Space and Time Efficient Parallel Algorithms and Software for EST Clustering[*]

Anantharaman Kalyanaraman, Dept. of CS, Iowa State University, Ames, IA 50011

Srinivas Aluru[†], Dept. of ECpE, Iowa State University, Ames, IA 50011

Suresh Kothari, Dept. of ECpE, Iowa State University, Ames, IA 50011

Abstract

Expressed sequence tags, abbreviated ESTs, are DNA molecules experimentally derived from expressed portions of genes. Clustering of ESTs is essential for gene recognition and understanding important genetic variations such as those resulting in diseases. In this paper, we present the design and development of a parallel software system for EST clustering. To our knowledge, this is the first such effort to address the problem of EST clustering in parallel. The novel features of our approach include 1) design of space efficient algorithms to keep the space requirement 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 the run-time and facilitate the clustering of large data sets. Using a combination of these techniques, we report the clustering of 81,414 Arabidopsis ESTs in under 2.5 minutes on a 64-processor IBM SP, a problem that is estimated to take 9 hours of run-time with a state-of-the-art software, provided the memory required to run the software can be made available.

1 The EST Clustering Problem

DNA is a sequence 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 $\Sigma = \{A, C, G, T\}$. *Genes* are stretches of DNA that encode for protein molecules. They are composed of alternating segments called *exons* and *introns*. A gene is *transcribed* to its corresponding $mRNA$, which is a molecule describing the concatenation of the exons. The mRNA is later translated into its corresponding protein molecule. A cell contains mRNAs from different genes in various concentrations depending on its necessity to produce proteins. Through experimentation, the mRNAs are isolated and converted to the corresponding DNA molecules, known as *cDNA*. Due to experimental limitations, 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 done from either end. The resulting sequences are called *Expressed Sequence Tags*, or *ESTs* (see Figure 1).

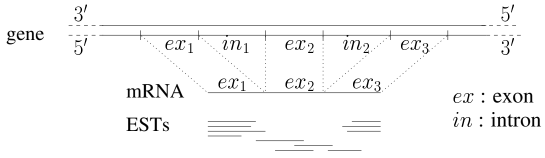


Figure 1. A simplified diagrammatic illustration of a gene, mRNA and ESTs.

Given a set of ESTs, **the EST clustering problem** is to partition the ESTs into clusters such that ESTs from each gene are put together in a distinct cluster. The input EST sequences contain errors due to the nature of experiments involved in deriving and sequencing them. A further complication arises due to the fact that DNA is actually 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 $C \leftrightarrow G$. The two strands of a DNA have opposite directionality. Thus, 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 $C \leftrightarrow G$.

The motivation for this work stems from the wide range of applications that require EST clustering. Some important applications are gene identification, gene expression studies, differential gene expression studies, Single Nucleotide Polymorphism (SNP) discovery and 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). With the number of ESTs for some organisms running into a few million, parallel processing is essential to cluster such large collections of ESTs.

The rest of the paper is organized as follows: In Section 2, we present our approach to EST clustering and highlight problems with current software. Section 3 contains details of our parallel algorithms. Experimental results are presented in Section 4. Section 5 concludes the paper.

[*]Research supported by NSF under ACI-0203782.
[†]Also supported by NSF Career under CCR-0096288.

Input	TIGR Assembler	Phrap	CAP3
50,000	X	23 mins	5 hrs
81,414	X	X	X

Table 1. Run-times of TIGR Assembler, Phrap and CAP3 on *Arabidopsis* ESTs using one IBM SP processor with 512 MB memory ('X' denotes insufficient memory to run program).

2 Our Approach

The primary information available to cluster ESTs is the potential overlaps between ESTs from the same gene. The overlap between two sequences can be computed by a pairwise alignment algorithm using dynamic programming [1, 9, 10, 12] in run-time quadratic in the length of the sequences, making it expensive to run for all pairs of ESTs. Hence, approximate overlap detection algorithms are used for fast identification of pairs of ESTs with potential for good quality overlap. The dynamic programming algorithm is then run on the more promising pairs.

