PARALLELISATION OF EST CLUSTERING

Pravesh Ranchod

A dissertation submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science

Johannesburg, 2005

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the degree of Master of Science in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

Pravesh Ranchod
23 rd day of March 2005

Abstract

The field of bioinformatics has been developing steadily, with computational problems related to biology taking on an increased importance as further advances are sought. The large data sets involved in problems within computational biology have dictated a search for good, fast approximations to computationally complex problems

This research aims to improve a method used to discover and understand genes, which are small subsequences of DNA. A difficulty arises because genes contain parts we know to be functional and other parts we assume are non-functional as there functions have not been determined. Isolating the functional parts requires the use of natural biological processes which perform this separation. However, these processes cannot read long sequences, forcing biologists to break a long sequence into a large number of small sequences, then reading these. This creates the computational difficulty of categorizing the short fragments according to gene membership.

Expressed Sequence Tag Clustering is a technique used to facilitate the identification of expressed genes by grouping together similar fragments with the assumption that they belong to the same gene.

The aim of this research was to investigate the usefulness of distributed memory parallelisation for the Expressed Sequence Tag Clustering problem. This was investigated empirically, with a distributed system tested for speed against a sequential one. It was found that distributed memory parallelisation can be very effective in this domain.

The results showed a super-linear speedup for up to 100 processors, with higher numbers not tested, and likely to produce further speedups. The system was able to cluster 500000 ESTs in 641 minutes using 101 processors.

Acknowledgements

I would like to thank my Supervisors, Dr. Scott Hazelhurst and Dr. Anton Bergheim, for time spent understanding the area and providing vital advice regarding the direction, detail and scope of this research. I would also like to thank Zsuzsanna Lipták for spending time familiarising me with the field.

I also appreciate the support of my family and the advice and encouragement of Leonard Hagger.

I would also like to thank Mike Mchunu and Professor Ian Sanders for their constructive input.

This research was supported by the NRF under GUN2050322.

Contents

Chapter 1

Introduction

In this chapter, the problem is introduced, showing the importance of finding faster methods for accomplishing Expressed Sequence Tag (EST) Clustering, placing this problem in context. A solution to this problem is then presented, with brief motivation. The results achieved by this solution are then shown.

1.1 Genetic Information

Hereditary information is stored by a molecule called Deoxyribonucleic Acid (**DNA**), present in all animal cells. DNA can be seen as a linear string, with genetic information stored by the sequence in which letters appear in the string.

DNA is used as a template to create proteins, with complex biological processes accomplishing the translation to protein. It is known that different parts of the DNA are translated into proteins in different cells and at different times. This is called "Gene Expression". The parts of the DNA that are translated are said to be "expressed". The problem lies in discovering which parts of the DNA are expressed in which cells and at what times. This is presented in more detail in section 2.1.

We currently have no method of inferring gene expression information from the sequence itself. Thus, even with a complete DNA sequence we cannot tell what proteins will be produced in particular cells at particular times. In order to obtain this information, we need to capture a record of the expressed portions of the DNA from the cell itself.

In order to accomplish the translation from DNA to protein, the cell must remove the parts that are not expressed, thereby creating a new set of sequences called **mRNA** containing only the expressed parts. By capturing these new sequences, we can obtain a record of what we think are the useful portions of the original DNA sequence.

Capturing the expressed sequences is made difficult because they are extremely unstable and therefore difficult to sequence. Because of this, the expressed sequences are sequenced as a large collection of unordered fragments. The fragments are known as Expressed Sequence Tags (**ESTs**) and are short substrings of DNA, being fragments of expressed DNA produced naturally. This is extremely useful because it allows us to identify which parts of

the DNA are expressed by examining the ESTs that are produced.

1.2 Problem Introduction

1.2.1 Definition of Expressed Sequence Tag Clustering

Expressed Sequence Tags are collected as an unordered set of DNA fragments. In order to accomplish some ordering of this data, we attempt to find out which fragments are derived from which mRNA sequence, as a first step in discovering the original mRNA sequence. The problem is to take a set of input ESTs and group them according to the mRNA they are originally from.

This is accomplished by taking into account the fact that each mRNA is covered by a large amount of overlapping ESTs. This is useful because it means that we can find sequences that display a large degree of similarity, indicating that they overlap to a large degree, and hence are from the same mRNA. Furthermore, it is likely that we can do this for any two non-overlapping ESTs in an mRNA as they will be connected to each other by a chain of overlapping sequences. Thus, grouping together ESTs according to their similarity would in effect group them according to the mRNA they are originally from.

The method we use to calculate similarity between the ESTs must satisfy the condition that any two ESTs which overlapped on the original mRNA must be calculated to be similar, and any that did not must be dissimilar. This similarity is then used as the criterion governing whether two sequences will be grouped or not. However, there is no agreement on the best way to measure similarity in ESTs, with many different similarity measures being in common use.

The EST Clustering problem is solvable, but is an extremely computationally intensive task because of the large sizes of the input sets. The complexity of the solution is quadratic, which becomes a problem as the input sizes can be extremely large.

The EST Clustering problem is explained in greater detail in section 2.2.1.

1.2.2 Proposed Solution

The aim of the research was to investigate the usefulness of distributed memory parallelisation to reduce the time needed to perform EST Clustering.

Because EST Clustering can be used to indicate which parts of DNA are expressed at what times in particular cells, it is a process routinely performed by biologists. The existing algorithms to accomplish this are extremely time consuming, limiting the usefulness of this technique. The ultimate aim was thus to reduce the time taken to perform EST Clustering, with parallelisation chosen as a likely technique to achieve this.

1.2.3 Motivation for Parallelisation

The solution proposed to reduce the time needed to perform EST Clustering involved using distributed memory parallelisation to share the computation among multiple processors.

This approach was chosen because the time taken for an algorithm to complete is closely related to the speed of the underlying hardware. By making use of parallel computation, the computational capability of the hardware can be increased dramatically if a good method of dividing the work is found. By using a parallel algorithm which returns the same results as an existing sequential one, a reduction in the time taken for the algorithm to execute can be achieved while leaving the accuracy of the results unaffected.

It is known that parallelisation works best on independent tasks. A smaller amount of data dependencies results in a larger gain from parallelisation [Subhlok 1993]. This means that parallelisation works best when the computations performed by the various machines do not depend on results from computations performed on other machines. This is because cooperating machines have no need to wait for the results of other operations before completing their own tasks. EST Clustering is suited to parallelisation because the task consists of numerous comparisons performed between ESTs, with no side effects from previous comparisons. This means that each comparison can be considered to be an isolated event, with no artifacts from other operations affecting its result, allowing us to conclude that work can be divided such that there will be very few data dependencies.

One bottleneck in the process of distributed memory parallelisation is the distribution of the input sets to the various processors. With a complex algorithm, this step becomes less significant because the computation time grows much faster than the input set size. The time taken to distribute this input set is then much smaller than the computation that must be performed on it for large input set sizes. This is magnified by the fact that all we are interested in here are large input set sizes, as small input sets can be processed in a reasonable time by the existing sequential algorithm.

The reason a distributed memory approach was chosen over a shared memory approach is that there is a large cost saving in that networks of machines cost less than single machine multiprocessors. Another factor is that researchers generally already have access to networks of machines, so there is no incremental cost to using these. Furthermore, some work already exists pertaining to the usefulness of a shared memory approach, making the unexplored area of distributed memory more attractive.

1.3 EST Clustering algorithm

The algorithm operates by initially assigning each EST to its own group. Thus, initially there will be as many clusters as there are ESTs. The ESTs are then compared against each other using some similarity measurement. If two ESTs are found to be significantly similar, their groups are merged, allowing larger groups to be built up. This technique follows the method used by Burke *et al.* [1999]. There are, however, two major differences, one being that the algorithm has been parallelised and the other being the strategy adopted regarding the similarity measurement method.

There is no similarity measure that is widely regarded as being the best measurement. This is partly because the measure must trade accuracy for speed, using heuristics and ap-

proximations that reduce the complexity of the algorithms. Because a parallel approach reduces the total time taken to run the algorithm, it might become feasible to use measurement methods that would otherwise be too expensive to consider. This gain might allow for more accurate comparisons. To facilitate this, the parallel system does not follow previous research by building the comparison method into the algorithm, instead including a plugin mechanism facilitating easy modification of the comparison system.

1.4 Results

The parallel implementation has been tested experimentally. The sequential and parallel clustering implementations were run using the same input sets to compare their outputs and computation times.

The output of the parallel system has been tested against that of the existing sequential implementation, returning identical results in all test cases. This shows that the parallelisation has not adversely affected the results, and is an expected result as the system was designed with identical results as a necessary condition.

The parallel system returned a very good speedup, with efficiency increasing for large numbers of sequences. The sequential algorithm took approximately 25 days to cluster 300000 sequences, whereas the parallel algorithm running with 101 machines (a server and 100 clients) took about 6 hours. The results allowed us to conclude that distributed memory parallelisation makes efficient use of the extra processors when used for EST clustering.

1.5 Limitations

The experiments were carried out using an otherwise unused network, ruling out the effects of external network traffic. The parallel system does rely on the network for communication and would thus be adversely affected by an increase in traffic. All machines were also located on the same subnet, allowing for the use of broadcasting to minimize communication costs. While multicasting could be used across multiple subnets, it is not yet widely supported and it will affect communication time.

A further limitation arises because the results were obtained using a particular similarity measurement. The results will vary when using different similarity measurement methods, so the results gained from this research cannot be used to infer that the approach will be useful when using other methods. It does however indicate that it is likely that algorithms with a complexity close to or above that of the measure used will benefit from the parallelisation. This is because the total time taken for the parallel algorithm to complete is made up of computation, communication, coordination and data distribution. Communication, coordination and data distribution are the overheads involved in the parallel algorithm, whereas computation must be performed by both the sequential and the parallel algorithms. With a larger amount of computation, the overheads are relatively smaller, making the parallelisation more efficient.

1.6 Contribution to Genetic Research

EST Clustering is an important problem as it can be used to yield valuable information relating to what parts of a DNA strand are expressed at particular times in particular places. This is especially useful for the identification of function in genes. The clustering of ESTs is also indispensable for gene structure prediction, gene discovery and gene mapping [Claverie 1997]. This means that EST Clustering is a necessary step toward identifying the functions of parts of DNA.

Furthermore, this clustering can be used to consolidate short sequences into larger assembled sequences. With EST data being by nature a fragmented version of a full length gene, this is especially useful for yielding a full-length gene. This is an important process, as it utilizes captured data to yield knowledge. However, the methods used to accomplish EST Clustering are computationally expensive, causing the time taken to become prohibitive.

Since this categorization is just one step in a larger sequential process performed by molecular biologists to glean useful information from collected data, increased efficiency will relax a significant bottleneck in information processing.

1.7 Document Structure

The rest of the document is structured as follows :

- Chapter 2 presents a brief explanation of the biology necessary to understand this research, along with the background work pertaining to the methods chosen.
- Chapter 3 provides a detailed look at the objective of this research, stating the hypothesis and outlining the experimental methodology of this research.
- Chapter 4 shows the experiments that were conducted and justifies their use in terms of the aims of the research.
- Chapter 5 presents and explains the results of the experiments conducted.
- Chapter 6 concludes the document, drawing inferences from the results presented, presenting some future work and outlining the significance of the research.

Chapter 2

Background and Related Work

In this chapter some of the knowledge that has already been gained in related fields is explored. First, some introductory genetics is presented in order to allow a reader unfamiliar with the field to understand the basic concepts used in this document.

Related work pertaining to the problem of grouping sequences is presented next. First, work pertaining to the various methods of calculating similarity between sequences is presented, followed by background material pertaining to the grouping mechanism.

Lastly, background material relating to parallelisation is presented to facilitate understanding of the methods used in this research and the reasons they were chosen.

2.1 Biological background

In this section, the biology necessary to understand the problem is presented, along with a restatement of the problem in biological terms. Firstly, the basic terms are defined, followed by a detailed description of the domain of this problem.

2.1.1 Introductory Genetics

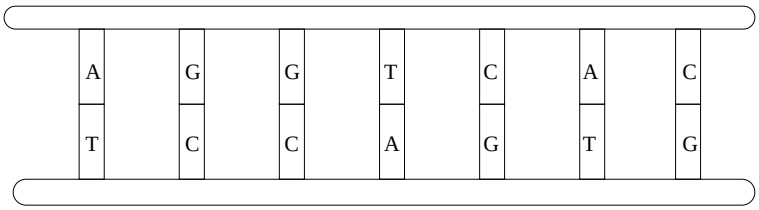

Figure 2.1: Untwisted DNA Molecule

Hereditary material in animals is stored in molecules called Deoxyribonucleic Acid (**DNA**). DNA molecules are made up of a double-helix structure, resembling a twisted ladder. The information is stored in the sequence of the particles that make up the "rungs" of the ladder called "bases". There are four different types of particles that are present in

these rungs – adenine, guanine, cytosine and thymine. These are represented by the letters ‘A’, ‘G’, ‘C’ and ‘T’ respectively. Each rung is made up of two of these, with a complementary relationship existing between the two halves such that if one half is adenine, the other must be thymine, and if one half is guanine, the other must by cytosine. This entire system can be abstracted such that it can be represented as a string made up of an alphabet of the four base characters. In figure 2.1, the untwisted DNA molecule represented by the string ‘AGGTCAC’ is presented. This string is complemented by the string ‘TCCAGTG’. DNA determines many aspects of our anatomy and physiology, as it controls the production of proteins that determine the structure and functions of living organisms [Koza and Andre 1996].

The function of a majority of the DNA contained in a strand is not known. The parts whose functions are known are found in **genes**, which are defined to be a portion of DNA that fully specifies the production of a particular protein or RNA molecule. Genes are thus thought of as the smallest unit of hereditary material. A gene contains **exons** and **introns**. The exons are the portions of the gene that get translated into proteins, with the introns being seemingly junk sequences.

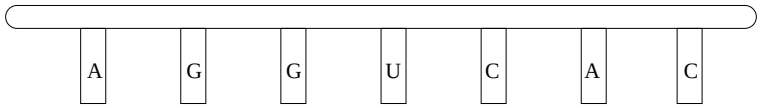

Figure 2.2: An RNA Molecule

Since the main purpose of DNA is to guide protein production, we will now look at the process by which proteins are produced. The information in the DNA strand is first unwound into its separate component strands. One of these strands is then used as a template from which a new single stranded molecule called RNA (Ribonucleic acid) is produced. This RNA corresponds to one strand of the DNA, but contains only the portions that make up genes, discarding the non-gene DNA. Another difference is that in RNA, Thymine is substituted for another particle, called Uracil, represented by the letter ‘U’. The RNA corresponding to the sequence ‘AGGUCAC’ can be seen in figure 2.2. The process that accomplishes this conversion is called “transcription” and results in a molecule known as mRNA Precursor.

mRNA precursor contains exons and introns making up genes. The introns are then removed in a process called “splicing”, which produces a type of RNA called mRNA or messenger RNA which contains only the functional parts of the original DNA. The process is illustrated in figure 2.3. This splicing process is unpredictable in that it sometimes removes some exons along with the introns, a phenomenon known as “alternative splicing”, illustrated in figure 2.4. The mRNA that is produced from this is a record of the genetic material that will be used to produce proteins in this cell.

