ABSTRACT

Comparison of biological (DNA or protein) sequences provides insight into molecular structure, function, and homology, and is increasingly important as the available data bases become larger and more numerous. One method of increasing the speed of the calculations is to perform them in parallel. We present the results of initial investigations using the Intel iPSC hypercube and the Connection Machine for these comparisons. Since these machines have very different architectures, the issues and performance trade-offs discussed have a wide applicability for the parallel processing of biological sequence comparisons.

Parallel Processing of Biological Sequence Comparison Algorithms[1]

Nolan G. Core[2], Elizabeth W. Edmiston[3],

Joel H. Saltz[4], and Roger M. Smith[4]

Research Report YALEU/DCS/RR-630

July 1988

[1]This research was supported in part by the Office of Naval Research under contact No. N00014-86-K-0310 and by NIH Grant T15 LM07056 from the National Library of Medicine.

[2]Yale University School of Medicine

[3]Duke University Department of Computer Science

[4]Yale University Department of Computer Science

1 Introduction

The advent of new DNA sequencing technologies has led to an explosive growth in the quantity of biological sequence information available to researchers which is likely to accelerate in the future [1]. The benefits of this sequence information have already been clearly established, with gains in knowledge of the biological structure and function of many genes and the proteins they encode, resulting in important insights into human biochemistry, physiology, and disease processes. The need to rapidly compare these sequences continues to grow as the accumulated body of information expands. As of August 1987, the Genbank database had approximately 15,000 entries with nearly 15 million nucleotides. Its rapid growth is indicated by the fact that during its first two years of operation it took about nine months to increase by 1 million bases, while a recent 1 million increase required only nine weeks [5]. The average length of a sequence is approximately 1000 nucleotides, and the longest is 172,282.

The algorithms used for this paper do not depend on the actual meaning of the symbols being compared. Thus although simulated sequences were used for the these studies, use of actual Genbank sequences would give similar results. We implicitly assume that only one sequence comparison is performed at a given time; in a Genbank search multiple sequences could be compared, yielding greater efficiencies. The comparison of biological sequences has at least two different, but related goals: (1) to find a measure of the difference between the two sequences, representing what changes would be needed to convert one into the other, and (2) to find the best matched subsequence(s) within a pair of sequences. Two variants of a dynamic programming algorithm can be used to achieve both goals.

Evolving multiprocesser computer architectures achieve their high performance either through the use of a moderate number of fast and complex processors or through the use of a large number of slower and simpler processors. In either situation, it may be infeasible to support extremely fast access from all processors to all of the memory in the multiprocessor. Memory is instead arranged hierarchically so that each processor has relatively fast preferential access to certain portions of memory.

Machines able to logically share memory may employ fast local memories or caches [6]. Other high performance machines use only local memories; in these architectures processors communicate by passing messages

[9,3]. In either of these cases the memory accessible to each processor is organized in a hierarchical manner.

If an algorithm is organized efficiently, most computation can use locally available data. The placement of data in the memory system of the multiprocessor makes a substantial difference in the time required for accessing information. A crucial aspect in the effective design and mapping of algorithms to high performance machines is the need to appropriately assign data and control so that large scale concurrency is possible, and so that the quantity and pattern of multiprocessor interaction is in keeping with the memory and communication characteristics of the machine. Two key factors in achieving good multiprocessor performance are (1) achieving adequate parallelism with an even distribution of load between processors, and (2) partitioning a problem between processors in a way that takes advantage of the hierarchical memory structure of the machine.

In this paper, implementations of two string matching algorithms are examined on two different multiprocessor architectures: the Intel iPSC/1 hypercube and the Thinking Machines Connection Machine (CM-I). The processors in the Intel iPSC/1 and the CM-I both communicate by passing messages: thus for processor A to access information in processor B's local memory, processor B must send processor A a message containing the needed information. The Intel iPSC/1 allows the use of up to 128 relatively fast processors, each of which address their own local memory. The cost of sending messages is quite expensive compared to the cost of performing computations. The CM-I allows use of up to 65,536 relatively slow bit serial processors; communication is fast relative to the speed of the processors. Each architecture embodies a crucial characteristic of high performance architectures: the Intel machine allows us to examine the issues raised by the existence of a memory hierarchy, while the CM-I allows us to explore the use of large scale parallelism in performing these comparisons.

