Parallel EST Clustering

Anantharaman Kalyanaraman
Dept. of CS
Iowa State University
Ames, IA 50011
ananthk@iastate.edu

Srinivas Aluru
Dept. of ECpE
Iowa State University
Ames, IA 50011
aluru@iastate.edu

Suresh Kothari
Dept. of ECpE
Iowa State University
Ames, IA 50011
kothari@iastate.edu

Abstract

Expressed sequence tags, abbreviated ESTs, are DNA fragments 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. The novel features of our approach include 1) 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 50,000 maize ESTs in 16 minutes on a 32-processor IBM SP. To our knowledge, this is the first effort in building a parallel software system for EST clustering.

1. Introduction

An expressed sequence tag (EST) is a sequenced portion of a full-length or a partial-length cDNA, experimentally obtained by reverse transcribing the corresponding mRNA. The mRNA corresponds to the transcribed portion of a gene. For a simplified diagrammatic illustration, see Figure 1. Given ESTs from multiple genes, the EST clustering problem is to partition them into clusters such that ESTs from the same gene are put together in one cluster. The motivation for developing an efficient parallel EST clustering software system stems from the wide range of current and future biological 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 include gene identification, gene expression studies, differential gene expression studies, SNP identification and design of microarrays.

The primary information available to cluster ESTs is the potential overlaps between ESTs drawn from the same gene. Most software programs currently used for clustering ESTs are actually developed for solving the related problem of *fragment assembly*. Fragment assembly is used to discover long stretches of genomic DNA from the sequences of several small fragments of it, and is used in genome sequencing. Once again, the assembly is based on detecting overlapping fragments, making the software usable for EST clustering as well. Fragment assembly software will actually assemble ESTs from the same gene into full length cDNAs (ideally), or into contiguous stretches of cDNAs (or contigs).

The overlap between two sequences that possibly contain errors can be computed by a pairwise alignment algorithm using dynamic programming and this method is accepted to be a good measure of overlap quality [1, 9, 10, 11]. As this algorithm takes time proportional to the product of the length of the sequences, it is expensive to run for all pairs of ESTs. Hence, approximate overlap detection algorithms are used to identify pairs of fragments 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 the TIGR Assembler [12], Phrap [3] and CAP [6], all originally designed for fragment assembly. Recently, researchers at TIGR evaluated the quality of EST clustering generated by the three programs and found that CAP produces the least number of erroneous clusters [7]. Phrap is the fastest of all, as it avoids expensive dynamic programming and instead relies on approximate matching. However, this causes the software to produce lower quality results.

We tested each of the three programs using $50,000$ maize EST sequences. The test run is done on a PC with dual Pentium 450MHZ processors and 512MB RAM. The TIGR Assembler ran out of memory during the assembly. The Phrap program finished in about 40 minutes. The CAP3 (version 3 of CAP) program took more than 24 hours. For the full database of more than 100,000 ESTs, both Phrap and CAP3

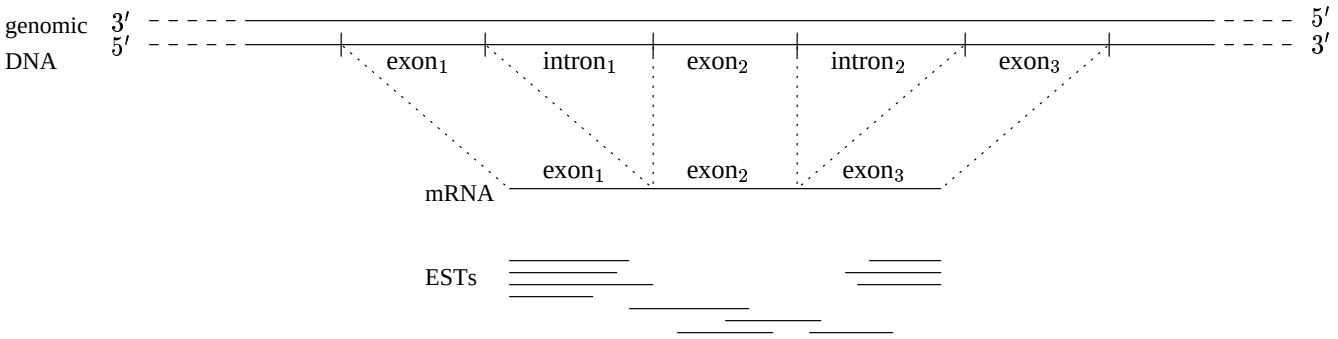

Figure 1. A simplified diagrammatic illustration of the EST problem.

