Abstract: A parallel implementation of an efficient method for comparison of multiple DNA sequences is presented. The method is described in terms of a conceptual tree data structure for the sequences begin compared. The parallel algorithm shows efficient utilization of processors on an Encore Multimax computer in a sample comparison of eleven sequences totaling over 4000 bases. Timing data show the strong influence of computer system details on this parallel program.

Also presented is a graphics program for diplaying multiple sequence comparison output data. The display is capable of representing large volumes of multiple sequence comparison data in a single plot. The program has several additional features that allow closer examination of subsets of sequences. A display of matches from the sample comparison reflects the known structure of these sequences.

Parallel Computation of Multiple Biological Sequence Comparisons

David E. Foulser and Nolan G. Core

Research Report YALEU/DCS/RR-727
July 1989

This research supported by the Office of Naval Research under grant N00014-86-J-1906 and by the National Library of Medicine under NIH Grant T15 LM07056.

Abstract

A parallel implementation of an efficient method for comparison of multiple DNA sequences is presented. The method is described in terms of a conceptual tree data structure for the sequences begin compared. The parallel algorithm shows efficient utilization of processors on an Encore Multimax computer in a sample comparison of eleven sequences totaling over 4000 bases. Timing data show the strong influence of computer system details on this parallel program.

Also presented is a graphics program for diplaying multiple sequence comparison output data. The display is capable of representing large volumes of multiple sequence comparison data in a single plot. The program has several additional features that allow closer examination of subsets of sequences. A display of matches from the sample comparison reflects the known structure of these sequences.

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 [1]. As of Release 58 in December 1988, the Genbank database had approximately 21,000 entries with over 24 million nucleotides [7]. The benefits of this sequence information have already been clearly established, with consequent 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 rapidly to compare these sequences continues to grow as the accumulated body of information expands.

Several varieties of sequence comparison algorithm exist, with the longest common subsequence (see *e.g.*, Needleman and Wunsch [14], Wilbur and Lipman [19], Smith and Waterman [16], and Lipman and Pearson [11]) and suffix-tree methods (see *e.g.*, Karlin *et al.* [9, 10] and Martinez [12, 17]) being the most common. The former have been used for pairwise sequence comparisons for several years. However, the latter seem to be particularly well-suited for multiple sequence comparisons, which are of increasing importance due to the proliferation of sequencing data. The computational task of complex

sequence comparisons seems well suited to parallel computer processing.

This paper presents a parallel computer implementation of a suffix-tree based method for rapid multiple sequence comparisons, as a variant on a method proposed recently by Karlin [9, 10]. We also describe a new graphics post-processor for the display of multiple sequence comparison output.

Section 2 introduces the computational problem formulation in terms of a sequence suffix tree. Section 3 presents the matching algorithms for a sequential computer and gives the implementation details of the computation's initial phase. Section 4 indicates the parallel formulation of the computation and details the synchronization steps to coordinate parallel processing. Section 5 describes the biological sequence data used in our computations. Section 6 gives an overview of the graphics postprocessor. Experimental results are presented in section 7. Conclusions are stated in section 8.

2 The Suffix and Match Trees

The comparison method we use finds local sequence similarities based on exact matches of a fixed minimum length in one or more sequences. The rationale behind this approach is that long regions of exact similarity are highly significant (in a probabilistic theoretical sense) and thus may be worth investigating for biological function. A prescribed form of inexact matching is then permitted to extend the original exact matches in order to allow for biological variability.

The exact matches within a single sequence can be represented by the projection of the sequence onto a tree structure of all possible substrings, where a substring is a contiguous subsequence of letters from a larger sequence. The sequence comparison algorithm can then be viewed in terms of the tree structure, which we now develop.

The root node $\hat{T}_\phi$ of the infinite suffix tree $\hat{T}$ represents the null or empty substring ϕ. Node $\hat{T}_\phi$ has four subnodes, labelled $\hat{T}_{\mathrm{A}}$, $\hat{T}_{\mathrm{C}}$, $\hat{T}_{\mathrm{G}}$, and $\hat{T}_{\mathrm{T}}$, representing the four single-nucleotide substrings. From these four nodes, the infinite tree is recursively constructed by concatenating letters A, C, G, and T to each node. For example, $\hat{T}_{\mathrm{ACG}}$ has subnodes $\hat{T}_{\mathrm{ACGA}}$, $\hat{T}_{\mathrm{ACGC}}$, $\hat{T}_{\mathrm{ACGG}}$, and $\hat{T}_{\mathrm{ACGT}}$. The *level* of a node in the tree is the length of its associated substring. For example, $\hat{T}_{\mathrm{ACGC}}$ is a node at level four. The general suffix tree may be constructed for a sequence drawn from an arbitrary alphabet.

Having established the form of $\hat{T}$, it is straightforward to map a given sequence Σ onto a finite subtree T of $\hat{T}$, by identifying a node T_S with all sequence locations in Σ at which the substring S begins. (We shall use Σ to denote a sequence or concatenation of sequences, the letter S to denote a substring, and the characters x and y to denote other sequence letters or substrings.) Thus the node T_S represents all occurrences of the exact match S within Σ. T is called a suffix tree because each node represents all sequence suffixes beginning with a given substring. The subnodes T_{SA}, T_{SC}, T_{SG}, and T_{ST} represent the occurrences of extensions of the match S in Σ. The union of sequence locations represented by T_{SA}, T_{SC}, T_{SG}, and T_{ST} is precisely the set of locations in T_S. The refinement of T_S into its subnodes is the basis of the algorithm described below.

After restricting the infinite $\hat{T}$ to the finite tree T of nodes determined by Σ, T can be pruned slightly to its exact match form. For our present purposes, the only significant nodes of T are those which represent a match of two or more substring occurrences. We can prune away all tree nodes representing a single substring occurrence, that is, all singleton branches, to form the *exact match tree* T. For example, the sequence AACGATCGACAA has the match tree T with nodes $T_\phi = \{1,2,3,4,5,6,7,8,9,10,11,12\}$, $T_A = \{1,2,5,9,11,12\}$, $T_C = \{3,7,10\}$, $T_G = \{4,8\}$, $T_{AA} = \{1,11\}$, $T_{AC} = \{2,9\}$, $T_{CG} = \{3,7\}$, $T_{GA} = \{4,8\}$, and $T_{CGA} = \{3,7\}$. Note that node T_T, among others, is absent from the tree in this example because it contains only a single substring occurrence.

The match tree has an obvious generalization to multiple sequences, in which the sequences are individually mapped onto $\hat{T}$ and the match tree T is then formed by pruning singleton branches. Nodes of T then correspond to substrings matching within a single sequence and/or among multiple sequences. There are many known sequence comparison algorithms that work, implicitly or explicitly, with T or a related data structure, including those of McCreight [13], Weiner [18], Karlin *et al.* [9, 10], Martinez [12, 17] and Blumer *et al.* [3].

As an aside, we note that T can be further refined by various requirements on its nodes. A few such requirements include: a minimum number of matching instances greater than two, the representation of a minimum number of sequences, the presence of a substring from a specified target sequence, and minimum or maximum length requirements on the substrings.

The exact matching algorithm computes the information in T making use of two efficiencies. First, an internal node of T with only a single subnode represents an incompletely extended exact match and need not be displayed. In our example, $T_{\mathrm{CG}} = \{3, 7\}$ has the sole subnode $T_{\mathrm{CGA}} = \{3, 7\}$. Thus the information in T_{CG} is subsumed by its descendant and need not be displayed. Such a node represents a substring that is incompletely extended to the right.

The second efficiency recognizes matches that are incompletely extended to the left. Any node T_{S} that has a single unique leftward extension for all its instances (*i.e.,* only one of AS, CS, GS, TS exists for all occurrences of S) is redundant and need not be displayed.

