


2006

Large-scale methods in computational genomics

Anantharaman Kalyanaraman
Iowa State University

Follow this and additional works at: <https://lib.dr.iastate.edu/rtd>

 Part of the [Bioinformatics Commons](#), and the [Computer Sciences Commons](#)

Recommended Citation

Kalyanaraman, Anantharaman, "Large-scale methods in computational genomics " (2006). *Retrospective Theses and Dissertations*. 1529.
<https://lib.dr.iastate.edu/rtd/1529>

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

Large-scale methods in computational genomics

by

Anantharaman Kalyanaraman

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:
Srinivas Aluru, Major Professor
Volker Brendel
Suraj C. Kothari
Patrick S. Schnable
Srikanta Tirthapura

Iowa State University

Ames, Iowa

2006

Copyright © Anantharaman Kalyanaraman, 2006. All rights reserved.

UMI Number: 3229090

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 3229090

Copyright 2006 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Graduate College
Iowa State University

This is to certify that the doctoral dissertation of
Anantharaman Kalyanaraman
has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Major Professor

Signature was redacted for privacy.

For the Major Program

DEDICATION

To Kirti

TABLE OF CONTENTS

LIST OF TABLES	vii
LIST OF FIGURES	ix
ACKNOWLEDGEMENTS	xiii
ABSTRACT	xv
CHAPTER 1. INTRODUCTION	1
CHAPTER 2. SEQUENCE ANALYSIS: BIOLOGICAL BACKGROUND AND TERMINOLOGY	7
2.1 Genomic Repeats	8
2.1.1 Transposons	9
2.2 Sequencing Technologies	9
2.2.1 Expressed Sequence Tag Sequencing	10
2.2.2 Whole Genome Sequencing	13
2.2.3 454 Sequencing	14
2.3 Pairwise Sequence Alignment Computation	15
CHAPTER 3. SEQUENCE CLUSTERING: PROBLEMS AND APPLI- CATIONS	18
3.1 Clustering DNA Sequences	19
3.1.1 Clustering of Expressed Sequence Tags	19
3.1.2 Clustering for Genome Assembly	26
3.1.3 Computational Challenges	29
3.2 Literature Review	30

3.2.1	Methods for EST Clustering and Genome Assembly	30
3.2.2	Discussion of Related Work	36
3.2.3	Performance Evaluation of Related Work	38
3.2.4	Need for Scalable High-performance Computing Methods	41
CHAPTER 4. A SCALABLE PARALLEL CLUSTERING FRAMEWORK		
FOR LARGE-SCALE SEQUENCE ANALYSIS		43
4.1	The Sequence Clustering Problem	43
4.2	A Serial Clustering Algorithm	44
4.2.1	Reducing the Number of Pairs Aligned	46
4.2.2	An Optimal Algorithm for On-demand Generation of Promising Pairs	48
4.3	A Space and Time Efficient Parallel Clustering Algorithm	58
4.3.1	Parallel Generalized Suffix Tree Construction	58
4.3.2	Detecting Overlaps and Clustering In Parallel	60
4.3.3	Software Availability	66
4.4	Results and Applications	66
4.4.1	EST Clustering	66
4.4.2	Clustering for Genome Assembly	75
CHAPTER 5. DETECTION OF LTR RETROTRANSPOSONS		83
5.1	Problem Description and Related Work	85
5.2	Notation	89
5.3	Our Approach	89
5.3.1	The Sequential Algorithm	90
5.3.2	Parallelization	96
5.3.3	Software Availability	97
5.4	Results	97
5.4.1	Quality Validation	97
5.4.2	Performance Results	101
5.4.3	A Large-Scale Application	102

5.5 Discussion	103
5.6 Concluding Remarks	104
CHAPTER 6. SCAFFOLDING GENOMIC CONTIGS USING LTR RETRO-	
TRANSPOSONS	105
6.1 Retroscaffolding	108
6.2 Proof of Concept of Retroscaffolding	111
6.3 A Framework for Retro-linking	115
6.3.1 Building a Database of LTR Pairs	115
6.3.2 An Algorithm to Establish Retro-links	117
6.4 Scaffolding with Clone Mates and Retro-links	119
6.5 Discussion	121
6.6 Concluding Remarks	123
CHAPTER 7. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS	124
BIBLIOGRAPHY	127

LIST OF TABLES

Table 3.1	Summary of various previously developed fragment assemblers and EST clustering methodologies. PaCE is our methodology.	37
Table 3.2	(a) Run-times (in minutes) of assembly programs on an arbitrary mouse EST collection downloaded from GenBank. 'X' denotes that 2 GB memory was not sufficient for the program to complete. (b) Pairwise overlaps stored by CAP3.	39
Table 3.3	Run-time scaling of PCAP on 322,000 gene-enriched maize fragments. "X" denotes that the program was hung performing I/O operation.	41
Table 4.1	Quality assessment of PaCE and CAP3 clusters using clusters generated from different portions of the benchmark data set.	69
Table 4.2	Quality assessment of PaCE clusters with and without clone mates (CM) information, against the <i>Arabidopsis</i> benchmark clusters.	70
Table 4.3	Time (in seconds) spent in various components of parallel EST clustering as a function of the number of processors (p) for 20,000 ESTs.	72
Table 4.4	PaCE clustering run-time (in minutes) on ≈ 3.78 million mouse ESTs using 1,024 IBM BlueGene/L processors.	75
Table 4.5	Phase-wise run-time (in minutes) of PaCE clustering on ≈ 3.78 million mouse ESTs using 1,024 IBM BlueGene/L processors.	75
Table 4.6	Maize genomic fragment data types and size statistics: Methyl-filtrated (MF), High- C_0t (HC), Bacterial Artificial Chromosome (BAC) derived, and Whole Genome Shotgun (WGS).	78

Table 5.1	Confidence levels for different scenarios depending on the presence or absence of TSDs and motifs.	96
Table 5.2	Parameter set for our program with default values.	97
Table 5.3	Quality validation of running <i>LTR_par</i> and <i>LTR_STRUC</i> programs on the entire yeast genome.	98
Table 5.4	Run-time results of <i>LTR_par</i> on different genomes.	101
Table 5.5	Parallel run-times (in seconds) of <i>LTR_par</i> on the yeast genome and the chromosome 3R of <i>Drosophila</i>	102
Table 6.1	<i>LTR_par</i> parameter settings.	111
Table 6.2	Summary of the LTR retrotransposons identified in 4 maize BACs using <i>LTR_par</i>	112
Table 6.3	Classification of the LTR pairs in 4 BACs, with respect to a set of 10 shotgun samples obtained from each BAC at different coverages.	113
Table 6.4	Number of retro-gaps vs. all sequencing gaps. Measurements are averaged over all 10 samples of each of the two BACs.	114
Table 6.5	Summary of LTR pairs predicted by <i>LTR_par</i>	116
Table 6.6	Results of (i) scaffolding contig data for <i>BAC</i> ₄ (136,932 bp) using clone-mate information, (ii) retroscaffolding, and (iii) combined scaffolding using both clone mate and retro-link information.	120

LIST OF FIGURES

Figure 2.1	Illustration of <i>transcription</i> and <i>translation</i> — the biological mechanisms that produce protein molecules from the genetic code encoded in genes.	8
Figure 2.2	Illustration of the EST sequencing procedure.	11
Figure 3.1	Non-uniform sampling of mRNA resulting from the EST sequencing procedure.	24
Figure 3.2	Overlap layout suggesting a case of a (a) sequencing error, and (b) natural variation.	26
Figure 3.3	Illustration of the context of clustering in whole genome sequencing projects. Clustering F would partition it into two clusters, one corresponding to G_1 , and another to G_2	27
Figure 3.4	Number of overlaps stored by the CAP3 program while clustering different subsets of a rat EST data set. The peak memory usage reached 2 GB for 150,000 ESTs.	40
Figure 4.1	Examples to show the effect of transitive closure clustering in the context of genome assembly.	44
Figure 4.2	A naive serial clustering algorithm. The worst-case run-time and space complexities of the algorithm are $\Theta(n^2 \times l^2)$ and $O(n \times l)$, respectively.	45
Figure 4.3	Algorithm 1 improved by the promising pairs, clustering and pair generation heuristics.	49

Figure 4.4	Examples showing two cases of maximal matches. (a) A match α is maximal in two pairs of locations (i,j) and (i,k) between s_1 and s_2 . (b) Two maximal matches α and γ exist between s_3 and s_4	50
Figure 4.5	Algorithm for generating promising pairs from a generalized suffix tree.	53
Figure 4.6	Our sequence clustering algorithm. Steps 1 and 2 are collectively called the “preprocessing phase” and the remainder of the algorithm is called “clustering phase”.	54
Figure 4.7	(a) Dynamic programming table showing the computation of an alignment between s and s' anchored on a maximal match α . (b) Overlap patterns resulting from suffix-prefix alignment computation and their corresponding paths in the table.	57
Figure 4.8	Organization of the PaCE software.	58
Figure 4.9	A single master-multiple workers design for detecting overlaps and clustering in parallel, with responsibilities designated as shown. Arrows indicate the direction of communication.	62
Figure 4.10	The algorithm for the master processor. Bold font indicates a communication step.	64
Figure 4.11	The algorithm for each worker processor. Bold font indicates a communication step.	65
Figure 4.12	Illustration of quality validation measurements True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN). ‘U’ refers to the set of all possible pairs of the input ESTs.	68
Figure 4.13	Parallel scaling of PaCE clustering.	71
Figure 4.14	(a) Promising pair generation and alignment statistics of PaCE, as a function of data size. (b) The number of clusters as a function of the cluster size for 168,200 ESTs.	73
Figure 4.15	PaCE run-time as a function of the number of pairs allocated at a time for pairwise alignment.	74

Figure 4.16	Number of promising pairs generated vs. number of pairs aligned by PaCE while clustering ≈ 3.78 million mouse ESTs.	76
Figure 4.17	Idle run-time characteristics of PaCE clustering of the mouse EST data. (a) Average percentage idle time for each worker processor. (b) Percentage idle time of the master processor.	77
Figure 4.18	Illustration of our cluster-then-assemble framework.	79
Figure 4.19	Parallel run-times for constructing GST on inputs of sizes: (a) 250 million, and (b) 500 million <i>bp</i>	80
Figure 4.20	(a) Total parallel run-time for the entire clustering algorithm excluding that of GST construction. (b) The number of pairs generated, aligned, and accepted as a function of input size.	81
Figure 5.1	The structure of a full-length LTR retrotransposon.	85
Figure 5.2	Illustration of the process of creating a bucket B_k during preprocessing.	91
Figure 5.3	Algorithm to generate candidate pairs from a given bucket B_k	92
Figure 5.4	Illustration of the candidate pair generation algorithm.	93
Figure 5.5	Two alignments are performed for each candidate pair (i, j) : s_1 vs. s_3 and s_2 vs. s_4 . Dotted arrows indicate the directions of the alignments, and the two ovals indicate the anchoring match.	94
Figure 5.6	A case of nested retrotransposons in chromosome 10 of <i>S. cerevisiae</i> with 3 LTRs. The bottom-most line indicates the genome (not to scale). Part (a) shows the benchmark co-ordinates for the LTRs. Parts (b) and (c) show the two LTR _{par} predictions.	100
Figure 6.1	An example showing 6 pairs of clone mate fragments (shown connected in dotted lines) sequenced from a given BAC. The relative order and orientation between contigs c_1 and c_2 (also, between c_3 and c_4) can be inferred from the clone mates.	106

Figure 6.2	(a) Structure of a full-length LTR retrotransposon. (b) An example showing two contigs c_1 and c_2 with a retro-link between them.	109
Figure 6.3	Classification of LTR pairs based on the location of sequencing gaps, LTRs, and contigs. Dotted lines denote sequencing gaps. Retro-links correspond to the class CgC	113
Figure 6.4	Validation of two retro-links — between contigs c_{10} and c_{16} , and contigs c_{41} and c_{24} . Vertically aligned ovals denote overlapping regions, and squares denote retrotransposon hit through <i>tblastx</i> against the GenBank <i>nr</i> database.	119

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my major adviser, Prof. Srinivas Aluru, for guiding and inspiring me through out my graduate study. His ability to think quickly, clearly, critically, and creatively is just one of the several wonderful qualities that I admire and would like to emulate in the rest of my career. He has kept me focused through out, made me question myself at critical times, and helped me set a wonderful platform to start my research career.

My sincere thanks to Prof. Volker Brendel, Prof. Suraj C. Kothari, Prof. Patrick S. Schnable and Prof. Srikanta Tirthapura for serving in my committee. I am grateful to Prof. Brendel and Prof. Schnable for working with me on various projects, and for providing me a research experience to cherish and take to the next stage of my career. I thank Prof. Kothari for his involvement that was key for success and for the encouragement he provided from time to time. I am thankful to both Prof. Fernández-Baca and Prof. Tirthapura for serving as mentors and sharing their experiences in building a research career. Special thanks to Prof. Manimaran Govindarasu and Prof. Tirthapura for advising me at critical times of my graduate study.

I thank the following researchers for serving as my mentor during my industry internships: Dr. Mark Whitsitt in Pioneer Hi-Bred International Inc., Mr. Sam Ellis and Mr. Kurt Pinnow in IBM Rochester, and Dr. Robert S. Germain and Dr. Frank Suits in IBM T.J. Watson Research Center. Special thanks to Mr. Brian Smith, Ms. Nalini Polavarapu, Dr. Xiaowu Gai, Dr. Yan Fu and Prof. Daniel Voytas for collaborations and comments.

This research was supported by NSF grants primarily by ACI-0203782, and in part by CCR-009628 and DBI-0527192. My graduate study at Iowa State University was supported in parts by a Pioneer Hi-Bred Graduate Research Fellowship from January to December 2003,

and by an IBM Ph.D. Fellowship from August 2004 through May 2006.

It has been wonderful interacting with all the past and present members of my research group: Natsuhiko Futamura, Bhanu Hariharan, Mahesh Narayanan, Scott Emrich, Pang Ko, Sudip K. Seal, Sarah Orley, Srikanth Komarina, Benjamin Jackson, Yogy Namara, Xiao Yang, Abhinav Sarje and Chad Brewbaker. Special thanks to Scott Emrich for engaging in several active collaborations and for useful comments and discussions.

My friends at Iowa State University have been an immense source of energy and enthusiasm. Without them my life as a graduate student would have been uninteresting, to say the least.

ABSTRACT

Computational genomics is the study of the composition, structure, and function of genetic material in living organisms through computational means. The focus of research in computational genomics over the past two decades has primarily been the understanding of genomes and their numerous functional elements through the analysis of biological sequence data. The explosive growth in sequence data coupled with the design and deployment of increasingly high throughput sequencing technologies has created a need for methods capable of processing large-scale sequence data in a time and cost effective manner. In this dissertation, we address this need through the development of faster algorithms, space-efficient methods, and high-performance parallel computing techniques in the context of some key problems involving large-scale sequence analysis.

The first problem we address is the clustering of large collections of DNA sequences based on a measure of sequence similarity. Let n denote the number of input sequences, and l denote the average length of a sequence. We developed a new sequence clustering framework with the following novel features: (i) a space-efficient algorithm to limit the worst-case space complexity to $\Theta(n \times l)$, in contrast to the $\Theta(n^2 + n \times l)$ space required by most of the previously developed approaches; (ii) an algorithm to identify pairs of sequences containing long maximal matches, that generates these pairs on-demand in the decreasing order of their maximal match lengths in $O(n \times l + \text{number of pairs})$ run-time; (iii) a combination of algorithmic heuristics to significantly reduce the number of pairs evaluated for checking sequence similarity while maintaining the quality of clustering; and (iv) parallel strategies that provide linear speedup and a proportionate reduction in space per processor to facilitate large-scale clustering. We applied our clustering approach in the context of two biological applications — clustering Expressed Sequence Tags (ESTs), and genome assembly. The results demonstrate that our

approach has significantly enhanced the problem size reach while also drastically reducing the time to solution. To the best of our knowledge, this is the first parallel solution to scale with a linear speedup on thousands of processors. We implemented our algorithm into a software program called *PaCE*.

Identical or highly similar copies of the same sequence can be present in numerous locations of a genome. Such sequences are called repeats. The next problem we address is the *de novo* detection of repeats called LTR retrotransposons. Given a genome of length n , our algorithm to detect LTR retrotransposons has the following characteristics: (i) a space complexity of $\Theta(n)$; (ii) a method to produce high quality candidates for prediction in $O(n + \text{number of candidates})$ run-time; and (iii) a thorough evaluation of each candidate to ensure high quality prediction. Validation of our approach on the yeast genome demonstrates both superior quality and performance results when compared to existing software. We implemented our algorithm into a software program called *LTR-par*, which can be run on both serial and parallel computers.

In a genome assembly project, the fragments experimentally sequenced from a target genome are computationally assembled into “contigs” that represent the various contiguous genomic stretches from which the fragments were sampled. The next task is called scaffolding which is to order these contigs along the target genome. In this dissertation, we introduce a new problem called *retroscaffolding* for ordering contigs based on the knowledge of their LTR retrotransposon content and present an algorithm to achieve the same. Through identification of sequencing gaps that span LTR retrotransposons, retroscaffolding provides a mechanism for prioritizing sequencing gaps for finishing purposes. Our solution for retroscaffolding combines the techniques in *PaCE* for detecting pairs of similar sequences and the techniques in *LTR-par* for detecting LTR retrotransposons.

While many of the problems addressed in this dissertation have been studied previously, the main contribution in this dissertation is the development of methods that can scale to the largest available sequence data collections. As an illustration, we clustered the mouse EST collection in GenBank, which is the second largest available EST collection with ≈ 3.78 million ESTs, in just under 10 hours using 1,024 processors of an IBM BlueGene/L supercomputer.

CHAPTER 1. INTRODUCTION

DNA or *Deoxyribonucleic Acid* is one of the fundamental molecular entities inside a cell of a living organism. DNA encodes the genetic instructions for a cell to carry out its cellular development, and is also the hereditary material of an organism. An eukaryotic cell contains DNA molecules both inside its nucleus (as *chromosomes*), and outside (as *mitochondrial* and *chloroplast DNA*). All cells in an organism contain copies of the same set of DNA molecules. The term *genome* is used to collectively refer to all the DNA molecules within a cell. For example, the human genome consists of 23 pairs of chromosomes and a mitochondrial DNA.

Along a genome are various segments called *genes* that encode for proteins and *Ribonucleic Acids* (or *RNAs*) that carry out designated cellular functions. *Transcription* is a biological process by which portions of a gene is copied into an RNA molecule. These RNA molecules are subsequently released into the cytoplasm of the cell, where they are *translated* into their corresponding protein molecules.

A DNA molecule contains two strands intertwined in the form of a double helix. Each strand has molecules bonded to one another as a chain or *sequence* of four *nucleotides*: *Adenine*, *Cytosine*, *Guanine* and *Thymine*, abbreviated as *A*, *C*, *G*, and *T*. Nucleotides are also referred to as *bases*. The nucleotides along the two strands are linked to one another by a complementary relationship: $A \Leftrightarrow T$ and $C \Leftrightarrow G$; therefore, the sequence of one strand can be inferred from the sequence of the other. For this reason, the sequence length of a DNA molecule is typically measured in *base pairs* (*bp*). In contrast to a DNA molecule, an RNA molecule is single stranded and contains *Uracil* (*U*) instead of *Thymine*.

The process of determining the sequence of a DNA molecule is called *sequencing*. While the structure of a DNA molecule was discovered in the early 1950s, it was not until 1975

that the first experimental procedure to sequence a DNA molecule was designed [Sanger and Coulson (1975)]. This invention marked the beginning of a new era in molecular biology research. Rapid advancement in high-throughput cost-effective sequencing technologies led to tremendous growth in sequence repositories. With it arose a need for developing computational methods and automated tools for analyzing these sequence databases.

For more than two decades now, biological sequence analysis has been in forefront of molecular biology research, providing vital headways into the fundamental understanding of cellular mechanisms and significantly accelerating the process of molecular level discovery in modern biology. Within a span of only two decades, numerous genomes from a wide range of organisms from viruses to microbes to more complex mammalian species including the human have been sequenced and deciphered. Genes in the genomes of various species are being discovered and cataloged in databases along with corresponding functional and structural annotation; in many cases such projects are undertaken much earlier to the sequencing of the underlying genomes. Understanding the structural and functional roles of several other genomic entities such as regulatory elements and repeats is also of current interest.

The key to the recent accomplishments in molecular biology research is the interdisciplinary alliance that is prevailing between biologists and computer scientists. Biologists generate experimental data and pose questions of interest. Computer scientists design algorithms and software suites for efficiently processing these data and producing biologically meaningful results. The outcomes from such concerted efforts have impacted the research conducted in both communities. The hardness of many of the biological problems have opened new venues for computer scientists to collaborate and participate, while the outcome of their efforts have reciprocated to the biologists in the form of automated suites and tools for biologists, accelerating biological discovery.

In this dissertation, we focus on problems in genomics that involve sequence analysis, especially those for which there is a compelling need for high performance computing solutions capable of analyzing large-scale data. Designing computational solutions for analyzing sequence databases is challenging because of various factors:

- **Large-scale data:** High-throughput sequencing has facilitated quick generation of biological sequences, leading to an exponential growth in many publicly available sequence databanks. Handling large-scale data imposes heavy memory and run-time requirements.
- **Sequence variations:** Sequences generated through experiments may contain errors. For example, a nucleotide may either be mis-read or missed. During analysis, it is essential for ensuring quality to have a capability that can both identify such sequencing errors and differentiate them from sequence variations induced by natural mutation events.
- **Experimental costs:** The costs associated with sequencing experiments have significantly reduced with the advancements in the underlying technologies. For instance, the cost of sequencing a DNA molecule reduced from over \$10 per finished base in the early 1990s, to less than 10¢ per base in the early 2000s, and recently to under a tenth of a cent per base in 2005. It is still, however, substantially expensive to carry out genome scale sequencing projects [Pennisi (2005)]. Given that most experimental effort is spent in generating data that can provide information that is sufficient to their subsequent computational analysis, it is important that computational methods are designed with a goal of extracting as much information as possible from as little experimental data. Moreover, if computational methods can provide an insight into the information content of the data, such insights can serve as feedback to help biologists reduce experimental costs without compromising on quality.
- **Computational requirements:** Many problems that involve sequence analysis are computationally hard, necessitating polynomial time approximation algorithms. Even such solutions, however, often require large run-time and memory requirements for large-scale input sizes.

Traditionally, most of the algorithms and software programs developed for analyzing sequence databases have been design-intended for serial computers. The complexities of problem instances aggravated by the factors mentioned above, however, have made it increasingly difficult for a continued deployment of such methods. Quick-fix solutions that run a serial code on

a high-end computer with tens of gigabytes of shared memory have been developed to alleviate this situation. Concurrent with these developments in computational genomics research, the supercomputing technologies have also experienced a phenomenal growth. Processing capabilities that can support thousands to tens of thousands of CPUs, with access to thousands of gigabytes of memory are now available in the form of distributed memory machines. Given the high complexities involved in analyzing large-scale sequence databases, these large-scale supercomputers can provide an excellent platform for carrying out research. The key challenge is therefore on the algorithm designers to design methods that can efficiently exploit the memory and compute power to produce high quality biologically meaningful results.

The contributions in this dissertation are as follows:

- **Scalable clustering framework:** We focus on problems for making sequence level discovery of genomic and genic data. In particular, we address two important problems: Expressed Sequence Tag (*EST*) clustering and genome assembly. The problems can be directly applied to various genome level and gene related studies such as gene discovery, gene structural and functional annotation, alternative splicing studies, and gene expression profiling.

We formulate the compute intensive phases of these problems as a sequence clustering problem that involves computation of pairwise sequence overlaps. We then provide a space and time efficient parallel algorithm for distributed memory parallel computers [Kalyanaraman *et al.* (2003a,b, 2006b)]. We demonstrate the utility of our algorithm by clustering large EST collections and applying our clustering framework in the on-going efforts to assemble the maize genome. This research has enabled the clustering of millions of genomic and EST sequences in matters of hours without compromising on quality. It is also the first method that has demonstrated a linear scaling to over thousands of processors. Our clustering framework provides a generic and efficient solution to any sequence analysis problem that can in principle, be solved by computing the overlap between each sequence in the input to every other sequence.

- **LTR retrotransposon identification:**

Long Terminal Repeat (LTR) retrotransposons constitute one of the most abundant classes of repetitive elements in several eukaryotic genomes. Detection of these repetitive elements require methods that can analyze genome-scale data. We developed a new algorithm for detecting genomic regions that contain the structural characteristics of a full-length LTR retrotransposons [Kalyanaraman and Aluru (2005b, 2006)]. The factors that distinguish our algorithm from other contemporary approaches are as follows: (i) a novel method to preprocess the entire genome sequence in linear time and produce higher quality “candidates” in constant time per candidate, (ii) a thorough evaluation of each candidate in order to ensure a high quality prediction, (iii) a robust parameter set encompassing both structural constraints and quality controls provided by the users with a high degree of flexibility, and (iv) serial and parallel software programs implementing our algorithm. Our validations conducted on the yeast genome show both superior quality and run-time when compared to other software. Performance studies on many large genomes such as that of *Arabidopsis* (≈ 119 million *bp*), *Drosophila* (≈ 118 million *bp*), and Chimpanzee (≈ 3 billion *bp*) also show multi-fold speedups over contemporary software.

- **Scaffolding using LTR retrotransposons:**

The presence of repeats in genomes has been traditionally viewed as a source of complication while assembling genomes. We introduce a problem called *retroscaffolding* [Kalyanaraman *et al.* (2006a)] that, on the contrary, can benefit from the abundance of LTR retrotransposons. Retroscaffolding is a new variant of the well-known problem of scaffolding, which is aimed at determining the order of the sequences output by a genome assembly program along a target genome. Scaffolding is the last computational step, after which the genome sequence is “finished” through experimental means. The retroscaffolding approach is not meant to supplant but rather to complement other scaffolding approaches. There are two more advantages in retroscaffolding: (i) it allows detection of regions containing LTR retrotransposons within the unfinished portions of a genome and can therefore guide the process of finishing, and (ii) it provides a mechanism to lower se-

quencing costs without impacting the quality of the assembled portions containing genes. Sequencing and finishing costs dominate the expenditures in whole genome projects, and it is often desired in the interest of saving cost to reduce such efforts spent on repeat regions of a genome. The retroscaffolding technique provides a viable mechanism to this effect.

The dissertation is organized as follows. Chapter 2 provides a brief overview on the biological concepts and terminology required to understand the problems and applications described in this dissertation. We also outline popular sequence overlap computation methods. In Chapter 3, we formulate the problems of EST clustering and genome assembly as problems involving sequence clustering. We then provide an extensive review of literature describing the various computational methods previously developed for these two problems. In Chapter 4, we describe our parallel clustering algorithm, and report the results we achieved in two main applications: clustering various large-scale EST data collections including 3.7 million mouse ESTs, and clustering 1.6 million maize genomic fragments as part of the on-going maize genome sequencing initiative. The platform used for our experiments is a 1,024 node IBM BlueGene/L supercomputer at Iowa State University. In Chapter 5, we describe the algorithms and software we developed for *de novo* identification of LTR retrotransposons. In Chapter 6, we introduce the retroscaffolding problem, describe our algorithm, and demonstrate its utility on maize genomic data. Chapter 7 concludes the dissertation with a discussion on future research directions.

CHAPTER 2. SEQUENCE ANALYSIS: BIOLOGICAL BACKGROUND AND TERMINOLOGY

The genome of an organism is the collection of all DNA molecules in a cell of a living organism. Genes are portions within a genome that encode for proteins and RNA molecules that are responsible for various functions in cellular development. Genes could be part of either strand of the genomic DNA. An eukaryotic gene can be viewed as a sequence of alternative segments called *exons* and *introns*. The biological mechanism that leads to the production of proteins in an eukaryotic genome is illustrated in Figure 2.1, and can be described as follows.

In the first stage called *transcription*, a copy of the gene is made into a preliminary RNA molecule called the *pre-mRNA*. Once this single stranded pre-mRNA is released into the nucleus, a splicing mechanism splices the exons by removing the intervening introns and creates a corresponding RNA molecule called the *messenger RNA* or (*mRNA*). The combination of exons selected during transcription need not be unique: different transcription events of the same gene could use different combinations of exons (and sometime even introns). This phenomenon, called *alternative splicing*, provides the capability for a gene to code for more than one mRNA molecule (and thereby multiple protein products). Once transcribed, the mRNA molecule, also called an *mRNA transcript*, is released into the cytoplasm of the cell. The overall process of a gene transcribing for an mRNA molecule is also referred to as *gene expression*. The number of mRNAs transcribed from a gene indicates its *expression level* under a set of provided conditions.

In the next stage, called *translation*, the mRNA molecule binds itself to a molecular complex, and the sequence is translated into a corresponding protein molecule; the translation reads the mRNA sequence in blocks of three nucleotides, where each three-letter sequence

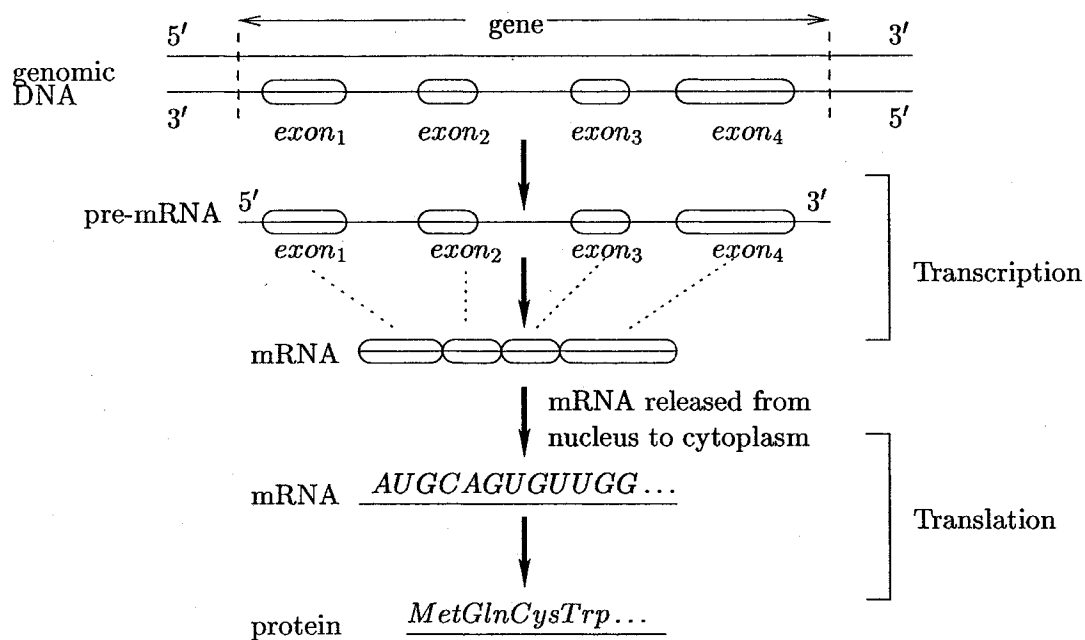


Figure 2.1 Illustration of *transcription* and *translation* — the biological mechanisms that produce protein molecules from the genetic code encoded in genes.

called a *codon* translates into one of 20 amino acids. The mapping from codons to amino acids is referred to as the *genetic code*, which is almost universal across organisms. It is now known that not all genes transcribe for protein-coding mRNAs. Such genes are labelled *pseudogenes* because of a lack of protein product. Nevertheless, it is also sometimes possible that a protein-coding mRNA is not successfully translated into its corresponding protein product.

2.1 Genomic Repeats

Identical or approximately identical copies of a subsequence could be present in multiple locations of a genome. These subsequences are called *repeats*. There are numerous types of repeats, and one of the most abundant type of repeats are the transposons.

2.1.1 Transposons

Transposition is a process by which a sequence of DNA can move to or copy itself at different positions within the genome. The DNA sequences that transpose themselves are called *transposons*. Based on the mechanism of transposition, there are two main types of DNA transposons. The first class of transposons are those that can cut themselves from their current genomic location and then insert themselves into another. These are simply referred to as *DNA transposons*. The second class of transposons make a copy of themselves into an intermediate RNA molecule, which is then reverse transcribed and inserted as a DNA molecule into another genomic location. Because this transposition mechanism is similar to that in retroviruses, these transposons are called *retrotransposons*. There are several subclasses of retrotransposons.

- **LTR Retrotransposons:** These retrotransposons are characterized by two long terminal repeats, and are therefore called *Long Terminal Repeat* (or *LTR*) retrotransposons.
- **Non-LTR Retrotransposons:** These include different subclasses of retrotransposons such as *long interspersed elements (LINEs)*, *short interspersed elements (SINEs)*, and *Alu* sequences.

2.2 Sequencing Technologies

Sequencing is the process of determining the chain of nucleotides in a DNA or RNA molecule, or the chain of amino acids in a protein molecule. In 1975, Sanger and Coulson [Sanger and Coulson (1975)] designed the first method to sequence DNA molecules and called it a “plus and minus” method. Two years later, Sanger *et al.* designed another method called the *chain termination* method [Sanger *et al.* (1977)] similar to the plus and minus method. Currently, almost all sequencing methods are based on the chain termination method. Since its invention, significant technological advancements have been made towards increasing the throughput and accuracy, and towards reducing the cost per base of sequencing.

With current methods for DNA sequencing, it is possible to sequence ≈ 500 -1000 *bp* nucleotides at a stretch with an accuracy of 98-99%, implying a maximum sequencing error rate of 1-2%. However in reality, biological molecules are much longer — genomes span a few tens of thousands to even billions of nucleotides; a gene may span thousands to a few tens of thousands of nucleotides; and a protein can span hundreds of amino acids. To extend the reach of sequencing a target molecule's full length, technologies adopt the following strategy of sequencing randomly chosen "fragments" from many copies of the molecule, and subsequently relying on computational means to group or assemble the target molecule.

In this section, we will briefly review the different sequencing technologies and the types of sequences that can be generated from them.

2.2.1 Expressed Sequence Tag Sequencing

Expressed Sequence Tags (ESTs) are sequences obtained from mRNA libraries. ESTs are sequenced as follows (illustrated in Figure 2.2): Depending on the conditions a living tissue is subjected, different genes in the tissue could express at different levels. The mRNA molecules transcribed during an experiment are extracted and isolated [Chomczynski and Sacchi (1987)]. Subsequently, the isolated mRNA molecules are subjected to reaction with an enzyme called *reverse transcriptase*. This converts the mRNA to its double stranded DNA counterpart (i.e., with an added complementary strand and with *U* replaced by *T*) called the *complementary DNA* (or *cDNA*) molecule. Due to underlying experimental limitations, however, this procedure may not complete on the entire mRNA thereby resulting in partial length cDNAs. In order to provide sufficient coverage over the entire mRNA, multiple and possibly redundant such partial length cDNAs can be generated and each cloned using a cloning vector. Using primers targeted for known vector sequences near the ends of the inserts, the nucleotide sequence of each inserted clone can then be read over a single pass from either end, resulting in fragments called ESTs that are about 500 *bp* long. Because this procedure may oversample the end regions of a cDNA clone, the untranslated regions at the ends of the corresponding mRNA may also get proportionately over-represented. If it is desirable to avoid such bias, sequencing

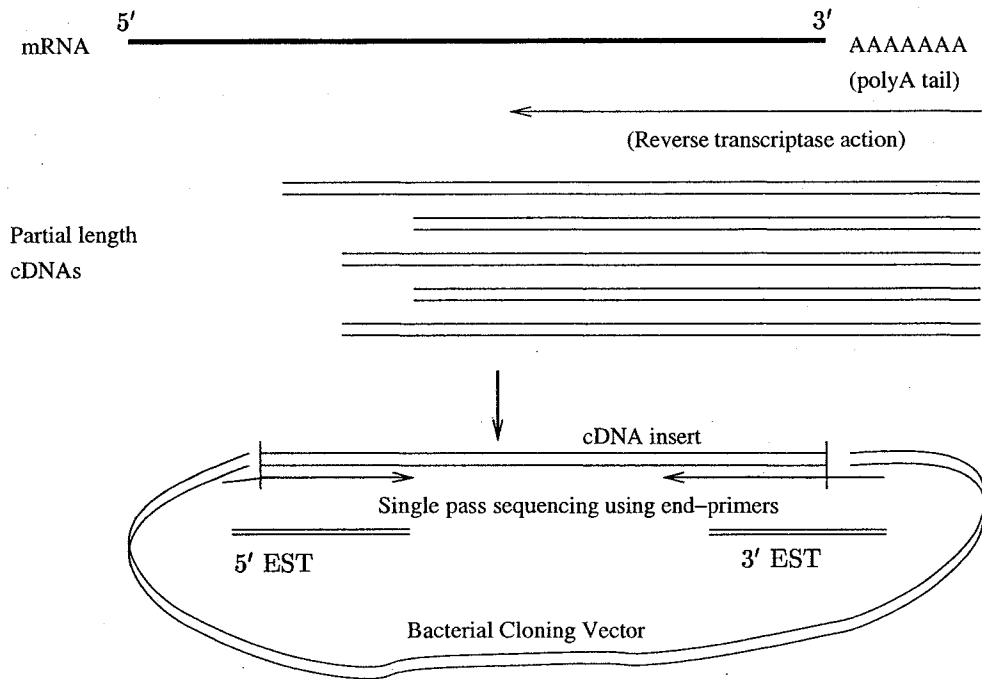


Figure 2.2 Illustration of the EST sequencing procedure.

is started from random locations on the cDNA insert using randomly created sequences as primers, or through application of restriction enzymes, breaking the cDNA insert before its shreds are sequenced from their ends.

