ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439-4801

ANL/MCS-TM--132

DE89 014163

Generating Alignments of Genetic Sequences

by

Ralph Butler, Tracye Butler, Ian Foster,
Nicholas Karonis, Robert Olson, Ross Overbeek,
Nathan Pfluger, Morgan Price, Steve Tuecke

Mathematics and Computer Science Division

Technical Memorandum No. 132

June 1989

This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy under Contract W-31-109-Eng-38.

Contents

Generating Alignments of Genetic Sequences

Ralph Butler, Tracye Butler, Ian Foster,
Nicholas Karonis, Robert Olson, Ross Overbeek,
Nathan Pfluger, Morgan Price, Steve Tuecke

Abstract

Molecular biologists have recently developed the technology required to determine the genetic information of complete organisms. They are now faced with a number of interesting computational problems as they attempt to process this data.

We are interested in developing a software environment to support molecular biologists. As their computational problems are frequently complex and computation-intensive, we believe that such an environment must both support rapid prototyping of new algorithms and allow high performance execution on a variety of multiprocessor configurations. We believe that this can be achieved using a bilingual approach, in which the upper levels of programs are coded in a concurrent logic programming language and the lower levels in C. The concurrent logic language provides ease of parallel programming and portability over a range of parallel computers; C provides efficient implementation of low-level algorithms.

To explore the suitability of this approach, we have investigated its use in attacking a prototypical computational problem, the problem of aligning a set of sequences of genetic material. This report introduces the algorithm used to generate alignments, outlines the techniques used to develop the bilingual program, and describes initial experiments in parallel execution of this program.

1 Introduction

The incredibly rapid progress in molecular biology is now making headlines in major newspapers. Advances are reported on a weekly basis. The growing interest in molecular biology will inevitably make it one of the more important application areas in computer science. Currently, computers play a secondary role but projected demands for computation are enormous. Let us give a few examples.

1. Genbank (one of the most widely distributed databases containing known sequences) contains just over 20,000 entries. This number has been doubling about every 15 months, which might well seem like rapid growth. However, in order to take on a project like sequencing the genetic material for an advanced organism such as a human, the database will have to grow (fairly rapidly) to the point where it can absorb over a million new sequences each day.

2. Most of the information that one wishes to include in such a database is currently either uncertain or unknown. For example, the 3-dimensional structure of proteins (information that is critical for many applications) is known for only an extremely small fraction of the sequenced proteins. This means that as data is added to the database, one would like to attempt to infer speculative data concerning closely related sequences. Specifically, if the structure of one sequence were discovered, it would become possible to make intelligent guesses concerning corresponding sequences in closely related organisms.

3. Since one is interested in sequences that are closely related, a common query involves searching the database for sequences that are genetically similar. Currently, such a search requires a modest amount of computing resources (a single search frequently takes on the order of 2 hours of VAX time). When one considers the issues involved with scaling up both the size of the database and the rate of requests by as much as 5 orders of magnitude, the problems become challenging.

It seems possible that the computational aspects of molecular biology will be so interesting and so intense that much of the basic research in this area will be completely dependent on the construction and maintenance of an integrated database.

One of our goals is to aid in creating such a database, along with the tools which would allow effective access to the stored information. We are working to establish a software environment and an actual database that satisfy the following objectives.

1. The implementation must be efficient in the sense that it must support loads of the sort projected above.

2. It must be easy to extract data from the database and to experiment with algorithms to utilize the data in unpredictable ways.

3. We should be able to easily exploit advances in hardware environments that would allow substantial improvements in basic capabilities. In particular, it seems likely that we will wish to exploit a range of multiprocessing configurations, as such systems are introduced into the commercial marketplace.

Given these objectives, a technology based on the use of concurrent logic programming, with some of the major computationally-intensive program components written in C, seemed appropriate. Concurrent logic programming languages provide ease of parallel programming and portability over a range of parallel computers; C provides efficient implementation of low-level algorithms.

To explore the suitability of such an approach, we have investigated its use in attacking a prototypical computational problem, the problem of aligning a set of sequences of genetic material. The concurrent logic programming system *Strand*[1] was used as the implementation vehicle. This system provides explicit support for bilingual programming; in addition, good-quality implementations are available on a variety of parallel machines.

This report introduces the algorithm used to generate alignments, outlines the techniques used to develop the bilingual *Strand*–C program, and describes initial experiments in parallel execution of this program. The latter sections of the report assume some familiarity with concurrent logic programming, as might be gained from [3].

2 The Problem

For the purposes of this discussion, a sequence of genetic material may be thought of as a string of characters from some fairly small alphabet. In particular, we will take our examples from sequences of RNA, which amount to strings from the alphabet {a,c,g,u}. For example, the following are typical short sequences of RNA.

```
augcgagucuauggcuucggccauggcggacggcucauu
augcgagucuaugguuucggccauggcggacggcucauu
augcgagucuauggacuucgguccauggcggacggcucagu
augcgagucaaggggcucccuucggggagcaccggcgcacggcucagu
```

The reader should note that these sequences are similar, but not quite identical. In fact, they represent corresponding pieces of genetic material from four distinct (but closely related) organisms. There is a great deal that can be learned from such related pieces of genetic material. One critical operation in extracting this information involves *aligning* the sequences. An alignment is created by lining up the sequences with corresponding sections directly above one another. To make

[1] *Strand* is a trademark of Artificial Intelligence Limited.

corresponding sections line up, dashes are inserted into the sequences. These dashes are called *indel* characters, since they represent areas in which insertions or deletions of characters are required to match up the corresponding sections of the sequences. For example, the following is an alignment of the four sequences given above:

```
augcgagucuauggc------uucg----gccauggcggacggcucauu
augcgagucuauggu------uucg----gccauggcggacggcucauu
augcgagucuauggac-----uucg---guccauggcggacggcucagu
augcgaguc-aaggggcucccuucggggagcaccggcgcacggcucagu
```

The application discussed in this report uses a concurrent logic program to automatically generate such alignments. The sets that we align will normally contain 2–50 sequences; the individual sequences will contain between 10 and 2000 characters. Although we will restrict ourselves to fairly short sequences when giving illustrations, the reader should remember that our algorithms are intended for much longer sequences.