programs ran out of memory as well. Based on the run-time behavior, we estimate that CAP3 would require at least 4 days to cluster the entire database, if sufficient memory is made available. We could not run the programs even on a HP-R390 machine with 3GB RAM. Compounding the problem of run-time and memory requirements, multiple runs using different parameters are typically needed in practice. All of the software programs described here are sequential. Working on a rat EST project, researchers at University of Iowa recently created UICLUSTER, a parallel EST clustering software. However, this software assumes all the ESTs are sequenced from the $3'$ end, an assumption not valid for most EST datasets.

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. In addition, we 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. As a result, we were able to cluster the same $50,000$ maize ESTs mentioned above in about 16 minutes using a 32-processor IBM SP-2. We targeted our effort to directly address the problem of EST clustering. Though fragment assembly software is often used for EST clustering, there are important reasons why it is not a perfect solution. Input to fragment assembly consists of random fragments from a long stretch of genomic DNA, providing uniform and often complete coverage of the DNA. In contrast, the number of ESTs collected per gene are proportional to the relative expressivity of the gene. Also, fragment assembly software attempts to combine overlapping sequences into contigs, a technique that does not always make sense for ESTs. ESTs are often collected from many different individuals, or even different strains in a population. In addition, phenomena such as intron-retention and exon-skipping result in different mRNAs from the same gene. On the other hand, such variations represent a valuable source of biological information, and often the reason why ESTs are sequenced and clustered.

The rest of the paper is organized as follows: In Section 2, we outline our main ideas used in developing time and space efficient parallel software for EST clustering. Sections 3 through 5 describe the various components of the software in detail. Experimental results, including quality and run-time assessments are presented in Section 6. Section 7 concludes the paper.

2 Our approach

Experimentation with current software indicates that most of the run-time is spent in pairwise alignment of promising pairs of ESTs. However, they run out of memory during the phase of identifying the set of promising pairs. The number of promising pairs is observed to increase quadratically with the number of ESTs. The promising pairs are typically defined to be pairs of ESTs that have a common substring of a certain length, say w. This approach results in generation of the same pair multiple times. For instance, if two pairs share a substring of length $l > w$, the pair is generated $l - w + 1$ times corresponding to the $l - w + 1$ substrings of length w that are common to the pairs. Such duplicates must be removed before the pairs are considered for pairwise alignment.

Consider 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 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 can not 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.

Significant savings in run-time can be achieved by fast

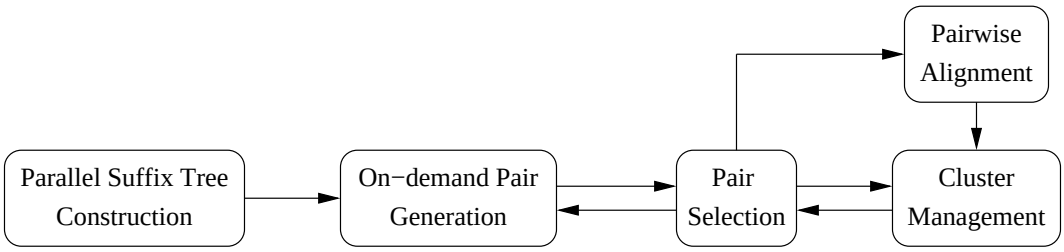

Figure 2. Organization of our parallel EST clustering software

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, we are interested in defining a suitable metric and producing the promising pairs in decreasing order of quality according to the metric.