In Section 2 we describe the sequence matching algorithms, while in Sections 3 and 4 we present methods and experimental results for the iPSC/1 and CM-I, respectively. In Section 5 we compare results from the iPSC/1 and the CM-I, and in Section 6 we summarize the paper.

2 Description of Algorithms

The algorithms take as input two sequences $\mathbf{A} = a_1 a_2 ... a_N$ and $\mathbf{B} = b_1 b_2 ... b_M$ where $M \leq N$. The algorithm S-to-S (sequence to sequence) returns the cost of changing **A** into **B** using the Needleman and Wunsch [7] method with appropriate changes to allow a linear cost function for consecutive *indels* (inserts or deletes). The subS-to-subS (subsequence to subsequence) algorithm uses the method of Smith and Waterman [10] as modified by Gotoh [4] to return the I best matches between subsequences of **A** and **B**, where the output for each match consists of the value of that match and the two subsequences.

2.1 Sequence to Sequence Matching

The cost for k *indels* is $w(k) = u \times k + v \geq 0$, $k > 0$. $w(0) = 0$. The comparison of sequence **A** with sequence **B** is performed by creating a cost matrix D as follows:

$$D(i,j) = \min \begin{cases} D(i-1,j-1) + c(b_i, a_j), \\ P(i,j), \\ Q(i,j), \end{cases} \qquad 0 < i \leq M,\ 0 < j \leq N.$$

where

$$P(i,j) = \min \begin{cases} D(i-1,j) + w(1), \\ P(i-1,j) + u. \end{cases}$$

and

$$Q(i,j) = \min \begin{cases} D(i,j-1) + w(1), \\ Q(i,j-1) + u. \end{cases}$$

$$P(0,k) = Q(k,0) = D(k,0) = D(0,k) = w(k),\ \forall k > 0.$$

$c(b_i, a_j)$ represents the similarity between elements a_j and b_i where

$$c(b_i, a_j) = \begin{cases} \geq 0 \text{ if } b_i \neq a_j, \\ \leq 0 \text{ if } b_i = a_j. \end{cases}$$

The value $D(M,N)$ represents the cost of changing **A** to **B**.

2.2 Subsequence to Subsequence Matching

The equations for the subS-to-subS algorithm are very similar to the the S-to-S algorithm and are given in detail in the Appendix.

3 Intel iPSC/1 Hypercube

We define a phase as a computational step during which nodes of the iPSC/1 calculate submatrices in parallel at the same time and a block as a submatrix which is calculated by a node of the iPSC/1 during one phase. If we have P processors numbered 0 through $P-1$, and assign strips of blocks to processors in a wrapped manner so that strip s is assigned to processor $PE_{mod(s,P)}$, then when processor $PE_{mod(s,P)}$ finishes the computation for a step s, it must send data from its lowest row to processor $PE_{mod(s+1,P)}$, to enable that processor to compute step $s+1$. In Figure 1 we depict the matrix that is generated when two sequences of length nine are compared using a three processor machine and blocks of size one, where the numbers depict computational steps that must be performed sequentially. Examination of Figure 1 suggests that the use of larger block sizes may lead to a reduction in communication delays at the expense of a deterioration in the balance of load. This performance tradeoff is quantified in the following subsections.

