Space and Time Efficient Parallel Algorithms and Software for EST Clustering

Anantharaman Kalyanaraman, *Student Member*, *IEEE*, Srinivas Aluru, *Senior Member*, *IEEE*, Volker Brendel, and Suresh Kothari

Abstract—Expressed sequence tags, abbreviated as ESTs, are DNA molecules experimentally derived from expressed portions of genes. Clustering of ESTs is essential for gene recognition and for understanding important genetic variations such as those resulting in diseases. In this paper, we present the algorithmic foundations and implementation of $PaCE$, a parallel software system we developed for large-scale EST clustering. The novel features of our approach include 1) design of space-efficient algorithms to limit the space required to linear in the size of the input data set, 2) a combination of algorithmic techniques to reduce the total work without sacrificing the quality of EST clustering, and 3) use of parallel processing to reduce runtime and facilitate clustering of large data sets. Using a combination of these techniques, we report the clustering of 327,632 rat ESTs in 47 minutes, and 420,694 *Triticum aestivum* ESTs in 3 hours and 15 minutes, using a 60-processor IBM xSeries cluster. These problems are well beyond the capabilities of state-of-the-art sequential software. We also present thorough experimental evaluation of our software including quality assessment using benchmark *Arabidopsis* EST data.

Index Terms—Computational biology, EST clustering, maximal common substring, parallel algorithms, suffix tree applications.

1 The EST Clustering Problem

DNA is a double helix composed of four different types of nucleotides, denoted by A, C, G, and T. For computational purposes, it can be considered as a string over the alphabet $\Sigma = \{A, C, G, T\}$. The term *genome* refers to the complete set of all DNA molecules (chromosomes) found in each cell of an organism. Certain contiguous stretches along genomic DNA, known as *genes*, encode the information for building proteins. The process of *transcription* produces a copy of a gene as an RNA molecule, called the *premessenger RNA*, or *pre-mRNA* for short. Genes in *eukaryotic* organisms (includes plants and animals) 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 essentially contains the coded recipe for manufacturing the corresponding protein. Because of the intron/exon phenomenon, the term *genomic DNA* is used for the entire gene and the term *complementary DNA*, *cDNA* for short, is used for DNA molecules that are artificially manufactured using the mRNA as a template. Due to the limitations of the experimental processes involved and due to a 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 nucleotides can be sequenced. The sequencing can be done from either end. The resulting sequences are called *ESTs* (Expressed Sequence Tags). A simplified diagrammatic illustration is shown in Fig. 1. In practice, EST sequences can contain errors, due to the nature of experiments involved in deriving and sequencing them.

An EST database consists of ESTs drawn from multiple cDNAs, and there could be potentially many ESTs drawn from each cDNA. Given such a database, *the EST clustering problem* is defined as follows: The ESTs should be partitioned into clusters such that ESTs from each gene are put together in a distinct cluster. A further complication arises due to the fact that DNA is a double stranded molecule and a gene could be part of either strand. The two strands are related according to the following nucleotide pairings: $A \leftrightarrow T$ and $C \leftrightarrow G$. Each strand has a directionality as well, with the two ends identified as $5'$-end and $3'$-end, respectively. It is customary to write a DNA molecule as the sequence of nucleotides of one of its strands from the $5'$-end to the $3'$-end. The two strands of a DNA have opposite directionality. Thus, the sequence of one strand can be obtained from the other using a *reverse complementation* operation, where complementation refers to substituting according to the pairing $A \leftrightarrow T$ and $C \leftrightarrow G$. For example, if one strand is represented by $ATGACCT$, then the other strand is $AGGTCAT$, and both are representations of the same DNA. mRNA is a single stranded molecule and the corresponding cDNA is obtained by converting it to its corresponding double stranded molecule.

EST clustering is an actively pursued problem of current interest [4], [6], [11], [13], [14], [16], [21]. The motivation for developing efficient parallel EST clustering software stems from the wide range of current and future biological

- *A. Kalyanaraman is with the Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, IA 50011. E-mail: ananthk@iastate.edu.*
- *S. Aluru is with the Department of Electrical and Computer Engineering, Iowa State University, 2215 Coover Hall, Ames, IA 50011. E-mail: aluru@iastate.edu.*
- *V. Brendel is with the Department of Zoology and Genetics and the Department of Statistics, Iowa State University, 2112 Molecular Biology Building, Ames, IA 50011. E-mail: vbrendel@iastate.edu.*
- *S. Kothari is with the Department of Electrical and Computer Engineering, Iowa State University, 3214 Coover Hall, Ames, IA 50011. E-mail: kothari@iastate.edu.*

Manuscript received 7 June 2002; revised 21 Apr. 2003; accepted 17 May 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 116726.

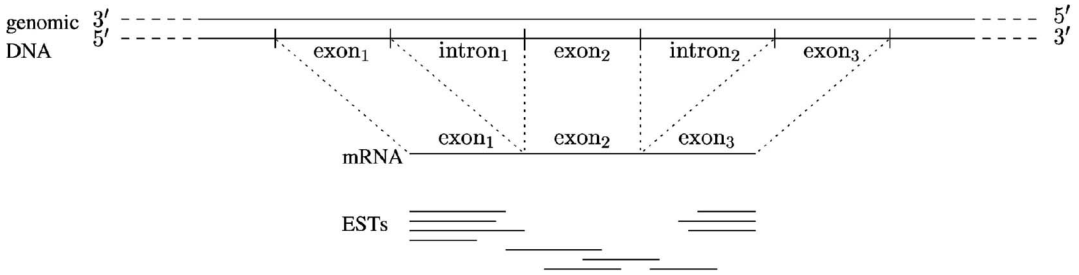


Fig. 1. A simplified diagrammatic illustration of genomic DNA, mRNA, and ESTs.

applications that require EST clustering and the pervasive nature of such applications in furthering knowledge in modern molecular biology. Some important biological applications of EST clustering are highlighted here, as a motivation for the work presented in the remainder of this paper:

- **Gene Identification:** The importance of current and recently completed projects in sequencing the genomes of various organisms cannot be overemphasized. However, this is only a step toward the goal of identifying genes and finding the functions of the corresponding proteins. ESTs provide the necessary clues to gene identification.
- **Gene Expression Studies:** In EST sequencing, genes that are expressed more will result in more ESTs. Thus, the number of ESTs in a cluster can be used to estimate the level of expression of the corresponding gene.
- **Differential Gene Expression:** ESTs collected from various organelles of an organism (such as leaf, root, and shoot of a plant) reveal the expression levels of genes in the respective organelles and provide clues to their possible function.
- **SNP Identification:** The same gene is present in slight variations, known as *alleles*, among different members of the same species. Many of these alleles differ in a single nucleotide, and some of these differences are the cause of genetic diseases. ESTs from multiple members of a species help identify such disease causing Single Nucleotide Polymorphisms, or SNPs.
- **Design of Microarrays:** Microarrays, also called DNA chips, are a recent discovery allowing gene expression studies of thousands of genes simultaneously. ESTs can be used in designing microarrays to detect the level of expression of the corresponding genes.

A repository of ESTs collected from various organisms is maintained at the National Center for Biotechnology Information (http://www.ncbi.nlm.nih.gov/dbEST). With the number of ESTs for some organisms approaching several millions, we believe that parallel processing is essential to cluster such large collections of ESTs.

The rest of the paper is organized as follows: In Section 2, we describe current practices in EST clustering and outline their limitations. The algorithmic ideas underlying the development of our time and space efficient parallel software PaCE are described in Section 3. Sections 4, 5, and 6 contain details of our parallel algorithms. Detailed runtime analysis and a brief assessment of quality of PaCE clustering are presented in Section 7. For a more thorough discussion on the quality assessment, using the software, and results of clustering various plant species using PaCE, see [10]. Section 8 concludes the paper.

2 Existing Approaches and Their Evaluation

The primary information available to cluster ESTs is the potential overlaps between ESTs drawn from the same gene. Many software programs currently used for clustering ESTs were developed to solve a related problem known as *fragment assembly* [5]. Fragment assembly is used to discover long stretches of genomic DNA from the sequences of several small fragments of it and is used for genome assembly. Once again, the assembly is based on detecting overlapping fragments, making the software useful for EST clustering as well [14]. Fragment assembly software will actually assemble ESTs from the same gene into full length cDNAs (ideally), or into contiguous stretches of cDNAs (also called *contigs*).

