Free Parallel Data Mining

by

Bin Li

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
September, 1998

Approved: ______________________

Professor Dennis Shasha
Research Advisor

© Bin Li

All Rights Reserved, 1998

To the memory of my father, Yongnian Li

To my mother, Yanyun Su

And to Natalie

Acknowledgments

I wish to express my deepest gratitude to my advisor, Professor Dennis Shasha, for having introduced me first to the field of parallel computing and later to the wonderful world of data mining. I am enormously indebted to Dennis for all his guidance, encouragement, and support along the way. Without him, this thesis would not have been possible.

I am very grateful to Professor Ralph Grishman and Professor Vijay Karamcheti for reading my thesis and giving me many good suggestions that greatly improved the presentation and enhanced the clarity of my work. I am deeply appreciative to Professor Edmond Schonberg and Professor Ernest Davis for serving on my thesis committee. Professor Ralph Grishman and Professor Edmond Schonberg also served on my oral exam committee.

My supervisor at work, Mr. David M. Jacobs, was a tremendous help during the final months. He was extremely flexible about my working schedule and gave me time off when I needed without hesitation.

Thanks to Karp Jeong for helping me write my first PLinda program; thanks to Peter Wyckoff for his excellent work on PLinda which I greatly benefited from; thanks to Peter Piatko for many valuable discussions about my work; thanks to Chin-Yuan Cheng for working so hard to bring NyuMiner into reality.

I would like to thank all the people who at one time shared an office with me at Courant: Chi Yao, Martin Garcia Keller, Liddy Shriver, Peter Piatko, Arash Baratloo, Eugene Agichtein, Tying-Lu Liu. They were a great help.

I thank all my friends and fellow students who made my life at Courant more enjoyable. In particular, thanks to Saugata Basu, Liang Cao, Chao-yu Cheng, Churng-wei Chu, Xianghui Duan, Ron Even, Zhijun Liu, Hsing-kuo Pao, Laxmi Parida, Jihai Qiu, Suren Talla, Hanping Xu, Yuanyuan Zhao.

I would also like to thank Dr. Ju Zhang for his interest in my work and his encouragement.

Finally, thanks to my sister Dr. Hong Li and her husband Dr. Jianming Cao for their love and support.

Abstract

Free Parallel Data Mining

Bin Li

New York University, 1998

Research Advisor: Professor Dennis Shasha

Data mining is the emerging field of applying statistical and artificial intelligence techniques to the problem of finding novel, useful, and non-trivial patterns from large databases. This thesis presents a framework for easily and efficiently parallelizing data mining algorithms. We propose an acyclic directed graph structure, *exploration dag* (*E-dag*), to characterize the computation model of data mining algorithms in classification rule mining, association rule mining, and combinatorial pattern discovery. An E-dag can be constructively formed in parallel from specifications of a data mining problem, then a parallel E-dag traversal is performed on the fly to efficiently solve the problem. The effectiveness of the E-dag framework is demonstrated in biological pattern discovery applications.

We also explore data parallelism in data mining applications. The cross-validation and the windowing techniques used in classification tree algorithms facilitate easy development of efficient data partitioning programs. In this spirit, we present a new classification tree algorithm called *NyuMiner* that guarantees that every split in a classification tree is optimal with respect to any given impurity function and any given maximum number of branches allowed in a split. NyuMiner can be easily parallelized

using the data partitioning technique.

This thesis also presents a software architecture for running parallel data mining programs on networks of workstations (NOW) in a fault-tolerant manner. The software architecture is based on Persistent Linda (PLinda), a robust distributed parallel computing system which automatically utilize idle cycles. Templates are provided for application programmers to develop parallel data mining programs in PLinda. Parallelization frameworks and the software architecture form a synergy that makes free efficient data mining realistic.

Table of Contents

Dedication **iii**

Acknowledgments **iv**

Abstract **vi**

List of Figures **xiv**

List of Tables **xix**

Chapter 1 Introduction **1**

1.1 Why Free Parallel Data Mining? 1

1.2 Approach and Contributions 3

1.3 Organization of Thesis 5

Chapter 2 Related Work **7**

2.1 Classification Rule Mining 7

2.1.1 An Example 7

2.1.2 Classification Rules 9

2.1.3 Decision Trees 9

2.1.4 Building Decision Trees 10
2.1.5 Attribute Selection 10
2.1.6 Parallelization Attempts 12
2.2 Association Rule Mining 14
2.2.1 An Example 14
2.2.2 Association Rules 15
2.2.3 Properties of Frequent Sets 15
2.2.4 Basic Scheme for Association Rule Mining 16
2.2.5 State of the Art 17
2.2.6 Parallelization Attempts 18
2.3 Pattern Discovery in Combinatorial Databases 19
2.3.1 An Example 19
2.3.2 Pattern Discovery 19
2.3.3 Sequence Pattern Discovery 20
2.3.4 Sequence Pattern Discovery Algorithm 20
2.4 Platforms for Parallel Data Mining 22
2.4.1 Networked Workstations as a Parallel Computing Platform . . . 22
2.4.2 Parallelism in Data Mining 23
2.4.3 Condor 23
2.4.4 Calypso 25
2.4.5 Piranha 30
2.4.6 Persistent Linda 34
2.4.7 Comparisons 37
2.5 Other Parallel Computing Systems on NOW 38
2.5.1 Cilk-NOW 38

2.5.2 TreadMarks 40
2.5.3 PVM 42
2.6 Parallel Search Algorithms 42
2.7 Domain-Specific Parallelism Frameworks 44
2.7.1 Multipol 44
2.7.2 LPARX 44

Chapter 3 E-Dag: Framework for Finding Lattices of Patterns 46
3.1 Exploration Dag 46
3.1.1 Modeling Data Mining Applications 46
3.1.2 Defining Data Mining Applications 48
3.1.3 Solving Data Mining Applications 50
3.1.4 Exploration Dag 51
3.1.5 A Data Mining Virtual Machine 51
3.2 A Parallel Data Mining Virtual Machine 56
3.2.1 A Parallel Data Mining Virtual Machine 56
3.2.2 PLinda Implementation of PDMVM 57
3.3 Optimal Implementation of PDMVM 59
3.3.1 Exploration Tree 59
3.3.2 E-tree Traversal 60
3.3.3 PLinda Implementation of PETT 64
3.3.4 Optimal PLinda Implementation of PDMVM 64
3.4 Summary 66

Chapter 4 Biological Pattern Discovery 67
4.1 Biological Pattern Discovery 67

4.1.1 Discovery of Motifs in Protein Sequences 67
4.1.2 Discovery of Motifs in RNA Secondary Structures 69
4.2 Parallel Pattern Discovery in PLinda 72
4.2.1 Applying the E-dag Model 72
4.2.2 PLinda Implementation 73
4.3 Experimental Results 75
4.3.1 Load-balanced vs. Optimistic 77
4.3.2 Adaptive Master 79
4.3.3 Experiments on a Large Network 80

Chapter 5 NyuMiner: Classification Trees by Optimal Sub-K-ary Splits 83

5.1 Introduction 83
5.2 Related Work 85
5.3 Optimal Sub-K-ary Splits 88
5.3.1 Numerical Variables 91
5.3.2 Categorical Variables 92
5.4 To Prune or Not to Prune 93
5.4.1 Minimal Cost Complexity Pruning 93
5.4.2 Multiple Incremental Sampling and Rule Selection 97
5.5 Comparing NyuMiner to C4.5 and CART 98
5.5.1 Description of Data Sets 99
5.5.2 Comparison of Accuracy 100
5.5.3 Complementarity Tests 101
5.6 Application: Making Money in Foreign Exchange 103
5.6.1 Preparing Data 103