2.1 Why Create Alignments?

A great deal can be learned by studying similarities and differences in sets of sequences. In the case of the alignments that we have generated, the sequences being aligned are known to come from corresponding genetic material in individuals from closely related species. Once such a set of sequences is aligned, it is possible to extract a fairly accurate guess about how the organisms relate in the tree of evolution. That is, it is likely that by studying the differences between such similar sequences that we can make accurate guesses about the progress of evolution. This in itself is of substantial interest to some biologists. One of the recent advances in understanding the evolution of early life forms was based on a carefully prepared alignment of RNA from bacteria ribosomes. See Chapter 28 of Watson *et al.* [7] for a discussion of the discovery of the relationships between eubacteria and archaebacteria.

Suppose that the sequences come from individual members of the same species (rather than from a variety of species). One may wish to search for genetic differences that relate to observable differences in the individuals. This can allow the differences that cause certain diseases (or a propensity to contract certain diseases) to be isolated. Again, aligning the sequences can reasonably be considered the first step.

Finally, consider the case where a biologist has just produced a given sequence in the laboratory, and hypothesizes that it represents genetic material that performs some well-defined function; let us call it a "widget". The biologist might well search through a growing database of all known genetic sequences in the hope of finding occurrences of similar widgets. Once a set of such sequences has been extracted, an alignment might be used to illustrate the exact variations on what is believed to be a common theme.

2.2 What Is a Correct Alignment?

Before describing an algorithm to generate alignments, let us consider the issue of exactly how one might determine whether or not such an algorithm produces "good" alignments. There are at least three reasonable positions that can be taken:

1. Sequences really represent molecules or pieces of molecules. These molecules usually have a fairly well-defined three dimensional structure. That is, they fold into a characteristic shape. This is certainly not true of all sequences, but it is true of many that biologists consider. Suppose that we increased the size of the molecules represented by a set of sequences until they were roughly the size of pieces of furniture. Further, suppose that all of these molecules had roughly the same shape. Then, it would make sense to carefully match up the corresponding sections. It is true that some might have unique bumps, and some might be missing sections altogether, but the essential structural theme might be apparent enough to allow a meaningful assignment of correspondence. In such cases, the notion of "correct alignment" might well be based on a **structural standard**. While biologists do not have the luxury of viewing molecules so directly, they do have access to a great deal of physical data accumulated through years of experiments designed to reveal structural information.

2. A second approach might be based on defining some notion of *genetic distance*. One sequence can be transformed into another by changing individual characters, inserting characters, and deleting characters. While there are infinitely many ways to perform such a transformation, it is possible to meaningfully define the notion of a "minimal" set of operations. If one then hypothesizes that evolution would most likely have occurred through such a minimal path, then a "correct alignment" should depict just such a minimal set of operations. We call such a view the **genetic distance standard**.

3. Finally, one can simply take alignments that have been carefully produced by biologists (frequently taking several years) and call them "correct". Clearly, these alignments reflect the myriad of considerations that really are weighed by the practicing biologist. This simplistic view does not allow one to make any judgement at all about alignments produced automatically. However, it can be generalized to say something like "any alignment that a competent biologists asserts is correct should be viewed as correct". We call this the **operational standard**.

We have adopted the operational standard. This has obvious drawbacks, since competent biologists do, in fact, argue over which of two alignments for the same set of sequences is correct. In such cases, we are perfectly content to call both

alignments "correct" and let them both stand (until further information leads the community of biologists to rule one out).

Our goal was to write a program that would take as input a set of sequences and produce as output a correct alignment. While not completely successful (our alignments do differ somewhat from those produced by expert humans), the program does produce alignments which (in the cases that we have studied) appear to be significantly better than other automatically generated alignments.

3 An Algorithm for Producing an Alignment

A fair body of literature has been generated about how to align two sequences. Most of it has been based on the genetic distance standard and is based on dynamic programming algorithms. A readable introduction can be found in Sankoff and Kruskal [6] (see, in particular, Chapter 2). A straightforward generalization of these approaches to larger sets of sequences is computationally too expensive (for m sequences of length n, the algorithm is $O(m^n)$). Hence, a variety of other approaches has been tried (for a summary, along with pointers to the relevant literature, see von Heijne [4]). Our experience has been that these approaches are viewed as useful to biologists, but that the results differ substantially from alignments produced by biologists manually. Comments like "they fail to take secondary structure into account" (a reference to structural information known to the biologists) are common. The algorithm that we present has, on a very limited set of alignments, been judged substantially superior by a competent biologist. It is too soon to claim that our approach is actually better (it has not been evaluated on a large enough set of alignments), but the techniques used to implement it are of interest in themselves.

Our algorithm is based on the notion of **critical subsequences**. A critical subsequence (of a single sequence) is a short string of characters (say, 8 to 20 characters in length) that occurs within a sequence and is "not at all similar to" any other string that occurs within the same sequence. For the purposes of this discussion, we will say that two strings are "similar" if they differ in less than 30% of their characters, and a string is a critical subsequence if it is not similar to any other string that occurs within the sequence.

Suppose that a string C is a critical subsequence of two sequences S1 and S2. If we think of S1 and S2 as genetically related, it seems highly likely that the two occurrences of C must line up exactly in the final alignment. Otherwise, there must exist a C1 in S1 and a C2 in S2 such that C aligns with each of these sequences:

$$\ldots \mathrm{C} \ldots \mathrm{C1} \ldots$$
$$\ldots \mathrm{C2} \ldots \mathrm{C} \ldots$$

However, the genetic distance between these corresponding pieces of the alignment are fairly large. While it is quite possible that a section of genetic material like C

will be displaced during the process of evolution, it is quite rare. Hence, we view the presence of identical critical subsequences as extremely important clues as to how the final alignment should appear.