Cost-effective high throughput sequencing of ESTs has largely been facilitated by the simplicity of the single pass sequencing technology. Nevertheless, the technology does not always generate accurate sequences. Nucleotides are sometimes misread or ambiguously interpreted resulting in low-quality sequences. It is also possible, although rare, that two cDNA sequences representing two distinct mRNA sequences are spliced together resulting in an artifact known as a *chimeric cDNA*. When cloned and sequenced, the resulting ESTs could contain portions from either cDNA, potentially confounding their subsequent analysis.

During sequencing, the two ESTs that originate from the ends of a cDNA insert are sometimes tagged with the clone identifier and stored in the header of the EST sequences in the database. This auxiliary information proves valuable in later stages of the sequence analysis —

pairs of ESTs having the same clone identifier are labeled *clone mate pairs* (or *clone pairs*) and are immediately associated with a common source transcript, obviating the need to compute additional evidence to establish their relationship. Mate pair information is not unique to EST sequencing; it is also common in genome sequencing techniques that involve sequencing from a clone insert. Also available sometimes with EST sequence data are “trace data” that contain the quality values for each base position of the ESTs. Such trace data are measures of sequence quality and are valuable during analysis.

Genes express differentially depending on the tissue they reside and the subjected experimental conditions. Consequently, EST data generated by conventional sequencing techniques have ESTs from overly expressed genes in a proportionately higher concentration than from sparsely expressed genes. Such non-uniformity may be desirable if the ESTs are used in gene expression related studies; otherwise, not only is the effort spent in sequencing multiple ESTs covering the same regions unnecessary, but such non-uniformity may also add significant challenges to the computational methods for EST analysis. For example, one unique EST per gene is sufficient for estimating the number of genes in an organism, while oversampling may significantly increase the computation as a function of the number of ESTs represented per gene. To alleviate this problem, many variations to the original sequencing technique have been invented and these methods can be classified into two groups: normalization and subtractive hybridization. Normalization achieves a balance in the cDNA population within a cDNA library [Patanjali *et al.* (1991); Soares *et al.* (1994)], while subtractive hybridization reduces overly represented cDNA population by selectively removing sequences shared across cDNA libraries [Duguid and Dinauer (1990); Fargnoli *et al.* (1990); Schmid and Girou (1987); Schweinfest *et al.* (1990); Travis and Sutcliffe (1988)]. For a survey of these two methods see [Baldo *et al.* (1996)].

2.2.2 Whole Genome Sequencing

2.2.2.1 Whole Genome Shotgun Sequencing

One of the most popular ways to sequence an entire genome is whole genome shotgun (WGS) sequencing, first used to sequence the genome of bacteriophage λ [Sanger *et al.* (1982)]. In this method, random locations of a target genome are sampled by a shotgun approach, and short sequences ($\approx 5,000$ bp) starting at these locations are extracted. The short sequences are then cloned in bacterial vector colonies, and are sequenced from both sides from each vector. The resulting sequences are of length ≈ 500 -1,000 bp and are called *shotgun fragments*.

In WGS sequencing, a target genome can be sampled such that each of its base can be expected to be *covered* by a specified number of fragments. This number is called *sequencing coverage* and is denoted by 'X'. The number of fragments sequenced in a WGS project is a function of the length of the target genome and the desired sequencing coverage. For example, a 6X coverage of a 3 billion bp genome will result in approximately 36 million fragments, assuming an average length of 500 bp for each fragment. Given the randomness of the shotgun procedure, however, it cannot be guaranteed that each base will be covered by at least one fragment. In practice, significantly long stretches of genome are left uncovered in sequencing, and each of these stretches is called a *sequencing gap*. Specifying a high coverage decreases the frequency and lengths of such gaps, although at a proportionately higher sequencing cost.

In general, whole genome shotgun sequencing is relatively cheaper when compared to other sequencing technologies because the locations to sequence are chosen at random. The approach has been used for sequencing a number of genomes including the human genome [Adams *et al.* (2000); Venter *et al.* (2001a,b, 2004)].

2.2.2.2 Hierarchical Sequencing

An alternative methodology to whole genome shotgun sequencing is hierarchical sequencing. In this approach, a genome is first broken into numerous smaller clones of size up to 200 kbp each called a *Bacterial Artificial Chromosome* (or *BAC*). Next, a combination of these BACs that provide a *minimum tiling path* based on their locations along the genome is deter-

mined. Each selected BAC is then individually sequenced using a shotgun approach generating numerous short (≈ 500 - $1,000$ bp long) shotgun fragments. This method is also called *BAC-by-BAC sequencing* or *clone-by-clone sequencing* because of its hierarchical strategy. Even though the associated costs of creating BAC colonies makes this a costlier alternative to whole genome shotgun sequencing, this method provides additional information that facilitate an accurate analysis of the fragments. Hierarchical methods similar to BAC-by-BAC sequencing involve different types of colonies such as *Yeast Artificial Chromosomes* and *Fosmids*. The BAC-by-BAC approach has been used for sequencing several complex eukaryotic genomes including that of the human [Consortium (2001)] and maize [NSF (2005)].

2.2.2.3 Gene-enriched Sequencing

A majority of the genomic DNA content in eukaryotic genomes are repetitive regions and only a very small portion typically contain genes, e.g., the maize genome is estimated to contain less than 20% of it in genes. To selectively sample genic portions of the genome during sequencing, biologists have developed two gene-enrichment sequencing strategies for plant genomes: Methyl Filtration (*MF*) [Rabinowicz *et al.* (1999)] and High- C_{0t} (*HC*) sequencing [Yuan *et al.* (2003)]. MF sequencing discards genomic portions that are highly methylated, which are a typical characteristic of repetitive regions. The HC technique isolates low-copy (or genic) regions of a genome based on hybridization kinetics. These two techniques have been used to sequence the gene-riched portions of the maize genome [Palmer *et al.* (2003); Yuan *et al.* (2003)],

2.2.3 454 Sequencing

The 454 sequencing is a recently developed sequencing technique [Margulies *et al.* (2005)] that uses microfabricated high-density picolitre reactors. While this technique is still at the early stages of its development, it has attracted the attention of several researchers in genome sequencing projects primarily because of its superior throughput. However, the average length of fragments that can be sequenced with current technology is relatively short — only ≈ 100

bp.

2.3 Pairwise Sequence Alignment Computation

For computational purposes, all biological sequences can be represented as strings over a finite alphabet — for DNA and RNA sequences the alphabet is 4 characters, and for proteins it is 20 characters. This property has been taken advantage of in various sequence analysis problems. The relationship between two sequences is typically established by comparing the two sequences, and detecting any potential “overlap” between them. Because the sequences typically represent much smaller pieces of the original source sequence, the presence of overlap can be used as an evidence to link two sequences without prior knowledge of the underlying source sequence. For the remainder of the dissertation, we use the terms “sequence” and “string” interchangeably. Also, we use the term “subsequence” to mean a substring throughout the remainder of this dissertation except in this section.

The problem of detecting an overlap between two sequences can be formulated as the problem of computing an “optimal” pairwise sequence alignment. An alignment between two strings is an ordered list of matches, mismatches, insertions, and deletions between the two strings. A contiguous stretch in an alignment containing more than one deletion (alternatively, insertion) is referred to as a “gap”. A “score” of an alignment is computed from the number of its matches, mismatches and gaps. An “optimal alignment” is one with the maximum score. Modeling biological pairwise sequence overlaps as sequence alignments provides an effective mechanism to account for sequencing errors and sequence level disagreements arising due to mutation events.

There are several types of alignments that can be computed between two strings based on the portions of the two strings considered for alignment scoring. Given two strings, s_1 and s_2 , of lengths $m \geq 0$ and $n \geq 0$ respectively:

- **Global Alignment:** align the whole of s_1 against the whole of s_2 [Needleman and Wunsch (1970)]. this alignment formulation is suited for comparing two highly similar strings;

- **Local Alignment:** align an arbitrary substring of s_1 against an arbitrary substring of s_2 [Smith and Waterman (1981)]; this is suited for detecting local similarities between two strings;
- **Semi-global Alignment:** align an arbitrary suffix of s_1 (alternatively, s_2) against an arbitrary prefix of s_2 (alternatively, s_1). Note that the global alignment is a special case of this alignment if the suffix and prefix are the entire strings. This alignment is also sometimes called the *suffix-prefix* or *end gaps free* alignment by virtue of the fact that the gaps at either end of an alignment is not penalized (i.e., given a score of 0). The semi-global alignment is a popular choice in fragment assemblers for detecting pairwise overlapping fragments;
- **Spliced Alignment:** align the whole of s_1 (alternatively, s_2) against an arbitrary subsequence of s_2 (alternatively, s_1) [Gelfand *et al.* (1996); Schlueter *et al.* (2003); Usuka *et al.* (2000)]. This formulation is suited for aligning an EST/cDNA sequence with genes/genomic regions. The outcome of a spliced alignment can be used to both locate expressed portions within genes and annotate them with their corresponding expressed products;
- **Syntenic Alignment:** align an arbitrary pair of subsequences from either strings [Delcher *et al.* (1999); Huang and Chao (2003); Rajko and Aluru (2004)]. This alignment formulation is appropriate for comparing genomes of two evolutionarily related organisms. A significant syntenic alignment is a chain of local similarities (representing conserved genic regions) interspersed by long gaps (representing the long stretches of divergent junk regions between genes).

Using dynamic programming, computing an optimal global, local, semi-global, spliced and syntenic alignments take $O(m \times n)$ time, and $O(m + n)$ space [Hirschberg (1975)]. Alignments are typically computed using a $(m + 1) \times (n + 1)$ table. Computing a global alignment between a pair of strings of similar lengths and expected high sequence similarity can be accelerated using a *banded computation* technique [Fickett (1984)]. In this technique, the alignment computation

starts on the diagonal of the dynamic programming table and progressively expands either side in a band until it can be guaranteed that no optimal alignment can lie outside the band. The main idea is to avoid computing the entire table, although it may be necessitated in the worst case. This banded technique can also be extended for non-global alignments if individual pairs of local regions that are potentially aligning can be identified through other quicker means.

For the above alignments, alignment scoring could vary depending on the mechanism used to penalize gaps. A straightforward mechanism is to penalize gaps proportional to their lengths. Another popular gap function is called the *affine gap penalty function* [Gotoh (1982)], in which gaps exceeding a cutoff length are given a constant penalty. Affine gap penalty functions are generally preferred because they provide a better model for biological events such as mutations and polymorphisms.

Besides alignment scoring, there are several other ways to measure pairwise sequence similarity [Burke *et al.* (1999); Ukkonen (1992)]. While computing these measures may not model the problem accurately for sequence errors and expected patterns in overlaps, these techniques are usually sought as faster alternatives to alignment based methods. For a survey of alignment and other sequence similarity measures and methods see [Jackson and Aluru (2005); Gusfield (1997a); Setubal and Meidanis (1997)].

CHAPTER 3. SEQUENCE CLUSTERING: PROBLEMS AND APPLICATIONS

Broadly, there are three main goals (in that order) in genomics research: (i) discover the composition and structure of all naturally occurring biological DNA, RNA and protein molecules, (ii) understand their behavior under different conditions both as an independent molecular entity and as part of a biomolecular complex system, and (iii) advance the state of genetic capabilities in medical and agricultural research towards the betterment of an organism's health and/or productivity. This chapter and the next focus on methods towards achieving the first goal, which is to be able to determine the composition of molecules. To this effect, sequence databases are as an immense source of information.

Given the large sizes of sequence databases and the vast diversity in the sources they represent, a necessary first step in their computational analysis is to identify the several sources they represent and organize them into several conceptual groups. For example, given a collection of ESTs sequenced from an organism, identifying the different genes represented among them provides a finer level insight into the genetic composition of the organism. It is customary to formulate this group identification task as a sequence clustering problem, with the criteria for clustering designed to model at best desired biological criteria.

In what follows, we will focus on two problems in clustering DNA sequence databases, and present their significance through their biologically motivated applications.

3.1 Clustering DNA Sequences

3.1.1 Clustering of Expressed Sequence Tags

ESTs represent sequences sampled from expressed portions of genes. Given a collection of ESTs collected from an experiment, we can ask the following questions:

- **Q1** What genes are expressed in the experiment?
- **Q2** What are the mRNA transcripts that correspond to the expressed genes?
- **Q3** Were any of the genes alternatively spliced during the experiment or relative to their expression in other experiments?
- **Q4** What are the expression levels of genes that are expressed in the experiment?

These are some of the important questions that can be answered by analyzing EST databases, even without any *a priori* knowledge on the genes in the underlying organisms including their count or composition. In other words, analyzing ESTs provide valuable insights into expressed genes regardless of the availability of the sequence of the underlying genome. EST sequencing also provides an alternative mechanism to sample gene-rich portions of the genome.

3.1.1.1 Problem Statement

Given an arbitrary collection of ESTs, partition the ESTs into “clusters” such that each output cluster corresponds to a unique gene (alternatively, mRNA transcript).

3.1.1.2 Applications

- **Transcriptome and Gene Discovery:** One of the earliest identified merits of EST data is in discovering genes with expression evidence [Adams *et al.* (1991); Boguski *et al.* (1994)]. A sequencing experiment can trigger the expression of multiple genes in a target cell/tissue, and so the resulting EST data is a segmented representation of the transcribed portions of all these expressed genes. Thus, clustering an EST collection is equivalent to reverse-engineering the process that sequenced the ESTs in the first place, and the set

of clusters would correspond to the portion of the *transcriptome* (or expressed portions of the genome) represented in the underlying sequence data. However, such EST-to-source mapping is not readily available and one of the main challenges in clustering is its inference from other information contained within the sequence data.

Any two ESTs that cover a common segment within their gene source are expected to show a significant sequence overlap in the corresponding region(s). Therefore, detection of pairwise overlaps among the EST data can serve as a basis to cluster ESTs. Furthermore, if it is possible to assemble the ESTs in each cluster consistent to the pairwise assembly, then the resulting supersequence is likely to correspond to the mRNA transcript that gave rise to the set of ESTs in that cluster. The UniGene project undertaken by NCBI is a typical example of clustering ESTs by gene source [Pontius *et al.* (2003)]; and the Gene Index project undertaken by The Institute of Genome Research (TIGR) clusters by transcript source [Quackenbush *et al.* (2000)].

Given the high costs associated with whole genome projects, the genomes of many organisms of interest are unlikely to be sequenced. In many cases, biologists still depend on EST data to help them with building transcriptomes and gene lists. Numerous transcriptome projects have benefited from EST databases in the past [Boguski and Schuler (1995); Camargo *et al.* (2001); Carninci *et al.* (2003); Caron *et al.* (2001); Okazaki *et al.* (2002)]. EST based gene discovery and transcriptome construction projects, however, are not guaranteed to cover the gene space entirely — i.e., genes that are not transcribed during sequencing will be missed subsequently by an EST based discovery process. For example, in Berkeley Drosophila Genome Project, only about 70% of the genes were covered by the cDNA/EST based gene discovery [Stapleton *et al.* (2002)].

- **Gene Annotation and Alternative Splicing:** Once clustered, ESTs within a cluster can be used to annotate their putative source gene's structure through spliced alignment techniques [Gelfand *et al.* (1996); Schlueter *et al.* (2003); Usuka *et al.* (2000)]. Exonic and intronic boundaries within expressed genes can be marked using the alignment pattern of a gene sequence with an EST derived from it. EST based gene annotation has been a

vibrant research area [Bailey *et al.* (1998); Bono *et al.* (2002); Huang *et al.* (1997); Jiang and Jacob (1998); Okazaki *et al.* (2002); Seki *et al.* (2002); Whitfield *et al.* (2002); Zhu *et al.* (2003)].

As ESTs are derived from mRNAs, they also provide a means to discover alternative splicing events of the underlying genes [Burke *et al.* (1998); Kan *et al.* (2001); Mironov *et al.* (1999); Modrek and Lee (2002); Modrek *et al.* (2001)].

- **Alternative Poly-adenylation:** Poly-adenylation occurs during transcription and is the process by which an mRNA sequence is terminated at its 3' end. At the terminated end, the transcription process appends a repeat sequence of the nucleotide adenine (termed as a "polyA tail"), which plays important roles in the mRNA's stability and translation initiation. Alternate choice of polyadenylation sites results in corresponding variations at the mRNA ends and is considered an important post-transcriptional regulatory mechanism. ESTs sequenced from the 3' ends of the mRNAs are used to determine alternate polyadenylation sites in genes [Gautheret *et al.* (1998)]. ESTs are first clustered and assembled into sequences representing the underlying mRNA transcript. While assembling, polyA discrepancies are detected in positions having additional evidence of conserved motifs for polyadenylation sites such as the hexamer *AAUAAA*, which are then recorded as possible sites of alternate polyadenylation.
- **Gene Expression Studies:** Before the advent of the microarray technology, gene expression and co-regulation related studies were primarily dependent on EST data. During a sequencing experiment, the number of ESTs derived from an expressed gene is correlated to its expression level under the experimental conditions. In 1995, a technique called Serial Analysis of Gene Expression (SAGE) was developed based on the above philosophy [Velculescu *et al.* (1995)]. For examples of EST based gene expression studies, see [Ewing *et al.* (1999); Mao *et al.* (1998)]. For a review on different approaches to differential gene expression studies including EST based analysis, see [Carulli *et al.* (1999)]. In addition to expression profiling, ESTs are also used to design oligos for microarray

chips [Kaprois *et al.* (1994); Zhu and Wang (2000)].

- **Single Nucleotide Polymorphisms:** Single Nucleotide Polymorphisms (SNPs) are the most abundant class of genetic variation occurring almost every 1,200 *bp* along the human genome. SNPs are studied for mapping complex genetic traits. SNPs that occur on coding and regulatory sequences could alter the expression pattern or even the transcriptional behavior of the gene. SNPs have also been identified as causes for various diseases [Collins *et al.* (1997)]. Such SNPs can be identified as nucleotide variations in assembled ESTs [Garg *et al.* (1999); Marth *et al.* (1999); Picoult-Newberg *et al.* (1999); Ye and Parry (2002)]. However, these variations need to be distinguished from those variations seen among ESTs from paralogous genes, or occurring in ESTs due to sequencing errors; otherwise the SNP identification process may result in false predictions. This is usually accomplished by observing a probabilistic distribution that also takes into account the quality values of nucleotides in question. A large database of all identified SNPs is maintained by the NCBI (<http://www.ncbi.nlm.nih.gov/projects/SNP/>) and is called dbSNP [Sherry *et al.* (2001)]. Although a majority of the SNPs in this database are that of human and mouse, the database is open to SNPs from any species and occurring anywhere within its genome.

3.1.1.3 Computational Challenges

Overlap Detection

The primary source of information to achieve clustering is the detection of pairwise overlaps between ESTs. Pairwise overlaps can be detected by computing alignments and the choice of an appropriate overlap detection scheme is dictated by the goal of clustering. If the goal is to cluster ESTs based on mRNA source, then a semi-global alignment computation is suited because it detects suffix-prefix overlap expected out of two ESTs derived from an overlapping region on the mRNA transcript. However, if clustering by gene source is desired, then in addition to a suffix-prefix type of alignment there is a need to detect overlaps between ESTs derived from different alternatively spliced transcripts of the same gene. This can be mod-

eled as finding a consistent chain of local alignments (better known as a *syntenic alignment*) corresponding to the regions containing shared exons.

A naive approach to clustering is to first choose the overlap detection scheme, run it on each pair of input sequences, and in the process form the clusters using only those pairs with a significant overlap. The main issue with this approach is that its scalability is limited by the quadratic increase in the number of pairs. This can be further aggravated by the high computation cost associated with detecting each overlap — the run-time for aligning two sequences through a standard dynamic programming approach is proportional to the product of their lengths.

Thus, a primary challenge in designing clustering algorithms is to be able to significantly reduce the run-time spent in detecting overlaps, and still obtain correct clustering that would have resulted had all pairs been considered. There are two independent ways of achieving this reduction: (i) reduce the cost of each pair computation by opting for a less rigorous and/or approximate method instead of aligning two sequences, and (ii) devise faster methods to detect sequence pairs in advance that exhibit significant promise for a good alignment and then perform rigorous alignment only on those selected “promising pairs”.

The inherent nature of sampling in EST data can add significantly to the computational complexity of the clustering process. Even if one were to devise a scheme that intelligently discards all non-overlapping pairs from overlap computation, the number of genuinely overlapping ESTs may still be overwhelming in practice. This is because the sequencing procedure may oversample the ends of the mRNA transcripts (giving them a deep coverage) while under-sampling their mid-regions. The result is what we see in Figure 3.1, i.e., a vertical tiling of ESTs on a source mRNA transcript. Thus the number of genuinely overlapping pairs could grow at a quadratic rate as a function of the number of ESTs covering each transcript, which could be very high for transcripts arising from over-expressed genes. This raises a critical issue when dealing with large inputs containing hundreds of thousands to millions of ESTs, especially limiting the applicability of those software packages designed to handle only uniformly sampled sets (e.g., fragment assemblers).

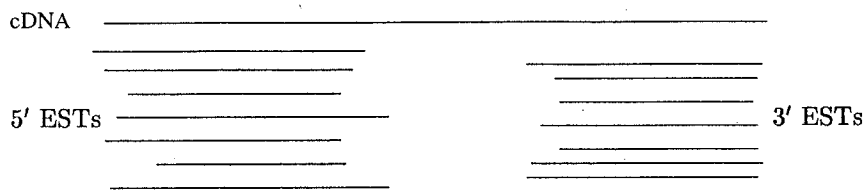


Figure 3.1 Non-uniform sampling of mRNA resulting from the EST sequencing procedure.

Sequencing Errors and Artifacts

With current technology, even though the error rates are as low as 1-2%, it is important for a clustering algorithm to handle errors in order to guarantee a high prediction accuracy. Errors such as an incorrectly interpreted, included, or excluded nucleotide in a sequence are typically handled during overlap detection — by modeling such errors as mismatches, insertions and deletions in alignments. There are other types of errors and artifacts that can be detected at an earlier stage and most of these errors are detected in a preprocessing step prior to overlap detection:

- During sequencing, ESTs may get contaminated with the vector sequences adjoining the cDNA clones. These sequences are easy to detect because they are part of the known vector DNA sequence and are expected to occur at ends of ESTs. During preprocessing, such sequences are detected and removed.
- The sequencing procedure may also have ambiguously read some bases and may have marked such bases with low quality values. In the resulting ESTs, these bases are marked with special characters such as ‘N’ or ‘X’, so that they can be treated accordingly by a subsequent overlap detection scheme.
- ESTs derived from 3’ ends of an mRNA usually retain portions of the mRNA’s polyA tail. The presence of such polyA tails in ESTs may be of interest only to alternative poly-adenylation related studies. In other studies, such regions are uninformative and if retained may only result in false overlaps. Thus as part of preprocessing, these polyA

tails are trimmed off the ends of the ESTs.

- ESTs can also sometimes contain portions of chimeric cDNA clones. Accurately detecting such artifacts is typically hard in a preprocessing step, and the task is generally deferred to a later stage of overlap detection. If the genome of the underlying organism has already been sequenced, then chimeric ESTs can be detected as those that have different portions in them aligning (through a spliced alignment method) to different genomic locations. Their detection, however, becomes much harder in the absence of the genome sequence. A common method is to flag those ESTs that “bridge” two otherwise distinct non-overlapping sets of ESTs. The problem with this approach, however, is that there could also be ESTs that genuinely bridge two ends of an mRNA transcript, and therefore this scheme could result in false labeling of such ESTs with chimeric origins. The number of ESTs in the two otherwise distinct sets of ESTs being bridged, can also serve as an additional indicator on the the confidence level of a chimeric prediction.

Natural Variations

If a pair of sequences overlap significantly but with a few mismatches and/or indels in their underlying best alignment(s), then there are two ways to explain such disagreements: (i) the underlying sequencing procedure incorrectly read the bases on one of the sequences, or (ii) the two ESTs being compared are from alleles or paralogous genes that have these natural variations because of mutations or single nucleotide polymorphisms. The choice between these two possibilities is made by looking at more than one overlapping pair at a time. For example, of the 10 overlapping sequences shown in Figure 3.2a, only one has a nucleotide that is different from the corresponding nucleotides in the other 9 sequences, indicating the high likelihood of a sequencing error that caused the variation in the singled out sequence. Figure 3.2b shows a different case where such a disagreement is equally distributed among the 10 sequences indicating that it is likely the result of a natural variation i.e., that the sequences were extracted from two different gene paralogues or polymorphic genes. The underlying assumption is that the probability of such a variation occurring at the same position evenly across multiple overlapping ESTs is too low to have likely occurred.

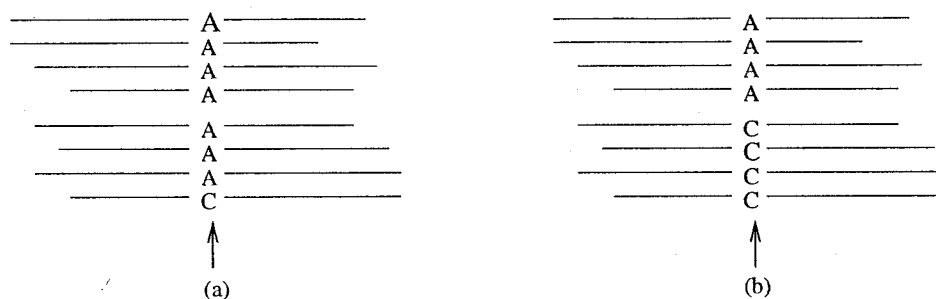


Figure 3.2 Overlap layout suggesting a case of a (a) sequencing error, and (b) natural variation.

Large data sizes

Since the initiation of cDNA sequencing projects in 1992 [Adams *et al.* (1991)], EST databases have tremendously grown in their sizes. The dbEST portion ([Boguski *et al.* (1993)], <http://www.ncbi.nlm.nih.gov/dbEST/index.html>) of the NCBI GenBank is a public repository for storing ESTs and full-length cDNAs generated by numerous sequencing efforts. As of May 2006, the dbEST database contains over 36.5 million ESTs, making it the largest public EST data repository. Also, the number of ESTs increased $\approx 29\%$ from 2004 to 2005. About 740 organisms are represented in this database, and human ESTs dominate the pool with about 7.7 million sequences, followed by mouse ESTs with 4.7 million sequences. Among plants, *Oryza sativa* (rice) has over 1.1 million ESTs, followed by *Triticum aestivum* (wheat) with 854,397 ESTs. Over 40 organisms have more than 100,000 ESTs.

3.1.2 Clustering for Genome Assembly

Once a genome is sequenced through one of the strategies discussed in Chapter 2, the set of sequenced fragments can be used to computationally “assemble” the genome. Despite rapid advances in hardware speeds and memory capacities over the last two decades, assembling tens of millions of fragments typical of large-scale genome projects places enormous computational demands. For example, one of the assembly efforts by Venter *et al.* of the ≈ 3 billion *bp* human genome took 20,000 CPU hours, a task that was brute-force parallelized to finish in 10 days

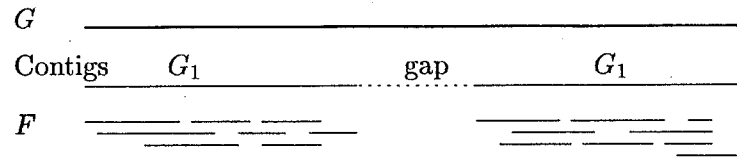


Figure 3.3 Illustration of the context of clustering in whole genome sequencing projects. Clustering F would partition it into two clusters, one corresponding to G_1 , and another to G_2 .

using ten 4-processor SMP clusters each with 4 GB RAM, along side a 16-processor NUMA machine with 64 shared memory machine [Venter *et al.* (2001b)]. A majority of the computational effort in assembling genomes is spent in detecting pairwise overlaps. For example, in the above mentioned human genome project, 10,000 CPU hours were spent only on computing pairwise alignments for detecting overlapping pairs of fragments. In this aspect, the problem of genome assembly is similar computationally to the problem of EST clustering described earlier in this section — both involve a compute-intensive overlap detection phase. However, clustering is typically an “easier” task, in that it is sufficient to form clusters based on detected overlaps, whereas, in genome assembly additional computation is required to reconstruct the supersequence(s) from the clustered sequences.

Due to this commonality in the nature of computation involved between genome assembly and sequence clustering, several genome assemblers follow a two-phase approach to genome assembly [Emrich *et al.* (2004); Havlak *et al.* (2004); Mullikin and Ning (2003)]: (i) first, “cluster” the genomic fragments based on pairwise overlap information, and (ii) assemble each of the clusters individually using traditional fragment assembly programs. The meaning of clustering in this context is, however, different from what we described for EST clustering (refer to the example in Figure 3.3): For example, let us assume a genome G , and a set of fragments F sequenced from it as shown in Figure 3.3 by one of the sequencing strategies explained in Chapter 2. Clustering F based on overlap information is expected to produce two clusters, one each for the sequence-sampled genomic stretches G_1 and G_2 . This is because of lack of overlap information to span the sequencing gap between G_1 and G_2 .

The advantage of clustering fragments prior to performing fragment assembly is that it breaks the initial problem into numerous subproblems (each corresponding to one output cluster), so that each of the subproblems can be individually tackled using a fragment assembler. In the above example, an assembly of the two clusters is expected to produce two supersequences, called *contigs* representing the contiguous stretches of genome sampled by sequencing. Therefore, a faster and more efficient clustering approach in the first phase on the entire input would subsequently allow for a thorough compute-intensive assembly phase that guarantees a highly accurate assembly.

This divide and conquer strategy of first clustering and then assembling is beneficial, however, only if the following assumptions hold:

1. There is a distinct computational advantage of using clustering ahead of running an assembly software — the clustering phase takes significantly less time and/or substantially reduced memory when compared to running the assembler on the input sequence data directly.
2. Given that each cluster is subsequently processed individually by an assembly program, the clustering phase should not separate any two sequences that may otherwise be assembled into a same contig. This is essential for the “correctness” of the final assembly, which is to ensure that the set of contigs output from a scheme that performs clustering followed by individual assembly of clusters is the same as the set of contigs produced by performing assembly directly on all the input sequences at once.
3. There are sufficient number of sequencing gaps that can then lead to substantially smaller sized problems. A perfect sequencing strategy that covers every base along a genome would produce no sequencing gaps, and therefore clustering would *not* be effective in reducing the problem complexity for the subsequent assembly step. In practice, substantial number of sequencing gaps result even if a genome is sequenced with high coverage. For example, a WGS sequencing of the human genome with a 5.11X coverage performed by Venter *et al.* produced over 100,000 sequencing gaps to be finished after scaffolding [Ven-

ter *et al.* (2001b)]. In other words, this clustering based approach to genome assembly presents a practically effective alternative.

3.1.3 Computational Challenges

While the challenges introduced by sequencing artifacts and large fragment collections are similar to those outlined for EST data in Section 3.1.1.3, there are two main challenges to performing clustering for genome assemblies:

- **Repeats:** Genome assembly is complicated by the presence of repeats. Fragments originating from different but repetitive regions of a target genome may have spurious overlaps with one another. During clustering, these spurious overlaps may cause the repetitive fragments to cluster together thereby affecting the effectiveness of clustering to break the initial problem size. Traditional methods to mask known repeats in fragments as a preprocessing step can be used to reduce the number/size of such repeat-induced clusters.
- **Uniform vs. Non-uniform sampling:** Unlike EST data, genomic fragment data generated from whole genome shotgun sequencing projects and hierarchical sequencing projects typically represent a uniform sampling over a target genome. This is because the fragments are generated with a particular coverage on the entire genome (or BAC, in case of hierarchical sequencing) specified at the time of sequencing. The implication of uniform sampling is that the number of genuine overlaps expected among fragment is linear in the size of the genome. However, this is not true with gene-enriched fragment data because the underlying sequencing selectively samples gene-rich portions of the genome, and the generated fragment data represent a non-uniform sampling over the genome. Therefore, the number of valid overlaps could be quadratic in the number of input fragments in the worst case. This makes the clustering of gene-enriched fragment data similar in complexity to the problem of clustering ESTs. None of the traditionally developed assemblers (including clustering based assemblers previously developed) are

suites for an efficient handling of gene-enriched assembly — to this end, we developed an efficient method, which will be discussed in the next chapter.

3.2 Literature Review

Numerous fragment assemblers and EST clustering methods have been developed over the past decade. In this section, we will review these methods with primary focus on their underlying algorithms to detect overlaps. Note that overlaps can be detected by the naive approach aligning every pair of sequences as discussed in Section 3.1.1.3. For convenience, we use the word “sequence” to refer to both an EST and a genomic fragment.

Among the assembler class of algorithms, we will discuss three programs, CAP3 [Huang and Madan (1999)], Phrap [Green (2003)] and TIGR Assembler [Sutton *et al.* (1995)], which are popular among EST clustering community as well [Liang *et al.* (2000)]; although in principle, any fragment assembly software can be used for clustering ESTs to the same effect. (For a detailed survey of fragment assembly and EST clustering algorithms, see [Huang (2005); Pop *et al.* (2002)] and [Kalyanaraman and Aluru (2005a)] respectively. Among the EST clustering algorithms, we will discuss UniGene [Pontius *et al.* (2003)], STACK [Christoffels *et al.* (2001); Miller *et al.* (1999)], UIcluster [Pedretti (2001)], TGICL [Perteau *et al.* (2003)] and xsact [Malde *et al.* (2003)].

PaCE [Kalyanaraman *et al.* (2003a)], which is our clustering method can be applied to cluster both EST data and genomic fragments, and will be described in Chapter 4.

3.2.1 Methods for EST Clustering and Genome Assembly

3.2.1.1 TIGR Assembler

The TIGR Assembler is one of the oldest fragment assembler programs, which has also been used in various EST clustering projects [Nelson *et al.* (1997); Rounsley *et al.* (1996); Satou *et al.* (2002); Ton *et al.* (2000)]. The algorithm is as follows: Given an input of n sequences, the overlap detection phase evaluates all $\binom{n}{2}$ pairs — for each pair, the algorithm identifies all fixed-length (≈ 10 bp) exact matches and then considers only those “promising pairs” that

have substantially long stretches of such matches for further alignment computations. From the aligned pairs, the algorithm selects only those pairs with a satisfactory sequence similarity over the overlapping regions. The clusters are then formed by initially assigning one unique cluster for every sequence (“or seeds”) that has “very small number” of overlaps and then iteratively merging clusters by considering the pairs in the decreasing order of their overlap quality. The output is a set of contigs. Storing and sorting overlaps implies a worst-case $\Theta(n^2)$ space complexity. The run-time is $\Theta(n^2)$ for evaluation of each pair of sequences plus the cost to align all the promising pairs identified by the algorithm.

3.2.1.2 Phrap

Phrap [Green (2003)] starts its overlap detection phase by building a list of sequence pairs with fixed-length matches and then sorting the list such that all matches of the same pair are consecutively placed. For each such “promising pair”, it computes an alignment band centered around the diagonal containing all matches and then computes a best alignment using a banded version of the Smith-Waterman technique [Smith and Waterman (1981)]. If there are many matches, then the band of diagonals is made wider to include all the word matches. Using only those pairs with a band score above a certain desired threshold, a layout of overlaps is then constructed and subsequently a contig is constructed from the layout using only the portions of ESTs that have a high sequence quality (or “quality value”). Because of storing and sorting pairs with fixed-length matching substrings, this algorithm has a space complexity of $O(n^2)$. Even though this is the worst-case complexity, the likelihood of such a quadratic requirement is high for EST data because of the underlying non-uniformity in sampling, as shown in Figure 3.1. The run-time complexity is worst-case quadratic and is dominated by the cost to align all the promising pairs identified by the algorithm.