5.6.2 Selecting Rules . . . 104
5.6.3 Making Money . . . 106

Chapter 6 Parallel Classification Tree Algorithms **108**
6.1 Parallelism in Cross Validation . . . 108
6.1.1 Parallel NyuMiner-CV . . . 109
6.2 Parallelism in Multiple Incremental Sampling . . . 112
6.2.1 Parallel C4.5 . . . 112
6.2.2 Parallel NyuMiner-RS . . . 114
6.3 Summary . . . 116

Chapter 7 Software Architecture **117**
7.1 Fault-Tolerance . . . 117
7.1.1 Parallel Virtual Machine in PLinda . . . 118
7.1.2 PLinda's Fault-Tolerance Guarantee . . . 118
7.1.3 PLinda Data Mining Programs Are Fault-Tolerant . . . 119
7.2 Running PLinda Programs . . . 119
7.2.1 How to Start the PLinda User Interface . . . 119
7.2.2 How to Start the PLinda Server . . . 120
7.2.3 How to Add Hosts . . . 121
7.2.4 How to Select a PLinda Program to Run . . . 121
7.2.5 How to Observe and Control Execution Behaviors . . . 123
7.3 Software Available on the WWW . . . 125

Chapter 8 Conclusions and Future Work **126**
8.1 Conclusions . . . 126

8.2 Future Work 127

Bibliography **129**

List of Figures

2.1 The decision tree based on the heart disease diagnosis records for the PLinda group. 8

2.2 An example of arbiter tree in Meta-Learning. T1, T2, T3, and T4 are subsets of the training database; C1, C2, C3, and C4 are classifiers; A12, A34, and A14 are arbiters. 13

2.3 A vector addition program in Calypso. 27

2.4 Vector addition master in Linda. 31

2.5 Vector addition slave in Linda. 31

2.6 Vector addition master in Persistent Linda. 35

2.7 Vector addition slave in Persistent Linda. 35

2.8 A Cilk procedure to compute the nth Fibonacci number. This procedure contains two threads, `Fib` and `Sum`. 39

3.1 A complete E-DAG for a sequence pattern discovery application on sequences FFRR, MRRM, and MTRM. 52

3.2 A complete E-DAG for an association rule mining application on the set of items $\{1, 2, 3, 4\}$. 53

3.3 A complete E-DAG for a classification rule mining application on a simple database with attributes A (possible values $a1$ and $a2$), and B (possible values $b1$, $b2$, and $b3$). 54
3.4 Pseudo-PLinda code of a PLED master. 58
3.5 Pseudo-PLinda code of a PLED worker. 58
3.6 An E-tree for a sequence pattern discovery application on sequences FFRR, MRRM, and MTRM. 60
3.7 An E-tree for an association rule mining application on the set of items $\{1, 2, 3, 4\}$. 61
3.8 An E-tree for a classification rule mining application on a simple database with attributes A (possible values $a1$ and $a2$), and B (possible values $b1$, $b2$, and $b3$). 62
3.9 Pseudo-PLinda code of a PLET master. 65
3.10 Pseudo-PLinda code of a PLET worker. 65

4.1 Sequence representation of a real protein named "CG2A_DAUCA G2/MITOTIC-SPECIFIC CYCLIN C13-1 (A-LIKE CYCLIN)". 68
4.2 Illustration of a typical RNA secondary structure and its tree representation. (a) Normal polygonal representation of the structure. (b) Tree representation of the structure. 71
4.3 (a) The set $\mathcal{S}$ of three trees (these trees are hypothetical ones used solely for illustration purposes). (b) Two motifs exactly occurring in all three trees. (c) Two motifs approximately occurring, within distance 1, in all three trees. 72

4.4 Pseudo-PLinda code of optimistic parallel sequence pattern discovery master. 74
4.5 Pseudo-PLinda code of optimistic parallel sequence pattern discovery worker. 74
4.6 Pseudo-PLinda code of load-balanced parallel sequence pattern discovery master. 76
4.7 Pseudo-PLinda code of load-balanced parallel sequence pattern discovery worker. 76
4.8 Comparison of optimistic and load-balanced parallel sequence pattern discovery programs on setting 1 of cyclins.pirx. 77
4.9 Comparison of optimistic and load-balanced parallel sequence pattern discovery programs on setting 2 of cyclins.pirx. 78
4.10 Comparison of load-balanced version with and without adaptive master on setting 1 of cyclins.pirx. 79
4.11 Comparison of optimistic version with and without adaptive master on setting 1 of cyclins.pirx. 80
4.12 Comparison of load-balanced version with and without adaptive master on setting 2 of cyclins.pirx. 81
4.13 Comparison of optimistic version with and without adaptive master on setting 2 of cyclins.pirx. 81
4.14 Running time of our parallel sequence pattern discovery program on 5, 10, 15, 20, 25, 30, 35, 40, and 45 machines. 82

5.1 A set of 27 data elements with values of a numerical variable showing. . . 86
5.2 The set of 27 data elements grouped into 10 baskets by value. 86

5.3 The 10 baskets with class labels. 86

5.4 7 baskets divided by boundary points. 87

5.5 A k-ary split S and a $(k-1)$-ary split S' resulted from merging two partitions in S into one. 90

5.6 A partial classification tree NyuMiner-RS built for the **yu** data set. . . . 105

6.1 Pseudo-PLinda code of Parallel NyuMiner-CV master. 110

6.2 Pseudo-PLinda code of Parallel NyuMiner-CV worker. 110

6.3 Running time and speedup results of Parallel NyuMiner-CV experiments on the **yeast** dataset. 111

6.4 Running time and speedup results of Parallel NyuMiner-CV experiments on the **satimage** dataset. 112

6.5 Running time and speedup results of Parallel C4.5 experiments on the **smoking** dataset. 113

6.6 Running time and speedup results of Parallel C4.5 experiments on the **letter** dataset. 114

6.7 Running time and speedup results of Parallel NyuMiner-RS experiments on the **yeast** dataset. 115

6.8 Running time and speedup results of Parallel NyuMiner-RS experiments on the **satimage** dataset. 116

7.1 A snapshot of PLinda runtime menu window. 119

7.2 A snapshot of PLinda runtime “Settings” window. 120

7.3 A snapshot of PLinda runtime “System” window. 121

7.4 A snapshot of PLinda runtime “Hosts” window. 122

7.5 A snapshot of PLinda runtime “Apps” window. 123

7.6 A snapshot of PLinda runtime “Monitor” window. 124

7.7 A screen snapshot of PLinda environment. 124

List of Tables

2.1 Imaginary heart disease diagnosis records for the PLinda group. 8

2.2 An imaginary sales transaction database from K-mart. 14

2.3 Comparison of Condor, Calypso, Piranha, and Persistent Linda. 38

3.1 A comparison of specifications of three classes of data mining applications. 47

4.1 A comparison of specifications of two biological pattern discovery applications. 73

4.2 Parameter settings and sequential program results of cyclins.pirx. 78

5.1 Descriptions of the 7 benchmark data sets. 99

5.2 Statistical features of the 7 benchmark data sets. 100

5.3 Comparison of classification accuracies of C4.5, CART, NyuMiner-CV, and NyuMiner-RS. 101

5.4 Complementarity test results on the benchmark data sets. 102

5.5 Descriptions of foreign exchange data sets. 104

5.6 Money made in foreign exchange. 107

6.1 Sequential running time (sec.) of NyuMiner-CV on the data sets **yeast** and **satimage**. ($V = 0$ means no cross validation.) 111
6.2 Sequential running time (sec.) of C4.5 on the data sets **smoking** and **letter**. 113
6.3 Sequential running time (sec.) of NyuMiner-RS on the data sets **yeast** and **satimage**. 115

Chapter 1

Introduction

1.1 Why Free Parallel Data Mining?

Traditional methods of data analysis, based mainly on a person dealing directly with the data, do not scale to voluminous data sets. While database technology has provided us with the basic tools for the efficient storage and lookup of large data sets, the issue of how to help humans analyze and understand large bodies of data remains a difficult and unsolved problem. The emerging field of data mining promises to provide new techniques and intelligent tools to encounter the challenge.