When a critical subsequence occurs in two or more sequences, we call the set of occurrences a **pin**. Our algorithm will attempt to create an alignment in which as many pins as possible align exactly. Indeed, our analysis shows that, for the alignments produced by biologists, the strings in pins do (almost always) line up. However, very occasionally incompatible pins are detected. In some cases, it appears that a section of genetic material has been "moved" over a substantial distance. It is important that such inconsistencies be detected and eliminated; we call such a procedure "cleaning a set of pins". If we consider two sequences as lines, and we view pins as arcs connecting points on the lines, then inconsistent pins are detected by looking for arcs that cross. By carefully removing a minimal number of arcs, we produce a set of pins that form a consistent set of constraints.

3.1 Inserting a Sequence into an Existing Alignment

Before discussing the overall algorithm, let us consider the easier problem of inserting a new sequence S into an existing alignment A. This is done by forming a clean set of pins that connect subsequences in S to subsequences that occur in A. These constraints are then used to partition the overall alignment problem into a set of much smaller subproblems, aligning all of the subproblems using some other approach, and then concatenating the set of smaller alignments. This approach is far different than the more usual approach of forming a "consensus" of the sequences in A, and then using a dynamic programming algorithm to align S with the consensus.

Once the original insertion problem has been reduced to a set of smaller problems, there remains the issue of exactly how to solve each of these (hopefully) much smaller problems. Since these smaller problems have exactly the same structure as the original problem, it might seem that recursion would be appropriate. Indeed, it can be used (since subsequences may be critical in one of the small problems, while not being critical in S). However, eventually we must face the issue of inserting a sequence into an alignment under the condition that no pins exist between the sequence and the sequences in the alignment. We have experimented with several approaches. The one that seems to work best is as follows:

1. Strip the indels (dashes) from each of the sequences in the alignments. Then use one of the dynamic programming algorithms to form pairwise alignments between the sequence to be inserted and each of the stripped sequences from the alignment. The object is to find the sequence in the alignment that is genetically "closest" to the sequence to be inserted.

2. Once the closest sequence from the alignment has been determined, examine

the alignment between that sequence and the sequence to be inserted. Find the longest chunk of the alignment that contains no indel characters, and form a pin P between the corresponding characters in the two sequences.

3. Use P to force a constraint between the sequence to be inserted and the small alignment. If the entire sequence is included in P, then the sequence can be directly inserted. Otherwise, again partition the problem into a smaller set of problems (to align the nonpinned sections of the sequence against the correspondingly smaller sections of the small alignment), and solve the smaller problems.

This rather strange little algorithm has performed surprisingly well on most of the alignments that we have examined. We now illustrate it in detail, showing the type of situation in which it fails to work optimally. Suppose that we start with the following alignment of four sequences:

```
1 uuu----ggcuaggg-ucgaaccuggguaacaagguagccguaggggaaccugcggcuggaucaccucc--
2 uuu----gguuaugg-ucgaaucuagguaacaagguagccguaggggaaccugcggcuggaucaccuccu-
3 uuc----ugcugugg-ucgaaucuggguaacaagguagccguaggggaaccugcggcuggaucaccuccu-
4 gcaaggaggcagcuaa-ccacgguagguaacaagguagccguaggggaaccugcggcuggaucaccuccuu
```

Now suppose that we wish to insert the following sequence into the alignment:

```
5 guucgcggaggggggcgccgaagguagggaacaagguagccguaccggaagguguggcuggaucaccucc
```

The algorithm first detects the regions of sequence 5 that can be constrained by pins with sequences 1–4. This breaks the sequence into 5 pieces, two of which are sections constrained by pins:

```
        not determined              pinned       not determined    pinned
5 guucgcggaggggggcgccgaagguagg uaacaag-guagccgua ccggaagguguggcu ggaucaccucc
```

Hence, the next step is to insert the unpinned sections into the corresponding sections of the original alignment. Let us begin with the leftmost section:

```
1 uuu----ggcuaggg-ucgaaccuggg
2 uuu----gguuaugg-ucgaaucuagg
3 uuc----ugcugugg-ucgaaucuggg
4 gcaaggaggcagcuaa-ccacgguagg

5 guucgcggaggggggcgccgaagguagg
```

We begin by computing alignments between sequence 5 and the character strings (stripped of indels) from the four shortened sequences. These alignments are as follows:

```
1 uuu------ggcuagggucgaaccuggg
5 guucgcggaggggggcgccgaagguagg

2 uuu------gguuauggucgaaucuagg
5 guucgcggaggggggcgccgaagguagg

3 uucugcugug------gucgaaucuggg
5 guucgcggaggggggcgccgaagguagg

4 g--caaggaggcagcuaaccacgguagg
5 guucgcggaggggggcgccgaagguagg
```

Here the closest match is between sequences 4 and 5. We then take the longest piece of these alignment that contains no indels (i.e., all but the left three characters) and force the corresponding characters to line up in the final alignment:

```
1 u   uu----ggcuaggg-ucgaaccuggg
2 u   uu----gguuaugg-ucgaaucuagg
3 u   uc----ugcugugg-ucgaaucuggg
4 g   caaggaggcagcuaa-ccacgguagg

5 guu cgcggaggggggcgc-cgaagguagg
```

Now the short sections on the left must be aligned:

```
1 u
2 u
3 u
4 g

5 guu
```

Here, the process of aligning the sequences with the shortened section from sequence 5 produces equivalent alignments. Hence, we choose to use sequence 1 (arbitrarily), giving

```
1 --u
2 --u
3 --u
4 --g

5 guu
```

At this point, we have aligned the entire section before the first pin, giving the following initial section of the final alignment:

```
1 --uuu----ggcuaggg-ucgaaccuggg
2 --uuu----gguuaugg-ucgaaucuagg
3 --uuc----ugcugugg-ucgaaucuggg
4 --gcaaggaggcagcuaa-ccacgguagg

5 guucgcggaggggggcgc-cgaagguagg
```

We now proceed to the other unpinned section:

```
1 ggggaaccugcggcu
2 ggggaaccugcggcu
3 ggggaaccugcggcu
4 ggggaaccugcggcu

5 ccggaagguguggcu
```