A simple metric for the quality of the overlap between a pair of strings is the length of a largest common substring, referred to as 'maximum overlap length' from here onwards. Ideally, an efficient algorithm with minimal memory requirements that produces promising pairs in decreasing order of maximum overlap length is sought. To minimize the memory requirements, an on-demand algorithm that remembers its state and produces the next set of pairs on demand is ideal. 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 our parallel EST clustering software and the interactions between various components is depicted in Figure 2. We first build a distributed representation of the generalized suffix tree data structure in parallel. This data structure is used for on-demand generation of promising pairs in decreasing order of the length of a maximal substring overlap. The pair generation itself is done in parallel. Maintaining and updating of the EST clusters is handled by a single processor, which acts as a master processor directing the remaining processors to both generate batches of promising pairs and perform pairwise alignment on promising pairs. It is not mandatory to perform pairwise alignment of each generated pair because the current set of EST clusters may obviate the need to do so. Hence, the master processor is also responsible for the selection of pairs to be aligned and is a necessary intermediary between pair generation and alignment. To provide an added degree of flexibility in balancing the load, we do not require that a pair generated on a processor be allocated to the same processor if a pairwise alignment is needed. The algorithms used for each of the components of the software are described in the following sections.

The following notation is used throughout the paper. Let $\Sigma = \{A, C, G, T\}$ denote the alphabet. Let n denote the number of ESTs, and $e_1, e_2, \ldots, e_n$ denote the ESTs, which are strings over Σ. For EST e_i, l_i denotes its length, and $\bar{e_i}$ denotes its reverse complement. Let l denote the average EST length, i.e. $l = \frac{1}{n}\sum_{i=1}^{n} l_i$. As we should consider each EST and its reverse complement, we refer to the $2n$ fragments as $f_1, f_2, \ldots, f_{2n}$ for convenience. The length of the ESTs varies in the range of 100 to a 1000 bases, with the average length in the $500 - 600$ range.

3 Constructing Parallel Generalized Suffix Tree

A Generalized Suffix tree (GST) is a compacted trie representing all suffixes of a set of strings [4]. We construct the GST for all ESTs and their reverse complements, and this data structure is used for on-demand pair generation. Sequentially, such a GST can be built in $O(nl)$ time, where nl is the total number of suffixes of all the ESTs [8, 14, 15]. Parallel 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. Thus we adopt the following approach:

Initially, the ESTs are distributed across processors such that each processor has an approximately equal share of the total input. Each processor scans its ESTs and partitions the suffixes of these ESTs 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. 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 $O\left(\frac{nl}{p}\right)$ as possible.

For each bucket, the processor responsible for it computes the compacted trie of 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 trie, we use the simple approach of scanning the strings one character at a time. Assuming each processor receives $O\left(\frac{nl}{p}\right)$ total suffixes with an average length of l, the run-time for the trie construction is $O\left(\frac{nl^2}{p}\right)$. This algorithm works well in practice because the length of an EST is limited by the length of a fragment that can be sequenced in the laboratory, and is independent of the number of ESTs.

Note that the trie for each bucket is a subtree in the GST for all ESTs and their reverse complements. The collection of tries can be thought of as the GST except the portion consisting of nodes with string-depth $< w$. Thus, we now have a distributed representation of the GST except for the top portion consisting of such nodes. Because of concern for space-efficiency, each trie is stored as follows: The nodes are generated and stored in the order of the depth-first search traversal of the trie. 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 the rightmost leaf pointers of a node and a child of it are identical, the child is the rightmost child of the node. Each node also stores its string-depth, where string-depth is measured with respect to the GST.

4 On-demand Pair Generation

Our algorithm for on-demand pair generation is a simple variant of the standard suffix tree algorithm for computing all maximal repeats of a string, such as the one presented in Gusfield's textbook ([4]; Section 7.12). Recall that each processor has a portion of the GST of the ESTs and their reverse complements, given by the collection of the tries in it. Ideally, a pair of fragments should be generated if they share a maximal common substring of length greater than or equal to a threshold value. The algorithm should allow on-demand generation of such pairs, and each pair should be generated only once even if they share multiple maximal common substrings of sufficient length. Our algorithm will generate duplicates, the number of which is at most the number of distinct maximal common substrings of the pair.