3.1 Predicted Performance of Various Block Sizes

With very general assumptions, we show below it is optimal to to make the vertical block size as large as possible and to decrease the horizontal block size until the increased communication cost becomes larger than the benefit of decreased idle time. Estimated total time without communications can be expressed as the sum of the time that would be required were the computation evenly distributed between the processors in the absence of any load imbalances plus the time wasted due to load imbalances :

$$= \frac{T_C\ mn}{P} + \frac{T_C\ min(m,n)(P-1)}{P}$$

where T_C = calculation time per block, m = number of horizontal blocks, n = number of vertical blocks, and P = number of processors. With these

definitions the number of phases = m + n - 1. Assuming that m and n are multiples of P, the term for the idle time can be derived by noting that during any phase $j \leq min(m,n) - 1$ when j is not a multiple of P, there are $P - j \ mod \ P$ processors idle. When j is a multiple of P, no processors are idle. Thus the sum of the processor idle time for $j \leq min(m,n) - 1$ is:

$$\frac{T_C min(m,n) \sum_{l=1}^{P} (l-1)}{P * P} = \frac{T_C min(m,n)(P-1)}{2P} \quad .$$

Through similar reasoning, the sum of the processor idle time for the last $min(m,n) - 1$ phases is the same. During the intermediate phases, the load is balanced with $min(m,n)$ blocks assigned to each processor. Thus the total idle time is:

$$\frac{T_C min(m,n)(P-1)}{P} \quad .$$

For simplicity of exposition, we shall continue this discussion in the context of the communication costs incurred in a message passing environment, although a similar cost function applies to shared memory machines in which processors have fast local memories or caches. If there are no communication costs, $T_C = \frac{S}{mn}$, where S = sequential time. Then the total estimated time equals

$$\frac{S}{P} + \frac{S(P-1)}{max(m,n)P} \quad . \tag{1}$$

Thus in the absence of communication costs, all terms involve m and n in a symmetric manner.

First we show that in the presence of communication costs, we should choose $m \geq n$. We calculate the size of the largest message that must be sent between two processors during each phase. We assume that the time required for communication is equal to the sum of the times required each phase to send the largest messages. This tacitly assumes that the system is essentially synchronous, that computation and communication occur in alternating non overlapping periods of time.

The time required for communication can be safely assumed to be an increasing function of message size. For phases 1 through min(m,n)-1, the maximum number of data values sent by any processor is $\lceil p/P \rceil * B_S$, where p = phase number, $B_S = X/m$, and X = horizontal dimension of the matrix.

For phases min(m,n) through m+n-min(m,n), the maximum number of data values sent by any processor is $\lceil min(m,n)/P \rceil * B_S$, and for phases m+n-min(m,n)+1 through m+n-1 a maximum of $\lceil (m+n-p)/P \rceil * B_S$ data values. If B_S were held fixed, the time required for communication would be symmetric in m and n. Since B_S is a decreasing function of m, it is always advantageous from the standpoint of communication cost to choose $m \geq n$. Since (1) is also symmetric in m and n, the minimum total time always occurs when $m \geq n$.

To minimize all terms involving n, we should chose n to be as small as possible, *i.e.* P. For $m \geq n$, (1) has no dependence on n. For any given m, the communication cost does not increase with decreasing n. If dependency graph G_1 has m by n_1 points and dependency graph G_0 has m by n_0 points, with $n_1 \leq n_0$, G_1 can be embedded in G_0. Since the communication cost per block ($B_S = X/m$) is dependent only on m, G_1 need have a commumication requirement no greater than G_0.

When $n = P$, we can combine all costs of calculating a block and communicating the block's data values into one number, T_B, which is the sum of T_C and the cost of communicating X/m data values. With these choices of m and n, total time = T_B $\times$ number of phases and the speedup equals

$$\frac{1}{(P-1)/mP + 1/P + (\alpha + \beta * (X/m)) * (P+m-1)/S}$$

where the communication time per block is $\alpha + \beta$*message size. We describe in Section 3.3 a scheme that uses estimates of T_B to adaptively choose partitions for strings of differing lengths.