Since all of the pieces from the original alignment are identical, only one small alignment must be computed:

```
4 ggggaaccugcggcu
5 ccggaagguguggcu
```

By gluing all of the pieces together, we get the following final alignment:

```
1 --uuu----ggcuaggg-ucgaaccuggguaacaagguagccguaggggaaccugcggcuggaucaccucc
2 --uuu----gguuaugg-ucgaaucuaggguaacaagguagccguaggggaaccugcggcuggaucaccucc
3 --uuc----ugcugugg-ucgaaucuggguaacaagguagccguaggggaaccugcggcuggaucaccucc
4 --gcaaggaggcagcuaa-ccacgguaggguaacaagguagccguaggggaaccugcggcuggaucaccucc

5 guucgcggaggggggcgc-cgaagguagggaacaagguagccguaccggaagguguggcuggaucaccucc
```

Unfortunately, the initial part of this alignment does not agree with the following alignment, produced by a human biologist:

```
1 -uuu-----ggcuaggg-ucgaaccuggguaacaagguagccguaggggaaccugcggcuggaucaccucc
2 -uuu-----gguuaugg-ucgaaucuaggguaacaagguagccguaggggaaccugcggcuggaucaccucc
3 -uuc-----ugcugugg-ucgaaucuggguaacaagguagccguaggggaaccugcggcuggaucaccucc
4 -gcaa-ggaggcagcuaa-ccacgguaggguaacaagguagccguaggggaaccugcggcuggaucaccucc

5 guucgcggaggggggcgc-cgaagguagggaacaagguagccguaccggaagguguggcuggaucaccucc
```

The fact that the alignments do not agree reflects an obvious problem in the algorithm that we proposed. The reader may wish to propose an improvement to our algorithm. The ability to formulate algorithms, rapidly implement them, look at the results, and iterate until success is achieved is critical; the software environment discussed below represents our attempt to provide these facilities.

3.2 The Complete Algorithm

Now we are in a position to describe our solution to the more difficult problem of aligning a set of sequences. The overall algorithm is as follows:

1. Compute a set of clean pins for the sequences to be aligned. Locate the "best pin", which is a pin containing critical subsequences from the largest number of input sequences (i.e., the pin fixes a position in the largest possible number of input sequences). If several pins contain the same number of critical subsequences, choose the one closest to the middle of the input sequences.
2. Reorder the sequences so that all sequences connected by the best pin occur above those that are unpinned. Now the pin may be thought of as dividing the original set of sequences into three "regions" – the section of the pinned sequences to the left of the pin, the section to the right of the pin, and the set of unpinned sequences.
3. Align the left section, align the right section, and then concatenate the alignments.
4. Integrate the unpinned sequences into the aligned upper section.

There are numerous unspecified aspects of the above approach. The major issues relate to determining what is meant by the last step. We have experimented with two approaches:

> 4a. Align the unpinned section. Then divide the upper alignment and the lower alignment into a set of subproblems, using pins that connect sequences in the two alignments. Align each of the subproblems (those for which pins do not constrain positioning) by forming "consensus" sequences and using a dynamic programming algorithm to align the two consensus lines. Finally, glue all of the pieces together.

> 4b. Integrate the unpinned sequences into the upper alignment one at a time, using the algorithm we described above. It should be noted that the order in which sequences are integrated can make a substantial difference. It seems best to integrate them by choosing the one that is "closest" to a sequence in the upper alignment and integrating it first (and perform this operation repeatedly, until all of the unpinned sequences have been added to the alignment).

We are not yet ready to report on the relative merits of these two approaches.

4 *Strand* as an Implementation Vehicle

A variety of considerations influence our choice of implementation strategy. First, there is the major choice of whether to use a relatively high level language (like *Strand* or Lisp) or a lower-level language (such as C). The tradeoffs are well known and hotly debated. High level languages offer the ability to rapidly alter algorithms and experiment with variations; lower-level languages offer improved performance.

Early experiments convinced us that performance might well prove to be an important issue. Versions of our algorithms (used on sets of 40–50 sequences, each of which contained 1500-2000 characters) written in C consumed in excess of 5 hours of processing time on a Sun 3/160 workstation. The computation of critical subsequences, in particular, has been studied by other researchers [5]; it is a computationally-intensive operation which can consume substantial processor and/or memory resources. Although we cannot precisely quantify the relative costs of doing such an operation in a higher-level language versus C, it seems likely that the ratio of execution times would be in the range of 5–10 using existing implementations.

The advantages of using a higher-level language for ease of alteration and eventual exploitation of multiprocessors also became apparent. Because the number of alternatives that all require evaluation seems quite large, the time between the proposal of an algorithm and the completion of its implementation is of critical significance. One wishes to be able to formulate conjectures and test them as rapidly as possible.

Our decision to adopt a bilingual approach, with the upper levels in *Strand* and a limited set of "kernels" in C, reflects our very subjective reaction to the above constraints. It is not necessarily critical that we have optimal performance on a program that computes alignments. After all, if the algorithm produces correct output, it replaces weeks or months of human effort. However, the problem of computing alignments is only one of a number of computational problems facing biologists. Many of these other tasks (such as searching a fairly large set of sequences for those that "are similar to" a given sequence) are both computationally intensive and occur often. We anticipate a situation arising within a few years in which thousands of such requests will have to be handled daily. In that environment, performance will definitely be an issue.

The kernel operations that we implemented in C were fairly limited. By far the most effort went into writing routines to compute critical subsequences, form pins, and to implement a dynamic programming algorithm. The latter routine is used to align two individual sequences in which no pins constrain the alignment.

5 Developing the Bilingual Program

The bilingual alignment program represented a development of an earlier program written entirely in C. In the process of developing the bilingual program, we refined both the implementation of the low-level routines and the top level algorithm. This refinement process was aided by the existence of a clear specification in a high level language.

