LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Parallel Characteristics of Sequence Alignment Algorithms

Computation Structures Group Memo 304
October 27, 1989

Arun K. Iyengar

To appear in *Proceedings of Supercomputing '89*,
Reno, Nevada, November 13-17, 1989.

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099. This author was supported in part by a graduate fellowship from the National Science Foundation.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Parallel Characteristics of Sequence Alignment Algorithms

Arun K. Iyengar
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Parallel algorithms for analyzing DNA and protein sequences are becoming increasingly important as sequence data continues to grow. This paper examines the parallel characteristics of four sequence alignment algorithms. The four algorithms presented are the dynamic programming algorithm developed by Needleman, Wunsch, and Sellers (the NWS algorithm), Fickett's algorithm, a parallel algorithm using some of Fickett's ideas, and an algorithm which uses some of Wilbur and Lipman's ideas for constructing alignments which are not always optimal. The NWS algorithm contains the most parallelism but also does more work than any of the other algorithms which we studied. Fickett's algorithm contains the least parallelism. However, a parallel algorithm which requires significantly fewer instructions than the NWS algorithm is obtained by modifying Fickett's algorithm. The algorithms have been implemented for a dataflow computer in the dataflow language Id.

1 Introduction

This paper analyzes the parallel characteristics of four algorithms for aligning DNA and protein sequences. Biological sequence data is accumulating very rapidly. Increasingly powerful computers will be needed for analyzing DNA and proteins as databases expand.

All living things transmit genetic information through DNA. Important structural and functional characteristics can be determined from an organism's DNA. Biological sequence data provides a very powerful tool for analyzing evolutionary relationships. Many biologists want to determine the DNA sequence of the entire human genome. As more information becomes available, it will become possible to determine important characteristics of a human being by DNA sequence analysis.

The four algorithms which we analyzed are the dynamic programming algorithm developed by Needleman, [Needleman 70], Wunsch, and Sellers [Sellers 74], Fickett's algorithm [Fickett 84], a parallel algorithm similar to Fickett's algorithm, and an algorithm which is similar to Wilbur and Lipman's algorithm [Wilbur 83]. A detailed presentation of these algorithms is given in [Iyengar 88].

2 Sequence Alignment

Sequence comparison algorithms are used in several different fields including molecular biology, string editing, speech processing, and codes and error control [Kruskal 83] . However, the algorithms presented in this paper are specifically intended for comparing biological sequences, which include DNA sequences and proteins. From a purely abstract point of view, a DNA sequence is a string defined over an alphabet consisting of four letters. A protein sequence is a string defined over an alphabet consisting of twenty-one letters. These definitions are sufficient to understand the four algorithms. The following presentation assumes no prior knowledge of biology. Of course, the reader with a strong biological background will have much more insight into the motivation behind the algorithms.

An *alignment* between two sequences defined over an alphabet Σ is a matrix consisting of two rows. The upper row contains the *source* sequence S_1 possibly interspersed with null characters. The bottom row contains the *target* sequence which may also be interspersed with null characters. A null character is represented by a "-". A column consisting of two null characters is not allowed. Let

$$x, y \in \Sigma.$$

A column

$$\begin{bmatrix} x \\ - \end{bmatrix}$$

is a *deletion* . A column

$$\begin{bmatrix} - \\ y \end{bmatrix}$$

is an *insertion* . A column

$$\begin{bmatrix} x \\ y \end{bmatrix}$$

is an identity if $x = y$; it is a substitution otherwise. A *gap* of length k is a series of k consecutive insertions or deletions.

DNA sequences are defined over the alphabet

$$\Sigma = \{A, C, G, T\}.$$

For example, one possible alignment betweeen the sequences

$$S_1 = CATGCATA$$

and

$$S_2 = CATTGAAA$$

is A_1 :

$$\begin{bmatrix} \texttt{CAT-GCATA} \\ \texttt{CATTGAA-A} \end{bmatrix}$$

A_1 contains two gaps of length one, one substitution, and six identities. Two sequences are *homologous* if they are very similar and the degree of similarity is much higher than what would be expected by chance.

