COMPUTATIONAL BIOLOGY ON PARALLEL COMPUTERS

Srinivas Aluru *

Department of Electrical & Computer Engineering
Laurence H. Baker Center for Bioinformatics and Biological Statistics
Iowa State University
Ames, IA 50011, USA
aluru@iastate.edu

Abstract

Several applications in computational biology have large run-time and memory requirements either because of large data sizes or the inherent time and memory complexity of the underlying algorithms. Parallel computing is an effective way to address both these concerns - run-time can be reduced by the use of multiple processors to solve the same problem and the scaling of memory with processors enables the solution of larger problems than otherwise possible. In this paper, we describe efficient parallel solutions for three important applications in computational biology: 1) Computing alignments of large stretches of genomes, 2) clustering Expressed Sequence Tags and 3) Computing the accessible surface area of protein molecules. We report experimental results on a 64-processor IBM xSeries parallel computer. We conclude the paper by arguing that *parallel computational biology* is an important subdiscipline that merits significant research attention.

1 Introduction

The field of computational molecular biology is replete with applications that require processing large amounts of data. The basic problem of finding DNA sequences that exhibit homology to a given query sequence requires searching databases containing over tens of billions of nucleotides, and still growing at an exponential rate. The recent assembly of the mouse genome required processing over 33 million fragments of a total size of over 17 billion bases to assemble the genome of size over 3 billion bases. In comparative genomics, two or more genomes of such enormous sizes must be compared to discover common genes and interesting evolutionary relationships among species. In order to construct trees representing evolutionary relationships among species, algorithms explore a large search space of potential trees. Biomolecular simulations such as protein structure determination require a large number of iterations, making it important to accelerate the run-time per iteration. In these and many other applications, parallel processing can enable the solution of realistic problem instances.

In this paper, we present parallel solutions for three important problems in computational biology $-$ 1) Computing alignments of large stretches of genomes, 2) clustering Expressed Sequence Tags and 3) Computing the accessible surface area of protein molecules. For each problem, we describe the motivating biological application and why parallel processing is useful in solving the problem. We then present an efficient parallel algorithm to solve the problem and demonstrate its performance with experimental results on a 64-processor IBM xSeries parallel computer.

*Research supported by NSF under ACI-0203782 and NSF Career under CCR-0096288.

2 Syntenic Alignments

2.1 Problem Formulation

It is widely recognized that evolutionary processes tend to conserve genes. Along a chromosome, genes are interspersed by large regions with no known function. A gene itself is comprised of alternating regions known as *exons* and *introns*, and the introns are intervening regions that do not participate in the translation of a gene to its corresponding protein. Homologous DNA sequences from related organisms, such as the human and the mouse, are usually similar over the exon regions but different over other regions. Because the different regions are much longer than similar regions, conserved sequences cannot be identified through global alignment. This results in the problem of aligning two sequences where an ordered list of subsequences of one sequence is highly similar to a corresponding ordered list of subsequences from the other sequence. We refer to this problem as the *syntenic alignment* problem. This is an important computational problem in the emerging field of comparative genomics. Given two syntenic sequences of lengths m and n respectively, this problem can be solved in $O(mn)$ time. Because the sequences are large, parallel processing can enable the alignment of large syntenic regions.

2.2 Parallel Algorithm

An alignment of two sequences $S = s_1 s_2 \ldots s_k$ and $T = t_1 t_2 \ldots t_l$ over an alphabet Σ is obtained by inserting gaps in chosen positions and stacking the sequences such that each character in a sequence is either matched with a character in the other sequence or a gap. The quality of an alignment is computed as follows: A scoring function $f : \Sigma \times \Sigma \rightarrow I\!R$ specifies the score for matching a character in one sequence with a character in the other sequence. Gaps are penalized by using an affine gap penalty function that charges a penalty of $h + gr$ for a sequence of r maximal gaps. Here, h is referred to as gap opening penalty and g is referred to as gap continuation penalty. An optimal alignment of S and T is an alignment resulting in the maximum possible score over all possible alignments. Let $score(S, T)$ denote the score of an optimal alignment.

Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be two sequences. A subsequence A' of A is said to precede another subsequence A'' of A, written $A' \prec A''$, if the last character of A' occurs strictly before the first character of A'' in A. An ordered list of subsequences of A, $(A_1, A_2, \ldots, A_k)$ is called a chain if $A_1 \prec A_2 \prec \ldots A_k$. The syntenic alignment problem for sequences A and B is to find a chain $(A_1, A_2, \ldots, A_k)$ of subsequences

in A and a chain $(B_1, B_2, \ldots, B_k)$ of subsequences in B such that the score

$$\left\{\sum_{i=1}^{k} score(A_i, B_i)\right\} - (k-1)d$$

is maximized. The parameter d is a large penalty aimed at preventing alignment of short subsequences which occur by chance and not because of any biological significance.