By suppressing the display of these two types of node correspondences, the algorithm effectively computes only maximal length sequence matches for later display. In the previous example, only A, C, AA, AC, and CGA would be displayed, with G, CG, and GA omitted as non-maximal matches. Note the differences between G and A: A is displayed because it has six occurrences, while no two-letter extension xA or Ax has six occurrences; but all extensions of G are GA.

3 Comparison Algorithms

Many methods are used to compare biological sequences (for a good overview of longest common subsequence based comparison algorithms see Sankoff and Kruskal [15]). An efficient algorithm for the determination of all repeats greater than a given length k has recently been described by Karlin [10]. That algorithm determines both exact and inexact repeats, where an exact repeat is an identically shared sequence of nucleotides of length at least k, and an inexact repeat is a grouping of two or more exact repeats separated by error blocks, each of length at most e nucleotides. Karlin's algorithm first determines the locations of all exact repeats of length k and then extends them to their length of maximum identity. The inexact repeats are then determined by finding neighboring exact matches separated by error blocks of at most e nucleotides. Our matching program follows the above approach, with an emphasis on parallel computation.

We now present our exact matching algorithm, using the match tree notation of the previous section. Let Σ be a sequence or concatenation of sequences, with total length N.

Exact Method

1. Initialize $T_\phi = \{1, 2, \ldots, N\}$, the set of all locations in Σ. Mark T_ϕ as active.

2. While any active nodes remain, execute steps 2a — 2f.

   (a) Select an active node $T_S = \{l_1, l_2, \ldots, l_n\}$, where the length $|S| = m$. Mark T_S as inactive.

   (b) Create sets for T_{SA}, T_{SC}, T_{SG}, and T_{ST} by examining letters at locations $l_1 + m$, $l_2 + m$, ..., $l_n + m$ in the concatenated sequences. Discard any substrings Sx that cross sequence boundaries within Σ (*i.e.,* for which $l_i + m - 1$ is in one sequence and $l_i + m$ is in another).

   (c) If every element of a set T_{Sx} extends to the left by an identical letter y to form ySx, discard the non-maximal node T_{Sx}.

   (d) If a set T_{Sx} has m elements as well, discard the non-maximal node T_S.

   (e) Discard nodes with sets of zero or one element.

   (f) Mark undiscarded nodes as active.

3. No active nodes. Display the undiscarded and inactive nodes that match the display criteria.

Although the set of exact matches is rapidly computed by this approach, it may not suffice in the analysis of actual sequences. Biological sequence similarities often embody a degree of inexact matching, which calls for a measure of flexibility in the method. We address the drawback of considering only exact repeats by extending the method to compute a certain class of inexact matches, namely those composed of exactly matching segments of a minimum length k separated by non-matching regions of no more than e bases between them.

The parameters (k, e) describe a family of sequence comparison schemes. When $k = 1$ and $e = 0$, this is simply the exact match method presented above. For $k = 1$ and $e = N$, it is the longest common subsequence calculation commonly implemented in dynamic programming approaches to sequence comparison. We employ small values of k and e, with $k = O(\log N)$

and e a constant. Our inexact matching method could thus be viewed as a local refinement of a longest common subsequence approach. In this regard it is similar to earlier methods of Wilbur and Lipman [19] and Martinez [12, 17].

Given that the classes of matches computed by this method are related to longest common subsequence calculations under certain restrictions, one can view this method as a replacement for such algorithms. In many cases, particularly for small e, this method is likely to be more efficient than straightforward dynamic programming approaches.

The inexact matching method allows a more flexible creation of subnodes in the tree structure. For a given node T_S, it allows additional subnodes T_{S+x}, where the '+' indicates an error block of length 0, 1, ..., e in each instantiation of the match. At least one instance of the inexact match S+x must have a non-zero length error block '+'. Superfluous error blocks are not allowed. For instance, A+A could be represented by AA, ATA, or ACGTA, among others, but not by AAA. The augmented, inexact match tree contains nodes of the inexact match type, as well as all the nodes of the exact match tree.

With the addition of inexact matchings, a bulk of the program necessary to implement our algorithm becomes devoted to removing duplicate matches in order to reduce processing time and eliminate redundant information in computation and display. Our criterion in constructing inexact matches is to form a minimal set of inexact matches, each of which is composed of maximally extended exact repeats separated by error blocks. The elimination of redundant matches implements this matching criterion from the elements of the pruned suffix tree.

The redundant matches have two basic forms. They include matches that could be extended to the left or right, by the exact or inexact methods, to give another match in the output set. Additionally, no inexact match is allowed where an exact match would suffice (thus no identical error blocks are allowed across all match instances).

Consider, as an example of this latter restriction, an inexact match of three sequences that extends to an inexact match of length 11 on two out of the three. But on examining the two, we see that the match is actually an exact matching pair, which is necessarily computed elsewhere in the tree. The present duplicate copy must be rejected. The table below shows AAA+AAA as such a three-way inexact match, where the two-way match

AAA+AAA+AAA is a redundant "inexact" match:

$$\begin{array}{c|c|c|c|c} AAA & C & AAA & G & AAA \\ AAA & C & AAA & G & AAA \\ AAA & T & AAA & T & xxx \end{array}$$

This is an example of a refinement of a distinct inexact match that yields a duplicate exact match.

The other type of duplication is the incompletely extended match. Examples of this class were already treated in steps 2c and 2d of the exact matching method. Analogously, certain matches extending inexactly to the left and right are redundant. (However, note that a match whose inexact leftward (rightward) extension can itself be extended exactly to the right (left) does not automatically qualify as a redundant match.) Steps must be taken in the inexact method to eliminate these duplicates.

The inexact matching algorithm can be described as follows. Note the slight modification of the exact method that fully extends exact matches before branching out to subnodes. Again, Σ is a sequence or concatenation of sequences of length N.

Inexact Method

1. Initialize $T_\phi = \{1, 2, \ldots, N\}$, the set of all locations in Σ. Mark T_ϕ as active.

2. While any active nodes remain, execute steps 2a — 2k.

   (a) Select an active node $T_S = \{l_1, l_2, \ldots, l_n\}$, where the length of instance l_i is m_i. Mark T_S as inactive.

   (b) Compute the maximal length $p \geq 0$ by which all elements of T_S may be extended and yet remain in a single set T_{Sx}. That is, extend S to its maximal exact matching length from the single set T_S. Extend T_S to T_{Sx}, and write $p = |x|$.

   (c) Create sets T_{Sxy} for T_{SxA}, T_{SxC}, T_{SxG}, and T_{SxT} by examining letters at locations $l_i + m_i + p$ in the concatenated sequences, for $1 \leq i \leq n$. Discard any substrings Sxy that cross sequence boundaries, as in the exact method.

(d) Create sets for T_{Sx+z}, where z is one of the 4^k words of length k, and where '+' represents an error block of length 0, 1, ..., e.

(e) If every element of a set T_{Sxy} extends to the left by an identical letter q to form qSxy, discard the non-maximal node T_{Sxy}.

(f) If every element of a set T_{Sx+z} extends to the left by an identical letter q to form qSx+z, discard the non-maximal node T_{Sx+z}.

(g) Discard nodes with sets of zero or one element.

(h) If every element of a set T_{Sx+z} contains a corresponding superfluous error block, discard the node T_{Sx+z}.

(i) If every element of a set T_{Sxy} or T_{Sx+z} extends to the left by an error block followed to the left by an exact match of length k or greater, and that leftward inexact match cannot be extended to the right, discard the node T_{Sxy} or T_{Sx+z}.

(j) If any remaining node T_{Sxy} or T_{Sx+z} has the same number of set members as T_S, discard T_S. Under this condition T_S extends completely by the inexact matching method and T_S is thus a non-maximal internal node that need not be displayed.

(k) Mark remaining undiscarded subnodes as active.