Data mining is sometimes called *knowledge discovery in databases (KDD)*. Both data mining and KDD are at best vaguely defined. Their definitions largely depend on the background and views of the definers. Here are some sample definitions of data mining:

- The essence of data mining is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data [35].
- Data mining is the use of statistical methods with computers to uncover useful

patterns inside databases [58].

- Data mining is the process of extracting previously unknown, comprehensible, and actionable information from large databases and using it to make crucial business decisions. – Zekulin [38]

- Data mining is a decision support process where we look in large databases for unknown and unexpected patterns of information. – Parsaye [38]

- Data mining is the process of discovering advantageous patterns in data [50].

From the beginning, data mining research has been driven by its applications. While the finance and insurance industries have long recognized the benefits of data mining, data mining techniques can be effectively applied in many areas. Recently, research in association rule mining, classification rule mining, and pattern discovery in combinatorial databases has provided many paradigms that are applicable to wide application areas. As a result, more and more organizations have become interested in data mining.

Data mining is computationally expensive by nature, as will be made evident by data mining algorithms described in later chapters. However, since the benefits of data mining results remain largely unpredictable, these organizations may not be willing to buy new hardware for that purpose. Therefore, acquiring computing resources needed by data mining applications is potentially a problem for these organizations. On the other hand, there is a huge amount of idle cycles in the machines these organization already have. Therefore, we have the need for systems that can harvest idle cycles for efficient data mining.

1.2 Approach and Contributions

Our approach to efficient data mining is parallelization, where the whole computation is broken up into parallel tasks. The work done by each task, often called its *grain size*, can be as small as a single iteration in a parallel loop or as large as an entire procedure. When an application can be broken up into large parallel tasks, the application is called a *coarse grain parallel application*. Two common ways to partition computation are *task partitioning*, in which each task executes a certain function, and *data partitioning*, in which all tasks execute the same functions but on different data. We explore both task parallelism and data parallelism in data mining applications.

Because of their wide availability and cost effectiveness, networks of workstations (NOW) have recently emerged as a promising computing platform for long-running, coarse grain parallel applications, such as data mining applications. Parallel data mining algorithms implemented on NOW provide an efficient and inexpensive solution to large scale data mining on non-dedicated hardware. *Persistent Linda* (PLinda)[20, 47, 49, 1], a distributed parallel computing system on NOW, is particularly suitable for data mining applications. PLinda automatically utilizes idle workstations which makes parallel data mining virtually free. Our implementations of parallel data mining programs use PLinda as the computing platform.

The main contributions of this thesis are:

1. **A framework for exploring task parallelism in data mining applications:**

   We present a framework for representing computation models of data mining algorithms—*Exploration Dag (E-dag)*. We identify four basic elements of a data mining application and show how the computation in the context of the four elements can be mapped to an E-dag. We give a generic parallel E-dag traversal

algorithm to optimally solve data mining problems mapped into E-dags. The effectiveness of the E-dag framework is demonstrated in biological pattern discovery applications [2].

2. **A framework for exploring data parallelism in data mining applications:**

   We present a framework to partition data in a data mining application such that each parallel thread runs the same program with a different partition of the data and the results are then combined together. Classification tree algorithms, in particular, can readily take advantages of data partitioning.

3. **A new classification tree algorithm:**

   We present a new classification tree algorithm, *NyuMiner*, for finding optimal splits for both numerical and categorical variables with respect to any given impurity function and any given maximum number of branches allowed in a split. NyuMiner generally achieves higher accuracy than classification tree algorithms having no optimality guarantee. Because the tree structures generated by NyuMiner are usually different, it can be used with other classifiers in a complementary way to obtain higher classifying confidence [4].

4. **A software architecture for running data mining applications on networks of workstations:**

   We present a software architecture based on PLinda which enables parallel data mining programs to run on networks of workstations in a fault-tolerant manner [3]. We provide templates for writing parallel data mining programs in PLinda and describe how to run these programs on a network of workstations.

1.3 Organization of Thesis

The rest of the thesis is organized as follows.

Chapter 2: We survey state-of-the-art algorithms in classification rule mining, association rule mining, and combinatorial pattern discovery. We discuss previous attempts to parallelize algorithms in these fields. We argue that the suitable platform for efficient and inexpensive data mining is networks of workstations (NOW). We also survey four distributed computing systems on NOW.

Chapter 3: This chapter describes the E-dag framework for task parallelism in data mining algorithms. We identify the four elements that each data mining problem has and describe how to map them into an E-dag. We then present a parallel E-dag traversal algorithm and discuss issues related to optimal parallel E-dag traversal.

Chapter 4: We describe the application of the E-dag framework to two biological pattern discovery applications, *discovery of motifs in protein sequences* and *discovery of motifs in RNA secondary structures.* We also present experimental results of various implementation strategies used in protein sequence pattern discovery.

Chapter 5: This chapter presents a new classification tree algorithm named *NyuMiner* which guarantees an optimal split at every tree node with respect to any given impurity function and any given maximum number of branches allowed in a split. We compare the classification accuracy of NyuMiner to those of C4.5[65] and CART[24], two popular classification tree algorithms. We also present the results of complementarity tests among NyuMiner, C4.5 and CART. Finally, we present the results of using NyuMiner to analyze historical foreign exchange rates.

Chapter 6: This chapter explores data parallelism in classification tree algorithms. We describe the PLinda implementations of Parallel NyuMiner and Parallel C4.5 and present experimental results.

Chapter 7: The chapter describes the software architecture used to implement parallel data mining applications. We discuss issues related to fault-tolerance of data mining programs implemented in PLinda. We also briefly describe how to run data mining programs on a PLinda-enabled network of workstations. Finally, we give URL's for downloading our PLinda data mining programs.

Chapter 8: This chapter summarizes the whole thesis and gives directions for future work.

Chapter 2

Related Work

A large amount of data mining research focuses on association rule mining, classification rule mining, and pattern discovery in combinatorial databases. Section 2.2, 2.1, and 2.3 survey related work in these three areas, including previous attempts to parallelize the algorithms surveyed. Section 2.4 argues why networks of workstations (NOW) are a suitable platform for parallel data mining. Four distributed computing systems that run on NOW are surveyed in section 2.4.

2.1 Classification Rule Mining

2.1.1 An Example

Let's take a look at the following imaginary heart disease diagnosis records for the PLinda group (Table 2.1). Karp wasn't here, so he missed the doctor. But can we decide whether he has heart disease or not?

Figure 2.1 is a *decision tree* built from all but Karp's records. According to this decision tree, Karp does not have heart disease (but he should go see a doctor anyway).

Name	Weight (lb.)	Age	Blood Pressure (BP)	Heart Disease?
Jihai	180	27	Low	Yes
Tom	140	20	Low	No
Hansoo	150	30	Med	No
Peter	150	31	Low	No
Bin	150	35	High	Yes
Dennis	150	62	Low	Yes
Karp	140	32	Low	?

Table 2.1: Imaginary heart disease diagnosis records for the PLinda group.