We solve this problem by computing the syntenic alignment between every prefix of A and every prefix of B. We compute 4 tables C, D, I and H of size $(m+1) \times (n+1)$. Entry $[i, j]$ in each table corresponds to the optimal score of a syntenic alignment between $a_1 a_2 \ldots a_i$ and $b_1 b_2 \ldots b_j$, subject to the following conditions: 1) In C, a_i is matched with b_j, 2) In D, a_i is matched with a gap, 3) In I, gap is matched with b_j, and 4) In H, either a_i or b_j is part of an unmatched subsequence.

It follows from these definitions that the tables can be computed using the following recurrence equations:

$$\begin{aligned} C[i,j] &= f(a_i, b_j) \\ &\quad + \max\{C[i-1,j-1],\ D[i-1,j-1], \\ &\qquad I[i-1,j-1],\ H[i-1,j-1]\} \end{aligned}$$

$$\begin{aligned} D[i,j] &= \max\{C[i-1,j] - g',\ D[i-1,j] - g, \\ &\qquad I[i-1,j] - g',\ H[i-1,j] - g'\} \end{aligned}$$

$$\begin{aligned} I[i,j] &= \max\{C[i,j-1] - g',\ D[i,j-1] - g', \\ &\qquad I[i,j-1] - g,\ H[i,j-1] - g'\} \end{aligned}$$

$$\begin{aligned} H[i,j] &= \max\{C[i-1,j] - d,\ I[i-1,j] - d, \\ &\qquad C[i,j-1] - d,\ D[i,j-1] - d, \\ &\qquad H[i-1,j],\ H[i,j-1]\} \end{aligned}$$

where $g' = (g + h)$.

Prior to computation, the top row and left column of each table should be initialized. These initial values can be directly computed. After computing the tables, the optimal score of a syntenic alignment is given by the maximum score in $C[m,n]$, $D[m,n]$, $I[m,n]$, or $H[m,n]$. Thus, the problem can be solved in $O(mn)$ time and space. If we draw links from each table entry to an entry which gives the maximum value in equation (1), (2), (3) or (4), the optimal syntenic alignment can be retrieved by tracing backward in the tables starting from the largest $[m,n]$ entry and ending at $C[0,0]$. Using the now standard technique of space-saving, introduced originally by Hirschberg [5], the space required can be reduced to $O(m+n)$, while increasing the run-time by at most a factor of 2.

Let p denote the number of processors, with id's ranging from 0 to $p-1$. Without loss of generality, assume that $m \leq n$. We compute the four tables C, D, I and H together in parallel. We use a columnwise decomposition to partition the tables to the processors. For simplicity, assume m and n are multiples of p. Processor i receives columns $i\frac{n}{p}+1, \ldots, (i+1)\frac{n}{p}$ of each table, and is responsible for computing the table entries allocated to it. The tables are computed one row at a time, in the order C, D, H and I.

Consider computing the i^{th} row of the tables. The recurrence relation for D uses entries from the already computed $(i-1)^{th}$ row and in the same column. These are readily available on the same processor. In computing C, entries that are in the previous row and previous column are needed. These are available on the same processor, except in the case of the first column assigned to each processor. After computing the $(i-1)^{th}$ row, each processor sends the last entry it computed in each of the four tables to the next processor. This is sufficient to compute the next row of C, and requires communicating just four entries per processor irrespective of the problem size. Next, we compute the i^{th} row of H. Let

$$\begin{aligned} v[j] &= max\{C[i-1,j] - d,\ I[i-1,j] - d, \\ &\qquad C[i,j-1] - d,\ D[i,j-1] - d, \\ &\qquad H[i-1,j]\} \end{aligned}$$

Because the i^{th} rows of C and D are already computed, the vector v can be computed directly in parallel using the information available within each processor. Then, $H[i,j]$ can be written as $\max\{v[j], H[i,j-1]\}$. It is easy to see that the computation of $H[i,j]$ can be done using the *parallel prefix*[1] operation with '$\max$' as the binary associative operator.

Now, let us turn to the computation of the i^{th} row of table I.

$$\begin{aligned} \text{Let} \quad & w[j] = \max\{C[i,j-1],\ D[i,j-1], \\ & \qquad\qquad H[i,j-1]\} - g' \\ \text{Then,} \quad & I[i,j] = \max\{w[j], I[i,j-1] - g\} \end{aligned}$$

$$\begin{aligned} \text{Let} \quad x[j] &= I[i,j] + gj \\ &= \max\{w[j] + gj,\ I[i,j-1] + gj - g\} \\ &= \max\{w[j] + gj,\ I[i,j-1] + g(j-1)\} \\ &= \max\{w[j] + gj,\ x[j-1]\} \end{aligned}$$

$$\begin{aligned} \text{Let} \quad & z[j] = w[j] + gj \\ \text{Then,} \quad & x[j] = \max\{z[j], x[j-1]\} \end{aligned}$$

Since the $z[j]$'s can be easily computed from the i^{th} row of C, D, and H, $x[j]$'s can be computed using parallel prefix with '$\max$' as the binary associative operator. In turn, $I[i,j]$ can be computed from $x[j]$ by simply subtracting gj from it.