3. No active nodes. Display inactive nodes matching display criteria.

The tree-based method can be modified to proceed directly to nodes at level k, by precomputing all substrings of length k in Σ. In the sequential algorithm this is done to avoid unnecessary work on insignificantly short repeats, while the parallel algorithm performs this step to quickly reach a level with appreciable parallelism. It is also true that $O(kN)$ work is needed on average to extend the tree to level $k = O(\log N)$ by creation of subnodes, whereas the modified approach requires only $O(N)$ work.

We proceed directly to level k by forming an array $W(1 : N + 1 - k)$ as the numerical representation of overlapping words of length k in Σ. We let A = 0, C = 1, G = 2, and T = 3. Denote by Σ_j the j^{th} letter of Σ. Then $W_j = \sum_{i=0}^{k-1} 4^i \Sigma_{j+i}$. For example, $W_1 = 4^0\Sigma_1 + 4^1\Sigma_2 + \cdots + 4^{k-1}\Sigma_k$. Finally, k-words crossing a sequence boundary are set to a prescribed value (*e.g.*, -1) that is not in the range $[0, 4^k - 1]$ of valid k-word values.

The array W of k-word locations is then assembled into tree node sets. Our method employs a table P of 4^k pointer entries, which reference linked lists representing the node sets of T. We indicate a node set T_S by refering to T_{W_l}, where W_l is the numerical representation of S. The essence of both methods is presented below.

1. **For** $j = 0$ **to** $4^k - 1$
   $P_j = 0$
   endfor

2. **For** $l = 1$ **to** N
   if $W_l \in [0, 4^k - 1]$ **then**
   $P_{W_l} = P_{W_l} + 1$
   $T_{W_l}(P_{W_l}) = l$ (essentially $T_{W_l} = T_{W_l} \cup l$)
   endif
   endfor

Step 1 initializes the counters in P to zero. Step 2 uses the counters to augment the nodes T_j by the locations of words j in Σ. At the end of step 2, each node T_{W_l} has its correct set of substring locations for the word W_l in locations $T_{W_l}(1)$, $T_{W_l}(2)$, ..., $T_{W_l}(P_{W_l})$. Note that T_{W_l} may be implemented as a linked list in this computation. Also, P_j contains the size of the set T_j after step 2.

Our method of implementing this approach allocates the table P as the full array of 4^k words. However, for even modest values of k, this may require a very large memory allocation for P, perhaps much larger than the size of Σ. In such a case, one should implement P as a hash table where unstored elements are understood to be zero. In this case, at most $N + 1 - k$ elements are filled since that is the number of k-letter words in Σ, and in general many fewer than N will be needed, as there will be a substantial number of k-word repeats.

Given the initial node sets of T, the full exact match computation may be started. The computation is organized by keeping a pool of active nodes to be processed. When a CPU becomes free, it executes step 2 of the algorithm, selecting an active node and performing the extensions, marking the initial

node as inactive and eventually placing several subsidiary active nodes back in the pool. We use a queue data structure to implement the pool of work, with the consequence that nodes are processed in a first in, first out fashion. The initial queue is composed of the tree nodes at level k, which has at most 4^k entries. Additional queue nodes are calculated by the exact and inexact matching methods.

4 Parallel Computing Considerations

Our parallel implementation of the inexact matching algorithm was written in C and run on an Encore Multimax [4] 320 with APC cards, using up to 17 processors. The Encore computer is a shared-memory multiprocessor in which the individual CPUs have local cache memories and are linked by a shared system bus to a larger, shared memory. Actions of the multiple CPUs can be coordinated through appropriate operations on the shared memory, or the processors can function as independent virtual-memory computers. The present system has 64 Mbytes of shared system memory and a peak system bus bandwidth of 100 Mbytes/sec. Each APC board holds a pair of CPUs and a local 64 Kbyte cache memory. The Multimax can contain up to 20 CPUs, each of which is rated at 2 MIPS.

The opportunities for parallel computation on T are inherent in its tree structure. T is, in effect, a dependency graph, in that computation for a node T_{Sx} depends only on completion of the work for the parent node T_{S}. Our parallel implementation strategy is to precompute a portion of T in sequential mode, then to switch over to parallel mode in computing the remainder of T. The initial portion computes level k of the tree. Various tactics can then be employed to divide up the remaining work so as to uniformly utilize multiple processors without incurring undue overhead costs. We shall present two approaches.

To implement the parallel processing of the algorithm, we set up a queue data structure of tree nodes, which the processors use to allocate their work. Each queue entry is a tree node, represented as a pointer to a linked list of identical exact matches. The construction of the original queue is discussed above in section 3, where its elements are labelled T_{W_i}. The top queue entry is the next active element to be examined, and newly created nodes are added to the bottom of the queue.

Both methods use shared memory locks to synchronize parallel access to the shared queue data structure. The synchronization is necessary to avoid anomalous conditions in which two processors might operate on the same node in the tree due to essentially simultaneous access to the queue.

Synchronization requires two counters in shared memory that point to the top and bottom of the the queue. The counters are manipulated in two "critical sections" of program, which are implemented with a spinlock mechanism to ensure access by only one processor at a time. The processor locks the critical section, reads and increments the "top" counter, then releases the lock. When nodes are returned to the bottom of the queue, a similar mechanism is used to synchronize access to the "bottom" index. When the top counter reaches the bottom of the queue, and no nodes are being processed, the computation is complete. We investigate two methods of using the queue and counter to allocate work to the processors:

- **Shared data:** Place all elements of the queue in shared memory. Each processor takes an entry from the top of the queue, completes one cycle of exact and inexact extension, and places all newly created repeats at the bottom of the queue. This process continues until no further extensions are possible.

  This method is meant to distribute the work as evenly as possible across the processors, which operate on small tasks. While this method balances the workload well, it calls for a large amount of data traffic across the system bus to the processors, a large number of accesses to shared memory, and a significant amount of synchronization.

- **Shared index:** Copy all entries of the initial queue to each processor's local memory. Each processor maintains a unique local queue. Available elements on the local queues are determined by a global "top" index. Each processor reads and increments the global top index, takes the indexed element off the top of its own queue, and completes in its local memory all possible cycles of extensions for this element. Processors repeat the process until all members of the initial queue have been processed. The only requirement for shared memory is the global counter describing the next available element of the queue.

  This method attempts to place a small load on the shared memory and bus, at some cost in load balancing. No subsidiary nodes are returned to

any shared queue. Instead, each processor completely forms the entire subtree of its selected node and displays the results before returning to the queue to select another node.

The shared data method implements the synchronization by maintaining two pointers to the queue. "Top" points to the next available active queue node, while "bottom" points to the next empty space for a node on the queue. If another processor has locked the section, processors seeking to access the queue wait idly. In both cases top and bottom start at 0. The actual operations for the shared data method are as follows:

```
LOCK
    If adding then
      b = bottom;
      bottom = bottom + #nodes added
    endif
    If removing and top < bottom then
      t = top;
      top = top + 1
    endif
UNLOCK
If adding then
  add elements b, b+1, ..., b+#nodes -1 to queue
endif
If removing and t is defined then
  remove element t from queue
endif
```

Very few CPU cycles are used in the critical locked section of the code. The expensive data transfer to or from the queue is performed outside (and immediately after) the locked section, after the appropriate elements are identified with pointers b and t to the queue.

The shared index method uses an analogous locking scheme. Since no data are ever returned to the queue, only the data removal section is used:

```
LOCK
    If (top < 4^k) then
      t = top;
      top = top + 1;
```

```
    else
      done
    endif
UNLOCK
If done then
  stop
else
  remove top queue element and extend its full subtree
endif
```

Each processor can eventually write its own partial queue information to the output display. For the timing runs reported below, no output was performed by either method.

5 Target Biological Sequences

We use as a sample problem the 11 known sequences of the ribonuclease P RNA, the catalytic element of a ribonucleoprotein enzyme [2, 8]. In addition to allowing a broad range of sequence similarities, this RNA is important because it was one of the first RNA's shown to have catalytic activity [2]. The length and name of these sequences are show in table 1 below.