3.2.1.3 CAP3

In the overlap detection phase, the CAP3 [Huang and Madan (1999)] algorithm detects pairs that show “promising” characteristics for good alignment, without having to enumerate

all pairs, as follows: concatenate all input sequences into one long string with adjacently placed strings separated by a special delimiter character. Quickly identify high scoring chains of “segment pairs” within each sequence against the concatenated string. This is implemented through a lookup table approach, similar to the method in BLAST [Altschul *et al.* (1990)]. A “segment pair” is an alignment without gaps and is initially computed by looking at all exact matches of a specified fixed-length and extending these matches as far as possible in either direction. Only those pairs that have a chain score greater than a specified threshold value are later considered for global alignment computation [Needleman and Wunsch (1970)]. The alignments are then considered in the decreasing order of their scores and an “overlap-layout” is constructed using the order and orientation of each aligning pair. In this greedy process, inconsistencies due to violating alignments can be resolved in favor of the higher scoring alignments. The final step is to compute a multiple sequence alignment from each overlap-layout component, thereby resulting in a consensus contig. The space complexity is worst-case quadratic because of storing and sorting all the promising pairs. The dominant run-time cost is that of aligning all the promising pairs.

A parallelized version of CAP3 is available. The parallel version called PCAP [Huang *et al.* (2003)] implements the serial CAP3 such that multiple independent serial jobs can be initiated simultaneously on multiple workstations of a cluster, each operating on an independent portion of the input sequence data.

3.2.1.4 UniGene

The UniGene project [Pontius *et al.* (2003)] undertaken by the NCBI is an initiative towards clustering all GenBank ESTs by organisms and by individual gene sources, i.e., ESTs from different spliced variants of the same gene are also clustered together. The UniGene clustering scheme performs incremental daily processing of ESTs submitted to the dbEST database, computed as BLAST alignments of each new EST with the contents of all individual clusters. Care is taken that each cluster contains at least one EST derived from the 3' terminus of the source mRNA transcript. This is ascertained by the presence of a polyA tail in the

corresponding EST(s) (which is not removed as part of its preprocessing step). Even though the run-time of the UniGene method is quadratic in the number of ESTs, incremental processing in batches allows for quick updating of clusters as new sequences are added to the database.

The entire UniGene cluster database is accessible online at the URL <http://www.ncbi.nlm.nih.gov/UniGene/>. As of September 2005, the database contains over 700,000 gene-oriented sequence clusters representing over 50 organisms, with the human and mouse collections leading the chart with 53,100 and 42,555 UniGene clusters respectively [Wheeler *et al.* (2005)].

3.2.1.5 STACK

STACK (Sequence Tag Alignment and Consensus Knowledgebase) [Christoffels *et al.* (2001); Miller *et al.* (1999)] is one of the first EST clustering programs and was developed to achieve tissue-specific clustering that groups ESTs by transcript source. The underlying algorithm performs simple all-versus-all pairwise comparisons with the overlap between each pair detected through a word-multiplicity measure called d^2 , a distance measure to assess sequence dissimilarities. Subsequently, the pairs with significantly small distances are used to form the clusters by an agglomerative approach called $d^2_cluster$ [Torney *et al.* (1990)], as follows: initially, each input sequence occupies a cluster of its own, and as the program progresses each significant overlap merges the corresponding clusters forming a supercluster. This mechanism achieves a transitive closure clustering, in which two entirely different sequences are brought together because of a common third sequence with which each share a good overlap. Each cluster is post-processed by the Phrap assembler to build transcript assemblies. The STACK algorithm has a run-time complexity that is proportional to the product of $\binom{n}{2}$ and the time taken to compute d^2 measure.

Because of its simplicity, the STACK algorithm is also easily parallelized [Carpenter *et al.* (2002)]. The all-pairs work is distributed evenly across processors, and the clustering results are collected and recorded serially by one processor. For an example of STACK's application, see [VanBuren *et al.* (2002)].

3.2.1.6 TGICL

The TGICL clustering software [Pertea *et al.* (2003)] was developed by TIGR. The algorithm achieves clustering by performing an all-versus-all pairwise alignment but using a greedy alignment algorithm called *megablast* [Zhang *et al.* (2000)]. The advantage of using *megablast* is that it provides a significant speedup (≈ 10 times) while aligning two highly similar sequences over its dynamic programming counterparts, although the run-time increases as the similarity decreases. Because of its simplicity, this algorithm can also be easily parallelized. Post-clustering, CAP3 is used for assembling the sequences of each cluster.

TIGR maintains a large database of clustered ESTs called the “TIGR Gene Indices”. The initiative is towards maintaining a compendium of transcriptomes of several organisms. The database has clusters built for ESTs collected from over 32 animal species including the human and mouse, and over 33 plant species including wheat and maize.

3.2.1.7 UIcluster

The UIcluster method [Pedretti (2001)] was originally developed for clustering 3' generated ESTs into 3' transcripts. The algorithm is based on the following incremental approach: Initially, each sequence is in its own cluster. At any point of execution, a list of “representative ESTs” is maintained for each cluster (typically its longest EST(s)). A global hash table is constructed by preprocessing all input ESTs, such that it indexes all fixed-length (< 16 bp) substrings within all ESTs. The ESTs are then considered one at a time. For a given EST, all clusters with at least one representative EST that has at least a specified number of fixed-length matches are identified. Alignment computations are then performed between the input EST and each of the representative ESTs identified in each cluster. The input EST (and its cluster) is then merged into one of the clusters containing the best overlapping representative, provided that best alignment(s) pass the specified similarity threshold; otherwise the clusters are left intact. The space complexity is proportional to the size of input plus the size of the global hash table. The worst case run-time complexity is $\Theta(n^2)$ multiplied by the average cost to align two sequences; for large clusters, the run-time is likely to be close to the worst-case behavior,

as all potential cluster merges are evaluated through alignments with ESTs considered one at a time.

A parallel version of the Ucluster algorithm has also been developed [Trivedi *et al.* (2002)]. The input ESTs and the initial set of clusters are evenly partitioned across processors, and each processor constructs the hash table for its local portion of the ESTs. The algorithm then performs one parallel step for each input EST, in which the sequential algorithm is run locally on each processor and the cluster to which the EST has to be merged with is decided through a collective communication at the end of the step. There are two main drawbacks with this parallel approach: (i) the number of parallel steps is proportional to the number of input ESTs, independent of the number of processors used, and (ii) the speedup achieved in each step is dictated by the processor with the most number of alignments to compute. The program is extensively used on clustering rat ESTs (<http://ratest.eng.uiowa.edu>).

3.2.1.8 *xsact*

Concurrent to our research, Malde *et al.* [Malde *et al.* (2003)] developed *xsact*, which is a serial program for EST clustering that also generates promising pairs based on maximal matches. The *xsact* algorithm first constructs a generalized suffix array on the input ESTs. This is achieved by recursively sorting prefixes, similar to the approach in [Manber and Myers (1993)] — this algorithm was developed prior to the development of linear time algorithms for directly constructing suffix arrays [Karkkainen and Sanders (2003); Kim *et al.* (2003); Ko and Aluru (2003)]. The algorithm then detects each pair with a maximal match of length l bp, l times, but reports only one instance of it to the alignment module. The pairs are generated in no particular order, and all reported pairs are aligned. Only those alignments which satisfy a specified similarity threshold are stored. The pairs are then sorted in decreasing order of their alignment scores and considered in that order for cluster merges. The space complexity of the algorithm is dominated by the number of pairs that have satisfactory alignments, which within each generated EST cluster is worst case quadratic in its number of ESTs. The run-time is dominated by the cost to align all promising pairs reported.

3.2.2 Discussion of Related Work

Given the complexities of sequence data, ensuring both high quality and high performance in a clustering method is the primary challenge faced by algorithm designers. Table 3.1 summarizes the various computational aspects of each of the EST clustering algorithms and fragment assemblers developed prior to or during this dissertation research. For comparison purposes, the table also shows the corresponding aspects of our method, PaCE. The input number of sequences is denoted by n . The average length of a sequence is assumed to be a large constant for the purpose of run-time and space complexity calculations. Both expected (“Exp.”) and worst-case (“WC”) space complexities are provided for each method. The run-time complexity is proportional to the product of the number of alignments computed and the taken to perform each alignment. The algorithm used to compute pairwise alignment is indicated against each entry; DP denotes one of dynamic programming table based methods. WGS, BAC and GE stand for fragment data generated from whole genome shotgun, BAC-by-BAC and gene-enriched sequencing projects respectively.

3.2.2.1 Run-time Concerns

Even though run-time intensive, alignment based methods provide the most accurate means to capture sequencing errors and natural variations. For this reason, methods such as TIGR assembler and UniGene perform all vs. all (i.e., $\binom{n}{2}$) pairwise sequence alignments, although as many alignments may not be necessary to arrive at the final answer. The all vs. all approach is not scalable because of a strict quadratic increase in run-time. For example, even assuming that it takes only about a microsecond to align two sequences on a GHz processor, performing all vs. all alignments for 1 million fragments will take 6 days of compute time; while for 2 million fragments it would 24 days.

One way to overcome this problem is to resort to faster methods of detecting overlaps, however, at risk of compromising on the optimality of overlap quality — e.g., STACK’s d^2 method. Alternatively, a more efficient approach to reduce the run-time is to compute alignments only for a reduced subset of the $\binom{n}{2}$ pairs, without missing any overlapping pair. A frequently used

Software	Target Application(s)	Space complexity		Alignments computed	Promising pairs scheme	Parallelism
		Exp.	WC			
TIGR assembler	Genome assembly (WGS, BAC)	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$ DP	look-up table	No
Phrap	Genome assembly (WGS, BAC)	$\Theta(n)$	$\Theta(n^2)$	$O(n^2)$ DP	look-up table	No
CAP3/PCAP	Genome assembly (WGS, BAC)	$\Theta(n)$	$\Theta(n^2)$	$O(n^2)$ DP	look-up table	limited scaling
UniGene	Incremental EST clustering	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2)$ DP	all vs. all	No
STACK	EST clustering	$O(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$ d^2 method	all vs. all	large memory SMP
TGICL	EST clustering	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$ megablast	all vs. all	large memory SMP
Ucluster	3' EST clustering	$\Theta(n)$	$\Theta(n)$	$O(n^2)$ DP	look-up table	limited scaling
xsact	EST clustering	$\Theta(n^2)$	$\Theta(n^2)$	$O(n^2)$ DP	maximal match approach	No
PaCE	EST clustering, genome assembly (GE)	$\Theta(n)$	$\Theta(n)$	$O(\#\text{promising pairs})$, DP	maximal match approach	massively parallel

Table 3.1 Summary of various previously developed fragment assemblers and EST clustering methodologies. PaCE is our methodology.

technique used to accomplish this is to preprocess the sequences before computing any alignment such that only those pairs that show a significant promise for a potential good overlap are subsequently considered for alignment computation. Detecting exact matches is often much quicker than detecting inexact matches, and therefore an exact match detection scheme can be used to identify such promising pairs. Most of the methods in Table 3.1 identify exact matches as short fixed length matches using lookup table [Aluru and Ko (2005)] approaches. The PaCE algorithm provides a more efficient alternative by identifying matches as variable length maximal matches. The difference in these methodologies will be elaborated in the next chapter.

3.2.2.2 Memory Concerns

In large-scale sequence analysis, more often than not, memory concerns pose a more serious problem than long run-times. As Table 3.1 shows, all the three fragment assembly programs have an expected linear memory requirement when applied to genome assembly. This is because of the uniform sampling that is expected in WGS and BAC sequenced data. However, if they are applied to non-uniformly sampled data (e.g., ESTs), the memory requirement grows quadratically. As for EST clustering, even the expected memory requirement is quadratic for all approaches except the incremental UniGene¹. This quadratic memory requirement is because the underlying methodologies store overlaps in memory — a strategy that is not necessary for clustering, as demonstrated by the PaCE method.

3.2.3 Performance Evaluation of Related Work

3.2.3.1 EST Clustering

For EST clustering, performance results have been published in the past by other groups working on different software programs. Carpenter *et al.* (2002) report the clustering of 15,876 human ESTs on an SGI Origin 2000 shared memory machine² using STACK's *d²_cluster*

¹The memory requirement is calculated assuming that a new increment of sequences is \ll the number of sequences already clustered; otherwise UniGene's memory requirement increases quadratically as well.

²Available memory is not specified in the paper.

Number of input sequences	TIGR Assembler	Phrap	CAP3	Number of saved pairwise overlaps (Reported by CAP3)
5,000	17	5	44	561,916
10,000	168	17	186	2,308,885
25,000	X	211	704	21,608,972
50,000	X	219	585	16,357,088
100,000	X	340	X	X
150,000	X	X	X	X
200,000	X	X	X	X

Table 3.2 (a) Run-times (in minutes) of assembly programs on an arbitrary mouse EST collection downloaded from GenBank. ‘X’ denotes that 2 GB memory was not sufficient for the program to complete. (b) Pairwise overlaps stored by CAP3.

method in 2 minutes. The paper also reports that the largest data clustered contains 1,198,607 ESTs; however, the performance results are not reported. The largest clustering reported by the TGICL program [Perteau *et al.* (2003)] was that of 1.7 million ESTs of an unspecified species in 1 hour of a PVM cluster with 20 Pentium III nodes. Overlaps are detected by the megablast program.

Of the fragment assemblers, the TIGR assembler, Phrap and CAP3 programs are popular for EST clustering as well [Liang *et al.* (2000)]. We evaluated these three serial programs using a single Intel Xeon CPU 3.06GHz processor of an IBM xSeries node, each with access to 2 GB RAM. Table 3.2 shows the results on various subsets of a mouse EST collection downloaded from GenBank and containing 200,000 ESTs. The table also shows the number of valid overlaps detected and saved by the CAP3 program for each of the input subsets. As can be seen, the number of overlaps increases quadratically from the 5,000 data point to 10,000, and rather disproportionately for 25,000 and 50,000 ESTs. This is because several overlaps are screened out by the program as “false” candidates due to chimeric ESTs in the input. The non-uniform increase in the run-time with input size is consistent with the non-uniformity in the detected overlaps.

The non-uniformity in the expected run-times and overlap information depends on the EST

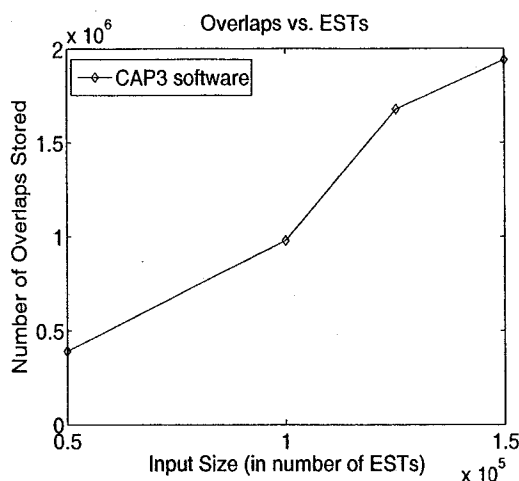


Figure 3.4 Number of overlaps stored by the CAP3 program while clustering different subsets of a rat EST data set. The peak memory usage reached 2 GB for 150,000 ESTs.

data. To further understand this, measured the number of overlaps detected by CAP3 on a different input data — an arbitrary collection of 150,000 rat ESTs downloaded from GenBank. The results plotted in Figure 3.4 indicate a quadratic increase in overlaps for this data, and also that CAP3 could complete for a bigger data (up to 150,000) with an available 2 GB RAM, without running out of memory. Both these observations confirm the strict input data dependency of the underlying problem.

3.2.3.2 Genome Assembly

Several whole genome projects have been conducted in the past, with one of the largest genomes being that of the human. The Celera genomics assembly team estimated that it would take tens of thousands of CPU hours and approximately 600 GB of memory to assemble the ≈ 3 billion *bp* genome based on their previous fruitfly assembly [Myers *et al.* (2000)]. To meet these high computational demands, Celera used ten 4-processor SMP clusters with 4 GB memory each, along side a 16-processor NUMA machine with 64 GB shared memory, and engineered an incremental approach that reduced the peak memory usage to 28 GB. The assembly took 20,000 CPU hours on 27.27 million WGS fragments.

[Huang *et al.* (2003)] report the assembly of mouse genome using PCAP, which is the

Number of processors	4	8	16	32	64
Run-time in minutes	21	17	14	81	X

Table 3.3 Run-time scaling of PCAP on 322,000 gene-enriched maize fragments. "X" denotes that the program was hung performing I/O operation.

parallel version of CAP3 assembler. The input comprised of 33 million mouse WGS fragments. The compute platform included 20 Compaq ES40 servers, each with 4 processors and 4 GB RAM, along side another Compaq ES40 server with 16 GB RAM. Also provided is a 32 GB shared file system and a 17 GB scratch space on each server. Multiple alignment jobs were launched simultaneously on each processor, and the implementation does not support any interprocessor communication; instead, data sharing is through the file system, making the program I/O intensive. The assembly was completed in 7 days (on 80 processors) and the number of overlaps was only 273 million, confirming the expected linear complexity with uniformly sampled WGS data.

To study the effect of I/O on scaling using a commodity cluster, we ran CAP3 on $\approx 322,000$ maize gene-enriched sequence data of total length 250 million *bp*. The platform used was an IBM xSeries cluster with Intel Xeon processors. The run-time scaling is reported in Table 3.3. As the table shows, the run-time actually increases after 16 processors, and on 64 processors, the program failed to respond.

3.2.4 Need for Scalable High-performance Computing Methods

Given the obvious limitations in compute power and memory capacities of a serial computer, several EST clustering and genome assembly programs have adopted parallelism as a means to increase available memory and achieve additional speedup. All these parallel approaches are direct extensions of their corresponding serial counterparts, implying a strong coupling among the parallel jobs due to high data inter-dependencies. As a result, the projects involving these methods resort to using high-end workstations with large shared memory so that the entire

data can be made available through the shared memory to all the processors. If the entire data does not fit in one shared memory machine, then multiple machines are used, different portions of the data are loaded from the file system into a local shared memory, and the processing is performed in batches.

The rudimentary nature of these parallel schemes results in poor scalability, I/O-intensive computation and very long run-times. Moreover, the inherent sequential approach within these algorithms limits the speedups achievable from these parallel systems. In addition, the shared memory architecture itself imposes a limitation on the number of processors — only a few tens to just over a hundred processors are available in state-of-the-art shared memory architectures. In contrast, several hundred gigabytes to even terabytes of memory is easily available in state-of-the-art distributed memory architectures — e.g., the IBM BlueGene/L architecture supports thousands to even tens of thousands of processors with an aggregate distributed memory of several terabytes available through a fast interconnection network. Such systems could serve as ideal platforms for performing large-scale sequence analysis, providing both an order of magnitude speedup and capability to scale up to much larger data sets. The main challenge, however, is to have an inherently parallel algorithm that can efficiently exploit the high compute power and memory capacities.

CHAPTER 4. A SCALABLE PARALLEL CLUSTERING FRAMEWORK FOR LARGE-SCALE SEQUENCE ANALYSIS

In this dissertation, we present the design, development and application of a scalable parallel algorithm and software for performing DNA sequence clustering. A preliminary version of this method was developed as part of my M.S. thesis research [Kalyanaraman (2002)].

4.1 The Sequence Clustering Problem

Problem Statement: Let $S = \{s_1, s_2, \dots, s_n\}$ denote the set of n input sequences over an alphabet Σ . Two sequences $s_i, s_j \in S$ are said to be *related* if either s_i and s_j show a “significant” overlap, or $\exists s_k \in S$ to which both s_i and s_j are *related*. The problem of sequence clustering is to partition S such that $\forall s_i, s_j \in S$, s_i and s_j are in the same subset (or “cluster”) if and only if s_i and s_j are related.

For generality, let us not assume anything on the type of DNA sequence data to be clustered — they can be ESTs, cDNAs, or genomic fragments, or any other type of biological sequence that can be computationally represented as a string over the DNA alphabet. Also, the above formulation is generic enough to accommodate any preferred alignment method. For instance, in the context of clustering for genome assembly, two sequences sharing a good suffix-prefix alignment are potential candidates to be genomic neighbors, and can therefore be considered to have a significant overlap. In the context of EST clustering, if the underlying objective is to cluster together sequences derived from the same gene, then overlaps can be detected as a chain of local alignments. Without loss of generality, we will henceforth assume that the choice of overlap detection method is suffix-prefix alignment computation.

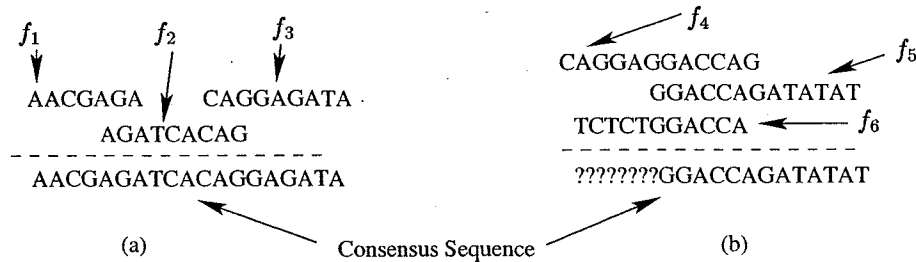


Figure 4.1 Examples to show the effect of transitive closure clustering in the context of genome assembly.

The above formulation of clustering is also sometimes referred to as *transitive closure clustering* because the definition of relationship between sequences is transitive in nature. Thus it is possible to have two entirely distinct sequences in the same cluster simply because there is a third sequence to which both are related. An example in the context of genome assembly in which three fragments are clustered together based on suffix-prefix alignments is shown in Figure 4.1a. Note that the clustering is effected regardless of the existence of an overlap between f_1 and f_3 . This formulation does not guarantee that the sequences in the same cluster conform to a consistent overlap layout. An illustration of an inconsistent layout shown by sequences in the same cluster is illustrated in Figure 4.1b. The idea is to defer the task of resolving such inconsistencies to later stages post-clustering, and instead primarily focus on breaking down the initial problem size through clustering.

In what follows, we will explain our parallel clustering algorithm. For ease of exposition, we first describe our serial algorithm, and later describe its parallelization.

4.2 A Serial Clustering Algorithm

Given that overlaps constitute the primary basis of clustering, a simple approach to cluster n sequences, each of length l bp on an average, is by evaluating all pairs of sequences for potential overlaps. Figure 4.2 outlines this approach. The “Find” operation on a sequence returns its current cluster, and the “Merge” operation on two clusters performs a union of the two clusters. This simple approach to clustering has the following advantage: even though

Algorithm 1 *A Naive Algorithm*

Input: Set $S = \{s_1, s_2, \dots, s_n\}$ of n sequences
Output: A partition $C = \{C_1, C_2, \dots, C_m\}$ of S , $1 \leq m \leq n$

1. Initialize Clusters:
 $C \leftarrow \emptyset$
 $\forall 1 \leq i \leq n, C_i \leftarrow \{s_i\}, C \leftarrow C \cup C_i$
2. FOR $\forall (s_i, s_j)$ DO
score $\leftarrow \text{Align}(s_i, s_j)$
IF score is significant THEN
 $C_p \leftarrow \text{Find}(s_i)$
 $C_q \leftarrow \text{Find}(s_j)$
Merge(C_p, C_q)
3. Output C

Figure 4.2 A naive serial clustering algorithm. The worst-case run-time and space complexities of the algorithm are $\Theta(n^2 \times l^2)$ and $O(n \times l)$, respectively.

Step 2 loops $\binom{n}{2}$ times, the overall number of “Merge” operations is limited to at most $n - 1$, regardless of the nature of sequence data. Each merge corresponds to an overlapping pair of sequences, although the converse need not be true. However, it is not possible to enumerate these pairs directly.

Step 1 takes $O(n)$ run-time. Each of the $\binom{n}{2}$ loops in Step 2 computes an alignment that costs $O(l^2)$. The “Merge” operation can be implemented as a simple set union operation that takes $O(n)$ time, and the “Find” operation can be implemented to run in $O(1)$ time through an array based implementation. The overall run-time complexity is $O(n^2 \times l^2) + O(n^2)$ ($= O(n^2 \times l^2)$), and the space complexity is $O(n \times l)$.

Observation 1 *At any stage of Algorithm 1, the set of clusters C represents a partition of the input set S . This implies that the “Merge” and “Find” operations are disjoint set operations.*

This observation can be exploited by implementing the set of clusters as a union-find data structure [Tarjan (1975)]. This enables each “Merge” and “Find” operations to be performed in time proportional to the inverse of Ackerman’s function, which is a very small constant for

all practical purposes.

4.2.1 Reducing the Number of Pairs Aligned

4.2.1.1 Promising Pairs

Algorithm 1 computes $\Theta(n^2)$ alignments. One way to reduce the number of the alignments computed without affecting the quality of clustering is to take advantage of the low frequency of sequencing errors and natural variations expected in sequence data. Because of low error rates, sequences that show significant overlaps are expected to contain long exact matches, while the converse is not necessarily true. This observation is exploited by several previously developed methods in the following manner: restrict alignment computations to only those pairs that contain exact matches of a specified fixed-length w . The underlying algorithms identify such pairs using a lookup table to index all w -length substrings within each input sequence [Aluru and Ko (2005)]. In practice, the value of w is limited to just over 10, even though low error rates may allow for higher values. This is because the lookup table's size is exponential in w . For example, a value of 12 implies $4^{12} = 16$ million entries to store (for DNA alphabet); while an expected error rate of 2% over a 100 bp long aligning region allows a value up to 33. Another downside to this fixed-length exact match based approach is that a long exact match of length l will reveal itself as $(l - w + 1)$ consecutive w -length matches.

To overcome the above limitations, we define a *promising pair* as follows:

Definition 1 *A maximal match between a pair of sequences is an exact match that cannot be extended on either side to result in a longer match.*

Definition 2 *A promising pair is a pair of sequences that has a maximal match of length at least w .*

The value of w can be calculated as follows: if ϵ denotes the expected sequence error rate ($0 \leq \epsilon \leq 1$), then for two sequences to align over a length l_a , it is necessary (but not sufficient) that the aligning region contains an exact match of length at least $w = \lfloor \frac{l_a}{\epsilon \times l_a + 1} \rfloor$. In

Section 4.2.2, we describe our algorithm to generate promising pairs in amortized $O(1)$ time per pair.

4.2.1.2 Clustering Heuristic

Another independent way to reduce the number of pairs aligned from $\binom{n}{2}$ is by taking advantage of the following observation.

Observation 2 *Once a pair of clusters have been merged, it is no longer necessary to evaluate any two sequences originating from the merged cluster, as the result has no further effect on clustering.*

This implies that it is sufficient to compute alignments only for pairs that have sequences in different clusters. If both sequences of a pair are in the same cluster, then they are not aligned, thereby resulting in run-time savings. If the sequences are in two different clusters, an optimal alignment is computed. If the resultant alignment quality satisfies the specified overlap criteria, then the clusters containing the two sequences are merged as in Algorithm 1; otherwise, the clusters are left intact and the alignment effort is wasted.

While the above technique has the potential to reduce the number of pairwise alignments computed, it does not guarantee the same. In the worst-case event of no overlapping pairs of sequences in an input, this scheme still evaluates all $\binom{n}{2}$ pairs. For this reason, this improvement is only a heuristic; henceforth, we will refer to it as the *clustering heuristic*. On a similar note, generating promising pairs is also only a heuristic; in the worst-case, all sequences can share an exact match of length w , while no sequence overlaps with any other sequence.

4.2.1.3 Pair Generation Heuristic

Run-time savings achieved using the promising pair heuristic is data dependent — the presence or absence of sequence pairs with sufficiently long maximal match is a property of the input data. In case of the clustering heuristic, however, this dependency is only part of it. A more important factor that dictates the number of pairs aligned is the order in which the promising pairs are processed. A pair that passes the overlap test leads to merging of clusters,

thereby obviating the need to perform any further alignments for pairs that are part of the same cluster. While such pairs cannot be predicted prior to their alignment evaluations, their early identification causes clusters to merge sooner, which in turn leads to potential savings in run-time. This hypothesis forms the basis of our *pair generation heuristic*, which is basically a greedy mechanism to maximize the run-time savings achievable from the combination of the promising pair and clustering heuristics.

The pair generation heuristic is as follows: Instead of generating promising pairs in an arbitrary order and considering them for potential overlaps in that order, generate and consider them in non-increasing order of their maximal match lengths — longer a maximal match between two sequences, higher the likelihood of the pair succeeding the overlap test. Therefore, evaluating pairs in non-increasing (“decreasing” for ease of exposition) order of their maximal match lengths is expected to result in early cluster merges, potentially reducing subsequent alignment computation. Note that the heuristic also requires that the promising pairs be not just considered but also *generated in decreasing order* of maximal match lengths — generating all promising pairs and later sorting them would involve storing all promising pairs, which could be quadratic in the worst case. For this reason, we developed an “on-demand” promising pair generation algorithm that does not necessitates storing of pairs. The overall pair generation algorithm is space optimal and will be described Section 4.2.2. Taking into account all the heuristics described so far, Algorithm 1 can be improved as shown in Figure 4.3.

4.2.2 An Optimal Algorithm for On-demand Generation of Promising Pairs

Ideally, each promising pair should be generated only once. But a given pair of strings may have multiple distinct maximal matches, or a given match could be maximal in multiple pairs of locations between the same two strings. See Figure 4.4 for an illustration. One way to avoid generating multiple copies of the same pair in such cases is to record a pair the first time it gets generated and later discard any future generation of the same pair. This simple scheme, however, requires storing all generated pairs, potentially requiring $O(n^2)$ memory. As a compromise, the algorithm described below operates in linear space and generates each pair

Algorithm 2 *Algorithm 1 with Promising Pairs, Clustering and Pair Generation Heuristics***Input:** Set $S = \{s_1, s_2, \dots, s_n\}$ of n sequences**Output:** A partition $C = \{C_1, C_2, \dots, C_m\}$ of S , $1 \leq m \leq n$

1. Initialize Clusters:

 $C \leftarrow \emptyset$ $\forall 1 \leq i \leq n, C_i \leftarrow \{s_i\}, C \leftarrow C \cup C_i$

2. REPEAT

 $(s_i, s_j) \leftarrow$ generate next promising pair in decreasing order of maximal match length $C_p \leftarrow Find(s_i)$ $C_q \leftarrow Find(s_j)$ IF $C_p \neq C_q$ THEN score $\leftarrow Align(s_i, s_j)$

IF score is significant THEN

 $Union(C_p, C_q)$

UNTIL no more promising pair to generate

4. Output C

Figure 4.3 Algorithm 1 improved by the promising pairs, clustering and pair generation heuristics.

at least once and at most as many times as the number of distinct maximal matches in it. For example in Figure 4.4, the algorithm will generate (s_1, s_2) exactly once, while (s_3, s_4) is generated at least once and at most twice.

4.2.2.1 Notation

Without loss of generality, assume that S is a set of DNA sequences over the alphabet $\Sigma = \{A, C, G, T\}$. For a string s : let $s[i]$ denote the character at position i ; $s(i)$ denote the suffix starting at i ; and $|s|$ denote the length of s . Let $N = \sum_{i=1}^n |s_i|$; i.e., $l = \frac{N}{n}$. A match α between two strings is said to be *left-maximal* (alternatively, *right-maximal*) if the characters that immediately precede (alternatively, follow) α in the two strings are different or if α is a prefix (alternatively, a suffix) of either string. Thus α is a maximal match if it is both left- and right-maximal.

By definition, a *suffix tree* of a string is a rooted compacted trie of all its suffixes [Weiner (1973)]. A *generalized suffix tree (GST)* of a set of strings is a rooted compacted trie of all

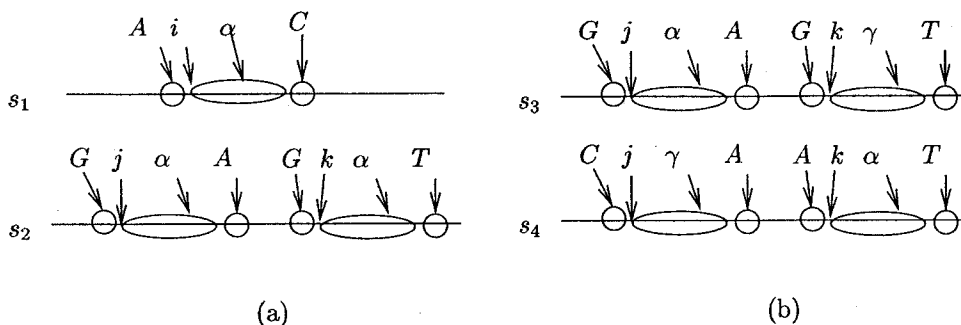


Figure 4.4 Examples showing two cases of maximal matches. (a) A match α is maximal in two pairs of locations (i,j) and (i,k) between s_1 and s_2 . (b) Two maximal matches α and γ exist between s_3 and s_4 .

suffixes all strings [Gusfield (1997a)].

Let G denote the GST of all strings in S . A special terminal character ‘\$’ is appended to each input string in S to ensure there exists a leaf node for every suffix of each string. Let the *path-label* of a node u be the string obtained by concatenating all edge labels from the root to u ; if u is a leaf node, the terminal character ‘\$’ is excluded in its path-label. Let the *string-depth* of a node u denote the length of its path-label.

4.2.2.2 The Algorithm

The GST G for S is first constructed using a linear time algorithm [Ukkonen (1995); Weiner (1973); McCreight (1976)]. The nodes in G with string-depth $\geq w$ are then sorted in decreasing order of string-depth. Because string-depth of any node in a GST is bounded by the length of the longest string in S , radix sorting is used to run in linear time.

The main idea behind our pair generation algorithm is the following: Sequences s_i and s_j share a maximal match α if and only if

- C1. $\exists u$ such that $path-label(u) = \alpha$.
- C2. $\exists k$ and l such that $f_i(k)$ and $f_j(l)$ are in subtree rooted at u .

- C3. (right maximality) If u is not a leaf, $f_i(k)$ and $f_j(l)$ are in subtrees of different children of u .
- C4. (left maximality) If $k \neq 1$ and $l \neq 1$, $f_i[k-1] \neq f_j[l-1]$.

Maximal matches can be generated by considering each node in the GST and identifying pairs of suffixes in the node's subtree that satisfy C3 and C4. To generate maximal matches in decreasing length order, we sort the nodes in GST in decreasing order of the lengths of their path-labels using radix sort, and process them in that order. Instead of checking C3 and C4 for each pair, we generate maximal matches in amortized $O(1)$ time per pair as follows: For node u and $c \in \Sigma$, let $\ell_c(u) = \{s_i(j) \mid s_i(j) \text{ is in subtree of } u; j > 1; s_i[j-1] = c\}$, and $\ell_\lambda(u) = \{s_i(1) \mid s_i(1) \text{ is in subtree of } u\}$. These are collectively known as *lsets* at u . The *lsets* at leaves are computed directly. For an internal node u and $c \in \Sigma \cup \{\lambda\}$, $\ell_c(u) = \bigcup_{u'} \ell_c(u')$ over all children u' of u . The *lsets* are maintained as linked lists to allow constant time union operations.

Consider pair generation at internal node u corresponding to $path-label(u)$ as the maximal match. At this stage, pair generation at u 's children would have been completed and their *lsets* are known. The set of pairs at u are obtained by computing $\bigcup \ell_c(u') \times \ell_{c'}(u'')$, where u' and u'' are two different children of u (to satisfy C3), and $c \neq c'$ or $c = c' = \lambda$ (to satisfy C4). After pair generation at u is finished, its *lsets* are computed from the *lsets* of its children. At a leaf u , right maximality is automatically satisfied. Hence, pairs are generated as $\bigcup \ell_c(u) \times \ell_{c'}(u)$, where $c \neq c'$ or $c = c' = \lambda$.

Note that the above scheme generates pairs in the form $(s_i(j), s_{i'}(j'))$ instead of $(s_i, s_{i'})$. This is needed if pairwise alignment computations are anchored to the maximal matches. If arbitrary suffix prefix alignments are computed, then it is wasteful to generate the same pair multiple times. In such a case, the above algorithm can be modified to reduce the number of duplicate generations of the same sequence pair, while still guaranteeing $O(1)$ generation time per pair. This improvement is explained below.

Instead of partitioning the suffixes in a node's subtree into its *lsets*, we now partition the strings represented in a node's subtree. If multiple suffixes from a string are present in the

subtree, an arbitrary suffix is chosen, and the string is placed into the *lset* corresponding to the character in the string that precedes this suffix. Formally, $s_i \in \ell_c(u)$ iff the suffix $s_i(j)$ exists in subtree of u , either $j = 1$ and $c = \lambda$ or $s_i[j - 1] = c$, and s_i is not part of any other *lset* at u . The partitioning of strings represented in the subtree of a node u may no longer be unique; however, any partitioning suffices.