The most popular software tools used for EST clustering are Phrap [3], CAP3 [6] and TIGR Assembler [13]. The run-times of each of these three programs on our benchmark *Arabidopsis* EST data sets are shown in Table 1. The programs are run on an IBM SP processor to enable direct comparison of results with our parallel software. We identified the generation of promising pairs as the memory-intensive phase and the computation of pairwise alignments on the promising pairs as the time-intensive phase. As for quality assessment, CAP3 produced the least number of erroneous clusters (see also [7]).

In light of experience with current software, the focus of our research is on developing memory-efficient algorithms and developing algorithmic strategies to minimize run-time without affecting quality. We also focused on parallel processing to achieve the twin objectives of further reducing run-time and facilitating clustering of large EST data sets by taking advantage of scaling of memory with the number of processors.

In our approach, each EST is initially considered a cluster by itself. Two clusters are merged when 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 do not show strong overlap, the corresponding clusters cannot be merged, and the effort in testing is wasted. Note that it may still be the case that the two clusters should be merged and our choice of the pair does not reflect that.

When two clusters are merged, it is no longer necessary

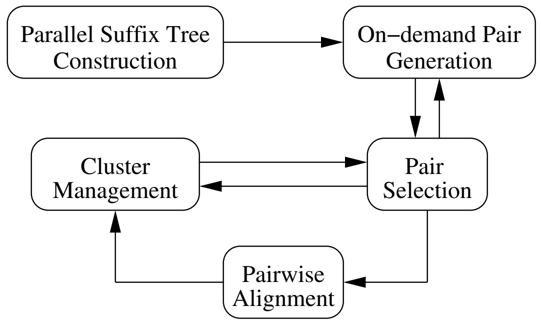


Figure 2. Organization of our Parallel EST Clustering Software.

to test pairs of ESTs where each is drawn from one of the two clusters. As success in merging of clusters depends on the choice of promising pairs being tested, significant savings in run-time can be achieved by generating pairs of ESTs in decreasing order of probability of strong overlap. We use length of a maximal common substring of pairs as the metric for predicting strongly overlapping pairs, and generate pairs of ESTs in the decreasing order of this metric. To minimize the memory requirements, our algorithm remembers its state and produces the next set of pairs on demand.

3 Parallel EST Clustering

The organization of our software is depicted in Figure 2. We first build a distributed representation of the generalized suffix tree data structure in parallel. This 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, and the algorithm is such that each processor need only access the portion of the suffix tree within itself. 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. Our algorithms for each of the components of the software are described in the following sections.

3.1 Parallel Construction of Generalized Suffix Tree

The suffix tree of a string is a compacted trie of all its suffixes [4]. Leaves in a suffix tree correspond to suffixes and internal nodes correspond to longest common prefixes shared by two or more suffixes. A Generalized Suffix Tree (GST) is a compacted trie of all suffixes of a set of strings, and can be constructed in time linear in input size [4].

We use the following notation: Let n be the number of ESTs and the set $\mathcal{E} = \{e_1, e_2, \ldots, e_n\}$ denote the ESTs. The total number of characters in 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 $e_i = s_{2i-1}$ and $\bar{e_i} = s_{2i}$, where $\bar{e_i}$ denotes the reverse complement of e_i. We use the terms *string* and *sequence*, and *substring* and *subsequence* in an equivalent manner.

We perform a parallel construction of the GST for $\mathcal{S}$, and this data structure is used for on-demand pair generation. Parallel algorithms for construction of suffix trees using the CRCW/CREW PRAM model are presented in [2, 5]. 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. Moreover, the average length of an EST is a fixed number (500-600) irrespective of the number of ESTs. Because of this, we use the following approach:

Initially, the ESTs are distributed across processors such that each processor has an approximately equal share of the total input, measured in number of characters. Each processor scans its ESTs and their reverse complements, and partitions their suffixes into at most $|\Sigma|^w$ buckets based on the first w characters. The total number of suffixes in each bucket over all the processors is computed using a parallel summation algorithm in $O(\log p)$ communication steps, where p is the number of processors. The buckets are then distributed to the processors such that 1) all the suffixes in a bucket are allocated to the same processor and 2) the total number of suffixes in all the buckets allocated to a processor is as close to $\frac{nl}{p}$ as possible. Care should be taken in choosing w. While assigning a large value to w may result in the loss of some potential overlapping pairs, assigning a low value will result in a small number of buckets for distribution among processors. Typically a value of 10 will allow us to generate $4^{10} > 1,000,000$ buckets, enough to distribute them in a load-balanced fashion on multiprocessor systems.

For each bucket, the processor responsible for it constructs the tree for all the suffixes in the bucket. Note that a sequential suffix tree construction algorithm can no longer be used because all suffixes of a string do not fall in the same bucket, unless the string is a repetition of a single character. To construct the tree, we use the simple approach of scanning all suffixes of a bucket one character at a time. As a result, a bucket is further subdivided into smaller buckets which are recursively subdivided, until each suffix is assigned a separate bucket. Assuming each processor receives approximately $\frac{nl}{p}$ total suffixes, the run-time for tree construction is $O\left(\frac{nl^2}{p}\right) = O\left(\frac{Nl}{p}\right)$. This algorithm works well in practice because l is independent of n. Note that the tree for each bucket is a subtree in the GST for $\mathcal{S}$. The collection of trees can be thought of as a distributed representation of the GST except for the top portion consisting of nodes with string-depth $< w$.

Because of 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. 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.

3.2 On-demand Pair Generation

Let *promising pair* refer to a pair of strings with a maximal common substring of length $\geq \psi$, a threshold value. The goal of the on-demand pair generation algorithm is to report the promising pairs on-the-fly, in decreasing order of maximal common substring length. 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[1] 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 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 $subtree(v)$ denote the set of nodes in the subtree of node v. Let $leaf(f)$ denote the leaf corresponding to a suffix f. Let *leaf-set*$(v) \subseteq \mathcal{S}$ be the set of strings that have a suffix f such that $leaf(f) \in subtree(v)$. The $leaf$-$set(v)$ is partitioned into five sets, $l_A(v)$, $l_C(v)$, $l_G(v)$, $l_T(v)$ and $l_\lambda(v)$, referred to as $lsets(v)$. If a string s is in $l_c(v)$ (for $c \in \Sigma \cup \{\lambda\}$), then the string has a suffix f such that $leaf(f) \in subtree(v)$ and f is left-extensible by c. Observe that such a partition need not be unique because a string s could have two suffixes f and f' such that $leaf(f)$ and $leaf(f')$ both are in $subtree(v)$, and f and f' are left-extensible by different characters. Then s could be either in $l_{c_i}(v)$ or $l_{c_j}(v)$. Any of

[1]The path-label of a node in GST is the concatenation of the edge labels from root to the node.