As mentioned before, processor i is responsible for computing columns $i\frac{n}{p}+1$ through $(i+1)\frac{n}{p}$ of the tables C, D, I and H. Distribution of sequence B is trivial because b_j is needed

[1] Given $x_1, x_2, \ldots, x_n$ and a binary associative operator $\otimes$, parallel prefix is the problem of computing $s_1, s_2, \ldots, s_n$, where $s_i = x_1 \otimes x_2 \otimes \ldots \otimes x_i$ (or equivalently, $s_i = s_{i-1} \otimes x_i$). Its run-time is $O\left(\frac{n}{p} + \log p\right)$. This is a well-known primitive operation in parallel computing, and is readily available on most parallel computers. For example, the function MPI_Scan computes parallel prefix.

only in computing column j. Therefore, processor i is given $b_{i\frac{n}{p}+1} \ldots b_{(i+1)\frac{n}{p}}$. Each a_i is needed by all the processors at the same time when row i is being computed. Sequence A is stored in each processor.

To summarize, each processor computes $\frac{n}{p}$ entries per row of each of the four tables. The run-time is dominated by parallel prefix, which takes $O\left(\frac{n}{p} + \log p\right)$ time. To achieve optimal $O\left(\frac{n}{p}\right)$ run-time, the number of processors used should be $O\left(\frac{n}{\log n}\right)$. To enable using as large a number of processors as possible, and more importantly because practical efficiencies are better when the problem size per processor is large, we choose the larger sequence to represent the columns of the table (i.e., $n \geq m$). The parallel run-time for computing all the tables is $O\left(\frac{mn}{p}\right)$, optimal with respect to the sequential algorithm. The space required is also $O\left(\frac{mn}{p}\right)$. The algorithm can be improved to reduce the memory usage to $O\left(m + \frac{n}{p}\right)$ with a parallel traceback capability to retrieve the actual alignment, and the details are omitted here for lack of space.

2.3 Experimental Results

We implemented the parallel syntenic alignment algorithm in C and MPI and experimentally evaluated its performance using an IBM xSeries cluster. The cluster consists of 64 Pentium processors each with a clock rate of 1.26GHZ and 512MB of main memory, connected by Myrinet, supporting peak bidirectional communication rates of 2Gb/sec. To study the scalability of the algorithm, the program is run using sequences of the same length and varying the number of processors. Note that the communication required in computing a row depends only on the number of processors and is independent of the problem size. Thus, it is interesting to determine the smallest problem size per processor (*grain-size*) that gives good scaling results. This can be used to calculate the largest number of processors that can be beneficially used to solve a given problem. On the IBM cluster, we determined that the grain-size required for efficient parallel execution is about $500 - 1000$ per processor.

The speedups as a function of the number of processors for a syntenic alignment of two sequences of length $30,000$ are shown in Figure 1. Notice that superlinear speed up is observed in several cases, due to the typical beneficial effect of caching. On 16 processors, each processor has an approximate row size of $2,000$ entries per table. We need to store 4 tables, 2 rows per table, and need 3 memory words (12 bytes) per entry. Thus, the memory required in the problem decomposition stage is 192KB per processor, which will nicely fit into the 256KB cache. On 8 processors, the rows will have to be continually swapped between cache and main memory, causing significant slowdown.

The program is used to compare two syntenic human and mouse sequences containing 17 genes [2]. The human sequence is of length $222,930$ bp (GenBank Accession U47924)

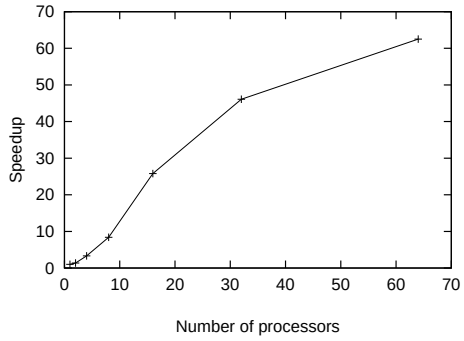


Figure 1: Speedup as a function of the number of processors for syntenic alignment of two sequences of length $30,000$.

and the mouse sequence is of length $227,538$ bp (GenBank Accession AC002397). The following parameters are used based on our prior experiences with standard alignment programs: match $= 10$; mismatch $= -20$; gap opening penalty, $h = 60$; gap continuation penalty, $g = 2$. A value for the parameter d was selected on the basis of internal exon lengths, often of length at least 50 bp. The score of 50 matches at 10 per match is 500. The value of 300 was used for the parameter d. The human and mouse sequences were screened for repeats with RepeatMasker [9]. The masked sequences are then used as input. The program produced a syntenic alignment of the two sequences in 23.32 minutes on 64 processors. The alignment consists of 154 ordered subsequence pairs separated by unmatched subsequences. The alignment fully displays the similar regions but omits most of the dissimilar regions. The 154 similar regions are mostly coding exon regions and untranslated regions. Gaps occur much more frequently in alignments of untranslated regions than in alignments of coding exon regions. The total length of the 154 similar regions is $43,445$ bp and their average identity is 79%. The 154 similar regions constitute about 19% of each of the two sequences.