The first seven sequences have been aligned by James *et al.* [8] by a series of iterations using additional information which would not be available to a general sequence alignment algorithm. This included implied secondary structure and conservation of pairs of nucleotides that are complementary. The first four sequences are all Bacillus species and fall within one phylum. The next three are members of another phylum, the "purple bacteria". Alignments within each of the phyla are easier than alignments between phyla, due to evolutionary distance of the sequences. For the latter, James *et al.* [8] used selected subsequences containing several nucleotides that are the same in all seven sequences. These are shown in table 2.

The last four sequences are more distant in evolutionary relatedness. We know of no published alignments among these four sequences or between these four and the preceeding seven. The correspondences between S. Octosporus and S. Pombe are readily apparent in figure 1, but S. Cerevisiae and Hela show little relationship to the other sequences. There is a need to obtain sequences of ribonuclease P RNA from additional species to provide a more

Length	Name
401	Bacillus Subtilis
417	Bacillus Stearothermophilus
408	Bacillus Megaterium
411	Bacillus Brevis
354	Pseudomonas Fluorescens
375	Salmonella Typhi
377	Escherichia Coli
282	Saccharomyces Octosporus
284	Saccharomyces Pombe
369	Saccharomyces Cerevisiae
339	Hela

Table 1: Sequence lengths and names.

Bacillus Subtilis	E. Coli
15 – 22	12 – 19
43 – 53	61 – 71
179 – 188	124 – 133
226 – 253	229 – 256
258 – 263	292 – 297
313 – 322	327 – 336
369 – 382	347 – 360

Table 2: Alignment subsequences.

FIGURE 1: Pairwise matches of length 15 or greater.

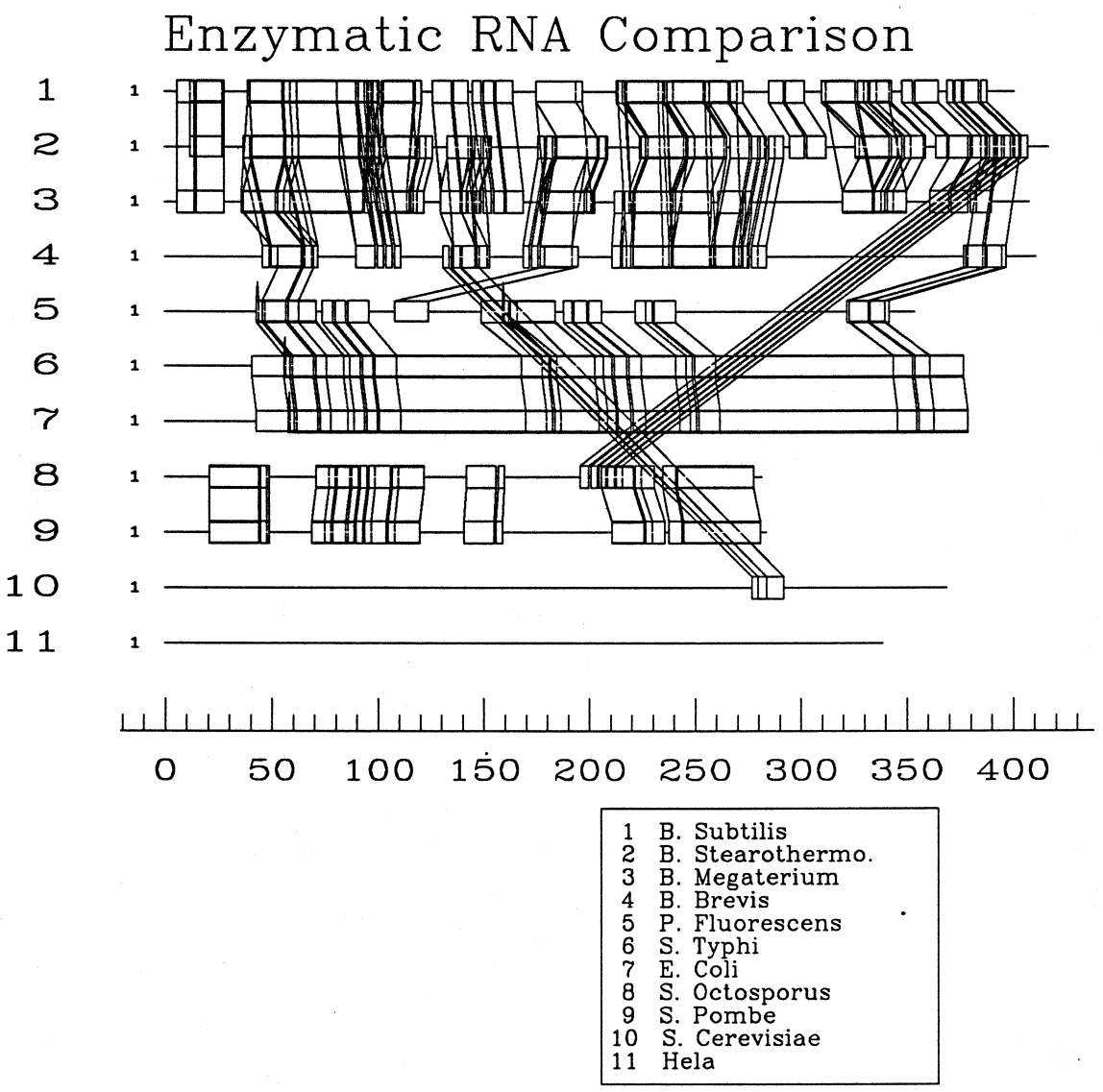
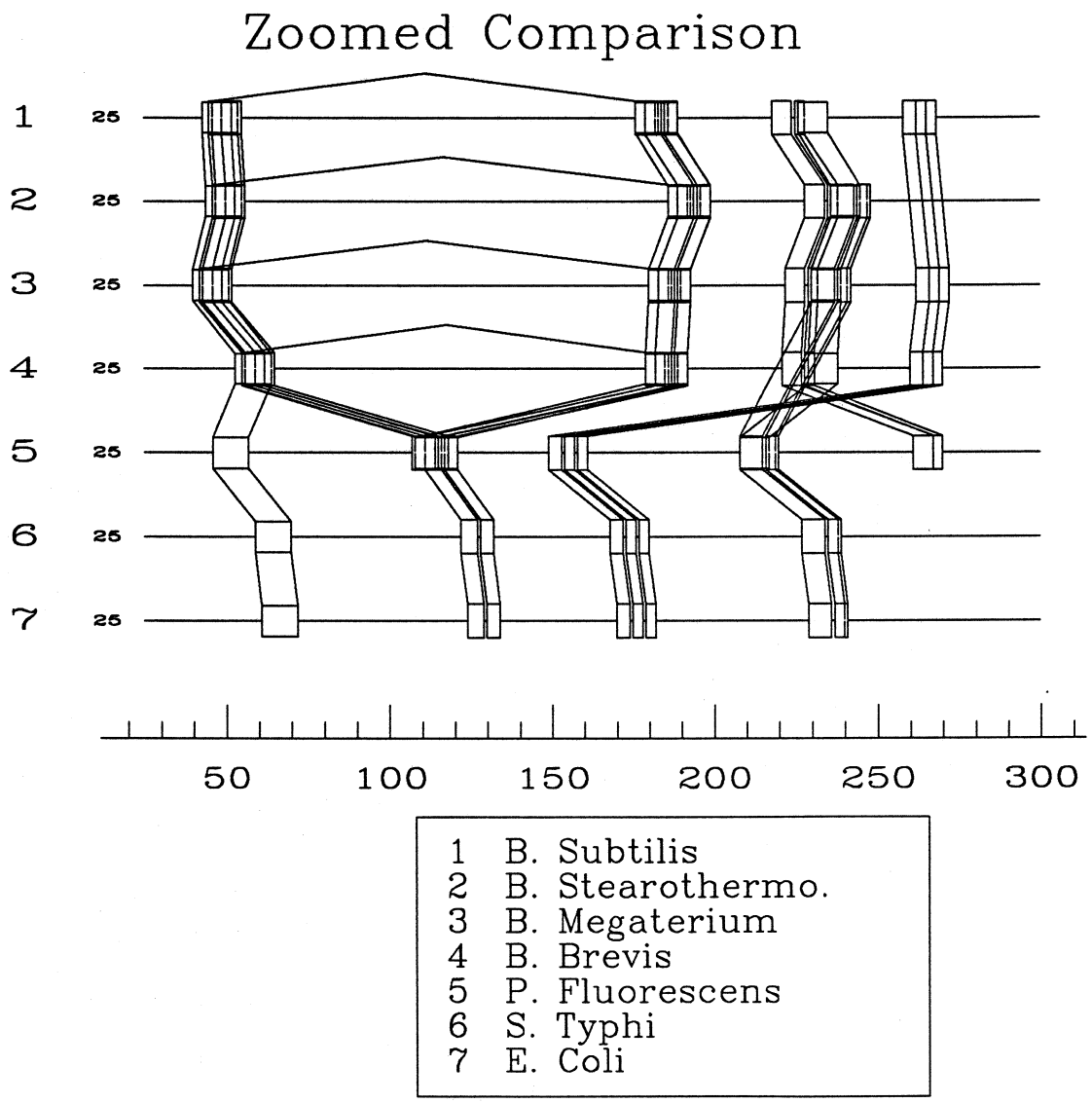