The overlap between two sequences that possibly contain errors can be computed by a pairwise alignment algorithm using dynamic programming; this method is accepted to be a good measure of overlap quality [18], [19], [23]. This algorithm takes time proportional to the product of the lengths of the sequences, and is expensive to run for all pairs of ESTs. A similar measure for computing overlap is the $d2\text{-}distance$ [26], which is used in the $d2\_cluster$ clustering program [2], [3]; the distance is computed for all pairs of sequences before the clusters are formed. To avoid considering all pairs, fragment assembly programs use approximate overlap detection algorithms to perform fast identification of pairs of sequences with the potential for good quality overlap. Pairwise alignments are then computed only for such identified pairs, referred to as *promising pairs* henceforth.

The most popular software tools used for EST clustering are Phrap (http://www.phrap.org), CAP3 [8], and TIGR Assembler [24], all originally designed for fragment assembly. Recently, researchers at The Institute for Genome Research (TIGR) evaluated the quality of EST clustering generated by the three programs and found that CAP3 produces the least number of erroneous clusters [14]. We tested each of the three programs using a single processor of an IBM xSeries node so that the runtimes can be compared easily with that of our parallel software. The results of the test runs performed on various subsets of a benchmark data

TABLE 1
Runtimes (in Minutes) of TIGR Assembler, Phrap, and CAP3 on Different Portions of the *Arabidopsis* Benchmark Data Set

Input (n)	TIGR Assembler		Phrap		CAP3	
	Run-time	PMU	Run-time	PMU	Run-time	PMU
50,000	321	1.3 GB	38	1.07 GB	72	711 MB
100,000	1,114	2.23 GB	85	1.17 GB	150	1.93 GB
168,200	X	X	224	2.2 GB	X	X

"PMU" denotes peak memory usage and "X" denotes that 2.25 GB memory was not sufficient to run the software.

set consisting of 168,200 *Arabidopsis thaliana* ESTs are shown in Table 1. With an available memory of 2.25 GB, Phrap was the only program to complete execution on 168,200 ESTs, and all the three programs ran out of memory for larger data sets we tested, such as 327,632 rat ESTs. The worst-case runtime and space complexity of each of these programs is quadratic in the number of ESTs, and the actual runtime and space required are highly sensitive to the specific problem instance. For the fragment assembly problem, the input consists of fragments from the target DNA sequence sampled uniformly at random. This has the effect of making the number of promising pairs linear in the size of the target sequence. However, for EST clustering, the number of ESTs per gene is highly nonuniform, resulting in a potentially quadratic number of promising pairs. This causes fragment assembly software to require quadratic space when applied to EST clustering. The motivation behind our work is to overcome this memory bottleneck, while also reducing the runtime for clustering.

Contemporaneous to our effort, Pertea et al. have developed a parallel EST clustering tool called $TGICL$ [20]. The algorithm computes overlaps for all pairs of ESTs using a modified version of *megablast* [29].

3 Space and Time Efficient EST Clustering

The main contribution of our research is a parallel software system named $PaCE$ (for Parallel Clustering of ESTs) that clusters ESTs that come from the same gene or a set of duplicated genes. Once clustered, the contigs corresponding to ESTs from each cluster are generated using fragment assembly software. This combined procedure of clustering using PaCE followed by contig assembly using fragment assembly software enables clustering and assembly of large-scale EST data because: 1) the memory required by our clustering algorithm is linear in the size of the input and 2) the largest input size to the assembly is now reduced from the entire data set to the size of the largest cluster generated by PaCE. In other words, the potential bottleneck arising because of the quadratic memory requirements of fragment assembly software when applied to large data sets is overcome.

Experimentation with fragment assembly software indicates that generation of promising pairs is the memory-intensive phase, and pairwise alignment of these pairs is the runtime intensive phase. Promising pairs are typically defined to be those pairs of ESTs that have a common substring of length greater than or equal to a threshold value. In the worst-case, the number of promising pairs is quadratic in the number of ESTs. Moreover, a procedure often used in identifying such pairs is as follows: Let w denote the threshold length. Create $|\Sigma|^w$ buckets to represent all possible strings of length w. For each EST sequence, consider each substring of length w and store the sequence number in the corresponding bucket. In this approach, every substring of length $l > w$ that is common to any pair of ESTs is represented in $l - w + 1$ buckets, making its detection take time proportional to its length.

We developed the following alternative approach to EST clustering. 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 a strong overlap using the pairwise alignment algorithm. This merging process is continued until no further merges are possible. If a pair of identified ESTs do not show a strong overlap, the corresponding clusters cannot be merged and the effort in testing the pair is wasted. However, it may still be the case that the two clusters should be merged and our choice of the pair does not reflect that.

We achieve significant savings in runtime by early identification of pairs that would likely yield a positive outcome when the pairwise alignment algorithm is run. A positive outcome helps in two ways: First, it causes merging of two clusters. Second, it is no longer necessary to test pairs of ESTs where each is drawn from one of the two clusters. Thus, instead of merely finding all pairs that meet certain test criteria (such as sharing a substring of length 20 or more), we are interested in defining a suitable measure and producing the promising pairs in decreasing order of quality according to the measure.

A simple measure for predicting the quality of overlap between a pair of strings is the length of a maximal common substring. A $maximal\ common\ substring$ of a pair of strings is a substring common to both the strings that cannot be extended at either end to result in a longer match. We seek an efficient algorithm with minimal memory requirements that produces promising pairs in decreasing order of maximal common substring length. To minimize the memory required, we developed an on-demand algorithm that remembers its state and produces the next set of pairs when requested. As will become evident later, an important problem is to avoid generation of the same pair multiple times, even though it is not possible to check for duplicates because the pairs are not stored.

The organization of PaCE and the interactions among its various components are depicted in Fig. 2. We first build in parallel, a distributed representation of the generalized

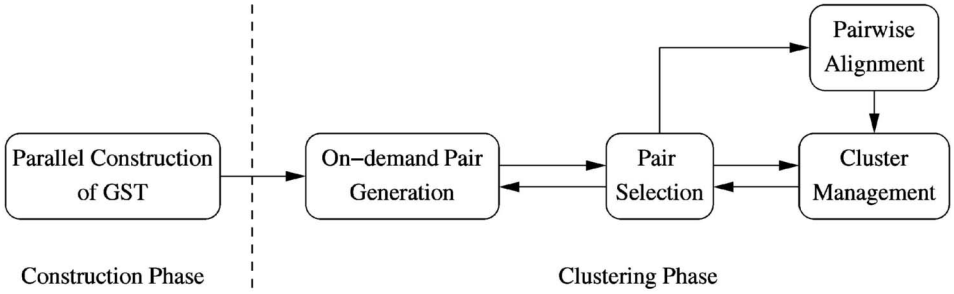


Fig. 2. Organization of PaCE software.

suffix tree data structure [5]. This data structure is used for on-demand generation of promising pairs in decreasing order of maximal common substring length. The pair generation is also done in parallel, and the algorithm is such that each processor needs only access to the portion of the suffix tree stored locally. 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. To provide an added degree of flexibility in balancing the load, we do not require that a pair generated on a slave processor be allocated to the same processor for pairwise alignment. Our algorithms for each of the components of PaCE are described in the following sections.

4 Parallel Construction of Generalized Suffix Tree

Let s be a string of length m over alphabet Σ. A suffix tree for $s\$$ is a directed tree with m-leaves numbered 1 through m [5]. Each internal node has at least two children. Each edge is labeled with a nonempty substring of s. No two edges leaving an internal node have labels beginning with the same character. The *path-label* of a node v, denoted $path\text{-}label(v)$, is the concatenation of the edge labels on the path from root to v. The *string-depth* of v, also denoted as $string\text{-}depth(v)$, is the number of characters in its path-label. The tree has the property that the path-label of the leaf labeled i is the suffix starting at position i of s. A Generalized Suffix Tree (GST) for a set of n strings is a suffix tree constructed using all suffixes of the n strings. If N is the total number of characters in all the n strings, the GST has at most N leaves, exactly N leaf labels, $O(N)$ size, and can be constructed in $O(N)$ time [5]. An example of a GST for two strings is shown in Fig. 3.