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

ProcessInternalNode(Internal Node: $v$)
    1. Traverse all $lsets$ of all children $u_1, u_2, \ldots, u_m$ of $v$. If a string is present in
            more than one $lset$, all but one occurrence of it are removed.
    2. Compute $P_v = \bigcup_{(u_k,u_l)} \bigcup_{(c_i,c_j)} l_{c_i}(u_k) \times l_{c_j}(u_l),\ \forall(u_k,u_l),\ \forall(c_i,c_j)$ s.t.,
            $1 \leq k < l \leq m,\ c_i \neq c_j$ or $c_i = c_j = \lambda$
    3. Create all $lsets$ at $v$ by computing :
            For each $c_i \in \Sigma \cup \{\lambda\}$ do
                $l_{c_i}(v) = \bigcup_{u_k} l_{c_i}(u_k),\ 1 \leq k \leq m$
```

Figure 3. Algorithm for generation of promising pairs.

these partitions will work for the pair generation algorithm.

The algorithm for generation of pairs is given in Figure 3. The nodes in local subtrees with string-depth[2] $\geq \psi$ 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. The set of pairs generated at node v is denoted by P_v. If v is a leaf, the cartesian products of $lsets$ corresponding to different characters are computed, in addition to a cartesian product of $l_\lambda(v)$ with itself, and their union 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 after the elimination, the $lsets$ at a child of v may no longer represent a partition of the $leaf\text{-}set$ of the child. After the elimination, cartesian products of $lsets$ corresponding to different characters and different children are computed, in addition to cartesian products of the $lset$s corresponding to λ of different children, and their union 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 $leaf\text{-}set(v)$.

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 identifiers is maintained. When a string is encountered in an $lset$ at a node, the entry in the array for this string is checked to see if it is marked with the identifier of the internal node being processed. If not, the array entry is marked with the node identifier. 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 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 will 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 duplicate generation of both (s, s') and (s', s) at the same node.

Let P_v denote the set of unordered pairs generated at any node v. In summary, if v is a leaf,
$$P_v = \{(s_1, s_2) \mid s_1 \in l_{c_1}(v), s_2 \in l_{c_2}(v), c_1, c_2 \in \Sigma \cup \{\lambda\}, ((c_1 < c_2) \vee (c_1 = c_2 = \lambda))\}$$
and if v is an internal node,

[2]The string-depth of a node in GST is the string length of its path-label.

$$
\begin{aligned}
P_v = \{(s_1, s_2) \mid\ & s_1 \in l_{c_1}(u_k), s_2 \in l_{c_2}(u_l), \\
& c_1, c_2 \in \Sigma \cup \{\lambda\}, u_k < u_l, \\
& ((c_1 \neq c_2) \vee (c_1 = c_2 = \lambda))\}
\end{aligned}
$$

The following lemmas are intended to prove the correctness and run-time 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.

For an internal node v, the algorithm implies that $s \in l_{c_i}(u_k)$ and $s' \in l_{c_j}(u_l)$, where u_k and u_l are distinct children of v and $c_i \neq c_j$ unless $c_i = \lambda$. Thus s and s' must have suffixes f and f' respectively, corresponding to leaves $x \in subtree(u_k)$ and $y \in subtree(u_l)$. These suffixes have the prefix α and f is left-extensible by c_i in s and f' is left-extensible by c_j in s'. If $c_i \neq c_j$, then the prefix α is not left-extensible by the same character in s and s'. If $c_i = c_j = \lambda$, then α is a prefix common to both s and s'. Also, since $u_k \neq u_l$, the prefix α is not right-extensible by the same character in s and s'. Thus, the prefix α of f and f' is a maximal common substring of s and s'. Figure 4 illustrates the proof for the case of an internal node. ■

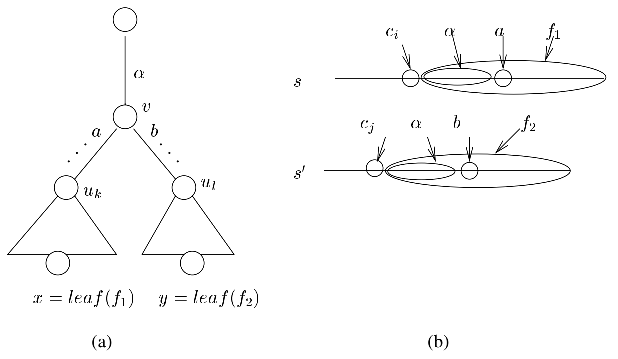


Figure 4. Illustration of the proof of Lemma 1.