Protein is then produced from the mRNA that has been constructed. Protein is the

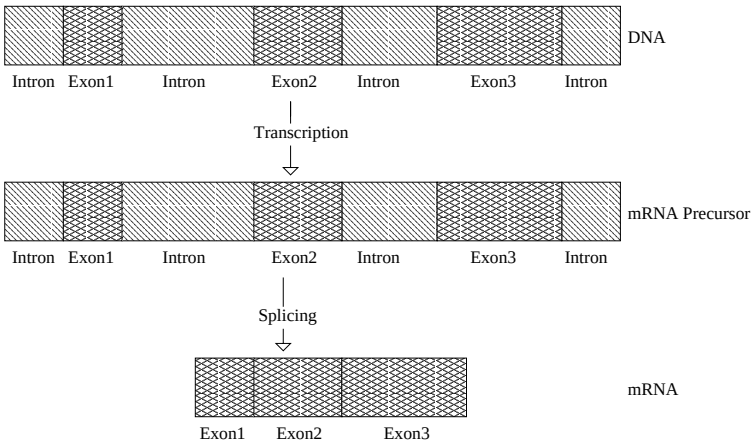

Figure 2.3: From DNA to mRNA

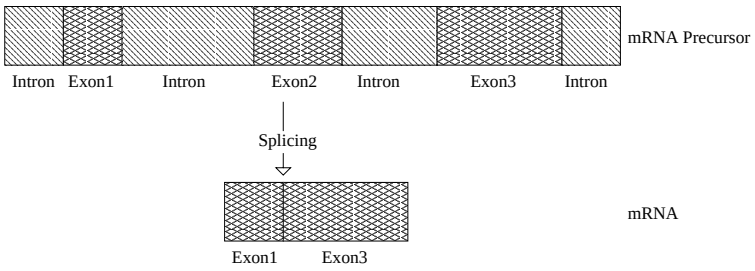

Figure 2.4: Alternative Splicing

determining factor in the structure and function of the organism.

2.1.2 Expressed Sequence Tags

Since the mRNA is considered to be a representation of the portion of the gene responsible for the synthesis of proteins, it is useful to be able to isolate it. However, mRNA is very unstable, in that it breaks apart quite easily and therefore difficult to sequence.

In order to sequence the mRNA, biologists make use of special enzymes to produce more stable cDNA from the mRNA. cDNA is a form of DNA made using an enzyme called reverse transcriptase. The process is the inverse of the usual process of transcription in cells because the procedure uses mRNA as a template to produce DNA. Unlike genomic DNA, cDNA contains only expressed DNA sequences, or exons, because it was generated from mRNA. However, the reverse transcriptase enzyme which facilitates the conversion does not function over the entire length of an mRNA, falling off after around 300-500 bases. The cDNA molecules produced are thus fragments from the original mRNA.

This cDNA is then sequenced to produce a single EST. This process is, however, quite error prone, resulting in a sequence read which deviates slightly from the original mRNA. This is partly because the sequencing is performed as a single-pass read, trading off accuracy

for quantity as more fragments can then be sequenced.

From a computing perspective, ESTs can be viewed as the strings generated by the grammar (A+C+G+T)*, and are generally constrained to a length of around 500 characters. These are a record of the exons that have been joined together by the removal of the introns. This is important as it gives us a picture of the functional genetic information in a cell. To summarize, ESTs are short sequences which comprise fragments from a full mRNA sequence rather than the sequence in its entirety [Aaronson *et al.* 1996].

A further complication arises because of alternative splicing which results in unpredictable transcription, with some exons being removed along with the introns [Mironov *et al.* 1999]. This alternative splicing needs to be taken into account when deciding on cluster membership.

One important characteristic of ESTs is that huge public databases of them exist. Since submission can be done by anyone, the ESTs are often error prone and badly annotated. The positive impact is that there is a lot of EST data available, and the error rates are higher than for other types of data but not unacceptable.

2.2 Clustering ESTs

In this section, the motivation and background relating to clustering expressed sequence tags is presented, along with an explanation of some of the factors that make this an expensive problem.

2.2.1 Introduction

EST clustering attempts to take the fragmented data presented as ESTs, and create maps of genes from them. The aim is therefore to categorize individual ESTs into clusters, such that each cluster contains only the information for a single gene, and each gene's information can be found in a single cluster [Burke *et al.* 1999]. The advantage of doing this is that it yields important data about the specific splicing events in that cell, information called "Gene Expression Data", and can be used to facilitate the early identification of genes.

EST Clustering is generally performed by utilizing some form of sequence comparison measure. Expressed sequence tags are clustered together if they display a large degree of similarity, making it likely that they are derived from the same gene. There is, however no agreement on a good measure of sequence similarity yet, with many vastly different methods being used to enumerate the level of difference between two ESTs. The purpose of EST Clustering is to identify gene sequences which come from the same gene family. This means that the genes share a common ancestor and is known as **homology**. So, we do not consider genes with slight mutations to be separate genes for the purposes of EST Clustering. We use similarity as an indicator of homology.

Some issues to consider when looking at sequence similarity are :

- Errors which occur in the sequencing process, when the individual EST is read.

- The phenomenon of Alternative Splicing, where a single gene can, in the process which separates the exons from introns units, discard some exons [Mironov *et al.* 1999]. Because of alternative splicing, it is generally considered to be a good idea to cluster ESTs if they seem to share a common exon. This is done by clustering ESTs if they share some substrings of a significant size called 'windows' which are similar.
- Mutations which alter the sequence, occasionally shifting the data forward or backward to accommodate new data or the loss of some data.

2.2.2 Clustering Techniques

In this section, some clustering methods are presented with an idea of their suitability for EST Clustering in particular.

Some characteristics of clustering techniques are discussed in Jain *et al.* [1999]. These include whether an algorithm operates by joining sets of initially disparate clusters, or splitting an initial cluster to produce the final set of clusters. These are known as "agglomerative" and "divisive" clustering algorithms respectively.

Another important characteristic of the algorithm is whether it generates clusters such that each element can belong to only one cluster. These algorithms are known as "hard" clustering algorithms. The other option is to allow a particular element to appear in many clusters, which is known as a "fuzzy" clustering. For EST Clustering,"hard" clustering needs to be performed as we need to be sure that each EST appears in only one cluster.

Algorithms can also be either "deterministic" or "stochastic". Deterministic algorithms always generate the same set of clusters for the same input, whereas stochastic methods have a random element which causes them to generate different results on different runs. EST Clustering should not contain a stochastic element, because the characterisation of an EST as belonging to a particular gene should not be changeable when the criteria for this decision are unchanged.

"Incremental" clustering is a technique whereby any additional elements that need to be considered are compared against the clusters that have already been created rather than against the original elements themselves. This is useful for large data sets, as it removes a lot of comparisons when we can compare against a cluster of elements rather than all individual elements.

2.2.3 Taxonomy of Clustering Algorithms

A taxonomy for clustering methods has been presented in Jain and Dubes [1988]. This taxonomy divides clustering algorithms into hierarchical clustering algorithms and partitional clustering algorithms.

Hierarchical Clustering

Hierarchical clustering constructs a graph of the similarities between sequences to yield clusters. "Single-link" methods construct a tree whose lowest level contains each sequence

in its own cluster, and represents the lowest similarity level at which no two sequences are similar. Each level of the tree represents a different similarity level, with subtrees joining at a particular level if the clusters represented by those subtrees would merge at the similarity level represented. It is thus possible to easily retrieve the set of clusters for a particular similarity level. "Complete-link" methods keep a set of graphs, with one graph for each similarity value. Each graph contains an edge between two nodes if the corresponding elements are closer than the value which the graph represents. This hierarchy of graphs can then be separated to yield the clustering corresponding to a desired similarity level.

Partitional Clustering

Partitional clustering is different from hierarchical clustering in that it produces a single set of clusters from the initial data. There is no hierarchy constructed and thus no way of changing the clustering similarity criterion once the clustering has been accomplished, short of reclustering. They are less computationally complex, making them attractive for clustering large data sets as is the case with EST Clustering. Most partitional algorithms, however, require us to have a fixed number of output clusters. This is not possible for EST Clustering, as it would require us to have prior knowledge of the number of genes the ESTs in our data set represent.

Nearest Neighbour clustering is an iterative method which can be used as the basis for clustering [Jain *et al.* 1999]. It works by symbolically joining elements to the element they are closest to, marking them as neighbours when the similarity between the two elements is below a certain threshold.

The clusters can then be created by finding the sets of mutual neighbours. This means that we construct neighbourhoods such that any element is in only one neighbourhood, and every neighbour of a particular element is in its neighbourhood. For the purposes of EST Clustering, this means that any sequence is in only one cluster, and all similar sequences are in the same cluster. In section 2.2.1 we stated that similarity indicated that sequences were in the same gene family. Thus we can see that nearest neighbour clustering would result in clusters which represented individual gene families. This method is functionally equivalent to "Agglomerative Clustering", wherein clusters are joined if any member of a particular cluster is similar to any member of another [Zhao and Karypis 2002].

For the purposes of EST Clustering, partitional clustering is attractive, as hierarchical clustering is not necessary, and would impose larger costs than partitional clustering would. This is because hierarchical clustering methods must result in a graph that represents enough information to construct the clusters for every similarity level. This representation would thus be a lot larger than that necessary for partitional clustering, which only needs to represent enough information to construct the clusters for a particular similarity level. Hierarchical clustering could have some value in order to determine which similarity level is best for satisfying the condition that any two sequences which are similar must be in the same gene and any two which are not in the same gene must not be in the same cluster. However, this value needs to be determined only once, so the general task of EST Clustering does not

require hierarchical clustering.

2.3 Similarity Measures

Since clustering is done by sequence comparison, it is important to ensure that the similarity measure used is both biologically significant and computationally efficient. A primary concern here is that for EST Clustering to be sensitive enough, it is necessary to search for sequences which display a similarity over only part of their length. Because of differing splicing events, some ESTs will have their exons in a different sequence to others, meaning that they will not be similar over their entire length even if they are from the same gene. However, we would like to categorize them as being from the same gene, leading us to rely on local similarity, which provides an indication that two sequences contain regions that are similar, indicating that they contain some of the same exons, and are therefore from the same gene.

2.3.1 Windowed comparison

One technique used to adapt similarity measures to calculate local similarity is windowed comparison [Horton 2001; Hide *et al.* 1994]. This process utilizes "windows" within which to compare. It works by partitioning the sequence into every possible subsequence of a specific word length. For example, the sequence 'AGGGCT' with word length 3 would yield the windows 'AGG', 'GGG', 'GGC' and 'GCT'. Every window in the first sequence is then compared against every window in the second sequence, leading to the multiple scores being calculated for each pair of sequences. The actual similarity measure is then taken as the distance between the most similar pair of windows.

A design decision that arises because of this is the choice for the size of the windows. A change in window size offers a chance to trade off sensitivity for selectivity and vice versa. A large window size would create a need for a similarity to be long, and so may miss shorter but still significant similarities. A small window size suffers from the opposite problem, as with too small a window size, the length of the similarity required to decide two sequences are similar will be too small, increasing the probability that sequences that are not homologous will be designated similar by chance. The computational advantages depend on the complexity of the similarity measure. In the experiments conducted in this research, a window size of 100 is used, in line with that used in previous research [Burke *et al.* 1999].

The number of windows in each sequence is proportional to the length of the sequence, as the window size is fixed.

A similarity measure must be chosen to operate within these windows. Similarity measures can be divided into two broad categories – those which use sequence alignment as a comparison tool and those which do not.

2.3.2 Alignment based Methods

Using sequence alignment involves lining up the two sequences to find out where they overlap. A score is then calculated which represents how good that alignment is, and can be based on factors such as the length of the overlap and the number of errors in the overlapping segment.

For the purposes of EST clustering, the alignment can be local rather than global. This means that we look for the highest scoring segments of the sequences rather than trying to find the score of the sequence as a whole. We do this because of alternative splicing, which we have discussed in section 2.2.1, as local alignments give us an indication of shared exons, which is sufficient for us to decide that two sequences belong to the same gene or cluster.

Alignment-based methods can miss some larger characteristics that make sequences similar [Vinga and Almeida 2003], but are widely used to compute sequence similarity. Edit distance [Allison 1992] and BLAST [Altschul *et al.* 1990] are alignment-based methods, as both attempt to find the overlaps between the sequences.

Edit Distance

The Edit Distance between two sequences s_1 and s_2 is defined as the minimum number of insertions, deletions and substitutions needed to transform s_1 to s_2. More formally, edit distance is defined as follows [Allison 1992]:

> Let Λ be the empty string, s, s_1 and s_2 be some sequences of characters, and ch_1 and ch_2 be some characters.
> $d(\Lambda, \Lambda) = 0$
> $d(s, \Lambda) = d(\Lambda, s) = |s|$ with $|s|$ being length of s

$$d(s_1 + ch_1, s_2 + ch_2) = min \begin{cases} d(s_1, s_2) & \text{if ch}_1 = ch_2; \\ d(s_1, s_2) + 1 & \text{if ch}_1 \neq ch_2; \\ d(s_1 + ch_1, s_2) + 1 & \\ d(s_1, s_2 + ch_2) + 1 & \end{cases}$$

This can be solved by a dynamic programming algorithm which runs in $O(nm)$ time for two sequences with lengths n and m [Needleman and Wunsch 1970]. This algorithm is generally not used for EST Clustering because the execution time is prohibitive, and more efficient alternatives which return similar results are available. It must be noted that this algorithm performs a test for similarity over the entire length of the EST. In order to find local similarity, it is necessary to compare every region of the first sequence to every region of the second sequence.

In practice, a weighting function is employed, giving a different penalty to each type of mismatch. This is more accurate because certain types of edits happen more frequently than others in the actual cells. Edit distance as shown above can be seen as assigning a penalty of 1 to each type of mismatch. When a weighting function is used, a different value is attached to the different types of edits. Furthermore, if an insertion has been performed at a particular

point on a sequence, the penalty for subsequent insertions is generally significantly reduced as an insertion of 1 character is not much more likely than an insertion of more characters.