We use the following notation throughout the remainder of the paper: Let n be the number of ESTs and the set $\mathcal{E} = \{e_1, e_2, \ldots, e_n\}$ denote the ESTs. The total length of all the ESTs is denoted by N. Let l be the average length of an EST, i.e., $l = \frac{N}{n}$. Because of the double stranded nature of DNA, each EST and its reverse complement must be considered. Let $\mathcal{S} = \{s_1, s_2, \ldots, s_{2n}\}$ denote the $2n$ strings such that $s_{2i-1} = e_i$ and $s_{2i} = \bar{e_i}$, where $\bar{e_i}$ denotes the reverse complement of e_i.

We perform a parallel construction of the GST for $\mathcal{S}$. Parallel algorithms for construction of suffix trees using the CRCW/CREW PRAM model are presented in [1], [7]. Due to the unrealistic assumptions underlying the PRAM model with respect to accessing remote memory, a direct implementation of these algorithms is unlikely to be practically efficient. Algorithms for parallel generation of suffix arrays, a data structure closely related to suffix tree, are available for the distributed memory model [12], [17]. We developed an algorithm that directly generates a distributed representation of GST on the distributed memory model. Also, we took advantage of the fact that the average length l

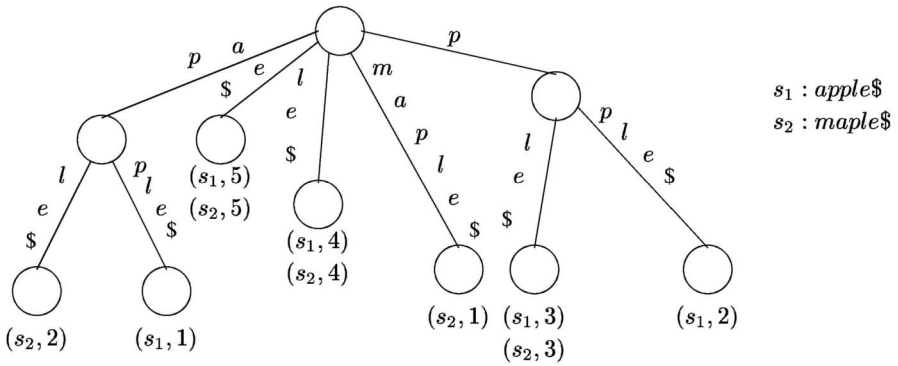


Fig. 3. Generalized suffix tree for the strings $apple$ and $maple$. (s_i, k) denotes the suffix of string s_i starting at position k. "\$" is a unique termination character.

of an EST is a fixed number $(500\text{-}600)$ irrespective of the number of ESTs.

Initially, the set $\mathcal{E}$ of ESTs is partitioned across p processors such that the sum of the lengths of the EST sequences assigned to each processor is approximately $\frac{nl}{p}$. Each processor then partitions the set of all suffixes of its ESTs and their reverse complements into $|\Sigma|^w$ buckets based on the w-length prefix of each suffix. This is done through a linear scan of the ESTs and their reverse complements. Through a parallel summation on the bucket sizes (in $O(\log p)$ communication steps), each processor computes the total number of suffixes in each bucket over all processors. Subsequently, the buckets are distributed in parallel such that 1) each bucket is assigned to one processor, and 2) the total number of suffixes over all buckets assigned to each processor is as close to $\frac{2nl}{p}$ as possible. The value of w should be carefully chosen. While a small value may result in too few buckets to ensure load balancing, large values may result in loss of some potential overlapping pairs.

Once a bucket is assigned to a processor, the latter constructs the tree for all suffixes in the bucket. A sequential suffix tree construction algorithm [15], [27], [28] cannot be used for this purpose, because all suffixes of a string need not be present in the same bucket, unless the string is a repetition of a single character. Our approach is a depth-first construction of the tree corresponding to each bucket. First, partition all suffixes in a bucket into at most $|\Sigma|$ subbuckets, based on their first characters. This partitioning is then applied recursively on each subbucket until each suffix is separated from others. As this involves scanning each character of every suffix at most once, the runtime is $O(\frac{nl^2}{p}) = O(\frac{Nl}{p})$. This algorithm is practically efficient because 1) l is relatively small and is independent of n and 2) once buckets are assigned, each processor computes the trees for its buckets without any communication. Note that the resulting set of trees over all processors represents a distributed collection of subtrees of the GST for $\mathcal{S}$, except for the top portion consisting of nodes with string-depth $< w$.

Because of the concern for space-efficiency, each tree is stored as follows: The nodes are generated and stored in the order of the depth-first search traversal of the tree. In addition to the string-depth, each node contains a single pointer to the rightmost leaf node in its subtree. All the children of a node can be retrieved using the following procedure. The first child of a node is stored next to it in the array. The next sibling of a node can be obtained by following the pointer to its rightmost leaf and taking the node in the next entry of the array. If a node and its parent have identical rightmost leaf pointers, the node has no next sibling. A leaf is one whose rightmost leaf pointer points to itself.

5 On-Demand Pair Generation

We define *promising pair* to be a pair of strings which have a maximal common substring of length at least equal to a threshold value ψ. The goal of the on-demand pair generation algorithm is to report on-the-fly, in the decreasing order of maximal common substring length, the set of promising pairs. Ideally, each such pair should be generated only once, because a reported pair is evaluated for alignment obviating the need to generate it again. However, this may require storing of the pairs generated so far. As a trade off, we generate, at no additional storage cost, a promising pair at most as many times as the number of distinct maximal substrings common to the pair. The algorithm operates on the following idea: If two strings share a maximal common substring α, then the leaves corresponding to the suffixes of the strings starting with α will be present in the subtree of the node with path-label α. Thus, the algorithm can generate the pair at that node.

A substring α of a string is said to be $left\text{-}extensible$ (alternatively, $right\text{-}extensible$) by character c if c is the character to the left (alternatively, right) of α in the string. If the substring is a prefix of the string, then it is said to be left-extensible by λ, the null character. Let $subtree(v)$ represent the set of nodes present in the subtree of a node v (including itself). Let β be a suffix of a string and $leaf(\beta)$ denote the leaf whose path-label is β. Let $leaf\text{-}set(v) \subseteq \mathcal{S}$ represent the set of strings that have a suffix β such that $leaf(\beta) \in subtree(v)$. We compute a partitioning of the $leaf\text{-}set(v)$ of a node v into five sets, $l_A(v)$, $l_C(v)$, $l_G(v)$, $l_T(v)$, and $l_\lambda(v)$. These sets are referred to as $lsets(v)$. If a string is in $l_c(v)$ (for $c \in \Sigma \cup \{\lambda\}$), then it has a suffix β such that $leaf(\beta) \in subtree(v)$ and β is left-extensible by c. Observe that such a partition need not be unique because a string s could have two suffixes β and β' such that $leaf(\beta)$ and $leaf(\beta')$ both are in $subtree(v)$ and β is left-extensible by c_i and β' is left-extensible by c_j, for $c_i \neq c_j$. Then, s could be either in $l_{c_i}(v)$ or $l_{c_j}(v)$. Any of these partitions will work for the pair generation algorithm.

The algorithm for generation of pairs is given in Fig. 4. Recall that each processor has a collection of subtrees that represents a portion of the GST of $\mathcal{S}$. The nodes in local subtrees with string-depth $\geq \psi$ are sorted in decreasing order of string-depth, and processed in that order. The $lsets$ at leaf nodes are computed directly from the leaf labels. Because of appending each string with $\$ \notin \Sigma$, multiple labels at a leaf must necessarily be from different strings. Hence, there is no need to find and eliminate duplicates in forming these $lsets$.

The set of pairs generated at node v is denoted by P_v. If v is a leaf, a Cartesian product of each of the $lsets$ at v corresponding to A, C, G, T, λ, with every other $lset$ of v corresponding to a different character is computed. In addition, a Cartesian product of $l_\lambda(v)$ with itself is computed. The union of these Cartesian products is taken to be P_v.

If v is an internal node, the $lsets$ of the children of v are traversed to eliminate multiple occurrences of the same string in the $lsets$ of different children of v. Note that the $lsets$ at a child of v may no longer represent a partition of the $leaf\text{-}set$ of the child. After the elimination, a Cartesian product of each $lset$ corresponding to A, C, G, T, λ of each child of v, with every other $lset$ corresponding to a different character in every other child node is computed. In addition, a Cartesian product of the $lset$ corresponding to λ of each child node with each of the $lsets$ corresponding to λ of every other child node is computed. The union of these Cartesian products is taken to be P_v. The $lset$ for a particular character at v is obtaining by taking a union of the $lsets$ for the same character at the children of v. Because of the elimination of multiple occurrences, the $lsets$ at v constitute a partition of $leaf\text{-}set(v)$.