Corollary 2 *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 3 *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 or an internal node v with path-label α. Also there exist suffixes f and f' 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 be present 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 4 *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 run-time 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 finite). ■

Finally, as each processor locally sorts the nodes in its local portion of GST, the order in which the promising pairs are generated is guaranteed to be in the decreasing order of their maximal common substring length only with respect to the local GST. In an ideal greedy approach the order has to be consistent across processors but as the GST is stored in a distributed fashion, this might involve a significant communication overhead. From our experiments we found that a compromise on the ideal greedy approach as opposed to incurring an additional communication overhead to be a better choice in terms of run-time. Note that the quality of clustering is unaffected by the order of pair generation.

3.3 Parallel Clustering

Our parallel clustering algorithm makes use of master-slave paradigm. The master processor is responsible for maintaining and updating clusters, and allocating promising pairs for pairwise alignment. The slave processors generate promising pairs in decreasing order of maximal common substring length and compute pairwise alignment on the pairs provided by the master processor. Pairwise alignment may not be performed for 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. For load-balancing, a pair generated on a slave processor need not be allocated to the same processor for pairwise alignment.

The master processor has two buffers: 1) $WORKBUF$, a large work buffer of pairs yet to be processed, and 2) $CLUSTERS$, the set of EST clusters. $WORKBUF$ is implemented as a queue and the promising pairs that are added to it for pairwise alignment are dispatched in units of *batchsize* to slave processors. The EST clusters are maintained using the union-find data structure [14]. We require two operations $-$ 1) to find the cluster of an EST (find), and 2) to merge two clusters (union). The amortized run-time per operation using the union-find data structure is given by the inverse Ackermann's function [14], a constant for all practical purposes.

The master processor performs a loop of interactions with the slave processors until all slave processors have run out of pairs and all pairs in $WORKBUF$ have been processed and their results collected. The sequence of operations during each iteration is as follows: A message received from a slave consists of two parts $-$ R results and P promising pairs. R results correspond to the results of the most recent pairwise alignments performed by the slave processor. The master processor updates $CLUSTERS$ for those results that indicate one of the alignment patterns shown in Figure 5b with a score above a certain threshold. Additional processing like detection of alternative splicing and consulting protein databases can be done to improve quality of the results. Following this, the master processor selectively adds a portion (say P') of the P promising pairs reported by the slave processor to $WORKBUF$. A pair is added only if the corresponding ESTs are in two different clusters, eliminating unnecessary work. If a slave processor runs out of pairs it is marked $passive$, and is considered $active$ otherwise.