3.2 Timings

We have assumed the matrix size in either dimension is a multiple of the number of processors. This will not have a large effect on the results for reasonably sized problems since the matrix to be calculated can always be slightly increased so that each dimension is a multiple of the number of processors, and the analysis will apply without loss of generality. The timings below make use of the observation in Section 3.1 that the vertical block size should be made as large as possible. The total time is the sum of the calculation, communication, and idle time (See Section 3.1). The calculation time is the time spent actually calculating elements of the matrix. The

communication time is the time transferring data between the processors, and the idle time is the time processors are neither calculating or communicating elements of the problem. The calculation time can be measured accurately and will be approximately the same for all processors since their workloads are very similar. The difference between the total time and the calculation time includes the idle time and the communication time. Since total time is defined to be the same for all processors (they all start and end at the same time), one processor can be used to represent the others. The time per block, T_B, is estimated as follows:

$$T_B = \frac{total\ time}{number\ of\ phases}$$

where as previously defined, number of phases = m + n - 1.

Observed times for a 512 x 2048 matrix for the 32 processor hypercube are as follows (The program is written in the C language and all times are in seconds except as indicated):

Block Size $size_x$ $size_y$	8×64	4×64	2×64	4×4	2×2	1×1
Number Phases	95	159	287	639	1279	2559
Total Time	13.9	12.5	13.9	18.1	27.1	54.7

More detailed analysis near the optimal block size is as follows:

Block Size $size_x$ $size_y$	*Number Phases*	*Cal Time*	*Comm Time*	*EstOpt Time*	*Total Time*	T_B *(msec)*
8 × 64	95	8.5	0.8	11.9	13.9	146
4 × 64	159	9.0	1.5	10.0	12.5	79
2 × 64	287	10.1	3.0	9.0	13.9	49

To estimate the communication time for a given problem, we ran the program with the computations removed. In obtaining this communication time estimate, the size and sequence of the messages sent remained the same. The estimated optimal time indicates the computation time that would be obtained in the absence of any multiprocessing overheads including communication delays. The estimated optimal time is defined as the sequential time divided by the estimated speed-up, calculated as:

$$\frac{P * m * n}{m * n + min(m, n) * (P - 1)} .$$

From this data we see that the optimal block size is 4×64. Block sizes near the optimal block size give observed times that are near the optimal time. Since the equivalent sequential time for this problem is 256.8 sec, the execution time of 12.5 sec for the optimal block size represents a speed-up of 20.5.

Observed times for a 1024×2048 matrix revealed that the optimal block size is again 4×64 with an execution time of 23.2 sec and a speed-up of 22.1. The optimal block size for a 2048×2048 matrix is 8×64 with an execution time of 43.0 and a speedup of 23.9. Using the T_B of 79 msec from above, the estimated total times ($= T_B \times$ number of phases) for these two problems are 22.6 and 41.9, and the estimated speed-ups are 22.7 and 24.5, respectively.

3.3 Implementation of the Adaptive Choice of Optimal Block Size

We use regression to determine β_2, β_1, and β_0:

$$T_B = \beta_2 \times size_x \; size_y + \beta_1 \times number\ of\ phases + \beta_0$$

where T_B = time per block, $size_x$ = size of block in horizontal direction, $size_y$ = size of block in vertical direction, β_2 = calculation coefficient, and β_1 = communication coefficient.

A good strategy to determine the initial regression equation is to select a problem size in the middle range of the expected problem sizes and vary the block sizes for that problem. Using the five data points obtained by calculating a 512×2048 matrix with $size_x$ = 1, 2, 4, 8, and 16 and $size_y$ = 64 to determine the initial regression equation (which is adaptively updated with each new observed point), the algorithm gives correct optimal block sizes and predictions of total time that are within one second of that observed for matrices ranging in size from 1024×4096 to 256×256. This method will perform well in similar environments since, as shown in Section 3.2, block sizes which are close to the optimal block size will give total observed times which are close to the optimal total time.