Consider a pair of fragments and a substring common to them. We call the substring *left-extensible* (alternatively, *right-extensible*) if the character to the left (alternatively, right) of the substring in each fragment is the same. A substring is maximal if it is neither left-extensible nor right-extensible. If two suffixes from different fragments are in the leaf set of an internal node in a GST, the fragments share a common substring of length equal to the string-depth of the node. The pair of fragments can be generated if the substring can be identified to be maximal.

We use the following algorithm for generating the pairs: We first sort the internal nodes of the GST in decreasing order of string-depth, and process them in that order. For each node v in the tree, we store the set of fragments found in the leaf set of its subtree. This set is partitioned into five lists $l_A(v), l_C(v), l_G(v), l_T(v)$ and $l_\lambda(v)$, where λ refers to the null character. If a fragment f_i is in list $l_c(v)$ (for $c \in \Sigma \cup \{\lambda\}$), then there exists a suffix of f_i in the leaf set of the subtree under v such that the suffix is left-extensible by the character c. If the suffix is the entire fragment, then it is considered left-extensible by λ. The lists at a node are generated after it is processed, and is removed after its parent is processed. This limits the total space required for storing such lists to $O(nl)$, linear in the size of the input size.

Consider a node v and its children $u_1, u_2, \ldots, u_m$. Before generating the pairs, the lists of each child node are scanned (in no particular order) to ensure that a fragment is present in at most one list of one child node. This scanning can be done in time proportional to the total length of all the lists. After the duplicates are removed, the pairs generated at v are given by $\{(f_i, f_j) \mid f_i \in l_{c_1}(u_k), f_j \in l_{c_2}(u_l), k \neq l, ((c_1 \neq c_2) \vee (c_1 = c_2 = \lambda))\}$. After generating the pairs at a node, the lists of the children corresponding to the same character are merged to form the lists at the node. Also, a generated pair is discarded if the fragment corresponding to the smaller EST subscript 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 $(\bar{e_i}, e_j)$.

It can be easily shown that the total run-time of this algorithm is proportional to the number of pairs generated plus the cost of sorting the nodes of the GST. The rejected pairs increase the run-time by the constant factor of 2. The cost of eliminating the duplicates by traversing the lists at the children of a node can be absorbed by the fact that a fragment is present in at most one list at each child node and the number of children is at most $|\Sigma|$. Eliminating duplicates reduces the total size of all lists by at most a factor of $|\Sigma|$, and the total length of all the lists after eliminating duplicates is dominated by the number of pairs generated due to the lists.

Finally, to generate the pairs in parallel using all the processors, note that each processor can work with its individual tries, provided the string-depth of a node in the GST corresponding to the root of a trie is less than the thresh-

old value. This is because we are not interested in maximal common substrings of length less than the threshold value. This criterion can be easily satisfied because a small threshold value of 10 will allow us to generate $4^{10} > 1,000,000$ tries, enough to distribute them in a load-balanced fashion even on a large number of processors.

5 Parallel Clustering

Our parallel EST clustering algorithm makes use of a master-slave paradigm. The master processor is responsible for maintaining and updating the clusters. It receives promising pairs of ESTs from slave processors and determines which of these pairs should be explored using a pairwise alignment algorithm. It dispatches pairs in units of *batchsize* to slave processors to perform pairwise alignments and return the results. Upon receiving the result of a pairwise alignment, it determines if the clusters corresponding to the pair should be merged based on the received results, and additional evidence if necessary. The slave processors are responsible for generating pairs as demanded by the master processor and to perform pairwise alignments of the pairs dispatched by the master processor.

The clusters are maintained by the master processor using the union-find data structure [13]. Initially, each EST is in a cluster of its own. We require two operations on the cluster – 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 [13], a constant for all practical purposes.