Before generating pairs from an internal node u , the *lsets* at u 's children are traversed to detect and remove duplicate occurrences of any string. After this duplicate elimination process, the pair generation algorithm is run as before.

To understand the logic behind this duplicate elimination method, note that duplicates of a given pair (s_i, s_j) implies one of following cases:

1. if a maximal match occurs as multiple substrings in at least one of the strings. An example of the latter case is shown in Figure 4.4;
2. if the two sequences share more than one maximal match.

In the first case, we can expect the string with multiple occurrences of the maximal match substring to be represented in more than one *lset* of u 's children. The above mechanism will detect and eliminate these duplicates. It cannot, however, guarantee the detection of duplicates arising due to the second case because such sequence pairs may get formed under nodes that do not share an ancestor-descendant relationship.

The algorithms for generating pairs from leaf and internal nodes are given in Figure 4.5. Traversing *lsets* of all child nodes to eliminate multiple occurrences of a string can be implemented to run in time proportional to the sum of the cardinalities of those *lsets*. A global array $M[1 \dots n]$, one entry for each input string, is maintained. Let u be an internal node currently being processed. The first time a string s_i is encountered, $M[i]$ is marked with u 's identifier. Any future occurrence of s_i under any of u 's child nodes is detected as a duplicate occurrence by directly checking $M[i]$. A linked list implementation of the *lsets* allows the union in *step 3* of *GeneratePairsFromInternalNode* to be computed using $O(|\Sigma|^2)$ concatenation operations. This restricts the overall space required to store *lsets* to $O(N)$. The assumed

Algorithm 3 *Pair Generation from a GST based on Maximal Matches***GeneratePairsFromLeaf(Leaf Node: u)**

1. Compute the *lsets* at u by scanning its labels.
2. Compute:

$$P_u = \bigcup_{(c_i, c_j)} \ell_{c_i}(u) \times \ell_{c_j}(u), \forall (c_i, c_j) \text{ s.t., } c_i < c_j \text{ or } c_i = c_j = \lambda$$

GeneratePairsFromInternalNode(Internal Node: u)

1. Traverse all *lsets* of all children u_1, u_2, \dots, u_q of u .
IF a string is present in more than one *lset* THEN
all but one occurrence of it are removed.
2. Compute:

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

3. Create all *lsets* at u by computing :

$$\text{FOR } \forall c_i \in \Sigma \cup \{\lambda\} \text{ DO} \\ \ell_{c_i}(u) = \bigcup_{u_k} \ell_{c_i}(u_k), 1 \leq k \leq q$$

Figure 4.5 Algorithm for generating promising pairs from a generalized suffix tree.

arbitrary orderings of the characters in $\Sigma \cup \{\lambda\}$ and the child nodes are to limit generating a pair at u to one of its forms: (s, s') and (s', s) .

In summary, the sets of sequence pairs generated at an arbitrary leaf node u and an arbitrary internal node v are given by:

$$P_u = \{(s, s') \mid s \in \ell_{c_i}(u), s' \in \ell_{c_j}(u), c_i, c_j \in \Sigma \cup \{\lambda\}, ((c_i < c_j) \vee (c_i = c_j = \lambda))\}$$

$$P_v = \{(s, s') \mid s \in \ell_{c_i}(v_k), s' \in \ell_{c_j}(v_l), c_i, c_j \in \Sigma \cup \{\lambda\}, k < l, ((c_i \neq c_j) \vee (c_i = c_j = \lambda))\}$$

The overall clustering algorithm can be divided into a preprocessing phase followed by a clustering phase, as shown in Figure 4.6. In the preprocessing phase, the GST for all n input sequences is constructed and its nodes are sorted based on their string-depths. The clustering phase is responsible for pair generation, alignment computation and management of clusters.

The following lemmas prove the correctness and run-time characteristics of the algorithm:

Algorithm 4 *The Sequence Clustering Algorithm***Input:** Set $S = \{s_1, s_2, \dots, s_n\}$ of n sequences**Output:** A partition $C = \{C_1, C_2, \dots, C_m\}$ of S , $1 \leq m \leq n$

1. Initialize Clusters:

 $C \leftarrow \emptyset$ $\forall 1 \leq i \leq n, C_i \leftarrow \{s_i\}, C \leftarrow C \cup C_i$ 2. $G \leftarrow$ Construct the Generalized Suffix Tree of S 3. Radix sort nodes in G with string-depth $\geq w$ in decreasing order of string-depth.4. FOR each u in the sorted order DO

REPEAT

 $(s_i, s_j) \leftarrow$ generate next promising pair from u $C_p \leftarrow Find(s_i)$ $C_q \leftarrow Find(s_j)$ IF $C_p \neq C_q$ THEN score $\leftarrow Align(s_i, s_j)$

IF score is significant THEN

 $Union(C_p, C_q)$ UNTIL no more pairs to generate from u 4. Output C

Figure 4.6 Our sequence clustering algorithm. Steps 1 and 2 are collectively called the “preprocessing phase” and the remainder of the algorithm is called “clustering phase”.

Lemma 1 *Let u be a node with path-label α . A pair (s, s') is generated at u only if α is a maximal match between s and s' .*

Proof: At a leaf node u , all pairs of strings represented in its *lsets* are automatically right-maximal by definition. If the algorithm generates a pair (s, s') at u , it is because the strings are either from *lsets* representing different characters or from the *lset* representing λ . In either case, α is a maximal match between s and s' . For an internal node u , the algorithm generates a pair (s, s') only if (i) s and s' are from *lsets* either representing different characters or λ , and (ii) s and s' are from *lsets* of two different children of u . The former ensures α is left-maximal; the latter ensures α is right-maximal. Thus α is a maximal match of s and s' . ■

Corollary 1 *The number of times a pair is generated is at most the number of distinct max-*

imal match substrings of the pair.

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

Note that the converse of Lemma 1 need not necessarily hold; i.e., due to duplicate elimination, some of the maximal matches between a pair may go undetected. This could, however, happen only if the same pair was generated elsewhere because of a longer maximal match. In other words, it is guaranteed that the algorithm guarantees each promising pair at least once, as proved below.

Lemma 2 *A pair (s, s') is generated at least once if it is a promising pair.*

Proof: Consider α , a largest maximal match of length $\geq w$ between strings s and s' . This implies that there exists either a leaf or an internal node u with path-label α . Also \exists suffixes $s(i)$ and $s'(i')$ represented in u 's subtree that share a common prefix α . Thus if u is a leaf node, then $s \in \ell_{c_1}(u)$ and $s' \in \ell_{c_2}(u)$ such that $c_1 \neq c_2$ or $c_1 = c_2 = \lambda$, implying that the algorithm will generate the pair at u . If u is an internal node, then the fact that α is a largest maximal match ensures that s and s' will occur, in *lsets* of different children, even after the duplicate elimination process at u ; these *lsets* will correspond either to different characters or to λ . Thus the algorithm will generate the pair at u . ■

Lemma 3 *The algorithm runs in time proportional to the number of pairs generated plus $O(N)$. The space complexity of the algorithm is $O(N)$.*

Proof: Each node at string-depth $\geq w$ is processed exactly once. At an internal node, the duplicate elimination process reduces the total size of *lsets* of all its children by at most a factor of $(|\Sigma| + 1)$. This is because a string is present in at most one *lset* of each child node and the number of children is bounded by $(|\Sigma| + 1)$. The total size of all the *lsets* of all the

children after duplicate elimination is bounded by the number of pairs generated at the node. Taken together, this implies that the cost of the elimination process is bounded by a constant multiple of the number of pairs generated at the node (assuming $|\Sigma|$ is a constant).

The space complexity of the GST data structure is $O(N)$. The space required by *lsets* is proportional to the total number of *lset* entries to be stored at all the leaf nodes, which is $O(N)$. This is because *lsets* at internal nodes are constructed from *lsets* of their children and so do not require additional space. ■

Asymptotically, the run-time is likely to be dominated by the time spent in computing alignments, a fact that is corroborated by our experiments on large collections of genomic fragments and ESTs (see Section 4.4). The alignment computation run-time can be reduced by using the maximal match information that caused a pair to be generated to “anchor” its alignment as shown in Figure 4.7. By anchoring, it is only required to compute alignment over the two flanking extensions, thereby saving the alignment run-time. Further savings can be achieved by extending this idea to include multiple maximal matches as part of the same anchor, and in addition computing alignment over a band of diagonals [Fickett (1984)] within each area of the table not covered by the anchored maximal matches. Anchoring may, however, produce a sub-optimal alignment, as it is possible that none of the optimal alignments contains an anchored maximal match.

In practice, partial clustering information may be available through alternative means for a subset of input sequences prior to clustering. For example, it may be known that two sequences were derived from ends of the same clone. Such “mate” information can be incorporated into the clustering algorithm by initializing the clusters such that all mates are already clustered.

4.2.2.3 Space requirement

While the space complexity is $O(N)$, the constant of proportionality is that of what is required to store the GST and the *lsets*. Since there are at most N leaf nodes in the GST, the total number of nodes is limited to $2 \times N - 1$. Because the above algorithm does a bottom-up traversal of the tree, in which a parent is visited only after all its children are visited, the tree

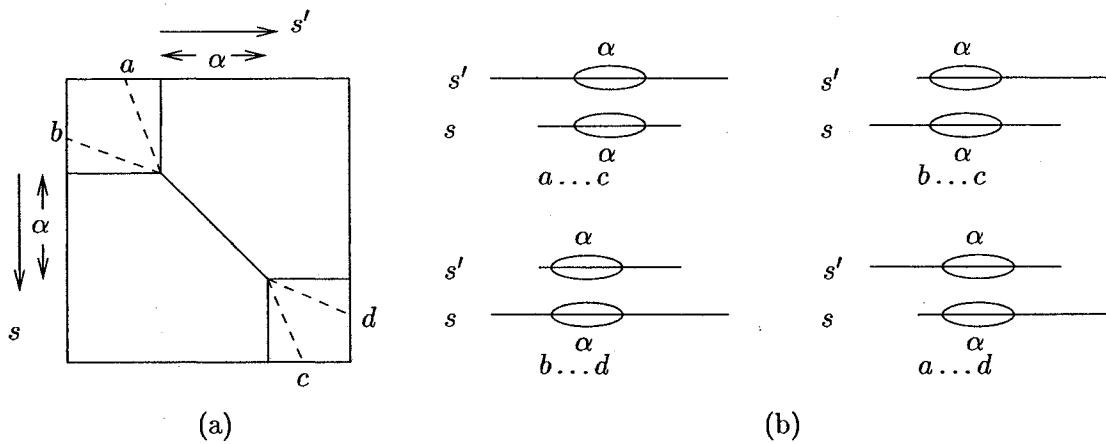


Figure 4.7 (a) Dynamic programming table showing the computation of an alignment between s and s' anchored on a maximal match α . (b) Overlap patterns resulting from suffix-prefix alignment computation and their corresponding paths in the table.

can be implemented as follows: The nodes are stored in an array in the depth-first traversal order. Each node in the tree stores its string-depth, a pointer to its *lsets* and a pointer to the rightmost leaf in its subtree. A leaf node's pointer points to itself. Given an internal node, all its children can be accessed as follows: The node immediately next to it in the array is its leftmost child. Its right sibling can be obtained by tracing the array entry next to its rightmost pointer entry. If a node's rightmost pointer points to the same as its parent's, then it is the rightmost child of its parent. The *lsets* need N entries, one for each suffix in the input. An additional array of at most $2 \times N - 1$ entries is required to store the node identifiers in sorted order of their string-depths.

Our implementation meeting the above storage requirements has a worst case constant of ≈ 40 bytes for every input character. Because DNA sequences are double stranded, a sequence should be considered both in its forward and reverse complemented form for overlaps. This doubles the constant to ≈ 80 bytes for every input base. As an example, on a set of whole genome shotgun sequences that are extracted with an 8x coverage over 1 *Mbp* long genomic stretch (i.e., for an input size of 8 megabases), this implementation requires 640 MB in the

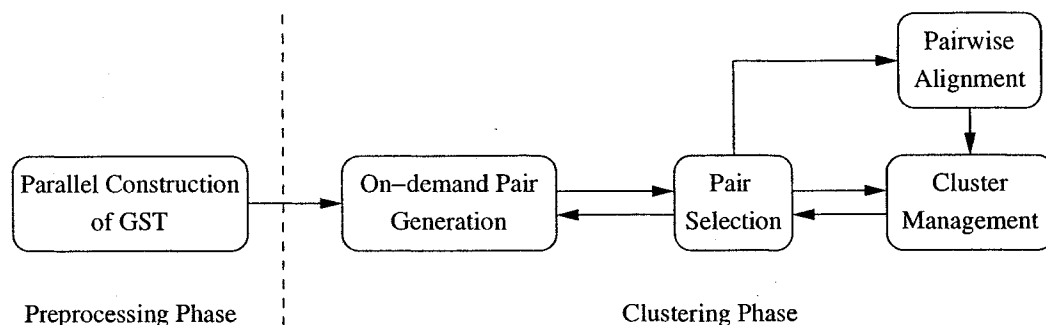


Figure 4.8 Organization of the PaCE software.

worst case. In comparison, this is expected to consume 1 GB in implementations of previously developed assemblers [Pop *et al.* (2002)].

4.3 A Space and Time Efficient Parallel Clustering Algorithm

In this section, we describe our parallel algorithm, *PaCE*, for clustering DNA sequences. The algorithm has two main phases: (i) a *preprocessing phase* to construct a distributed representation of the GST on the input sequences, and (ii) a *clustering phase* to generate promising pairs, detect overlaps and perform clustering in parallel. The organization of the PaCE software and the interactions among its components are depicted in Figure 4.8. The design of the parallel algorithm follows a single master-multiple workers paradigm. The GST construction involves only the worker processors.

4.3.1 Parallel Generalized Suffix Tree Construction

Serial construction of suffix trees is a well-studied problem with many linear-time construction algorithms [Gusfield (1997a)]. There are algorithms for constructing suffix trees in parallel under CREW/CRCW PRAM models of computing [Apostolico *et al.* (1988); Hariharan (1997)]. However, due to the unrealistic assumptions underlying the PRAM model, a direct implementation of these algorithms is not practically useful. We developed the following practically efficient algorithm, suited to exploit the distributed memory machine model. Our algorithm constructs a distributed representation of GST in parallel over the set S of all input

sequences and their reverse complements. Recall that the sum of lengths all sequences in S is denoted by N , and the average length of an input sequence is denoted by l . Let p denote the number of worker processors. Also, recall that the cutoff length for maximal matches in promising pairs is denoted by w .

In the first step, given an integer parameter k , all suffixes are sorted based on their k -length prefixes. The value of k is chosen large enough to result in $\approx \frac{N}{p}$ suffixes per processor after sorting. Also, $k \leq w$, to suit purpose of the GST, which is identifying maximal matches of length at least w . Empirically, a value of 11 was found appropriate for genomic data and many EST data, for the range of processors tested (up to 1,024 processors). Sorting is achieved as follows: The set S is initially partitioned such that each processor gets $\approx \frac{N}{p}$ nucleotides. Through a linear scan, each processor partitions the suffixes of the local sequences into Σ^k buckets based on their first k characters. The suffixes are then globally redistributed such that those belonging to the same bucket are in the same processor, and the number of suffixes per processor is $\approx \frac{N}{p}$.

Because the buckets correspond to the set of suffixes sorted based on their k -length prefixes, building one subtree for each bucket would construct the GST without the top portion with path-label length $< k$. We perform a depth-first construction of each subtree by partitioning the suffixes in its bucket into Σ sub-buckets based on their $(k+1)^{th}$ character, and recursively subdividing each sub-bucket similarly until all suffixes separate or their lengths exhausted. At worst-case, this procedure visits all suffixes to their full lengths, implying a run-time of $O(\frac{N \times l}{p})$.

In the above approach, not all sequences that have suffixes in a local bucket may be available in a processor's local memory before construction of the corresponding subtree. This is because the initial sorting based on k -length prefixes may assign suffixes of sequences in different processors to local buckets. The main challenge is therefore to ensure availability of all required sequences needed to construct the local subtrees. Storing all sequences with suffixes in all local buckets requires $\min\{\frac{N \times l}{p}, N\}$ space in the worst case, which is not a viable solution. Our first solution was to construct one subtree after another, such that before constructing each subtree,

all sequences required for its construction are read from disk. Given that the disk latencies are in the order of milliseconds (in the absence of a parallel I/O) as opposed to microseconds in fast communication networks, a communication-based alternative is likely to be more practically efficient. For this reason, we implemented the following communication-based solution for subtree construction.

Each processor partitions its buckets into variable-sized batches, such that the sequences required to construct all buckets in each batch would occupy $O(\frac{N}{p})$ space. Before constructing a batch, all sequences needed for its construction are fetched through two collective communication steps — the first to request the processors that have the required sequences, and the second to service the request. The processor that has a given sequence is determined in constant time by recalling the initial distribution of S . A processor may exhaust all its batches, in which case it continues to participate in the remaining communication rounds to serve requests from other processors.

In the above communication based solution, each processor receives $O(\frac{N}{p})$ characters from all other processors per communication step. However, the size of the buffer used to send sequences to other processors is not bound by $O(\frac{N}{p})$. This is because requests from different processors may intersect, in the worst case over all of $O(\frac{N}{p})$ local data; for which the likelihood of happening increases with the number of processors. We resolved this issue by implementing a *customized Alltoallv*, which ensures $O(\frac{N}{p})$ size for the buffers by doing $p-1$ sends and receives instead of one collective communication.

4.3.2 Detecting Overlaps and Clustering In Parallel

Once a distributed representation of GST is constructed, the next phase detects overlaps and performs clustering in parallel. We designed this clustering phase as a master worker paradigm with one master and p worker processors. In addition to concerns typical to a single master-multiple workers setup such as keeping the master processor available and all worker processors busy, designing our master-worker model presents other unique challenges.

In a traditional master-worker model, the master processor generates and distributes work,

while the worker processors process the work. This traditional model, however, is not appropriate for our clustering algorithm because the GST required to generate promising pairs (i.e., generate “work”) is stored in a distributed fashion among the worker processors. Therefore, we designed a variant of this traditional model in which the master processor serves as a necessary intermediary only to maintain clusters and distribute work in a load balanced fashion; while the worker processors in addition to processing the work (by aligning pairs) also generate work (by generating pairs). Care must be taken that the rate of work generation is neither too fast to result in a memory overflow (because pairs have to be stored in the master processor’s local memory until they are allocated for alignment computation) nor too slow to result in unnecessary processor wait times. Moreover, as not all generated pairs are necessarily selected for alignment, it is necessary to regulate the rate of pair generation in order to maintain a steady rate in alignment computation. Another cause of concern is that workers will start to run out of pairs to generate from their portion of the GST dynamically as execution progresses. Henceforth, we call such workers *passive* while those that still have pairs to generate as *active*. In the interest of maintaining parallel efficiency, it is necessary to keep passive workers busy computing alignments. Also, allocating pending alignment computations to these passive workers ahead of any active worker can help balance the work generated vs. processed dynamically.

With the above goals in mind, we designed an iterative solution with responsibilities as shown in Figure 4.9. The master and worker processors interact iteratively until all promising pairs are generated and all alignments identified as necessary have been computed. The master processor is responsible for maintaining the clusters, selecting and allocating pairs for alignment computation, and load balancing. Each worker processor is responsible for generating promising pairs from its local GST portion in decreasing order of maximal match length, computing alignments for pairs allocated by the master processor, and report the alignment results to the master processor. To reduce communication setup costs, the worker processors send pairs in batches instead of one pair at a time. Similarly, the master processor also allocates pairs for alignment computation and collects their results in batches.

The master and workers store and maintain the following information.

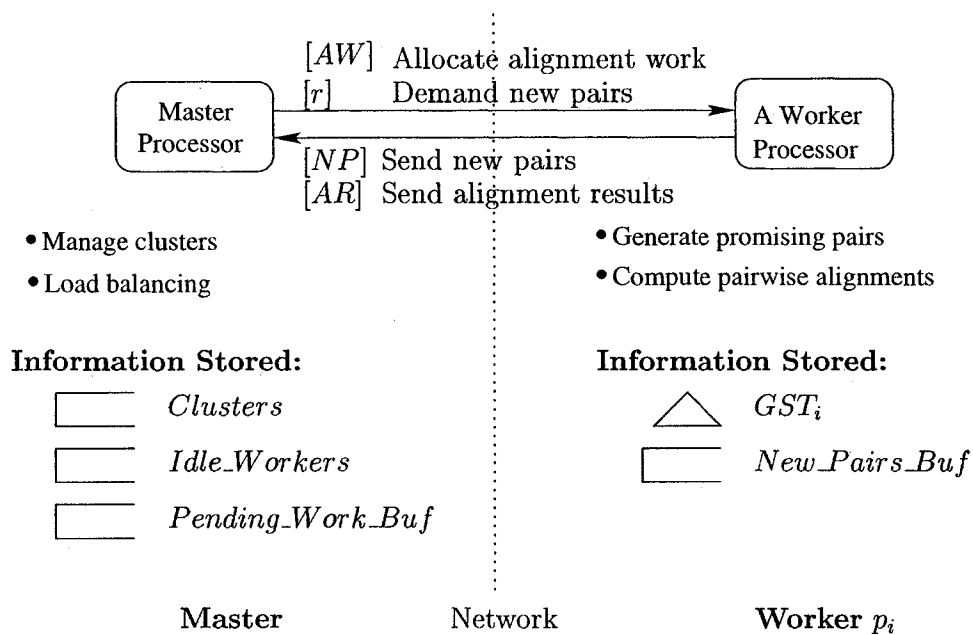


Figure 4.9 A single master-multiple workers design for detecting overlaps and clustering in parallel, with responsibilities designated as shown. Arrows indicate the direction of communication.

Information at the Master Processor:

- *Clusters*: the set of all sequence clusters maintained and updated dynamically. This is implemented using the union find data structure;
- *Pending_Work_Buf*: a fixed size buffer to temporarily store the pairs selected but not yet allocated to any worker for alignment computation. This is implemented as a circular queue; and
- *Idle_Workers*: a list of all passive workers that do not have any alignment work allocated to them. This is implemented as a queue.

Information at a Worker Processor p_i :

- GST_i : the local portion of GST; and

- *New_Pairs_Buf*: a fixed size buffer to temporarily store newly generated promising pairs that have not yet been sent to the master processor. This is implemented as a circular queue.

The following messages are exchanged between the master and an arbitrary worker processor p_i at a given iteration:

- *AW*: a new batch of alignment work allocated by master to p_i . The number of pairs sent in each batch is user-specified and is fixed — we call this number the *batch size* and denote it by b . *AW* is implemented as an array;
- r : the number of promising pairs that the master requests p_i to send during its next communication with the master. This number is variable and is determined dynamically by the master as explained later;
- *NP*: a batch of new promising pairs sent by p_i to the master processor. This is implemented as an array;
- *AR*: a list of alignment results sent by p_i to the master processor. The results are for the alignments computed over the most recent batch of pairs allocated by the master to p_i . *AR* is also implemented as an array;

Figures 4.10 and 4.11 detail the algorithms for the master and a worker processor, respectively.

In each iteration, the master processor polls for messages from any of the workers. When a message arrives from a worker p_i , the master updates *Clusters* using the alignment results that are satisfactory, scans the batch of newly generated pairs from p_i , and adds only those pairs for which alignments are necessary to *Pending_Work_Buf*. It then repeatedly extracts batches of size b from *Pending_Work_Buf*, dispatching each batch to an idle worker. If all workers become idle, then it signals the end of clustering. If no more idle workers remain and if there is more work left in the *Pending_Work_Buf*, then the next batch of b pairs are allocated to p_i . In the same message, the master also piggybacks the number of new pairs,

Algorithm 5 *Algorithm for Master Processor*

1. $Clusters \leftarrow$ Initialize such that each sequence is in a cluster of its own
 $p_{active} \leftarrow p$
 $Idle_Workers \leftarrow \emptyset$
2. REPEAT
 - Blocking Receive** until message from an arbitrary processor p_i
 - $NP \leftarrow$ new promising pairs
 - $AR \leftarrow$ alignment results
 - IF $NP = \emptyset$ AND p_i is active THEN
 - Mark p_i as passive
 - Decrement p_{active}
 - Update $Clusters$ based on AR
 - $NP' \leftarrow$ Identify pairs in NP that need alignment computation
 - $r \leftarrow \min\{\frac{|NP|}{|NP'|} \times \frac{p}{p_{active}} \times b, \frac{|Pending_Work_Buf|}{p_{active}}\}$
 - Add NP' into $Pending_Work_Buf$
 - FOR EACH $p_j \in Idle_Workers$ DO
 - $AR \leftarrow$ Dequeue $\min\{b, |Pending_Work_Buf|\}$ pairs
 - IF $AR \neq \emptyset$ THEN
 - Send AR to p_j**
 - Remove p_j from $Idle_Workers$
 - $AR \leftarrow$ Dequeue $\min\{b, |Pending_Work_Buf|\}$ pairs
 - IF $AR \neq \emptyset$ OR $r > 0$ THEN
 - Send (AR, r) to p_i**
 - ELSE
 - Add p_i into $Idle_Workers$
 - UNTIL all workers become idle
3. Send termination signal to all workers
4. Output $Clusters$

Figure 4.10 The algorithm for the master processor. Bold font indicates a communication step.

Algorithm 6 *Algorithm for a Worker Processor p_i*

1. $AW \leftarrow$ Generate next b promising pairs from GST_i
2. $AR \leftarrow$ Compute alignments on AW
3. $AW \leftarrow$ Generate next b promising pairs from GST_i
4. $NP \leftarrow$ Generate next b promising pairs from GST_i
5. $r \leftarrow b$
6. REPEAT
 - Send NP and AR to master**
 - $AR \leftarrow$ Compute alignments on AW
 - $(AW, r) \leftarrow$ **Non-blocking Receive** from master
 - REPEAT
 - Generate r pairs from GST_i and add to New_Pairs_Buf
 - UNTIL message arrives from master OR New_Pairs_Buf is full
 - IF no message from master THEN
 - $(AW, r) \leftarrow$ **Blocking Receive** until master sends a message
 - $NP \leftarrow$ Extract r pairs, first from New_Pairs_Buf and then from GST_i if necessary
 - UNTIL no more promising pairs to generate from GST_i
7. REPEAT
 - $AW \leftarrow$ **Blocking Receive** from master
 - $AR \leftarrow$ Compute alignments on AW
 - Send AR to master**
 - UNTIL master sends termination signal

Figure 4.11 The algorithm for each worker processor. Bold font indicates a communication step.

r , that it expects to receive from p_i in its next communication; r is given by: $\min\left\{\frac{|NP|}{|NP'|} \times \frac{p}{p_{active}} \times b, \frac{|Pending_Work_Buf|}{p_{active}}\right\}$, where p_{active} denotes the number of active processors. The main idea is to request as many pairs as necessary to expect that b of them would be selected for alignment computation. In other words, this load balancing strategy aims at regulating the inflow of work so as to keep the outflow roughly constant.

In each iteration, a worker processor generates as many new promising pairs as requested by the master processor and sends them in a message along with the results of the latest alignments it computed. While waiting for the master to reply, the worker computes alignments on the batch of pairs allocated by the master during the previous iteration. This is effective in masking the communication wait time with computation. If alignment computation is completed before

the master replies, then the workers processor resumes from its earlier state of pair generation and generates fresh batches of promising pairs from its local GST portion until either a master's reply arrives or its temporary store *New_Pairs_Buf* is full. If a worker becomes passive, it keeps itself busy by computing alignments that the master allocated.

4.3.3 Software Availability

The PaCE software is implemented in C, and requires a multiprocessor cluster with support for MPI (e.g., MPICH [Gropp *et al.* (1996)]). The software is copyrighted by Iowa State University, and is available free for academic use.

4.4 Results and Applications

In this section, we present two different applications of our PaCE clustering method — EST clustering and genome assembly.

4.4.1 EST Clustering

4.4.1.1 Quality Assessment

We validated the accuracy of PaCE clustering in the context of clustering ESTs using a benchmark data set consisting of 168,200 *Arabidopsis thaliana* ESTs [Zhu *et al.* (2003)]. It was possible to create this benchmark data because the genome of *Arabidopsis* has already been sequenced. The benchmark data was created by spliced alignment of the ESTs to their cognate locations in the *Arabidopsis* genome, with subsequent clustering based on genome location: ESTs that overlap in at least 40 *bp* to same or proximate genomic locations were clustered. By this exercise, out of the 168,200 ESTs, 146,527 ESTs mapped to unique locations, while the remaining 21,673 ESTs mapped to more than one cognate location. Each EST aligning to more than one genome location was mapped to the cluster corresponding to the location that gave the maximum alignment score with the EST. This procedure of generating the benchmark clusters captures various interesting cases: (i) ESTs originating from the same

gene are clustered irrespective of their mRNA transcript source; and (ii) ESTs originating from highly similar genes are separated. Chimeric ESTs were excluded from the benchmark.

Our procedure was to cluster the benchmark EST data with PaCE, and then run the CAP3 assembly program [Huang and Madan (1999)] on each resulting cluster, so that consensus sequences (contig) representing the putative source mRNA transcripts can be generated for the clusters output by PaCE. In this process, it is possible that a given cluster output by PaCE gives rise to multiple contigs — this could happen because of either of the two following reasons: (i) ESTs from alternatively spliced variants of the same gene are clustered together by PaCE, or (ii) ESTs from similar/related but different genes are clustered together by PaCE. It is, however, guaranteed that no EST is assembled into more than one contig. Once assembled, the ESTs are grouped based on their corresponding contigs. The set of clusters resulting from this grouping is henceforth referred to as the “PaCE clusters”.

The platform for our PaCE experiments was a 30 node IBM xSeries machine, with each node containing two 1.26 GHz Intel Pentium III processors and 2.25 GB RAM. The nodes are interconnected by Myrinet. After running PaCE in parallel, the task of running the serial CAP3 program on each individual output cluster was trivially parallelized by distributed the clusters across processors initially and running multiple instances of CAP3 on each local set of clusters.

We compared the PaCE clusters against the benchmark clusters. Given that CAP3 is also one of the popular programs used for clustering ESTs [Liang *et al.* (2000)], we ran CAP3 directly on the 168,200 ESTs and compared the resulting CAP3 clusters against both PaCE and benchmark clusters. Running CAP3 directly on 168,200 ESTs was enabled by running it on a computer with 3.25 GB RAM.

To assess quality as a function of data size, two subsets of clusters were extracted from the benchmark clusters, such that the number of ESTs represented in the sets were $\approx 50,000$ and $\approx 100,000$, respectively. The ESTs were input to the programs in no particular order.

A set of clusters is compared against the benchmark as follows: Let “test clusters” denote either the PaCE clusters or CAP3 clusters. For both the test clusters and benchmark clusters,

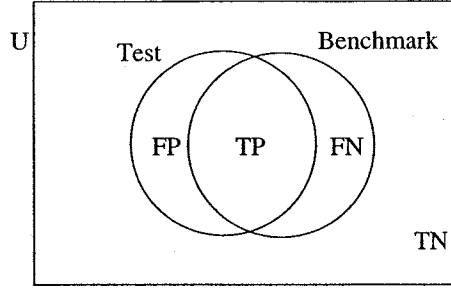


Figure 4.12 Illustration of quality validation measurements True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN). 'U' refers to the set of all possible pairs of the input ESTs.

generate pairs of ESTs such that both ESTs in a pair are from the same cluster. Based on the number of such pairs, the following measurements are defined; see Figure 4.12 for an illustration. A pair generated from the test clustering is called a *true positive* (TP) if it is also paired in the benchmark clustering; it is called *false positive* (FP) otherwise. A pair absent from the test clustering is called a *true negative* (TN) if it is also absent from the benchmark clustering; it is called *false negative* (FN) otherwise. Based on these measurements, another set of quality measures are defined: *Overlap quality* is the ratio of the number of TPs to the total number of unique pairs extracted from the clusters of both results, and is given by $OQ = \frac{TP}{TP+FN+FP}$; OQ is also known as Jaccard index [Jain and Dubes (1988)]. *Specificity* is the fraction of correctly predicted pairs with respect to the total number of pairs predicted by the test clustering, and is given by $SP = \frac{TP}{TP+FP}$. *Sensitivity* is the fraction of correct pairs (from benchmark) predicted in the test clustering, and is given by $SE = \frac{TP}{TP+FN}$. Overall accuracy can be given by the correlation coefficient, which is given by:

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

Ideally, the test clusters should exactly match benchmark clusters, i.e., $OQ=SP=SE=CC=100\%$.

The results of assessing the quality of our software and CAP3 using the benchmark data sets are shown in Table 4.1. Observing the measurements OQ , SP , SE and CC , our results are very close to the results of CAP3, with CAP3 showing slightly better results than PaCE. For either programs, the sensitivity is lower than the specificity and this is attributable to

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

Table 4.1 Quality assessment of PaCE and CAP3 clusters using clusters generated from different portions of the benchmark data set.

the conservative nature of clustering criteria used. The results are based on the choice of the quality threshold parameters of PaCE and CAP3, experimentally found to optimize specificity and sensitivity simultaneously. For the PaCE run, we used the following alignment scoring scheme: match = 2, mismatch = -2, opening gap penalty = 6, gap continuation penalty = 1. Also, the cutoff length for maximal match length (w) was set to 40 *bp*, and an alignment was deemed significant if its score is within 75% of a perfect matching score. CAP3 was also run under similar parameter settings for overlap percentage identity.

To enable a direct comparison with CAP3, we compared the PaCE clustering directly against CAP3 clustering, treating CAP3 as a benchmark clustering and the PaCE clustering as the test clustering, for the purpose of analysis. The measurements obtained for the 168,200 data collection are as follows: OQ=95.25%, SP=98.76%, SE=96.4% and CC=97.58%.

Quality Assessment using Clone Mates Information:

We also evaluated the effect of clone mate information on the quality of PaCE clustering. This was achieved as follows: ESTs are grouped as pairs based on their source cDNA clones [Seki *et al.* (2002)]. Following this, the benchmark clusters are updated by merging the clusters that are linked by at least one clone mate pair. Note that it might happen that clone information is not available for some ESTs in the input data, and/or there are ESTs that were originally derived from non-overlapping cDNA transcripts of the same gene. In such cases, the benchmark clustering is based only on spliced alignments. For the 168,200 ESTs, 16,992 pairs of ESTs were linked with clone identifiers.

n	50,012		100,003		168,200	
	w/o CM	w/ CM	w/o CM	w/ CM	w/o CM	w/ CM
OQ	84.29	88.06	81.94	87.46	85.89	88.74
SP	98.71	97.75	96.28	94.98	96.43	94.94
SE	85.23	89.88	84.62	91.7	88.71	93.14
CC	91.21	93.72	90.26	93.32	92.49	94.04

Table 4.2 Quality assessment of PaCE clusters with and without clone mates (CM) information, against the *Arabidopsis* benchmark clusters.

PaCE clustering allows for input with clone mate identifiers. Using the clone mate information, updated PaCE clusters were obtained in the following manner for the different subsets of the benchmark data: cluster the benchmark data using PaCE with the added input of clone mates; run CAP3 on each resulting cluster and group ESTs based on contigs. Clone mates information was also passed as input to CAP3 so that the final clustering is with the clone mates information. The results were compared against the corresponding benchmark clusters. To measure the improvement in quality of clustering, we also compared the PaCE clusters obtained without clone mates information against the new benchmark clusters.

The results for 168,200 ESTs and its subsets are shown in Table 4.2. It can be observed that on the 168,200 data set, the clone mate information was instrumental in improving the overall CC from 92.49% to 94.04%.