3 EST Clustering

3.1 Problem Formulation

Gene is a contiguous stretch of genomic DNA that encodes the information necessary for building a protein. The first step in protein production process is transcription, in which a copy of the gene is made on an RNA molecule known as pre-mRNA. Genes are composed of alternating segments called *exons* and *introns*. The introns are spliced out from the pre-mRNA and the resulting molecule is called *mRNA*. The mRNA is later used as a template for building a protein. Molecular biologists capture such mRNA and convert it to the corresponding DNA molecule, known as complementary DNA, or *cDNA* for short. Due to the limitations of the experimental processes involved and due to breakage of sequences in chemical reactions, several cDNAs of various lengths are obtained instead of just full-length cDNAs. Part of the cDNA fragments of average length about $500 - 600$ can be sequenced. The sequencing can be

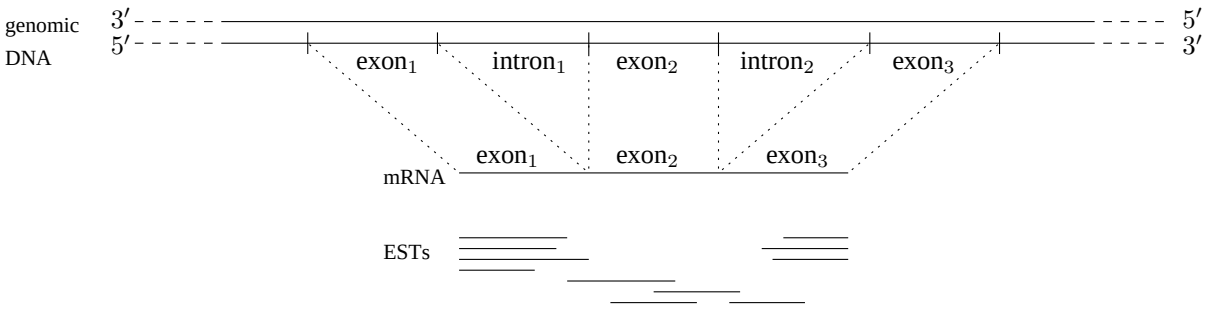


Figure 2: A simplified diagrammatic illustration of genomic DNA, mRNA and ESTs.

done from either end. The resulting sequences are called *ESTs* (Expressed Sequence Tags). For a simplified diagrammatic illustration, see Figure 2.

An EST collection will contain ESTs from expressed genes in proportion to their expression levels. **the EST clustering problem** is to partition the ESTs into clusters such that ESTs from each gene are put together in a distinct cluster. A wide range of biological applications require EST clustering including gene identification, studies of gene expression and differential gene expression, identification of disease-causing single nucleotide polymorphisms (SNPs) and the design of microarrays. A repository of ESTs collected from various organisms is maintained at the National Center for Biotechnology Information (http://www.nicb.nlm.nih.gov/dbEST). At present, there exists a two orders of magnitude gap between the largest EST data sets and the size of data set that can be clustered with serial EST clustering software. The human EST collection is over 4.9 million and the mouse EST collection is over 3.6 million currently, and these data sets are continually growing.

3.2 Parallel Algorithm

The fundamental information available for EST clustering is the potential overlaps between ESTs expressed from the same gene. A naive solution would take each pair of ESTs and compute if they have significant overlap. This can be done by using a dynamic programming algorithm, similar to sequence alignment discussed in the previous section. Once significant pairwise overlaps are determined, this knowledge can be used to perform EST clustering. The run-time of this approach is $O(n^2k^2)$, where n is the number of ESTs and k is the average length of an EST, which is prohibitively expensive for large n.

In order to reduce the run-time, the following approach is used: Because the percentage of positions where two overlapping sequences differ is small (allowing both for sequencing errors and naturally occurring variations), any overlapping region must also contain significantly long exact matching regions. Thus, a filter is applied to determine pairs of ESTs that share an exact matching region (common substring) of length at least a threshold value. The dynamic programming algorithm is run only on those pairs, which we refer to as *promising pairs*. Experimentation with current software indicates that pairwise alignment using dynamic programming is the run-time intensive part and generation of promising pairs is the memory intensive part. The worst case scaling of both the memory and run-time with the number of ESTs is quadratic.

We developed a solution that 1) reduces the memory requirement from quadratic to worst-case linear, 2) reduces the number of pairwise alignments without affecting quality of clustering, and 3) employs parallel processing to enable clustering of larger EST data sets. The approach is explained in detail below: Initially, each EST can be thought of as a cluster by itself. Two EST clusters can be merged provided an EST from each cluster can be identified that show strong overlap using the pairwise alignment algorithm. This process is continued until no further merges are possible. If a pair of identified ESTs does not show strong overlap, the corresponding clusters cannot be merged, and the effort in testing is wasted. However, there may be another pair of ESTs from these clusters that may have strong overlap, causing the clusters to merge when this pair of ESTs is aligned. The order in which promising pairs are processed does not affect the final set of clusters formed. However, the order does have significant influence on the run-time it takes to compute the clusters, as explained below.