Algorithm 1 *Pair Generation*

GeneratePairs(Forest of local GST subtrees with roots of string-depth $< \psi$)

1. Compute the string-depth of all nodes in local GST subtrees.
2. Sort nodes with string-depth $\geq \psi$ in decreasing order of string-depth.
3. For each node v in that order

   IF v is a leaf THEN

   ProcessLeaf(v)

   ELSE

   ProcessInternalNode(v)

ProcessLeaf(Leaf: v)

1. Compute

$$P_v = \bigcup_{(c_i,c_j)} l_{c_i}(v) \times l_{c_j}(v),\ \forall (c_i, c_j) \text{ s.t., } c_i < c_j \text{ or } c_i = c_j = \lambda$$

ProcessInternalNode(Internal Node: v)

1. Traverse all $lsets$ of all children $u_1, u_2, \ldots, u_q$ of v. If a string is present in more than one $lset$, all but one occurrence of it are removed.
2. Compute

$$P_v = \bigcup_{(u_k,u_l)} \bigcup_{(c_i,c_j)} l_{c_i}(u_k) \times l_{c_j}(u_l),\ \forall (u_k, u_l),\ \forall (c_i, c_j) \text{ s.t., } 1 \leq k < l \leq q,\ c_i \neq c_j \text{ or } c_i = c_j = \lambda$$

3. Create all $lsets$ at v by computing :

   For each $c_i \in \Sigma \cup \{\lambda\}$

$$l_{c_i}(v) = \bigcup_{u_k} l_{c_i}(u_k),\ 1 \leq k \leq q$$

Fig. 4. Algorithm for generation of promising pairs.

Traversing $lsets$ of all child nodes to eliminate multiple occurrences of a string is implemented to run in time proportional to the sum of the cardinalities of those $lsets$. A global array of size $2n$, indexed by string id number, is maintained. Let v be an internal node being processed. When a string is encountered in an $lset$ at a child node of v, the entry in the array for this string is checked to see if it is marked v. If not, the array entry is marked v. If it is already marked, the occurrence of this string from this $lset$ is removed. A linked list implementation of the $lsets$ allows the union in $Step\ 3$ of $ProcessInternalNode$ to be computed using $O(|\Sigma|^2)$ concatenation operations. At this point, the $lsets$ at the internal node's children are removed. This limits the total space required for storing $lsets$ to $O(N)$, linear in the size of the input.

A pair generated at a node v is discarded if the string corresponding to the smaller EST id number is in complemented form. This is to avoid duplicates such as generating both (e_i, e_j) and $(\bar{e_i}, \bar{e_j})$, or generating both $(e_i, \bar{e_j})$ and $(\bar{e_i}, e_j)$ for some $1 \leq i, j \leq n$. Thus, without loss of generality, we denote a pair by (s, s'), where $s = e_i$ and s' is either e_j or $\bar{e_j}$ for some $i < j$. The relative orderings of the characters in $\Sigma \cup \{\lambda\}$ and the child nodes avoid generation of both (s, s') and (s', s) at the same node.

In summary, if v is a leaf,

$$P_v = \{(s, s') \mid s \in l_{c_i}(v), s' \in l_{c_j}(v), c_i, c_j \in \Sigma \cup \{\lambda\}, ((c_i < c_j) \vee (c_i = c_j = \lambda))\},$$

and if v is an internal node,

$$P_v = \{(s, s') \mid s \in l_{c_i}(u_k), s' \in l_{c_j}(u_l), c_i, c_j \in \Sigma \cup \{\lambda\}, k < l, ((c_i \neq c_j) \vee (c_i = c_j = \lambda))\}.$$

The following lemmas are intended to prove the correctness and runtime characteristics of the algorithm:

Lemma 1. *Let v be a node with path-label α. A pair (s, s') is generated at v only if α is a maximal common substring of s and s'.*

Proof. At a leaf node v, if the algorithm generates a pair (s, s'), it is because the strings are either from $lsets$ representing different characters or from the $lset$ representing λ. In either case, α is a maximal common substring of s and s'.

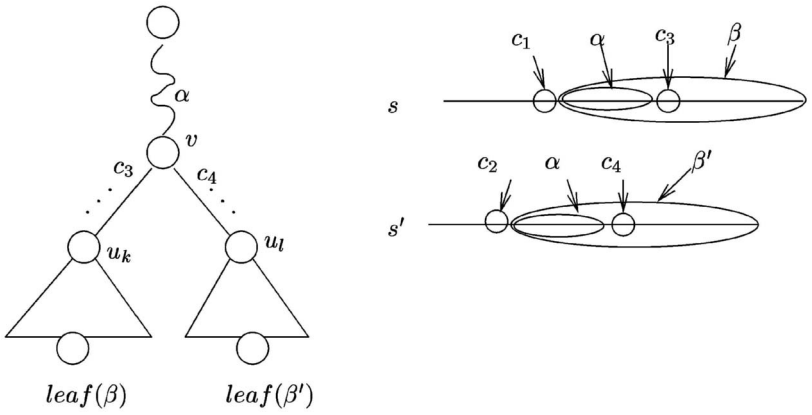


Fig. 5. Illustration of the proof of Lemma 1 for an internal node v with path-label α. β and β' are suffixes of s and s', respectively, $c_1, c_2 \in \Sigma \cup \{\lambda\}$ and $c_3, c_4 \in \Sigma \cup \{\$\}$.

For an internal node v, the algorithm generates a pair (s, s') only if 1) s and s' are from either $lsets$ representing different characters or $lsets$ representing λ, and 2) s and s' are from $lsets$ of two different children of v. The former ensures α is not left-extensible; the latter ensures α is not right-extensible. Thus, α is a maximal common substring of s and s'. Fig. 5 illustrates the proof for the case of an internal node. □

Note that the converse of Lemma 1 need not hold because the elimination of multiple occurrences of strings, while processing an internal node may remove the corresponding occurrences that would otherwise lead to the generation of a pair. This could happen only in cases where a maximal common substring of the pair is contained in another maximal substring common to the same pair.

Corollary 1. *The number of times a pair is generated is at most the number of distinct maximal common substrings of the pair.*

Proof. Follows directly from Lemma 1 and the fact that a pair is generated at a node at most once. The latter is true because for any internal node, the algorithm retains only one occurrence of a string before generating pairs, and for any leaf there can be at most one occurrence of any string in its $lsets$. While this bounds the maximum number of times a pair is generated, a pair may not be generated as many times. □

Lemma 2. *A pair (s, s') is generated at least once if it has a maximal common substring of length $\geq \psi$, where ψ is the threshold value.*

Proof. Consider α, a largest maximal substring of length $\geq \psi$ common to strings s and s'. As α is maximal, there exists either a leaf v with path-label α or an internal node v with path-label α. Also, there exist suffixes β and β' of s and s', respectively, that belong to $subtree(v)$ and that have α as a prefix, which is neither left-extensible nor right-extensible by the same characters in both s and s'. Thus, if α is the path-label of a leaf, then s and s' will be present in the leaf's $lsets$ corresponding to different characters or the $lset$ corresponding to λ, implying that the algorithm will generate the pair at this leaf. If α is the path-label of an internal node, then the fact that α is a largest maximal common substring ensures that s and s' will occur once in the $lsets$ of different children, and the $lsets$ will correspond either to different characters or to λ. Thus, the algorithm will generate the pair at this internal node. □

Lemma 3. *The algorithm runs in time proportional to the number of pairs generated plus the cost of sorting the nodes of the GST.*

Proof. Once the nodes are sorted by string-depth, each node of string-depth $\geq \psi$ is processed exactly once. For every pair generated and reported at any node, there is an equivalent reverse complemented pair which is generated and discarded elsewhere. This increases the runtime by a constant factor of 2. At an internal node, eliminating duplicate string ids reduces the total size of all $lsets$ of all its children by at most a factor of $(|\Sigma| + 1)$. This is because a string is present in at most one $lset$ of each child node and the number of children is bounded by $(|\Sigma| + 1)$. The total size of all the $lsets$ of all the children after duplicate elimination is bounded by the number of pairs generated at the node. Taken together, this implies that the cost of elimination by traversing the $lsets$ of the child nodes is bounded by a constant multiple of the number of pairs generated at the node (assuming $|\Sigma|$ is a constant). □

Prior to pair generation, each processor sorts the nodes in its local portion of GST in decreasing order of string-depth. Therefore, the order in which the promising pairs are generated on a processor is guaranteed to be in the decreasing order of their maximal common substring length, only with respect to the local GST. Though one can merge the lists of pairs generated on different processors into one list that reflects decreasing order on the entire GST, such an effort is not likely to improve runtime significantly. Note that the quality of clustering is unaffected by the order of pair generation.