A major concern when designing a bilingual program is to achieve a clean separation between the two layers of code. A first step in this direction is to identify additional abstract data types that must be implemented. In our case, a single data type *sequence* was required. A sequence is thought of as a string of characters that come from some specified organism and have an attached "location" number. For example:

aagcgc from homo sapiens at 1437

This might be used to represent a short sequence of genetic material from a human. If there are embedded indels, the location is thought of as applying to the first non-indel that occurs in the sequence. For example:

-aa-gcgc from homo sapiens at 1437

This might represent a sequence produced during an alignment; in this case, the "location" 1437 applies to the first "a" that occurs in the sequence. With these comments in mind, a sequence is composed of

1. An *identifier.*
2. A *location* that specifies the location in an input sequence of the first non-indel character.
3. A *length* (the number of characters in the sequence).
4. A *string* of characters that make up the sequence.

We found it convenient to represent a sequence as a tuple, with the general form:

$$\{Identifier, Location, Length, String\}$$

This made it possible to define most basic operations on sequences, such as "retrieve the length of a sequence", as *Strand* processes. We also needed to provide a small set of user-defined operations to perform other operations on sequences. These are described here in full to emphasize how few operations were required. The following set of user-defined operations performed basic manipulations.

read_sequence_set(*File,ListOfSequences*) reads a set of sequences from a file and constructs a list of sequences.

extract(*String,Start,Length,SubString*) assigns the substring of *String* specified by *Start* and *Length* to *SubString*.

char_in_sequence(*String,Disp,Char*) accesses a single character at a displacement of *Disp* into *String*.

We also required the following more complex functions that manipulate and create sequences:

combine_alignments(*Align1,Align2,Alphabet,Output,Distance*) produces an *Output* alignment of two existing alignments, using a specified *Alphabet*; also generates a measure of the *Distance* between the two alignments.

critical_points(*Seq,CPs*) determines the critical subsequences of a given sequence. *Seqs* specifies the input sequences, and *CPs* is assigned a list of critical subsequences.

form_pins(*CPList,PinList*) constructs a set of pins from the critical subsequences that occur in a set of sequences. *CPList* is a list, each element of which is a list of critical subsequences from a single sequence (as produced by *critical_points*). *PinList* is assigned a list of pins. Each pin is a list of critical subsequences from distinct input sequences.

glue(*ListOfAlignedChunks,ConcatenatedAlignment*) concatenates a set of alignments.

strip_indels(*Sequence,SequenceWithoutIndels*) removes indels from a sequence.

The implementation of these operations required about 2000 lines of C code. The size of this code may have to be expanded slightly, and we may well implement alternative versions of some of the critical operations. However, it appears likely that the bulk of future development will be in the *Strand* code, which currently consists of about 600-700 lines.

6 Using Multiprocessors

The ability to implement a database (and the assorted tools required to effectively utilize it) will hinge critically on exploitation of multiprocessors. Hence, we are extremely interested in investigating the effectiveness of *Strand* as a vehicle for coordinating a computation in a distributed environment.

Multiprocessors are likely to be important in genome projects in two ways. First, they may be used to perform searches against databases of thousands or millions of sequences. Second, they may be used to speed up particular time-consuming computations involving a small number of sequences. We envisage that both applications will be important: in a typical scenario, a scientist will perform a simple search against a large database to retrieve a small set of sequences and will then perform more sophisticated analyses on these sequences.

It is a fairly simple exercise to reduce database search times using a multiprocessor. In the absence of appropriate indexes, this type of search currently involves comparing a given sequence with each entry. Partitioning and distribution are hence straightforward [2]. Effective parallel execution of a single computationally-intensive program can be substantially more difficult. We may have little information about how best to define and construct subtasks; in addition, irregularities in the problem may lead to subtasks varying widely in number and size. Data-dependencies between tasks can also cause difficulties. We hence chose to investigate the use of *Strand* as a tool for executing a program of this latter type: our alignment program.

6.1 The *Strand* Program

Program 1 presents a small fragment of the *Strand* alignment program. This program implements the top level of the alignment algorithm presented in Section 2.2. User-defined operations are labeled.

The *align_chunk* process aligns a set of sequences (a **chunk**) by attempting to split the chunk using a *pin*. This splitting yields three chunks: left and right pinned chunks and an unpinned chunk. These are aligned independently and the three subalignments are combined to produce the complete alignment (R5). At each recursive call the algorithm computes critical points for each sequence in the chunk (R3,R4), forms a clean set of pins using *form_pins* and selects the best of these as the best pin (R2).

As noted previously, the alignment problem is sufficiently computationally intensive to benefit from parallel execution. For example, aligning a relatively small test data file required about 50 minutes on a single processor of an Encore Multimax. Furthermore, the divide-and-conquer strategy employed in our algorithm is naturally suited for parallel evaluation: each sub-alignment that results when a chunk is partitioned can potentially be performed on a different processor. How-

```
align_chunk(Chunk, AlignedChunk) :-                          % R1
        pins(Chunk, BestPin),
        divide(Chunk, BestPin, AlignedChunk).

pins(Chunk, BestPin) :-                                      % R2
        cps(Chunk, CpList),
        form_pins(CpList, PinList),                          % User
        choose_best_pin(Chunk, PinList, BestPin).

cps([Seq | Sequences], CpList) :-                            % R3
        CpList := [CPs| CpList1],
        critical_points(Seq,CPs),                            % User
        cps(Sequences, CpList1).
cps([], CpList) :- CpList := [].                             % R4

divide(Seqs, BestPin, Algnmnt) :-                            % R5
    BestPin ≠ [] |
        split(Seqs, BestPin, Left, Right, UnPinned),
        align_chunk(Left, LAlgnd),
        align_chunk(Right, RAlgnd),
        align_chunk(UnPinned,UnPAlgnd),
        combine(LAlgnd, BestPin, RAlgnd,UnPAlgnd, Algnmnt).
divide(Seqs, [], Algnmnt) :-                                 % R6
        basic_align_chunk(Seqs, Algnmnt).
```

Program 1 Alignment program top level.

ever, parallel execution is not straightforward because the alignment problem has an *irregular structure.* The number and size of processes created is totally data-dependent, cannot easily be predicted from the input data, and varies considerably from one problem to another.