Significant savings in run-time can be achieved by fast identification of pairs that would likely yield a positive outcome when the pairwise alignment algorithm is run. A positive outcome helps in merging of two clusters. As a result, it is no longer necessary to test pairs of ESTs where each is drawn from one of the two clusters. Hence, by early identification of promising pairs of ESTs that cause clusters to merge, it becomes unnecessary to align many promising pairs generated at later stages. Thus, instead of merely finding all pairs that meet certain test criteria (such as sharing a substring of length 20 or more), we generate pairs in decreasing order of overlap quality, as measured by an efficiently computable measure. As the measure, we use maximal common substring length. A maximal common substring of a pair of sequences is a substring common to both the sequences that cannot be extended at either end to result in a longer match. The rationale for using the measure is that pairs of ESTs with larger length exact matches are more likely to pass the alignment test. To eliminate the large memory required for storing the promising pairs, we designed an on-demand algorithm that remembers its state and produces

the next set of pairs as and when required. We also address the important problem of avoiding generation of the same pair multiple times, even though it is nontrivial to do so because we do not store previously generated pairs. Our algorithm uses the generalized suffix tree (GST) data structure [4].

The organization of our software is as follows: We first build a distributed representation of the generalized suffix tree data structure in parallel. This data structure is used for on-demand generation of promising pairs in decreasing order of maximal common substring length. The pair generation itself is done in parallel. Maintaining and updating of the EST clusters is handled by a single processor, which acts as a master processor directing the remaining processors to both generate batches of promising pairs and perform pairwise alignment on promising pairs. It is not mandatory to perform pairwise alignment of each generated pair because the current set of EST clusters may obviate the need to do so. Hence, the master processor is also responsible for the selection of pairs to be aligned and is a necessary intermediary between pair generation and alignment. In order to reduce communication overhead, the master processor dispatches the selected pairs in batches of size batchsize, a configurable parameter. To provide an added degree of flexibility in balancing the load, we do not require that a pair generated on a processor be allocated to the same processor if a pairwise alignment is needed.

The main idea behind the on-demand pair generation algorithm is the following: A suffix tree of all the ESTs is a compacted trie of all the suffixes of all the ESTs. An internal node in a suffix tree corresponds to a substring common to all the ESTs represented as leaves in the subtree of the node. The length of this common substring is called the string depth of the node in the suffix tree. In order to general pairs in decreasing order of maximal common substring length, we sort all the internal nodes in decreasing order of string depth and process them in that order. While at a particular node, pairs of ESTs where each is drawn from one of the leaves in the node's subtree are generated. A number of algorithmic strategies are developed to avoid duplicate generation of promising pairs. The key here is that we report the presence of a maximal common substring for a pair directly from the suffix tree, without having to look at all the smaller length non-maximal common substrings contained within it.

3.3 Experimental Results

We ran our software on *Arabidopsis thaliana* ESTs using different numbers of processors. The total run-time as a function of the number of processors is shown in Figure 3(a). As can be observed, the run-times show near perfect scaling with the number of processors. The total number of promising pairs generated and the number of these pairs assigned for pairwise alignment as a function of the data size are shown in Figure 3(b). This clearly explains the reduction in run-time achieved as a consequence of generating the promising pairs in decreasing order of maximal common substring length, as opposed to the traditional way of generating them in an arbitrary order. As an illustration of the capability of our software to solve large problems, we clustered 327,632 rat ESTs on 64 processors in under 47 minutes. The preprocessing phase took about 15 minutes while the clustering phase took about 32 minutes. This problem is beyond the computational capabilities of any current serial software. We estimate that the human EST collection can be easily clustered using a 256 processor system with 1GB of memory per processor.

4 Protein Accessible Surface Area Computation

4.1 Problem Formulation

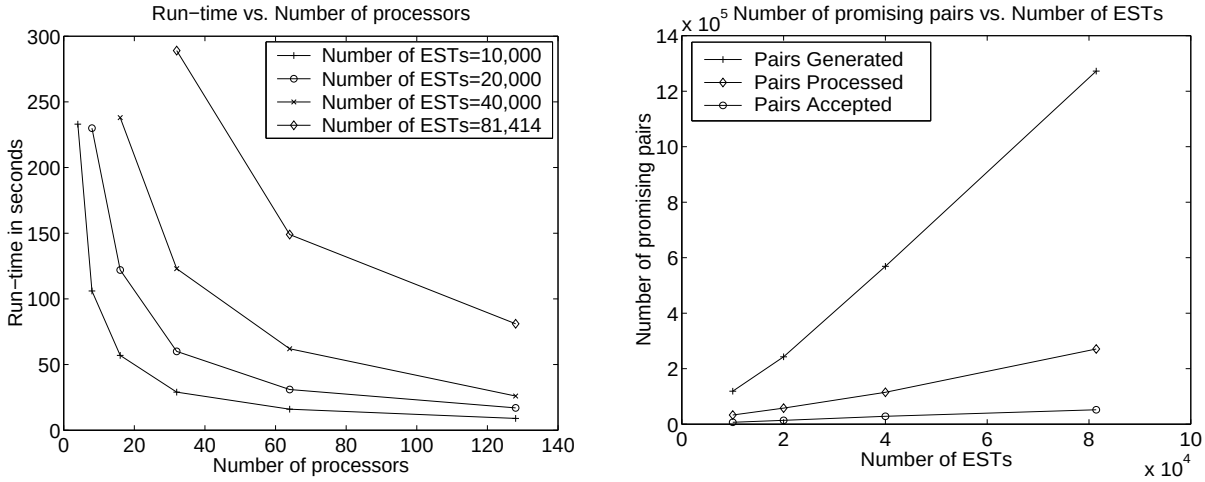
The accessible surface area of a protein molecule is the cumulative surface area of the individual atoms that is accessible to a solvent molecule. Two atoms of the protein may be close enough that the solvent molecule cannot access the surface area of the atoms completely. The atoms and the solvent molecules are modelled as spheres using their van der Waals' radii. The problem can be further simplified by reducing the solvent molecule to a point and increasing the van der Waals' radii of all the atomic spheres by the van der Waals' radius of the solvent molecule. The problem is now abstracted as: Given n spheres, find the total surface area of the spheres that is accessible, i.e. the surface area that does not lie inside any other sphere.