FIGURE 2: Five-way matches of length 9 or more on a subset of sequences.



INSERT FIGURE 1 HERE

Figure 1: Pairwise matches of length 15 or greater.

INSERT FIGURE 2 HERE

Figure 2: Five-way matches of length 9 or more on a subset of sequences.

gradual transition from the first seven to the final one. With the increasing availability of large numbers of closely related sequences, the ability to consider the 11-way comparison simultaneously is an important strength of the algorithm. The ability to find repeats within as well as between sequences also is useful for certain sequences.

Several of the alignments of table 2 are visible in our figures. Of the seven conserved segments, figure 2 clearly shows the second, third and fourth as being present in at least 5 out of the 7 sequences at lengths 9 or greater. Figure 1 shows the conservation of the first segment in three of the Bacillus sequences, and the seventh segment's presence in 6 out of 7 sequences. Figure 2 presents an interesting deviation from those tabulated data for the fifth segment, as it shows the matching contributions in P. Fluorescens, S. Typhi, and E. Coli occuring in the range from bases 150 — 180, instead of 347 — 360.

6 Graphical Display of Multiple Sequence Comparisons

We have developed a prototype graphical display program to present the output of our parallel multiple sequence comparisons. Multiple sequence comparisons generate large volumes of sequence similarity data, particularly when inexact matches are allowed. Graphical analysis of the data elucidates the overall framework of sequence similarities and can help guide further analysis at the sequence level. Furthermore, a multiple sequence comparison plot can present information more succinctly than a series of pairwise comparisons.

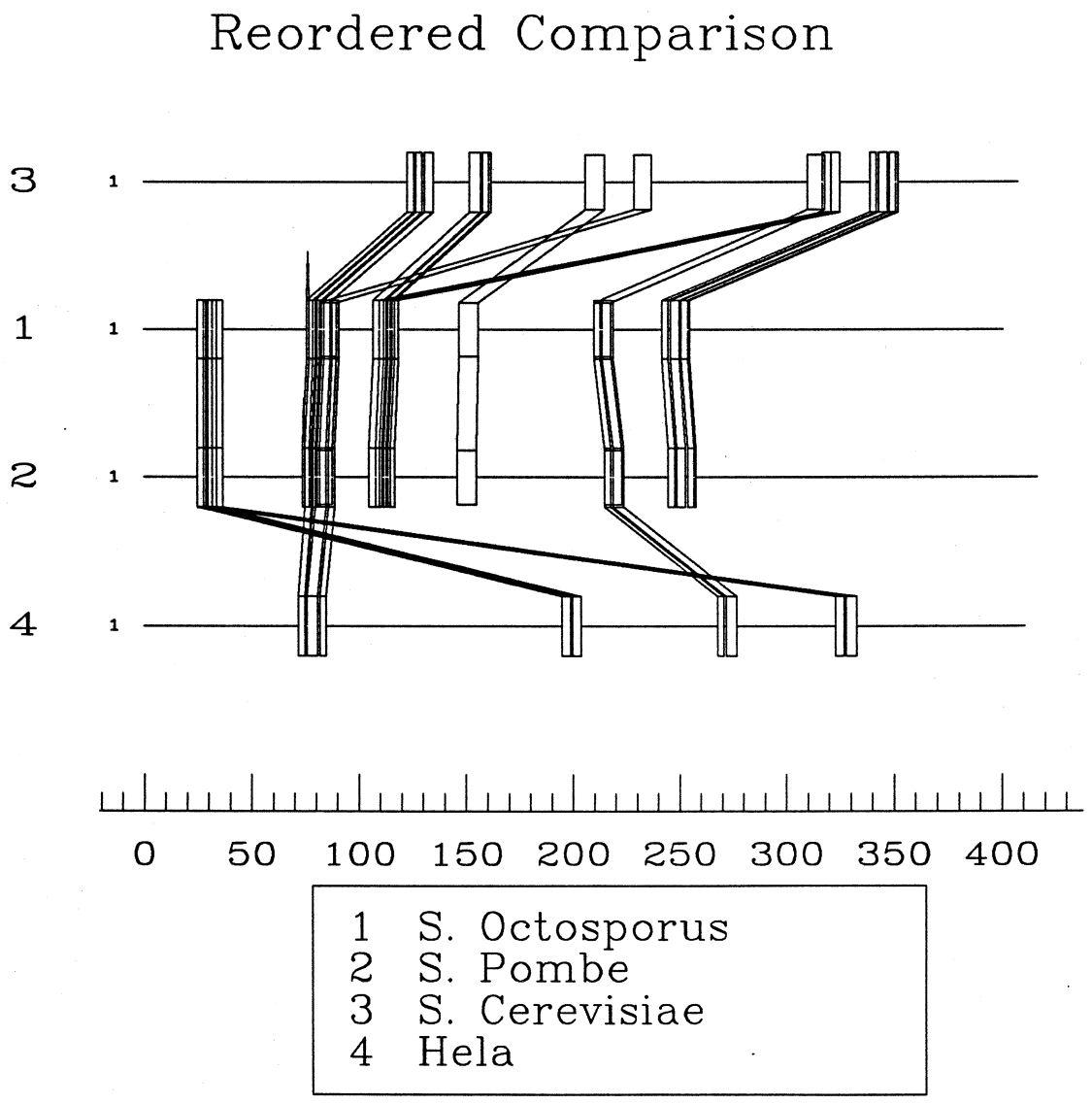
The graphics program is based on an interactive programming and graphics language CLAM^{R} (the Computational Linear Algebra Machine[1])[6]. CLAM is an interactive numerical computing environment that runs on a variety of UNIX^{TM} based computers. It is capable of producing graphics output for Sun workstations using Suntools^{TM}, for X Window System^{TM} server workstations or displays, and for PostScript^{TM} and Impress^{TM} hardcopy printers. In addition to the line drawing capabilities of CLAM used in our program, there are 2D plotting, 3D surface and contour drawing, color, and animation graphics features.

The main feature of the graphical display is the representation of multiple sequences and their inexact matches. As shown in Figure 1, exact matches are drawn as boxes on each sequence with lines joining the match instances from one sequence to the next. Multiple occurrences of a match on one sequence are joined by angled lines. An inexact match is drawn as a sequence of exact matches. A horizontal axis is provided to ease location of matches within a sequence.