Toward the end of the study, the new Intel iPSC/2 hypercube became available. When the algorithm was run on the iPSC/2 hypercube, the same method of using a 512×2048 matrix with the same five block sizes to de-

termine the initial regression equation resulted in the following comparison of predicted versus observed total times, using the block sizes specified by the algorithm:

Block Size $size_x$ $size_y$	*Array Size* $size_x$ $size_y$	*Predicted Total Time*	*Observed Total Time*
8 × 8	512 × 512	0.7	0.7
4 × 16	512 × 1024	1.3	1.1
4 × 32	512 × 2048	2.2	2.1
4 × 64	1024 × 4096	7.5	7.4

All of the above suggested block sizes were optimal except for the 512 × 512 matrix where a 4 × 8 block was slightly faster (by .07 sec).

3.4 Implementation of the Traceback Procedure

The traceback procedure is needed to demonstrate the alignment of the optimal subsequences of the two sequences. This requires the determination of the source of each matrix element so that one can start with any maximum in the matrix and determine the sequence of calculations that led to that value. One advantage of the hypercube is that each node has approximately 4.5 million bytes of memory, which is more than enough to store the needed information.

By using a binary code, each entry in the traceback matrix incorporates in a single number the path by which the corresponding entry in the similarity matrix was obtained. Since an entry in the traceback matrix has all the information available about the source of any similarity matrix entry, it can be used to reverse the process, *i.e.* perform the traceback. The following table shows the additional time needed for the traceback:

Array Size $size_x$ $size_y$	*Without Trace*	*With Trace*
960 × 960	18.2	23.5
512 × 512	5.6	6.7
256 × 256	1.8	2.1

The additional time for the traceback is primarily due to the need to calculate the additional traceback matrix. The need to determine the maximum in the matrix over all the nodes and the need to follow the trace back

across individual nodes also result in additional communication time. Once the traceback matrix is calculated, the actual traceback is very fast (*e.g.* less than one second for the 960×960 matrix). Since the overall effect is equivalent to increasing the size of the original matrix, the traceback procedure results in a constant percentage increase in total time (here approximately 30 percent). Thus the methods of Section 3.3 to choose optimal block size can still be applied.

4 Connection Machine

The algorithms are parallelized on the CM-I by assigning a processor, PE, to each row of the matrix and proceeding in a wave-like fashion [2], similar to the method described above in Section 3 for the hypercube. To keep track of the I best subS-to-subS matches, a list of the I best matches found so far at PE_i (row i of the matrix) is associated with each processor PE_i. Each time a new entry is computed a new set of starting coordinates is determined which corresponds to the new entry. After each new entry is computed a check is done to see if that value belongs on the best list for that PE. Only one best value in a PE is allowed to start with a given coordinate, since a second match with the same starting coodinate as a better match does not provide a significant amount of new information.

Once the similarity matrix has been computed, each PE contains the I best values which occurred in its row, along with their starting coordinates. A global maximum is obtained to determine the overall best match, which is output. A check is obtained to eliminate all values on the best lists with the same start coordinate. Again a global maximum is obtained to determine the next best match. This continues until the I best matches have been found.

While we can determine where the best match occurs, we are presently unable to determine what the match actually looks like, *i.e.* which elements are matched to which elements. Normally the match is determined by performing a traceback through the similarity matrix. However, since the sequences we are dealing with can be quite large and the memory in a single PE is limited, it is impossible to remember an entire row of the matrix on a PE. We have developed, but not yet implemented, a method of finding the best match which is described in Section 4.2. We estimate that this

method will increase the run time of the algorithm by a factor of 3 to 5, depending on the amount of memory available. The idea is to execute the algorithm once and determine approximately where the best match occurs within the matrix. The algorithm is then run a second time and only that portion of the matrix which was found to be of interest is remembered. A traceback is then performed to determine the best match.

4.1 Timings

We performed various timings on a set of problem sizes ($M \times N$). In order to obtain the parallel times we looped through the entire algorithm (with the exception of $CM\_init()$ which initializes the CM) ten times and divided that time by ten to obtain the execution time. The sequential times were obtained by running the same algorithm on the VAX 8650 which serves as a front-end for the CM-I . Times are given in sec. The programs are written in the CM-I 's PARIS language.