Protein ASA computation is used in computational methods for protein folding, for estimating the interaction free energy of a protein with solvent, in studies on protein stability and protein-protein interactions. Because proteins are relatively small, with a typical protein containing only several hundred amino acids, it may appear that the problem is not large enough for parallel computation. However many applications require ASA calculation for many structures with the ASA calculation for one structure influencing the next structure. Thus, parallel computation of the ASA can significantly speed up such computations. A noteworthy example of such an application is the protein structure determination using molecular dynamics simulation, where ASA is one of the terms used in energy calculation.

4.2 Parallel Algorithm

Protein ASA computation can be decomposed into two parts: Firstly, the set of spheres that intersect each sphere have to be determined. Secondly, we have to find the accessible surface area of each sphere, knowing the spheres that intersect it. For the first part, a naive algorithm checks all pairwise intersections and takes $O(n^2)$ time. As the spheres in ASA calculations correspond to atoms in proteins, prior information is available which can be used to devise more efficient algorithms. For instance, the atoms are of a few types (such as Carbon, Nitrogen etc..) whose radii information is known.

Number the atoms (and the corresponding spheres) $1, 2, \ldots, n$ and let r_i denote the radius and $C_i = (x_i, y_i, z_i)$ denote the center of the i^{th} atom. Let r be the radius of the solvent



(a) (b)

Figure 3: The graph to the left shows parallel run-times for EST clustering as a function of the number of processors. The number of generated pairs, pairs which are aligned, and pairs that pass the alignment test are shown in the graph to the right.

molecule. Let $r_{max} = max\{max_{i=1}^{n}\{r_i\}, r\}$ and $r_{min} = min\{min_{i=1}^{n}\{r_i\}, r\}$. From the knowledge of atoms that occur in amino acids and taking the solvent to be water, r_{max} is 2Åand r_{min} is 1.2Å. It is reasonable to assume that the ratio of r_{max} to r_{min} is bounded by a small constant. By a simple geometric argument, it is easy to see that the number of spheres intersecting a given sphere must be a constant. Thus, the total number of sphere intersections is bounded by $O(n)$.

Without loss of generality, let $[0, X_{max}] \times [0, Y_{max}] \times [0, Z_{max}]$ be the smallest parallelepiped containing the protein and that X_{max}, Y_{max} and Z_{max} are all divisible by $4r_{max}$. This is the domain relevant to computation of the accessible surface area. Consider an implicit partitioning of the domain into cubes of side length $4r_{max}$. Each cube can be identified with an integer 3–tuple (u, v, w), where $(u*4r_{max}, v*4r_{max}, w*4r_{max})$ is the corner of the cube having the minimum x, y and z co-ordinates. We say a cube contains a sphere if it contains the center of the sphere. The main idea behind the decomposition is that all the spheres that intersect a given sphere lie either in the cube containing the sphere or in one of its 26 neighboring cubes.

Use an array A of size n stored in a distributed manner over the p processors such that processor P_i has $A[i\frac{n}{p} \ldots (i+1)\frac{n}{p} - 1]$. Entry $A[i]$ stores the i^{th} sphere (C_i, r_i) and the cube (u_i, v_i, w_i) containing the center C_i. Note that, the latter can easily be calculated given the former as input, by each processor in $O(\frac{n}{p})$ time. The algorithm then sorts A in parallel using cube 3-tuples as keys. As a result, spheres whose centers are contained in the same cube now occupy consecutive positions in A. Each processor will now compute the ASA of the spheres stored in its section of sorted A. The general idea behind sorting is to let the same processor handle all spheres whose centers belong to the same cube, to potentially reduce the communication required. For large n, sorting A requires $O(\frac{n \log n}{p})$ parallel time.

To compute its portion of the ASA, P_i requires information about spheres intersecting any sphere stored at P_i. To accomplish this task efficiently, the following strategy is adopted: Build an auxiliary array B of size $2p$ by gathering from each processor the 3-tuple of the cubes containing the centers of its first and last spheres. Processor P_i then composes a message for each processor, requesting the information needed from it, as follows: for each distinct cube c at P_i, compute the set of cubes, D, whose spheres may intersect spheres in cube c. For each $d \in D$, use a binary search in B to identify all the processors P_j that contain spheres from d. Append cube d to the request message being sent to all such P_j's. The duplication in the request message can be easily avoided (for example, by using local sorting). All requests from all processors are communicated using one *all-to-all* communication. For each cube request that a processor P_j receives, it retrieves the spheres in this cube using a binary search in its portion of A. In this fashion, it composes an answer message for each processor. All answers are communicated using a single *all-to-all* communication.