The display functions as a graphics filter, in that it manipulates the file output of the parallel comparison program. Thus one need not recalculate a complicated or costly sequence comparison each time a plot is desired. In fact, the CLAM graphics filter need not execute on the same computer as the sequence comparison. However, CLAM does support calls to Fortran or

[1]CLAM is a registered trademark of Scientific Computing Associates, Inc. UNIX is a trademark of AT&T. Suntools is a trademark of Sun Microsystems Inc. The X Window System is a trademark of MIT. PostScript is a trademark of Adobe Corporation. Impress is a trademark of Imagen Corporation.

FIGURE 3: Three-way matches of length 8 or more on a reordered subset.



INSERT FIGURE 3 HERE

Figure 3: Three-way matches of length 8 or more on a reordered subset.

C subroutines, so that we could have called the parallel sequence comparison program directly from CLAM.

Additional features of the graphics display include legend plotting, sequence reordering, sequence masking, offset insertion, and zoom. As shown on figure 1, the sequence names are displayed in the boxed legend. The number next to the name indicates the sequence number in the figure.

Sequences may be reordered to change their vertical placement. This is useful in grouping highly related sequences, thereby reducing the number of match-joining lines crossing non-matching sequences. The user specifies a permutation vector to control the reordering. The graphics filter updates the legend to reflect the reordered sequence numbers.

The user may select a subset of the available sequences for display, in order to highlight particular matches. This is accomplished by setting a mask vector of logical values. Matches on the masked-off sequences are not displayed. The vertical spacing and legend can be adjusted to reflect the new selection. Only matches on the masked sequences are displayed.

The vertical alignment of matches depends on the correct alignment of sequence fragments. The program allows the user to specify an alignment offset, indicating the number of bases to shift each entire seqeuence to the right before plotting. The vertical axis displays the sequence offset (in smaller type) next to the sequence number.

Finally, it may be of interest to focus on a particular segment of the compared sequences for greater clarity. This is accomplished by resetting the lower and upper sequence limits in the plot. The program then replots the sequence comparison output with the new limits. CLAM automatically handles the zoom function, including clipping of data outside the specified plotting region.

Figure 2 shows the use of several of these features. A subset of the se-

quences has been selected, its order changed, offsets added for better match alignment, and a zoomed view of part of the sequence displayed. The displayed matches occur in at least five out of the seven sequences shown. This figure shows the strong correspondence of the first seven sequences.

Figure 3 shows the reordering of sequences in a comparison of S. Octosporus, S. Pombe, S. Cerevisiae, and Hela, in which only matches with at least three occurrences are displayed. The reordering is used to place related sequences in proximity, for clarity of the image. Plots such as this can be used to search for relationships between a subset of the entire set of compared sequences.

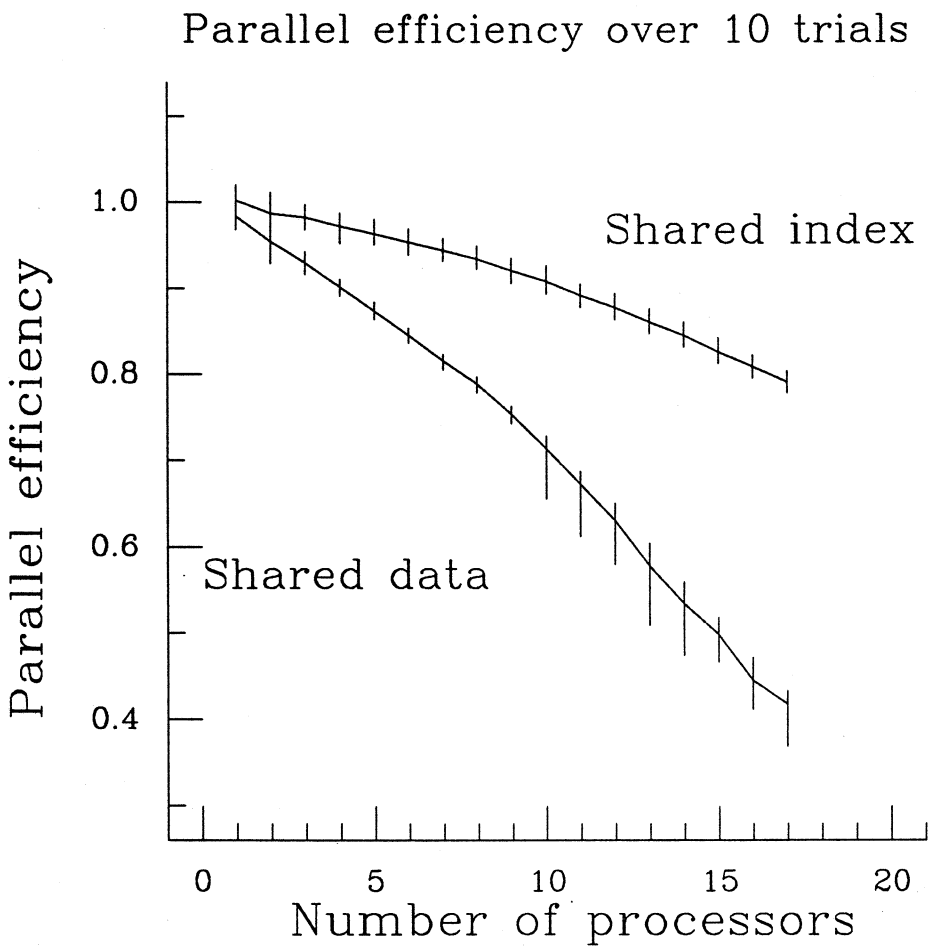
7 Experimental Results

It is worth noting that the total parallel computation time for the comparison of these 11 sequences, of average length approximately 370, is about 13 seconds on a 2 MIPS computer. The initial sequential computation requires 2.3 seconds. This is in direct contrast to the cost of $370^{11} \approx 10^{28}$ operations necessary with the direct dynamic programming computation, an operation count that renders such a multiple sequence comparison by that method infeasible on any computer. We attempted to make our timing runs at times when few or no other jobs were running on the system.

A useful measure of the effectiveness of a parallel implementation is its efficiency ratio. Figure 4 shows the parallel efficiency of the two methods on various numbers of processors for our test problem. (The efficiency was computed as the time for a non-parallel single processor version of the program divided by the total parallel time on the same problem. Both methods compute timings only for the extension of the initial queue; the formation of the initial queue is not included.) In each case, ten repetitions were computed to identify the variation in timings due to other jobs running on the computer and competition of processors for system resources. The curves are drawn through the average of 10 repetitions of the calculations; tick marks extend to the minimum and maximum times out of the 10 trials. We note that both methods produce identical sequence comparison output.

It is clear from figure 4 that the shared index method is substantially more efficient than the shared data method. We shall now explain why this is so, with the goal of deducing general principles for similar parallel programs.

FIGURE 4: Parallel processor efficiency measures.



INSERT FIGURE 4 HERE

Figure 4: Parallel processor efficiency measures.

The curve labelled "shared data" represents the method in which all queue elements are maintained in shared memory. The second curve, labelled "share index," represents the method that keeps the queue nodes in disjoint local memories and shares only the index of the next active node on the queue, as discussed above.

The resource usage of the programs are as follows. The number of spin-locks controlling access to critical program segments is two for each queue element. The shared memory requirement can be estimated by adding the following items needed during a run:

1. Number of queue entries (pointers) $\times$ 4 bytes
2. Number of queue elements (repeats) $\times$ 20 bytes
3. Number of error blocks $\times$ 12 bytes

As an example, for the sample 11-way comparison described in section 5, the memory usage in bytes is approximately as follows:

$$940,000 = 5000 \times 4 + 40,000 \times 20 + 10,000 \times 12.$$

The data transfer to and from the main memory is approximately that required to move all of the data in each direction, for a total of 1.9 Mbytes of data traffic. Individual queue elements reside in adjacent storage locations, so there is low likelihood of thrashing due to cache misses.