The following table shows times for the S-to-S algorithm on the VAX and in parallel, and for the subS-to-subS algorithm on the VAX when keeping track three matches and in parallel when keeping track of one, three, and ten matches.

Problem Size	*VAX* $S-to-S$ *Time*	*CM* $S-to-S$ *Time*	*VAX subS Time*	*CM One Match*	*Three Matches*	*Ten Matches*
$4K \times 4K$	1865	28.7	8070	51.1	60.5	108.7
$4K \times 8K$	3434	34.1	15639	75.3	90.2	166.3
$4K \times 16K$	6693	57.9	28583	124.4	150.0	270.3
$8K \times 8K$	8494	47.4	21880	101.3	121.7	222.0
$8K \times 16K$	16575	72.0		150.7	181.0	338.9
$8K \times 32K$		116.7		248.7	303.3	542.4

The above data show that parallel processing of the comparison results in times which are 65 to 230 times as fast as the VAX 8650, with this factor increasing as the problem size increases. The following two factors are important in considering this data:

1. The sequential algorithm was executed on the VAX 8650 front-end

to the CM-I, which is much faster than a bit serial processor of the CM-I.

2. Parallelizing an algorithm causes a number of additional instructions to be added to the algorithm since various processors need to be turned on or off for different phases of the computation.

Another interesting point is that the execution time needed for three matches is approximately 1.2 times that needed for a single match, and for ten matches is approximately 2.2 times that for a single match. This implies that the calculation of the matrices in Section 2.2 is the most computationally expensive part of the algorithm. The time needed for additional matches is a relatively small incremental cost for the user. This is a natural and desirable property since in order to find the best subsequence, the algorithm must look at all possible subsequences.

4.1.1 Communication Time

Comparisons of run times with and without the communication (CM_send) step for matrices of size $4K \times (4K, 8K, 16K)$ and $8K \times (8K, 16K, 32K)$ for both the S-to-S and subS-to-subS algorithms resulted in ratios of run times with communication to run times without communication ranging from 0.95 to 1.11. Given the experimental error, this is consistent with a small percentage contribution by the communication step to the total time.

4.2 Performing a Traceback with Limited Memory

As stated earlier, each processor in the CM-I has insufficient memory to retain an entire row of the traceback matrix. The following algorithm can be used to the reduce the amount of the traceback matrix which needs to be stored in order to perform a traceback. This method is designed for the S-to-S problem. A similar method appears to be applicable to the subS-to-subS problem. This method has not yet been implemented on the CM.

1. Assign *special columns* C_k, $0 \leq k \leq H$ (H is dependent on the amount of memory available), such that $C_0 = 0$ and $C_k - C_{k-1} = \lfloor \frac{N}{H} \rfloor$, $1 \leq k \leq H$.

2. As each entry in the cost matrix is calculated, the variable ROW in each processor keeps track of the originating row with respect to the most recent special column for that entry.

3. If the entry's column is itself a special column, then set $R_{i,k} = ROW$ and $ROW = i$.

4. At the end of the cost matrix calculation (PASS 1) the variable ROW in processor M allows a traceback that defines a path linking the special columns C_k.

5. A *local origin* is the intersection of the traceback path with a special column. Each local origin is the upper left corner of a rectangle extending right to the next special column (inclusive) and down to the row of the next special origin (exclusive). These rectangles define areas in which the traceback must occur. Their reduced combined size with respect to the original cost matrix, $\frac{1}{H}$, will in many cases allow the reduced traceback matrix (requiring 4 bits per entry) to be calculated (PASS 2.1). If there is still insufficient memory, each rectangle can be processed in the same way as the original cost matrix (Now this is PASS 2.1.), giving a traceback matrix which is further reduced in size by an additional factor of H (PASS 2.2).