6 Parallel Clustering

Our parallel EST clustering phase is a slight variant of the master-slave paradigm. In contrast to typical master-slave applications where the master processor generates tasks

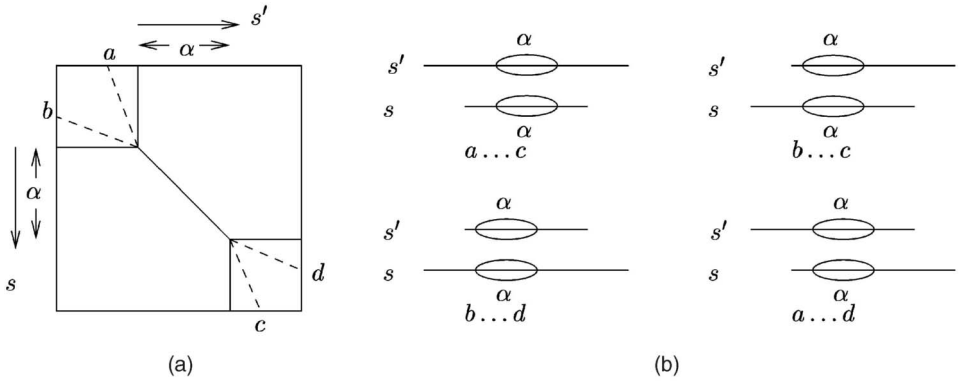


Fig. 6. (a) Dynamic programming table showing the extension of a maximal common substring match α, at both its ends. Note that the extension can include mismatches and gaps as well. (b) The four types of overlaps accepted as indication to merge clusters, and their corresponding optimal paths in the dynamic programming table.

and the slave processors perform tasks, the slave processors both generate and perform the tasks in our algorithm. The master processor acts as a conduit to determine the necessity of carrying out generated tasks and also distributes the tasks to ensure load balancing.

The basic task is to perform pairwise alignment on a promising pair. The master processor is responsible for 1) maintaining and updating the EST clusters, and 2) selecting and distributing tasks (in units of *batchsize*) to the slave processors. Each slave processor is responsible for 1) on-demand generation of promising pairs in batches and supplying them to the master processor for distribution, and 2) performing the task of pairwise alignment on the promising pairs assigned by the master processor to it and returning the results. Not all promising pairs generated by a slave processor are selected for pairwise alignment. If a generated pair of ESTs is already in the same cluster, no pairwise alignment is performed on it. The clusters are incrementally updated based on the set of results returned to the master processor.

The states of the master and slave processors are based on four variables: P, R, W, E. These state variables are defined local to the scope of every instance of a master-slave interaction. An $interaction$ on the master processor is the period between two successive receives from slaves. An $interaction$ on a slave processor is the period between two successive sends to the master processor. The variables P, R, W, E denote the following:

P: Number of promising pairs sent by a slave processor to the master processor.

R: Number of alignment results sent by a slave processor to the master processor.

W: Number of promising pairs assigned by the master processor to a slave processor.

E: Number of promising pairs requested from a slave processor for the next interaction.

The EST clusters are maintained by the master processor using the union-find data structure [25]. Initially, each EST is in a cluster of its own. We require two operations—1) to find the cluster corresponding to an EST (find) and 2) to merge two clusters (union). The amortized runtime per operation using the union-find data structure is given by the inverse Ackermann's function [25], a constant for all practical purposes.

The master processor maintains a large work buffer of pairs yet to be processed. The sequence of operations that are performed during an interaction on the master processor is as follows: A message received from a slave processor consists of two parts—R results and P promising pairs. The results are used to update the EST clusters as follows: A result that indicates any one of the four alignment patterns shown in Fig. 6b, with a score that passes a certain quality threshold, is accepted by merging the clusters containing the corresponding ESTs. Otherwise, the corresponding clusters are unchanged. Apart from this, additional processing can be done to decide if the pair of ESTs should belong in the same cluster. Examples of such processing include 1) detection of alternative splicing,[1] 2) consulting protein databases to see if the two ESTs have homology to the cDNA of the same protein, etc. The additional processing can be used to enhance the quality of EST clustering, and can even be organism specific, if so desired. Once the results are incorporated, the master processor selectively adds to the work buffer, only those pairs out of the P promising pairs whose corresponding ESTs are currently in different clusters. This is to eliminate unnecessary work. Let P' denote the number of pairs added to the work buffer. The master processor then dispatches a message to the slave processor consisting of 1) W (equal to *batchsize* or fewer, if not available) number of pairs from the work buffer and 2) E, the number of requested pairs, computed as $E = \delta \times min(\mu \times batchsize, \frac{nfree}{p})$, where $\mu = \frac{P}{P'}$, δ is the ratio of total number of slave processors to the number of slave processors that have not exhausted their pair generation, and $nfree$ is the number of free slots in the work buffer. This is to enable sending *batchsize* number of useful promising pairs to each slave in the future, without running the risk of overflowing the buffer in case all the received pairs are added to the buffer.

To initiate the process, each slave processor generates *batchsize* promising pairs, performs pairwise alignment on them and stores the results in a message buffer. Subsequently, it generates an additional set of $2 \times batchsize$

1. *Alternative splicing* is the phenomenon by which multiple mRNAs are produced from the same gene. This happens through exon-skipping and intron-retention.

promising pairs; the first *batchsize* pairs are marked as the set of pairs to be processed next for alignment, and the second *batchsize* pairs are appended to the message buffer as newly generated promising pairs. The message is sent to the master processor and the slave enters a loop of interactions. In all subsequent interactions, the slave processor always has a batch of promising pairs to process for alignment, between the time of sending the results from the previous interaction and the time of receiving the next batch of work. This way, the communication overhead is masked by an overlapping computation. At each slave processor, the promising pairs are enqueued in a large buffer at generation, and dequeued when dispatched to the master processor.

The sequence of operations that are performed during an interaction on a slave processor is as follows: The processor computes pairwise alignment on the set of pairs to be processed next. Once these R (equal to W of the previous interaction) results are obtained, the processor waits for the next message from the master processor. While waiting, it generates more promising pairs until either the message arrives, or the buffer that stores the pairs is full, or there are no more pairs to be generated. This further ensures the slave processor is not idle waiting for the master processor to respond. A message received from the master processor consists of W pairs and the number E. The processor checks its buffer to see if E pairs are available. If not, it generates more promising pairs on-the-fly from its local tree until either E pairs are in its buffer or there are no more promising pairs to be generated. The processor immediately dispatches a message to the master processor consisting of P (equal to the minimum of E and the number of pairs in its buffer) promising pairs and the R results, ending the current interaction.

The pairwise alignment algorithm computes the semi-global alignment of two strings [22], and takes advantage of the fact that a maximal common substring of the strings is already known. The corresponding dynamic programming table is shown in Fig. 6a. The alignment work is reduced by ensuring the maximal common substring to be part of the alignment, and extending the alignment at both of its ends to allow gaps and mismatches. This restricts the area of table computed to the two rectangles shown above and below the maximal common substring in Fig. 6a. To further limit work, we use banded dynamic programming, where the band size is determined by the number of errors tolerated. Quality is controlled by the usual set of parameters such as match and mismatch scores, and gap opening and gap continuation penalties [22]. Also, we define a quality threshold based on the ratio of an alignment score to its corresponding ideal score comprising all matches over the aligning region.

7 Experimental Results

We implemented PaCE using C and MPI. We report results on the quality of EST clustering produced by the software and its runtime performance on an IBM xSeries cluster with Myrinet interconnect. The cluster consists of 30 dual-processor nodes each with two 1.26 GHz Intel Pentium III processors and 1.25 GB RAM.