We addressed these difficulties by implementing a *scheduler* that supports a load balancing strategy. Schedulers are typically used in the following manner [3]. An application-independent scheduler is implemented in *Strand*; the application program is then modified to permit it to execute in conjunction with the scheduler. The techniques we used to achieve this modification have some attractive qualities. We could have decided on a partitioning for our program and transformed it by hand. However, initial experiments convinced us that it was important to be able to rapidly express and evaluate a range of possible partitioning strategies. We hence developed *source-to-source transformation* tools that could be applied to a program to automatically generate a program that exploited a particular strategy. Two such tools were developed. The first requires the programmer to specify which processes are to be passed to the scheduler. The second determines this automatically using compile-time analyses. The design and implementation of these tools is beyond the scope of this report. However, the ease with which they could be developed is in our opinion a significant advantage of *Strand* technology.

6.2 The Scheduler

The load-balancing strategy that we used is based on the *manager-worker* model [1,3]. The scheduler consists of a central manager plus a set of workers. Each worker repeatedly obtains a unit of work (or **task**) from the manager and executes this work to completion; the manager allocates tasks to workers as required.

Our scheduler differs from other schedulers in two respects. First, it allows workers to contribute to the task pool maintained by the manager. Second, it allows for *data dependencies* between tasks. A data dependency exists between two tasks A and B if B requires data produced by A. As our scheduler requires that each worker execute only a single task at a time, deadlock can occur if data dependencies are not taken into account. For example, consider what happens in a system containing a single worker if task B is allocated to that worker before task A.

The process structure used to implement the scheduler is effectively a star with a *manager* and *filter* at the center and *workers* at the spokes. Each worker has a stream to the manager; the worker uses this to communicate requests for work. Each worker also has a stream to a filter, to which it can append contributions to the task pool. Finally, a stream links the filter and the manager.

Workers pass tasks to the filter in **bundles.** A bundle is a set of tasks, some of which may be nominated as dependent on others. The function of the filter process is to delay each task until other tasks on which it is dependent have completed

execution. The manager hence only receives tasks that can be executed immediately.

Strand implementations provide portability over a variety of physical architectures by supporting **virtual machines** such as ring and torus [3]. **Mapping annotations** permit the programmer to control the allocation of processes to nodes in a virtual machine; *Strand* systems provide embeddings of the virtual machine on particular physical architectures.

A manager-worker scheduler can naturally be executed on a ring virtual machine. The manager is created on the initial node; a worker is created on each successive node using the mapping annotation *@fwd*. Each worker is given a stream to the manager, thus creating the star topology. In outline:

```
-machine(ring).

scheduler(...) :-
        manager(...),
        filter(...),
        workers(...)@fwd.

workers(...) :-
        worker(...),
        workers(...)@fwd.
workers(...).
```

The complete scheduler program is given as Program 2. It exports a single process definition, *scheduler*. This has the form:

scheduler(Count, WorkerMod, FirstTask)

where *Count* is the number of workers to create, *WorkerMod* is the name of the module that contains the worker definition and *FirstTask* is an initial task to be placed in the task pool. The scheduler creates the filter and manager (R3) and spawns N workers around a ring (R7,8). The manager simply matches requests for work (R) with tasks (W), until no more work is available (R4-6).

The filter receives bundles of tasks from workers (R9). A bundle has the form:

{*Tasks, DependentTask, Done*}

The first component, *Tasks*, is a list of *immediate* tasks. These have no data dependencies and can hence be passed to the manager for immediate allocation. The second component, *DependentTask*, is a single *dependent* task: this is not be executed until all *Tasks* have been completely processed. The third component, *Done*,

```
-exports([scheduler/3]).                                        % R1
-machine(ring).                                                 % R2

scheduler(N, WMod, FirstTask) :-                                % R3
        filter(Wks, Ws1), merger(Ws1, Ws2),
        manager([{FirstTask, _} | Ws2], Rs),
        merger(Rs1, Rs), merger(Wks1, Wks),
        workers(N, WMod, Wks1, Rs1)@fwd.

manager([W | Wk], [R | Rs]) :- R := W, manager(Wk, Rs).         % R4
manager([], [R | Rs]) :- R := halt, manager([], Rs).            % R5
manager(_, []).                                                 % R6

workers(N, WMod, Ws, Rs) :-                                     % R7
    N > 0 |
        N1 is N - 1,
        Ws := [merge(W) | Ws1], Rs := [merge(R) | Rs1],
        WMod: worker(W, R),
        workers(N1, WMod, Ws1, Rs1)@fwd.
workers(0, _, Ws, Rs, _) :- Ws := [], Rs := [].                 % R8

filter([work(Wk, Dep, D) | In], Ss) :-                          % R9
        Ss := [merge(S1), merge(S2) | Ss1],
        forward(Wk, Vs, S1), await(Vs, Dep, D, S2),
        filter(In, Ss1).
filter([], Ss) :- Ss := [].                                     % R10

forward([P | Wk], Vs, Ss) :-                                    % R11
        Vs := [Term | Vs1], Ss := [{P, Term} | Ss1],
        forward(Wk, Vs1, Ss1).
forward([], Vs, Ss) :- Vs := [], Ss := [].                      % R12

await([done|Vs], T, D, Ss) :- await(Vs, T, D, Ss).              % R13
await([], T, D, Ss) :- T ≠ true | Ss := [{T, D}].               % R14
await([], true, D, Ss) :- Ss := [].                             % R15
```

Program 2: Manager-worker Scheduler.

is a variable to be assigned a value when *DependentTask*, and hence *Tasks*, have completed.

The *filter* process creates a *forward* process to pass each immediate task to the manager. A task is passed as a tuple of the form {*Task,Done*}, where *Done* is a variable to be assigned a value when the task is completed (R10,11). An *await* process is also created: this waits until the *Done* variables associated with immediate tasks are assigned values and then passes any dependent task to the manager (R13-15).