6. Once the reduced traceback matrix is calculated, the traceback can be performed to determine the optimal alignment (PASS 3).

Table 1 shows a schematic representation of the process and Table 2 shows the marked reduction in storage needed. The number of bits needed per PE to remember the $R_{i,k}$ values is $16 \times H$. The number of bits needed per PE to remember the once reduced traceback matrix is $4 \times \frac{N}{H}$. To determine the minimum amount of memory needed per PE these values are minimized. Thus, $16 \times H$ is set equal to $4 \times \frac{N}{H}$ and H is determined. The number of bits needed per PE to remember the twice reduced traceback matrix is $4 \times \frac{N}{H^2}$. Now $16 \times H$ is set equal to $4 \times \frac{N}{H^2}$ and again H is determined.

	C_0 ...	C_1 ...	C_2 ...	C_3 ...	C_4 ... N	
PE_1						
:						
PE_{i_1}						$R_{i_1,1} = 0$
:						
PE_{i_2}						$R_{i_2,2} = i_1$
:						
PE_{i_3}						$R_{i_3,3} = i_2$
:						
PE_{i_4}						$R_{i_4,4} = i_3$
:						
PE_M						$ROW_M = i_4$

Table 1: Reduced Traceback Matrix

	Number of bits		
N	Full Traceback Matrix	One Pass to Reduce the Traceback Matrix	Two Passes to Reduce the Traceback Matrix
$4K$	$16K$	512 bits	163 bits
$8K$	$32K$	728 bits	208 bits
$16K$	$64K$	1024 bits	256 bits
$32K$	$128K$	1456 bits	327 bits
$64K$	$256K$	2048 bits	419 bits
$128K$	$512K$	2896 bits	512 bits
$256K$	$1M$	4096 bits	655 bits
$512K$	$2M$	5793 bits	816 bits
$1M$	$4M$	8192 bits	1024 bits

Table 2: Number of bits needed in each PE to perform a traceback is dependent upon the size of N.

5 Comparison of Connection Machine and the Intel iPSC/1 Hypercube

The following table shows the comparative times for the subS-to-subS algorithm on the CM-I and the Intel iPSC/1 hypercube. Neither algorithm includes the traceback. Since the traceback has not been implemented on the CM-I, we cannot yet make any judgments on the relative performance of usable sequencing algorithms on the two architectures. It does seem to be extremely likely that the extra cost required to perform the traceback will be proportionally much larger on the CM-I than on the iPSC/1.

Array Size		*Connection*	*Intel*
$size_x$	$size_y$	*Machine*	*Hypercube*
1024 ×	1024	13.1	14.1
2048 ×	2048	25.9	43.0
4096 ×	4096	55.4	157.0

The CM-I and the Intel iPSC/1 hypercube are comparable for smaller sequences, but diverge as the size of the sequences gets larger. This is because the time increases only by the increased factor of one dimension for the CM-I (up to the maximum number of processors available), while the time for the hypercube increases by the product of the increased factors of both dimensions. The CM-I would be expected to be slower than the iPSC/1 for smaller problems.

6 Summary

The work in this paper demonstrates methods of implementing biological (DNA or protein) sequence comparison algorithms on the Intel iPSC/1 hypercube and the CM-I, machines which have very different architectures. The CM-I is significantly faster when the sizes of the sequences to be compared are relatively large. However, the small memory for each of the CM-I processors prevents the storage of the necessary information for the traceback procedure. Consequently, the traceback must be performed in more than one pass. The Intel iPSC/1 hypercube, on the other hand, has more than enough storage for the traceback procedure, but is slower for larger

size sequences because it has two orders of magnitude fewer processors than the CM-I. However, the new Intel iPSC/2 hypercube is faster by a factor of 4 to 5, and the new CM-II has more memory. The conceptual framework and analytical tools developed by this work on these machines will be useful for their newer versions, as well as other machines which share some of their architectural characteristics.