3 Sequence Alignment Algorithms

3.1 Algorithm 1: The NWS algorithm

Needleman and Wunsch [Needleman 70] were two of the first people to use computers for comparing biological sequences. Their algorithm calculates an alignment which maximizes the similarity between two sequences. The dynamic programming algorithm of Sellers calculates an alignment which minimizes the difference between the two sequences. The two approaches calculate the same alignment if parameters are selected appropriately [Smith 81]. We will henceforth refer to the NWS algorithm.

We can assign a difference score d to each alignment:

$$d = s + \sum_{i=1}^{n} gp(gs_i)$$

where s is the number of substitutions, n is the number of gaps, gs_i is the size of gap i, and gp is a gap penalty function assigning positive values to all gap sizes. We will assume that gap penalties grow linearly with gap sizes. Thus,

$$gp(gs) = gs * gap\_penalty$$

where $gap\_penalty > 1$. An optimal alignment with respect to a difference score is an alignment which possesses the lowest difference score.

The NWS algorithm calculates an optimal alignment between S_1 and S_2 by memoizing optimal alignments between all prefixes of S_1 and S_2. Two matrices may be allocated for storing alignments between prefixes and their difference scores.

$$score\_matrix[a, b]$$

contains the difference score of an optimal alignment between the first a elements of S_1 and the first b elements of S_2.

$$path\_matrix[a, b]$$

contains the index of the previous cell in the alignment. The alignment is constructed by following the indices stored in *path_matrix*.

The 0th row corresponds to alignments possessing gaps at the beginning of S_1. Thus,

$$score\_matrix[0, a] = gp(a).$$

Similarly, the 0th column corresponds to alignments possessing gaps at the beginning of S_2. Therefore,

$$score\_matrix[b, 0] = gp(b).$$

The remainder of the matrix is calculated inductively using the formula:

$$\begin{aligned} score\_matrix[i, j] = min(&score\_matrix[i-1, j] + gap\_penalty, \\ &score\_matrix[i, j-1] + gap\_penalty, \\ &score\_matrix[i-1, j-1] + distance(S_1[i], S_2[j])) \end{aligned}$$

where

$$\begin{aligned} distance(x, y) &= 0 \quad if \quad x = y \\ &= 1 \quad if \quad x \neq y. \end{aligned}$$

The indices of the previous cell are stored in $path\_matrix[i, j]$. After the matrices have been completely determined, an optimal alignment between the entire sequences is constructed by following indices starting at $path\_matrix[|S_1|, |S_2|]$. The time and space complexity are both $O(|S_1| * |S_2|)$.

The data dependencies in the NWS algorithm define diagonal wavefronts across the alignment matrices. Cells lying on the same wavefront may be calculated concurrently once the previous wavefront has been determined (figure 1). An ideal parallel implementation runs in time $O(|S_1| + |S_2|)$ using $min(|S_1|, |S_2|)$ processors [Edmiston 87].

3.2 Algorithm 2: Fickett's algorithm

Fickett's algorithm requires the user to provide an upper bound d_{max} for the difference score of an optimal alignment. The algorithm assumes that the difference score of an optimal alignment does not exceed d_{max}.

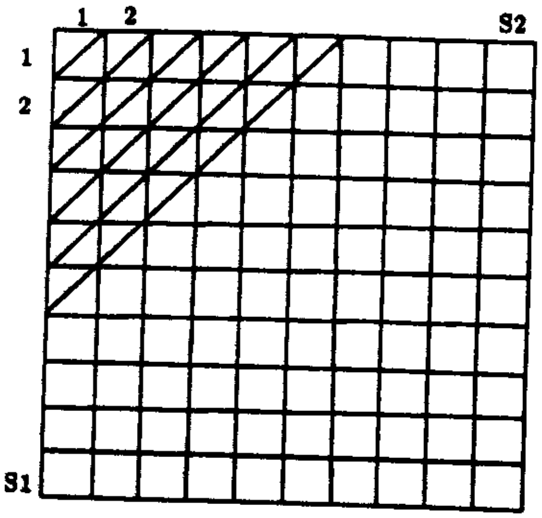

Figure 1: This figure illustrates how alignment matrices are calculated in a parallel implementation of the NWS algorithm. All cells belonging to the same diagonal wavefront may be determined concurrently once the previous wavefront has been calculated.

Under this assumption, an optimal alignment cannot pass through matrix cells which possess difference scores greater than d_{max}.