This concludes the presentation of the scheduler. It is important to note that the scheduler, although developed for the alignment program, is completely *application-independent.* It can be used to execute *any* program that adheres to its protocols, which can be summarized as follows.

1. A worker is defined by a process definition *worker* with two arguments, corresponding to a request stream and a work stream.
2. A worker generates a stream of variables representing requests for work units and accepts replies in the form {*Unit,Done*} or *halt.*
3. A worker either:
   (a) executes *Unit* to completion and then assigns the value *done* to the task's *Done* variable; or
   (b) executes part of the task and then links the task's *Done* variable with (one or more) bundles of new tasks, which it appends to the work stream.

6.3 The Transformation

A source-to-source transformation must be applied to the alignment program before it can be executed with our scheduler. The transformation takes the original program and constructs a new program capable of both generating and processing tasks. An important feature of this transformation is that it can easily be performed *automatically.*

The essential aspects of this transformation are demonstrated using a simple example. Consider this outline program:

```
p(...) :-
        a(...),
        b(...),
        c(...),
        d(...).

a(...).
```

$b(\ldots).$
$c(\ldots).$
$d(\ldots).$

Let us assume that the process p will be given to the scheduler as an initial task. We must decide which of the processes created by execution of p are to be dispatched for remote execution. Either programmer-supplied information or automatic analysis may be used to determine that after process a performs some initial computation, processes b and c are able to execute independently, and that when these processes terminate, d can execute. If b and c are judged sufficiently substantial, then this program is transformed to execute a locally and pass processes b, c and d to the scheduler. The process d is made dependent on b and c.

The result of transforming the example program is outlined in Program 3. The *worker* process starts by requesting a task (R1). It then repeatedly requests and processes tasks (R2,3) until told to halt (R4). A task is represented by a term of the form {Process,Done}. Program 3 shows the *worker1* rules that process requests to execute p and d tasks. For brevity, rules for b and c are not shown; they are similar to the d rule.

The process p is transformed to a process that executes the process a (R5) and then passes a bundle of tasks, containing processes b, c and d, to the scheduler (R6). Recall that a bundle has the form: $\{Tasks, DependentTask, Done\}$.

The transformed p process is invoked by the *worker1* rule that deals with p requests (R2). The variable *Done* associated with the p request is not assigned a value at this point, as the task has not been completed: instead, this variable is passed to the scheduler with the bundle. The scheduler will ensure that the *Done* variable is assigned a value only after the dependent process d has completed (at which point the other processes must also have completed).

The other processes, b, c and d, are simply augmented with a short circuit and executed directly. The short-circuit is used to detect termination and assign a value to the associated *Done* variable.

To illustrate the application of this transformation to the alignment program, we consider the following rule from Program 1:

$$
\begin{array}{ll}
divide(Ss, BP, Algmt) :- & \% \text{ R5} \\
\quad BP \neq [\,] \mid & \\
\qquad split(Ss, BP, L, R, UP), & \\
\qquad align\_chunk(L, LA), & \\
\qquad align\_chunk(R, RA), & \\
\qquad align\_chunk(UP, UnPA), & \\
\qquad combine(LA, BP, RA, UnPA, Algmt). &
\end{array}
$$

Assume that the three *align_chunk* processes are selected as likely-looking pieces of work. Transformation of this rule then yields the following rules:

$worker(Rs, Ws)$:- % R1
    $Rs := [R \mid Rs1]$,
    $worker1(done, Rs, Ws, R)$.

$worker1(done, Rs, Ws, \{p(\ldots), D\})$:- % R2
    $Rs := [R \mid Rs1]$,
    $p(\ldots, D, N, Ws, Ws1)$,
    $worker1(N, Rs1, Ws1, R)$.

⋮

$worker1(done, Rs, Ws, \{d(\ldots), D\})$:- % R3
    $Rs := [R \mid Rs1]$,
    $d(\ldots, done, D)$,
    $worker1(D, Rs1, Ws, R)$.

$worker1(done, Rs, Ws, halt)$:- $Rs := [], Ws := []$. % R4

$p(\ldots, D, N, Ws, Ws1)$:- % R5
    $a(\ldots, done, N)$,
    $p1(N, \ldots, D, Ws, Ws1)$.

$p1(done, \ldots, D, Ws, Ws1)$:- % R6
    $Ws := [\{[b(\ldots), c(\ldots)], d(\ldots), D\} \mid Ws1]$.

$a(\ldots, D, D1)$:- $D1 := D$. % R7
$b(\ldots, D, D1)$:- $D1 := D$. % R8
$c(\ldots, D, D1)$:- $D1 := D$. % R9
$d(\ldots, D, D1)$:- $D1 := D$. % R10

Program 3: A Transformed Program

$$
\begin{array}{l}
divide(Ss, BP, Algmt, Rs, Rs1, D, N) :- \qquad \% \text{ R5'} \\
\quad BP \neq [] \mid \\
\qquad split(Ss, BP, L, R, UP, N), \\
\qquad divide1(N, L, R, UP, BP, Algmt, Rs, Rs1, D).
\end{array}
$$

$$
\begin{array}{l}
divide1(done, L, R, UP, BP, Algmt, Rs, Rs1, D) :- \\
\quad Rs := [\{[align\_chunk(L, LA), \\
\qquad\qquad align\_chunk(R, RA), \\
\qquad\qquad align\_chunk(UP, UnPA)], \\
\qquad\qquad combine(LA, BP, RA, UnPA, Algmt), D\} \mid Rs1].
\end{array}
$$

The original rule spawns processes to align the left, right and unpinned chunks and to combine the aligned sections. In contrast, the transformed rules pass these processes to the scheduler, hence making them available for execution by other workers. The *combine* process is made dependent on the three *align_chunk* processes.

The transformation that we have developed also supports what we call *conditional dispatch.* The divide and conquer strategy adopted by the alignment program generates a large number of tasks. These rapidly become too small to be worthwhile distributing to other processors. We hence allow the programmer to specify a minimum size for task data. Tasks with data smaller than this minimum are executed locally.