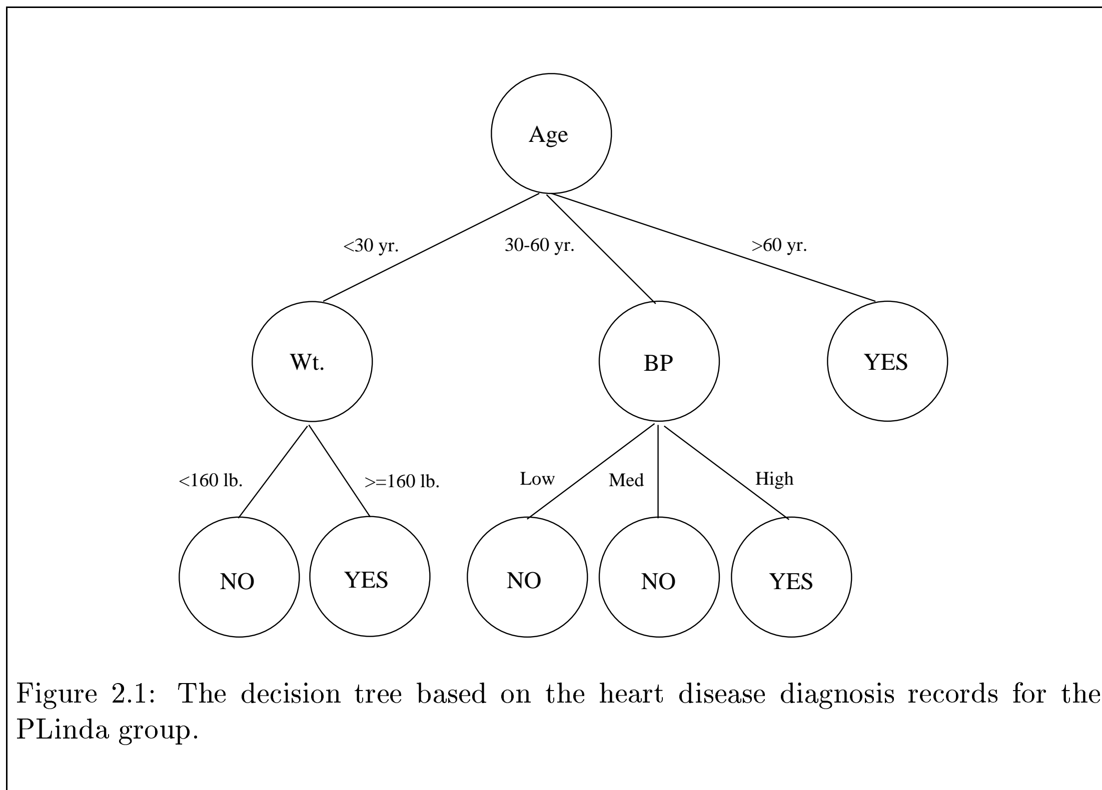


Figure 2.1: The decision tree based on the heart disease diagnosis records for the PLinda group.

Rules like the following can also be induced from the tree:

$$(Age > 60) \rightarrow Yes$$

$$(Age < 30 \;\&\; Wt. \leq 160) \rightarrow No$$

2.1.2 Classification Rules

Classification rules are of the form: $D \rightarrow C$, where D is a sufficient condition (formed of a conjunction of attribute–value conditions) to classify objects as belonging to class C. Classification rules are sometimes represented by decision trees.

2.1.3 Decision Trees

A decision tree is a tree data structure with the following properties:

- Each leaf is labeled with the name of a class;
- The root and each internal node (called a *decision node*) is labeled with the name of an attribute;
- Every internal node has a set of at least two children, where the branches to the children are labeled with disjoint values or sets of values of that node's attribute such that the union of these constitutes the set of all possible values for that attribute.

Thus, the labels on the arcs leaving a parent node form a partition of the set of legal values for the parent's attribute.

A decision tree can be used to classify a case by starting at the root of the tree and moving through it until a leaf is reached [65]. At each decision node, the case's outcome for the test at the node is determined and attention shifts to the root of the

subtree corresponding to this outcome. When this process finally (and inevitably) leads to a leaf, the class of the case is predicted to be that labeled at the leaf.

2.1.4 Building Decision Trees

Every successful decision tree algorithm (e.g. CART [24], ID3 [63], C4.5 [65]) is an elegantly simple greedy algorithm:

1. pick as the root of the tree the attribute whose values best separate the training set into subsets (the best partition is one where all elements in each subset belong to the same class);

2. repeat 1 recursively for each child node until a stopping criterion is met.

Examples of stopping criteria are:

- every subset contains training examples of only one class;

- the depth of the tree reaches a predefined threshold;

- lower bound on the number of elements that must be in a set in order for that set to be partitioned further is reached (CART [24], C4.5 [65]).

The dominating operation in building decision trees is the gathering of histograms on attribute values. As mentioned earlier, all paths from a parent to its children partition the relation horizontally into disjoint subsets. Histograms have to be built for each subset, on each attribute, and for each class individually.

2.1.5 Attribute Selection

Attribute selection is performed in three steps:

1. Build histograms on attribute values;

2. Assign scores to attributes according to a quality metric:

   - **Information Gain Metric** [63, 65]

     The information conveyed by a message depends on its probability and can be measured in bits as minus the logarithm to base 2 of that probability. So, for example, if there are eight equally probable messages, the information conveyed by any one of them is $-\log_2(1/8)$ or 3 bits. Thus, if T is a set of training cases, the average amount of information needed to identify the class of a case in T is defined as:

     $$info(T) = -\sum_{i=1}^{c} \frac{n_i}{N} \log_2 \frac{n_i}{N}$$

     where c is the number of classes, n_i is the number of examples in class i, and $N = \sum_{i=1}^{c} n_i$.

     Now consider a similar measurement after T has been partitioned in accordance with the v possible values of an attribute A. The expected information requirement can be found as the weighted sum over the subsets, as

     $$info_A(T) = \sum_{j=1}^{v} \frac{\sum_{i=1}^{c} n_{ij}}{N} \times info(T_j)$$

     where n_{ij} is the number of examples of class i having attribute value j and T_j is the jth partition. So $\sum_{i=1}^{c} n_{ij}$ is the total number of examples of all classes having attribute value j. The quantity

     $$gain(A) = info(T) - info_A(T)$$

     measures the information that is gained by partitioning T in accordance with the attribute A.

The *gain criterion* is used in ID3. Although it gives quite good results, this criterion has a serious deficiency—it has a strong bias in favor of tests with many outcomes. The bias inherent in the gain criterion can be rectified by a kind of normalization in which the apparent gain attributable to tests with many outcomes is adjusted. Consider the information content of a message pertaining to a case that indicates not the class to which the case belongs, but the outcome of the test:

$$split\ info(A) = -\sum_{i=1}^{v} \frac{\sum_{i=1}^{c} n_{ij}}{N} \times \log_2 \frac{\sum_{i=1}^{c} n_{ij}}{N}.$$

This represents the potential information generated by dividing T into v subsets, where the information gain measures the information relevant to classification that arises from the same division. Then,

$$gain\ ratio(A) = gain(A)/split\ info(A)$$

expresses the proportion of information generated by the split that is useful, i.e., that appears helpful for classification.

The gain ratio criterion, used in C4.5, is robust and typically gives a consistently better choice of test than the gain criterion [64].

3. Pick the attribute with the highest score.

2.1.6 Parallelization Attempts

There have been two different approaches to parallelizing decision tree algorithms. The first approach is to use a parallel database management system as a back end. Holsheimer and Kersten [43] used this approach with the Monet [53, 22] database server developed at CWI. In their implementation, a database is partitioned vertically into

Binary Association Tables and the Monet server processes search queries in parallel. The search process is controlled by a front-end mining tool. Their experimentation platform is a 6-node SGI machine with 150MHz processors and 256 MB of main memory. Up to 4 nodes are used in their experiments and they achieved a speedup of 2.5 when 4 nodes are used.

The second approach is a divide–and–conquer scheme used in Meta-Learning [31, 32]. In this approach, a database is horizontally divide into subsets and sequential mining algorithms are applied to each subset. A tree of *arbiter* then combine results from individual classifiers into final results (Figure 2.2). In order to get good final results, arbiters have to be trained to suit the set of classifiers. This takes $\log s$ iterations if the databases is partitioned into s subsets. Therefore, the theoretical speedup of this approach is $O(s/\log s)$. They have achieved this theoretical speedup on a network of Sun IPXs. However, training of arbiters is not done automatically.