The key to the algorithm is to keep track of a group of contiguous cells g at the beginning and end of a row which contain difference scores greater than d_{max}. An optimal alignment cannot pass through any cells in g. g consists of cells containing X's in figure 2. Any alignment passing through a cell containing a circle must also pass through g. Therefore, cells containing circles do not have to be calculated.

Fickett's algorithm calculates alignment matrices sequentially from lower to higher rows. If cells 0 through k of row $i-1$ possess difference scores greater than d_{max}, cells 0 through k of row i do not have to be calculated. Similarly, if cells j through $|S_2|$ of row $i-1$ and cell $[i, j]$ of *score_matrix* are greater than d_{max}, cells $j+1$ through $|S_2|$ of row i do not have to be determined.

It is frequently necessary to determine the value of a matrix cell, even though the value of one or two adjacent cells which define its value are unknown. For example, in figure 2, cell $[i, k+1]$ is defined by cells $[i, k]$, $[i-1, k]$, and $[i-1, k+1]$. Cell $[i-1, k+1]$ is the only one of the three which has been calculated. Since an optimal alignment cannot pass through cells $[i, k]$ or $[i-1, k]$, these cells are ignored. Thus,

$$score\_matrix[i, k+1] = score\_matrix[i-1, k+1] + gap\_penalty.$$

If the score of an optimal alignment exceeds d_{max}, Fickett's algorithm terminates without returning an alignment. This is useful for comparing several sequences at a time and identifying pairs which are homologous. Fickett's algorithm will spend less time comparing sequences which are not homologous than the NWS algorithm because it will terminate after a row is found in which the difference scores of all cells exceed d_{max}.

Fickett's algorithm can be improved slightly. If

$$score\_matrix[i, j] > d_{max},$$

Fickett's algorithm concludes that an optimal alignment cannot pass through cell $[i, j]$. *score_matrix*$[i, j]$ is a lower bound on the difference score for an optimal alignment passing through cell $[i, j]$. If a greater lower

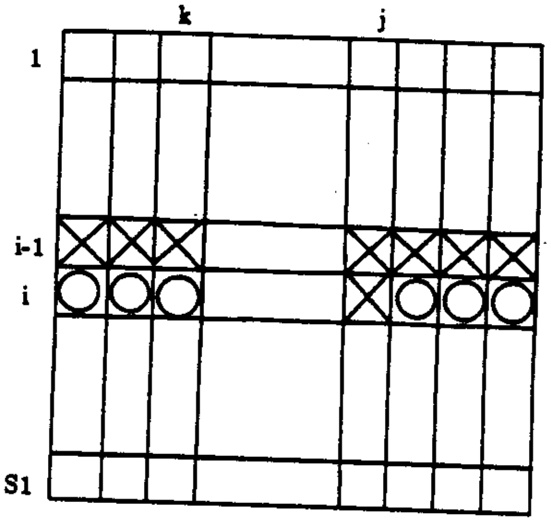

Figure 2: Cells at the beginning and end of rows are pruned by Fickett's algorithm. If the cells containing X's have been eliminated, the cells containing circles can also be eliminated.

bound $l_{i,j}$ can be calculated, more matrix cells may be ignored.

$$a_1 = |S_1| - i$$

elements of S_1 and

$$a_2 = |S_2| - j$$

elements of S_2 have not been added to the alignment at the time cell $[i, j]$ is determined. We may calculate $l_{i,j}$ by assuming that the remaining elements will constitute $min(a_1, a_2)$ identities and $|a_1 - a_2|$ insertions or deletions. Thus,

$$l_{i,j} = score\_matrix[i, j] + |a_1 - a_2| * gap\_penalty.$$

In the improved version of Fickett's algorithm, we conclude that an optimal alignment cannot pass through cell $[i, j]$ if $l_{i,j} > d_{max}$. Our implementation uses the improved version of Fickett's algorithm.

3.3 A parallel algorithm similar to Fickett's algorithm

Fickett's algorithm calculates alignment matrices sequentially. It usually requires fewer total instructions than the NWS algorithm. However, Fickett's algorithm contains far less parallelism.