After incorporating a received message, the master processor sends to the slave processor a message containing: W pairs extracted from $WORKBUF$ for pairwise alignment, and the number of pairs to request from the slave processor during their next interaction (E). The value of W is *batchsize*, or fewer if not available. Once a pair is assigned for pairwise alignment it is removed from $WORKBUF$. The value of E is determined as follows: Let $\mu = \frac{P}{P'}$ and δ be the ratio of the total number of slave processors (p) and the number of active slave processors. Let $nfree$ be the number of free slots in $WORKBUF$. Then $E = min(\mu \times \delta \times batchsize, \frac{nfree}{p})$. This is to receive approximately $\delta \times batchsize$ useful pairs from each active slave, without running the risk of overflowing $WORKBUF$. The δ factor ensures that there is enough work supplied to reactivate the passive slave processors for doing alignments. If both E and W are zero, no message is sent but instead, the slave processor is kept on a $wait$-$queue$. Later in one of the ensuing interactions, when there is excess work in $WORKBUF$, it is assigned to the slave processors in the wait-queue, thus removing such slave processors from the queue.

Each slave processor has three buffers: 1) Γ, the local GST, 2) $PAIRBUF$, to store the promising pairs generated on-demand but not yet sent to the master processor, and 3) $NEXTWORK$, the next batch of pairs for alignment. To get the process started, each slave processor initially generates three equal portions of *batchsize* number of pairs. After pairwise alignment is computed on the first portion, the results along with the third portion are sent to the master processor. The processor then marks the second portion as $NEXTWORK$ and enters a loop of interactions. Henceforth, the processor always has the next batch of pairs to work on, between submitting the results of the previous batch and receiving another set of pairs from the master processor, thus overlapping communication with computation.

The sequence of tasks performed by a slave processor during its interaction with the master processor is as follows: The processor computes pairwise alignment on the set of pairs in $NEXTWORK$. Once the (R) results are obtained, the processor waits for the next message from the master processor. While waiting, it generates more promising pairs until either a message arrives, or $PAIRBUF$ is full, or it runs out of promising pairs. This again ensures that the processor is not idle while waiting for the master processor to respond. A message received from the master processor consists of W pairs and the number E. If E pairs are not available in $PAIRBUF$, more promising pairs are generated on-the-fly from Γ until either E pairs are in $PAIRBUF$ or it runs out of promising pairs. The processor dispatches a message to the master processor consisting of P ($= min(E$, pairs in $PAIRBUF$)) promising pairs and the R results, ending the current interaction.

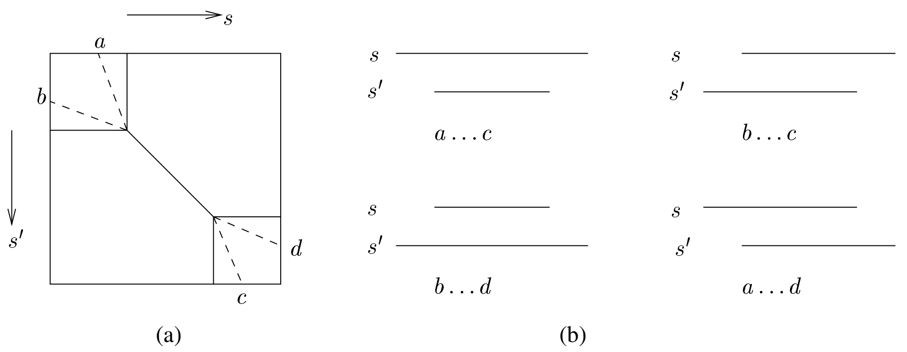


Figure 5. Figure showing pairwise alignment computed by extending a maximal common substring match at both ends. Also shown are the four types of alignments accepted as evidence to merge clusters, and their corresponding optimal paths in the dynamic programming table.

n	10,051		30,000		60,018		81,414
	Ours	CAP3	Ours	CAP3	Ours	CAP3	Ours
OQ	94.82	95.74	84.69	86.81	88.12	89.60	87.36
OV	0.04	0.15	7.67	6.70	4.79	4.54	6.02
UN	5.14	4.13	8.90	7.42	7.80	6.42	7.46
CC	97.37	97.83	91.71	92.93	93.69	94.51	93.25

Table 2. Quality assessment results of our software and CAP3 using benchmark data sets. CAP3 could not be run for $n = 81,414$ due to memory limitation.

Pairwise alignment is computed as shown in Figure 5a. Instead of aligning entire strings, we reduce work by merely extending the already computed maximal substring match at both ends using gaps and mismatches. This limits the area of computation as shown in the figure. To further limit work, we use banded dynamic programming, where the band size is determined by the number of errors tolerated. Quality can be controlled by the usual set of parameters, such as match and mismatch scores, gap opening and gap continuation penalties, and the ratio of score obtained to the ideal score consisting of all matches [11].

4 Experimental Results