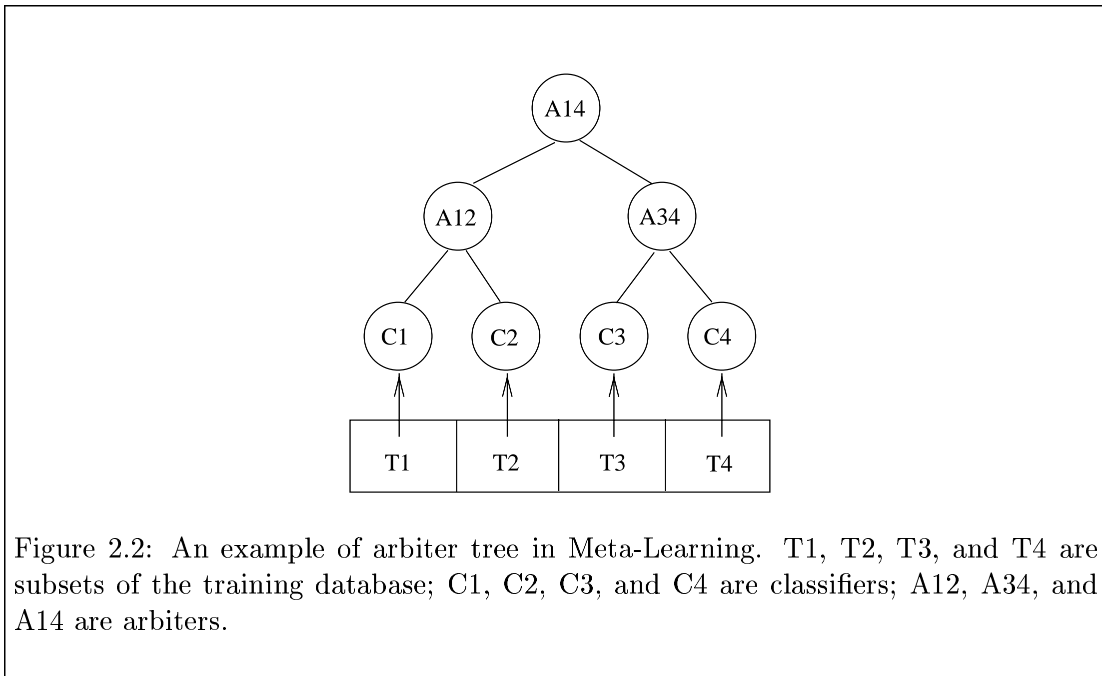


Figure 2.2: An example of arbiter tree in Meta-Learning. T1, T2, T3, and T4 are subsets of the training database; C1, C2, C3, and C4 are classifiers; A12, A34, and A14 are arbiters.

Trans ID	Items
1	pamper, soap, lipstick
2	soda, pamper, lipstick, candy
3	beer, soda
4	beer, candy, pamper

Table 2.2: An imaginary sales transaction database from K-mart.

The first approach is not suited for networks of workstations. The second approach is natural for networks of workstations because each individual classifier can run independently on a workstation. However, we are more interested in parallelizing the mining algorithms themselves. Clearly, building histograms on attribute values and computing gain ratios for attributes can be done in parallel.

2.2 Association Rule Mining

2.2.1 An Example

Let's look at the following imaginary sales transaction database from K-mart (Table 2.2).

Here is an association rule mined from the above database:

$$\mathrm{PL} : (pamper) \rightarrow (lipstick)$$

which, translated into plain English, means that pampers sell well (75% of all trans), and lipsticks usually (67% of the time) go with them.

2.2.2 Association Rules

Let $\mathcal{L} = \{i_1, i_2, \ldots, i_m\}$ be a set of m distinct literals called *items*. D is a set of variable length transactions over $\mathcal{L}$. An *association rule* is an implication of the form

$$\mathrm{R} : X \rightarrow Y$$

where $X, Y \subset \mathcal{L}$, and $X \cap Y = \emptyset$. X is called the *antecedent* and Y is called the *consequent* of the rule.

In general, a set of items (such as the antecedent or the consequent of a rule) is called an *itemset*. The number of items in an itemset is called the *length* of the itemset. Itemsets of some length k are referred to as k-itemsets.

Each itemset has an associated measure of statistical significance called *support* defined as follows.

$$supp(X) = \text{ \# of transactions containing } \textit{all} \text{ items in } X$$

If $supp(X) \geq s_{min}$ for a given minimum support value s_{min}, the set X is called *frequent*.

A rule has a measure of its strength called the *confidence* defined as follows.

$$conf(R) = \frac{supp(X \cup Y)}{supp(X)}$$

The support for rule R is defined as $supp(X \cup Y)$. A rule R *holds* with respect to some fixed minimum confidence level c_{min} and a fixed minimum support s_{min} if $conf(R) \geq c_{min}$ and $supp(R) \geq s_{min}$. Note that as a necessary condition for a rule to hold, both the antecedent and consequent of the rule have to be frequent.

2.2.3 Properties of Frequent Sets

Four properties of frequent sets form the foundation of all association rule mining algorithms.

1. *Support for Subsets*

   If $A \subseteq B$ for itemsets A, B, then $supp(A) \geq supp(B)$ because all transactions in $\mathcal{D}$ that support B necessarily support A also.

2. *Supersets of Infrequent Sets are Infrequent*

   If itemset A lacks minimum support in $\mathcal{D}$, i.e., $supp(A) < s_{min}$, every superset B of A will not be frequent either because $supp(B) \leq supp(A) < s_{min}$ according to property 1.

3. *Subsets of Frequent Sets are Frequent*

   If itemset B is frequent in $\mathcal{D}$, i.e., $supp(B) \geq s_{min}$, every subset A of B is also frequent in $\mathcal{D}$ because $supp(A) \geq supp(B) \geq s_{min}$ according to property 1. In particular, if $A = i_1, i_2, \ldots, i_k$ is frequent, all its k $(k-1)$-subsets are frequent. Note that the converse does not hold.

4. *Inferring Whether Rules Hold* [18]

   For itemsets L, A, B and $B \subseteq A$,

   $$conf(A \rightarrow (L - A)) < c_{min} \Longrightarrow conf(B \rightarrow (L - B)) < c_{min}.$$

   Using $supp(B) \geq supp(A)$ (property 1) and the definition of confidence we obtain $conf(B \rightarrow (L - B)) = \frac{supp(L)}{supp(B)} \leq \frac{supp(L)}{supp(A)} < c_{min}$. Likewise, if a rule $(L - C) \rightarrow C$ holds, so does $(L - D) \rightarrow D$ for $D \subseteq C$ and $D <> \emptyset$.

The last property will be used to speed up the generation of rules once all frequent sets and their support is determined. See section 2.2.5 for details of the algorithms.

2.2.4 Basic Scheme for Association Rule Mining

An association rule mining process consists of two phases:

```
Phase I.    Find all frequent itemsets—generate-and-test.
Phase II.   Rule construction:
            for every frequent itemset X
                for every Y ⊂ X
                    if conf(Y → (X − Y)) ≥ c_min then
                        new rule
                    else
                        no subset of Y needs to be considered
                    end if
                end for
            end for
```

2.2.5 State of the Art

Phase II is relatively straightforward. Various algorithms have been developed for Phase I of the mining process. Among them, Apriori [18, 19] and Partition [66] are more promising.

Apriori-gen has been so successful in reducing the number of candidates that it is used in every algorithm that has been proposed since it was published [42, 62, 66, 67].

Partition takes a divide-and-conquer approach. Its four major steps are:

1. partition the database horizontally;
2. generate all frequent itemsets for each partition using Apriori;
3. merge these itemsets to get global candidate itemsets;
4. generate global frequent itemsets.

Apriori

for $k = 1$ to $|\mathcal{L}| - 1$ **do**

1. generate candidate $(k+1)$-itemsets from frequent k-itemsets

   —*Apriori-gen*

   for each pair of frequent k-itemsets I_1 and I_2 that have their $k-1$

   smallest items in common

   generate a prospective $(k+1)$-itemset $S = I_1 \cup I_2$

   if every k-itemset $\subset S$ other than I_1 and I_2 is also frequent **then**

   insert S into the set of candidate $(k+1)$-itemsets

   end if

   end for

2. count support for each candidate $(k+1)$-itemset and determine if it is frequent

end for

2.2.6 Parallelization Attempts

There have been a few attempts to parallelize association rule mining algorithms. PEAR and PPAR [61], for example, parallelize Apriori and Partition.