A parallel algorithm which only calculates part of the alignment matrices is obtained by filling in the matrices along wavefronts instead of rows (figure 1). As each wavefront is calculated, we keep track of contiguous blocks of cells at the edges of the two previous wavefronts which can be ignored because their difference scores are too high. Let wavefront k consist of all matrix cells whose row and column indices sum to k. If all cells along wavefronts $k - 1$ and $k - 2$ with row numbers greater than or equal to j possess difference scores which are two high, we do not have to calculate any matrix cells on wavefront k with row numbers exceeding j. Similarly, if all cells along wavefronts $k - 1$ with row numbers less than i and $k - 2$ with row numbers less than $i - 1$ possess difference scores which are too big, we do not have to calculate any cells on wavefront k with row numbers less than i (figure 3).

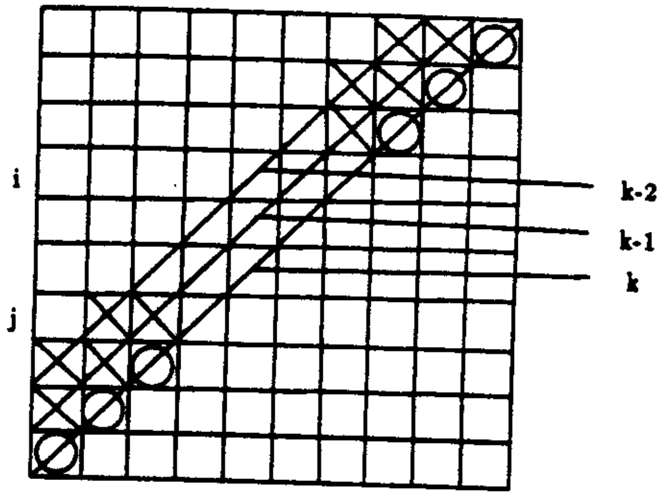

Figure 3: If the cells on wavefronts $k-1$ and $k-2$ containing X's possess difference scores which are too high, the cells on wavefront k containing circles do not have to be calculated. This strategy is used by algorithm 3.

3.4 A fast algorithm which does not always find an optimal alignment

Fast alignment algorithms exist which do not always find optimal alignments. We have implemented an algorithm which locates the matrix region containing the largest number of exact k-tuple matches between S_1 and S_2 and only calculates cells in this vicinity. A k-tuple match between S_1 and S_2 is a sequence of k elements which occurs in both S_1 and S_2.

Cells corresponding to exact k-tuple matches between S_1 and S_2 form diagonal line segments across the alignment matrix (figure 4). The best alignment is likely to pass through the region of the matrix containing the most k-tuple matches. Our algorithm locates a group of contiguous diagonals g containing the most k-tuple matches. It then uses dynamic programming to calculate the best alignment which only passes through diagonals which are no farther than w_2 diagonals from the middle of g. w_2 is supplied by the user. The idea of locating k-tuple matches to quickly calculate alignments which are not always optimal was suggested by Wilbur and Lipman [Wilbur 83].

K-tuple matches between S_1 and S_2 may be located by utilizing a hash table of size $|\Sigma^k|$ (where $|\Sigma|$ is the size of the alphabet over which S_1 and S_2 are defined). Each of the $|\Sigma|^k$ different k-tuples is assigned to a different bucket of the hash table. The index identifying the position of each k-tuple belonging to S_1 is stored in the appropriate hash table bucket. After this has occurred, k-tuples belonging to S_2 are assigned to buckets in the hash table. If a k-tuple in S_2 is assigned to a bucket containing an index into S_1, a match has been found.

4 Results and Discussion

The four algorithms were implemented in the dataflow language Id. The parallelism in Id is implicit. This means that the programmer does not have to insert any explicit parallel constructs in order to write a