The master processor maintains a large buffer of pairs yet to be processed. A message received by the master processor from a slave processor consists of two parts – results of the pairwise alignment for a number of pairs (same as the *batchsize*, unless as many pairs are not available), and a batch of next set of promising pairs generated on the slave, the number of which is based on a previous request from the master processor. The master processor immediately dispatches a message to the slave processor consisting of a) *batchsize* (fewer, if not available) number of pairs from its buffer and b) the number of promising pairs to be returned along with results of running pairwise alignment algorithm on each pair in (a). It then updates the EST clusters using the results received from the slave. Apart from using the results of pairwise alignment, additional processing can be done by the master to decide if the pair of ESTs should belong in the same cluster. Examples of such processing include 1) detection of alternative splicing, 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 into the clusters, the master proceeds with the task of adding the promising pairs received to the buffer of to-be-processed pairs. A pair is added to the buffer if the corresponding fragments are in different clusters. Otherwise, the pair is rejected.

The number of promising pairs the master processor requests a slave processor is determined as follows: The master keeps track of the ratio μ of the the total number of pairs from the most recent set of promising pairs received from the slave to the number of these pairs actually added to the buffer. Let $nfree$ denote the number of free slots in the buffer maintained by the master processor. The number of promising pairs requested from the slave processor is determined by $min(\mu \times batchsize, \frac{nfree}{p})$. This is to receive at least *batchsize* number of useful promising pairs from each slave, without running the risk of overflowing the buffer in case all the received pairs are added to the buffer.

To get the process started, each slave processor initially generates $3 \times batchsize$ number of pairs, consisting of three equal portions of *batchsize* number of pairs. At the beginning, all promising pairs must be explored. The processor immediately sends the third portion to the master processor, and starts pairwise alignments on the first portion. Once the results of the first portion are obtained, it sends the results along with a newly generated batch of pairs to the master processor. While waiting to receive another batch of pairs from the master processor, it works on the second portion. Thus, 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. Much of the overhead in communication is masked by this overlapping of computation and communication.

To perform pairwise alignment, recall that a maximal common substring of the pair is already known. Figure 3 shows the dynamic programming table corresponding to computing the pairwise alignment. Instead of aligning entire strings, we reduce work by merely extending the maximal substring match at both ends using gaps and mismatches. This limits the area of the table to be computed 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.

6 Experimental Results

We implemented the parallel EST clustering framework described in this paper using C and MPI. In this section, we report results on testing the quality of the software as well as its run-time behavior. We tested the run-time behavior us-

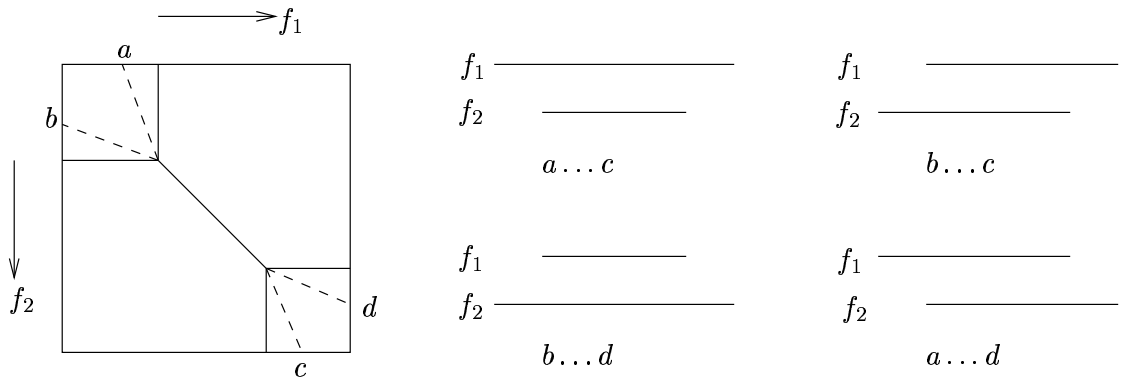

Figure 3. Figure showing pairwise alignment strategy of extending a maximal common substring match at both ends. Also shown are the four types of overlaps accepted as indication to merge clusters.

ing maize ESTs (http://www.zmdb.iastate.edu), clustering of which provided the motivation behind this work. For the purpose of assessing the quality of the software, ESTs from *Arabidopsis thaliana* are used, because its genome is available.

6.1 Quality Assessment

The quality of the software is assessed using a data set and its correct clustering, as provided by a colleague in Zoology & Genetics, Volker Brendel. The data set consists of ESTs from the model plant *Arabidopsis thaliana*, whose complete genome is available. Using a spliced alignment program, each EST is aligned to the genome. ESTs that do not align at all (because of defects in the alignment program or the genome), or that align in multiple spots are discarded. The ESTs are then clustered by walking along the genome.