PEAR and **PPAR**—parallel Apriori and Partition

- Modifications to Apriori: new data structure—prefix tree (stores frequent itemsets and candidate itemsets together), optimizations—dead branches and pass bundling.

- Parallel program platform: IBM SP2 (SPMD, Message Passing Library).
- Basic parallel scheme: processors count local support and compute global support for candidates in parallel; candidate generation (Apriori-gen) is done sequentially.
- Speedup: 3.7 on 4 nodes and 12.2 on 16 nodes.

Although the implementation of PEAR and PPAR is on a massive parallel computer, the parallel scheme can be effectively implemented on networks of workstations.

2.3 Pattern Discovery in Combinatorial Databases

2.3.1 An Example

Consider a toy database of sequences $\mathcal{D}$={FFRR, MRRM, MTRM, DPKY, AVLG} and the query "Find the patterns P of the form $*X*$ where P occurs in at least 2 sequences in $\mathcal{D}$ and $|P| \geq 2$." (X can be a segment of a sequence of any length, and $*$ represents a variable length don't care.) The good patterns are $*$RR$*$ (which occurs in FFRR and MRRM) and $*$RM$*$ (which occurs in MRRM and MTRM).

2.3.2 Pattern Discovery

Combinatorial databases store structures such as sequences, trees, and graphs. Such databases arise in biology, chemistry, linguistics, document retrieval, botany, neuroanatomy, and many other fields. An important set of queries on such a database concern the comparison and retrieval of similar combinatorial structures. Such *pattern matching* queries are useful when comparing a new structure to an existing set. A less constrained but equally important problem is *pattern discovery*—find the pattern that approximately characterizes a set of structures in the database given a pattern metric.

2.3.3 Sequence Pattern Discovery

Pattern discovery in sets of sequences concerns finding commonly occurring (or *active*) subsequences (sometimes called *motif*s). The structures of the motifs we wish to discover are regular expressions of the form $*S_1 * S_2 * \ldots$ where $S_1, S_2, \ldots$ are *segments* of a sequence, i.e., subsequences made up of consecutive letters and $*$ represents a variable length don't care (VLDC). In matching the expression $*S_1 * S_2 * \ldots$ with a sequence S, the VLDCs may substitute for zero or more letters in S. Segments usually allow certain number of mutations; a mutation is a insertion, a deletion, or a mismatch.

Let $\mathcal{S}$ be a set of sequences. The occurrence number (or activity) of a motif is the number of sequences in $\mathcal{S}$ that match the motif within the allowed number of mutations. We say the occurrence number of a motif P with respect to mutation i and set $\mathcal{S}$, denoted $occurrence_no^i_{\mathcal{S}}(P)$, is k if $*P*$ matches k sequences in $\mathcal{S}$ within at most i mutations, i.e., the k sequences contain P within i mutations. Given a set $\mathcal{S}$, we wish to find all the active motifs P where P is within the allowed Mut mutations of at least $Occur$ sequences in $\mathcal{S}$ and $|P| \geq Length$, where $|P|$ represents the number of the non-VLDC letters in the motif P. (Mut, $Occur$, $Length$ and the form of P are user-specified parameters.)

2.3.4 Sequence Pattern Discovery Algorithm

The best sequence pattern discovery algorithm was developed by Wang *et al* [68]. The basic subroutine in the algorithm is to match a given motif against a given sequence after an optimal substitution for the VLDCs in the motif. The algorithm consists of two phases: (1) find candidate segments among a small sample $\mathcal{A}$ of the sequences; (2) combine the segments to form candidate motifs and evaluate the activity of the motifs

in all of $\mathcal{S}$ to determine which motifs satisfy the specified requirements.

Phase (1) consists of two subphases. In subphase A, a *generalized suffix tree* [44] (GST) for the sample of sequences is constructed. A suffix tree is a trie-like data structure that compactly represents a string by collapsing a series of nodes having one child to a single node whose parent edge is associated with a string [60, 54]. A GST is an extension of the suffix tree, designed for representing a set of strings. Each suffix of a string is represented by a leaf in the GST. Each leaf is associated with an index i. The edges are labeled with character strings such that the concatenation of the edge labels on the path from the root to the leaf with index i is a suffix of the ith string in the set. The GST can be constructed asymptotically in $O(n)$ time and space where n is the total length of all sequences in the sample $\mathcal{A}$.

In subphase B, we traverse the GST constructed in subphase A to find all segments (i.e., all prefixes of strings labeled on root-to-leaf paths) that satisfy the length minimum. If the pattern specified by the user has the form $*X*$, then the length minimum is simply the specified minimum length of the pattern. If the pattern specified by the user has the form $*X_1 * X_2*$, we find all the segments V_1, V_2 where at least one of the V_i, $1 \leq i \leq 2$, is larger than or equal to half of the specified length and the sum of their lengths satisfies the length requirement. If the user-specified pattern has the form $*X_1 * X_2 * \ldots * X_k*$, we find the segments $V_1, V_2, \ldots, V_k$ where at least one of the V_i, $1 \leq i \leq k$, is larger than or equal to $1/k$th of the specified length and the sum of the lengths of all these segments satisfies the length requirement.

An important optimization heuristic that was implemented is to eliminate the redundant calculation of occurrence numbers. Observe that the most expensive operation in the discovery algorithm is to find the occurrence number of a motif with respect to the entire set, since that entails matching the motif against all sequences.

We say $*U_1 * \ldots * U_m*$ is a *subpattern* of $*V_1 * \ldots * V_m*$ if U_i is a subsegment of V_i, for every $1 \leq i \leq m$. One can observe that if motif P is a subpattern of motif P', $occurrence_no^k_{\mathcal{S}}(P) \geq occurrence_no^k_{\mathcal{S}}(P')$ for any mutation parameter k. Thus, if $P^{'}$ is in the final output set, then we need not bother matching P against sequences in $\mathcal{S}$. If P is not in the final output set, then $P^{'}$ won't be either, since its occurrence number will be even lower.

2.4 Platforms for Parallel Data Mining

2.4.1 Networked Workstations as a Parallel Computing Platform

Recently, networks of workstations (NOW) have emerged as a promising parallel computing platform. Their advantages over massively parallel computers are wide availability and cost-effectiveness. First, unlike supercomputers installed in a few institutions, these machines are widely available; many institutions have hundreds of high performance workstations which are unused most of the time. Second, they are already paid for and are connected via communication networks; no additional cost is required for parallel processing. Finally, they can rival supercomputers with their aggregate computing power and main memory.

However, most machines are "private" and the owners of these workstations do not want to allow compute-intensive jobs to be run on their machines for fear of performance degradation of their own processes. Therefore, it is crucial to guarantee that workstations will be used only while they are idle. Also, it is necessary for a distributed computing systems to run on networks of heterogeneous workstations because most institutions have heterogeneous workstations in their computing environments.

When networks of workstations are used for parallel computing, the probability of

failure grows as execution time or the number of processors increases. Since the suitable applications for networks of workstations are long-running, coarse grain applications, fault tolerance is crucial for a distributed computing system that runs on networks of workstations. Without fault tolerance, a single component failure can cause an entire computation to be lost.

2.4.2 Parallelism in Data Mining

Various pruning mechanisms are used extensively in data mining applications. Because a powerful pruning mechanism leads to a highly variable search process that conflicts with a uniform workload requirement for good performance, many data mining applications have proved difficult to parallelize. Fortunately, the computation is coarse grain parallel, i.e., it can be parallelized into large, seldom interacting tasks. Coarse grain parallel computations are suitable computations for networks of workstations [48]. The time it takes to create, coordinate, and terminate parallel tasks is critical. Therefore a good parallel programming environment and runtime environment is critical to parallel data mining. In the remainder of this section, we survey four distributed computing systems that run on networks of workstations, namely Condor, Calypso, Piranha, and Persistent Linda.