The shared data method uses approximately this much storage, with most arrays lying in shared memory. The shared index method assigns larger, more variable, amounts of work to the individual processors. It requires more storage, but significantly less global communication. By duplicating

the initial queue in all p processors, the second method uses up to $(p-1)(4^{k+1}+20(N+1-k))$ additional bytes. For our model problem with $p \leq 17$, $k=3$, and $N \approx 4000$, this is up to 1.28 Mbyte additional memory. The actual memory requirements of the program include $p-1$ additional work buffers of 0.8 Mbyte each, but much of this space is never used. Thus the shared index method more than doubles the required memory usage of the shared data method.

The increase in memory is significant, because the individual CPUs have rather small local memories. Thus all 2.2 Mbyte of data must be stored in main memory and accessed over the system bus. With a very low number of computations per datum, which is independent of problem size for both methods, there is a heavy demand on the bus when the number of processors is large. The bus appears to be fast enough to satisfy the requirements of our program, for figure 4 does not appear to show the effects of bus saturation.

The shared index method uses significantly fewer global synchronizations. Both methods require two synchronizations or locks for each global queue element. The shared data method has about 5000 such queue elements, while the shared index method has only 4^k, in this case 64, for 10000 and 128 global synchronizations, respectively. Because it is possible that $p-1$ other processors wait idly when one CPU executes the locked program, the expected degredation due to a spinlock increases with processor number. The chance of a processor encountering a lock also increases as the computation time decreases, because the fixed number of locks must be accomodated in a reduced actual time.

The shared index method has the added advantage of being applicable to disjoint memory machines with many processors, since there is little contention for any shared resource relative to the amount of computation carried out, assuming sufficient local memory.

The data of figure 4 clearly show the deleterious effect of excessive use of synchronization. We infer that the synchronization is the primary cause, because the shared index method actually has higher memory and bus traffic requirements; the shared data method requires two orders of magnitude more synchronizations. The only additional requirement of the shared data method is that it requires execution of the two critical sections of locked code for each of its approximately 5000 nodes on the queue. Thus we attribute its relatively poor performance entirely to synchronization penalties.

The second method operates at above 80% efficiency for 2 — 17 proces-

sors. The factors of synchronization, bus traffic, and memory usage probably contribute to the loss of 20% efficiency for large numbers of processors, but it seems likely that they are not the causes of the immediate decrease in efficiency as we increase the number of processors beyond 1.

There are only 128 synchronizations in a total of several seconds of computation, so the processor idle time due to these waits does not seem a probable cause. Saturation of the bus with data traffic might be at cause, but should not be felt for very few processors. That is, one would not expect to see the observed drop in efficiency for 2 or 3 processors.

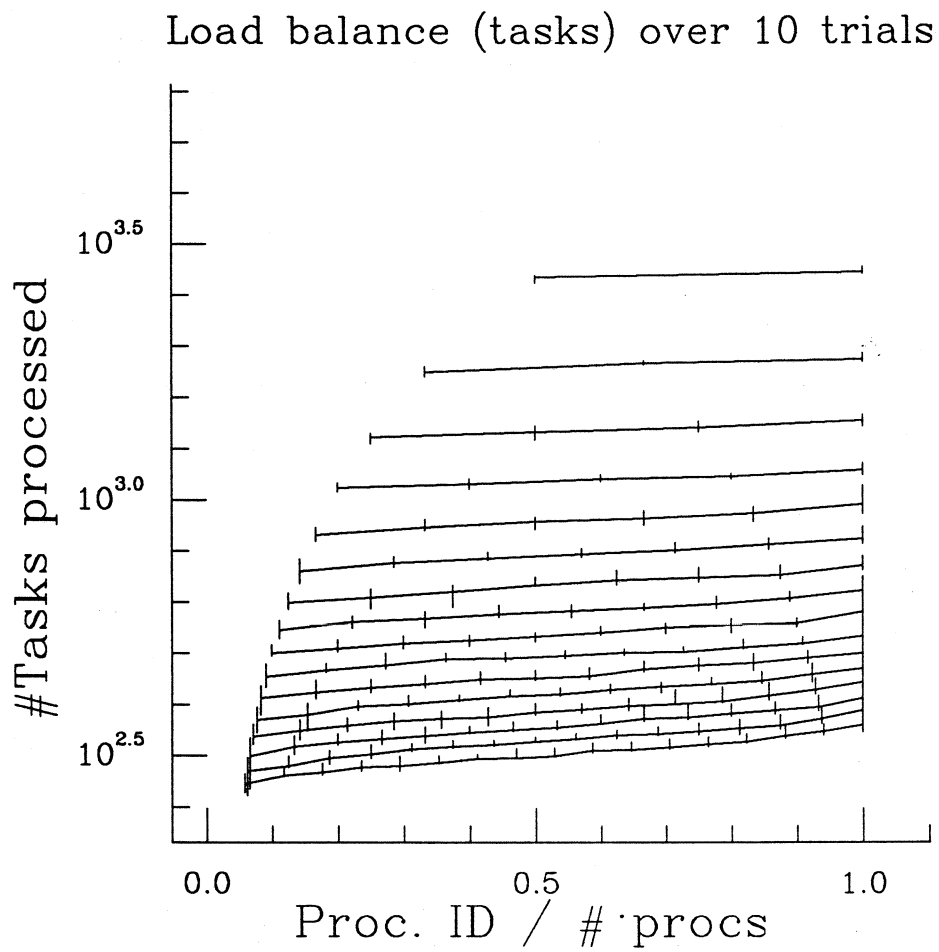
Similarly, excessive shared memory traffic does not seem to be at fault. Only approximately $\log N$ operations are performed per set element in the original queue, and only a constant number of operations are performed per set element in the entire tree. Thus there is a relatively small ratio of fast computation to slow I/O in the algorithm. Again, these effects should not be felt for a small number of processors.

There appear to be two primary causes of the 20% efficiency loss in using 17 processors on the shared index method, one for small numbers of processors and one for large numbers. The first reason must incorporate critical system resources that are fully exercised by one processor. It appears likely that the cost of page faults, as the system creates queue elements on their first access by the CPU, is the true cause of much of this performance loss for a small number of processors. There is a substantial cost in trapping to the operating system during a page fault to bring an uninitialized page from disk to memory.

We have performed a small experiment in C to time the cost necessary to initialize to zero an array of M integers. For $M = 20000$, the time was 74.6 milliseconds on the first initialization pass, but only 48.0 milliseconds on the second pass in the same program. Initializing from location 1 to M and then from M to 1 drove down the second time to 43.2 milliseconds. Comparable times for $M = 80000$ are 304.6, 192.9, and 171.7 milliseconds, respectively.

These numbers reflect a 35% cost due to page faults, with another 6% to 7% due to cache misses. Therefore we should observe better performance from our program if we initialize the large data areas before starting the actual computation. The improved times of the test problem reflect an upper bound on the performance improvements we expect to see, as our program performs more than one operation per datum and thus spends a lower fraction of its time waiting for page faults and cache misses. The cost of page

FIGURE 5: Load balance by node refinement tasks per processor.



INSERT FIGURE 5 HERE

Figure 5: Load balance by node refinement tasks per processor.