Each processor now concatenates, in an array C, all the information received and its own information according to the processor ids. This ensures that each processor has all the required spheres arranged in sorted order according to their cube 3-tuples. Now each processor can calculate the set of spheres intersecting each of the spheres it is responsible for by doing binary searches in array C. Since the size of the array C is $O(\frac{n}{p})$ and the total number of spheres that might potentially be tested for intersection with each sphere is bounded by a constant, the total computation time required in this step is $O(\frac{n}{p} \log \frac{n}{p})$. Putting together the run-times from the various components of the algorithm above, it is clear that the total runtime for the algorithm is dominated by sorting and is $O(\frac{n \log n}{p})$.

Any ordering of the cube 3-tuples can be used to sort array A. Irrespective of the ordering, the algorithm has the same optimal worst-case performance. However, the practical performance of the algorithm can be improved significantly by employing a distribution that preserves locality. Suppose we use an order-

Protein Molecule	Number of atoms	Total surface area (in $\mathring{A}^2$)	$m = 200$ ASA (in $\mathring{A}^2$)	$m = 200$ Confidence that error is $\leq 1\%$	$m = 2000$ ASA (in $\mathring{A}^2$)	$m = 2000$ Confidence that error is $\leq 1\%$
DNAse I	4945	584107	25578	81%	25584	98%
HIV-I RT	11647	1377740	67930	94%	67996	99%
DNA helicase	13284	1564523	74610	93%	74792	99%
Cyclooxygenase-2	17896	2123492	86145	94%	86292	99%

Table 1: Information about the four proteins used for testing. Also shown are the estimated accessible surface areas and error bounds.

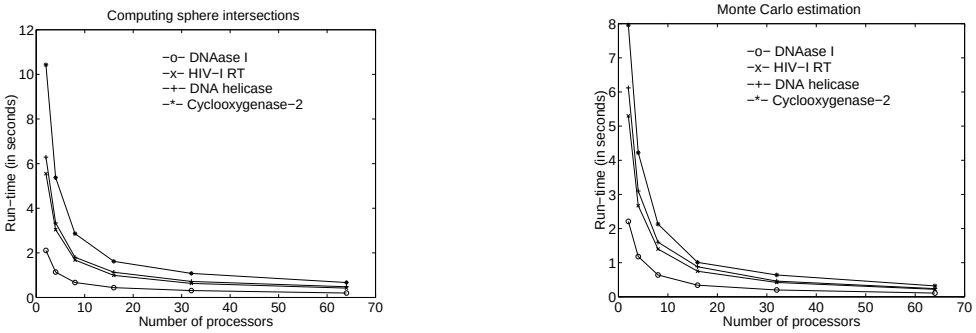


Figure 4: The left and the right graphs show the run-times in seconds for computing sphere intersections and Monte Carlo ASA estimation, respectively, as a function of the number of processors for the four proteins used in testing. A sample size of 200 is used.

ing that attempts to assign spheres in nearby cubes to the same processor. With such an ordering, most of the spheres that intersect spheres in a processor's portion of sorted A are found on the same processor. This reduces computation, communication and storage in the subsequent steps of the algorithm. We achieve this by using a Z-space filling curve [7]. Space-filling curves are mappings from multidimensional points with integer coordinates to a one-dimensional ordering. The path implied in the multidimensional space by the linear ordering, i.e. the sequence in which the multidimensional points are visited according to the linear ordering, forms a non-intersecting curve.

We now turn to the computation of the ASA itself. Once all the spheres intersecting a given sphere are known, the ASA of that sphere can be computed. The total ASA is simply the sum of the ASAs of the individual spheres. Thus, this phase is trivially parallelizable. Several methods have been proposed to compute the ASA of an individual sphere including numerical integration [6], a direct analytical method [8], using predetermine sample points on a sphere to estimate surface area [10], and a Monte Carlo method based on random sampling [3].

4.3 Experimental Results

We implemented the parallel algorithm for ASA computation, using the Monte Carlo method for ASA estimation. To generate random points on the surface of a sphere for the Monte Carlo method, a random value between $-r$ and $+r$ (r denotes the radius of the sphere under consideration) is selected for the z-coordinate, followed by a random angle between $-\pi$ and $+\pi$. The software is written in C and MPI, and run using an IBM SP-2 containing 4-way SMP nodes. It is evaluated using the following four proteins, whose structure data is taken from the Protein Data Bank (http://www.pdb.org): DNAse complex I with actin (1ATN), HIV-1 Reverse Transcriptase (2HMI), Gp4D helicase From Phage T7 (1E0K) and uninhibited mouse cyclooxygenase-2 (5COX). The PDB identifiers of each protein are given in parenthesis next to it. For convenience, these proteins will be referred to as DNAse I, HIV-I RT, DNA helicase and cyclooxygenase-2, respectively. As is standard practice, hydrogen atoms are ignored in computing the ASA. Table 1 summarizes the protein data used in testing, their total surface areas and the estimated accessible surface areas using sample sizes of 200 and 2000. For each sample size, the minimum probability that the estimated ASA is within 1% of the correct ASA is shown.