	Sequence lengths	Total instructions	% of Algorithm 1	Critical path length	% of Algorithm 1
Algorithm 1					
	15	26,339	100.0%	793	100.0%
	30	96,604	100.0%	1,513	100.0%
	50	259,239	100.0%	2,473	100.0%
	100	1,010,704	100.0%	4,873	100.0%
Algorithm 2					
$d_{max} = 3$	15	13,209	50.15%	2,283	287.9%
$d_{max} = 4$	30	26,284	27.21%	4,518	298.6%
$d_{max} = 5$	50	55,044	21.23%	9,016	364.6%
$d_{max} = 8$	100	133,431	13.20%	21,152	434.1%
Algorithm 2					
$d_{max} = 6$	15	16,248	61.69%	2,681	338.1%
$d_{max} = 8$	30	39,247	40.63%	6,242	412.6%
$d_{max} = 10$	50	77,027	29.71%	11,956	483.5%
$d_{max} = 20$	100	266,166	26.33%	38,996	800.2%
Algorithm 2					
$d_{max} = 9$	15	21,631	82.13%	3,381	426.4%
$d_{max} = 16$	30	62,321	64.51%	9,306	615.1%
$d_{max} = 20$	50	127,802	49.30%	18,746	758.0%
$d_{max} = 40$	100	449,272	44.45%	63,602	1305%
Algorithm 3					
$d_{max} = 3$	15	17,151	65.12%	1,743	219.8%
$d_{max} = 4$	30	34,358	35.57%	3,458	228.6%
$d_{max} = 5$	50	68,033	26.24%	5,978	241.7%
$d_{max} = 8$	100	158,199	15.65%	12,414	254.8%
Algorithm 3					
$d_{max} = 6$	15	20,010	75.97%	1,818	229.3%
$d_{max} = 8$	30	46,584	48.22%	3,734	246.8%
$d_{max} = 10$	50	88,733	34.23%	6,435	260.2%
$d_{max} = 20$	100	283,185	28.02%	15,195	311.8%
Algorithm 3					
$d_{max} = 9$	15	25,100	95.30%	1,925	242.7%
$d_{max} = 16$	30	68,360	70.76%	4,208	278.1%
$d_{max} = 20$	50	136,560	52.68%	7,495	303.1%
$d_{max} = 40$	100	455,632	45.08%	19,024	390.4%
Algorithm 4					
$w_2 = 2$	15	22,150	84.10%	1,510	190.4%
$w_2 = 2$	30	49,578	51.32%	3,049	201.5%
$w_2 = 3$	50	86,915	33.53%	5,006	202.4%
$w_2 = 5$	100	209,968	20.77%	10,083	206.9%
Algorithm 4					
$w_2 = 4$	15	26,115	99.15%	1,570	198.0%
$w_2 = 4$	30	60,032	62.14%	2,994	197.9%
$w_2 = 5$	50	105,138	40.56%	5,027	203.3%
$w_2 = 8$	100	265,269	26.25%	10,114	207.6%

Table 1: Simulation statistics for the four alignment algorithms. The critical path length is the longest chain of operations which must be executed sequentially in any parallel implementation.

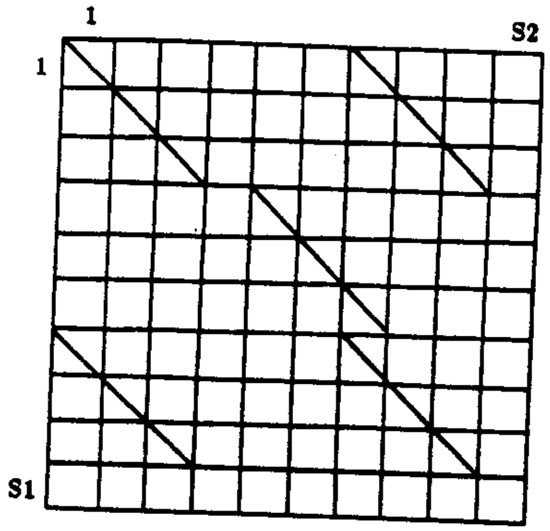

Figure 4: Cells corresponding to k-tuple matches form diagonal line segments across alignment matrices. Algorithm 4 only calculates matrix cells in the vicinity of the diagonals containing the most k-tuple matches.

parallel program. Concurrency is detected automatically by the compiler. This makes Id very easy to use. Programming in Id is no more difficult than programming in a sequential language.

We are currently in the process of building a dataflow computer which executes Id programs. The results presented in this paper were produced by a dataflow simulator. Our simulator is not powerful enough to align two sequences with lengths significantly longer than 100.

Table 1 displays the statistics produced by the four algorithms. The same set of sequences were used to test each algorithm. In every case, algorithms 2, 3, and 4 constructed alignments using fewer instructions than algorithm 1. However, algorithm 1 invariably possessed a shorter critical path length. Furthermore, algorithm 1 is the easiest of the four to implement.