The modification does not affect the complexity of the algorithm.

BLAST

BLAST (Basic Local Alignment Search Tool) [Altschul *et al.* 1990] is widely used for sequence comparison. BLAST attempts to approximate the maximal segment pair measure efficiently. This measure is an indicator of the difference between the most similar regions of the two sequences involved. This is useful for EST Clustering because of the possibility of sequencing errors, and alternative splicing. Specifically for alternative splicing, local similarity measures are able to detect common exons as high similarity regions, rather than discarding the possibility of the ESTs being from the same gene because their global similarity is too low.

The BLAST method of calculating a similarity score is based on assigning scores and penalties to the various parameters it must use to get two sequences to align. Thus, there will be penalties for gaps and mismatches, and rewards for matches. A score calculated this way can then be used as an indication of how similar two sequences are. BLAST finds maximal scoring pairs by comparing words of a user specified word length in one sequence to every word of the same length in the other sequence. When it finds a pair of words whose similarity exceeds a value calculated by BLAST from a user specified threshold, it extends the words in an attempt to find a pair with a score above the threshold that constitutes a match. The value calculated by BLAST is used to decide whether the pair is likely to have an extension that exceeds the threshold.

This method can be used to decide whether to cluster the ESTs involved, by deciding on a maximal scoring pair threshold that is necessary for a match. BLAST is able to operate more efficiently due to the ability to discard regions which are unlikely to exceed the threshold similarity score [Karlin *et al.* 1990].

BLAST is an extremely popular comparison method because it is extremely efficient, and provides a good approximation for edit distance.

2.3.3 Alignment free methods

Alignment free methods are methods for evaluating sequence similarity which do not require an alignment of the sequences. One indicator used in these methods is word frequency, although other methods such as chaos game representation have also been explored [Vinga and Almeida 2003].

In general, alignment free methods calculate some statistics regarding the frequency of words of a predefined length, calculating a similarity score based on the comparative statistics of the two sequences. This does not perform an alignment, and the theory is that it is able to capture some similarity which alignment does not.

d^2 distance

d^2 determines whether two sequences are similar by looking at the frequency with which each word of a particular size appears. The user selects a word size for d^2 to operate on. The program then counts the number of times each word of a length not exceeding the user inputted word size occurs in each of the sequences. The difference in the number of times a word is found in each of the sequences is the important factor here, with the d^2 score reported as the sum of the squares of the differences. This works as a dissimilarity measure, with a higher number indicating that the sequences share a smaller degree of commonality in terms of the words they contain [Hide *et al.* 1994].

The d^2 score is calculated as the weighted sum of the squares of the difference in word frequencies between the input sequences for all words whose length is not greater than a specified length. More formally:

- Choose a word length n.
- Let $\theta_s(x)$ denote the number of times word x occurs in sequence s.
- Let $x_{i,k}$ denote the ith word in a lexicographical ordering of all words of length k.
- Let $w(m)$ denote the weighting assigned to word m.
- Then,

$$d^2(a,b) = \sum_{n=1}^{u} \sum_{i=1}^{4^n} w(x_{i,n})(\theta_a(x_{i,n}) - \theta_b(x_{i,n}))^2$$

However, because we are looking for regions of high similarity which would indicate shared exons, we calculate the similarity within windows as described in section 2.3.1. For a user-specified window size, the d^2 comparison is performed between every window in one sequence and every window in the other. We thus have numerous scores calculated between two sequences. The d^2 score for the sequence as a whole is taken to be the minimum of all the scores between the individual windows [Burke *et al.* 1999].

This requires the calculation of the d^2 similarity measure between every two subsequences of a specific length and is therefore quadratic in terms of the number of windows, and therefore in the sequence length.

In current implementations of d^2, this specification is not followed closely, with some elements being discarded to save computation time. There is no word weighting mechanism implemented, giving all words an equal significance. This is equivalent to setting the weighting of all words to 1.

Another decision made by current implementations is to discard the word sizes less than the maximum. The calculation is thus done using only words of exactly the maximum length.

The revised d^2 calculation thus looks as follows:

Choose a word length n.
Let $\theta_s(x)$ denote the number of times word x occurs in sequence s.

Let x_i denote the ith word in a lexicographical ordering of all words of length n.

$$d^2(a,b) = \sum_{i=1}^{4^n} (\theta_a(x_i) - \theta_b(x_i))^2$$

The choice of word length is also an important one, with different word sizes yielding different results. Previous research has focused on a word size of 6 and a window size of 100 [Hide *et al.* 1994].

2.4 Clustering Algorithm

A clustering algorithm groups sequences according to their similarity. We have already discussed similarity, and now move on to background work relating to the grouping technique.

2.4.1 d2_cluster

Burke *et al.* [1999] presents a method for accomplishing EST clustering according to the results obtained from a d^2 comparison. This algorithm, called the d2_cluster algorithm, has been shown to be both relatively fast and accurate. d2_cluster works by running through the following steps.

- First a **threshold** score is derived from a user definable **Stringency** setting which indicates, as a percentage, the required level of similarity in order to consider two ESTs similar enough to be clustered.
- Initially, each EST is assigned to its own cluster, so there exist as many clusters as there are ESTs.
- Then, compare each EST i to every other EST j
  - If $d^2(i,j)$ <**threshold**, merge the cluster containing i with the cluster containing j. For details of d^2, see section 2.3.3
- The clusters left after this process are the target EST clusters [Burke *et al.* 1999]. The clustering method used can be seen as equivalent to agglomerative clustering.

This algorithm operates in $O(n^2)$ time in relation to the number of strings involved in the clustering, multiplied by the complexity of the measurement method.

2.4.2 wcd

Another clustering method called wcd [Hazelhurst 2003a,b] uses d^2 as its measure. It constructs the same clusters as d2_cluster, but uses a different method to do so. In order to minimize the execution time of the clustering process, a heuristic is used. Before a pair of sequences are compared using d^2, they are first examined to see whether they share at least

5 non-overlapping common words. It has been empirically determined that this is a safe criterion to use as it is unlikely that any sequences that do not share at least 5 non-overlapping common words would have a d^2 score below a threshold of 40 used by wcd.

Another interesting characteristic of wcd is the use of compressed sequences, which allow it to hold more sequences in memory for increased efficiency. Like d2_cluster, wcd uses agglomerative clustering as its clustering technique.

The clustering algorithm employed by wcd is presented below.

Algorithm 1 The wcd clustering algorithm

```
Let n be the number of sequences
initially, assign every sequence to its own cluster
for i = 1 to n do
  for j = i to n do
    if sequence i is not in the same cluster as sequence j then
      if sequence i has a chance of being similar to sequence j then
        if d^2 score between sequence i and sequence j is below the threshold then
          merge the clusters containing the two sequences
        end if
      end if
    end if
  end for
end for
Output clusters
```

2.4.3 TIGR

TIGR, a clustering system named after The Institue for Genomic Research which developed it, is a system that seeks to produce an index of the sequences represented by EST Sequences [Quackenbush *et al.* 2000]. TIGR's goal is not EST Clustering, as its primary focus is the assembly of the ESTs into full genes. The assembly of ESTs into genes is a process whereby a sequence is constructed which is the shortest sequence from which all the ESTs could have come, and represents the mRNA sequence from which the ESTs were originally produced. It uses EST Clustering as one step in this process.

TIGR uses an incremental system, building on the results of previous assemblies to produce its new assembled sequences. Initially, it uses reference sequences to help classify ESTs. This approach is known as "Supervised Clustering".

TIGR works by using mRNA sequences as reference sequences. It adds these already assembled sequences to its database, for later comparison to ESTs. It then gets its EST data from a database, and performs a pairwise comparison between all the sequences it now has using the edit distance metric, checking for local similarities. If any pair are found to be sufficiently similar, they form a cluster. When an already assembled sequence appears in a cluster, it is separated into the sequences that were used to assemble it. New mRNA sequences are then assembled from these clusters, and added to the database of assembled sequences.

TIGR is different from wcd and d2_cluster in that it has as its goal the creation of assembled genes rather than clusters of ESTs. The use of an incremental approach is also a vital difference, as the comparison to assembled mRNA rather than the original ESTs could yield different results.

2.4.4 UniGene

Unigene [Schuler 1997] is a system for clustering Expressed Sequence Tags. Like d2_cluster, it does not attempt to assemble the sequences into mRNAs, instead producing clusters of ESTs as its final goal.

Unigene uses a similar approach to that of TIGR, using mRNAs as reference sequences, but uses some further knowledge to help produce correct clusters. First, UniGene gets clusters of genes and mRNAs from a database. These serve as reference. Next, the ESTs are introduced, and compared to the existing clusters using BLAST as a similarity measure.

Now, one of the ends of an mRNA, known as the 3' end, is a recognizable sequence, and can thus be identified by the algorithm. This is useful, as in order for a sequence to constitute a full mRNA, it must have this recognizable sequence on its end. So, when we compare sequences to clusters, and find that a sequence seems to belong in two separate clusters, we know that this cannot be so because the mRNA resulting from this would then have to have two distinct 3' endings.

Furthermore, we know that any cluster that does not contain an EST with a 3' end is incomplete and cannot represent a full mRNA, so we discard that cluster. At some point, ESTs are likely to be added which will complete that sequence, at which point its cluster will be added.

The resulting clusters are known as "anchored clusters" because their 3' end is known. This anchoring of the clusters is a major difference when compared to the other clustering systems, and makes clear that the goal of UniGene is the construction of clusters which fully describe the genes they are derived from. d2_cluster, wcd and TIGR allow EST clusters which do not have a 3' end, showing that they are incomplete clusters. This means that they allow clusters which do not represent an entire gene, but a portion of one.

2.4.5 Conclusion

The comparison methods presented here have different characteristics with regard to EST Clustering. TIGR is a system for assembling ESTs into full length gene transcripts. Its goal differs from that of this research in that it attempts to construct mRNA sequences from the ESTs, rather than just grouping the sequences. This is an important distinction, as the stated goal of grouping ESTs from the same gene family means that the desired clusters would not necessarily assemble into a single gene sequence.

Unigene is inappropriate for a similar reason. Anchoring the sequences by their 3' end, and then not allowing clusters to have multiple 3' ends means that we will exclude sequences from the same gene family. Modifications to a gene close to its 3' end will cause

them to end up in different clusters, when a similar mutation further away will not. This is not desirable behaviour.

d2_cluster can be used to find the set of gene families, but is less efficient than wcd, which was the method adopted for this research.

2.5 Parallelisation

Parallelisation is the process whereby work is split up among a number of processors with the goal of reducing execution time. In this section issues relating to parallelisation will be discussed with reference to related work.

2.5.1 Parallel Models

Parallel systems are generally classified according to Flynn's Taxonomy [Flynn 1972], which classifies parallel systems based on the number of different instruction streams and memory pools. Three main classifications of interest are :

- Single-Instruction Single-Data (SISD)
- Single-Instruction Multiple-Data (SIMD)
- Multiple-Instruction Multiple-Data (MIMD)

SISD systems consist of a single instructions stream with a single data stream, and is thus a sequential computer. SIMD systems consist of a number of processors executing the same instructions at the same time, but each processor has its own data stream. This does not mean that they have a different memory pool, as all data streams could be taken from one shared memory pool. Multiple data streams refers to the flow of that data, and indicates that while the processors are executing the same instructions, they are executing those instructions on different inputs. MIMD systems consist of multiple processors, each processor has a separate instruction stream as well as operating on a separate data stream.

2.5.2 MIMD Systems

MIMD systems can be further divided into two distinct categories. Systems where all processors have a single memory pool from which the data streams are read are known as shared-memory systems whereas systems where each processor has its own memory pool are known as distributed-memory systems [Levine 1994]. This affects the communication between processes as well as the distribution of the input data.

Parallel programs written for MIMD systems can employ one of two strategies for communication between processors. These are the shared-memory model and the message passing model [Chong and Agarwal 1996]. It is important to note that this is independent of the physical memory system employed, as it is possible to use a shared memory programming model on a distributed memory system, and is also possible to use a message passing model on a shared memory system.

In the shared-memory model, the actions of individual processes can be observed by their effect on the shared memory pool. This is extremely fast, as there does not have to be explicit communication between processors. However, the memory generally needs to be explicitly synchronized, to avoid the problem of race conditions and overwriting [Yang 1993]. This synchronization cost increases markedly as more processors are introduced, making these algorithms less scalable than message passing algorithms.

The other communication model is known as the message passing model. Message passing systems do not use a single pool of data to operate from. Instead, each processor operates on its own local store, with messages passed between the machines in order to share resources and results. These have been shown to be more scalable than shared-memory machines. They do, however, require more programming effort as there is a need to ensure consistency in the different memory spaces [Nikolopoulos and Polychronopoulos 2001]. The model minimizes the need for synchronization as the individuals involved do not need to be given the ability to force the overwriting of data on another individual. The problem, however, is that in a domain where a large amount of data will need to be distributed, the processors spend a lot of idle time waiting for input from other machines [Yang 1993]. It is possible for a message passing implementation to be faster than a shared memory implementation. This is because the cost of ensuring cache consistency and data integrity can overwhelm the overheads involved with the mechanics of message passing. This result has been experimentally verified for certain problems [Lew 1995].

2.5.3 Distributed Systems

Distributed Systems are systems wherein multiple processors with seperate memory are used to cooperate in solving a particular problem. Since these machines have no single pool of memory, the message passing model must be used. While it is possible to simulate a shared pool of memory, the message passing model results in less communication [Martonosi and Gupta 1989]. Distributed computing has become an attractive idea thanks to the prevalence of extremely cheap, powerful desktop machines. This could prove to be a more cost effective solution than having specialist multiple processor shared memory machines.

In order to accomplish multiprocessing on a network, it is essential to reconsider issues like security and efficiency in your systems. Decisions which may have been appropriate for a standard network may hamper parallel computing. The handling of all these issues for efficient parallel computing is known as ‘clustering’, not to be confused with EST Clustering. Some policies which may need to be changed include remote login facilities on slave machines. In order to preserve efficiency, clusters also generally isolate the local network so that it is invisible from the outside, thereby preventing unnecessary traffic [Brown 2000].

2.5.4 Design issues in distributed systems

The use of a distributed system introduces tasks which are not present in sequential algorithms. The distributed system must allow the various processors to communicate, it must ensure all necessary data is distributed to the processors, and ensure that work is fairly allocated so as to minimise the idle time of the processors.

Work allocation

It is important to allocate work so as to minimize the total idle time of all processors in the distributed system.

