of sequence comparison, and Scott Tingey and Antoni Rafalski, also from DuPont, for answering our biological questions patiently.

References

1. Michael S. Waterman. *Introduction to Computational Biology: Maps, Sequences, and Genomes (Interdisciplinary Statistics)*. CRC Press, 1995.
2. Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two sequences. *Journal of Molecular Biology*, 48:443–453, 1970.
3. Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.
4. Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.
5. William R. Pearson and David J. Lipman. Improved tools for biological sequence comparison. In *Proceedings of the National Academy of Sciences of the USA*, volume 85, pages 2444–2448, 1988.
6. Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.
7. E.W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4:11–17, 1988.

8. Kevin Bryan Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, McGill University, Montréal, Québec, May 1999.
9. Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.
10. Kevin B. Theobald, Rishi Kumar, Gagan Agrawal, Gerd Heber, Ruppa K. Thulasiram, and Guang R. Gao. Developing a communication intensive application on the EARTH multithreaded architecture. In *Proceedings of the 6th International Euro-Par Conference*, number 1900 in Lecture Notes in Computer Science, pages 625–637, Munich, Germany, August–September 2000. Springer-Verlag.
11. Bowen Alpern, Larry Carter, and Kang Su Gatlin. Microparallellism and high-performance protein matching. In *Proceedings of Supercomputing '95*, San Diego, California, December 1995.
12. A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Computer Applications in the Biosciences*, 13:145–150, 1997.
13. Eric Lander, Jill P. Mesirov, and Washington Taylor IV. Protein sequence comparison on a data parallel computer. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume III, pages 257–263, St. Charles, Illinois, August 1988.
14. Douglas L. Brutlag, Jean-Pierre Dautricourt, Ron Diaz, Jeff Fier, Bruce Moxon, and Richard Stamm. BLAZE: An implementation of the Smith-Waterman sequence comparison algorithm on a massively parallel computer. *Computers and Chemistry*, 17:203–207, 1993.
15. Elizabeth W. Edmiston, Nolan G. Core, Joel H. Saltz, and Roger M. Smith. Parallel processing of biological sequence comparison algorithms. *International Journal of Parallel Programming*, 17(3):259–275, June 1988.
16. Xiaoqiu Huang. A space-efficient parallel sequence comparison algorithm for a message-passing multiprocessor. *International Journal of Parallel Programming*, 18(3):223–229, June 1989.
17. Richard Hughey. Parallel hardware for sequence comparison and alignment. *Computer Applications in the Biosciences*, 12:473–479, 1996.
18. P. Guerdoux-Jamet and D. Lavenier. SAMBA: Hardware accelerator for biological sequence comparison. *Computer Applications in the Biosciences*, 13:609–615, 1997.
19. A. Deshpande, D. Richards, and W. Pearson. A platform for biological sequence comparison of parallel computers. *Computer Applications in the Biosciences*, 7:237–247, 1991.