7.1 Quality Assessment

Here, we report a brief assessment of the quality of PaCE clustering. For a more detailed report, discussion of clustering decisions made by PaCE on selected EST sequences, and the results of clustering ESTs from various plant species, see [10].

The accuracy of the results is assessed using a benchmark data set consisting of 168,200 ESTs from *Arabidopsis thaliana*, and their correct clustering [30]. Because the complete genome of this plant is available and is relatively small, correct clustering can be obtained through alternative means. Using a spliced alignment program, each EST is directly aligned to the genome. All ESTs that align to locations spanning a gene locus are clustered. In cases where an EST aligns to multiple gene loci, the EST is mapped to a cluster corresponding to the gene locus that gives the maximum spliced alignment score. ESTs that do not align to any location on the genome are discarded. In addition to capturing such interesting cases, this method ensures that correct clusters are generated. Note that this method is not applicable when the genome is not available. Further, it is not suitable when the genome size is large and/or the EST collection is very large (both are true in the case of human EST data).

For assessment, the clusters were created using the following procedure: We first ran PaCE on various subsets of the benchmark data to generate clusters, and then ran CAP3 on each of these clusters to generate contigs. ESTs that belong to the same contig are clustered, and the resulting set of clusters are referred to as *PaCE clusters*. For the purpose of comparison, we also generated *CAP3 clusters* by running CAP3 directly on the subsets of the benchmark data.[2]

We individually compared the PaCE clusters and CAP3 clusters against the benchmark clusters. To make a comparison, we adopted the following approach: For a given cluster of ESTs, generate all pairs of ESTs with the property that both ESTs of a pair are from the same cluster. Based on the number of such pairs generated the following measurements are defined (illustrated in Fig. 7): A pair, according to our clustering, is called a true positive (TP) if it is also paired in the correct clustering, and is called a false positive (FP) otherwise. A pair that is not corresponding to our clustering is called a true negative (TN) if it is also not paired according to the correct clustering, and is called a false negative (FN) otherwise. Based on these measurements, another set of quality measurements are defined: $Overlap\ quality$ indicates the ratio of the number of TPs to the total number of unique pairs extracted from clusters of both results, and is given by $OQ = \frac{TP}{TP+FP+FN}$. OQ is also known as the *Jaccard coefficient* [9]. $Specificity$ is the fraction of correctly predicted pairs with respect to the total number of pairs predicted, and is given by $SP = \frac{TP}{TP+FP}$. $Sensitivity$ is the fraction of correct pairs predicted and is given by $SE = \frac{TP}{TP+FN}$. Overall performance is given by the $correlation\ coefficient$,

2. Running of CAP3 on 168,200 ESTs was enabled by running it on a machine with 3.25 GB memory.

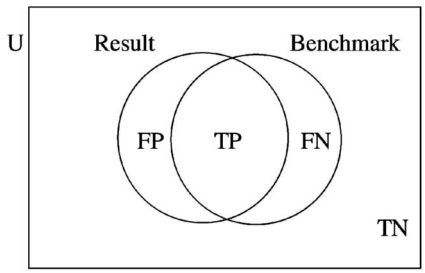


Fig. 7. Diagram illustrating the measurements *True Positives (TP)*, *True Negatives (TN)*, *False Positives (FP)*, and *False Negatives (FN)*. "U" refers to the set of all possible pairs of the input ESTs. For the Result and Benchmark, two ESTs are paired if they are in the same cluster.

$$CC = \frac{TP.TN - FP.FN}{\sqrt{(TP+FP).(TN+FN).(TP+FN).(TN+FP)}}.$$

Ideally, $OQ = SP = SE = CC = 100\%$.

The results of assessing the quality of our software and CAP3 using the benchmark data sets are shown in Table 2. Observing the measurements OQ, SP, SE, and CC, our results are very close to the results of CAP3, with CAP3 showing slightly better results than PaCE. In general, the sensitivity rate is lower than the specificity and this is attributable to the conservative nature of clustering criteria used. The results are based on the choice of the quality threshold parameters of PaCE and CAP3, experimentally found to optimize specificity and sensitivity simultaneously.

7.2 Runtime Assessment

The software is run for various subsets of the *Arabidopsis* EST data set using different numbers of processors. The total runtimes as a function of the number of processors for various data sets are shown in Fig. 8a. A window size of nine is used in partitioning the ESTs into buckets for parallel construction of GST; this generates potentially $4^9 = 262,144$ buckets, large enough to be distributed in a load-balanced fashion on multiprocessor systems. *Batchsize*, the unit of work given by a master processor to perform pairwise alignments on a slave, is chosen to be 60 pairs. As can be observed, the runtimes show near perfect scaling with the number of processors. We are also interested in the growth of runtime as a function of the data size for a fixed number of processors. While the memory required scales linearly with the problem size, the total runtime cannot be analytically determined and depends on the input data set. These runtimes for various data set sizes are shown in Fig. 8b.

TABLE 2
Quality Assessment of PaCE and CAP3 Clusters Using Clusters Generated from Different Portions of the Benchmark Data Set

n	50,012		100,003		168,200	
	PaCE	CAP3	PaCE	CAP3	PaCE	CAP3
OQ	86.87	89.32	84.84	89.13	88.87	90.35
SP	98.67	98.13	96.2	95.62	96.5	96.15
SE	87.91	90.87	87.78	92.92	91.83	93.74
CC	93.12	94.42	91.89	94.26	94.13	94.94

A subdivision of the runtimes into the time spent on various components of the software for 20,000 ESTs is shown in Table 3. Asymptotically, the largest contributor to the total runtime is the time spent in performing pairwise alignments during the clustering phase. The GST construction phase scales linearly (treating the average length of an EST to be a large constant). The clustering phase is expected to take quadratic runtime. In our approach, the time spent in pairwise alignments is significantly reduced because our algorithm 1) avoids unnecessary duplicates in generating promising pairs and 2) processes high-quality promising pairs first which has the effect of eliminating other promising pairs from further consideration. Because of these reasons, for smaller data sizes, the alignment phase

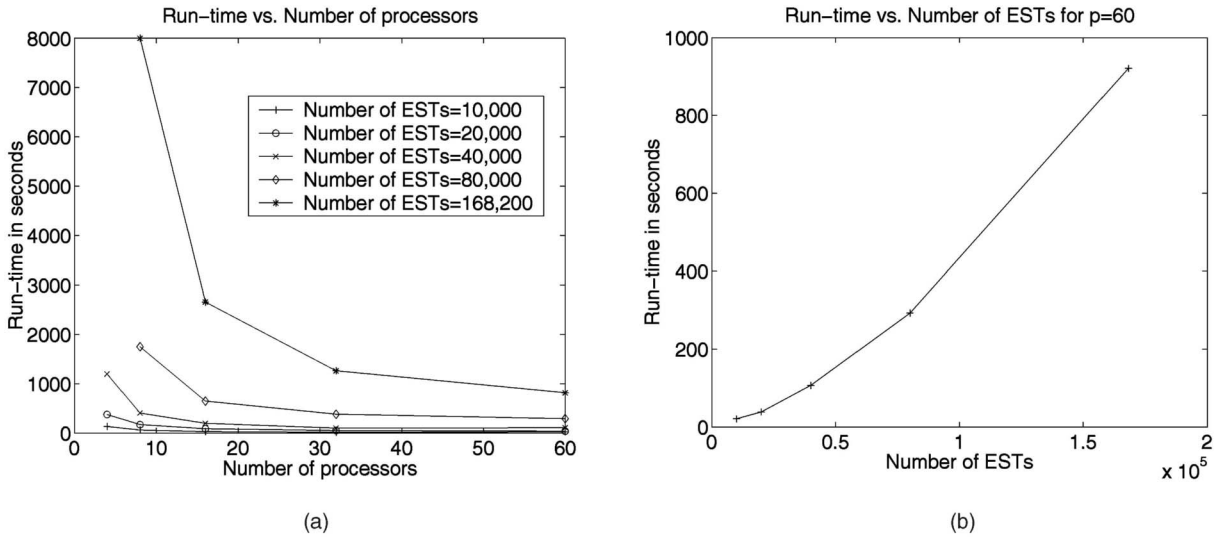


Fig. 8. (a) The parallel runtimes of PaCE as a function of the number of processors. (b) The parallel runtimes as a function of the data size when the number of processors is fixed to 60.