We implemented our parallel EST clustering algorithms using C and MPI. We report results on the quality of EST clustering produced by the software and its run-time performance on an IBM SP with 375 MHz Power3 processors.

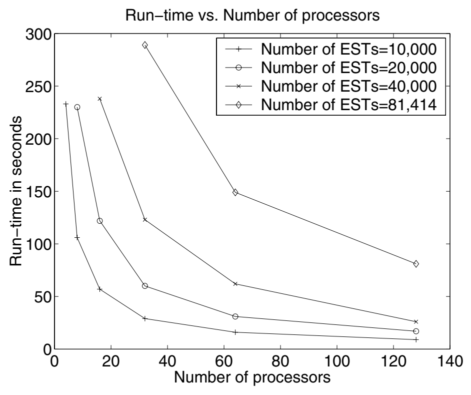
4.1 Quality Assessment

The accuracy of the results is assessed using a benchmark data set consisting of $81,414$ ESTs from *Arabidopsis thaliana*, and their correct clustering. As the complete genome of this plant is available and is relatively small, correct clustering can be obtained through alternative means. We compared the clusters generated by our software and CAP3 against the correct set of clusters generated using the above approach. To make a comparison, we adopted the following approach: For a given cluster of ESTs, generate all pairs of ESTs from the same cluster. Based on the number of such pairs generated the following measurements are defined: A pair according to output is called a true positive (TP) if it is also paired in the correct clustering, and a false positive (FP) otherwise. A pair not in output is called a true negative (TN) if it is also not paired according the correct clustering, and a false negative (FN) otherwise. Based on these measurements quality metrics are defined as follows [8]: $Overlap\text{-}quality$ is the proportion of TPs over the total number of unique pairs extracted from clusters of both results, and is given by $OQ = \frac{TP}{TP+FP+FN}$. $Over\text{-}prediction$ is the proportion of over-predicted pairs, and is given by $OV = \frac{FP}{TP+FP}$. $Under\text{-}prediction$ is the proportion of unpredicted pairs, and is given by $UN = \frac{FN}{TP+FN}$. Overall performance is given by the $correlation\text{-}coefficient$,

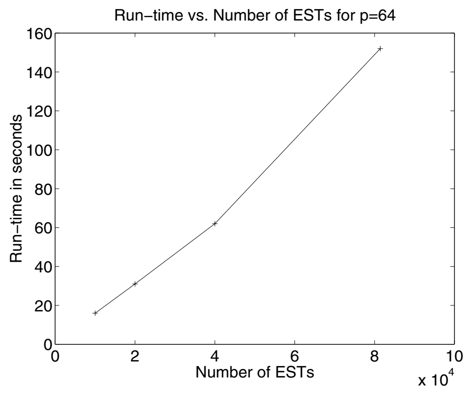
$$CC = \frac{TP.TN - FP.FN}{\sqrt{(TP+FP).(TN+FN).(TP+FN).(TN+FP)}}.$$

Ideally $OQ = CC = 100\%$ and $OV = UN = 0\%$.

The results of assessing the quality of our software and CAP3 using the benchmark data sets are shown in Table 2. Observing the metrics OQ, OV, UN and CC, our results are very close to the results of CAP3. In general, the under-prediction rate is greater than the over-prediction rate and this is attributable to the conservative nature of clustering criteria used. The results are based on the choice of quality threshold experimentally found to result in the least number of false positives and false negatives.



(a)



(b)

Figure 6. The graph to the left shows parallel run-times as a function of the number of processors. The run-times as a function of the data size for a fixed number of processors are shown in the graph to the right.

p	Partitioning	Construction of GST	Sorting Nodes	Pairwise Alignment	Total Time
8	3	180	5	42	230
16	1	91	2	27	121
32	1	45	1	13	60
64	0.5	22	0.5	8	31
128	0.5	11	0.5	5	17

Table 3. Time (in seconds) spent in various components of parallel EST clustering for 20,000 ESTs.

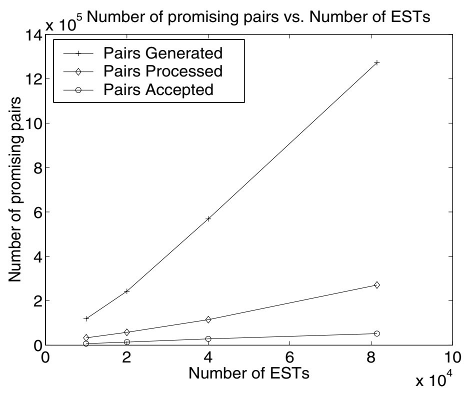


Figure 7. The number of pairs generated and the number of pairs that are aligned as a function of data size.

4.2 Run-time Assessment

The software is run for various subsets of the *Arabidopsis* EST data set using different numbers of processors (see Figure 6a). A window size of eight is used in partitioning the ESTs into buckets for parallel GST construction and $batchsize$ is chosen to be sixty pairs. As can be observed, the run-times scale with the number of processors. Figure 6b shows run-time as a function of the data size for a fixed number of processors. Although the memory required scales linearly with the problem size, the total run-time cannot be analytically determined and depends on the input data set. The run-time spent in various components of the software for $20,000$ ESTs is shown in Table 3. Asymptotically, the largest contributor is the time spent in performing the necessary alignments, followed by the time spent in parallel construction of GST. However for smaller data sizes, the alignment phase runs faster than the GST construction phase as seen from Table 3.