To evaluate the run-time performance of the software, we consider the two stages: 1) Computing sphere intersections, and 2) Monte Carlo ASA estimation. This is because a different ASA estimation algorithm can be substituted, if desired. The total run-times for each of the two stages as a function of the number of processors for the four proteins are shown in Figure 4. As can be observed, both stages exhibit very good scaling. For the Monte Carlo estimation, a sample size of 200 is used. For

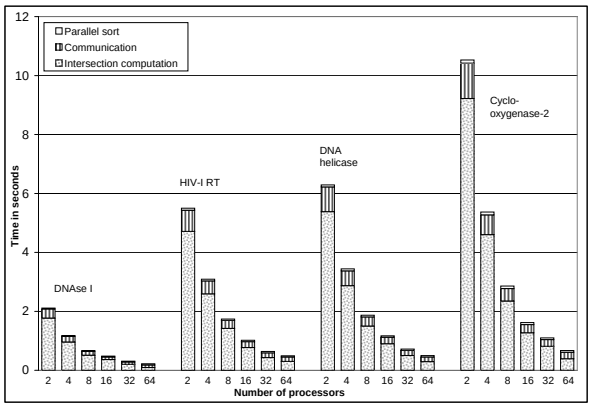


Figure 5: Run-times of the three components of our algorithm for computing sphere intersections.

this sample size, the run-time spent in Monte Carlo estimation is about the same as the run-time spent in computing sphere intersections. The run-time for Monte Carlo estimation is linear in the sample size.

The three components of the run-time for computing sphere

intersections, namely, a) parallel sorting, b) communication and incorporation of the received communication to locally contain information about spheres intersecting allocated spheres, and c) intersection computation, are shown in Figure 5. The run-time is dominated by the intersection computation. Of the three components, intersection computation shows near perfect scaling, communication shows good scaling and parallel sorting shows poor scaling. The poor scaling of parallel sorting is due to the small size of the problem (number of atoms to be sorted) relative to the number of processors. However, parallel sorting is a negligible percentage of the run-time, and hence does not affect the overall scaling. The run-time of the Monte Carlo method shows near perfect scaling for all four proteins. Note that this need not be the case because the actual number of intersecting spheres may vary considerably from sphere to sphere, even though the number of spheres intersecting each sphere is bounded by a large constant. Our experimental results indicate that allocating equal number of spheres per processor is sufficient to ensure near perfect load balancing. Adjusting the load based on the number of intersecting spheres of each sphere is unlikely to yield any benefit, especially in light of the overhead that will be incurred in doing so.

5 Conclusions

In this paper, we present efficient parallel algorithms for three important problems in computational biology and demonstrate their effectiveness through experimental results on a 64-processor parallel computer. As data sizes continue to grow and more computationally demanding biological questions are being asked, parallel processing is becoming more and more important in computational biology. While numerical simulations have long driven the need for high performance parallel computers, we believe that life sciences applications will become equally important in the design, development and application of high performance parallel computers.

References

[1] S. Aluru, N. Futamura and K. Mehrotra, Biological sequence comparison using prefix computations, *Proc. International Parallel Processing Symposium* (1999) 653-659.

[2] M.A. Ansari-Lari, J.C. Oeltjen, S. Schwartz, Z. Zhang, D.M. Muzny, J. Lu, J.H. Gorrell, A.C. Chinault, J.W. Belmont, W. Miller and R.A. Gibbs, Comparative sequence analysis of a gene-rich cluster at human chromosome 12p13 and its syntenic region in mouse chromosome 6, *Genome Research, 8* (1998) 29-40.

[3] N. Futamura, S. Aluru and D. Ranjan, Efficient parallel algorithms for solvent accessible surface area of proteins, *IEEE Transactions on Parallel and Distributed Systems, 13(6)*, (2002) 544-555.

[4] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, Cambridge, London, 1997.

[5] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, *Communications of the ACM, 18(6)* (1975) 341-343.

[6] B. Lee and F.M. Richards, The interpretation of protein structures: Estimation of static accessibility, *Journal of Molecular Biology, 55* (1971) 379-400.

[7] G.M. Morton, A computer oriented geodetic data base and a new technique in file sequencing, IBM Technical Report, Ottawa, Canada (1966).

[8] T.J. Richmond, Solvent accessible surface area and extended volume in proteins – Analytical equations for overlapping spheres and implications for the hydrophobic effect, *Journal of Molecular Biology, 178* (1984) 63-89.

[9] A. Smit and P. Green, http://ftp.genome.washington.edu/RM/RepeatMasker.html, 1999.

[10] A. Shrake and J.A. Rupley, Environment and exposure to solvent of protein atoms, Lysozyme and Insulin, *Journal of Molecular Biology, 79* (1973) 351-371.