We tested the clusters generated by our software 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 with the property that both ESTs of a pair are from the same cluster. If a pair according to the correct clustering is not paired in our output, it is considered a *false negative*. If a pair is not in accordance with the correct clustering but is paired in our output, it is considered a *false positive*. As a result of testing our software, we found that the false negatives are in the 5% range, and the false positives are a negligible percentage. For instance, on $10,000$ ESTs the number of pairs according to the correct clustering is found to be $162,648$. Of these, $154,298$ pairs are identified by our output. The number of false negatives is $8,350$ (5.13%) and the number of false positives is 60 ($< 0.04\%$). The reason behind the larger number of false negatives when compared to false positives is the conservative nature of clustering criteria used. The results presented are solely based on the quality of particular types of pairwise alignments as shown in Figure 3. These results are based on the choice of quality threshold experimentally found to result in the least number of false positives and false negatives. As mentioned before, quality can be further improved by using additional criteria for rejecting or accepting a pair, such as taking care of alternative splicing effects. We are working on adding such features to enhance the quality of EST clustering produced by the software.

6.2 Run-time Assessment

The software is run for various subsets of the EST database using different numbers of processors. The total run-times as a function of the number of processors for several fixed problems sizes are shown in Figure 4. A window size of 5 is used in partitioning the ESTs into buckets for parallel suffix tree construction. The unit of work given by a master processor to perform pairwise alignment on a slave is chosen to be ten pairs. As can be observed, the run times scale approximately linearly with the number of processors. We are also interested in the growth of run-time 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 run-time can not be analytically determined and depends on the input data set. These run-times for various subsets of the maize data set are also shown in Figure 4.

A subdivision of the run-times into the time spent in various components of the software for $5,000$ ESTs is shown in Table 1. Asymptotically, the largest contributor to run-time is the time spent in performing the necessary alignments. The next time-consuming component is parallel generalized suffix tree construction. Notice that the time spent in pairwise alignments is no longer prohibitively expensive.

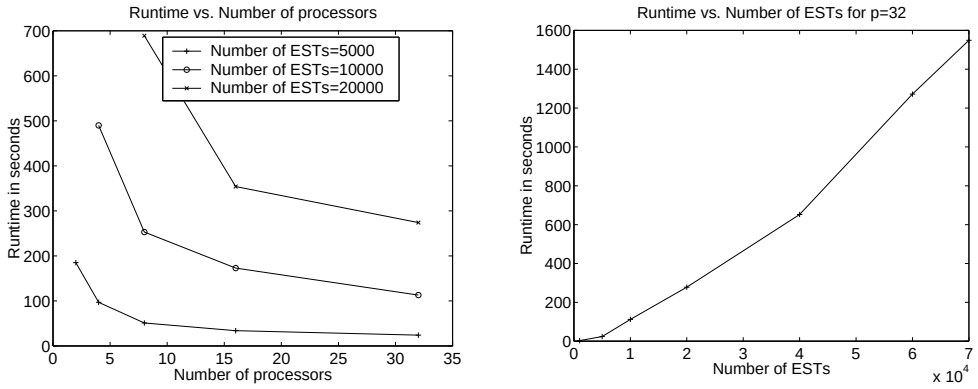

Figure 4. The left graph shows parallel run-times for EST clustering as a function of the number of processors. The run-times as a function of the data size for a fixed number of processors are shown in the right graph.