A choice must be made as to whether to allocate the processing statically or dynamically [Kim *et al.* 1997]. Static allocation of work entails dividing the work among the processors as the system starts up. Dynamic allocation is adapative, dividing the work while the system is running. Static allocation achieves a good distribution of work if the size of the tasks to be performed and speed of the hardware to be utilized is known beforehand. Otherwise it does not achieve a very good balance. Dynamic scheduling allocates work during the execution of the parallel algorithm. This allows it to be more adaptive in its allocation, taking into account the the amount of work the processors have to do [Mao and Tu 1995]. Dynamic scheduling is more suited to the purpose of EST Clustering, because the execution time of the tasks involved is not known beforehand, and can vary greatly in relation to each other. This is because of the use of a heuristic which can reduce the execution time dramatically.

An important decision arising from the use of dynamic allocation concerns the responsibility for the division of work. It is possible to have a central coordinator that is responsible for the division of work. This is known as the master/slave model of distributed computation. Another method involves a system of cooperating peers [Oden and Patra 1994].

The master/slave architecture is simpler to implement, with a single master responsible for control of the system. It has the disadvantage that the master machine could become a bottleneck, primarily because of communication overheads [Bonacina 2000]. Using a system of cooperating peers is an alternative strategy, wherein there is no single machine that would become the bottleneck. It is, however, more difficult to negotiate control.

The EST Clustering consists of a large number of expensive, independent calculations. It thus has few data dependencies, and the order of computation is unimportant. The master/slave model is well suited to this type of problem [Shao 2001] and was thus chosen for this research.

Communication

Distributed systems involve multiple processors acting in a cooperative manner. This makes communication an important concern. Various techniques exist to accomplish information sharing between the various processors in a distributed system.

Remote Procedure Call (RPC) and message-oriented communication are two important techniques which have some suitability for a distributed system for EST clustering. These

techniques are discussed below.

Remote Procedure Call [Birrel and Nelson 1984] is a technique whereby procedures can be executed on remote machines. Machines which need a remote procedure executed must know the name of the procedure and the machine responsible for executing it. The Remote Procedure Call mechanism takes care of the mechanics of the communication required to transmit the function parameters and the procedure results between the machines.

The mechanism does this by wrapping the parameters and responses into network messages which are transmitted via an existing underlying network. An important characteristic of Remote Procedure Call is that communication is synchronous. This means that the process requesting a remote procedure becomes blocked or idle while waiting for the results of the operation. This idle time is undesirable as it increases the time taken for the task to be completed.

While extensions of RPC exist which allow asynchronous communication, it is often easier to use message-oriented communication [Tanenbaum and van Steen 2002].

Message-oriented communication [Tanenbaum and van Steen 2002] is a technique whereby processes communicate by sending messages to each other. There is no built-in mechanism for requesting work to be performed by remote machines. This must be added to the programs using this form of communication, by designating certain messages to be requests for work, and standardising the message format.

Message-oriented communication can be either transient or persistent. Transient communication requires that the receiver be ready to receive a message when the sender is sending it. When persistent communication is necessary, the communication system must hold the message until the receiver is ready to receive it.

Another issue is whether communication should be synchronous or asynchronous. When using synchronous communication, the sender is blocked until the message is delivered [Eklund 1998]. Synchronous communication is not necessary for this system, and is a waste of processor time. Asynchronous communication, which allows the sender to continue immediately after sending a message, is more attractive. Achieving asynchronous communication requires some sort of local buffer or intermediate server to hold the messages until the receiver is ready to process them. The approach used in this research was to use transient asynchronous communication with a local buffer on each machine.

A factor in favour of a successful parallelisation is that the EST Clustering consists of many independent jobs. This lack of data dependencies means that there is a smaller amount of necessary communication [Sheliga *et al.* 1998].

Data Distribution

Where data needs to be accessed by multiple machines, as is the case for a distributed system for EST Clustering, there are three strategies that can be employed. These are the central-server algorithm, the full-replication algorithm and the read-replication algorithm [Richard and Singhal 1993].

The central-server algorithm dictates that a central server is responsible for managing

the data. It contains the data in its memory, and client machines request the data they require from it. The advantage of this approach is that maintaining data consistency across the memory space is easy as all data is stored in a central location. The major disadvantage of this approach is that the cost of a data access in a client is the cost of the transfer of that data item over the network. This is typically much larger than the cost of an access to local memory.

The full-replication scheme works by copying data to every machine involved in the system. All machines thus access data items from local memory. The advantage of this approach is that the cost of a memory access is the same as for a sequential program, and is much lower than the cost of a memory access. A major disadvantage is that maintaining data consistency is very difficult, as a change to a data item requires all machines to update the copies in their local memories.

Read-replication is a hybrid approach whereby data is distributed across the set of machines, but a central server controls writes to the data items. This means that all machines can read the data, but only one can modify the data. This scheme has the advantage of a low read time on a data item. A write to a data item is more expensive, as it requires a network access, but because there is now a central server responsible for maintaining data consistency, this task is easier.

None of these techniques is the best in all situations, with the characteristics of the data that must be distributed determining which scheme is the most efficient. Data which is not going to be modified often would benefit from read-replication or full-replication, as data item reads are much faster than under the central server method. When data must be written to often, the central-server approach would be more effective as it is easier to maintain data consistency using this scheme than any other. Because of these factors, a hybrid approach was chosen, with data which did not need modifying being distributed using full-replication, and data which required mostly writing and not much reading being managed by a central server.

2.5.5 Implementation of parallel algorithms

Distributed systems, and indeed parallel algorithms in general, are difficult to implement. However, tools exist to make this task less complex than it previously was.

The Parallel Virtual Machine [Sunderam 1990] is a framework for parallel computing which provides functionality that makes it easier to implement a parallel algorithm. The framework hides the details of communication and some coordination from the programmer, with the goal of making it appear that a collection of processors are a single computing resource.

The Message Passing Interface (MPI) [Dongarra *et al.* 1992; Forum 1994] is a standardised set of cross-platform communication tools that provide a standard way to communicate with parallel programs. This is useful when a parallel algorithm is implemented on multiple platforms and must interact across those platforms. MPI also implements higher level communication tools to allow communication across groups of processes. MPI was not used for

this research because the system involved was not intended to run on multiple architectures, and standard communication tools were sufficient.

Deshpande and Schultz [1992] discusses a general framework for parallel computing which minimizes programming effort on a parallelisation task by integrating parallel features into sequential languages. This system, called Linda, maintains a common shared memory space administered by the Linda system itself, without the intervention of the programmer. The central system is allowed to create processes on remote machines, and all processes can access the shared memory space. A common method of using this system is to have worker processes which run on slave machines reading and processing tasks from the shared memory space, and grabbing new tasks as they have processing power available. The major advantage of this system is that it reduces programming effort. Linda was not used as the potential for increased efficiency in a custom system outweighed the extra programming effort.

Ramanujam and Sadayappan [1989] discuss optimizations that can be made regarding both data partitioning and optimal communication. A mechanism is outlined for the partitioning of data after constructing a model of the data dependencies involved. This is necessary because if the data dependencies are not taken into account, incorrect results could be returned as results from operations that have not been executed yet may be needed. The EST clustering problem is easily divided because there are few data dependencies. The pairwise computations that can be done here are independent, meaning that they do not require results from previous computations.

2.5.6 Performance

In this section, the issues governing the performance of a parallel system will be highlighted. Speedup and efficiency are two of the most important measures of the performance of a parallel algorithm. The speedup demonstrated by an algorithm running on p processors is defined as the ratio between the time taken for the sequential algorithm to complete and that taken for the parallel algorithm to complete on p processors. The efficiency of the algorithm executed on p processors is the speedup on p processors divided by p [Quinn 1987, page 19].

One way of determining the speedup achievable from a parallelisation is Amdahl's Law [Quinn 1987, page 19] which relates the possible gain from parallelisation to the cost of the operations that are inherently sequential. The relation established is presented below.

> Let f be the fraction of operations that must be performed sequentially. The Speedup S that can be achieved by a machine with p processors can thus be constrained as:
>
> $$S \leq \frac{1}{f + (1 - f)/p}$$

From this equation, it can be seen that a factor constraining the maximum speedup achievable by a parallel system is the fraction of sequential operations. For instance, if the fraction

of sequential operations is 0.1, the maximum value for S is 10, regardless of the number of processors used.

A discussion of the effect of this on the EST Clustering problem is presented in section 3.3.2.

2.5.7 Case studies

d2_cluster has been successfully parallelised using the shared memory model, with results showing that the work can be effectively shared on a shared memory machine. The clustering was performed on an SGI Origin 2000 multiprocessor. The speedup achieved by the algorithm when executed on a 126 processors was 100 [Carpenter *et al.* 2002]. This parallelisation made use of sequential merge operations, indicating that the merge is a simple enough operation to not become a bottleneck for a single processor.

A parallel system has also been designed to do EST Clustering using sequence alignment as a comparison method [Kalyanaraman *et al.* 2002]. This system is also built for shared memory machines, and when run on 32 processors, produces a speedup of 8.3 over the 2 processor parallel version. This indicates an efficiency of 0.5. Unfortunately, no timing results are shown for a single processor version of the algorithm. The system constructed here has different constraints to a d^2 system, as it uses a completely different algorithm to accomplish the comparisons.

The algorithm makes use of a master/slave paradigm, with the master assigning calculations to individual processors and storing the result. Then, in a post-calculation stage, all merges are performed at once. Because the memory space required to store the results of the comparisons is prohibitive, this is modified slightly so that there are alternating comparison calculation and merge stages, meaning that the results of the comparisons only need to be stored until the next merge stage.

This algorithm uses a heuristic to identify promising pairs. The heuristic is also parallelized, so that the computation performed by slaves can be either determining whether a pair of sequences is promising or performing an alignment on these sequences.

Yap *et al.* [1998] present a comparison between parallel methods of Human Genome Database searching. One method uses a master-slave method, wherein the master assigns calculations that need to be performed to slave machines, who then return the results of their computation to the master. This allocation is done dynamically, so that the load on the various slaves is balanced. The other method partitioned the problem by assigning calculations to the slaves at startup. The results showed that assigning the calculations at startup achieved better performance as the master became a bottleneck in the master-slave system. The master became a bottleneck for short sequences, with performance for sequences of the average length of ESTs (approximately 300) showing a speedup of approximately 100 on 128 processors, and performance on sequences of length 50 showing a speedup of only 32 on 128 processors.

2.6 Conclusion

These papers collectively indicate that parallel d2 clustering could be successful. We have seen that the sequential component of d2 clustering is not large enough to be a problem [Carpenter *et al.* 2002], We have seen that a master-slave system can be tailored to allow impressive performance, as in [Yap *et al.* 1998], and that when there is a small amount of data dependencies, master/slave parallelisation is useful [Shao 2001].

Since parallel systems involve many different design choices, some of these have also been discussed to justify the approach outlined in chapter 3. Experimental results from the parallelisation of other algorithms has also been presented to support the results shown in chapter 5.

Chapter 3

Proposed EST Clustering System

3.1 Introduction

This section will be used to state the formal hypothesis, and the formulation of the experiment used to test it. It will thus present the project more clearly, giving details of the experiment and justification for design decisions made. First, the hypothesis will be presented, making clear what the goals of the research were. Then, implementation details regarding the parallel system and the similarity measure will be given. Finally, the design decisions made in the implementation of the system will be discussed.

3.2 Hypothesis

The ultimate purpose of this work was to investigate the usefulness of distributed memory parallelisation as a technique to reduce the time taken to perform EST clustering. This purpose was evaluated according the results gained by implementing a parallel program using d^2 as a metric, and comparing it against a sequential d^2 clustering program. The hypothesis can be stated as follows.

> A parallel EST Clustering algorithm can be constructed to share the computational load efficiently among multiple processors with separate memory.

The efficiency required was a decrease in the amount of time in relation to the number of processors (p) which would hold up to 100 processors, such that the efficiency should have been 0.6.

The primary reason for this was the result shown in Carpenter *et al.* [2002] which showed an efficiency of 0.8 on 128 machines for a parallel d^2 clustering algorithm. The major differences between the approach taken in Carpenter *et al.* [2002] and this research, which led to the expectation of a lower efficiency, were:

- The Carpenter *et al.* [2002] parallelisation was designed for processors having a shared memory, and did not use message passing. It did not incur the overhead of sequence distribution, as all processors had access to the input set in the shared memory

space, meaning that the sequences did not need to be transferred into each processor's space. The latency of communication was also much lower as network latencies are higher than memory latencies.

- The algorithm that was parallelised by Carpenter *et al.* [2002] was a different d^2 clustering algorithm, d2_cluster instead of wcd. The characteristics of the algorithm would affect the efficiency of the parallelisation. In particular, the use of a heuristic in wcd to avoid unnecessary calculations could cause a decrease in the efficiency of the algorithm.

- The Carpenter *et al.* [2002] parallelisation did not use a master-slave architecture, instead using a system of peers for the parallel portion.

An efficiency of 0.6 was chosen as a conservative estimate based on the extra costs imposed by the modifications necessary to the strategy adopted by Carpenter *et al.* [2002].

An examination of whether this is likely to be achievable in terms of Amdahl's Law indicates that an efficiency of 0.6 should not present a problem. Amdahl's Law, presented in section 2.5.6 was stated as

$$S \leq \frac{1}{f + (1-f)/p}$$

. In order to satisfy an efficiency of 0.6 on 100 processors, the equation read as follows

$$60 \leq \frac{1}{f + (1-f)/100}$$

. Solving for f, which represents the sequential part of the algorithm, we get

$$f \leq \frac{1}{148.5}$$

.

This number means that in order for the speedup to be achievable, the sequential portion of the program must be no more than $1/148.5$ of the program according to Amdahl's law, presented in section 2.5.6. This was reasonable because there are at most $n-1$ merges whereas there are at most n^2 comparisons. The smallest value of n to be tested in this research was 10000. For this value, the condition holds, as $10000^2/148.5 > 10000$. The load-balancing procedure is $O(n^2)$ but does not take much processor time. The procedure by which load-balancing is accomplished is detailed in 3.4.3. For a large n, this makes the fraction achievable.

An efficiency of 0.6 would have meant that when run on 100 machines, the program should have taken $\frac{1}{60}$th of the time taken when run on a single machine. As the number of machine increases, the costs of communication and coordination increase, leading to a smaller gain until eventually no gains are made, so no claims about the efficiency are made for $p > 100$.

3.3 Overview of the Parallel System

The approach that was taken by this research was to implement a parallel algorithm to perform d^2 clustering. The resulting clusters would need to be the same as those calculated by the wcd implementation.

The strategy that was employed in this experiment was to use a Master or Coordinating machine, to which all other machines are subordinate. This master machine holds the EST database and the find-union graph of the clustering, which is initially a collection of EST fragments, and eventually consolidates into a group of clusters. This master machine is responsible for performing the merge operations that are necessary when a comparison satisfies the merge criterion. As an example of a merge criterion, using the d^2 metric, the merge is performed if the score is below a user-defined threshold.