2.4.3 Condor

Condor, developed at the University of Wisconsin, aims at utilizing idle workstations for background sequential jobs [57]. It is a distributed resource management system that can manage large heterogeneous clusters of workstations. Its design has been motivated by the needs of users who would like to use the unutilized capacity of such clusters for their long-running, computation-intensive jobs.

Condor pays special attention to the needs of the interactive user of the workstation. It is the interactive user who defines the conditions under which the workstation can be allocated by Condor to a batch user. Condor preserves a large measure of the originating machine's environment on the execution machine, even if the originating and execution machines do not share a common file and/or password systems. Condor jobs, which consist of a single process, are automatically checkpointed and migrated between workstations as needed to ensure eventual completion.

The Condor System is not a multiprocessing system per se, but it supports parallelism in the sense that multiple independent processes may be run in parallel. Also, the checkpointing capability allows fault tolerance.

Due to the limitations of the remote execution and checkpointing mechanisms, there are several restrictions on the type of program which can be executed by the Condor facility. Most importantly only single process jobs are supported, (i.e. the $fork()$ and $exec()$ system calls are not supported). Secondly, signal and IPC calls are not implemented, (e.g. $signal()$, $kill()$, and $socket()$ calls will fail). Therefore, there is no communication primitive among Condor processes.

There are also some practical limitations regarding the use of disk space for execution of Condor jobs. During the time when a job is waiting for a workstation to become available, it is stored as a "checkpoint file" on the machine from which the job was submitted. The file contains all the text, data, and stack space for the process, along with some additional control information. This means that jobs which use a very large virtual address space will generate very large checkpoint files. Some advantage can be gained by submitting multiple jobs which share the same executable as a single unit or "job cluster". In that case, all jobs in a cluster share a single copy of the checkpoint file until they begin execution. Also the workstations on which the jobs will actually

execute often have very little free disk. Thus it is not always possible to transfer a Condor job to a machine, even though that machine is idle. Since large virtual memory jobs must wait for a machine that is both idle, and has a sufficient amount of free disk space, such jobs may suffer long turnaround times.

2.4.4 Calypso

CALYPSO is a shared memory parallel language currently under development at New York University [21]. The language is novel in that it addresses fault tolerance and provides high performance with the same mechanism: eager scheduling supported by evasive memory.

CALYPSO programs are written in the CALYPSO Source Language (CSL) which is C++ extended with shared variables and parallel threads. Shared variables must be in a struct defined at the file level. The syntax for creating parallel threads is as follows.

parbegin

  $routine_1$[number of instances] {

    C++ statements

  }

  ...

  $routine_n$[number of instances] {

    C++ statements

  }

parend

Routines can concurrently read shared variables and each shared variable may

be written by at most one routine. This is known as concurrent read and exclusive write (CR&EW). Parallel routines are not allowed to do any sort of input or output and nested parbegins are not implemented.

The CALYPSO system consists of four components: a pre-processor, the *progress manager*, the *memory manager* and the *compute server(s)*. In the current implementation the memory manager and the progress manager are the same process. We separate the explanation of their functionality as much as possible, although there is some overlap due to shared data structures.

The run-time CALYPSO system consists of the centralized memory manager/ progress manager and at least one compute server. The memory manager/progress manager process executes the sequential code of the CALYPSO program, services the shared memory requests of the compute servers, and schedules parallel routines on the compute servers. The memory manager has four states: executing sequential code, calling the progress manager function with information on a parallel step that needs to be executed, servicing the shared memory requests of the compute servers, and updating the shared memory after a parallel step. Each compute server has three states: executing a routine, sending dirty shared variable updates to the memory manager upon completion of a routine, and waiting for the progress manager to send the next routine.

For any given CALYPSO program, there is only one executable. The executable can function as a compute server or memory manager. This simplifies shared memory management—virtual memory addresses are the same in the compute servers and the memory manager.

The C++ statements of each routine in a parbegin are wrapped into a function by the pre-processor. The function takes as parameters the number of instances of the routine (The number of instances may not be known at compile-time since it can be

```
int main()
{
    SHARED {
        int result[100];
        int a[100];
        int b[100];
    }

    parbegin
        routine doaddition(int numHosts,int me)[5] {
            int offset = me * 20;
            for(int i = 0 ; i < 20 ; i++)
                result[offset + i] = a[offset + i] + b[offset + i];
        }
    parend;
}
```

Figure 2.3: A vector addition program in Calypso.

a general C++ expression) and the routine id which ranges from 0 to the number of instances − 1. These functions along with a standard template that executes the three states of a compute server are what the pre-processor generates for the compute servers.

The memory manager consists of the original source program with each parbegin converted into a call to the progress manager with instructions on what routines need to be run, code to handle compute server shared memory requests and code to update the shared data after a parallel step.

The progress manager function takes as parameters the address and number of instances of each function to be executed in a parallel step. From this information, the progress table is constructed. The rows contain the addresses of functions and the columns indicate whether the function has been executed at least once, whether this function has been assigned to any compute server yet, the instance number and total

number of instances for this function, and a linked list of the pages that were modified by this function. The progress manager gives each idle compute server the address of a function and the instance number information through TCP/IP. Because the memory manager and compute server are the same executable, the virtual address of the function is the same in both processes.

The CALYPSO progress manager uses eager scheduling to allocate routines to processors. Once all unstarted routines have been allocated to compute servers, routines that have been started but not yet completed will be allocated. CALYPSO guarantees that an execution schedule in which a routine is executed multiple times produces the same result as an execution in which the routine is executed once. This is why threads are not allowed to do I/O or have any interaction with the outside world. The benefit of eager scheduling is that a computation will not be slowed down by slow or busy processors. This can be especially useful in a networked workstation environment where machines may become busy with other user processes. This provides the same effect as the the bag-of-tasks and master-worker paradigm in Linda when the tasks are small. The benefit of CALYPSO's method is apparent when there are only a few tasks left; it ensures that the fast machines will not be left idle while waiting for slow machines to finish the tasks. There are three drawbacks to this approach. First, redundant computation entails extra communication and the network is a limited resource. Second, hosts which are doing a redundant task need to be notified if that task is completed by another machine before they finish. Third, redundant task scheduling means extra contention for a centralized server. For example, if task A is replicated on three machines and task A requires 10 server requests (we are assuming that each machine needs the same amount of requests to run the task, which may not be true because of caching) to the server, the replication will cause 20 extra requests at the server.

Because a failed processor is an infinitely slow processor, fault tolerance is a side effect of eager scheduling.

Evasive memory allows writes to shared variables to be idempotent. Each write goes to a new location along with a time-stamp. When servicing a read operation the evasive memory returns the value of the variable corresponding to the time step of the read. Thus, slow processors cannot clobber the memory with out of date values.

The progress manager does not mark a routine as completed until all the modifications it made to shared memory are communicated by the compute server that executed the routine. Once a routine is marked as complete, subsequent data from other compute servers that may have been working on that routine are ignored. This is what provides the illusion of evasive memory.

Page protection, *mprotect*(), provides CALYPSO with an easy implementation of the shared memory. The compute servers protect the memory pages that the shared variables reside on. A page fault occurs whenever a compute server accesses a shared variable. In the case of a read, the compute server requests the page from the memory manager. In the case of a write, if the page is old [1] the compute server requests the page from the memory manager. For a first-time write, the page must be twinned (copied) so that upon completion of the routine the compute server can send the memory manager the diff of the old page and the updated page. Sending only the diff is actually crucial, since different threads may update different parts of the same page.

Locality of reference is exploited in CALYPSO. The memory manager records the time step of each write to every page. This information is sent to the compute servers at the beginning of each parallel step. The compute servers keep information about the time step number of each page they have in their local memory. In this way, they

[1] Old in that the compute server does not have an up to date copy of this page.

know if a page has been modified since the last time they received it from the memory manager.