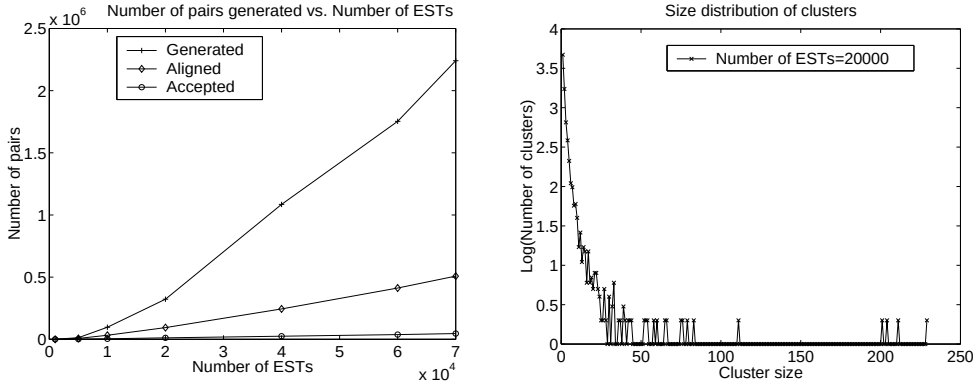

Figure 5. The left graph shows the number of pairs generated, aligned and accepted as a function of data size. The right graph shows the number of clusters as a function of the cluster size (in number of ESTs) for 20,000 ESTs.

This is because our algorithm 1) avoids unnecessary duplicates in generating promising pairs and 2) processes the more promising pairs first which has the effect of eliminating other promising pairs from further consideration.

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 5. Because of the nature of master-slave interactions during EST clustering, the number of pairs that are actually aligned varies slightly as the number of processors changes. We found the variation to be insignificant. Figure 5 also shows the number of clusters as a function of the cluster size. Typically, as many as half the clusters formed contain a single EST, and we refer to these clusters as singleton clusters. The number of clusters of a particular size (measured by the number of ESTs in the cluster) is a decreasing function of the size. A few clusters contain as many as several hundred ESTs. It is because of this variation that fragment assembly software is much slower when applied to EST clustering.

The effect on the run-time as the batch size is varied for clustering $10,000$ ESTs on 32 processors is shown in Figure 6. If the batch size is small, the master processor must communicate often with the slave processors, increasing the communication overhead. If the batch size is large, the slaves will become less responsive to the need for generating subsequent pairs. Also, a slave processor does not take advantage of the latest clustering information available to determine if alignment of a pair is necessary. The optimal *batchsize*, which is expected to increase with increase in the number of processors, can be found by experimentation. We found the optimal batchsize to be in the range of $10 - 20$ for the range of processors used in our experiments. When the

p	Partitioning	Generalized Suffix Tree	Sorting Nodes	Clustering	Total Time
2	4	90	6	83	183
4	2	48	3	44	97
8	1	23	1	24	49
16	1	16	1	13	31
32	4	8	1	9	22

Table 1. Time (in seconds) spent in various components of parallel EST clustering for 5,000 ESTs.

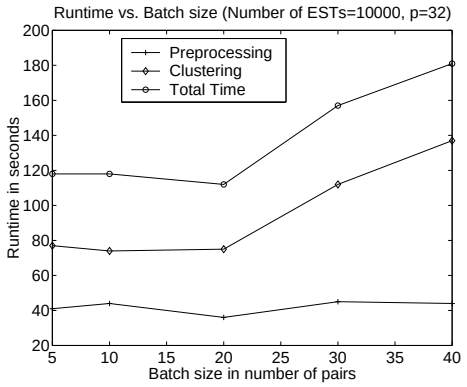

Figure 6. Variation in the run-time as a function of the number of pairs allocated at a time for pairwise alignment.

batch size is fixed and the number of slave processors is increased, there is a gradual increase in the percentage of the total time the master processor is busy and the percentage is well under 1% even on 32 processors. Thus using a single master processor will not be a bottleneck even for a large number of slave processors.

7 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 clustering of large EST datasets, which is accomplished through the use of memory-efficient algorithms, algorithmic heuristics and parallel processing. 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?

Acknowledgements

We wish to thank Volker Brendel for introducing the EST clustering problem to us, providing the data sets and for many valuable suggestions. We thank Richa Agarwala for discussions that led to an improved understanding of the problem characteristics.

References

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. 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] R. 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] E. McCreight. A space economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[9] S. Needleman and C. 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. Pearson and D. Lipman. Improved tools for biological sequence comparison. *Proc. National Academic of Sciences USA*, 85:2444–2448.

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

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

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

[14] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[15] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.