In summary, the alignment program has to be transformed to take advantage of the manager-worker scheduler. We defined and implemented a source-to-source transformation that takes as input a a source program and either obtains from the programmer or infers which processes are to be dispatched as tasks. The transformation generates a new program that is capable of both generating and executing these tasks. Automation of the transformation process made it easy to experiment with alternative partitioning and scheduling strategies.

6.4 Performance Studies

Preliminary performance studies of the bilingual alignment program were conducted on an Encore Multimax. This machine consists of 20 National Semiconductor 32332 processors and 64 MBytes of shared memory, accessed using a high-speed bus. The *Strand* implementation employed for these experiments used the shared-memory simply to simulate message-passing. Hence the results of these performance studies can also be expected to apply to message-passing machines. The studies involved a single, relatively small test data file and two different transformations of the alignment program.

The test data file used for this investigation incorporated annotations provided by a biologist. These annotations permit the initial *align_chunk* problem to be immediately divided into a number of independent sub-alignment problems. Our

first attempt at transforming the alignment program for parallel execution only created a task for each such sub-alignment. Execution times on a varying number of processors are listed in Table 1. Except for $N = 1$, N processors were used to execute $N - 1$ workers and a single manager.

Table 1: Parallel Execution of the Alignment Program.

No. of Computers	Time (secs)
1	3360
2	2878
6	1020
11	859
16	578

The transformed program was not found to execute appreciably slower than the original program on a single processor. This is not surprising, as only a few additional process reductions are required to create and distribute work. However, the transformed program shows a maximum of only 5.4 times speedup on 16 processors.

Monitoring of the program was able to explain this result. The largest task generated was found to take 478 seconds to execute. In addition, the initial task which splits the data into subsequences and the terminating task which combines these subsequences take a total of 36 seconds. Hence the minimum execution time possible is 478 + 36 = 514 seconds: not much less than the best time of 578 seconds. These figures indicate that *ramp-down* (and to a lesser extent *ramp-up*) times are significant: processors are idle while the big task and terminating tasks are executing.

A second attempt at transformation sought to *reduce task size* by dispatching *align_chunk* and *combine* processes generated by the divide-and-conquer algorithm (Rule R5 in Program 1). This is essentially the transformation illustrated in Section 5.3. An initial experiment showed that dispatching *all align_chunk* tasks did not lead to performance gains. This was attributed to the creation of too many small tasks. Hence we chose to only dispatch tasks for which the size of the input chunk exceeded a certain threshold. This threshold, expressed in terms of the number and size of the sequences comprising the input chunk, was made a parameter of the program so as to permit experimentation with different values. Results obtained for various threshold values on 11 processors are summarized in Table 2. S is the number of sequences to be aligned; L is the length of the first sequence.

Table 2 shows that the number of tasks increases as the threshold is progressively decreased. The execution time first reduces and then increases as tasks become too small.

Using 16 processors, a best time of 350 seconds was achieved with S=5, L=50. This represents a speedup of 8.9: considerably better than that achieved using the

first transformation. We expect to achieve even better results on larger problems and using alternative transformations.

Table 2: Effect of Task Size on Run-time: 11 Processors.

S	L	No. of Tasks	Mean Task Time (secs)	Total Time (secs)
5	25	760	5.2	472
5	50	269	10.7	394
10	50	177	16.9	405
20	50	137	22.0	414
10	100	71	37.9	486
20	100	65	41.8	464

In summary, the manager-worker scheduler allowed us to achieve a significant reduction in execution time for a highly irregular problem. Two concepts helped us to achieve good performance: the recognition of data dependencies between tasks and the use of run-time tests on the size of input data to determine whether to dispatch tasks for remote execution.

It would be interesting to compare the performance of the bilingual program and an equivalent program written entirely in C. However, the *Strand* rewrite of the top levels of the original C program led us to improve the algorithm used. This reduced overall run-time and hence prevented comparison of the two programs.

7 Summary

It is our belief that computational problems from molecular biology may represent one of the most significant uses of computers during the coming decade. The problems posed in this area are frequently computation-intensive, and parallel computation may well prove to be required. Hence, we explored the potential use of *Strand* as a vehicle for achieving both ease of programming and with effective use of multiprocessing capabilities. By coding critical kernels in C, we achieved good performance.

The programming experiment reported in this report involved the coding of a novel sequence-alignment algorithm in *Strand* and C. The bilingual program was executed on a multiprocessor with encouraging results. A key feature of our approach was the use of a manager-worker scheduler to handle an irregular problem. Another was the use of a source-to-source transformation of an original program that generated tasks at compile-time. Our experience developing an effective parallel implementation of the alignment program emphasized the importance of a software environment that encourages exploratory programming.

Our experience with *Strand* has been gratifying. It has been possible to write bilingual programs that are clear, that perform well, and that can be executed on a range of available multiprocessing environments. We intend to continue our work by implementing a distributed database to support searches for genetic sequences that display similarities to a given sequence. *Strand* offers an attractive tool for implementing just such a system.

References

[1] Boyle, J. et al., *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, New York, 1987.

[2] Carriero, N. and Gelernter, D., Applications experience with Linda, *Proc. ACM/SIGPLAN PPEALS 1988*, 173-187.

[3] Foster, I.T. and Taylor, S., *Strand: New Concepts in Parallel Programming*, Prentice-Hall, Englewood Cliffs, N.J. 1989.

[4] von Heijne, G. *Sequence Analysis in Molecular Biology*, Academic Press, New York, 1987.

[5] Landau, G.M. and Vishkin, U. An efficient string matching algorithm with K substitutions for nucleotide and amino acid sequences. *Journal of Theoretical Biology*, 126, 483-490, 1987.

[6] Sankoff, D. and Kruskal, J.B. (Eds). *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, Mass., 1983.

[7] Watson, J.D, Hopkins, N.H., Roberts, J.W., Steitz, J.A., and Weiner, A.M. *Molecular Biology of the Gene (4th edition)*, The Benjamin/Cummings Publishing Co., Inc., Menlo Park, California, 1987.