4.4.1.2 Performance Evaluation

We first studied the performance of the PaCE software on the same 168,200 *Arabidopsis* EST collection used in our quality assessment experiments. The total run-times as a function of the number of processors for various data sets are shown in Figure 4.13a. For these experiments, we used a batch size of 60. As can be observed, the run-times show near perfect scaling with the number of processors. We are also interested in the growth of run-time as a function of the data size for a fixed number of processors. While the memory required scales linearly with the problem size, the total run-time cannot be analytically determined and depends on the input

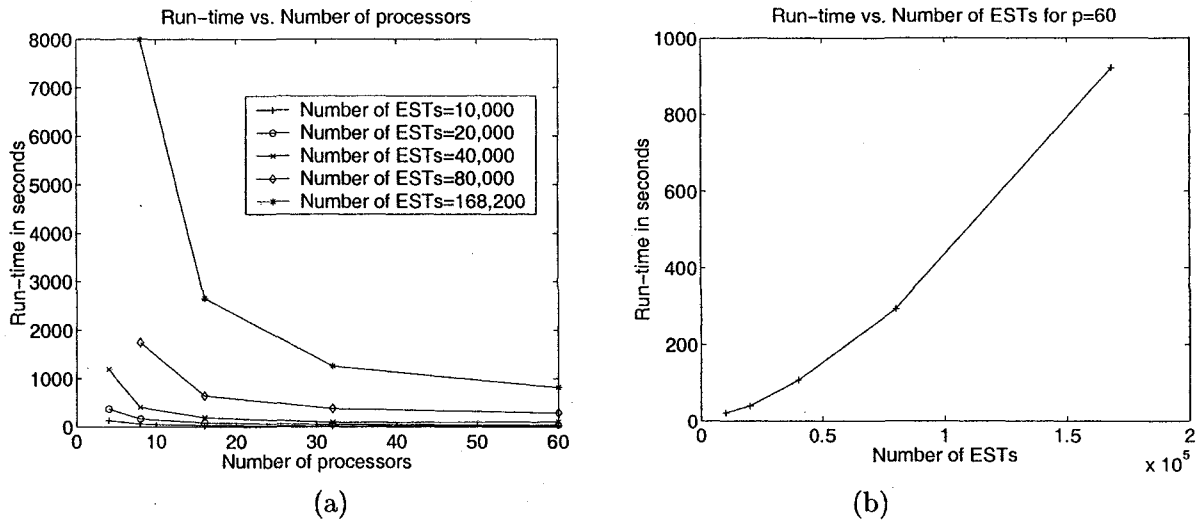


Figure 4.13 Parallel scaling of PaCE clustering.

data set. These run-times for various data set sizes are shown in Figure 4.13b.

A subdivision of the run-times into the time spent on various components of the software for 20,000 ESTs is shown in Table 4.3. Asymptotically, the largest contributor to the total run-time is the time spent in performing pairwise alignments during the clustering phase. The GST construction phase scales linearly (treating the average length of an EST to be a large constant). The clustering phase is expected to take quadratic run-time. In our approach, the time spent in pairwise alignments is significantly reduced because our algorithm (i) avoids unnecessary duplicates in generating promising pairs and (ii) processes high-quality promising pairs first which has the effect of eliminating other promising pairs from further consideration. Because of these reasons, for smaller data sizes, the alignment phase runs faster than the GST construction phase as seen from Table 4.3.

Figure 4.14a shows the total number of promising pairs generated as a function of the data size. Observe that the alignment work is done for only a small portion of the pairs generated (for e.g., 22% for the 168,200 data set). This illustrates the reduction in work achieved by processing the pairs in the decreasing order of maximal common substring length, as opposed to processing them in an arbitrary order. Also note that the number of aligned pairs that contribute to merging of clusters is linear in n , as at most $n - 1$ union operations can be

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

Table 4.3 Time (in seconds) spent in various components of parallel EST clustering as a function of the number of processors (p) for 20,000 ESTs.

performed. Because of the nature of master-slave interactions during the clustering phase, the number of pairs that are actually aligned varies slightly as the number of processors changes. We found the variation to be insignificant.

Figure 4.14b shows the number of clusters as a function of the cluster size for 168,200 ESTs. About 44% of the clusters formed contain a single EST. A few clusters contain as many as several hundred ESTs (e.g., there are 34 clusters with size above 200). This non-uniformity in the size distribution in clusters is the primary reason why fragment assembly software has large memory and run-time requirements when applied to EST clustering.

The effect of varying batch size on the run-time of the clustering phase of PaCE is shown in Figure 4.15. When the batch size is small, the master and workers exchange messages more frequently, thereby making the communication overhead dominant. With a large batch size, EST clusters are less frequently updated, causing alignment of more promising pairs than necessary. Empirically, we found the optimal batch size for the benchmark data set to be in the range of 20–60. Note that this optimal range may vary depending on the size of input, the number of processors, and the speed of network interconnect of the underlying parallel platform.

Mouse EST Clustering

For the purpose of demonstrating the capability of large-scale clustering, we also evaluated

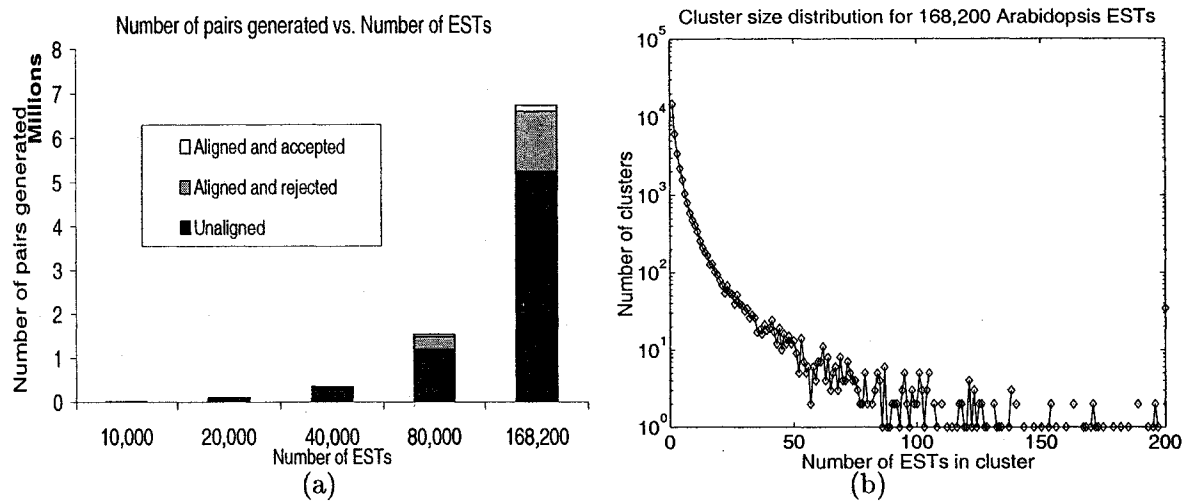


Figure 4.14 (a) Promising pair generation and alignment statistics of PaCE, as a function of data size. (b) The number of clusters as a function of the cluster size for 168,200 ESTs.

the performance of PaCE clustering on the largest available mouse EST data in GenBank as of April 2006. This data, after cleaning and polyA tail removal, comprised of 3,783,854 ESTs. We conducted this large-scale experiment on the IBM BlueGene/L (BG/L) supercomputer [Adiga *et al.* (2002)] at Iowa State University. Each BG/L node contains two 700 MHz IBM PowerPC architecture-based processors and a 512 MB RAM. There is no additional node-level storage available in the form of any swap space. The nodes are connected through a 3D-torus network. Each dual-processor node was used in *co-processor mode*, i.e., one processor was used for computation and the other processor was used for communication. For this reason, we will use “nodes” and “processors” synonymously.

Tables 4.4 and 4.5 show the total and phase-wise run-times respectively, as a function of both the input and processor sizes. Both these tables show the range of processors on each input up to which the run-time scales linearly, and beyond which the problem size becomes too small for the processor size. It can be observed that as many as 512 processors can be used efficiently for even an input containing as few as 100,000 ESTs.

The increase in run-time with input size in Table 4.4 conforms with the asymptotic quadratic behavior expected on EST data. Also, observe that the run-time for clustering phase dom-

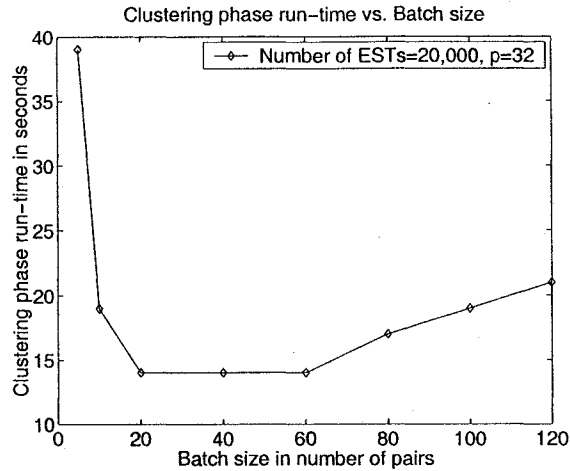


Figure 4.15 PaCE run-time as a function of the number of pairs allocated at a time for pairwise alignment.

inates the total run-time for larger input sizes. Figure 4.16 shows the number of promising pairs generated by PaCE based on a minimum cutoff maximal match length of 30 *bp*, as a function of the input number of ESTs. As shown in the figure, the number of pairs is expected to grow quadratically with the number of input ESTs. Figure 4.16 also shows that alignments are computed for only $\approx 10\text{-}12\%$ of the generated pairs, demonstrating the significant run-time savings achieved through the PaCE heuristic techniques.

We evaluated the effectiveness of the master-worker paradigm of the PaCE algorithm as follows: Figure 4.17a shows the average run-time (as a percentage of the total run-time) spent by a worker processor waiting for the master processor without performing any computation. As can be expected, this idle time decreases with increase in input size for a given processor size. Figure 4.17a shows that the idle time ranges from 7% to 22% on the input tested. We also evaluated idle time on the master processor to check its availability to the worker processors. Figure 4.17b shows that the master processor is available for at least 80% of the run-time, for even a small input size of 100,000 ESTs on as many as 1,024 processors. This suggests that the master processor is not a bottleneck even as the number of processors is increased to the order of thousands.

The PaCE software is used by the NSF funded PlantGDB project (<http://www.plantgdb.org>),

Total run-time

Number of ESTs	Number of processors					
	32	64	128	256	512	1024
100,000	80	43	20	11	6	5
250,000	225	114	50	25	14	11
500,000		275	146	74	34	25
1,000,000			281	138	78	46
2,000,000				421	272	153
3,783,854					1198	574

Table 4.4 PaCE clustering run-time (in minutes) on ≈ 3.78 million mouse ESTs using 1,024 IBM BlueGene/L processors.

Number of ESTs	GST construction phase run-time						Clustering phase run-time					
	32	64	128	256	512	1024	32	64	128	256	512	1024
100,000	44	25	11	6	3	2	36	18	9	5	3	3
250,000	110	56	29	15	8	6	115	58	22	11	6	5
500,000		123	63	32	16	12		152	83	42	18	13
1,000,000			135	69	38	22			146	69	40	24
2,000,000				174	91	51				247	181	102
3,783,854					248	158					950	416

Table 4.5 Phase-wise run-time (in minutes) of PaCE clustering on ≈ 3.78 million mouse ESTs using 1,024 IBM BlueGene/L processors.

which has EST clusters for ≈ 100 plant EST collections. The sizes of these collections range from as small as a few thousands to over a million rice ESTs.

4.4.2 Clustering for Genome Assembly

Our next application of PaCE was on the problem of genome assembly. The problem of genome assembly primarily relies on pairwise overlap information among an input set of fragments sequenced from the target genome. Despite technological advances, however, genome assembly is still largely treated as a problem of serial computers that result in long run-times and exorbitant memory requirements. (See Sections 3.1.2 and 3.2 for more details.)

As earlier mentioned in Section 3.1.2, a strategy of clustering before performing the as-

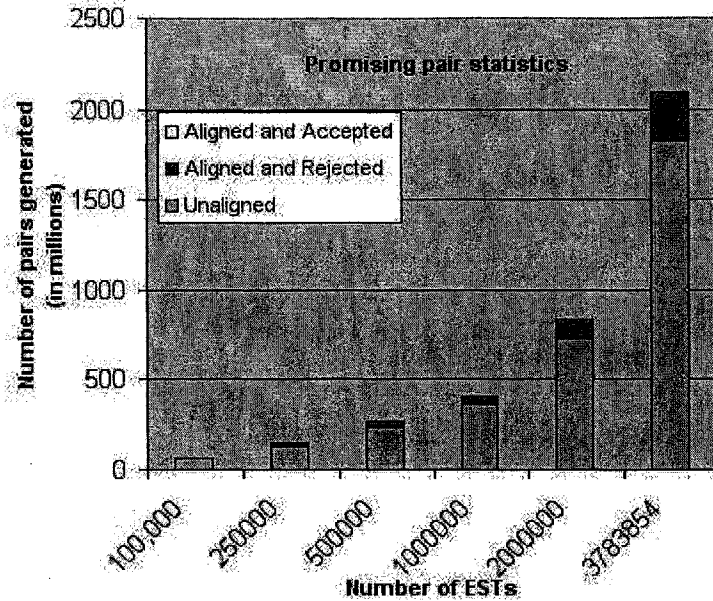


Figure 4.16 Number of promising pairs generated vs. number of pairs aligned by PaCE while clustering ≈ 3.78 million mouse ESTs.

sembly has several advantages, if: (i) clustering can break a large problem size into numerous independent subproblems for assembling; (ii) clustering requires less memory than assembly as otherwise there will be no memory advantage in using clustering prior to assembly, and the problem sizes solvable using the clustering based assembly approach cannot be larger than the direct assembly approach; and (iii) clustering provides a faster alternative to detect overlaps and consumes less run-time — the latter can be expected because it involves lesser work than assembling (i.e., its task is only to group the sequences that belong the same contig together, without actually assembling the contig). All the above desired properties are met by the PaCE clustering method.

We applied PaCE on maize gene-enriched genomic fragments. Gene-enrichment, as described in Section 2.2.2.3, is a technique by which gene-rich portions of a genome are selectively sampled. Gene-enrichment is ideal for the use of clustering based approach to assembly because of the following reason: Because of selectively sampling the gene-rich portions of the genome and repeat masking, an initial assembly of gene-enriched fragments generates a large number of contigs that correspond to the many sparsely located genomic stretches from which

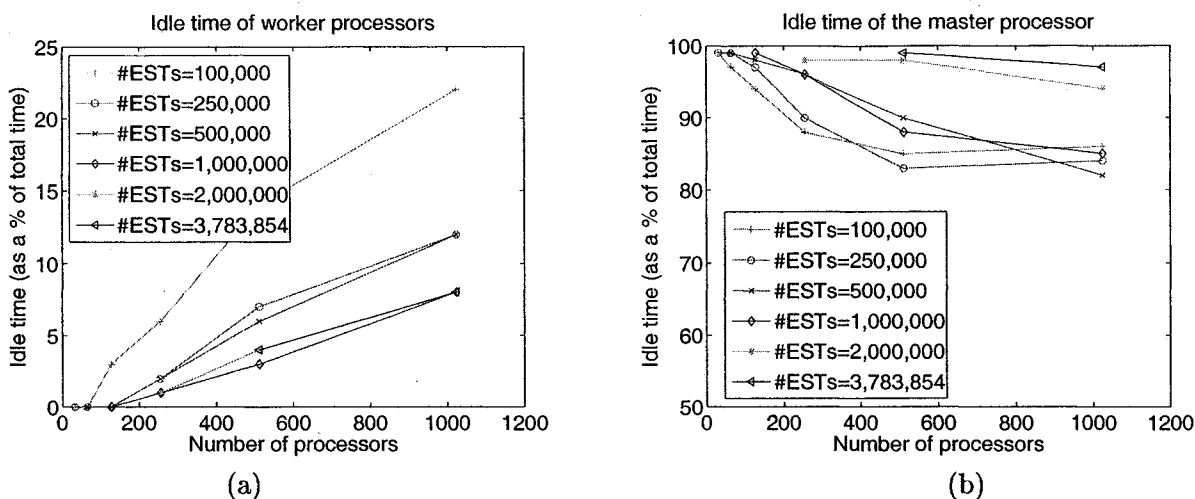


Figure 4.17 Idle run-time characteristics of PaCE clustering of the mouse EST data. (a) Average percentage idle time for each worker processor. (b) Percentage idle time of the master processor.

the fragments were originally derived [Emrich *et al.* (2004)]. Thus, we can expect clustering to partition the input fragments into “clusters” corresponding to each of the enriched genic regions. These individual regions can be assembled post-clustering using any serial assembler of choice [Kalyanaraman *et al.* (2006b)]. While our PaCE clustering framework supports space optimality, run-time efficiency and massive parallelism, the assembly task is trivially parallelized by distributing the clusters across multiple processors and running multiple instances of a serial assembler in parallel. The space and other limitations of these assemblers will now not be a limiting factor because of the relatively small size of each cluster.

4.4.2.1 Data

Two gene-enrichment methods, Methyl-Filtrated (MF) [Rabinowicz *et al.* (1999)] and High-C₀t (HC) [Yuan *et al.* (2003)], were used to sequence the maize genome, as of April 2005. The maize genomic data composed of 3,124,130 fragments with total length over 2.5 billion *bp*. This includes 852,838 MF and HC fragments. Also available are fragments from WGS and BAC sequencing. A summary of the entire maize data is provided in the first three columns of Table 4.6.

Fragment Type	Before Preprocessing		After Preprocessing	
	Number of Fragments	Total length (in millions)	Number of Fragments	Total length (in millions)
MF	411,654	335	349,950	288
HC	441,184	357	427,276	348
BAC	1,132,295	964	425,011	307
WGS	1,138,997	870	405,127	309
Total	3,124,130	2,526	1,607,364	1,252

Table 4.6 Maize genomic fragment data types and size statistics: Methyl-filtrated (MF), High- C_{0t} (HC), Bacterial Artificial Chromosome (BAC) derived, and Whole Genome Shotgun (WGS).

As with any other assembler, the first step in our framework is to screen the input fragments for known repeats and vector sequences. This preprocessing step was designed and performed by Emrich *et al* [Emrich *et al.* (2004)]. A brief overview of this step is as follows: raw fragments obtained from sequencing strategies can be contaminated with foreign DNA elements known as vectors, which are removed using the program *Lucy* [Chou and Holmes (2001)]. In addition, a database of known and statistically-defined repeats was designed, and all fragments were screened against it. The matching portions are masked with special symbols such that our clustering method can treat them appropriately during overlap detection. The last two columns in Table 4.6 show the results of preprocessing the data using our repeat masking and vector screening procedures. As expected, preprocessing invalidates a significant number of shotgun fragments ($\approx 60\text{--}65\%$) because of repeats, while most of the fragments resulting from gene-enrichment strategies are preserved. An efficient masking procedure is important because unmasked repeats cause spurious overlaps that cannot be resolved in the absence of paired fragments spanning multiple length scales. Furthermore, it provides a computational means to preferentially assemble non-repetitive regions of the genome that may be gene-enriched.

This framework, illustrated in Figure 4.18, is a divide-and-conquer strategy that reduces the task of assembling one large set of fragments to the task of first identifying clusters containing genomic neighboring fragments and then assembling each cluster individually.

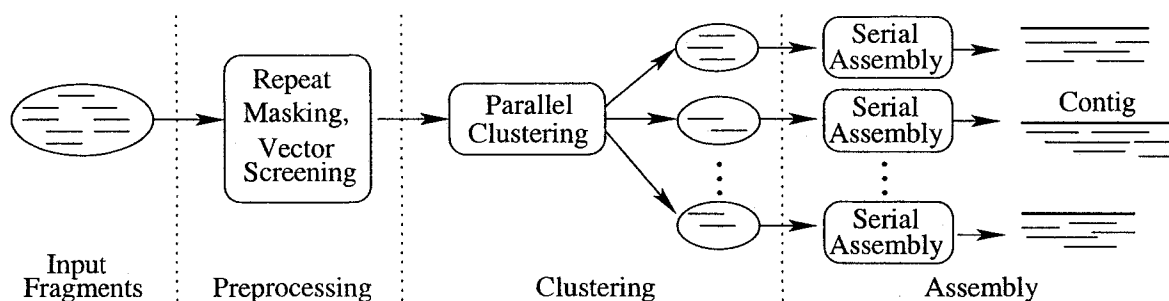


Figure 4.18 Illustration of our cluster-then-assemble framework.

4.4.2.2 Results and Validation

The results of applying our parallel genome assembly framework on the entire maize data is as follows: Cleaning the 3,124,130 fragments downloaded from GenBank took 1 hour by trivially parallelizing on 40 processors of an IBM xSeries cluster with 1.1 GHz Pentium III processors and 1GB RAM per processor. The PaCE clustering method partitioned the resulting 1,607,364 fragments (over 1.25 billion *bp*) in 102 minutes on 1,024 nodes of the BG/L, with the GST construction taking only the first 13 minutes. We used CAP3 for assembling the fragments in each resulting cluster. This assembly step finished in 8.5 hours on 40 processors of the IBM xSeries cluster through trivial parallelization.

Performance Evaluation of PaCE clustering

We studied the performance of our PaCE clustering algorithm on varying processor sizes ranging from 256 to 1,024. In what follows, we first present the performance results for the preprocessing phase, followed by the entire algorithm (including the clustering phases).

For evaluating the preprocessing phase (GST construction), experiments were conducted on two subsets of the maize data, with sizes 250 and 500 million *bp* that comprised 322,009 and 649,957 fragments, respectively. Figure 4.19 shows the parallel run-times and their breakdown into communication and computation times, all of which show linear scaling with both processor and input sizes.

We report the performance of the entire PaCE clustering algorithm, with its single master-

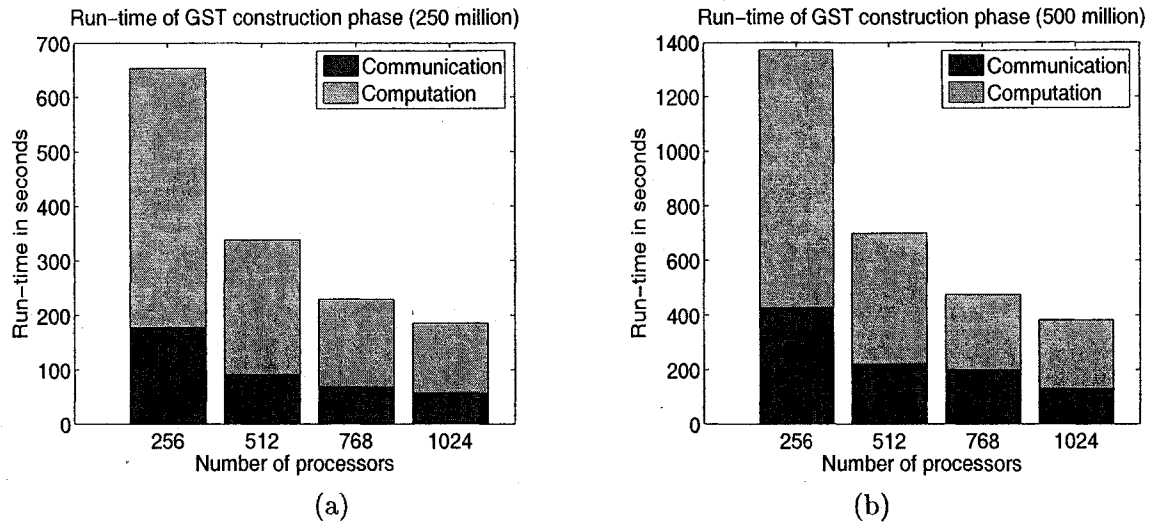


Figure 4.19 Parallel run-times for constructing GST on inputs of sizes: (a) 250 million, and (b) 500 million *bp*.

multiple worker implementation on the IBM BlueGene/L supercomputer. The results are shown in Figure 4.20. The results show a better scaling for the larger (500 million) input than the smaller (250 million) input. Upon increasing the number of processors from 256 to 1,024, we observe relative speedups of 2.6 for the 250 million input and 3.1 for the 500 million input. Further investigation revealed that the percentage average idle time for the processors increased from 16% on 256 processors to 26% on 1,024 processors on the 250 million input, and from 9% to 16% for the 500 million input — indicating that more processors can be deployed while maintaining efficiency if the problem size is larger. Note that a full sequencing project will generate over 22 billion *bp* (30 million fragments each about 750 *bp*), on which tens of thousands of processors can be utilized with our scheme.

Figure 4.20b shows the number of promising pairs generated as a function of the input size. This figure reinforces the effectiveness of our promising pair, clustering and pair generation heuristics in significantly reducing the number of alignments computed. For the entire maize data, which has 1,607,364 fragments of total size 1.252 billion *bp*, only about 40% of the pairs generated are aligned. However, less than 1% of the pairs aligned contributed to merging of clusters, indicating the presence of numerous medium-sized (≈ 100) repetitive elements that

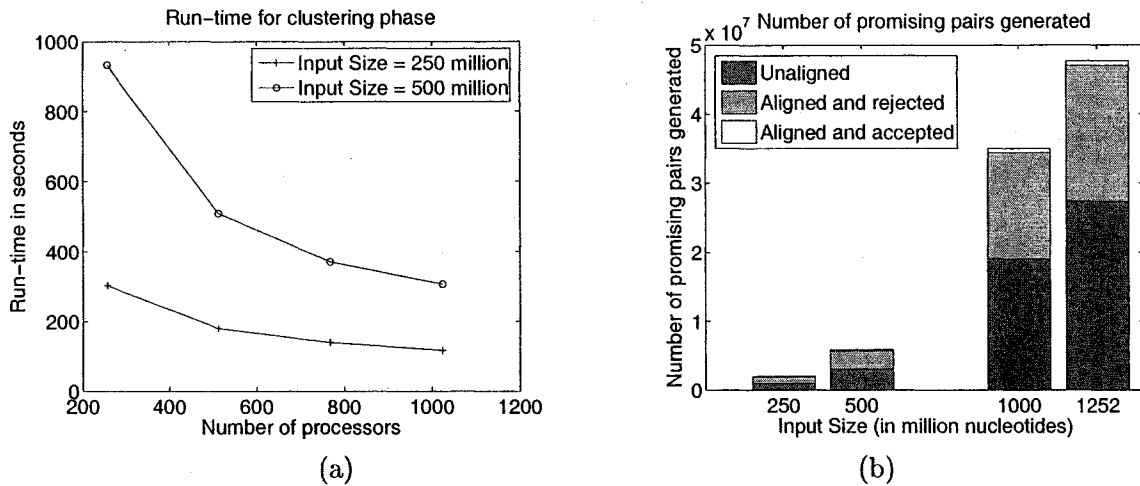


Figure 4.20 (a) Total parallel run-time for the entire clustering algorithm excluding that of GST construction. (b) The number of pairs generated, aligned, and accepted as a function of input size.

survived initial screening procedures. Growth in the number of promising pairs is a direct reflection of the expected worst-case quadratic growth in the maize data. The number of promising pairs generated and the relative savings in the alignment work are highly data sensitive. For example, recall that only 22% of generated pairs were aligned on the *Arabidopsis* EST data, as reported in Section 4.4.1.2.

Biological Validation

The biological validation of the maize gene-enriched assembly was performed by Emrich *et al.* [Emrich *et al.* (2004); Fu *et al.* (2005)]. A summary of the validation results are as follows: Our assembly resulted in a total of 163,390 contigs formed by two or more input fragments, and 536,377 singletons. Singletons are fragments that do not assemble with any other fragment because of sharing no overlap and/or having a high repetitive content that was masked during preprocessing. On an average, each cluster assembled into 1.1 contigs; given that the CAP3 assembly is performed with a higher stringency, this result indicates the high specificity of our clustering method and its usefulness in breaking the large assembly problem into disjoint pieces of easily manageable sizes for conventional assemblers. The overall size of our contigs is about 268 million *bp*, which is roughly 10% of the entire maize genome. Upon validation by Emrich *et al.* using independent gene finding techniques, it was confirmed that

our contigs span a significant portion ($\approx 96\%$) of the estimated gene space. The results of our assembly can be graphically viewed at <http://www.plantgenomics.iastate.edu/maize>. For further biological details on our on-going effort to assemble the maize genome and a thorough discussion of the results on an earlier version of maize data with less than a million fragments, see [Fu *et al.* (2005)].

The PaCE clustering framework for gene-enriched genome assembly has allowed us to focus on developing parallel methods while benefiting from and not duplicating the painstakingly built-in biological expertise of current assemblers. Furthermore, this allows one to generate assemblies consistent with what would have been generated by any conventional assembler, except that the problem size reach and speed of the assembler is significantly enhanced.

Our strategy is applicable even for conventional whole genome shotgun assembly. This is because gaps invariably occur in sampling, or through repeat masking, or owing to the difficulty in sequencing certain regions of the genome. As a result, an initial assembly is expected to consist of a large number of contigs that are subsequently scaffolded, followed by targeted biological experiments to fill in the gaps. As an example, in the human genome project [Venter *et al.* (2001b)], using whole genome shotgun sequencing resulted in an initial assembly with over 221,000 contigs, and the largest contig spanned only under 2 million *bp* of the genome.

CHAPTER 5. DETECTION OF LTR RETROTRANSPOSONS

In the previous two chapters, we described algorithms and software for performing sequence analysis that enable the discovery and understanding of genes, of their related sequences that constitute the transcriptome of an organism, and of the entire genome itself. Besides genes, there are other substructures within a genome that are subjects of active research topics. Identifying these substructures within a genome is the first step towards understanding their biological role. Comprising a majority of these genomic patterns are the *repeat* elements, which are, as the name suggests, recurrent genomic sequence patterns. In this chapter, we focus on the problem of identifying one of the most abundant classes of genomic repeat elements called the *LTR retrotransposons*.

Retrotransposons are DNA sequences that reside within cells of a host organism, copying and inserting themselves into the host genome. Studies have revealed their ubiquity in many eukaryotic organisms, both plants and animals — they constitute more than 50% of the maize genome [Meyers *et al.* (1998); SanMiguel *et al.* (1998, 1996)], up to 90% of the wheat genome [Flavell (1986)] and at least 45% of the human genome [Lander *et al.* (2001)]. *LTR retrotransposons* form a special class of retroelements that are typically characterized by two long terminal identical repeat sequences, one at the 5' end and the other at the 3' end of the inserted retrotransposon; these terminal repeats are referred to as *Long Terminal Repeats* or *LTRs*. LTR retrotransposons were originally discovered in maize and tobacco [Grandbastien *et al.* (1989); Johns *et al.* (1985); Varagona *et al.* (1992)], and are now known to be abundant in numerous complex eukaryotic plant (e.g., barley, rice, maize, wheat, etc.) and animal (e.g., *Drosophila*, human, mouse, etc.) genomes.

Ever since their discovery, LTR retrotransposons have been a topic of great research in-

terest to biologists. Understanding the behavior of these retroelements has been key to many significant advances in molecular genetics and functional genomes [Charlesworth *et al.* (1994); Feschotte *et al.* (2002); Ganko *et al.* (2003); Miller *et al.* (1998); SanMiguel *et al.* (1996)]. Because of their mobile nature, retrotransposons play key roles in genomic rearrangements [Bennetzen (1996); Feschotte *et al.* (2002); Kim *et al.* (1998)] and in the evolution of genes and genomes [Ganko *et al.* (2003); Jordan and McDonald (1999); Vicienta *et al.* (1999); Wessler *et al.* (1995); White *et al.* (1994)]. LTR retrotransposons have also been identified to be sources of spontaneous and induced mutations and are an important subject in studies relating to mutations and genetic variations [Hirochika *et al.* (1996); Kidwell and Lisch (1997); Varagona *et al.* (1992)]. The transposition mechanism by which LTR retrotransposons copy and relocate involves an RNA-intermediary — a copy of the retrotransposon is made into an RNA molecule, which is then inserted back as a DNA molecule in another location of the host genome, with the aid of an enzyme called reverse transcriptase. This mechanism being highly similar to the transposition mechanism of retroviruses such as the *HIV* has contributed to a continued interest in retrotransposon research [Bushman (2003); Coffin *et al.* (1997)]. The structural attributes of LTR retrotransposons provides significant insights into species evolution because of the following property: the 5' and 3' LTRs of a retrotransposon are completely identical when the retrotransposon inserts itself, but can undergo mutations and become increasingly divergent with time [Peterson-Burch *et al.* (2004); Xiong and Eickbush (1990)].

The aforementioned applications and many others have been contributing to a sustained research interest in LTR retrotransposons. Also with the continued advancement in sequencing technology and with various new large-scale genome sequencing projects of complex eukaryotic organisms either currently underway or finished, understanding retrotransposons and their biological role in all these genomes has become imperative in furthering research for functional and molecular genomics.

In this research, we propose an efficient algorithm for *de novo* identification of full-length LTR retrotransposons with key emphasis on quality and performance [Kalyanaraman and Aluru (2006)]. The main contributions of this research are the following:

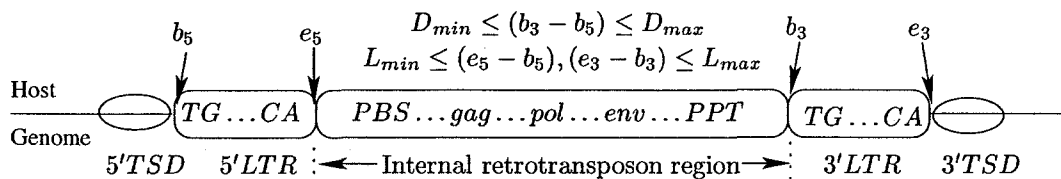


Figure 5.1 The structure of a full-length LTR retrotransposon.

- an efficient algorithm for quickly generating high-quality candidates, significantly reducing the search space. The algorithm has a run-time complexity proportional to the input size plus the number of candidates (i.e., amortized constant time per candidate);
- a thorough alignment-based evaluation of candidates using standard dynamic programming techniques that guarantees optimality in the alignment score;
- support for a robust parameter set encompassing both structural constraints and quality controls; and
- an implementation of our algorithm that can run on both serial and parallel computers.

Preliminary validations of our software indicate better quality results and significantly faster run-times when compared to previously developed software. For example, our software took 10 minutes on the yeast genome (on a 1.1 GHz Pentium III) and made better predictions than *LTR_STRUC* [McCarthy and McDonald (2003)], which took 210 minutes (on a 1 GHz Pentium III) despite not computing rigorous alignments. Furthermore, the parallel implementation of our algorithm can be used to further reduce the run-time in proportion to the number of processors used. Our approach also provides a flexible framework to incorporate more LTR-specific improvements with minimal changes to the algorithmic core.

5.1 Problem Description and Related Work

The structure of a full-length LTR retrotransposon has been well characterized in literature, and is illustrated in Figure 5.1. A full-length LTR retrotransposon is characterized by an

internal region containing the retrotransposon that is flanked by two identical repeats (5' and 3' *Long Terminal Repeats* or *LTRs*), and two identical short repeats 5-6 *bp* long (5' and 3' *Target Site Duplications* or *TSDs*) that are a result of a duplication event when the retroelement inserts itself into the host genome. The internal retrotransposon region contains the following sequences from 5' to 3': *Primer Binding Site (PBS)* is a region complementary to a tRNA 3' terminal sequence used during reverse transcription at a later stage of the retroelement's life cycle; the *gag* region is a gene that codes for a capsid-like protein; the *pol* region contains genes coding for protease, integrase and reverse transcriptase enzymes; the *env* region contains the gene coding for an envelope protein; and the 3' end of the retroelement contains a purine-rich sequence called the *Poly-Purine Tract (PPT)*.

For computational detection, these structural attributes can be modeled as follows:

- L1 **Similarity Constraint:** The 5' and 3' LTRs show a good sequence homology that can be demonstrated by a high-scoring global alignment between them. While the LTRs are identical when a retroelement inserts into a host genome, they may accumulate mutational variations — hence the need for computing an alignment.
- L2 **Distance Constraint:** The starting positions of the 5' and 3' LTRs are at least D_{min} *bp* and at most D_{max} *bp* apart along the genome. The value for D_{min} (alternatively, D_{max}) is given by the sum of minimum (alternatively, maximum) expected lengths of an individual LTR and an internal retrotransposon region; biologically reasonable values are: D_{min} under 1,000 *bp*, and D_{max} in the range [10000,15000] *bp*.
- L3 **LTR motif:** LTRs are typically known to start with *TG* and end with *CA*.
- L4 **Target Site Duplications:** The 5 (or 6) *bp* immediately left of the 5' LTR are highly similar, if not identical, to the 5 (or 6) *bp* immediately right of the 3' LTR. This repeat is called a *Target Site Duplication* or a *TSD* because it corresponds to the sequence duplicated in the host genome at the time and site of the retrotransposon's insertion.
- L5 **Other Signals:** The region between a 5' and 3' LTR pair contains a series of special purpose genes and sequences corresponding to the inserted retrotransposon: *primer*

binding site (PBS), gag, pol, env, and Poly-purine tract (PPT).

Henceforth, we refer to the above attributes also by their corresponding labels L1 through L5.