Algorithm 1 is a good choice if enough processors are available because it contains the most parallelism. However, it may be possible to efficiently utilise the processors on a parallel machine by constructing many alignments concurrently. In this situation, each alignment does not have to be constructed by a parallel algorithm. Any algorithm which requires fewer instructions than the NWS algorithm will produce a faster solution, even if a single iteration of the algorithm does not contain much parallelism.

Algorithms 2 and 3 were tested using 3 different values for d_{max}. d_{max} is an upper bound on the difference score which is supplied by the user. The instruction counts for both algorithms increase with d_{max}. Maximum efficiency is obtained by making d_{max} as small as possible. However, both algorithms will terminate without returning an alignment if the difference score of an optimal alignment exceeds d_{max}.

Algorithm 3 contains significantly more parallelism than algorithm 2. This becomes more noticeable when longer sequences are compared with higher values specified for d_{max}. For example, when sequences of length 15 are aligned and $d_{max} = 6$, the ratio of the critical path length of algorithm 3 to that of algorithm 2 is .678. By contrast, this ratio becomes .299 when sequences of length 100 are aligned and $d_{max} = 40$.

Algorithm 4 was tested using two different sets of values for w_2. A trade-off exists between the instruction count and the quality of alignments constructed by the algorithm. If w_2 is small, a small number of matrix cells are calculated. As w_2 increases, more matrix cells are calculated, and the instruction count increases.

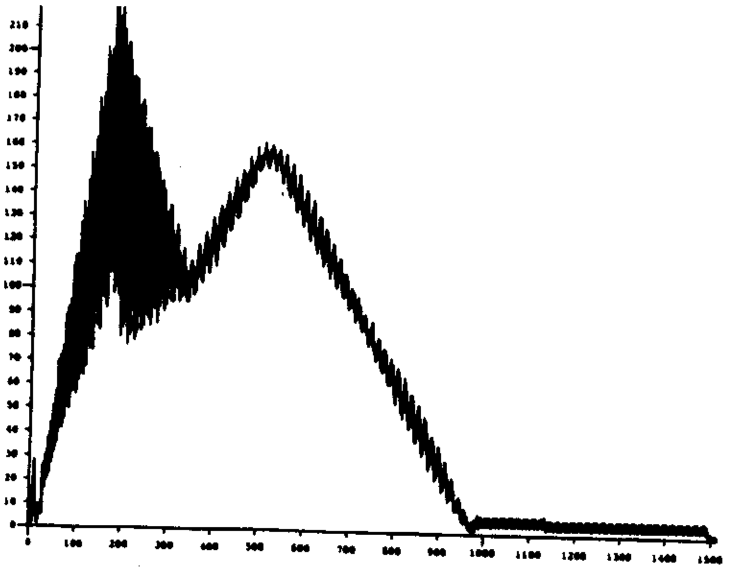

Figure 5: The parallelism profile which results when algorithm 1 aligns two homologous sequences of length 30. The instruction count and critical path length are 96,604 and 1,513 respectively. The sequential tail beginning just before 1000 on the X-axis results from constructing the alignment after the matrices have been determined.

However, the probability of finding an optimal alignment also increases. Unlike the instruction count, the critical path length does not change much with w_2.

The lower instruction counts algorithms 2, 3, and 4 possess relative to algorithm 1 become more apparent when longer sequences are compared. The differences are not significant for sequences of lengths 15 or less. When longer sequences are compared, algorithms 2, 3, and 4 require significantly fewer instructions than algorithm 1. This effect would become more pronounced if we were able to compare sequences significantly longer than 100.

5 Summary and Conclusions

None of the algorithms which we studied is unequivocally better than the others. They all have strengths and weaknesses. The correct algorithm to use clearly depends upon the nature of the application.

The NWS algorithm contains the most parallelism. It is also the easiest algorithm to implement. However, it requires many instructions.

Fickett's algorithm does not contain much parallelism. However, it can align homologous sequences using much fewer instructions than the NWS algorithm. It is a good algorithm to use for searching through a database of sequences and locating pairs which are homologous. Parallelism may be obtained by aligning several pairs of sequences at the same time even though a single iteration is sequential.