The total number of promising pairs and the number of pairs on which the pairwise alignment algorithm is run as a function of the data size are shown in Figure 7. This clearly illustrates the reduction in run-time achieved as a consequence of generating promising pairs in decreasing order of maximal common substring length, as opposed to the traditional way of generating pairs in an arbitrary order.

The effect on the run-time as the *batchsize* is varied for clustering $20,000$ ESTs on 32 processors is shown in Figure 8. A small *batchsize* results in more communications between the master processor and the slave processors. With a large *batchsize*, the slave processors become less responsive to pair generation, thus not taking advantage of the latest clustering information available to determine if alignment of a pair is necessary. We found the optimal *batchsize*, which is expected to increase with increase in the number of processors, to be in the range of 40-60 for our experiments. When the *batchsize* is fixed and the number of slave processors is increased, there is a gradual increase in the percentage of the total time the master is busy and the percentage is well under 2% even on 128 processors. Thus using a single

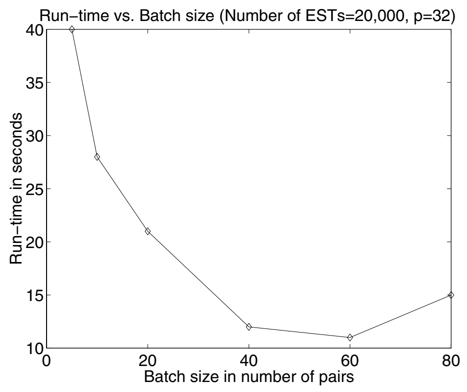


Figure 8. Run-times for parallel clustering as a function of $batchsize$.

master processor will not be a bottleneck even for a large number of slave processors.

5 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, which is accomplished through the use of memory-efficient algorithms, algorithmic heuristics and parallel processing. We are working on improving the prediction accuracy of the software by doing additional processing such as detection of alternative splicing. Several interesting problems remain, whose solution can be used to improve the run-time and functionality of the software. Can a parallel GST construction algorithm with optimal parallel run-time 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

We wish to thank Volker Brendel for introducing the EST clustering problem to us, providing benchmark data and for many valuable suggestions. We thank Richa Agarwala from National Institute of Health for discussions that led to an improved understanding of the problem characteristics. The run-time measurements are taken using the *Blue Horizon* IBM SP at the San Diego Supercomputer Center.

References

[1] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347–365, 1988.

[3] P. Green. *http://www.mbt.washington.edu/phrap.docs/phrap.html*, 1996.

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

[5] Ramesh Hariharan. Optimal parallel suffix tree construction. *Journal of Computer and System Sciences*, 55(1):44–69, 1997.

[6] X. Huang and A. Madan. CAP3: A DNA sequence assembly program. *Genome Research*, 9(9):868–877, 1999.

[7] F. Liang, I. Holt, G. Pertea, S. Karamycheva, S. Salzberg, and J. Quackenbush. An optimized protocol for analysis of EST sequences. *Nucleic Acids Research*, 28(18):3657–3665, 2000.

[8] P. A. Pevzner M. S. Gelfand, A. Mironov. Gene recognition via spliced alignment. In *Proc. National Academy of Sciences*, volume 93, pages 9061–9066, 1996.

[9] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[10] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proc. National Academic of Sciences USA*, 85:2444–2448.

[11] J. Setubal and J. Meidanis. *Introduction to computational molecular biology*. PWS Publishing Company, Boston, MA, 1997.

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

[13] G. Sutton, O. White, M. Adams, and A. Kerlavage. TIGR assembler: A new tool for assembling large shotgun sequencing projects. *Genome Science and Technology*, 1:9–19, 1995.

[14] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.