While the sequence identity expected between 5' and 3' LTRs of a retrotransposon could vary across different retroelement families, typically ranging between 70%-100% [Kim *et al.* (1998)], a high identity (>90%) has been observed in most cases [Kim *et al.* (1998); Promislow *et al.* (1999)]. Because of the strong homology expected between 5' and 3' LTRs, they are also expected to contain long exact matches. Thus, identification of exact matching repeats serves as a good starting point for LTR retrotransposon detection. Repeat detection is a well studied problem and a number of excellent programs are already available. These include *RepeatMasker* [Smit and Green (1999)], *REPuter* [Kurtz *et al.* (2001); Kurtz and Schleiermacher (1999)] and *RECON* [Bao and Eddy (2002)]. LTR retrotransposons, on the other hand, are uniquely characterized by L2. Therefore, the repeats identified by general purpose repeat identification software must be screened to eliminate repeats that do not satisfy L2. For instance, the *SMaRTFinder* program [Morgante *et al.* (2002)] designed for retrotransposon detection utilizes *REPuter* for repeat detection prior to screening for additional LTR features. The problem with this approach is the extra run-time cost incurred in initially generating repeats that are either too close or too far apart to be part of any valid LTR retrotransposon — an issue for highly repetitive genomes. Even on genomes with abundant LTR retrotransposon content, there could be an LTR sequence that is common across numerous members of the same retrotransposon family, and generic repeat finding tools will generate all pairs of these LTRs before invalid pairs are sieved out.

A more efficient solution is to build software that is specifically designed for LTR retrotransposon detection, and *LTR_STRUC* [McCarthy and McDonald (2003)] is the only available program that is so designed. It has been successfully used for detection of full-length LTR retrotransposons in *Oryza sativa* [McCarthy *et al.* (2002)], *Mus musculus* [McCarthy and McDonald (2004)] and *Drosophila melanogaster* [Franchini *et al.* (2004)]. The underlying algorithm, however, is a brute-force approach that results in unnecessarily long run-times, which could be

problematic for large genomic sequences. A more efficient algorithm will significantly reduce the cost of identifying potential LTR pairs, and the resulting time savings could be utilized to improve prediction quality.

The underlying algorithm in *LTR_STRUC* can be viewed as a two-step procedure: (i) detect all pairs of genomic locations that both satisfy L2 and are starting positions of two highly similar substrings (or “seeds”) of a particular fixed length ω (say 40 bp). Each such pair can be considered a “candidate pair”; (ii) for each candidate pair generated, extend the seeds in either direction as long as the alignment continues to satisfy L1. The resulting aligning regions are reported as a full-length LTR retrotransposon. Alignment of an extension is computed by a simple greedy strategy that aligns longer exact matches before aligning the remainder of the region with shorter matches. This method does not guarantee a best possible alignment of the predicted LTRs, and therefore has the potential danger of missing some LTR pairs. Ideally, an alignment method that computes a combinatorially optimal alignment score is desirable to ensure that no such genuine LTR pairs are missed.

Candidate pairs are generated by the following brute-force approach: Let s denote the input genomic sequence of length n . Walk along s and for each position i , $1 \leq i \leq n$, scan all positions j such that $(i + D_{min}) \leq j \leq (i + D_{max})$. For each (i, j) -pair, compute the percentage identity of the two ω -length substrings starting at i and j . If the identity is above the similarity threshold (say 70%), then the pair (i, j) is reported as a “candidate pair” and is further evaluated for alignment as described above. The algorithm has a worst-case run-time complexity of $O(n \times (D_{max} - D_{min}) \times \omega)$. In practice, D_{max} could be as high as 10,000 - 15,000 and D_{min} could be as low as 100. In an attempt to save run-time, the algorithm’s implementation resorts to a technique of sampling the search interval, i.e., the value of i is incremented by some $\Delta i > 1$ instead of 1. This would reduce the run-time cost by a factor of Δi , but also at the expense of prediction accuracy. Moreover, this algorithm will consider many redundant or “duplicate” pairs of locations corresponding to the same matching pair of regions. To see this, note that if a 5'-3' LTR pair share a long exact match of length $l > \omega$ bp, then there are $(l - \omega + 1)$ pairs of ω -length identical substrings and the algorithm will generate all

these pairs of locations even though they correspond to the same longer exact match. Ideally, generation of such “duplicate” pairs should be completely avoided in the interest of run-time. Also note that the run-time complexity is independent of the repetitive nature of the genome, i.e., while at long stretches of the genome that have no LTRs, this algorithm would search for an entire $(D_{max} - D_{min})$ -length interval only to result in more wasted effort.

In a pilot study on a Windows machine with 1 GHz Pentium III processor conducted by one of our colleagues [Gai (2005)], *LTR_STRUC* took 3.5 hours on the entire yeast genome (over 12 *Mbp*) and over 15 hours on chromosome 1 of *Arabidopsis thaliana* genome (over 30 *Mbp*). These high run-times are likely to be a major limiting factor in the applicability of the *LTR_STRUC* software on larger genomes such as the human, maize, etc., mainly because a biologist would like to run a *de novo* prediction tool such as *LTR_STRUC* multiple times under different parameter settings before arriving at a high-quality repository of predictions.

5.2 Notation

Let s denote the input DNA sequence comprising of n nucleotides. For computational purposes, we view s as a string of n characters in alphabet $\Sigma = \{A, C, G, T, N\}$, where ‘ N ’ may denote either a low-quality or masked base in the input sequence. Let $s[i]$ denote the character at position i in s ($1 \leq i \leq n$). Let $s[i..j]$ denote the substring $s[i]s[i+1] \dots s[j]$. Let $left(i) = s[i-1]$, if $i > 1$, and ‘ N ’ otherwise; similarly, let $right(i) = s[i+1]$, if $i < n$, and ‘ N ’ otherwise. Two identical substrings $s[i_1..(i_1+k)] = s[i_2..(i_2+k)]$ are said to be *left-maximal* (respectively *right-maximal*) if and only if $left(i_1) \neq left(i_2)$ (respectively $right(i_1+k) \neq right(i_2+k)$). They are said to be a *maximal matching pair* if they are both right- and left-maximal. We will assume that aligning ‘ N ’ with any other character should be treated as a mismatch.

5.3 Our Approach

The main idea of our approach is to have an efficient linear time preprocessing of the entire input sequence, followed by an algorithm that provides a direct mechanism (as opposed

to a searching mechanism) for generation of “candidate pairs”. Our definition of “candidate pairs” is based on maximal matches subject to LTR retrotransposon length constraints. Each candidate pair is then subjected to a rigorous alignment test that guarantees an alignment with the combinatorially best score for testing against L1.

The advantage of generating candidate pairs based on maximal matches instead of fixed-length matches is that it provides a direct means of detecting a “long” exact match rather than as a chain of smaller fixed-length exact matches. While the detection of maximal matches is well studied in literature using the suffix tree data structure [Gusfield (1997b)], our pair generation algorithm follows a related strategy using the suffix array and Longest Common Prefix (LCP) array [Gusfield (1997b)] data structures, taking into account L2. The suffix array of a string is the lexicographically sorted array of all its suffixes, and the following property is key for our pair generation algorithm: any two identical substrings starting at a pair of positions can be represented as a common prefix shared by the two suffixes starting at these positions.

5.3.1 The Sequential Algorithm

Let L_{min} (L_{max}) denote the minimum (maximum) allowed length of an LTR (as shown in Figure 5.1). Let L_{ex} denote a length such that any 5'-3' pair of LTRs will share at least one exact match of that length. This user-specified parameter can be analytically estimated as follows: if ψ , the rate of mutation (as a fraction) in the host genome is known, a reasonable value is $\frac{L_{min}}{(\psi \times L_{min}) + 1}$.

Definition 3 Candidate Pair: *A pair of genomic positions (i_1, i_2) ($1 \leq i_1, i_2 \leq n$) is defined to be a candidate pair if and only if it satisfies the following properties:*

1. *the positions satisfy L2, i.e., $(i_1 + D_{min}) \leq i_2 \leq (i_1 + D_{max})$.*
2. *the substrings $s[i_1 .. (i_1 + L_{ex} - 1)]$ and $s[i_2 .. (i_2 + L_{ex} - 1)]$ are left-maximal.*

Note that there is a one-to-one correspondence between the set of maximal matching pairs of minimum length L_{ex} and left-maximal pairs of length L_{ex} . Our algorithm comprises of three phases: a preprocessing phase, a candidate pair generation phase, and an alignment phase.

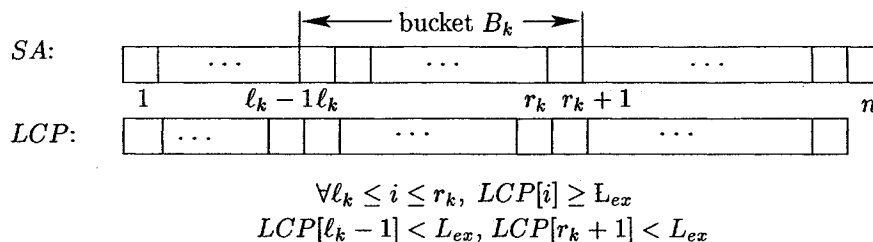


Figure 5.2 Illustration of the process of creating a bucket B_k during preprocessing.

5.3.1.1 Preprocessing

The goal of preprocessing is to “arrange” the positions $\{1, 2, \dots, n\}$ in s in a manner that allows quick generation of candidate pairs as per Definition 3. This is achieved in two steps — (i) partition the positions based on their L_{ex} -length substrings and then internally subpartition them based on the character preceding each position so that any two left-maximal substrings are in different subsets, and (ii) sort the positions within each subset so that the check for L2 can be quickly performed. The algorithm is as follows.

In the first step, construct a suffix array (denoted by SA) data structure [Manber and Myers (1993)] on s in linear time [Karkkanen and Sander (2003); Kim *et al.* (2003); Ko and Aluru (2003)] and also the corresponding longest common prefix array (denoted by LCP) [Kasai *et al.* (2001)]. As a result, $SA[i]$ is the i^{th} lexicographically smallest suffix in s ($\forall 1 \leq i \leq n$), and $LCP[i]$ is the length of the longest common prefix between suffixes $SA[i]$ and $SA[i + 1]$ ($\forall 1 \leq i \leq n - 1$). Next, a set $B = \{B_1, B_2, \dots, B_m\}$ of m buckets is generated such that $\forall i, j \in B_k, \forall 1 \leq k \leq m, s[i .. (i + L_{ex} - 1)] = s[j .. (j + L_{ex} - 1)]$. This is achieved by linearly scanning the $LCP[1..n - 1]$ array and recording all maximal intervals in which the LCP values are all greater than or equal to L_{ex} . The value of m is therefore the number of such maximal intervals. For each maximal interval the set of all suffix entries that it covers in the array $SA[1..n]$ is then assigned to a unique bucket in B . See Figure 5.2 for an illustration. Because every LCP entry covers two consecutive suffix entries in SA , each resulting bucket contains at

Algorithm 7 *Candidate Pair Generation*

Input: Bucket B_k
 L_1 : FOR EACH $c_1 \in \Sigma$ DO
 L_2 : FOR EACH $i \in Lset_{c_1}^k$ DO
 L_3 : FOR EACH $c_2 \in \Sigma$ and $(c_1 \neq c_2$ or $c_1 = c_2 = 'N')$ DO
 S_1 : $b_i \leftarrow \min\{j \mid j \in Lset_{c_2}^k, D_{min} \leq (j - i) \leq D_{max}\}$
 S_2 : $e_i \leftarrow \max\{j \mid j \in Lset_{c_2}^k, D_{min} \leq (j - i) \leq D_{max}\}$
 S_3 : Generate pairs (i, j) , $\forall j \in Lset_{c_2}^k, b_i \leq j \leq e_i$

Figure 5.3 Algorithm to generate candidate pairs from a given bucket B_k .

least two suffix entries, i.e., $0 \leq m \leq \lfloor \frac{n}{2} \rfloor$. Choosing maximal intervals in the LCP array with values $\geq L_{ex}$ ensures that $\forall 1 \leq k \leq m, \forall i \in B_k$, all substrings $s[i..(i + L_{ex} - 1)]$ are identical.

The next step is to sort each bucket in ascending order of the position numbers. This is done once for all buckets through a stable integer sort. Each bucket B_k is then further partitioned into $|\Sigma|$ ordered sets called *Lsets*: $\forall c \in \Sigma, Lset_c^k = \{i \mid left(i) = c, i \in B_k\}$. It is easy to see that one can partition every B_k into these individual *Lsets* still maintaining the internal sorted order within each *Lset*. Maintaining the sorted order is critical for efficient generation of candidate pairs, as will soon become evident.

5.3.1.2 Candidate Pair Generation

Once the input sequence is preprocessed, candidate pairs can be generated from within each bucket. The algorithm for generating candidate pairs is presented in Figure 5.3, and an illustration to help understand the algorithm is provided in Figure 5.4.

For each bucket B_k , all *Lsets* are scanned in the ascending order of the position number. A position i in $Lset_{c_1}^k$ is paired with a position j if and only if $j \in Lset_{c_2}^k$ such that $c_2 \neq c_1$ or $c_2 = 'N'$ (i.e., the substrings $s[i..(i + L_{ex} - 1)]$ and $s[j..(j + L_{ex} - 1)]$ are left-maximal), and $(i + D_{min}) \leq j \leq (i + D_{max})$ (i.e., the pair (i, j) satisfies L2). This guarantees that a pair (i, j) is generated only if it is a candidate pair by Definition 3. Enumerating all j (and only those j) that should be paired with i is achieved in the following manner. Since each

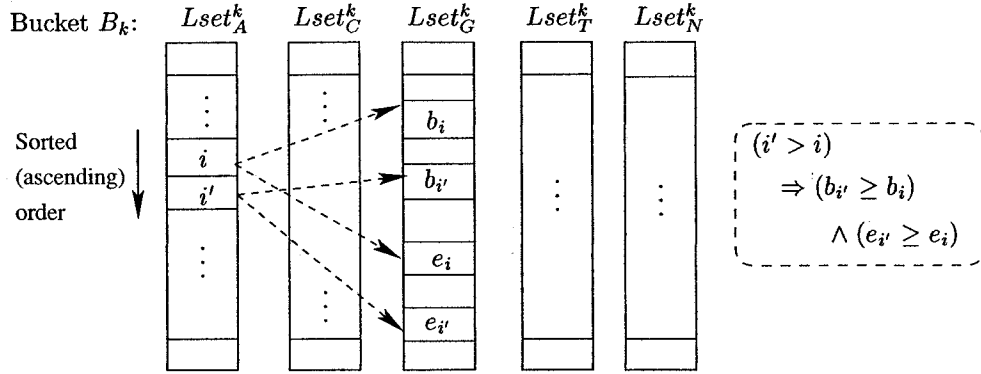


Figure 5.4 Illustration of the candidate pair generation algorithm.

$Lset$ is internally sorted by position numbers, the valid entries for j for a given value of i will be placed consecutively in $Lset_{c_2}^k$, defined by a range, say $[b_i, \dots, e_i]$. If i is the first entry of the ordered set $Lset_{c_1}^k$, b_i can be located in $Lset_{c_2}^k$ by performing a linear scan until a value that is $\geq (i + D_{min})$ is encountered. Once b_i is located, we can continue pairing i with all subsequent elements from b_i in $Lset_{c_2}^k$ until $(i + D_{max})$ is exceeded or the $Lset$ is exhausted. The last element to be paired is e_i . Henceforth, in advancing each i to its next position say i' in $Lset_{c_1}^k$, it is sufficient to start searching for $b_{i'}$ from b_i onwards, since $b_{i'} \geq b_i$ as $i' > i$. Even better, one can record the location of $b_{i'}$ if it is found before e_i , while generating pairs for i , and directly start from $b_{i'}$ for i' .

Since the algorithm ensures every entry in each bucket is considered for i , and that for each such i all valid entries for j from the same bucket are considered, it can be seen that our candidate pair generation does not miss any candidate pair satisfying Definition 3. Moreover, since each entry in a bucket is considered for i exactly once it is also easy to see that each candidate pair is generated exactly once.

Lemma 4 Let $s[i_1..(i_1 + k - 1)]$ and $s[i_2..(i_2 + k - 1)]$ be two maximal matching substrings, for some $k \geq L_{ex}$, and $(i_1 + D_{min}) \leq i_2 \leq (i_1 + D_{max})$. Then (i_1, i_2) is generated exactly once.

Proof: If $s[i_1..(i_1 + k - 1)]$ and $s[i_2..(i_2 + k - 1)]$ are two maximal matching substrings of length $\geq L_{ex}$ then there is exactly one pair of left-maximal L_{ex} -long substrings: $s[i_1..(i_1 + L_{ex} - 1)]$

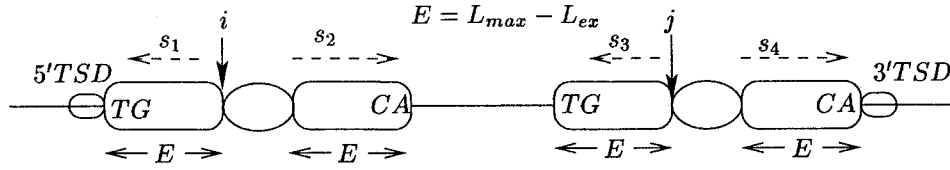


Figure 5.5 Two alignments are performed for each candidate pair (i, j) : s_1 vs. s_3 and s_2 vs. s_4 . Dotted arrows indicate the directions of the alignments, and the two ovals indicate the anchoring match.

and $s[i_2..(i_2 + L_{ex} - 1)]$. Therefore (i_1, i_2) is a candidate pair by Definition 3 and is generated exactly once by the algorithm. ■

5.3.1.3 Run-time Analysis

For the preprocessing phase, the construction of suffix array [Karkkanen and Sander (2003); Kim *et al.* (2003); Ko and Aluru (2003)] and LCP array [Kasai *et al.* (2001)] take $O(n)$ time. Generating the set of buckets also takes $O(n)$ time because the algorithm performs a linear scan of the arrays. Sorting each bucket by position numbers and generating all $Lsets$ for all buckets are integer sorting operations. The outermost loops, L_1 and L_2 of Algorithm 5.3, over all iterations visit each position in $\{1, \dots, n\}$ at most once, although in an arbitrary order. By Lemma 4, the cost of Step S_3 over all iterations is proportional to the number of candidate pairs generated. For steps S_1 and S_2 , note that at worst case, locating a particular b_i may take $O(n)$ if it is the first entry in its $Lset$. However, the amortized worst case total cost is still $O(n)$ because each entry is considered exactly once for choice of i and at most $2 \times |\Sigma|$ times for j , implying a run-time cost of $O((2 \times |\Sigma| + 1) \times n) = O(n)$ (taking $|\Sigma| = 5$ to be a constant). Thus the pair generation algorithm has an optimal run-time, i.e., $O(n)$ plus the number of candidates pairs in s .

5.3.1.4 Alignment and LTR Prediction

Once a candidate pair is reported, the regions flanking the corresponding match are evaluated to check if the aggregate region indeed has an expected LTR structure. This is achieved by computing an alignment as follows: For each candidate pair (i, j) , four substrings each of length $L_{max} - L_{ex}$ are extracted as indicated in Figure 5.5, following which two alignments are computed — one alignment between $s[i + L_{ex} .. i + L_{max} - 1]$ and $s[j + L_{ex} .. j + L_{max} - 1]$, and another alignment between the reverse of $s[i - (L_{max} - L_{ex}) .. i - 1]$ and the reverse of $s[j - (L_{max} - L_{ex}) .. j - 1]$. We use standard dynamic programming techniques for computing an optimal global alignment score between two sequences using affine gap penalties [Gotoh (1982)]. In order to save run-time, alignment computation is restricted over a band of diagonals while ensuring the optimality of score. Once the alignments are computed, an aggregate alignment score is calculated by adding the scores of the best aligning prefixes in the two computed alignments plus the matching score of the anchored match in the middle. If this aggregate score satisfies L1, the boundaries of the two aligning regions is reported as a *predicted pair of LTRs*.

As part of the above outlined alignment method, we also account for the presence of TSDs and LTR motifs. TSDs are detected by looking for an exact match of length 5-6 *bp* in the left and right vicinity of the predicted 5' and 3' LTRs, respectively (as shown in Figure 5.5). Also, the 5' and 3' ends of each of the two LTRs and their vicinity are searched for a presence of the motifs *TG* and *CA* respectively. Along the process of this search for motifs and TSDs, the alignment boundaries are adjusted between the predicted pair of LTRs.

In order to be able to distinguish among the predictions made by our algorithm based on the presence and absence of LTR structural signals, we associate a “confidence level” and report it as part of each prediction. The confidence level for each prediction is given by the following formula:

$$Confidence\ Level = Weight_{TSD} \times TSD_{code} + Weight_{motif} \times Motif_{code}$$

where $0 \leq Weight_{TSD}, Weight_{motif} \leq 1$ are weights assigned by the user to specify the relative importance of the presence of identical TSDs and motifs; note that $Weight_{TSD} +$

TSDs	5' Motif (<i>TG</i>)	3' Motif (<i>CA</i>)	Confidence Level
Identical	Present	Present	1.0
Identical	Present	Absent	0.75
Identical	Absent	Present	0.75
Identical	Absent	Absent	0.5
Not Identical	Present	Present	0.5
Not Identical	Present	Absent	0.25
Not Identical	Absent	Present	0.25
Not Identical	Absent	Absent	0.0

Table 5.1 Confidence levels for different scenarios depending on the presence or absence of TSDs and motifs.

$Weight_{motif} = 1$. For example, if the presence of the motifs in both LTRs is only half as important as the presence of identical TSDs, then the values can be: $Weight_{motif} = 0.33$ and $Weight_{TSD} = 0.67$. For a given prediction: TSD_{code} is set to 1 if the two predicted TSDs are identical, and 0 otherwise; and $Motif_{code}$ is set to 1 if both 5' and 3' LTRs start and end with *TG* and *CA* respectively, 0.5 if only one motif is found, and 0 otherwise. Given weights of 0.5 for both $Weight_{TSD}$ and $Weight_{motif}$, Table 5.1 shows the different confidence levels and their meanings.

5.3.2 Parallelization

The sequential algorithm presented in the previous section is parallelized in the following manner: The input sequence can be distributed evenly across processors. To ensure that no pairs are missed, the last $D_{max} - 1$ characters in the local portion of the input are duplicated as a prefix in the local portion of the next processor, resulting in at most $\frac{n}{p} + D_{max} - 1$ characters per processor. Once distributed, each processor can run the serial algorithm on its local portion of the input without needing to further communicate. The speedup of the preprocessing phase is proportional to $\frac{n}{\frac{n}{p} + D_{max}}$, thereby achieving a linear speedup as long as $D_{max} \ll \frac{n}{p}$. The speedup of the candidate pair generation and alignment phases are however highly dependent on the input, and the distribution of the repetitive elements among processors. Although one can dynamically balance this workload, our current implementation does not support such

Parameter Name	Default Value	Comment
D_{min}, D_{max}	100, 15000	Distance constraints (L2) for 5'-3' LTR pair
L_{min}, L_{max}	100, 1000	Length constraints for 5'-3' LTR pair
L_{ex}	20	Exact match length requirement for 5'-3' LTR pair
τ	75%	Similarity threshold of a 5'-3' LTR pair (L1)
match	2	Match score
mismatch	-5	Mismatch score
open_gap	6	Gap opening penalty
continuation_gap	1	Gap continuation penalty
$Weight_{TSD}$	0.5	Weight for presence/absence of TSD
$Weight_{motif}$	0.5	Weight for presence/absence of the motif <i>TG...CA</i>

Table 5.2 Parameter set for our program with default values.

features.

5.3.3 Software Availability

We developed a software program called *LTR_par* that implements our LTR retrotransposon detection algorithm. The implementation is in C and can be run either serially or on multiprocessor computers and clusters with support for MPI (e.g., MPICH [Gropp *et al.* (1996)]). The software is available free for academic use.

5.4 Results

5.4.1 Quality Validation

Validation of our software was performed by running the program on the entire yeast genome and comparing the results against a “benchmark” of known LTR retrotransposon locations ([Kim *et al.* (1998)], see the URL <http://www.public.iastate.edu/~voytas> for more details). The list of parameters and their values used while running our *LTR_par* software is shown in Table 5.2.

The yeast genome has 16 chromosomes, and the benchmark has a total of 50 known full-length LTR retrotransposons. *LTR_par* predicted a total of 191 elements with different confi-

Chromosome	<i>LTR_par</i>			<i>LTR_STRUC</i>		
	TP	FP	FN	TP	FP	FN
1	1	0	0	1	0	0
2	3	0	0	2	0	1
3	1	0	1	1	0	1
4	7	0	1	1	2	7
5	1	1	1	1	0	1
6	1	0	0	1	0	0
7	6	3	0	5	0	1
8	2	0	0	2	0	0
9	1	0	0	1	0	0
10	2	0	0	2	0	0
11	0	0	0	0	0	0
12	5	0	1	4	0	2
13	3	1	1	3	0	1
14	3	0	0	2	0	1
15	3	0	1	3	0	1
16	5	0	0	5	0	0
Total	44	5	6	34	2	16

Table 5.3 Quality validation of running *LTR_par* and *LTR_STRUC* programs on the entire yeast genome.

dence levels: 49 with a confidence level of 1, 11 with a level of 0.75, 44 with a level of 0.5, 57 with a level of 0.25, and 30 with a level of 0. We extracted the 49 predictions with confidence level 1, and evaluated them against the benchmark entries as follows. Each prediction made by *LTR_par* is categorized as a “true positive” if the prediction is part of the benchmark, and a “false positive” otherwise. Those retroelements that were not part of our prediction are labeled “false negative”. The results are shown in Table 5.3, listed by each chromosome.

All the 44 true positives accurately predicted the LTR boundaries along with their TSDs and motif locations. Of the 6 false negatives, 3 were not identified because they do not have identical TSDs; all these 3 were however accurately reported at a lower confidence level (0.5). Of the remaining three, one was not identified because of a low LTR sequence identity (69%) and another was identified with a lower confidence (0.5) because of boundary mis-predictions resulting from its computed optimal alignment not matching the “biologically-preferred” LTR

boundaries in the benchmark entry. *LTR-par* identified the last false negative entry although with its predicted LTR boundaries inconsistent with that of the benchmark entry; however, this approximate prediction was made with a confidence of 1.

Of the 5 false positives, 3 of them were LTRs part of other full-length retroelements but reported because they were similar and proximate along the genome. We invalidated these candidates by ensuring that there is no known reverse transcriptase coding sequences intervening the predicted 5' and 3' sequences (by running *tblastx* [Altschul *et al.* (1990)]). Of the remaining two false positives, one shares its 3' LTR with that of a true positive prediction, while the 5' is different. We speculate that this is a nested retrotransposon, although further investigation is required to validate this claim. The last false positive is same as the last false negative case we discussed above — the 3' LTR (334 *bp* long) matches accurately with that of the benchmark; however, instead of the 5' LTR (140 *bp* long) reported in the benchmark, our prediction has a longer 5' prediction that is 338 *bp* long, covering the benchmark's 5' region. The fact that there is a 5' LTR that matches closely in length and sequence identity with that of the 3' LTR (along with a pair of identical TSDs and motifs as predicted by *LTR-par*) suggests that the length discrepancy between this LTR pair in the benchmark record is probably outdated with respect to the current sequence in GenBank.

For comparison purposes, we also ran the *LTR-STRUC* program on the yeast genome and compared its results against the benchmark. The program was run in its default parameter settings (which has a similarity threshold of 75%), and at the highest level of “thoroughness” permitted by the program [McCarthy and McDonald (2003)]. These results are also shown in Table 5.3. Of the total 16 false negatives, the program misses all the three elements with non-identical TSDs (note that *LTR-par* identifies these as low confident predictions). As expected, the program also misses the retroelement with 69% LTR sequence identity. We could not ascertain the reason(s) for missing of the remaining 12 LTR pairs. It is likely that the program either failed to generate candidate pairs because of jumping by Δi characters as a means to save run-time or that an alignment that was inferior to a best alignment was computed on aligning them.

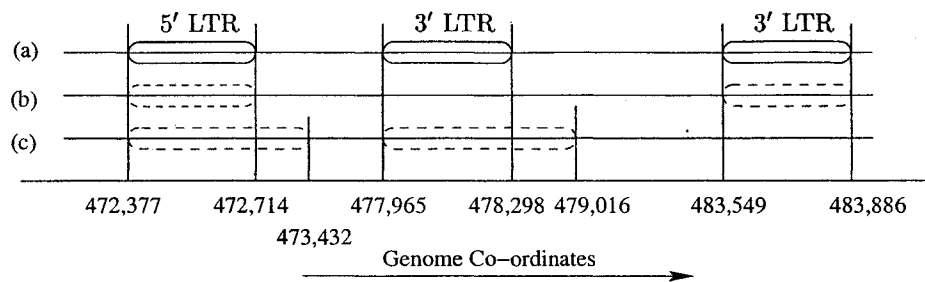


Figure 5.6 / A case of nested retrotransposons in chromosome 10 of *S. cerevisiae* with 3 LTRs. The bottom-most line indicates the genome (not to scale). Part (a) shows the benchmark co-ordinates for the LTRs. Parts (b) and (c) show the two *LTR_par* predictions.

The above results show that *LTR_par* has a better sensitivity than *LTR_STRUC*, while *LTR_STRUC* has a better specificity than *LTR_par*. For a *de novo* prediction program, while it is important to keep the number of false predictions low in the interest of saving further validation efforts, it is more important to have high sensitivity because a missed prediction cannot be found through subsequent post-processing of the program's output. As for the false predictions made by *LTR_par*, we observed that most of these predictions are due to generic repeats that have both high sequence identities and genomic proximities. In addition to offering a higher sensitivity, the scheme of predicting at different confidence levels provides additional flexibility in handling false predictions. For instance, in case of the above results on the yeast genome, even though a total of 191 predictions were reported by *LTR_par*, 44 of the total 50 retroelements were predicted with a confidence of 1, while a majority of these false predictions were reported at lower confidence levels. This allows a user to evaluate the predictions in the order of confidence reported.

There was also a case of "nested" retrotransposon in the benchmark data set along chromosome 10. Figure 5.6 shows this case, where one 5' LTR is shared between two full-length retrotransposons. As illustrated in the figure, our software also predicted the two retrotransposons, one of which with consistent boundary and TSD predictions as well.

Besides the yeast genome, we also ran *LTR_par* on a collection of 9 rice BAC sequences, randomly selected from a larger set of rice BACs analyzed using *LTR_STRUC* by McCarthy *et*

Organism	Genome Size (in <i>bp</i>)	Number of processors	Total Time (in minutes)
<i>Saccharomyces cerevisiae</i>	12,070,811	8	1.2
<i>Arabidopsis thaliana</i>	119,186,497	32	67
<i>Drosophila melanogaster</i>	118,357,599	32	33
<i>Pan troglodytes</i>	3,084,092,060	50	491

Table 5.4 Run-time results of *LTR_par* on different genomes.

al [McCarthy *et al.* (2002)]. Both the programs detected 8 full-length LTR retrotransposons in common. However, *LTR_par* detected 4 predictions that were absent in the *LTR_STRUC*'s list of predictions. On the other hand, *LTR_STRUC* predicted 2 solo-LTRs (i.e., non full-length elements) which were not predicted by *LTR_par*.

5.4.2 Performance Results

As for run-time on the yeast genome (over 12 *Mbp*), *LTR_STRUC* took about 210 minutes on a Windows Intel Pentium III 1 GHz machine, while *LTR_par* took 10 minutes on a single Intel Pentium III 1.1 GHz processor. While *LTR_par* spends much less time on candidate pair generation than *LTR_STRUC*, it spends most of its time in performing alignments simply because it does more work to guarantee optimality. For example, on the yeast genome, *LTR_par* spent only 8% of the time in preprocessing and generating pairs, while the remaining 92% was spent in aligning the LTR candidates. This extra effort spent in ensuring a thorough alignment is supported by a better sensitivity of our software when compared to *LTR_STRUC*, as was seen in the above validation studies. We also studied the performance of *LTR_par* on a Linux cluster of 25 nodes, each with 2 Intel Xeon 3.06 GHz processors and 2 GB RAM. The parallel run-times taken by *LTR_par* for genomes of different sizes are shown in Table 5.4.

In order to assess the parallel scalability of our current implementation, we ran *LTR_par* on different number of processors by keeping the input size fixed. Table 5.5 shows these results on the entire yeast genome (11 *Mbp*) and on the chromosome 3R (27 *Mbp*) of the *Drosophila* genome. As can be observed in both cases, the parallel efficiency decreases with increase in the

Input	Number of processors					
	1	2	4	8	16	32
Yeast Genome	226	150	96	71	53	40
Chromosome 3R	820	598	455	294	270	255

Table 5.5 Parallel run-times (in seconds) of *LTR_par* on the yeast genome and the chromosome 3R of *Drosophila*.

number of processors. This is expected because the current implementation does not distribute alignment workload across processors, i.e., a candidate pair generated on a given processor is aligned on the same processor, regardless of the repetitive nature of the portion of genome assigned to the processor. Thus the parallel bottleneck is the processor with the maximum alignment work.

5.4.3 A Large-Scale Application

In order to validate the applicability of the software on newly sequenced genomes, we ran *LTR_par* on the entire chimpanzee (*Pan troglodytes*) genome [Sequencing and Consortium (2005)]. The chimpanzee genome comprises of 23 pairs of autosomal chromosomes and 2 pairs of sex chromosomes. The sequence data downloaded from GenBank as of September 2005 has over 3 billion *bp*. On 50 processors of the Linux cluster described above the program took under 8.5 hours to complete on the entire genome. On the longest chromosome (≈ 229 *Mbp* long chromosome 1) the program took about 27 minutes, while the longest run-time was 58 minutes on chromosome 4 (≈ 209 *Mbp* long). Running on this genome takes a week to 10 days using LTR_STRUC [Polavarapu (2005)].

As part of an ongoing research initiative, a team at Georgia Institute of Technology is working on identification of full-length LTR retrotransposons in the chimpanzee genome [Polavarapu *et al.* (2006)]. Recently, the team identified a set of full-length retroelements using a custom-developed framework that performs an extensive search for LTRs accompanied by other important intervening patterns such as known reverse transcriptase sequences, primer binding site, poly-purine tract, etc. Due to its elaborate treatment, the full-length elements detected by this

procedure are expected to be highly accurate and conservative; so we used the resulting set as a benchmark for our studies and performed a case study on a randomly chosen chromosome (chromosome 12 which is ≈ 135 *Mbp* long).

The benchmark set for chromosome 12 comprised of 19 full-length elements. Under the default parameter settings in Table 5.2, our program predicted only 7 of the 19. When the similarity threshold (τ) was decreased to 70% and L_{max} was increased 2,000 *bp*, 12 of the 19 predicted correctly. Note that only 3 of these 12 were at confidence level 1. Of the remaining 7 not predicted by *LTR-par*, 5 were predicted when the similarity threshold was further reduced from 70% to 60%. The remaining two were not predicted because one has two LTRs of largely differing lengths (658 *bp* and 960 *bp*) and another has less than 50% sequence similarity between the LTRs. In addition to the benchmark hits, *LTR-par* predicted a total of 895 elements, including 38 at confidence level 1. Upon investigation of randomly chosen predictions, we found that many candidates do not contain any known reverse transcriptase sequences. However, the 38 predictions with confidence 1 show promise and need further validation.

5.5 Discussion

The results of validating our software are encouraging. The sensitivity on the yeast genome is better than that of the *LTR-STRUC* because our algorithm more accurately models mutation events in LTR and TSD regions. Moreover, *LTR-par* offers good flexibility by providing the user with a better control — the user can assign weights to the presence/absence of TSDs and *TG...CA* motifs, and the software can output its predictions at different confidence levels reflecting the weights specified by the user. Also, if a user is searching a newly sequenced genome for LTR retroelements, the user can try different combinations of weights and scoring parameter values and observe changes in the predictions before deciding on an appropriate set of parameters. The speed of our software plays a critical role in facilitating multiple such experiments under different parameter settings. The software can also be used to identify nested retroelements; cases that correspond to multiple nested insertions can be detected by running our software iteratively on the genomic sequence with all the full-length elements found

in previous iterations excised out.

Given that the current version of the software accounts only for the structural attributes such as TSDs and motifs, we recommend using our software for *de novo* full-length LTR retrotransposon prediction on genomes in which the two LTRs of each retrotransposon are expected to be highly conserved. The specificity of the current state of our software can be extended to incorporate other structural attributes typical of an LTR retrotransposon: The genomic region between a pair of LTR sequences typically contains special-purpose sequences such as PPT, PBS, *gag*, *pol*, and *env*, and detecting these patterns is important in confirming the biological identity of each prediction. Poly-Purine Tract can be detected by searching for a purine-rich (bases A/G) region of an approximate length of 10 *bp* immediately upstream of the predicted 3' LTR boundary. Similarly, the region immediately downstream of the predicted 5' LTR sequence can be searched for presence of Primer Binding Site. PBS is usually a complement of a known tRNA 3' terminal sequence — a pattern that can be input by the user. The genes in the *gag* and *env* regions can be detected by looking for sequences that encode retroviral capsid and envelope proteins respectively. The genes in the *pol* region can be searched for sequences that encode for protease, integrase and reverse transcriptase enzymes.