The third algorithm which we implemented uses some of Fickett's ideas. It contains significantly more parallelism than Fickett's algorithm. Furthermore, it requires about the same number of instructions. Both algorithms will not return an alignment if the sequences differ by too much.

The fourth algorithm we implemented does not always find an optimal alignment. However, it finds a good alignment in the vast majority of cases. Unlike algorithms 2 and 3, it always returns an alignment.

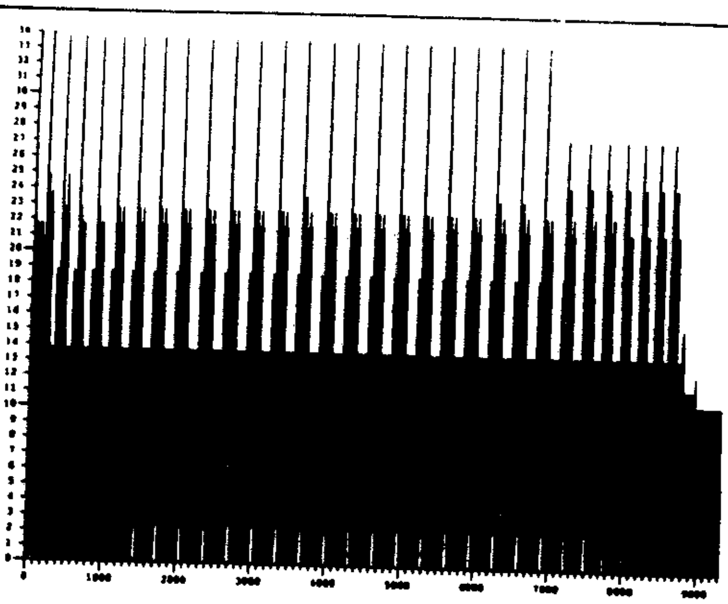

Figure 6: The parallelism profile which results when algorithm 2 aligns two homologous sequences of length 30 and $d_{max} = 16$. The instruction count and critical path length are 62,321 and 9,306 respectively.

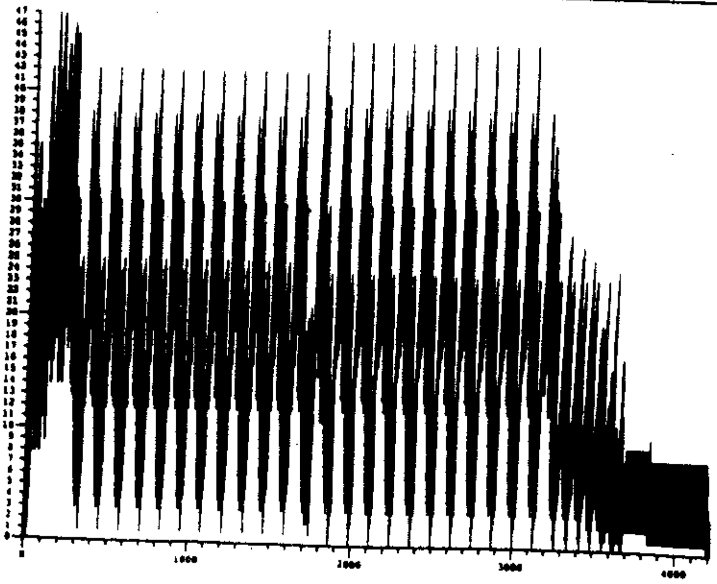

Figure 7: The parallelism profile which results when algorithm 3 aligns two homologous sequences of length 30 and $d_{max} = 16$. The instruction count and critical path length are 68,360 and 4,208 respectively.