TABLE 3
Time (in Seconds) Spent in Various Components of Parallel EST Clustering as a Function of the Number of Processors (p) for 20,000 ESTs

p	Partitioning	Construction of GST	Sorting Nodes	Clustering Phase	Total Time
2	28	715	30	271	1044
4	13	250	10	102	375
8	5	110	4	50	119
16	2	57	2	26	87
32	1	36	1	15	53
60	1	27	1	10	39

runs faster than the GST construction phase as seen from Table 3.

Fig. 9a shows the total number of promising pairs generated as a function of the data size. Observe that the alignment work is done for only a small portion of the pairs generated (for example, 22 percent for the 168,200 data set). This illustrates the reduction in work achieved by processing the pairs in the decreasing order of maximal common substring length, as opposed to processing them in an arbitrary order. Also, note that the number of aligned pairs that contribute to merging of clusters is linear in n, as at most $n-1$ union operations can be performed. Because of the nature of master-slave interactions during the clustering phase, the number of pairs that are actually aligned varies slightly as the number of processors changes. We found the variation to be insignificant.

Fig. 9b shows the number of clusters as a function of the cluster size for 168,200 ESTs. About 44 percent of the clusters formed contain a single EST. A few clusters contain as many as several hundred ESTs (e.g., there are 34 clusters with size above 200). This nonuniformity in the size distribution in clusters is the primary reason why fragment assembly software has large memory and runtime requirements when applied to EST clustering.

The effect of *batchsize* on the clustering phase of PaCE is shown in Fig. 10. When the *batchsize* is small, the master and slave processors exchange messages more frequently, thereby making the communication overhead dominant. With a large *batchsize*, EST clusters are less frequently updated, causing alignment of more promising pairs than necessary. Empirically, we found the optimal *batchsize* for the benchmark data set to be in the range of 20-60. With the *batchsize* fixed, increasing the number of slave processors also increases the percentage of the total-time the master is busy. However, this percentage is well under two percent, even on 60 processors. Thus, using a single master processor is not likely to be a bottleneck for a large number of slave processors.

As an illustration of the capability of our software to solve problems of various sizes, we clustered EST data sets from 23 different plant species. The sizes range from 501 ESTs (*Avena sativa*) to 420,694 ESTs (*Triticum aestivum*). The results of PaCE clustering and subsequent CAP3 assembly are available at http://www.plantgdb.org.

8 Conclusions and Future Work

We reported on the development of a parallel software system for EST clustering. In creating this software, our overarching goal has been to facilitate fast and accurate clustering of large EST data sets. The primary contributions of this work are:

1. reducing the worst-case space requirement from quadratic to linear,
2. generating pairs of sequences in decreasing order of maximal common substring length without actually storing the pairs, and
3. reducing the number of pairwise alignments without affecting the quality of EST clustering.

PaCE is freely available for nonprofit, academic use. The source code and executables can be obtained by e-mail request to ananthk@cs.iastate.edu.

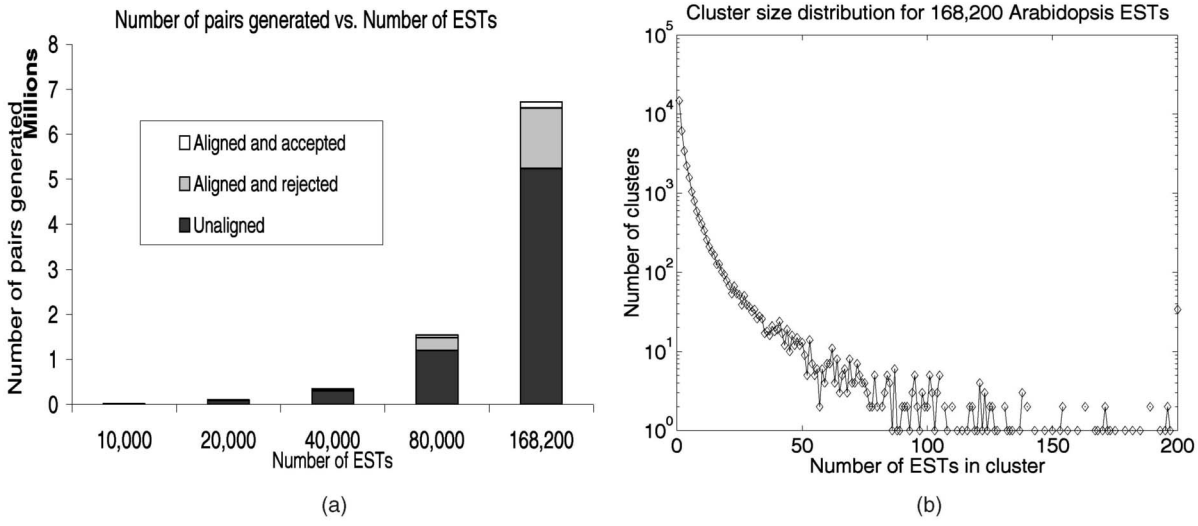


Fig. 9. (a) The number of pairs generated, the number of pairs that are aligned, and the number of pairs accepted as functions of the data size. (b) The number of clusters as a function of the cluster size for 168,200 ESTs. There are 34 clusters with size above 200 that are not shown the graph.

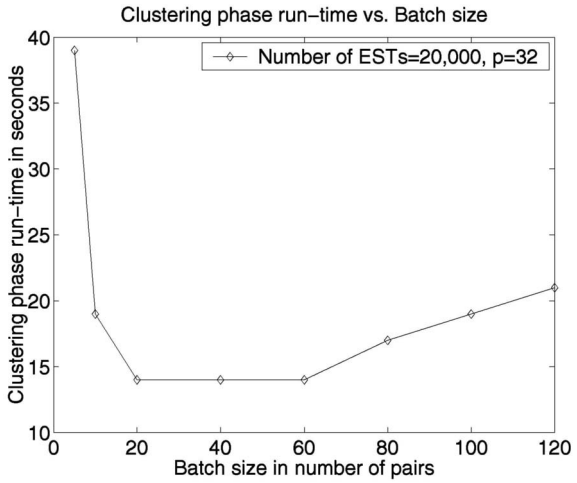


Fig. 10. Variations in runtime for the clustering phase as a function of the number of pairs allocated at a time for pairwise alignment.

Our next goal is to extend PaCE to do EST assembly and build contigs in parallel. We are working on improving the prediction accuracy of the software through additional processing such as detection of alternative splicing. We are also working on space and runtime improvements to enable solving problems an order of magnitude larger quickly using modest parallel computing capabilities. This is needed to facilitate clustering of the mouse and human EST data. Several interesting problems remain, whose solution can be used to improve the runtime and functionality of the software. Can a parallel algorithm for GST construction with optimal parallel runtime be designed for a practical model of parallel computation? Is there a way to incrementally adjust the EST clusters when a new batch of ESTs is sequenced, instead of the current method of clustering all the ESTs from scratch?

ACKNOWLEDGMENTS

The authors wish to thank Qunfeng Dong and Shannon Schlueter for their help in providing the benchmark data sets. They would also like to thank Richa Agarwala, Alejandro Schäffer, Greg Schuler, and Jean Terry-Mieg for discussions that led to an improved understanding of the problem characteristics. They thank Natsuhiko Futamura for contributing some useful implementation techniques that were critical in reducing the memory requirements. This research was supported in part by the US National Science Foundation under ACE-0203782 and CCR-0096288.

REFERENCES

[1] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Schieber, and U. Vishkin, "Parallel Construction of a Suffix Tree with Applications," *Algorithmica,* vol. 3, pp. 347-365, 1988.

[2] J. Burke, D. Davison, and W.A. Hide, "d2_Cluster: A Validated Method for Clustering EST and Full-Length cDNA Sequences," *Genome Research,* vol. 9, no. 11, pp. 1135-1142, Nov. 1999.

[3] J.E. Carpenter, A. Christoffels, Y. Weinbach, and W.A. Hide, "Assessment of the Parallelization Approach of d2_Cluster for High Performance Sequence Clustering," *J. Computational Chemistry,* vol. 23, no. 7, pp. 755-757, 2002.

[4] E. Coward, S.A. Haas, and M. Vingron, "SpliceNest: Visualizing Gene Structure and Alternative Splicing Based on EST Clusters," *Trends in Genetics,* vol. 18, no. 1, pp. 53-55, 2002.

[5] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge Univ. Press, 1997.