5.6 Concluding Remarks

In this chapter, we provided efficient algorithms and software towards detection of full-length LTR retrotransposons. One salient feature of our method is a space- and time-efficient algorithm for generating candidate LTR pairs, which facilitates use of rigorous methods for aligning the candidates in order to ensure high quality LTR predictions. The software has been designed with the intent of giving a high degree of flexibility to the user. The various planned functional improvements to the software, such as incorporation of detection strategies for PPT, PBS, *gag*, *pol* genes in the structure finding procedure, should strengthen the specificity of the software. Due to the ubiquity of LTR retroelements in complex eukaryotic genomes, developing a highly accurate and fast LTR retrotransposon discovery tool can significantly advance the state of knowledge in retrotransposon research topics.

CHAPTER 6. SCAFFOLDING GENOMIC CONTIGS USING LTR RETROTRANSPOSONS

The abundance of LTR retrotransposons in several eukaryotic genomes have traditionally been viewed as a source of complication in their assembly. In this chapter, we present a novel approach [Kalyanaraman *et al.* (2006a)] that provides an alternative — a method that can exploit the presence of these repeat elements in genomes and provide valuable information for performing one stage of genome assembly.

Hierarchical sequencing [Consortium (2001)] is being used to sequence the maize genome [NSF (2005)]. In this approach, a genome is first broken into numerous smaller BAC clones, each of size up to 200 *kb*. Next, a combination of these BACs that provide a minimum tiling path based on their locations along the genome is determined. Each selected BAC is then individually sequenced using a shotgun approach that generates numerous short (≈ 500 -1,000 *bp* long) *fragments*. The problem of assembling the target genome is thereby reduced to the problem of computationally assembling each BAC from its fragments.

The fragments generated by a shotgun experiment approximately represent a collection of sequences originating from positions distributed uniformly at random over each BAC. As with a jigsaw puzzle, the idea is to generate fragments such that each genomic position is expected to be covered (or sampled) by at least one fragment — and also ensuring that there is sufficient computable evidence in the form of overlaps between fragments to carry out the assembly. Regardless of the coverage specified, however, gaps invariably occur during sequencing, i.e., it cannot be guaranteed that every position is covered by at least one fragment. Coverage affects the nature of gaps — a low coverage typically results in several long gaps, while a high coverage results in fewer and shorter gaps. Because of gaps, assembling a set of fragments

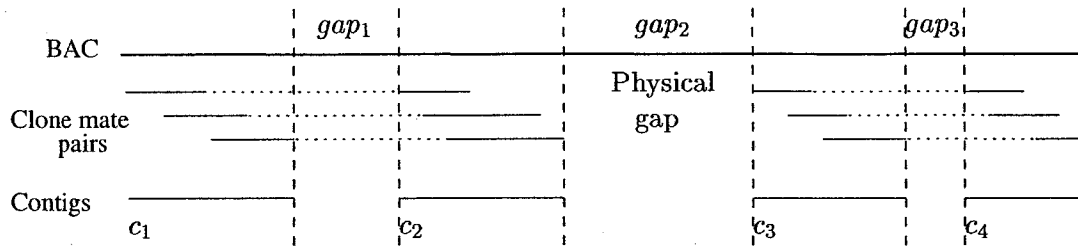


Figure 6.1 An example showing 6 pairs of clone mate fragments (shown connected in dotted lines) sequenced from a given BAC. The relative order and orientation between contigs c_1 and c_2 (also, between c_3 and c_4) can be inferred from the clone mates.

sequenced from a BAC typically results in not one but many assembled sequences (or contigs) that represent the set of all contiguous genomic stretches sampled. The next step, scaffolding, is aimed at determining the order and orientation of the contigs relative to one another. Once scaffolded, the identified gaps between contigs can be filled through targeted experimental procedures called *pre-finishing* and *finishing*. For simplicity, we use the term “finishing” to collectively refer to both these procedures.

The main focus of this chapter is the scaffolding step. The need for scaffolding arises from the fact that there could be gaps in sequencing. To be able to identify a pair of contigs corresponding to adjacent genomic stretches, current methods generate shotgun fragments in “pairs” — each BAC is first broken into smaller clones of length ≈ 5 *kbp*, and each such clone is sequenced from both ends thereby producing two fragments which are referred to as *clone mates* (or a *clone pair*). During scaffolding, the fact that a pair of clone mates originated from the same ≈ 5 *kbp* clone can be used to impose distance and orientation constraints for linking contigs that span the corresponding fragments [Batzoglou *et al.* (2002); Huson *et al.* (2001); Jaffe *et al.* (2003); Mullikin and Ning (2003); Pop *et al.* (2004)]. Figure 6.1 illustrates an example of scaffolding contigs based on clone mate information. This technique is not, however, sufficient to link contigs surrounding gaps without a flanking pair of clone mates (*gap*₂ in Figure 6.1). Such gaps, called *physical gaps*, are typically harder to “close”, and involve costly finishing efforts. Performing a higher coverage sequencing is an effective but expensive approach to reduce the occurrences of gaps. The approach proposed in this dissertation provides an

alternative mechanism to scaffold around physical gaps as well, subject to their repeat content.

In this research, we introduce a new variant of the scaffolding problem called the *retroscaffolding* problem. The problem is to order and orient contigs based on their span of LTR retrotransposon-rich regions of the genome. This approach has the following advantages:

- It does not require clone mate information. Thus, our approach complements existing scaffolding approaches for genomes with significant LTR retrotransposon content. Also, with the advent of newer sequencing technologies [Margulies *et al.* (2005)] that do not generate clone mate information, the importance of our approach is further enhanced.
- It can be used to identify LTR retrotransposon-rich portions within the unfinished genomic regions. Such information can be useful if it is decided to not finish repetitive regions in the interest of saving costs, as is the case with the maize genome project [NSF (2005)].
- In genome projects of highly repetitive genomes, most of the sequencing and finishing efforts are expected to be spent on repeat rich regions. This is one of the main concerns in the on-going efforts to sequence the maize genome, at least 50% of which is expected to be retrotransposons. The retroscaffolding technique provides a mechanism to reduce sequencing coverage without affecting the quality of the genic portion of the final assembly, thereby providing a means to reduce the sequencing costs.

In Section 6.1, we describe the retroscaffolding idea, formulate it as a problem, and discuss the various factors that affect the ability to retroscaffold. For obtaining a proof of concept, we conducted experiments on previously sequenced maize BAC data. The results show that (i) 3X/4X coverage sequencing is suited for exploiting the data's repeat content towards retroscaffolding, (ii) retroscaffolding can yield over 30% savings in finishing costs, and (iii) with retroscaffolding it is possible to opt for a lower sequencing coverage. These and other experimental results assessing the effects of various factors on retroscaffolding are presented in Section 6.2. As part of the NSF/DOE/USDA maize genome project [NSF (2005)], we are working on applying the retroscaffolding technique to the maize data as it becomes available.

To this effect, we are developing an algorithmic framework to perform retroscaffolding as described in Section 6.3. In Section 6.4, we present the results of our experiments to assess the effect of applying both clone mate based scaffolding and retroscaffolding on maize genomic data. Various strengths and limitations of the retroscaffolding technique are discussed in Section 6.5. Given that retrotransposons are abundant in genomes of numerous plant crops yet to be sequenced (e.g., wheat, barley, sorghum, etc.), the capability of retroscaffolding to exploit this repeat content can provide a significant means to reduce sequencing and finishing costs.

6.1 Retroscaffolding

Long Terminal Repeat (LTR) retrotransposons constitute one of the most abundant classes of retrotransposons. As earlier described in Chapter 5, they are distinctly characterized in their structure by two terminal repeat sequences — one each at the 5' and 3' ends of a retrotransposon inserted in a host genome. Given that these retrotransposons are typically 10-15 *kbp* long, their flanking LTRs can also be expected to be separated by as many *bps* along the genome¹. Moreover, the LTR sequences are identical at the time a retrotransposon inserts itself into a host genome, and gradually diverge over time due to mutations. Yet, the LTRs flanking most retrotransposons are similar enough for detection. These properties form the basis of our retroscaffolding idea, as explained below.

Low coverage sequencing of a genome with significant LTR retrotransposon content is likely to result in a proportionately large number of gaps that span these repetitive regions. If it so happens that the sequencing covers only the two LTRs of a given retrotransposon, a subsequent assembly can be expected to have two contigs each spanning one of the LTRs. Therefore, the detection of two identical or highly similar LTR-like sequences in two contigs is a necessary (but not sufficient) indication that the contigs sample the flanking regions of an inserted retrotransposon. If this indication can be further validated to sufficiency by searching for other structural signals of an LTR retrotransposon, then the contigs can be relatively ordered and oriented (because LTRs are directed repeats). In addition, this implies that the

¹Sometimes, LTR retrotransposons can be nested within one another, accordingly affecting the distances between the 5' and 3' LTRs.

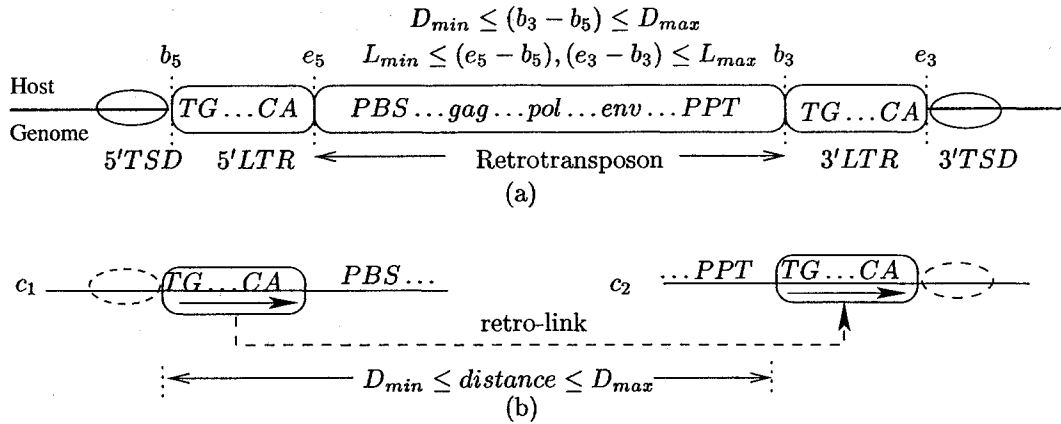


Figure 6.2 (a) Structure of a full-length LTR retrotransposon. (b) An example showing two contigs c_1 and c_2 with a retro-link between them.

intervening region between two consecutively ordered contigs contains retrotransposon related sequences — an information that can be used to prioritize the gaps for finishing, and potentially reduce efforts spent on finishing repetitive regions, if so desired.

The structure of a full-length LTR retrotransposon was described in detail in Chapter 5 (see Section 5.1). For ease of exposition, we follow the same labeling convention from L1 through L5 as introduced in Section 5.1.

For a sequence s , let $s^f = s$, and s^r denote its reverse complement. A sequence c is said to *contain* a sequence l if there exists between c and either l^f or l^r , a “good quality” semi-global alignment. Let an *LTR pair* $(l_{5'}, l_{3'})$ denote the two LTRs of a given LTR retrotransposon.

Definition of a Retro-link: Given a set L of n LTR pairs, two contigs c_i and c_j are said to be *retro-linked* if $\exists (l_{5'}, l_{3'}) \in L$ such that both c_i and c_j contain $l_{5'}$ or $l_{3'}$ or both.

Note that the same pair of contigs can be retro-linked by more than one LTR pair. An example of a retro-link between two contigs is shown in Figure 6.2b. The above definition can be easily extended to account for additional structural attributes such as L3, L4 and L5, to ensure that a retro-link indeed spans the same full-length LTR retrotransposon.

The Retroscaffolding Problem: Given a set C of m contigs and a set of retro-links, partition C such that:

- each subset is an ordered and oriented set of contigs,
- every pair of consecutive contigs in each subset is retro-linked and there is no contig that participates in two retro-links in conflicting orientations, and
- the sum of the number of LTR pairs corresponding to all used retro-links is maximized.

The retroscaffolding problem can be viewed as a new variant of the traditional scaffolding problem, which is called the *Contig Scaffolding Problem* [Huson *et al.* (2001)]. In the latter, the input is a set of contigs and a set of clone mate links, where each clone mate link corresponds to a distance and orientation constraint imposed by a pair of fragments sequenced from the same clone of a known approximate length. This is similar to the distance and orientation constraints imposed by a retro-link between the two contigs. Also, like in the retroscaffolding problem, not all clone mate links may be used in the final ordering and orientation, while the problem is to maximize the overall number of mate pair evidence corresponding to the clone mate links used in scaffolding. Therefore, the computational complexity of the retroscaffolding problem is same as that of the contig scaffolding problem, which is NP-complete [Huson *et al.* (2001)].

The effectiveness of retroscaffolding on a genome is dictated by the following factors:

LTR retrotransposon abundance: The ability to retroscaffold depends on the number of retro-links that can be established, which is limited by the number of detectable LTR retrotransposons in the genome. Note that this approach of exploiting the abundance in retrotransposons offers a respite from the traditional view that these are a source of complication in genome projects.

Presence of distinguishable LTRs: LTRs from different retrotransposons but from the same “family” may share substantial sequence similarity. Therefore, it is essential to take into account other structural evidence specific to an insertion before establishing a retro-link between two contigs. Even if the same LTR retrotransposon is present in two different locations of a genome, it can be expected that the TSDs are different because they correspond to the host genomic sequence at the site of insertion. It may still happen that a target genome contains the same family retrotransposons in abundant quantities, and other structural attributes become

Parameter Name	Default Value	Description
D_{min}/D_{max}	600/15,000 <i>bp</i>	Distance constraints between 5' and 3' LTRs (L2)
τ	70%	% identity cutoff between 5' and 3' LTRs (L1)
L_{min}/L_{max}	100/2,000 <i>bp</i>	Minimum/maximum allowed length of an LTR
Match/mismatch	2/-5	Match and mismatch scores
Gap penalties	6/1	Gap opening and continuation penalties

Table 6.1 *LTR_par* parameter settings.

less distinguishable as well. If BAC-by-BAC sequencing is used, the above situation can be alleviated by applying retroscaffolding to contigs corresponding to the same BAC (instead of across BACs). This is because the likelihood of the same family occurring multiple times at a BAC level is much smaller than at a genome level.

Sequencing coverage: Retroscaffolding targets each sequencing gap that spans an inserted retrotransposon such that its flanking LTRs are represented in two different contigs. Henceforth, we will refer to such gaps as *retro-gaps*. Given the length of such an insert ranges from 10-15 *kbp* (greater, if it is a nested retrotransposon), the coverage at which the genome is sequenced is a key factor affecting the ability to retroscaffold. If the sequencing coverage is too high (e.g., 10X), then there are likely be so few (short) sequencing gaps that the need for any scaffolding technique diminishes. Whereas at very low coverage (e.g., 1X) long sequencing gaps that span entire LTR retrotransposons are likely to prevail.

6.2 Proof of Concept of Retroscaffolding

In this section, we provide a proof of concept for retroscaffolding. For this purpose, four finished maize BACs (listed in Table 6.2) were acquired from Cold Spring Harbor Laboratory [McCombie (2005)]. The first step was to determine the LTR retrotransposon content of these BACs. *LTR_par*, which is our program for identifying LTR retrotransposons as described in Chapter 5, was used to analyze each BAC with the parameters specified in Table 6.1. Table 6.2 summarizes the findings. As can be observed, the fraction of LTR retrotransposons in these BACs averages 42%, consistent with the latter's estimated abundance in the genome.

	GenBank Accession	BAC Length (in <i>bp</i>)	Number of LTR retrotransposons	Retrotransposons in BAC Length in <i>bp</i>	% <i>bp</i>
<i>BAC</i> ₁	AC157977	107,631	3	29,578	27%
<i>BAC</i> ₂	AC160211	132,549	6	60,391	46%
<i>BAC</i> ₃	AC157776	147,470	8	73,099	50%
<i>BAC</i> ₄	AC157487	136,932	6	57,783	42%

Table 6.2 Summary of the LTR retrotransposons identified in 4 maize BACs using *LTR_par*.

The effect of sequencing at different coverages was assessed as follows. A program that “simulates” a random shotgun sequencing over an arbitrary input sequence at a user-specified coverage was acquired from Scott Emrich at Iowa State University [Emrich (2005)]. Each run of the program produces a set (or *sample*) of fragments, along with the information of their originating positions. We ran this program on each BAC for coverages 1X through 10X, and for each coverage 10 samples were collected to simulate sequencing 10 such BACs. For each sample, using the knowledge of the fragments’ originating positions, the set of all contiguous genomic stretches covered (and thereby the set of sequencing gaps) was determined. Ideally, assembling the sample would produce a contig for each contiguous stretch. Based on the placement information of the contigs on the BAC and that of the LTR pairs (Table 6.2) on the BAC, each LTR pair was classified into one of these three classes (see Figure 6.3):

- **CgC**: both LTRs are contained in two different contigs,
- **C_C**: both LTRs are contained in the same contig, and
- **GgX**: at least one LTR is not contained by any contig (i.e., it is located in a gap).

In this classification scheme, it is easy to see that retro-links can be expected to be established only for *CgC* LTR pairs. Therefore, the ratio of the number of *CgC* LTR pairs to the total number of LTR pairs is indicative of the maximal value of retroscaffolding at a given coverage. We computed this ratio for each of the 4 BACs used in our experiments, by considering one coverage at a time, and counting the LTR pairs in each of the three classes over all 10 samples. From Table 6.3, we observe that the ratio is maximum for a 3X coverage for 3

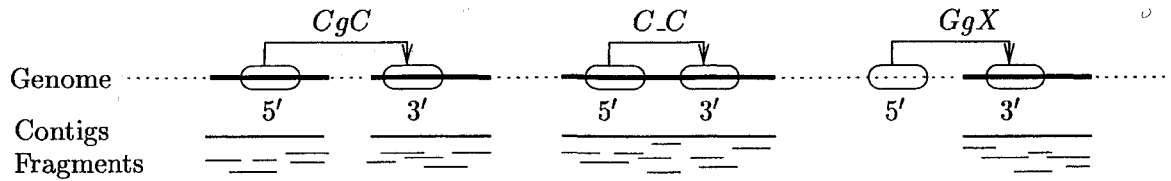


Figure 6.3 Classification of LTR pairs based on the location of sequencing gaps, LTRs, and contigs. Dotted lines denote sequencing gaps. Retro-links correspond to the class *CgC*.

Coverage	<i>BAC</i> ₁				<i>BAC</i> ₂	<i>BAC</i> ₃	<i>BAC</i> ₄
	<i>CgC</i>	<i>C_C</i>	<i>GgX</i>	<i>CgC</i> %	<i>CgC</i> %	<i>CgC</i> %	<i>CgC</i> %
1X	16	1	13	53	83	63	63
2X	26	0	4	87	95	77	92
3X	25	3	2	83	100	97	100
4X	27	3	0	90	100	88	100
5X	24	6	0	80	95	93	95
6X	22	8	0	73	83	76	98
7X	19	11	0	63	83	61	100
8X	18	12	0	60	77	64	67
9X	16	14	0	53	48	50	60
10X	7	23	0	23	37	31	43

Table 6.3 Classification of the LTR pairs in 4 BACs, with respect to a set of 10 shotgun samples obtained from each BAC at different coverages.

out of the 4 BACs, and 4X for the other BAC. This implies that a 3X/4X coverage project is expected to best benefit from the retroscaffolding approach. To understand the above results intuitively, observe that a very high coverage has a high likelihood of sequencing an LTR retrotransposon region to entirety, making retroscaffolding unnecessary; while a very low coverage results in a high likelihood of LTRs falling in gaps, making retroscaffolding ineffective. Both these expectations are corroborated in our experiments — in Table 6.3 the gradual increase in *C_C* and the decrease in *GgX* with increasing coverage. The *C_C* increase with coverage also indicates the amount of efforts spent in sequencing retrotransposon-rich regions.

In our next experiment, we assess the potential savings that can be achieved at the finishing step through the information provided by retroscaffolding on gap content. Table 6.4 shows the

Coverage	BAC_2			BAC_4		
	All gaps	Retro-gaps	%Retro-gaps	All gaps	Retro-gaps	%Retro-gaps
1X	70.5	26.4	37.4	78.0	24.8	31.8
2X	88.7	33.6	37.9	93.5	33.4	35.7
3X	84.6	32.2	38.1	84.0	31.0	36.9
4X	65.7	26.6	40.5	64.5	19.5	30.2
5X	50.6	19.3	38.1	46.4	16.7	36.0
6X	37.4	13.7	36.6	39.5	13.2	33.4
7X	28.3	9.5	33.6	26.6	9.1	34.2
8X	18.7	6.5	34.8	19.1	6.3	33.0
9X	13.0	3.0	23.1	11.9	5.9	49.6
10X	9.3	2.7	29.0	9.5	2.3	24.2

Table 6.4 Number of retro-gaps vs. all sequencing gaps. Measurements are averaged over all 10 samples of each of the two BACs.

number of gaps generated at various sequencing coverages, and the number of which can be detected using retroscaffolding (i.e., retro-gaps). While the results are shown only for two BACs, we observed a similar pattern in all four BACs. As each retro-gap corresponds to a potential region of the genome that may not necessitate finishing, the ratio of the number of retro-gaps to the total number of sequencing gaps indicates the potential savings achievable at the finishing step because of retroscaffolding. From the table we observe this ratio ranges from 23%-40% for BAC_2 , and 24%-49% for BAC_4 ; averaging over 34% savings for both BACs.

Table 6.4 also shows that sequencing BAC_2 at a 6X coverage is expected to result in ≈ 37 sequencing gaps; while sequencing at a 4X coverage and subsequently applying retroscaffolding is expected to result in an effective 39 gaps ($\approx 65.7 - 26.6$). This implies that through retroscaffolding it is possible to reduce the coverage from 6X to 4X on BAC_2 without much loss of scaffolding information. As retroscaffolding can be used independent of clone mate information, we are working on evaluating the collective effectiveness of both clone mate-based scaffolding and retroscaffolding approaches. If similar results can be shown at a much larger scale of experimental data for a target genome, then retroscaffolding can be used to advocate for a low coverage sequencing, directly impacting the sequencing costs of repetitive genomes.

6.3 A Framework for Retro-linking

We developed the following two-phase approach to retroscaffolding. In the first phase, retro-links are established between contigs that show “sufficient” evidence of spanning two ends of the same LTR retrotransposon. Once retro-links are established, the process of scaffolding the contigs is the same as scaffolding them based on clone mate information, i.e., each retro-link can be treated equivalent to a clone mate pair that imposes distance and orientation constraints appropriate for LTR retrotransposon inserts. Therefore, in principle, any of the programs developed for the conventional contig scaffolding problem [Batzoglou *et al.* (2002); Huson *et al.* (2001); Jaffe *et al.* (2003); Mullikin and Ning (2003); Pop *et al.* (2004)] can be used to achieve retroscaffolding from the retro-linked contigs². In what follows, we describe our approach to establish retro-links.

There are two types of retro-links that can be established among contig data: (i) those that correspond to LTR retrotransposons that are already known to exist in the genome of the target organism or closely related species, and (ii) those that are *de novo* found in the contig data. The first class of retro-links can be established by building a database of known LTR retrotransposons and detecting contigs that overlap with LTR sequences of the same retrotransposon. However, such a database of already known LTR sequences of a target genome may hardly be complete in practice. For this reason, the second class of retro-links that are based on a *de novo* detection of LTR sequences in the contig data is preferable. However, additional validation will be necessary to ensure the correctness of such retro-links.

In what follows, we describe the algorithmic framework we developed to establish retro-links based on already known LTR retrotransposons, and the results of applying it on maize genomic data.

6.3.1 Building a Database of LTR Pairs

Given that the entire genome of maize has not yet been assembled, the first step in our approach is to build a database of maize LTR pairs from previously sequenced maize genomic

²For our experiments, we used the *Bambus* [Pop *et al.* (2004)] program.

Input	Number of sequences	Number of full-length predictions	Number of LTR pairs
LTR retrotransposons [Miguel (2005)]	560	556	556
Solo-LTRs [Miguel (2005)]	149		149
Maize BACs [Emrich (2005)]	470	1,234	1,234
		Total	1,939

Table 6.5 Summary of LTR pairs predicted by *LTR_par*.

data. A set of 560 known full-length LTR retrotransposons and 149 solo LTRs³ was acquired from San Miguel [Miguel (2005)]. In addition, a set of 470 maize BACs were downloaded from GenBank [Emrich (2005)]. Because the information about the LTR sequences within the full-length retrotransposons and BACs was not available, we used the *LTR_par* program to identify LTR retrotransposons and their location information. We did not include the LTRs identified in the four maize BACs listed in Table 6.2, so that they can be used as benchmark data for validating retroscaffolding.

Given a set of sequences, *LTR_par* identifies subsequences within each sequence that bear structural semblance to full-length LTR retrotransposons. Desired values for structural attributes can be input as parameters. We used the values shown in Table 6.1. As part of each prediction, the locations of both the 5' and 3' LTRs are output. A prediction is made only if the identified region satisfies LTR sequence similarity (L1) and LTR distance (L2) conditions. Based on the presence of other signals such as the *TG..CA* motif (L3) and TSDs (L4), each prediction is also associated with a "confidence level". A confidence level of 1 implies presence of both L3 and L4, 0.5 implies either L3 or L4 but not both, and 0 implies only L1 and L2. In this research, we use level 1 predictions, although we are currently evaluating other combinations of LTR pairs from across confidence levels. Table 6.5 shows the statistics over the resultant total of 1,939 LTR pairs.

³Solo LTRs are typically the result of a deletion/recombination event at a site of an inserted LTR retrotransposon, in which only either a 5' or a 3' LTR (or a part of it) survives.

6.3.2 An Algorithm to Establish Retro-links

Let C denote a set of m contigs generated through an assembly of maize fragments corresponding to one BAC, and let L denote the set of n LTR pairs ($n = 1,939$ in Table 6.5). Our algorithmic framework performs the following steps:

- **S1** Compute $P = \{(c, (l_{5'}, l_{3'})) \mid c \in C, (l_{5'}, l_{3'}) \in L, c \text{ contains } l_{5'} \text{ or } l_{3'} \text{ or both}\}$.
- **S2** Construct a set $G = \{G_1, G_2, \dots, G_n\}$, such that $\forall G_i \subseteq C, \forall c \in G_i, (c, (l_{5'}^i, l_{3'}^i)) \in P$.
Note that G need not be a partition of C . We call each G_i a *contig group*.
- **S3** $\forall G_i \in G$, compute $R_i = \{(c_i, c_j) \mid c_i, c_j \in G_i, c_i \text{ and } c_j \text{ are retro-linked by } (l_{5'}^i, l_{3'}^i)\}$.

A naive way to perform step S1 is by evaluating each of the $m \times n$ pairs of the form (*contig, LTR pair*), to check if a contig contains one of the LTRs. The check can be performed through standard dynamic programming techniques for computing semi-global alignments that take time proportional to the product of the lengths of the sequences being aligned. As reverse complemented forms also need to be considered, this approach involves $4 \times m \times n$ alignments in the worst case.

An alternative and faster way to detect overlapping pairs of contigs and LTRs follows from our PaCE approach discussed in Chapter 4: Instead of evaluating all pairs by alignment computation, compute alignment only for pairs that show significant promise through sufficiently long maximal matches. For this approach, we directly use the PaCE algorithm for first constructing a GST (see Section 4.3.1) for all contigs and LTRs, and detecting maximal matches between pairs (see Section 4.2.2). Since there is no clustering functionality in the current context, the order of pairs generated is irrelevant. A pair generated is always further evaluated through alignment computation. It is, however, sufficient to report and consider only those promising pairs that contain a contig and an LTR. This can be easily achieved without affecting the complexity of PaCE pair generation algorithm by extending it to keep track of another level of subpartitioning of *lsets* based on the sequence type (i.e., a contig or an LTR).

For each generated promising pair, an optimal semi-global alignment is computed. A significantly aligning pair of contig and LTR is reported directly in the output. As pairs are

output, the set G is computed as well in constant time per pair (step S2).

Steps S1 and S2 ensure that two contigs are paired if and only if they contain LTRs from the same LTR pair. To perform S3, it is therefore necessary only to establish additional structural evidence such as the presence of TSDs, PPT, PBS, and/or retrotransposon genes. The attributes to look for, however, depends on the location of the subsequences corresponding to the LTRs within the contigs — for e.g., it may not be possible to look for retrotransposon genic sequences if the LTR regions within the contigs are a suffix of one contig and a prefix of another (see Figure 6.2b). We perform S3 as follows: we concatenate each pair of contigs under consideration in each of the 4 possible orientation combinations, and run *LTR_par* on the concatenated sequence. A retro-link is established between a pair only if sufficient structural evidence is detected.

Preliminary Validations:

We validated the retro-linking algorithm on BAC_1 of Table 6.2 as follows. Shotgun fragments were experimentally sequenced at a 3X coverage of the BAC [McCombie (2005)], and were assembled [Emrich (2005)] using the CAP3 assembler [Huang and Madan (1999)]. The resulting 45 contigs were input along with the 1,939 LTR pairs (in Table 6.5) to our retro-linking program. Note that the 1,939 LTR pairs do not include the 3 LTR pairs in BAC_1 as identified by *LTR_par* (Table 6.2) — that way, the validation reflects an assessment of retro-linking under practical settings in which a target BAC sequence and its LTR pairs are unknown prior to the retroscaffolding step. The experiment resulted in 44 contig groups ($= |G|$), and upon investigation we found that most of the groups were “equivalent”, i.e., the corresponding LTR pairs share a significant sequence identity ($> 95\%$). The equivalent groups were merged.

The subsequent step was to evaluate each contig pair of a merged group for a valid retro-link. For detecting retrotransposon genic sequences in contigs, we queried the contigs against the GenBank *nr* database using the *tblastx* program. Other structural attributes were detected using *LTR_par*. This step resulted in only two retro-linked pairs: $(c_{10}^r \rightarrow c_{16})$, and $(c_{24} \rightarrow c_{41})$ with the arrows implying the order in which the contigs can be expected to occur along the “unknown” BAC sequence (BAC_1) in the specified orientations. We verified the predictions

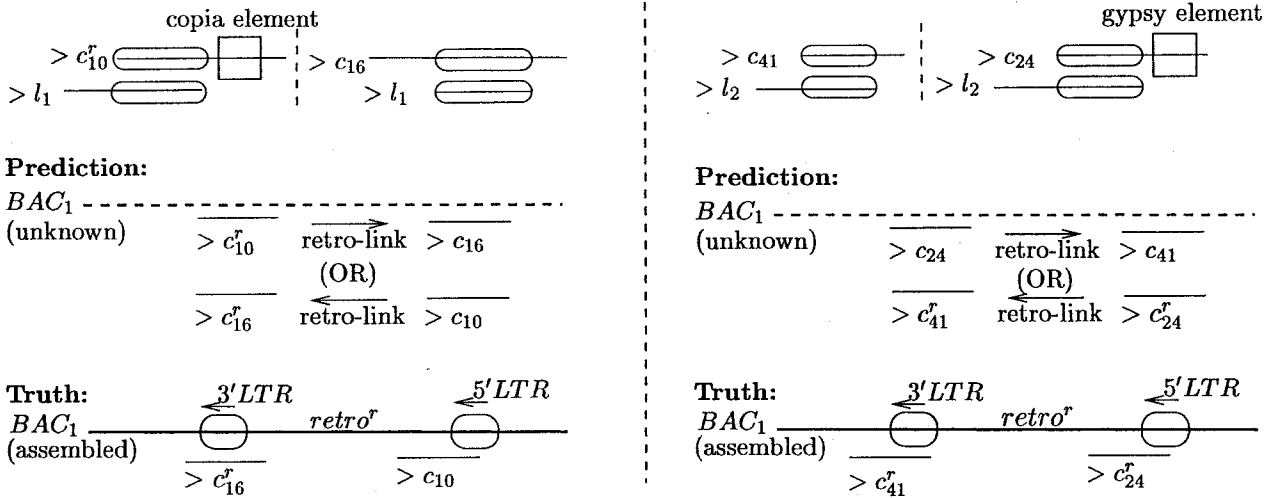


Figure 6.4 Validation of two retro-links — between contigs c_{10} and c_{16} , and contigs c_{41} and c_{24} . Vertically aligned ovals denote overlapping regions, and squares denote retrotransposon hit through *tblastx* against the GenBank *nr* database.

by aligning each of these 4 contigs directly against the known sequence of BAC_1 and found that the retroscaffolding prediction is correct (see Figure 6.4).

6.4 Scaffolding with Clone Mates and Retro-links

Retroscaffolding differs from conventional contig scaffolding as it relies on the presence of LTR retrotransposons instead of the clone mate information. While this suggests that either of the techniques can be applied independent of one another, the results may themselves be not mutually exclusive — i.e., it is possible that the relative ordering and orientation between the same two contigs are implied by both the techniques. While such redundancies in output can be used as additional supporting evidence for bolstering the validity of scaffolding, the actual value added by either of these two techniques is dictated by its respective unique share in output scaffolding. Ideally, we would hope that these two results to complement one another.

We assessed the effect of a combined application of retroscaffolding and clone mate based scaffolding on maize genomic contig data as follows: 62 contigs were generated by performing a CAP3 assembly over a 3X coverage set of fragments sequenced from BAC_4 . Ideally, all 62

	Clone mate scaffolding	Retroscaffolding	Combined scaffolding
Number of scaffolds	32	5	27
Total span of scaffolds (<i>bp</i>)	120,350	65,605	138,356
Average span of scaffold (<i>bp</i>)	3,760	6,246	4,457
Number of contig pairs scaffolded	42	10	71
Number of assembly gaps covered	22	17	28

Table 6.6 Results of (i) scaffolding contig data for BAC_4 (136,932 *bp*) using clone mate information, (ii) retroscaffolding, and (iii) combined scaffolding using both clone mate and retro-link information.

contigs would be part of just one “scaffold” if the contigs were all to be ordered along the target BAC.

The scaffolding achievable from just the clone mate information was first assessed by running the Bambus [Pop *et al.* (2004)] program on the contigs. This resulted in 32 scaffolds spanning an estimated total of 120,350 *bp* and each with an average span of 3,760 *bp*. (Note that the “span” of a scaffold output by Bambus is only an estimate, because it includes the size estimated for sequencing gaps between the scaffolded contigs.) We then assessed the scaffolding achieved by retroscaffolding the contig data — retro-links were first established using the framework described in Section 6.3 and the output was transformed as input to Bambus. While retroscaffolding resulted in many fewer scaffolds (5), the total span was smaller (65,605 *bp*) when compared to clone mate scaffolding. However, the average span of each scaffold was almost twice as large in retroscaffolding. This is as expected because the distance constraint used for each retro-link was longer ([5000, 15000]) than that of clone mate links ([2200, 3800]).

In the next step, we input both the retro-link and clone mate information with their respective distance and orientation constraints to Bambus. This combination resulted in fewer scaffolds (27) and a longer total span (138,356 *bp*) than was achieved by just clone mate scaffolding — implying that retroscaffolding provides added information that is not provided by clone mate information. The above results are summarized in Table 6.6. The table also shows the number of contig pairs scaffolded as a result of the respective scaffolding strategies;

the higher this number is, the more inclusive scaffolding is on the contigs — ideally, we would expect all contigs to be in one scaffold thereby implying $\binom{62}{2}$ contigs pairs.

We also assessed the individual effect of these scaffolding techniques on “assembly gaps”: Each of the 62 contigs was individually aligned to the assembled BAC_4 sequence and the stretch along which each has a maximum alignment score was selected to be its locus on the BAC. A maximal stretch along the BAC not covered by any of the 62 contigs was considered an “assembly gap”. There were a total of 42 such gaps. For each of the three scaffolding strategies (i.e., clone mate based, retroscaffolding and combined), an assembly gap is said to be “covered” (alternatively, “not covered”) if there exists a (alternatively, does not exist any) pair of scaffolded contigs spanning the gap. Based on this definition, the number of covered assembly gaps was 22 for clone mate scaffolding, 17 for retroscaffolding, and 28 for the combined scaffolding. This further demonstrates the value added by retroscaffolding.