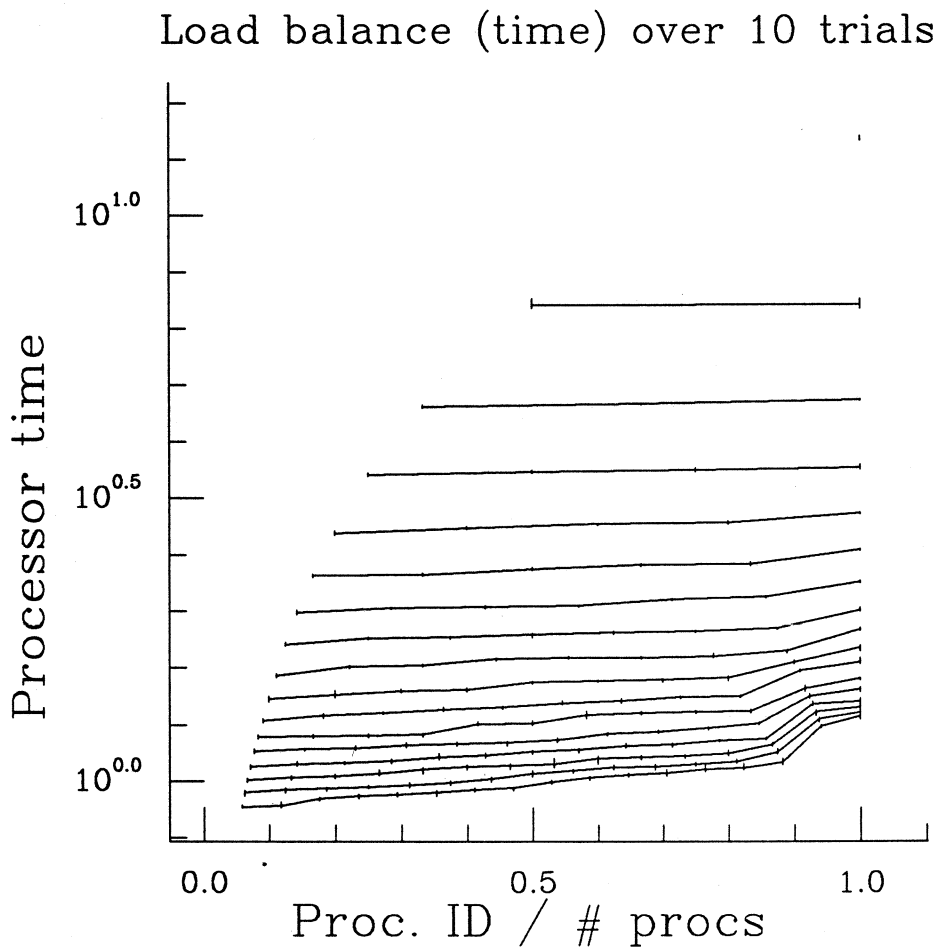
faults explains the immediate loss of efficiency in going from 1 to 2 or more processors in figure 4. In the absence of contention for a shared resource, the efficiency rate should stay at 100%, which it clearly does not. The disk accesses for page faults seem to be the shared resource in question.

One also observes a "knee" in the efficiency curves of figure 4 at approximately 10 processors. As well, the shared data method exhibits increased timing variability after 10 processors. The sharing of each cache by two processors on the Multimax seems to be the cause of these changes. We conclude that the Multimax operating system schedules new tasks on unoccupied processors, effectively giving as many as 10 processors an entire 64 Kbyte cache each. When two processors share a cache (11 tasks are necessary, unless system processes are running), they incur additional waits due to cache misses. Thus we see performance degredation with over 10 processors.

Figure 5 shows the effect of the "shared index" scheme in terms of load balance on the active processors. Note that the shared data method should exhibit better load balancing due to its small task size, but the loss of load balancing in the "share index" method is more than compensated for by reduced usage of shared system resources. Each curve represents a given number P of processors applied to the problem, with curve data points plotted at abscissae of $1/P$, $2/P$, ..., P/P. The data points forming the curve are the number of tasks completed by the P processors, in sorted order. Thus the ordinate for $2/P$ is the second least number of nodes processed by any processor when P CPUs solved the problem in parallel. The tick marks extend to the minimum and maximum values over 10 repetitions of the computation. The curves are drawn through the average values over the 10 repetitions. This scheduling system achieves a good balance of work by this measure. This balance is better than anticipated.

Figure 6 shows the processor load balance in terms of CPU time. Here

FIGURE 6: Load balance by CPU time per processor.



INSERT FIGURE 6 HERE

Figure 6: Load balance by CPU time per processor.

the distribution of work is visibly uneven. For nine or more processors, the top one to three processor loads are substantially greater than an even distribution of work would provide. This uneven distribution is due to the large blocks of work assigned to individual processors; each processor gets one or more subtrees of T and works to completion on those nodes. It is only surprising that the load imbalance was not substantially worse than observed, particularly for the case of 17 processors. The effect of the load imbalance is not reflected in the efficiency measure of figure 1, where we simply sum processor times to give the overall CPU time. However, on a system where all processors were held idle until the last one finished, decreasing the load imbalance would be important.

8 Conclusions

Our computational results establish the suitability for parallel implementation of a powerful and efficient sequence comparison algorithm. Our parallel program achieves satisfactory speedups on up to 17 processors of an Encore Multimax. The combination of efficient algorithm and high parallel processor efficiency indicates the suitability of this method for larger sequence comparison tasks. Parallel sequence comparison by this or similar methods can provide a powerful tool for analysis of large sequence databases such as Genbank.

We have developed a prototype graphical display program that renders usable the volumes of data that arise from a multiple sequence computation. The graphical display program presented above allows the user easily to deduce essential alignment information from such a comparison. The program's additional features enable one to focus on smaller features of interest as well.

Our sequence analysis reflects the alignment information of James, with additional information about matches found within a subset of his original seven sequences. In particular, the figures produced by our graphics program show the strong relationships of sequence groups 1 — 4, 5 — 7, 8 — 9, and 10 — 11. It is apparent that sequences 10 and 11 bear only moderate relationship to the others in terms of our matching criteria.

We conclude from our timing experiments that very careful use must be made of synchronization and implicit bus and memory traffic on a multiprocessor such as the Encore Multimax. Even page faults must be carefully accounted for. This is particularly true for programs, such as ours, that do very few computations per datum. In any case, allowing a large number of computations per data transfer or synchronization step is essential to good processor utilization.

9 Acknowledgements

We thank George McCorkle for generously providing us with the sequence data used in our computations. Our appreciation goes to Stan Eisenstat for helpful discussions about the behavior of our parallel program.

References

[1] Academy backs genome project. *Science*, 239:725–726, 1988.

[2] M. Baer and S. Altman. A catalytic RNA and its gene from Salmonella typhimurium. *Science*, 228:999–1002, 1985.

[3] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. Technical report, University of Denver, Denver, Colorado, 1985.

[4] Encore Computer Corporation, Marlboro, MA. *Multimax Technical Summary*, 1987.

[5] D. E. T. Foulser. *On Random Strings and Sequence Comparisons.* PhD thesis, Department of Computer Science, Stanford University, 1986.

[6] W. D. Gropp, D. E. Foulser, and S. Chang. *CLAM User's Guide: The Computational Linear Algebra Machine.* Scientific Computing Associates, Inc., New Haven, Connecticut, USA, 1989.

[7] Intelligenetics, Inc., Mountain View, CA. *News from Genbank,* vol. 2, no. 1, pg. 1, January/February 1989.

[8] B. D. James, G. J. Olsen, J. Liu, and N. R. Pace. The secondary structure of a ribonuclease P RNA, the catalytic element of a ribonucleoprotein enzyme. *Cell*, 52:19–26, 1988.

[9] S. Karlin, G. Ghandour, F. Ost, S. Tavare, and L. J. Korn. New approaches for computer analysis of nucleic acid sequences. *Proceedings of the National Academy of Sciences USA*, 80:5660–5664, 1983.

[10] S. Karlin, M. Morris, G. Ghandour, and M.-Y. Leung. Efficient algorithms for molecular sequence analysis. *Proceedings of the National Academy of Sciences USA*, 85:841–845, 1988.

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

[12] H. M. Martinez. A flexible multiple sequence alignment program. *Nucleic Acids Research*, 16:1683–1691, 1988.

[13] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23(2):262–272, 1976.

[14] 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.

[15] D. Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences USA*, 69(1):4–6, 1972.

[16] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[17] E. Sobel and H. M. Martinez. A multiple sequence alignment program. *Nucleic Acids Research*, 14:363–374, 1986.

[18] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automatic Theory*, pages 1–11, 1973.

[19] W. J. Wilbur and D. J. Lipman. Rapid similarity searches of nucleic acid and protein data banks. *Proceedings of the National Academy of Sciences USA*, 80:726–730, 1983.