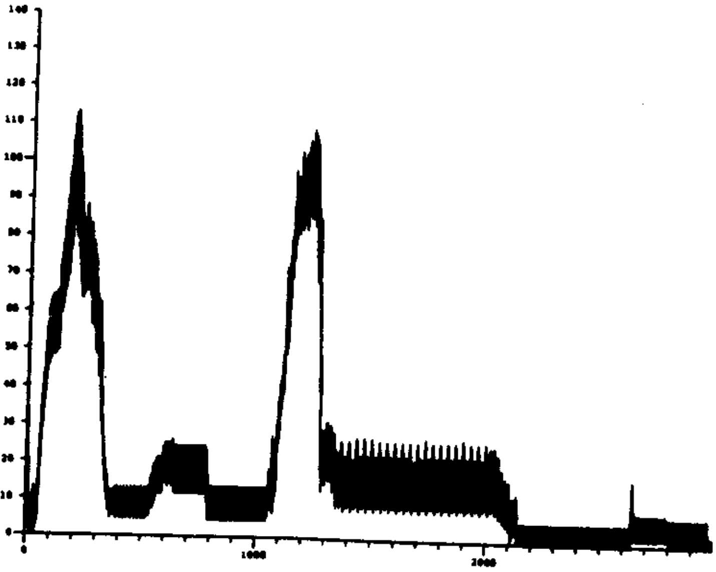

Figure 8: The parallelism profile which results when algorithm 4 aligns two homologous sequences of length 30 and $w_2 = 4$. The instruction count and critical path length are 60,032 and 2,994 respectively. k-tuple matches are located using hash tables from 0 to about 380 time steps. The matrix region containing the most k-tuple matches is located from 380 to about 1080 time steps. Matrix cells surrounding this region are calculated by dynamic programming from 1080 to about 2100 time steps. The alignment is constructed from the matrices from 2100 to 2,994 time steps.

Furthermore, it requires fewer instructions than the NWS algorithm and contains more parallelism than Fickett's algorithm.

All four algorithms were encoded in the dataflow language Id. The parallelism in Id is implicit. This greatly simplifies the programmer's job. Explicitly parallelizing the alignment algorithms is a fairly significant task. The NWS algorithm is the easiest of the four algorithms to parallelize. However, implementing the NWS algorithm efficiently on a SIMD computer such as a Connection Machine requires a lot of work [Lander 88]. The other algorithms are even harder to parallelize explicitly because of their added complexity. An Id programmer does not encounter any difficulties which a sequential programmer would not also face. He is totally insulated from the characteristics of the machine executing his program.

Acknowledgements: The simulation tools for generating the statistics presented in this paper were developed by the Computation Structures Group at MIT. Rishiyur S. Nikhil made many useful suggestions which improved the quality of this research. The author has been supported in part by a graduate fellowship from the National Science Foundation.

References

[Edmiston 87] Edmiston, E., and Wagner, R. A. Parallelization of the Dynamic Programming Algorithm for Comparison of Sequences. In *Proceedings of the 1987 International Conference on Parallel Processing*. pp. 78-80, Penn State Press, Pennsylvania.

[Fickett 84] Fickett, J. W. Fast Optimal Alignment. In *Nucleic Acids Research 12:1*. 1984, pp. 175-179.

[Iyengar 88] Iyengar, A. K. *Parallel DNA Sequence Analysis*. Master's Thesis, Technical Report TR-428, Laboratory for Computer Science, Massachusetts Institute of Technology. Cambridge, MA 02139, 1988.

[Kruskal 83] Kruskal, J. B. An Overview of Sequence Comparison, Time Warps, String Edits, and Macromolecules. In *Siam Review 25:2*. April 1983, pp. 201-237.

[Lander 88] Lander, E., Mesirov, J., and Taylor, W. Protein Sequence Comparison on a Data Parallel Computer. In *Proceedings of the 1988 International Conference on Parallel Processing, Volume 3*. pp. 257-263, Penn State Press, Pennsylvania.

[Needleman 70] Needleman, S. B., and C. D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequences of Two Proteins. In *Journal of Molecular Biology 48*. 1970, pp. 443-453.

[Nikhil 88] Nikhil, R. S. *Id Version 88.1 Reference Manual*. CSG Memo 284, M. I. T. Laboratory for Computer Science, August 29, 1988.

[Sellers 74] Sellers, P. H. On the Theory and Computation of Evolutionary Distances. *SIAM J. Appl. Math. 26:4*. June 1974, pp. 787-793.

[Smith 81] Smith, T. F., et. al. Comparative Biosequence Metrics. *Journal of Molecular Evolution 18*. 1981, pp. 38-46.

[Wilbur 83] Wilbur, W. J., and D. J. Lipman. Rapid Similarity Searches of Nucleic Acid and Protein Data Banks. *Proc. Natl. Acad. Sci. USA 80*. February 1983, pp. 726-730.