6.5 Discussion

Our preliminary studies on maize genomic (Section 6.2) and the experimental results on maize contig data (Section 6.4) demonstrate a proof of concept and the value added by retroscaffolding in genome assembly projects. For retroscaffolding to be effective in a genome project, it is necessary that the LTR retrotransposons in the genome are both abundant and distinguishable. LTR sequences within the same family of LTR retrotransposons are harder to distinguish, and repeat-rich genomes (e.g., maize) could have numerous copies of the same family scattered across the genome. Therefore, applying retroscaffolding at a genome level may cause several spurious retro-links to be established, thereby confounding the process of scaffolding. It is for this reason that retroscaffolding is more suited for genome projects involving hierarchical (e.g., BAC-by-BAC) sequencing. Retroscaffolding can also be used to order and orient BACs, if the overlapping ends of two consecutive BACs along a tiling path span an LTR retrotransposon.

In genome projects which generate clone mate information, the scaffolding information derived from retroscaffolding may in part be already provided by clone mates. In the worst case,

even if no new scaffolding information is provided by retroscaffolding, we can benefit from the scaffolding information provided by retroscaffolding in two ways: (i) we will have information about not only the genomic loci but also the composition of the assembly gaps covered by retroscaffolding, as they are expected to contain sequences corresponding to a retrotransposon insert. Therefore, we can prioritize the gaps to finish based on this information, and (ii) the scaffolding output by retroscaffolding can be used to as supporting evidence to validate the output of clone mate information.

Retroscaffolding will be useful in projects which do not generate clone mate information. New sequencing technologies such as the 454 sequencing [Margulies *et al.* (2005)] that do not generate clone mate information are increasingly becoming popular due to their high throughput and cost-effectiveness. Such sequencing technologies may be an appropriate choice for low-budget sequencing projects, and retroscaffolding could make the task of carrying out the assembly in such projects practically feasible.

Retroscaffolding also provides a mechanism to explore the feasibility of a lower coverage sequencing in genome projects. While reducing the sequencing coverage as low as 3X may expose more gaps to span LTR retrotransposons in a target genome, it also implies that there is less redundancy in fragment data. This might affect the quality of the output assembly, especially of those contigs corresponding to the non-repetitive regions of the genome. To circumvent this issue in a hierarchical sequencing project, we propose the following iterative approach to sequencing and assembly: first, sequence all the BACs at a low coverage and assemble them. If a subsequent retroscaffolding reveals the low repeat content in a subset of the input BACs, then perform additional coverage sequencing selectively on these BACs, and reassemble them with the fragments sequenced from all sequencing phases. In practice, this procedure can be repeated until sufficient information is gathered to completely assemble and scaffold each BAC. This approach provides a cost-effective mechanism to sequence repeat-rich genomes without compromising on the quality of the output assembly.

6.6 Concluding Remarks

Genome projects of several economically important plant crops such as maize, barley, sorghum, wheat, etc., are either already underway or are likely to be initiated over the next few years. Most of these plant genomes contain an enormous number of retrotransposons that are not only expected to confound the assembly process, but are also expected to consume the bulk of the sequencing and finishing budget. In contrast to this perspective, the retroscaffolding approach proposed in this research offers the possibility of exploiting the abundance of LTR retrotransposons, thus serving three main purposes: (i) to scaffold contigs that are output by an assembler, (ii) to guide the process of finishing by providing information on the unfinished regions of the genome, and (iii) to reduce sequencing coverage without loss of information regarding the sequenced genes and their relative ordering. Given that sequencing and finishing account for much of the cost in genome projects, continued research in developing this new methodology further could have a high impact.

Several developments have been planned as future work on this research. Specifically, we plan to evaluate the collective effectiveness of retroscaffolding and clone mate based scaffolding at a larger scale. The algorithmic framework for retroscaffolding is still at an early stage of development. Further validation of the framework on sequenced genomes and at much larger scales are essential to ensure an effective and high-quality application of our methodology in forthcoming complex genome projects. To this effect, the application of retroscaffolding on the on-going maize genome project will provide a good starting point.

CHAPTER 7. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

The need for efficient computational methods for the advancement of genomics research cannot be overemphasized. In this doctoral research, we (i) identified some key problems in computational genomics that lead to sequence level discoveries of genes, transcriptomes and genomes, (ii) advanced the state of research through the design and development of scalable efficient algorithms and software solutions, and (iii) applied these new techniques on large real world problem instances and established their biological relevance.

During the early stages of this research, we focused on developing an efficient solution to the EST clustering problem, which has been actively pursued for over a decade. While several approaches were developed prior to our effort, all these approaches were designed to run on serial computers, and have quadratic run-time and/or memory requirements. Our effort resulted in the development of the PaCE parallel clustering algorithm and software, which we later extended to cluster fragment data in the context of gene-enriched genome assembly. The novelty of the PaCE approach lies in its space and time efficiency and its capability to exploit the vast computing power and memory easily available through distributed memory parallel computers. The results of applying PaCE for clustering several large EST data collections and for performing maize gene-enriched genome assembly demonstrate that this research has significantly enhanced the problem size reach while also drastically reducing the time to solution. To the best of our knowledge, the PaCE method is the first and only available massively parallel approach. The PaCE software is freely available to the academic community, and has been distributed to over 40 research groups as of this writing.

We also designed and developed an efficient algorithm and software program to identify LTR

retrotransposons, which constitute one of the most abundant classes of repetitive elements in several eukaryotic genomes. The results of validating our method, LTR_par, against benchmark data shows both superior performance and quality in comparison to previously developed approaches. The parallelization supported by our method also makes it a scalable solution for identifying LTR retrotransposons in large genomes.

One of the important stages of a genome assembly project is to scaffold a set of assembled contigs so that their order and orientation along a target genome can be identified, and sequencing gaps filled through finishing efforts. In this dissertation, we introduced the retroscaffolding problem which is a variant of the conventional contig scaffolding problem. This new approach to achieving scaffolding does not depend on the availability of clone mate information, and can be useful in projects involving the 454 sequencing strategy. Moreover, in projects where clone mate information is available, retroscaffolding would serve as additional supporting evidence in the validity of clone mate links and/or complement the scaffolding information provided by clone mate information. Our results on maize BAC data demonstrate the utility of retroscaffolding at providing both scaffolding information and valuable insights that can be used to potentially reduce finishing and/or sequencing costs in projects targeted for genomes with similar or higher LTR retrotransposon content.

Several functional and algorithmic improvements and developments can be carried out along the lines of this dissertation research:

- The algorithmic ideas and techniques underlying the PaCE method can be easily extended for application in any overlap detection based problem that can be solved by performing an all vs. all pairwise sequence comparison. This was partly demonstrated by our application of the PaCE method for clustering DNA sequences in the context of two different problems — EST clustering and gene-enriched genome assembly.
- The current functionality of PaCE is limited to analyzing a collection of one “type” of DNA sequences — either EST or genomic fragments. This can be generalized into a broader functionality that has the capability to analyze a heterogeneous collection of sequence data. Several applications can benefit from such a generalized functionality (see [Emrich *et al.*

(2005)] for details): (i) In a genome assembly project, detecting a contig that “overlaps” with several ESTs can provide both structural information and expression evidence for a corresponding gene on the contig; while an EST “overlapping” at its two ends to two different contigs can be used to identify contigs spanning the same gene. (ii) In a sequence clustering project, sequences may be available over a period of time. It is sufficient to detect overlaps between the already clustered sequences and a new batch of sequences to effect an incremental clustering. For this purpose, we can treat the already clustered sequences as one “type” and the new batch as another.

To achieve this generic functionality of analyzing heterogenous sequence databases, the PaCE clustering problem formulation can be expanded into a “rule”-based clustering formulation, in which pairwise sequence overlaps are expected to arise only between sequences of different types and the overlap detection mechanism is dictated by the type of sequences being compared — eg., an overlap between a contig and EST can be detected through a spliced alignment technique, while an overlap between a contig and a protein sequence can be detected through a DNA-protein alignment technique.

- Besides LTR retrotransposons, there are several other types of DNA retrotransposons, one of which is called the *Miniature Inverted Transposable Elements* (or *MITEs*). Although much smaller in their lengths, the MITEs have a structure similar to that of LTR retrotransposons. They are characterized by inverted terminal repeats (as opposed to terminal repeats in the same direction in LTR retrotransposons), target site duplications, and a non-coding internal sequence. It will be interesting to see if the ideas underlying our LTR_par algorithm can be extended for identifying MITEs.

BIBLIOGRAPHY

- Adams, M., Celniker, S., Holt, R. *et al.* (2000). The genome sequence of *Drosophila melanogaster*. *Science*, 287(5461):2185–2195.
- Adams, M., Kelley, J., Gocayne, J. *et al.* (1991). Complementary DNA sequencing: expressed sequence tags and human genome project. *Science*, 252(5013):1651–1656.
- Adiga, N., Almasi, G., Almasi, G. *et al.* (2002). An Overview of the BlueGene/L Supercomputer. In *Proc. IEEE/ACM Supercomputing Conference*.
- Altschul, S., Gish, W., Miller, W. *et al.* (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410.
- Aluru, S. and Ko, P. (2005). Chapter 5: Lookup tables, suffix trees and suffix arrays. In *Handbook of computational molecular biology*. CRC Press.
- Apostolico, A., Iliopoulos, C., Landau, G., *et al.* (1988). Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347–365.
- Bailey, L., Searls, D., and Overton, G. (1998). Analysis of EST-Driven Gene Annotation in Human Genomic Sequence. *Genome Research*, 8(4):362–376.
- Baldo, M., Lennon, G., and Soares, M. (1996). Normalization and subtraction: Two approaches to facilitate gene discovery. *Genome Research*, 6:791–806.
- Bao, Z. and Eddy, S. (2002). Automated *de novo* identification of repeat sequence families in sequenced genomes. *Genome Research*, 12:1269–1279.

- Batzoglou, S., Jaffe, D., Stanley, K. *et al.* (2002). ARACHNE a whole-genome shotgun assembler. *Genome Research*, 12(1):177–189.
- Bennetzen, J. (1996). The contributions of retroelements to plant genome organization, function and evolution. *Trends in Microbiology*, 4(9):347–353.
- Boguski, M., Lowe, T., and Tolstoshev, C. (1993). dbEST - database for “expressed sequence tags”. *Nature Genetics*, 4(4):332–333.
- Boguski, M. and Schuler, G. (1995). ESTablishing a human transcript map. *Nature Genetics*, 10(11):369–371.
- Boguski, M., Tolstoshev, C. and Bassett, D.E. Jr. (1994). Gene discovery in dbEST. *Science*, 265(5181):1993–1994.
- Bono, H., Kasukawa, T., Furuno, M. *et al.* (2002). FANTOM DB: database of Functional Annotation of RIKEN Mouse cDNA Clones. *Nucleic Acids Research*, 30(1):116–118.
- Burke, J., Davison, D., and Hide, W. (1999). d2_cluster: A validated method for clustering EST and full-length cDNA sequences. *Genome Research*, 9(11):1135–1142.
- Burke, J., Wang, H., Hide, W., and Davison, D. (1998). Alternative gene form discovery and candidate gene selection from gene indexing projects. *Genome Research*, 8(3):276–290.
- Bushman, F. (2003). Targeting survival: integration site selection by retroviruses and LTR-retrotransposons. *Cell*, 115:135–138.
- Camargo, A., Samaia, H., Dias-Neto, E. *et al.* (2001). From the Cover: The contribution of 700,000 ORF sequence tags to the definition of the human transcriptome. *Proc. National Academy of Sciences*, 98(21):12103–12108.
- Carninci, P., Waki, K., Shiraki, T. *et al.* (2003). Targeting a Complex Transcriptome: The Construction of the Mouse Full-Length cDNA Encyclopedia. *Genome Research*, 13(6):1273–1289.

- Caron, H., Schaik, B., Mee, M. *et al.* (2001). The human transcriptome map: Clustering of highly expressed genes in chromosomal domains. *Science*, 291(5507):1289–1292.
- Carpenter, J., Christoffels, A., Weinbach, Y., and Hide, W. (2002). Assessment of the parallelization approach of d2_cluster for High Performance Sequence Clustering. *Journal of Computational Chemistry*, 23(7):755–757.
- Carulli, J., Artinger, M., Swain, P. *et al.* (1999). High throughput analysis of differential gene expression. *Journal of Cellular Biochemistry*, 72(S30-31):286–296.
- Charlesworth, B., Sniegowski, P., and Stephan, W. (1994). The evolutionary dynamics of repetitive DNA in eukaryotes. *Nature*, 371:215–220.
- Chomczynski, P. and Sacchi, N. (1987). Single-step method of RNA isolation by acid guanidinium thiocyanate-phenol-chloroform extraction. *Analytical Biochemistry*, 162(1):156–159.
- Chou, H. and Holmes, M. (2001). DNA sequence quality trimming and vector removal. *Bioinformatics*, 17(12):1093–1104.
- Christoffels, A., Gelder, A., Greyling, G. *et al.* (2001). STACK Sequence Tag Alignment and Consensus Knowledgebase. *Nucleic Acids Research*, 29(1):234–238.
- Coffin, J., Hughes, S., and Varmus, H. (1997). Retroviruses. *Plantview*.
- Collins, F., Guyer, M., and Chakravarti, A. (1997). Variations on a theme: cataloging human DNA sequence variation. *Science*, 278(5343):1580–1581.
- Consortium, I. H. G. S. (2001). Initial sequencing and analysis of the human genome. *Nature*, 409:860–921.
- Delcher, A., Kasif, S., Fleischmann, R. *et al.* (1999). Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376.
- Duguid, J. and Dinauer, M. (1990). Library subtraction of in vitro cDNA libraries to identify differentially expressed genes in scrapie infection. *Nucleic Acids Research*, 18(9):2789–2792.

- Emrich, S. (2005). *Personal Communication*.
- Emrich, S., Kalyanaraman, A., and Aluru, S. (2005). Chapter 13: Algorithms for large-scale clustering and assembly of biological sequence data. In *Handbook of computational molecular biology*. CRC Press.
- Emrich, S. J., Aluru, S., Fu, Y. *et al.* (2004). A strategy for assembling the maize (*Zea mays* L.) genome. *Bioinformatics*, 20:140–147.
- Ewing, R., Kahla, A., Poirot, O. *et al.* (1999). Large-Scale Statistical Analyses of Rice ESTs Reveal Correlated Patterns of Gene Expression. *Genome Research*, 9(10):950–959.
- Fargnoli, J., Holbrook, N., and Fornace, A.J. Jr. (1990). Low-ratio hybridization subtraction. *Analytical Biochemistry*, 187(2):364–373.
- Feschotte, C., Jiang, N., and Wessler, S. (2002). Plant transposable elements: Where genetics meets genomics. *Nature Reviews (Genetics)*, 3:329–341.
- Fickett, J. (1984). Fast optimal alignment. *Nucleic Acids Research*, 12(1):175–179.
- Flavell, R. (1986). Repetitive DNA and chromosome evolution in plants. *Philosophical Transactions of the Royal Society of London. B.*, 312:227–242.
- Franchini, L., Ganko, E., and McDonald, J. (2004). Retrotransposon-gene associations are wide-spread among *D.melanogaster* populations. *Molecular Biology and Evolution*, 21:1323–1331.
- Fu, Y., Emrich, S., Guo, L. *et al.* (2005). Quality assessment of Maize Assembled Genomic Islands (MAGIs) and large-scale experimental verification of predicted novel genes. *Proc. National Academy of Sciences USA*, 102:12282–12287.
- Gai, X. (2005). *Personal Communication*.
- Ganko, E., Bhattacharjee, V., Schliekelman, P., and McDonald, J. (2003). Evidence for the contribution of LTR retrotransposons to *C. elegans* gene evolution. *Molecular Biology and Genetics*, 20:1925–1931.

- Garg, K., Green, P., and Nickerson, D. (1999). Identification of candidate coding region single nucleotide polymorphisms in 165 human genes using assembled expressed sequence tags. *Genome Research*, 9(11):1087–1092.
- Gautheret, D., Poirot, O., Lopez, F. *et al.* (1998). Alternate Polyadenylation in Human mRNAs: A Large-Scale Analysis by EST Clustering. *Genome Research*, 8(5):524–530.
- Gelfand, M. S., Mironov, A., and Pevzner, P. (1996). Gene recognition via spliced alignment. *Proc. National Academy of Sciences*, 93(17):9061–9066.
- Gotoh, O. (1982). An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708.
- Grandbastien, M., Spielmann, A., and Caboche, M. (1989). Tnt1, a mobile retroviral-like transposable element of tobacco isolated by plant cell genetics. *Nature*, 337:376–380.
- Green, P. (1994, (Date accessed 12 Apr 2003)). Phrap - the assembler. <http://www.phrap.org>.
- Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789–828.
- Gusfield, D. (1997a). *Algorithms on strings, trees and sequences Computer Science and Computational Biology*. Cambridge University Press, Cambridge, London.
- Gusfield, D. (1997b). *Algorithms on strings, trees and sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, London.
- Hariharan, R. (1997). Optimal parallel suffix tree construction. *Journal of Computer and System Sciences*, 55(1):44–69.
- Havlak, P., Chen, R., Durbin, K. *et al.* (2004). The ATLAS genome assembly system. *Genome Research*, 14:721–732.
- Hirochika, H., Sugimoto, K., Otsuki, Y. *et al.* (1996). Retrotransposons of rice involved in mutations induced by tissue culture. *Proc. National Academy of Sciences*, 93(15):7783–7788.

- Hirschberg, D. (1975). A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343.
- Huang, X. (2005). Chapter 8: Computational methods for genome assembly. In *Handbook of computational molecular biology*. CRC Press.
- Huang, X., Adams, M., Zhou, H., and Kerlavage, A. (1997). A tool for analyzing and annotating genomic sequences. *Genomics*, 46:37–45.
- Huang, X. and Chao, K. (2003). A generalized global alignment algorithm. *Bioinformatics*, 19(2):228–233.
- Huang, X. and Madan, A. (1999). CAP3: A DNA sequence assembly program. *Genome Research*, 9(9):868–877.
- Huang, X., Wang, J., Aluru, S. *et al.* (2003). PCAP: A whole-genome assembly program. *Genome Research*, 13:2164–2170.
- Huson, D., Reinert, K., and Myers, E. (2001). The greedy pathmerging algorithm for sequence assembly. In *Proc. International Conference on Research in Computational Biology (RECOMB)*, pages 157–163.
- Jackson, B. and Aluru, S. (2005). Chapter 1: Pairwise sequence alignment. In *Handbook of computational molecular biology*. CRC Press.
- Jaffe, D., Butler, J., Gnerre, S. *et al.* (2003). Whole-genome sequence assembly for mammalian genomes: Arachne 2. *Genome Research*, 13:91–96.
- Jain, A. and Dubes, R. (1988). *Algorithms for clustering data*. Prentice Hall, Englewood Cliffs, NJ.
- Jiang, J. and Jacob, H. (1998). EbEST: An Automated Tool Using Expressed Sequence Tags to Delineate Gene Structure. *Genome Research*, 8(3):268–275.
- Johns, M., Mottinger, J., and Freeling, M. (1985). A low copy number, copia-like transposon in maize. *The EMBO Journal*, 4(5):1093–1101.

- Jordan, I. and McDonald, J. (1999). Comparative genomics and evolutionary dynamics of *Saccharomyces cerevisiae* Ty elements. *Genetica*, 107:3–13.
- Kalyanaraman, A. (2002). Parallel clustering of expressed sequence tags. *Masters Thesis, Iowa State University*.
- Kalyanaraman, A. and Aluru, S. (2005a). Chapter 12: Expressed Sequence Tags: Clustering and applications. In *Handbook of computational molecular biology*. CRC Press.
- Kalyanaraman, A. and Aluru, S. (2005b). Efficient algorithms and software for detection of full-length LTR retrotransposons. In *Proc. IEEE Computational Systems Bioinformatics Conference*, pages 56–64.
- Kalyanaraman, A. and Aluru, S. (2006). Efficient algorithms and software for detection of full-length LTR retrotransposons. *Journal of Bioinformatics and Computational Biology*, 4(2):197-216.
- Kalyanaraman, A., Aluru, S., Brendel, V., and Kothari, S. (2003a). Space and time efficient parallel algorithms and software for EST clustering. *IEEE Transactions on Parallel and Distributed Systems*, 14(12):1209–1221.
- Kalyanaraman, A., Aluru, S., Kothari, S., and Brendel, V. (2003b). Efficient clustering of large EST data sets on parallel computers. *Nucleic Acids Research*, 31(11):2963–2974.
- Kalyanaraman, A., Aluru, S., and Schnable, P.S. (2006a). Turning repeats to advantage: Scaffolding genomic contigs using LTR retrotransposons. In *Proc. LSS Computational Systems Bioinformatics Conference*.
- Kalyanaraman, A., Emrich, S., Schnable, P.S., and Aluru, S. (2006b). Assembling genomes on large-scale parallel computers. In *Proc. IEEE International Parallel and Distributed Processing Symposium*.
- Kan, Z., Rouchka, E., Gish, W., and States, D. (2001). Gene Structure Prediction and Alternative Splicing Analysis Using Genomically Aligned ESTs. *Genome Research*, 11(5):889–900.

- Kapros, T., Robertson, A., and Waterborg, J. (1994). A simple method to make better probes for short DNA fragments. *Molecular Biotechnology*, 2(1):95–98.
- Karkkainen, J. and Sanders, P. (2003). Simple linear work suffix array construction. *Lecture Notes in Computer Science*, 2719:943–955.
- Karkkanen, J. and Sander, P. (2003). Simpler linear work suffix array construction. In *Proc. International Colloquium on Automata, Languages and Programming*.
- Kasai, T., Lee, G., Arimura, H. *et al.* (2001). Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. Combinatorial Pattern Matching*, pages 181–192.
- Kidwell, M. and Lisch, D. (1997). Transposable elements as sources of variation in animals and plants. *Proc. National Academy of Sciences*, 94:7704–7711.
- Kim, D., Sim, J., Park, H., and Park, K. (2003). Linear-time construction of suffix arrays. *Lecture Notes in Computer Science*, 2676:186–199.
- Kim, J., Vanguri, S., Boeke, J. *et al.* (1998). Transposable Elements and Genome Organization: A Comprehensive Survey of Retrotransposons Revealed by the Complete *Saccharomyces cerevisiae* Genome Sequence. *Genome Research*, 8:464–478.
- Ko, P. and Aluru, S. (2003). Space efficient linear time construction of suffix arrays. In *Proc. Combinatorial Pattern Matching*, pages 200–210.
- Kurtz, S., Choudhuri, J., Ohlebusch, E. *et al.* (2001). REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29:4633–4642.
- Kurtz, S. and Schleiermacher, C. (1999). REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15:426–427.
- Lander, E. and *et al.* (2001). Initial sequencing and analysis of the human genome. *Nature*, 409:860–921.

- Liang, F., Holt, I., Pertea, G. *et al.* (2000). An optimized protocol for analysis of EST sequences. *Nucleic Acids Research*, 28(18):3657–3665.
- Malde, K., Coward, E., and Joassen, I. (2003). Fast sequence clustering using a suffix array algorithm. *Bioinformatics*, 19(10):1221–1226.
- Manber, U. and Myers, G. (1993). Suffix arrays: A new method for on-line search. *SIAM Journal of Computing*, 22:935–948.
- Mao, M., Fu, G., Wu, J., Zhang, Q. *et al.* (1998). Identification of genes expressed in human CD34+ hematopoietic stem/progenitor cells by expressed sequence tags and efficient full-length cDNA cloning. *Proc. National Academy of Sciences*, 95(14):8175–8180.
- Margulies, M., Egholm, M., Altman, W. *et al.* (2005). Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, pages 376–380.
- Marth, G., Korf, I., Yandell, M. *et al.* (1999). A general approach to single-nucleotide polymorphism discovery. *Nature Genetics*, 23:452–456.
- McCarthy, E., Liu, L., Lizhi, G., and McDonald, J. (2002). Long terminal repeat retrotransposons of *oryza sativa*. *Genome Biology*, 3:0053.1–0053.11.
- McCarthy, E. and McDonald, J. (2003). LTR.STRUC: a novel search and identification program for LTR retrotransposons. *Bioinformatics*, 19:362–367.
- McCarthy, E. and McDonald, J. (2004). LTR Retrotransposons of *Mus musculus*. *Genome Biology*, 5:R14.
- McCombie, R. (2005). *Personal Communication*.
- McCreight, E. (1976). A space economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272.
- Meyers, B., Tingey, S., and Morgante, M. (1998). Abundance, distribution, and transcriptional activity of repetitive elements in the maize genome. *Science*, 274:765–768.

- Miguel, P. (2005). *Personal Communication*.
- Miller, J., Dong, F., Jackson, S. *et al.* (1998). Retrotransposon-related DNA sequences in the centromeres of grass chromosomes. *Genetics*, 150:1615–1623.
- Miller, R., Christoffels, A., Gopalakrishnan, C. *et al.* (1999). A Comprehensive Approach to Clustering of Expressed Human Gene Sequence: The Sequence Tag Alignment and Consensus Knowledge Base. *Genome Research*, 9(11):1143–1155.
- Mironov, A., Fickett, J., and Gelfand, M. (1999). Frequent alternative splicing of human genes. *Genome Research*, 9(12):1288–1293.
- Modrek, B. and Lee, C. (2002). A genomic view of alternative splicing. *Nature genetics*, 30:13–19.
- Modrek, B., Resch, A., Grasso, C., and Lee, C. (2001). Genome-wide detection of alternative splicing in expressed sequences of human genes. *Nucleic Acid Research*, 29(13):2850–2859.
- Morgante, M., Policriti, A., Vitacolonna, N., and Zuccolo, A. (2002). Automated search for LTR retrotransposons. <http://citeseer.ist.psu.edu/644336.html>.
- Mullikin, J. and Ning, Z. (2003). The phusion assembler. *Genome Research*, 13:81–90.
- Myers, E., Sutton, G., Delcher, A. *et al.* (2000). A Whole-Genome Assembly of *Drosophila*. *Science*, 287:2196–2204.
- Needleman, S. and Wunsch, C. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453.
- Nelson, M., Kang, S., Braun, E. *et al.* (1997). Expressed sequences from conidial, mycelial, and sexual stages of *Neurospora crassa*. *Fungal Genetics and Biology*, 21:348–363.
- NSF (2005). NSF, USDA and DOE Award \$32 Million to Sequence Corn Genome. http://www.nsf.gov/news/news_summ.jsp?cntn_id=104608&org=BIO&from=news, Press Release 05-197.

- Okazaki, Y., Furuno, M., Kasukawa *et al.* (2002). Analysis of the mouse transcriptome based on functional annotation of 60,770 full-length cDNAs. *Nature*, 420:563–573.
- Palmer, L., Rabinowicz, P., O'Shaughnessy, A. *et al.* (2003). Maize genome sequencing by Methylation Filtration. *Science*, 302(5653):2115–2117.
- Patanjali, S., Parimoo, S., and Weissman, S. (1991). Construction of a uniform-abundance (normalized) cDNA library. *Proc. National Academy of Sciences*, 88(5):1943–1947.
- Pedretti, K. (2001). Accurate, parallel clustering of EST (gene) sequences. *Masters Thesis, University of Iowa*.
- Pennisi, E. (2005). Cut-rate genomes on the horizon? *Science*, 309(5736):862–862.
- Pertea, G., Huang, X., Liang, F. *et al.* (2003). TIGR Gene Indices clustering tool (TGICL) a software system for fast clustering of large EST datasets. *Bioinformatics*, 19(5):651–652.
- Peterson-Burch, B., Nettleton, D., and Voytas, D. (2004). Genomic Neighbourhoods for *Arabidopsis* retrotransposons: a role for targeted integration in the distribution of the Metaviridae. *Genome Biology*, 5:R78.
- Picoult-Newberg, L., Ideker, T., Pohl, M. *et al.* (1999). Mining SNPs From EST Databases. *Genome Research*, 9(2):167–174.
- Polavarapu, N. (2005). *Personal Communication*.
- Polavarapu, N., Bowen, N., and McDonald, J. (2006). Identification, characterization and comparative genomics of chimpanzee endogenous retroviruses. *Genome Biology*, In Press.
- Pontius, J., Wagner, L., and Schuler, G. (2003). UniGene a unified view of the transcriptome. *The NCBI Handbook*.
- Pop, M., Kosack, D., and Salzberg, S. (2004). Hierarchical scaffolding with Bambus. *Genome Research*, 14:149–159.

- Pop, M., Salzberg, S., and Shumway, M. (2002). Genome sequence assembly: algorithms and issues. *IEEE Computer*, 35(7):47–54.
- Promislow, D., Jordan, I., and McDonald, J. (1999). Genomic demography: a life-history analysis of transposable element evolution. *Proc. Royal Society of London B: Biological Sciences*, 266(1428):1555–1560.
- Quackenbush, J., Liang, F., Holt, I. *et al.* (2000). The TIGR gene indices reconstruction and representation of expressed gene sequences. *Nucleic Acids Research*, 28(1):141–145.
- Rabinowicz, P., Schutz, K., Dedhia, N. *et al.* (1999). Differential methylation of genes and retrotransposons facilitates shotgun sequencing of the maize genome. *Nature Genetics*, 23:305–308.
- Rajko, S. and Aluru, S. (2004). Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems*, 15(12):1070–1081.
- Rounsley, S., Glodek, A., Sutton, G. *et al.* (1996). The construction of Arabidopsis expressed sequence tag assemblies. *Plant Physiology*, 112:1177–1183.
- Sanger, F. and Coulson, A. (1975). A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase. *Journal of Molecular Biology*, 94(3):441–448.
- Sanger, F., Coulson, A., Hong, G. *et al.* (1982). Nucleotide sequence of bacteriophage lambda DNA. *Journal of Molecular Biology*, 162:729–773.
- Sanger, F., Nicklen, S., and Coulson, A. (1977). DNA sequencing with chain-terminating inhibitors. *Proc. National Academy of Sciences USA*, 74(12):5463–5467.
- SanMiguel, P., Gaut, B., Tikhonov, A. *et al.* (1998). The paleontology of intergene retrotransposons of maize. *Nature Genetics*, 20:43–45.
- SanMiguel, P., Tikhonov, A., Jin, Y. *et al.* (1996). Nested retrotransposons in the intergenic regions of the maize genome. *Science*, 274:765–768.

- Satou, Y., Yamada, L., Mochizuki, Y. *et al.* (2002). A cDNA resource from the Basal Chordate *Ciona intestinalis*. *Genesis*, 33:153–154.
- Schlueter, S., Dong, Q., and Brendel, V. (2003). GeneSeqer@PlantGDB: gene structure prediction in plant genomes. *Nucleic Acids Research*, 31(13):3597–3600.
- Schmid, D. and Girou, C. (1987). Cloning of cDNA derived from mRNA of the electric lobe of *Torpedo marmorata* and selection of putative cholinergic-specific sequences. *Journal of Neurochemistry*, 48(1):307–312.
- Schweinfest, C., Henderson, K., Gu, J. *et al.* (1990). Subtraction hybridization cDNA libraries from colon carcinoma and hepatic cancer. *Genetic Analysis : Techniques and Applications*, 7(3):64–70.
- Seki, M., Narusaka, M., Kamiya, A. *et al.* (2002). Functional annotation of a full-length Arabidopsis cDNA collection. *Science*, 296(5565):141–145.
- Sequencing, T. C. and Consortium, A. (2005). Initial sequence of the chimpanzee genome and comparison with the human genome. *Nature*, 437:69–87.
- Setubal, J. and Meidanis, J. (1997). *Introduction to computational molecular biology*. PWS Publishing Company, Boston, MA.
- Sherry, S., Ward, M., Kholodov, M. *et al.* (2001). dbSNP: the NCBI database of genetic variation. *Nucleic Acids Research*, 29(1):308–311.
- Smit, A. and Green, P. (1999). RepeatMasker. <http://ftp.genome.washington.edu/RM/RepeatMasker.html>.
- Smith, T. and Waterman, M. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197.
- Soares, M., Bonaldo, M., Jelene, P. *et al.* (1994). Construction and characterization of a normalized cDNA library. *Proc. National Academy of Sciences*, 91(20):9228–9232.

- Stapleton, M., Liao, G., Brokstein, P. *et al.* (2002). The Drosophila Gene Collection: Identification of Putative Full-Length cDNAs for 70% of *D. melanogaster* Genes. *Genome Research*, 12(8):1294–1300.
- Sutton, G., White, O., Adams, M., and Kerlavage, A. (1995). TIGR assembler: A new tool for assembling large shotgun sequencing projects. *Genome Science and Technology*, 1:9–19.
- Tarjan, R. (1975). Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225.
- Ton, C., Hwang, D., Dempsey, A. *et al.* (2000). Identification, characterization, and mapping of expressed sequence tags from an embryonic zebrafish heart *cDNA* library. *Genome Research*, 10(12):1915–1927.
- Torney, D., Burks, C., Davison, D., and Sirotkin, K. (1990). *Computers and DNA*. Addison-Wesley, New York.
- Travis, G. and Sutcliffe, J. (1988). Phenol emulsion-enhanced DNA-driven subtractive cDNA cloning: isolation of low-abundance monkey cortex-specific mRNAs. *Proc. National Academy of Sciences*, 85(5):1696–1700.
- Trivedi, N., Bischof, J., Davis, S. *et al.* (2002). Parallel creation of non-redundant gene indices from partial mRNA transcripts. *Future Generation Computer Systems*, 18:863–870.
- Ukkonen, E. (1992). Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, 92(1):191–211.
- Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, 14:249–260.
- Usuka, J., Zhu, W., and Brendel, V. (2000). Optimal spliced alignment of homologous cDNA to a genome database ZmDB. *Bioinformatics*, 16(3):203–211.
- VanBuren, V., Piao, Y., Dudekula, D. *et al.* (2002). Assembly, verification, and initial annotation of the NIA mouse 7.4K cDNA clone set. *Genome Research*, 12(12):1999–2003.

- Varagona, M., Purugganan, M., and Wessler, S. (1992). Alternative splicing induced by insertion of retrotransposons into the maize *waxy* gene. *Plant Cell*, 4:811–820.
- Velculescu, V., Zhang, L., Vogelstein, B., and Kinzler, K. (1995). Serial analysis of gene expression. *Science*, 270(5235):484–487.
- Venter, J., Adams, M., Myers, E. *et al.* (2001a). The sequence of the human genome. *Science*, 291:1304–1351.
- Venter, J., Adams, M., Myers, E. *et al.* (2001b). The sequence of the human genome. *Science*, 291:1304–1351.
- Venter, J., Remington, K., Heidelberg, J. *et al.* (2004). Environmental genome shotgun sequencing of the Sargasso Sea. *Science*, 304(5667):66–74.
- Vicianta, C., Suoniemi, A., Ananthawat-Jansson, K. *et al.* (1999). Retrotransposon BARE-1 and Its Role in Genome Evolution in the Genus *Hordeum*. *Plant Cell*, 11:1769–1784.
- Weiner, P. (1973). Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11.
- Wessler, S., Bureau, T., and White, S. (1995). LTR-retrotransposons and MITEs: important players in the evolution of plant genomes. *Current Opinion In Genetics and Development*, 5:814–821.
- Wheeler, D., Barrett, T., Benson, D. *et al.* (2005). Database resources of the National Center for Biotechnology Information. *Nucleic Acids Research*, 33:D39–D45.
- White, S., Habera, L., and Wessler, S. (1994). Retrotransposons in the Flanking Region of Normal Plant Genes: A Role for Copia-Like Elements in the Evolution of Gene Structure and Expression. *Proc. National Academy of Sciences*, 91:11792–11796.
- Whitfield, C., Band, M., Bonaldo, M. *et al.* (2002). Annotated expressed sequence tags and cDNA microarrays for studies of brain and behavior in the honey bee. *Genome Research*, 12(4):555–566.

- Xiong, Y. and Eickbush, T. (1990). Origin and evolution of retroelements based upon their reverse transcriptase sequences. *EMBO Journal*, 9:3353–3362.
- Ye, Z. and Parry, J. (2002). The discovery and confirmation of single nucleotide polymorphisms in the human p53R2 gene by EST database analysis. *Mutagenesis*, 17(5):361–364.
- Yuan, Y., SanMiguel, P., and Bennetzen, J. (2003). High-C₀t sequence analysis of the maize genome. *The Plant Journal*, 34:249–255.
- Zhang, Z., Shwartz, S., Wagner, L., and Miller, W. (2000). A greedy algorithm for aligning DNA sequences. *Journal of Computational Biology*, 7(1-2):203–214.
- Zhu, T. and Wang, X. (2000). Large-Scale Profiling of the Arabidopsis Transcriptome. *Plant Physiology*, 124:1472–1476.
- Zhu, W., Schlueter, S., and Brendel, V. (2003). Refined annotation of the *Arabidopsis thaliana* genome by complete Expressed Sequence Tag mapping. *Plant Physiology*, 132:469–484.