The subordinate machines are assigned calculations by the master machine and then calculate the required comparisons, and send back a merge request, if one is necessary, or a done signal to indicate that the particular calculation has completed. The master machine then performs a merge on the clusters if necessary, and keeps a record that the comparison has been done.

The basic algorithm for the master machine is presented below.

Algorithm 2 Algorithm for the master machine

```
Let n be the number of sequences
Send all sequences to all clients
while there exist some calculations to be performed do
  if there is a client whose queue is not full then
    send a batch of calculations to that client
  end if
  if a client indicates two sequences are similar then
    merge the clusters containing the relevant sequences
  end if
end while
```

The algorithm for the slave machines is presented below.

Algorithm 3 Algorithm for the slave machine

```
Let n be the number of sequences
Receive all sequences from server
while the server has not exited do
  Receive batches of work and put in queue
  for each comparison in the batch do
    if comparison shows similarity then
      send message indicating similarity to server
    end if
  end for
  send message to server indicating completion
end while
```

3.3.1 Union-Find Data Structure

In order to efficiently accomplish the clustering, we need to be able to merge clusters efficiently. We must also be able to easily identify the cluster containing a particular sequence. A suitable data structure for this task is the Union-Find data structure.

Hopcroft and Ullman [1973] present an efficient method of performing set union operations that can be used to perform cluster merges, as these are essentially set merge operations.

The algorithm works by using a tree data structure to represent each set. Two operations are allowed on the structure, `union` and `find`. The `union`(a,b) operation connects the trees containing vertices a and b, and `find`(a) returns the root of the tree containing vertex a.

Each vertex in the structure maintains a pointer to its parent. An internal `link`(a,b) operation is defined which sets the parent pointer of b to a. The `union`(a,b) operation is then performed as `link`(`find`(a),`find`(b)). This means that the `find` operation is the pivot to ensuring efficient operation. The `find` operation is performed by following the parent pointers until the root of the tree is reached. It is thus clear that the time taken to perform the `find` operation is directly related to the depth of the node in the tree.

We thus employ heuristics to keep the trees short. Two important heuristics are union by rank and path compression. The union by rank heuristic is useful because it makes sure that the shorter tree is appended to the larger one, rather than the other way around. This keeps the trees shorter as the tree height only increases when two trees of equal height are joined. Path compression is a method added to the find operation. When we perform a find on an element, we set its parent pointer to point directly to the root. Since we have to find the root anyway, the operation does not add extra complexity. Note that in order to perform a find operation on a vertex a, we need to find the root of its parent. The result of this is that all vertices along the chain from a to the root have find operations performed on them, and hence have their parent pointers set directly to the root.

The worst case complexity for a series of n union and find operations is $O(n(\log^*(n)))$ [Quinn 1987, page 168]. The time spent per individual operation thus amortizes to $O(\log^*(n))$ which is almost constant. The function $\log^* n$ can be defined as follows:

$\log^*(k)$=1 for $k < 1$
$\log^*(k)$=1+$\log^*(\log(k))$ for $k \geq 1$

For example, $\log^* 4 = 2$, $\log^* 16 = 3$, $\log^* 65536 = 4$.

3.3.2 Effect on the parallelisation

For the purposes of EST Clustering, the inherently sequential fraction of the program is made up of the merge operation, which is $O(n \log^* n)$ in the worst case, with n representing the number of sequences to be clustered, as was shown in section 3.3.1. The parallel fraction

is made up of the sequence comparisons, which are in general $O(n^2)$ multiplied by the complexity of the comparison measure, which, for d^2, is $O(m^2)$ with m representing the sequence length. This indicates that the portion of the program which is sequential will be decreasing for increasing problem sizes, hence making the parallelisation work better for larger instances of the problem.

This demonstrates that the portion of the program that cannot be parallelised is not likely to be a large fraction of the total instructions, and will thus allow a large speedup.

Another limitation caused by parallel code is that if there is parallel code dependent on the sequential code, all processors must wait for an individual processor to complete the sequential code. This does present a problem for EST Clustering, as processors could be waiting for instructions if the processing of the sequential operations become a bottleneck. This processing can be one of two operations, a merge operation or an ignore operation. Individual merge instructions have a close to constant cost, as was shown in section 3.3.1, and an individual ignore instruction has a constant cost as it just decrements the number of requests known to be in a particular slave's queue. With p processors, this means that conceptually, p operations need to be performed during each comparison calculation on a single processor, because this is the amount of sequential work that needs to be done before more sequential work comes in. The operations are thus collectively close to $O(p)$ while the comparison calculation is $O(m^2)$. Since p is unlikely to be larger than the average m (approximately 300), a large difference in the constant factors would be the only reason for the merges to constitute a bottleneck.

3.4 Implementation of the Distributed System

The decision to implement the system as a master-slave system was made for a variety of reasons. Primary among these is that a central coordinator imposes less costs than requiring the machines to cooperate in making decisions about what work would be handled by the various machines. It is thus necessary to have a machine dedicated to coordinating the activities of the various machines, ensuring that the work is divided evenly among the machines.

The slaves are responsible for the similarity calculation, and the processing that is necessary to decide whether two sequences are similar enough to be clustered together is the sole responsibility of the slave machines. The master machine decides which computations to send to which slaves. These slaves then tell the master which sequences should be clustered together, and the master performs the clustering on the basis of this. A diagrammatic view of the system is presented in figure 3.1.

In order to help hide the delays which network latency would impose, requests are queued on the slave and the master, allowing them to continuously process the requests at the heads of their queues rather than pause to wait for the next job upon completion of the previous one. This queuing system complicates the load balancing, as it is now necessary to keep the queue sizes balanced, rather than the simpler task of ensuring that a machine

is available before sending a sequence to it. The saving is significant because the network latency time cost is incurred once if queuing is used, which is when the first item is being added to the queue. If no queuing is used, the network latency is incurred every time a request is sent to the slave, which is $O(n)$ times.

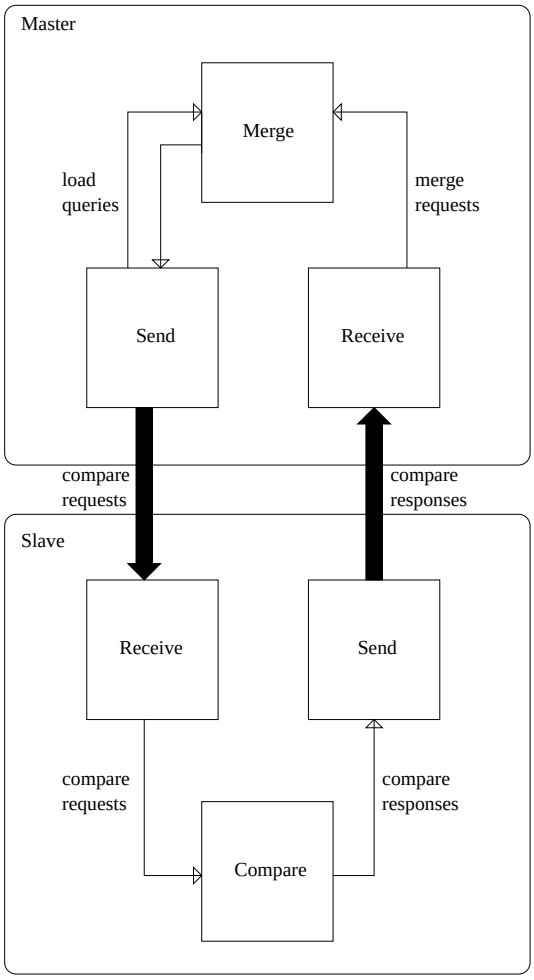

Figure 3.1: Diagrammatic view of the system

3.4.1 The master

The master algorithm was implemented as three coordinated threads. These are the ‘send thread’, the ‘receive thread’ and the ‘merge thread’.

The send thread

The send thread’s first responsibility is to distribute the sequences to the slave machines. The thread reads sequences from a file and then distributes these sequences to every slave.

After distributing the sequences, the master must decide which machines to distribute requests to, and must then distribute that work. To minimize network usage, the master

sends batches of work rather than individual sequence pairs for comparison. The master sends a sequence number to an available slave, which must then process all pairs of sequences involving this sequence. However, in order to avoid pairs being compared twice, the slave will compare the sequence with all sequences having an index higher than its own.

In order to decide which machine to send a query to, the send thread sends a load query to the merge thread. The merge thread picks a slave to send the request to using the algorithm outlined in 3.4.3. It picks this slave on the basis of how many pending calculations that slave has queued. If all slaves are operating at full capacity, the master waits for a small amount of time and checks again. This small delay was included because if there was no delay the master would continuously check if there was an available slave, consuming an unnecessarily large amount of computation time. If the queue size is chosen to be sufficiently large, this delay will not be large enough to allow a slave to reach zero pending requests and therefore waste time. The idea is that the master waits while a slave gets through enough work to open space in its queue for more requests.

The receive thread

The receive thread listens for responses from slaves. When one is received, the thread checks which slave the response is from and records this for load-balancing purposes, which is discussed in greater detail in Section 3.4.3. Next, the receive thread checks whether the slave had decided the sequences should be clustered together or not. If they should, the request is forwarded to the merge thread, so that it can merge the find-union structures containing the sequences appropriately.

The merge thread

The merge thread contains the find-union structure. It is responsible for all operations requiring this structure. Another option would have been to make the find-union structure a shared structure available to all threads. The decision to make it part of the merge thread was taken for two major reasons. The first is that the find-union structure is large enough to make it problematic to allocate shared memory to it. It is therefore easier not to share the structure, meaning that only one thread can have access to it.

The other major reason is that the operations on the find-union structure are processor driven tasks, whereas the master's other tasks are more I/O driven. Keeping these in a separate process thus allows us to make more effective use of the processor.

The merge thread is also responsible for load-balancing. This task was integrated into the merge operation because its operation is tightly coupled with that of the find-union structure. The load-balancing system is also a processor driven task rather than an I/O driven task, and because the load-balancing structure must be accessed by both the send and receive threads, they submit requests to the merge thread rather than accessing a shared structure directly.

The merge thread serves two main functions. One of these is to determine which slave

has the smallest current load, and the other is to perform the union operation on clusters when the receive thread indicates that it should. The merge thread listens for requests, and when one is received, it checks what type of request it is.

If the received request is a `find` request, the merge thread checks which slave currently has the smallest queue. It returns a message containing the number of this slave to the send thread. It then increments the recorded queue size of that slave, because that slave's queue will increase due to the request the send thread is about to distribute to it.

`Done` requests contain a slave number, and indicate that the slave has completed the batch of work that was sent to it. When the merge thread receives one of these, it decrements the recorded queue size of the slave, indicating that the slave has less pending work.

If the request is a `union` request, a union operation is performed on the clusters containing the sequences in question.

3.4.2 The Slave

The slave system has a similar structure to that of the master, being divided up into three constituent threads, the 'receive thread', the 'compare thread', and the 'send thread'.

The receive thread

The receive thread on the slave can receive two different types of requests from the master. The two request types are handled as follows:

- When a compare request is received, the receive thread forwards this request to the compare thread.
- When a done request is received, the slave is stopped, as this request from the master means that all necessary calculations have been successfully performed.

The receive thread handles all communication from the master.

The compare thread

The compare thread is where most of the work of the entire system is done, as it handles the comparisons between the sequences. Before it can do this, it must first receive all the sequences from the master.

Once all sequences have been received and stored, the slave dynamically loads the comparison method to be used. Dynamically loading the comparison method rather than statically linking it allows the use of multiple similarity measures without modification of the main code. The comparison methods thus act as plugins into the main system.

The compare thread then listens for incoming compare requests from the receive thread. These compare requests contain a sequence number. The compare thread compares the sequence referred to in the request with every sequence that has a higher sequence number, and if they are found to be similar, it sends a `union` request to the send thread. These

comparisons determine whether the sequences should be clustered together. Once all the comparisons for a particular request have been performed, a `done` request is sent to the send thread.

The send thread

The send thread waits for results from the compare thread, and forwards these to the master. As can be inferred from the operation of the compare thread, the messages which could be sent back to the master are `union` requests and `done` requests. The facility to batch these requests to further optimize network usage is included, but the batch size was set to 1 for the experiments, as batching them did not have any effect on the times. This may become useful as the number of slave machines is increased beyond the numbers explored in this research, as the network usage could become important.

3.4.3 The Load-Balancing Algorithm

The load-balancing algorithm is responsible for making sure there is an even division of work among the slave machines, to avoid wasting computation time as some machines are busy and others sitting idle. In order to do this, it tries to balance the number of requests in the request queue in each slave. This request queue should not be emptied until all comparisons have been performed. This is done on the master, so the master keeps a record of the amount of requests in the queue of each slave. In order to keep the load balanced, we assign all calculations to the slave with the least requests in its queue at the time of generating the calculation.

The algorithm thus has three main functions governing the selection of a suitable slave.

- It must be able to find the slave with the smallest recorded queue size.
- The recorded queue size of a slave must be able to be incremented.
- The queue size recorded for a slave must also be decrementable.

The algorithm displays all of these characteristics, with each function taking a constant time to perform.

Data structures

The data structures used for the load-balancing system were chosen to minimize the computational complexity rather than the memory usage. They do not, however, impose a large memory overhead.

The structure takes the form of an array of doubly linked lists, with one linked list being created for every integer from 0 to the maximum queue size of the slaves. In the linked lists, we store a node for every slave that has a number of queued elements that corresponds to the array position. In other words, every slave that has i comparisons waiting in its queue appears in the linked list referenced by the ith position of the array. An important decision

made here was to keep the address of the head and the tail in the array. While this does cost more memory, it allows us to avoid traversing the entire linked list when we want to attach a node to the end.

A record is kept of the array position with the smallest index which has at least one node in its linked list, called `least`. This record allows us to find the slave with the least number of pending requests in constant time.

Furthermore, an array is created which keeps a reference to each node, indexed by a slave number. This allows us to access the node for a particular slave in constant time.

Algorithm

Initially, we create a node for every slave, forming a linked list with slave number 0 at the head. Then, to select a slave, we always pick the head of the linked list pointed to by `least`, removing it from the linked list and attaching it to the tail of the linked list representing the new number of sequences in its queue. This is done in constant time, as the tail of the queue is easily accessible from the array. Now, if the linked list which the node was removed from is now empty, we increment `least`. This can be done because the slave that we selected now has one more queued request than it had initially, so we know that at least one node has just one more queued request than the slave had initially.