7 Appendix: Subsequence to Subsequence Matching

SubS-to-subS uses the method of Smith and Waterman [10] as modified by Gotoh [4] for determining subsequence to subsequence matches. The method maximizes a match instead of minimizing a cost.

$$D'(i,j) = \max \begin{cases} D'(i-1,j-1) + c'(b_i, a_j), \\ P'(i,j), \\ Q'(i,j), \\ 0, \qquad\qquad 0 < i \leq M,\ 0 < j \leq N. \end{cases}$$

where

$$P'(i,j) = \max \begin{cases} D'(i-1,j) + w'(1), \\ P'(i-1,j) + u'. \end{cases}$$

and

$$Q'(i,j) = \max \begin{cases} D'(i,j-1) + w'(1), \\ Q'(i,j-1) + u'. \end{cases}$$

$$P'(0,k) = Q'(k,0) = D'(k,0) = D'(0,k) = 0,\ \forall k > 0.$$

In this case $w'(k) = -w(k),\ \forall k > 0$ and

$$c'(b_i, a_j) = \begin{cases} \leq 0 \text{ if } b_i \neq a_j, \\ \geq 0 \text{ if } b_i = a_j. \end{cases}$$

The best match between subsequences is evaluated at $D'(i_2, j_2)$ where

$$D'(i_2, j_2) \geq D'(i,j),\ 0 \leq i \leq M,\ 0 \leq j \leq N.$$

This corresponds to matching the subsequence $a_{j_1}..a_{j_2}$ to subsequence $b_{i_1}..b_{i_2}$ where $0 < i_1 \leq i_2$ and $0 < j_1 \leq j_2$. (i_1, j_1) is the starting coordinate for this match.

References

[1] "Academy Backs Genome Project." *Science* (1988), **239**, 725-726.

[2] Edmiston, E. W. and Wagner, R. A. "Parallelization of the Dynamic Programming Algorithm for Comparison of Sequences." *Proceedings of the 1987 ICPP*, 78-80.

[3] Gabriel, R. P. "Massively Parallel Computers: The Connection Machine and NON-VON." *Science* (1986), **231**, 975-978.

[4] Gotoh, O. "An Improved Algorithm for Matching Biological Sequences." *Journal of Molecular Biology* (1982),**162**, 705-708.

[5] Hilofsky, H.S. and Burks C. "The Genbank Genetic Sequence Databank" *Nucleic Acids Research* (1988), **16**, 1861-1863.

[6] Kuck, D. J. *et al.* "Parallel Supercomputing Today and the Cedar Approach." *Science* (1986), **231**, 967-974.

[7] Needleman S. B. and Wunsch C. D. "A General Method Applicable to the Search for Siimilarities in the Amino Acid Sequence of Two Proteins." *Journal of Molecular Biology* (1970), **48**, 443-453.

[8] Saltz, J. H. "Automated Problem Scheduling and Reduction of Synchronization Delay Effects." *Technical Report 87-22*, ICASE, July 1987.

[9] Saad, Y. and Schultz, M. H. "Topological Proerties of Hypercubes." *Technical Report RR-389*, Yale University Department of Computer Science, May 1985.

[10] Smith, T. F. and Waterman, M. S. "Identification of Common Molecular Subsequences." *Journal of Molecular Biology* (1981), **147**, 195-196.

Processor 1	1	2	3	4	5	6	7	8	9
Processor 2	2	3	4	5	6	7	8	9	10
Processor 3	3	4	5	6	7	8	9	10	11
Processor 1	4	5	6	7	8	9	10	11	12
Processor 2	5	6	7	8	9	10	11	12	13
Processor 3	6	7	8	9	10	11	12	13	14
Processor 1	7	8	9	10	11	12	13	14	15
Processor 2	8	9	10	11	12	13	14	15	16
Processor 3	9	10	11	12	13	14	15	16	17

Figure 1 Computational Steps for 1 X 1 Blocks in a 9 X 9 Matrix