[6] S.A. Haas, T. Beissbarth, E. Rivals, A. Krause, and M. Vingron, "GeneNest: Automated Generation and Visualization of Gene Indices," *Trends in Genetics,* vol. 16, no. 11, pp. 521-523, 2000.

[7] R. Hariharan, "Optimal Parallel Suffix Tree Construction," *J. Computer and System Sciences,* vol. 55, no. 1, pp. 44-69, 1997.

[8] X. Huang and A. Madan, "CAP3: A DNA Sequence Assembly Program," *Genome Research,* vol. 9, no. 9, pp. 868-877, 1999.

[9] A.K. Jain and R.C. Dubes, *Algorithms for Clustering Data.* Englewood Cliffs, N.J.: Prentice Hall, 1988.

[10] A. Kalyanaraman, S. Aluru, S. Kothari, and V. Brendel, "Efficient Clustering of Large EST Data Sets on Parallel Computers," *Nucleic Acids Research,* vol. 31, no. 11, pp. 2963-2974, 2003.

[11] Z. Kan, E.C. Rouchka, W.R. Gish, and D.J. States, "Gene Structure Prediction and Alternative Splicing Analysis Using Genomically Aligned ESTs," *Genome Research,* vol. 11, pp. 889-900, 2001.

[12] J.P. Kitajima, G. Navarro, B.A. Ribeiro-Neto, and N. Ziviani, "Distributed Generation of Suffix Arrays: A Quicksort-Based Approach," *Proc. Workshop String Processing,* vol. 1264, pp. 53-69, 1997.

[13] A. Krause, S.A. Haas, E. Coward, and M. Vingron, "SYSTERS, GeneNest, SpliceNest: Exploring Sequence Space from Genome to Protein," *Nucleic Acids Research,* vol. 30, 2002.

[14] F. Liang, I. Holt, G. Pertea, S. Karamycheva, S. Salzberg, and J. Quackenbush, "An Optimized Protocol for Analysis of EST Sequences," *Nucleic Acids Research,* vol. 28, no. 18, pp. 3657-3665, 2000.

[15] E. McCreight, "A Space Economical Suffix Tree Construction Algorithm," *J. ACM,* vol. 23, pp. 262-272, 1976.

[16] B. Modrek and C. Lee, "A Genomic View of Alternative Splicing," *Nature Genetics,* vol. 30, pp. 13-19, 2002.

[17] G. Navarro, J.P. Kitajima, B.A. Ribeiro-Neto, and N. Ziviani, "Distributed Generation of Suffix Arrays," *Proc. Symp. Combinatorial Pattern Matching,* vol. 1264, pp. 102-115, 1997.

[18] S.B. Needleman and C.D. Wunsch, "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins," *J. Molecular Biology,* vol. 48, pp. 443-453, 1970.

[19] W.R. Pearson and D.J. Lipman, "Improved Tools for Biological Sequence Comparison," *Proc. Nat'l Academic of Sciences USA,* vol. 85, pp. 2444-2448, 1988.

[20] G. Pertea, X. Huang, F. Liang, V. Antonescu, R. Sultana, S. Karamycheva, Y. Lee, J. White, F. Cheung, B. Parvizi et al., "TIGR Gene Indices Clustering Tool (TGICL): A Software System for Fast Clustering of Large EST Datasets," *Bioinformatics,* vol. 19, no. 5, pp. 651-652, 2003.

[21] J. Quackenbush, J. Cho, D. Lee, F. Liang, I. Holt, S. Karamycheva, B. Parvizi, G. Pertea, R. Sultana, and J. White, "The TIGR Gene Indices: Analysis of Gene Transcript Sequences in Highly Sampled Eukaryotic Species," *Nucleic Acids Research,* vol. 29, pp. 159-164, 2001.

[22] J. Setubal and J. Meidanis, *Introduction to Computational Molecular Biology.* Boston, Mass.: PWS Publishing Company, 1997.

[23] T.F. Smith and M.S. Waterman, "Identification of Common Molecular Subsequences," *J. Molecular Biology,* vol. 147, pp. 195-197, 1981.

[24] G. Sutton, O. White, M. Adams, and A. Kerlavage, "TIGR Assembler: A New Tool for Assembling Large Shotgun Sequencing Projects," *Genome Science and Technology,* vol. 1, pp. 9-19, 1995.

[25] R.E. Tarjan, "Efficiency of a Good But not Linear Set Union Algorithm," *J. ACM,* vol. 22, no. 2, pp. 215-225, 1975.

[26] D.C. Torney, C. Burks, D. Davison, and K.M. Sirotkin, *Computers and DNA.* New York: Addison-Wesley, 1990.

[27] E. Ukkonen, "On-Line Construction of Suffix Trees," *Algorithmica,* vol. 14, pp. 249-260, 1995.

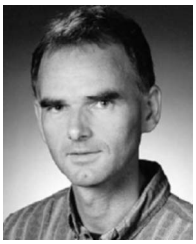
[28] P. Weiner, "Linear Pattern Matching Algorithm," *Proc. 14th IEEE Symp. Switching and Automata Theory,* pp. 1-11, 1973.

[29] Z. Zhang, S. Shwartz, L. Wagner, and W. Miller, "A Greedy Algorithm for Aligning DNA Sequences," *J. Computational Biology,* vol. 7, pp. 203-214, 2000.

[30] W. Zhu, S.D. Schlueter, and V. Brendel, "Refined Annotation of the *Arabidopsis Thaliana* Genome by Complete EST Mapping," *Plant Physiology,* June 2003.

Anantharaman Kalyanaraman received the BE degree in computer science from Nagpur University in 1998, and the MS degree in computer science from Iowa State University in 2002. He is a PhD student in computer science at Iowa State University. His current research focuses on computational biology, parallel algorithms, and string algorithms. He is a student member of the IEEE.

Srinivas Aluru received the BTech degree in computer science from the Indian Institute of Technology, Chennai, India, in 1989, and the MS and PhD degrees in computer science from Iowa State University in 1991 and 1994, respectively. He is an associate professor and associate chair of graduate education in the Department of Electrical and Computer Engineering at Iowa State University. He is affiliated with the Laurence H. Baker Center for Bioinformatics and Biological Statistics, and serves as associate chair for the Bioinformatics and Computational Biology graduate program. Earlier, he held faculty positions at New Mexico State University and Syracuse University. His research interests include parallel algorithms and applications, bioinformatics and computational biology, and combinatorial scientific computing. He cochairs an annual workshop in High Performance Computational Biology (http://www.hicomb.org) and has served as a guest editor of the *Journal of Parallel and Distributed Computing* for a special issue on this topic. Dr. Aluru served on the program committees of several conference and workshops on parallel processing and computational biology. He is a recipient of a US National Science Foundation CAREER award, an IBM faculty award, and the Young Engineering Faculty Research Award from Iowa State University. He has coauthored a book and more than 40 articles in peer-reviewed journals and conferences. He is a member of ACM, and a senior member of the IEEE and the IEEE Computer Society.



Volker Brendel received the MSc degree in applied statistics from the University of Oxford, United Kingdom, in 1981, and the PhD degree in life sciences from the Weizmann Institute of Science, Israel, in 1986. He is the Bergdahl Professor of bioinformatics at Iowa State University. Prior to joining the Iowa State University faculty in 1998, Dr. Brendel was at Stanford University as postdoctoral associate, research associate, and lecturer. Dr. Brendel's research interests span molecular biology, statistical modeling, and algorithm development, with applications in the field of genome informatics. Several software applications from his group are in wide use, including programs for gene structure prediction and sequence comparisons (http://gremlin1.zool.iastate.edu/~volker/b2go/). Recently, he has also been involved in the development and implementation of databases for genetic data, in particular maize (http://www.maizegdb.org) and other plants (http://www.plantgdb.org).

Suresh Kothari received the PhD degree in mathematics from Purdue University in 1977. After teaching at the University of Oklahoma, he joined the Computer Science Department at Iowa State University in 1984. He joined the Electrical and Computer Engineering Department in 1999. He leads the Software Systems Group in the department. He has pioneered the Knowledge-Centric Software (KCS) technology for automation tools for maintenance and evolution of large software. He has developed significant applications of the technology in a number of areas including parallel computing, high assurance software, and legacy software in business applications. He is founder of EnSoft, a company at ISU Research Park. He teaches courses in distributed computing, operating systems, parallel computing, and software engineering.