When we receive a `done` response from a slave, we must decrement the number of requests we record it as having. In order to accomplish this more efficiently, we use an array which has one entry for each slave. This array stores a pointer to the node created for each slave, and allows us to access that node in constant time. Once we have found the node, we remove it from its linked list, which can be done in constant time because the list is doubly linked. We then attach its node to the tail of the linked list representing slaves with one less queued request. Then, if this number of requests is less than `least`, we decrement `least`.

This system can thus perform all of its operations in constant time, and was therefore chosen for the load-balancing algorithm. This is important because load-balancing must perform its operations every time a compare request is sent out to a slave, and every time a `done` response is received. The number of times this is done is $O(n)$, so the problem of a time consuming load-balancing system would be compounded dramatically.

3.4.4 The wcd plugin

The d^2 calculation has been implemented as a plugin to the clustering system, using the wcd algorithm presented in section 2.4.2. The calculation is performed in a manner described in Hide *et al.* [1994], but does not take into account sequence weights. This is similar to the approach taken in current implementations and will thus allow for more useful comparison in relation to already existing clustering methods.

The calculations are performed on subordinate machines, with these machines immediately deciding whether to merge or not, on the basis of the calculated d^2 score. The

implemented plugin operates on compressed sequences, a decision which has been justified in section 3.4.1.

It also makes use of a heuristic to speed up the calculation. This is a decision which was made in the sequential implementation due to the complexity of the d^2 calculation. The heuristic filters out pairs which could not have a d^2 score below the clustering threshold. The sequences which make it past the heuristic must still be evaluated by the main d^2 algorithm. This heuristic dramatically reduces the execution time of the sequential algorithm and was thus included in the parallel implementation.

3.5 Design Decisions

In this section, alternative approaches to the decisions made for the system will be presented, along with reasons they were not used.

3.5.1 Architecture

For the system architecture, it was possible to choose a master/slave system, or one of cooperating peers. One issue here is the division of work.

The division of work among the machines is an issue which must be addressed in a parallel system. In a system of cooperating peers, the lack of a coordinating mechanism means that there is no central authority to dictate that division.

One technique used to deal with this is to have the machines negotiate among themselves as to which calculations they are responsible for. This is problematic as it requires a large amount of communication between processors.

Another approach is to divide the work beforehand, performing some sort of arithmetic division of the work such that a machine is responsible for its partition of the work. An obvious advantage of this is the reduction in the communication required, as machines know beforehand what their responsibilities are. A disadvantage is the possibility of uneven loads as the division of work must be done beforehand by attempting to assign each machine a set of comparisons that would take an equal amount of time to perform. For this problem, calculating this division of work is likely to be extremely difficult. This is because the algorithm makes use of a heuristic which drastically reduces the time taken to perform the calculation in certain cases, and leaves it unaffected in others, leading to a difficulty in calculating the time a set of comparisons is likely to take.

A master/slave architecture would allow a master machine to distribute calculations to the sequences, and since it knows the current state of the clustering, it would be able to avoid redundant calculations. This would however, lead to an increase in the communication cost, as every calculation would then generate a request which would need to be transmitted from the master to a slave. The costs of avoiding these redundant calculations would outweigh the benefit. This has been tested experimentally for the purposes of this system.

A possible disadvantage of the master/slave architecture is that the master could become a bottleneck, as all machines must wait for the master to assign calculations to them before

they can perform any work. In order for this to happen, the master must be too busy dealing with comparison results for new comparison requests to be generated. For this to happen, p merges must be more expensive than the combined time of one comparison calculation, one `find` operation, and one `done` operation, as this is the amount of work the master has to do in comparison with the slave. This is unlikely to happen, as the merge is a small operation in relation to a comparison.

The architecture chosen was the master/slave architecture, as it provides more flexibility regarding the division of work among the machines.

3.5.2 Distribution of work

In the master/slave architecture, there is a need to distribute work to the slaves. This can be done in many ways, including partitioning the work beforehand, which was discussed in 3.5.1. This results in a small network overhead, as comparison results are the only network traffic that results. The other option, of requesting calculations from the slaves as they are deemed necessary, would result in more effective balancing of the amount of work performed by each machine, but this balancing also comes at the cost of increased communication and processing.

While it remains to be tested which is the more efficient approach, a dynamic division of work rather than a predefined partitioning was chosen. Within this approach, there are two possibilities for assigning work. The slaves can either request work when they recognize that they need more, or the master can poll slaves to evaluate whether they have enough work. Polling slaves involves the master requesting some sort of status report from each slave, which would then send this back, so that the master can then decide whether to distribute work to that slave. This results in a lot of unnecessary communication, but would help facilitate load-balancing. A difficulty arises in choosing the frequency of the polling. Another approach is to have the slaves request work. This is the approach that was chosen here. A message from the slave indicating that a particular calculation has finished is treated by the master as a request for more work. This approach was chosen as there is no need to decide how often to poll slaves, with them indicating when they have capacity for more calculations.

3.5.3 Sequence Distribution

In order for the slaves to perform their comparisons, they must know what the sequences are. There are many possibilities for the distribution of sequences to the slaves. One is that all hard drives must contain a copy of the input file. This would make it impossible to run this system on diskless machines. Since the machines used in the experiments conducted for this research are diskless, this technique was not an option. A problem with using a shared storage system such as NFS [Sandberg 1985] for this is that the number of reads from that individual file would dramatically increase the network usage. Using NFS to distribute n bytes to p processors would use $O(pn)$ bytes of network traffic.

Another option is to distribute the sequences to slaves via TCP. This would also be $O(pn)$ and would be more difficult to implement than using NFS, but would have increased compatibility as the slaves would not need to have a shared storage medium for the system to operate.

A more adaptive option would be to distribute the sequences with the comparison requests. This would, however, cause a huge increase in the amount of network traffic, but would dramatically reduce the memory usage on the slaves as they would only have to keep the sequences that they were working on, rather than the entire data set. It would use $O(n^2)$ bytes of network traffic, which would not be acceptable. This could be reduced by keeping a record of which sequences have been distributed to which slaves and only including the sequence with the comparison request if it is necessary. This would cost $O(pn)$, and would negate the memory usage saving as the slaves would have to store the sequences.

The option which was used in the system was UDP broadcasting, which is a technique allowing packets to be addressed to all machines on a particular subnet. UDP is, however, an unreliable protocol, in that it does not guarantee that the packets will be delivered correctly. This meant that reliability had to be built on top of it to ensure that all the sequences were correctly delivered. A further disadvantage is that broadcast packets are not forwarded by routers or switches, so all slave machines must be on the same subnet as the master machine. The decision to use broadcasting was made because communication within a subnet is reasonably reliable without the intervention of the protocol, and the advantage of being able to address individual packets to all slave machines means network traffic is greatly reduced. The cost of sending the sequences to each machine individually would be $O(cn)$ where c is the number of slaves and n the number of characters in all the sequences. Using broadcasting, this is reduced to $O(n)$.

3.5.4 Sequence Storage

The sequences are stored in a compressed form. This compression does incur a speed penalty as it is more complex to extract data from these than from the uncompressed sequences. However, it is beneficial to the system as a whole as the compression reduces the sequence size to around 25% of its original size. This has a positive effect on the number of sequences which can be processed, as the implementation requires all input sequences to be stored in the memory of every slave.

A further benefit is that the distribution of compressed sequences to all slave machines involves much less network traffic than the uncompressed sequences would. The combined effect of broadcasting the sequences and using compressed sequences reduces the network usage required by sequence distribution to $\frac{1}{4c}$ of the usage required otherwise.

3.5.5 Load-balancing

The load-balancing approach used in the system functions by having the slaves return a `done` request when they have completed a request. This allows the load-balancing system

to calculate how many comparison requests are in the queue of each slave, allowing the system to effectively balance the queue sizes, which leads to a balancing of the loads.

An alternative to this would be to have each slave send a work request when the number of requests in its queue falls below a particular threshold. Because the master will also be aware of what this threshold is, it would then distribute enough work to that slave to fill its queue. This would result in a smaller amount of network traffic. This alternative was not used because it would have complicated the implementation, and because the queue size chosen was too small for this approach to make too much of a difference.

3.5.6 Interprocess communication

Both the master and the slaves are multi-threaded, improving the efficiency of the system due to the ability to receive input and send output while processor intensive tasks are simultaneously executing. This, however, required the implementation of an interprocess communication system.

A common method of communication is to use shared memory. The difficulty here is that this memory must be synchronized, to avoid race conditions and overwriting.

In the system developed for this research, interprocess communication was accomplished through the use of interprocess sockets. Interprocess sockets were used to pass messages between the various threads. These were chosen for ease of implementation, as they present an interface that is consistent with network sockets. They are also reliable and efficient.

3.5.7 Messages

The messaging system made use of batching for the comparison requests, by consolidating groups of comparisons into a single request. This was accomplished using the system outlined earlier, whereby a request containing a sequence number indicates that the sequence should be compared to all sequences with a higher number. Because the sequences are distributed in increasing order of sequence number, this results in a constantly decreasing batch size.

An alternative approach would have been to algorithmically divide the comparisons into equal batch sizes. This would, however, have been more difficult to implement. Another factor in favour of decreasing batch sizes is that it helps fine tune the load-balancing toward the end of the run. This is because unless the master machine becomes a bottleneck, the slaves are operating at full capacity, due to the master machine making sure their queues are full. This holds until the master machine has no more batches of work to distribute. At this point, the queues of the slaves empty at different rates. Large batch sizes at this point could result in some slaves having no more work to do while others had a large number of comparisons in their queues. Decreasing batch sizes ensure that by the time the master machine runs out of batches to distribute the batch size is small enough that no machine has a large number of comparisons in its queue.

3.6 Conclusion

The distributed system constructed for this research is a multithreaded, master/slave system. Design decisions presented in this chapter affect the efficiency of the system. The results obtained from the experiments presented in the next chapter must thus be understood in relation to the implementation decisions discussed in this chapter.

Chapter 4

Experiments

In this chapter, the experiments used in this research are presented, with some justification for these as well as an indication of the limitations that results obtained from these experiments would suffer from. The system outlined in chapter 3 was run using inputs presented later in this chapter, and then tested for performance against an existing sequential implementation.

4.1 Hardware

The experiments were run on a network of Linux machines. The machines have 128MB of RAM. The processors are not homogeneous, with a majority of Intel Celeron processors running at 1.7GHz and some running at 2GHz. These machines were chosen due to their availability. There is a certain amount of inaccuracy introduced because the hardware is disparate. This was handled by erring on the side of caution, measuring the time taken by the sequential runs on the fastest available hardware, while testing the parallel runs on a mix of the machines. This means that the speedup is underestimated.

The machines use the Networked File System [Sandberg 1985], meaning that they use a shared file system physically located on a central server. Because the system makes minimal use of the file system, this does not make a large difference to the running time of the program. Furthermore, the processing costs are so much larger than the I/O costs that gains in this area will make a small difference to the results. This is partly because the use of broadcasting has meant that the I/O costs do not scale up with the number of machines, instead staying constant for a particular number of sequences.

The experiments were run after hours, when the machines are not in use. This meant that the experiments would not be interfered with by other jobs.

4.2 Data Set

The data set used was a disparate set of human gene sequences derived from various human genes collected from a publicly available EST database known as Genbank. The data set

does have some effect on the results, as the input sets affect some of the behaviour of the algorithm.

One reason the results would be affected is that the client machines employ a heuristic which determines whether it is necessary to perform the d^2 calculation. If it is found that two sequences cannot have a sufficiently low d^2 score, the calculation is not performed. In a pathological case, where almost all cases would be discarded by this heuristic, the amount of work that must be performed by the client machines will be so small that it could be overwhelmed by the costs of communication, coordination and sequence distribution. The speedups gained will thus be less impressive, and it is quite possible that the sequential algorithm would be faster than the parallel system. This would, however, be an extremely unlikely set of cases, and would in any case be processed by the fast heuristic rather than the slow d^2 calculation and would thus not be in much need of efficient parallelisation.

Another reason the input set is of some importance is that the sequential algorithm does not compare sequences to each other if they are already in the same cluster, saving some time. In most cases this is a very small number of pairs, and this time reduction is not significant. The parallel algorithm does not use this technique as the cost of letting client machines know which pairs not to process would outweigh the time saving in most cases. However, in a pathological case where there were many more similar sequences and hence a smaller number of large clusters rather than the large number of small clusters we find in general, the sequential algorithm would have much less work to do, and would hence perform better in relation to the parallel one than it currently does. This is because the cost of communication would become larger than the cost of computation.

4.3 Experiment Setups

The sequential version of the wcd algorithm was run multiple times, varying the number of sequences in the input set and testing the time taken. This was done to allow comparison with the parallel algorithm. The sequential algorithm was run on a 2GHz Celeron machine with 128MB of RAM.

The distributed version was run using different numbers of sequences as well as different numbers of processors. The distributed system was run on a set of machines ranging from 1.7GHz Celeron machines to 2GHz Celeron machines, all with 128 MB of RAM. The numbers of sequences used correspond to those used with the sequential algorithm, and include time measurements for higher numbers of sequences that could not be tested with the sequential implementation due to the large amount of time the sequential implementation would take to complete. The number of sequences used are shown in table 4.1. In the case of experiments run on the parallel system, a server processor is required, increasing the total number of machines used by one. The total number of machines used for 10000 sequences for instance would then become 1, 11, 31, 51, 81, and 101. Note that the sequential experiment is not affected as no server processor is present.

Higher numbers of sequences were not tested due to the amount of time it would take

Number of Sequences	No. Of Client Machines Used
10000	1, 10, 30, 50, 80, 100
50000	1, 10, 30, 50, 80, 100
100000	1, 10, 30, 50, 80, 100
200000	1, 10, 30, 50, 80, 100
300000	1, 10, 30, 50, 80, 100
500000	50, 80, 100

Table 4.1: Input Sizes and No. of Clients used in the experiments

for runs using higher numbers of sequences to complete. The numbers of sequences used do provide a good picture of the performance of the algorithms, as they represent a large variety of data sizes which could be used in practice.

4.4 Limitations

One limitation of these experiments is that all the machines are located on the same subnet, allowing the use of UDP broadcasting to limit the bandwidth usage. This means that the speed results cannot be extrapolated to machines that are not on the same subnet. This has particular significance in grid computing where the machines making up the parallel system are not on the same subnet. It also places a limit of 254 machines in the distributed system as this is the maximum number of machines that can be placed on a single subnet.

The test hardware was chosen on the basis of its availability and has some characteristics that make its results noisy. Part of the reason for this is that network response is often unpredictable, so that two identical runs yield slightly different results. Another problem is that the machines used are not homogeneous, so that the mix of processors used affects the final result. An important factor to note here is that these characteristics do not affect the sequential runs, and serve to slow down the parallel runs, with network performance either at optimal or less than optimal, and the sequential runs performed on the fastest available hardware. This meant that all noise resulted in an underestimation of the speedup possible from this approach.

Although the parallel system uses a plugin system that can accommodate many different measurement methods, the only plugin used implements the wcd algorithm for d2 clustering. This means that no claims can be made about the performance of the system when used with other methods. This was considered to be outside the scope of this research, and can still be tested in the future.

4.5 Conclusion

The experiments chosen give enough information to decide on the accuracy of the hypothesis, giving us an indication of the suitability of a distributed system for EST Clustering. The

results from these experiments, shown in the next chapter, must be recognized to be slightly inaccurate due to the test setup, but can still be used to draw conclusions about the system.

Chapter 5

Results

5.1 Introduction

The results presented in this chapter are those captured from running the sequential and parallel experiments outlined in the previous chapter. The results are presented in tabular form as well as a series of graphs showing the speedups gained from using multiple machines as opposed to a single machine. The results are then analysed.

5.2 Execution Times

Table 5.2 shows the time in minutes EST clustering took for data sizes of the indicated sizes on the stated number of machines. A point to note here is that the results for 500000 sequences are incomplete. This is because the execution would take too long on small numbers of machines, based on an estimation resulting from the complexity of the algorithm.

The graphs in the next section present this information in graphical form, showing the relationship between the execution times and number of machines used. When the number of machines is greater than 1, the total machines used includes a server, with the rest of the machines being client machines. So, when the number of machines is 101, this refers to 100 client machines.

	Number of sequences used					
Total Machines	10000	50000	100000	200000	300000	500000
1	9	652	2305	12127	35787	n/a
11	2	60	185	1061	3247	n/a
31	1	21	73	358	1080	n/a
51	1	13	48	213	642	1190
81	1	8	31	127	394	730
101	1	8	27	111	343	641

Table 5.1: Time in minutes for the various runs, rounded to the nearest minute

	Number of sequences used				
Total Machines	10000	50000	100000	200000	300000
11	6.0	10.8	12.4	11.4	11.0
31	18.0	31.8	31.6	33.8	33.1
51	18.0	52.2	48.0	57.0	55.7
81	18.0	81.5	75.6	95.5	90.8
101	18,0	81.5	87.0	109.3	104.3

Table 5.2: Speedup

5.3 Discussion of Results

The graphs show the speedup relations for the different input sizes and numbers of machines. Figure 5.3 shows the speedup, which is calculated as the time taken by the sequential algorithm divided by the time taken by the parallel algorithm. It is presented for each of the input sizes for which a sequential run was performed, since a sequential run is necessary for us to calculate the speedup. It thus presents a graphical version of table 5.2, making

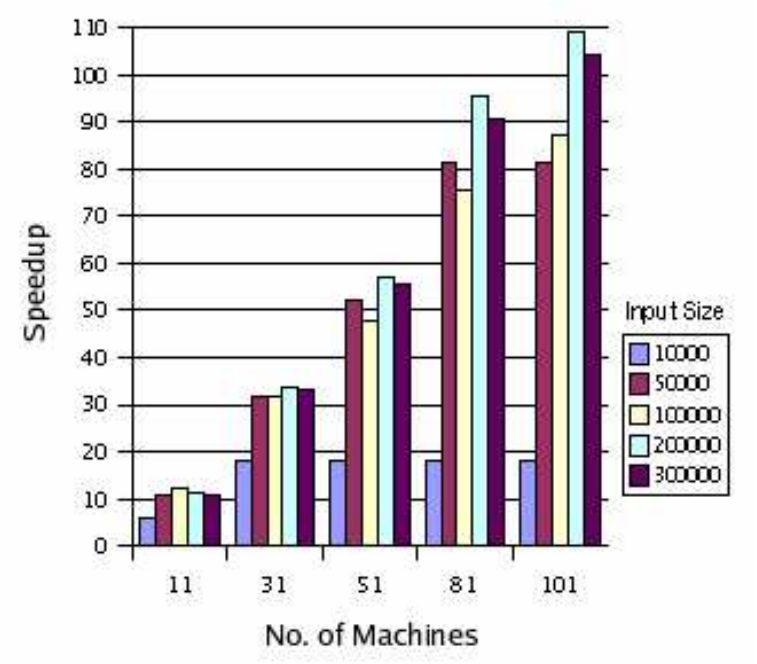

Figure 5.1: Speedup Graph

the information contained easier to observe. From figure 5.3 we can see that the speedup increases with the number of machines, indicating that the distributed system is functioning effectively.

This is not true of the experiments done for 10000 sequences. No incremental speedup is gained for more than 31 machines, as the time taken for execution to complete is excep-

tionally small. Once the costs of the comparisons are shared among the slaves, each slave spends very little time on comparison calculation, with almost all of the total time spent on inherently sequential operations like the distribution of the sequences to the slaves, the setting up of the sockets and the file handling. This means that however small we make the calculation time for the comparisons by using more machines, the time taken cannot be lower than a particular threshold.

Theoretically, this makes sense as we have seen Amdahl's Law in section 2.5.6 which limits the execution time of the program to that of the operations that are inherently sequential. The cost which has remained regardless of the parallelisation is primarily the cost of the inherently sequential operations, and a small cost for the sequence comparison.

For 50000 sequences, the same phenomenon occurs at 101 machines, where no gains are made above those for 81 machines. Here, the time taken is much higher than at the point where increasing the number of machines ceased to help the 10000 sequence execution. One reason for this is that the inherently sequential portion of the algorithm has grown, with the find-union data structure now performing more work. With the larger number of processors, setting up the connections and performing the load-balancing operations is also slower.

For 100000 or more sequences, the speedup increases all the way to 100 processors. This means that the scalability of the system is strong. All gains from parallelisation must eventually taper off as the costs of coordination and communication begin to outweigh the benefits of sharing the workload. This does begin to happen, as the gains are not as impressive between 81 and 101 processors. Since there is still a significant speedup, we can see that the scalability of the algorithm has not been fully tested.

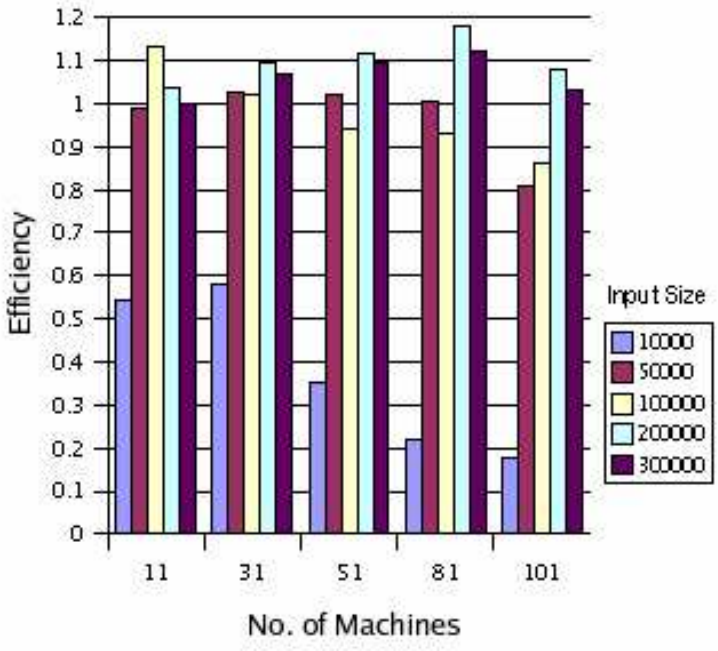

Figure 5.2: Efficiency Graph

Figure 5.3 shows the speedup divided by the number of machines. This gives an indication of the efficiency of the parallelisation, with lower numbers indicating that an increase in the number of processors does not result in as big a decrease in execution time as the experiments with higher numbers of sequences.

For the experiments with 10000 sequences, the efficiency of the system reaches a peak at 31 machines, after which the system becomes less efficient at sharing the work among the machines. This has been discussed earlier, and is due to the sequential component of the program being relatively large, so that the decrease in the execution time of the parallel portion is insignificant by comparison. Another factor is the increase in the coordination and communication overhead resulting from the use of multiple machines.

The curve for 100000 sequences shows a decreasing efficiency, so that every machine added yields a smaller gain. This is because the overheads from communication are increasing. Another factor is the master machine, which sends out batches of work. The smaller these batches of work are, the more work the master has to do in comparison to the slaves. This work also increases for a higher number of machines. At some point, the master becomes a bottleneck when it takes longer to send out a batch to every slave than those slaves take to process them. This is open to some optimization by varying the batch sizes.

For data sets larger than 100000 sequences, the efficiency increases for a while, before decreasing again. Part of this increase is due to the use of UDP broadcasting. Using this technique keeps the cost of distributing the sequences close to constant, meaning that with an increased number of processors, its proportion of the total execution time is decreasing. Also, the master does not become a bottleneck until much later because the batch sizes are larger for these large numbers of sequences.

5.4 Super-linear Speedup

Some of the experiments exhibit what is known as “super-linear speedup”. This means that for p processors, their execution times have decreased to less than $1/p$ of the original time. This is theoretically impossible if the machine on which the sequential algorithm is executed has enough memory, and the sequential algorithm is perfect.

One reason why this does not hold in practice is the use of tiered memory [Gustafson 1990]. Because of the parallelisation, there is very little data that needs to be used on the master. The calculation of the comparisons requires quite a bit of memory, which in the sequential implementation would push the find-union structure out of memory, thus incurring a penalty when it is used again. In the parallel implementation, the master machine does not have calculations to perform, thus making it less likely that calling the find-union structure will cause misses in the cache.

Another reason for super-linear speedup could be inaccuracy in the results, as the test system is not perfect. A characteristic of the machines used to run the experiments is that they are multitasking systems with networking. This means that network requests could have caused more context switches and processor sharing while the sequential algorithm

was executing than while the distributed one was. The distributed system would also be more tolerant of this, as a larger number of machines means a disruption to one causes a smaller overall disruption than if the one machine running the sequential implementation was interrupted.

This was minimized by performing the experiments during off-peak times, when the hardware is normally not in use. It is thus unlikely that the sequential implementation was interrupted.

5.5 Analysis

The results show that parallelisation, and distributed computing in particular, is useful for solving the EST Clustering problem. The results show that a message passing system communicating over the high latency medium of a network can outperform a shared memory system such as that used by Carpenter *et al.* [2002].

One possible reason for this is that message passing systems are more scalable than shared memory systems, because the cost of ensuring memory consistency among the multiple processors eventually becomes too high [Ahamad *et al.* 1994].

While the latency of network communication is higher than that of memory, this was effectively hidden for this problem because it does not have many data dependancies. This lack of data dependancies means that there are few points where the completion of a task requires information from other tasks. The only major communication that must be done is thus communicating the results of calculations to the master machine. Since the next calculation is not dependent on this task, the processor does not need to wait to make sure the communication was succesful before continuing. This hides the latency of the network communication.

The Carpenter *et al.* [2002] implementation also did not perform any load-balancing, with a static division of work performed at the beginning of the processing. This meant that processors would occasionally be idle if they were assigned small computations when other processors were assigned larger ones.

The result achieved were also better than those obtained by Kalyanaraman *et al.* [2002]. A possible reason for this is that the Kalyanaraman *et al.* [2002] system made use of a heuristic to generate promising pairs based on overlaps. This function was also parallelised, with a possible pitfall being that the heuristic was not computationally intensive enough to allow successful parallelisation.

The approach in Kalyanaraman *et al.* [2002] may also have incurred the overheads of ensuring cache consistency as it was implemented on shared memory hardware.

The results achieved by this distributed system are very good in relation to previous research. Some factors influencing this strong showing have been presented to show that the results make sense in light of previous research.

5.6 Limitations

The results that were observed do not allow us to determine the point at which the algorithm stops yielding speedups for increases in the numbers of processors used. The scalability of the algorithm has thus not been fully tested.

5.7 Conclusion

While there is some noise in the test data, the results indicate that the distributed system performs effectively. The decrease in efficiency for 101 processors indicates that the system is beginning to reach the limits of its scalability. This is a good result as our goal was efficiency on up to 100 processors. Overall, the experiments yielded good results.

Chapter 6

Conclusion

6.1 Introduction

This section summarizes the work that was done and states what conclusions were reached in light of the results obtained. It then gives some suggestions for future work which needs to be done in this area, showing the limitations of this research. Finally, the contribution of the research is outlined.

6.2 Summary of research

The research aimed to investigate the usefulness of distributed memory parallelisation as a technique to reduce the time taken to perform EST Clustering. In order to test this, a distributed system was built to perform a particular type of EST Clustering, called d^2 clustering. This system was then tested against an existing sequential implementation to investigate whether any gains were made. The system was also tested with differently sized input sets because the difference in performance between the sequential and parallel implementations would change for different input sizes.

The input sizes tested were 10000, 50000, 100000, 200000, 300000 and 500000. The numbers of slave machines used were 1, 10, 30, 50, 80 and 100. Including the master machine used in the parallel experiments, the total numbers of machines used were 1, 11, 31, 51, 81 and 101. The success of the distributed system was then evaluated using the timing data from these experiments.

6.3 Summary of results

The results showed that a distributed system can efficiently share the computational load of EST Clustering, and can therefore provide a large speed increase. The research showed linear and occasionally super-linear speedup. Because the test setup was slightly noisy, the results have a slight error introduced.

The results showed that the distributed system performs more effectively for large input

sets. This is useful because the distributed system is most likely to be used on large input sets, as small ones can be handled quickly enough on the sequential implementation.

The results thus allowed us to accept the hypothesis, concluding that it was possible to build a parallel algorithm to perform EST Clustering that would share the load efficiently among multiple machines with separate memory.

6.4 Future Work

This work offers some opportunity for expansion, with various avenues which could be explored to make this technique more useful and to test its effectiveness further.

6.4.1 Measurement Methods

The distributed system has been tested using d^2 clustering only, and as such, the results can only be applied to this form of EST Clustering. Since the distributed system uses a plugin system facilitating the use of different measurement methods, it would be interesting to check whether similarly good results can be obtained using other measurement methods. Other quadratic methods are particularly promising as they should yield similar results to those obtained for d^2.

Furthermore, the system can be used to more quickly and more exhaustively examine the effectiveness of different sequence comparison methods in terms of their accuracy for partitioning sets of sequences according to gene membership.

6.4.2 Optimizing

Some parameters of the system were not fully tested for performance. It is thus possible that a different configuration will result in better performance, yielding larger speedups than that of the current configuration.

Queuing batches on the slaves helps to hide latency and bandwidth problems. This is because the slaves request work when their queues are not full rather than when they have no more work to perform. This means that the delay between sending a request for work and receipt of a batch of work is hidden because the slave continues working on the other batches of work in its queue while waiting for the new batch. It is important to have a queue that is large enough to accomplish this, so the slave is never idle. However, the drawback of a large queue is that load-balancing becomes less efficient. This is because it is possible that when all slaves exit, one slave which had been assigned what turned out to be time-consuming batches may have a large amount of work in its queue while the other slaves are idle. Because of this trade-off, this is a parameter which can affect the results. A queue size of three batches was chosen for the experiments conducted in this research as the batches are large enough to ensure that the processing of a batch takes longer than the receipt of another.

Another parameter which could be altered is the batching of slave requests for work. This would reduce the network bandwidth required as multiple requests could be contained in one message. Batching the slave requests involves delaying the sending of a request until enough requests are received to make it worthwhile to send a batched request. This is useful for small batch sizes coupled with large queue sizes. When we have small batch sizes and large queue sizes, a large number of slave requests are issued from each slave, making network usage a problem. Batching slave requests reduces this problem as multiple requests are contained in a single message. This technique was not used in the experiments conducted here as the batch sizes are large and the queues small.

6.4.3 Reliability

Reliability was not factored into the design of the distributed system. When the system failed, the experiment involved was repeated. The only observed cause of failure was the disappearance of a machine from the setup, which was caused by another user rebooting the machine. There would be some costs involved in making the system reliable, which would reduce the efficiency of the parallelisation, with these costs not being present in the sequential version of the algorithm.

On failure of a particular slave, it is necessary for the master to recognise that a failure has occurred and reassign the work that the failed slave should have done. The difficulty here is that for the failure to be recognised within a reasonable time, we cannot rely on the lack of results begin returned. This is because the batches of work assigned to slaves can become so large that results might take a long time to be returned from working slaves. To more quickly detect failures, the slaves could be forced to periodically send a message to the master indicating that they are alive. This would increase network traffic and thereby decrease the efficiency of the parallelisation.

The handling of a recognised failure would also require changes, as the current system does not keep a record of what work a slave still has pending. The master only knows how much work a particular slave has, not which particular batches make up that work. In order to reassign this work on a slave failure, this must be stored. This would introduce extra memory usage, but this would not be a large amount as the slaves do not keep many batches in their queue. The master would also need to be able to mark a slave as failed so as to ensure that no new work is assigned to it. This would not introduce any extra costs as it can be done within the current system.

Since the communication between the master and slaves is accomplished using TCP, it is reliable and needs no modification. The distribution of the sequences is accomplished by employing UDP broadcasting, which is unreliable. Reliability has already been added on top of this, so no modification is necessary here.

6.4.4 Data Set Sizes

The experiments used data sets of up to 500000 sequences. The gain in speed allows us to consider processing data sizes that we would not previously have been able to process. The algorithm in question is quadratic, indicating that for a tenfold increase in the problem size we would encounter a hundredfold increase in the execution time. Note that on 100 machines, we achieve a speedup of approximately 100. This means that using the system on 100 machines we can cluster a data set of around 10 times the size we could on the sequential algorithm in the same time.

This is similar to the indications of the results, with 50000 sequences taking 652 minutes on a single processor and 500000 taking 641 minutes on 101 processors. Extrapolating this, the time that 3000000 sequences would take to cluster on 101 processors can be estimated at around 35787 minutes. Since batch sizes are related to the data set size, large data sets induce better performance in the parallel algorithm than small ones do. This is because it is easier to ensure that all machines are busy when there is more work to do, allowing us to use larger initial batch sizes while still ensuring that there are no idle slaves.

The scalability of the algorithm is not a problem in terms of computational complexity, but memory usage will become a problem for large data sets. Although the sequences are compressed, they still require a large amount of space. Since ESTs are on average around 500 characters long and the compression reduces this to 125 bytes per EST, 3000000 sequences would require 375MB of RAM for sequence storage alone. Other memory costs are not large in relation to this and can thus be safely ignored. The problem here is that all sequences must be transferred to all slaves, meaning that all slaves must have at least 375MB of RAM to store the sequences.

The system was designed to accommodate a number of sequences that could fit in the memory of the slave machines and was tested on this basis. Should the sequences not fit in main memory, they will be swapped in and out of main memory by the OS as they are needed. This will impose a large speed penalty, hampering the performance of the system.

It is possible, however, to make different decisions which will not be as harsh on large numbers of sequences. One possibility is to interlace the sequences any slave must store, separating the slave machines into two groups. All machines in one group contain half the sequences in the input set, with the machines in the other group all containing the other half. Then, in order to perform a comparison between a sequence and all others, the sequence is sent to one machine in each group, which then performs a comparison between it and every sequence it stores.

6.4.5 Architectures

The distributed system built to test the hypothesis in this research used a master/slave architecture. This decision was taken because it simplified communication and coordination. It is not, however, the only possible architecture for a distributed system, with different costs and benefits associated with various architectures. These could be explored to determine

whether another architecture yields a good result. A master/slave architecture introduces a few problems because it is centralized. This means that there is a single point of failure, and also the central point, the master, can become a performance bottleneck.

It is possible to avoid this entirely, using a system of cooperating peers. These processors would allocate work amongst themselves in a distributed manner. One way would be for each processor to broadcast what work it decided to do, allowing all other processors to mark that work as having been done. When a processor required more work, it would grab some work that had not yet been done, and broadcast a message saying that it had chosen to do this work. This process would continue until all work was done. This would create the problem of constructing a unified set of resulting clusters from the disparate clusters calculated by the various machines. A possible scheme to circumvent this is to have all slaves broadcast merge requests so that the merge is performed on all clients. This would increase the total amount of work and hence decrease efficiency. Another option is to have the clusters combined once all the slaves have completed their clustering, computing composite clustering from the individual clusters. This merging of the clusters can be explored further in order to find efficient methods of doing this.

6.5 Contribution

The research showed that a distributed system can be used to dramatically reduce the time needed to perform EST Clustering. This is a result that could immediately be adopted into real-world systems. This would enable faster clustering and hence facilitate a better understanding of gene structure and operation. Because the system uses current measurement methods and does not affect output, the interface presented to biologists is not different and thus presents no barrier to adoption.

The research also showed that shared memory computers are not needed to perform EST clustering, with the relatively inexpensive option of a network of sequential machines yielding good performance. An important side effect of this is that it makes it cheaper and therefore more attarctive to search for better comparison methods, and might make it feasible to use more accurate measures that were previously ignored because they would slow execution down.

This research has thus made a significant contribution to the field of EST Clustering.

References

[Aaronson *et al.* 1996] J.S. Aaronson, B. Eckman, R.A. Blevins, J.A. Borkowski, J. Myerson, S. Imran, and K.O. Elliston. Toward the development of a gene index to the human genome: An assessment of the nature of high-throughput EST sequence data. *Genome Research*, 6:829–845, 1996.

[Ahamad *et al.* 1994] Mustaque Ahamad, Phillip W. Hutto, Gil Neiger, James E. Burns, and Prince Kohli. Causal memory: Definitions, implementation and programming. Technical Report GIT-CC-93/55, Georgia Institute of Technology, 1994.

[Allison 1992] L. Allison. Lazy dynamic-programming can be eager. *Information Processing Letters*, 43(4):207–212, 1992.

[Altschul *et al.* 1990] 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.

[Birrel and Nelson 1984] A. Birrel and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[Bonacina 2000] Maria Paola Bonacina. A taxonomy of parallel strategies for deduction. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):223–257, 2000.

[Brown 2000] R.G. Brown. Maximizing beowulf performance. citeseer.nj.nec.com/422752.html, 2000.

[Burke *et al.* 1999] J. Burke, Dan. Davison, and W. Hide. d2_cluster: A validated method for clustering EST and full-length cDNA sequences. *Genome Research*, 9:1135–1142, 1999.

[Carpenter *et al.* 2002] J.E. Carpenter, A. Christoffels, Y. Weinbach, and W.A. Hide. Assessment of the parallelization approach of d2_cluster for high-performance sequence clustering. *Computational Chemistry*, 23:1–3, 2002.

[Chong and Agarwal 1996] Frederic T. Chong and Anant Agarwal. Shared memory versus message passing for iterative solution of sparse, irregular problems. Technical Report MIT/LCS/TR-697, MIT, October 1996.

[Claverie 1997] J. M. Claverie. Computational methods for the identification of differential and coordinated gene expression. *Human Molecular Genetic Science*, 8:1821–1832, 1997.

[Deshpande and Schultz 1992] Ashish Deshpande and Martin H. Schultz. Efficient parallel programming with Linda. In *Supercomputing*, pages 238–244, 1992.

[Dongarra *et al.* 1992] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level message-passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, Knoxville, TN, USA, 1992.

[Eklund 1998] N. Eklund. Robust transaction service for distributed control systems. Master's thesis, Uppsala University, 1998.

[Flynn 1972] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computing*, 21:948–960, 1972.

[Forum 1994] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[Gustafson 1990] J. Gustafson. Fixed time, tiered memory, and superlinear speedup. In *Proceedings of the fifth distributed memory computing conference*, October 1990.

[Hazelhurst 2003a] S. Hazelhurst. An implementation of the d^2 distance function for DNA sequences. Technical Report TR-Wits-CS-2003-6, School of Computer Science, University of the Witwatersrand, September 2003.

[Hazelhurst 2003b] S. Hazelhurst. The *wcd* manual: EST clustering using d^2. Technical Report TR-Wits-CS-2003-5, School of Computer Science, University of the Witwatersrand, July 2003.

[Hide *et al.* 1994] W. Hide, J. Burke, and D. Davison. Biological evaluation of d^2, an algorithm for high performance sequence comparison. *Computational Biology*, 1:199–215, 1994.

[Hopcroft and Ullman 1973] J. E. Hopcroft and J. D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, December 1973.

[Horton 2001] P. Horton. A branch and bound algorithm for local multiple alignment of DNA and protein sequences. *Journal of Computational Biology*, 8(3):249–282, 2001.

[Jain and Dubes 1988] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[Jain *et al.* 1999] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[Kalyanaraman *et al.* 2002] A. Kalyanaraman, S. Aluru, and S. Kothari. Parallel EST clustering. *First IEEE International Workshop on High Performance Computational Biology*, April 2002.

[Karlin *et al.* 1990] S. Karlin, A. Dembo, and T. Kawabata. Statistical composition of high-scoring segments from molecular sequences. *The Annals of Statistics*, 18, 1990.

[Kim *et al.* 1997] Jong Kim, Heejo Lee, and Sunggu Lee. Replicated process allocation for load distribution in fault-tolerant multicomputers. *IEEE Transactions on Computers*, 46(4):499–505, 1997.

[Koza and Andre 1996] John R. Koza and David Andre. Automatic discovery of protein motifs using genetic programming. In Xin Yao, editor, *Evolutionary Computation: Theory and Applications*. World Scientific, Singapore, 1996.

[Levine 1994] D. Levine. *A Parallel Genetic Algorithm for the Set Partitioning Problem*. PhD thesis, Illinois Institute of Technology, May 1994.

[Lew 1995] Kevin A. Lew. A case study of shared memory and message passing: The triangle puzzle. Master's thesis, MIT, January 1995.

[Mao and Tu 1995] C. Mao and S. Tu. Self-adaptive load sharing for distributed image file processing. In *Proceedings of the 7th IASTED/ISMM International Conference on Parallel and Distributed Computing and Systems*, pages 110–112, 1995.

[Martonosi and Gupta 1989] M. Martonosi and A. Gupta. Tradeoffs in message passing and shared memory implementations of a standard cell router. In *Proceedings of the International Conference on Parallel Processing*, volume 3, pages 88–96, 1989.

[Mironov *et al.* 1999] A.A. Mironov, J.W. Ficket, and M.S. Gelfand. Frequent alternative splicing of human genes. *Genome Research*, 9:1288–1293, 1999.

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

[Nikolopoulos and Polychronopoulos 2001] D. S. Nikolopoulos and C. D. Polychronopoulos. Scaling irregular parallel codes with minimal programming effort. In *SC2001: High Performance Networking and Computing*, pages 10–16, Denver, November 2001. ACM Press and IEEE Computer Society Press.

[Oden and Patra 1994] J.T. Oden and A. Patra. A parallel adaptive strategy for hp finite element comparisons. Technical report, Texas Institute for Computational and Applied Mathematics, 1994.

[Quackenbush *et al.* 2000] J. Quackenbush, F. Liang, I. Holt, G. Pertea, and J. Upton. The TIGR gene indices: reconstruction and representation of expressed gene sequences. *Nucleic Acids Research*, 28(1):141–145, 2000.

[Quinn 1987] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill Computer Science Series. McGraw-Hill, 1987.

[Ramanujam and Sadayappan 1989] J. Ramanujam and P. Sadayappan. A methodology for parallelizing programs for multicomputers and complex memory multiprocessors, 1989.

[Richard and Singhal 1993] G.G. Richard and Mukesh Singhal. Using logging and asynchronous checkpointing to implement recoverable distributed shared memory. In *Symposium on Reliable Distributed Systems*, pages 58–67, 1993.

[Sandberg 1985] R. Sandberg. Sun Network Filesystem protocol specification. Technical report, Sun Microsystems Inc., 1985.

[Schuler 1997] G. D. Schuler. Pieces of the puzzle: expressed sequence tags and the catalog of human genes. *Journal of Molecular Medicine*, 75:694–698, 1997.

[Shao 2001] G. Shao. *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources*. PhD thesis, University of California at San Diego, May 2001.

[Sheliga *et al.* 1998] M. Sheliga, Z. Yu, F. Chen, and E. H. M. Sha. Graph transformation for communication minimization using retiming. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 207–219, 1998.

[Subhlok 1993] Jaspal Subhlok. Automatic mapping of task and data parallel programs for efficient execution on multicomputers. Technical Report CS-93-212, Carnegie Mellon University, 1993.

[Sunderam 1990] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[Tanenbaum and van Steen 2002] A. S. Tanenbaum and M. van Steen. *Distributed Systems : Principles and Paradigms*, chapter 2. Prentice Hall, 2002.

[Vinga and Almeida 2003] S. Vinga and J. Almeida. Alignment-free sequence comparison – a review. *Bioinformatics*, 19(4):513–523, 2003.

[Yang 1993] Tao Yang. *Scheduling and Code Generation for Parallel Architectures*. PhD thesis, Rutgers University, May 1993.

[Yap *et al.* 1998] Tieng K. Yap, Ophir Frieder, and Robert L. Martino. Parallel computation in biological sequence analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):283–294, 1998.

[Zhao and Karypis 2002] Y. Zhao and G. Karypis. Evaluation of hierarchical clustering algorithms for document datasets. Technical Report 02-022, University of Minnesota, 2002.