The CALYPSO model is more powerful than SIMD and is indeed MIMD, but it does not provide synchronization primitives or asynchronous communication between routines. CALYPSO has yet to be implemented on networks of heterogeneous workstations.

2.4.5 Piranha

Piranha [52] is based on the Linda system developed at Stony Brook and Yale by Gelernter and Carriero [26, 27, 29, 30, 55]. Linda also being the base of the Persistent Linda system we describe in section 2.4.6, it is necessary for us to briefly review Linda.

Linda

Linda is a set of extensions which can be added to any programming language. The extensions consist of six operations which provide a means for process creation, communication, and coordination. The beauty of Linda is that the extensions are orthogonal to the sequential semantics of the language in which they are used, and the extensions are simple and yet provide a powerful programming model.

Linda's shared memory is generative—meaning that it is generated and destroyed as a computation proceeds. Each memory entity is a tuple. The memory is addressed by field matching, rather than a static memory addressing scheme. The shared memory is called *Tuple Space* and has been implemented both as a centralized server and as decentralized servers. We discuss the decentralized approach.

Generative communication results in un-coupled communication: two processes can communicate even though they are not executing at the same time and they can do

so anonymously. This frees the programmer from some of the nuances of coordination at the cost of an extra level of indirection which hurts performance.

```
void parallelVectorAddition(int a[100],int b[100], int result[100])
{
    for(i = 0 ; i < 5 ; i++)
        eval("doaddition", i, &a[i] : 20,&b[i] : 20)
    for(i = 0 ; i < 5; i++)
        in("result", i, ? &result[i]);
}
```

Figure 2.4: Vector addition master in Linda.

```
void doaddition(int me, int a[20], int b[20])
{
    int result[20];
    for(int i = 0 ; i < 20; i++)
        result[i] = a[i] + b[i];
    out("result", me , result : 20);
}
```

Figure 2.5: Vector addition slave in Linda.

A *tuple* is a sequence of typed values. A *pattern* is a sequence of types. A *template* is a pattern and a sequence of values for some of the types in the pattern. The fields of the template whose values are specified are *actuals* and the other fields are *formals*.

Here is a description of the Linda operations:

- `in`(template): If a tuple with the same pattern exists, and the values of the corresponding actual fields match, assign the corresponding values of the tuple to the formals. Otherwise, block until a match exists. The tuple is destroyed from tuple space.

- `inp`(template): the same as `in` except do not block if the tuple does not exist. True is returned if the match is successful and false is returned on failure.

- `rd`(template): the same as `in` except do not destroy the tuple in tuple space.

- `rdp`(template): the same as `inp` except do not destroy the tuple in tuple space.

- `out`(tuple): create a tuple in tuple space.

- `eval`(tuple): create processes for each actual that is a function call and continue execution of the process that made the `eval` call. When all the spawned processes have terminated the tuple is created.

Linda supports the *master-worker* and the *bag-of-tasks* programming paradigms. In the master-worker paradigm, a master process has a large task that can be broken into smaller pieces. Worker processes are created to do these tasks. In the bag-of-tasks paradigm there are a number of tasks to be done and each process will get a task and do the task until there are no more tasks. In both schemes, the number of workers is much less than the number of tasks. This provides automatic load balancing: if one task is larger or some machines are slower, workers that are doing smaller tasks and/or workers on faster machines will do more tasks.

Implementation of X-Linda—where X is the base language that Linda is being added to—can be broken into two areas: compile time and run time.

A Linda compiler takes an X-Linda program and produces an X program with the Linda statements transformed into function calls. The compilation process involves first partitioning the tuples and templates into an equivalence relation and then classifying each partition of the relation [26].

The partitioning phase splits the tuples and templates in such a way that no

template in one partition can match a tuple in another partition. Thus, runtime matching is reduced and in a distributed tuple space implementation each partition could reside on a different machine.

The classification phase uses information about the number, the types, and the values of the fields in each tuple to determine the best data structure for storing each set of tuples. For example, if a partition consists of only the tuple/template ("ticket"), this partition can be represented as an integer counter.

The runtime implementation needs to do three things: handle client `out` requests, client `in` (`inp`, `rd` and `rdp` are treated in the same manner) requests, and client `eval` requests.

How to handle `in` and `out` requests depends on where tuples are stored. There are a number of ways in which tuples can be mapped to machines in a distributed implementation of Linda. These include replicating each tuple across all machines, mapping locally outed tuples to the local machine, and mapping outed tuples to all machines in the row of the machine that performed the out for mesh type architectures. For loosely-coupled systems, communication is expensive and the memory size on each machine is limited. Therefore, it is advantageous to have each tuple stored on only one machine and for each machine to know a priori which machine has all the tuples that match a given template. If tuples are mapped to machines using the equivalence relation produced during compilation, these properties are satisfied.

`Eval`s can be handled by compiling all the functions that will be used in `eval`s into one program with a system driver loop. The driver loops by `in`ing system tuples which denote that an `eval` has been requested. The appropriate function is then called and upon its return the function's result is `out`ed into tuple space.

Piranha

Piranha enables Linda programs to run on idle distributed workstations. Each task in a Piranha program is called a *Piranha*. When a user return to his or her workstation, the Piranha "retreats" after perhaps writing some state into tuple space. The Piranha system encourages programmers to write their program in a restricted form of the master-worker paradigm, whereby each Piranha process reads a work description "tuple", does some computation, outputs a tuple, then dies. Because data mining algorithms hold substantial state and each Piranha process must read that state, retreats can be very expensive. The current implementation of Piranha is not fully fault-tolerant and it does not run on networks of heterogeneous workstations.

Commercial products based on Linda and Piranha are available from Scientific Computing Associates.

2.4.6 Persistent Linda

Persistent Linda (PLinda) [20, 47, 48] is a set of extensions to Linda to support robust parallel computation. It is currently under development at New York University.

The three major extensions to Linda are:

- Lightweight transactions. Transactions are a sequence of operations that are always executed *atomically* regardless of failure. That is, all the operations take effect (called *commit*) or none of them do (called *abort*). No partial effect of a transaction is accessible to other transactions (or, processes) until the transaction commits. In PLinda, a process is executed as a series of transactions. If a process fails while executing a transaction, the runtime server detects the failure and aborts the transaction automatically. Runtime mechanisms ensure that no other

```
int main()
{
    int a[100],b[100],result[100],tranNumber = 0;
    if(xrecover(a : 100, b: 100, tranNumber))
         cout << "Master process recovering from failure" << endl;

    if(tranNumber == 0) {
        xstart;
        for(i = 0 ; i < 5 ; i++)
            proc_eval(SLAVENAME);
        out("task", i,  a[i * 20] : 20, b[i *20] : 20);
        xcommit(a,b, ++tranNumber);
    }

    if(tranNumber == 1) {
        xstart;
        for(i = 0 ; i < 5; i++)
            in("result", i, ?result[i * 20] : 20);
        xcommit(a,b,++tranNumber)
    }
}
```

Figure 2.6: Vector addition master in Persistent Linda.

```
int main()
{
    int a[20],b[20],result[20],which;
    while(1) {
        xstart;
        in("task", ?which, ? a : 20,? b : 20);
        for(int i = 0 ; i < 20; i++)
            result[i] = a[i] + b[i];
        out("result", me , result : 20);
        xcommit();
    }
}
```

Figure 2.7: Vector addition slave in Persistent Linda.

processes have accessed intermediate updates to the tuple space and the updates are removed from the tuple space on abort. Thus, the resulting effect appears as if the transaction had never occurred.

Lightweight transactions are used to maintain a consistent global state efficiently regardless of failure. A transaction is started with the `xstart` command and ended with the `xcommit` command. These transactions use locking for concurrency and recoverability control and therefore inhibit communication. That is, the effects of the `out`s and `eval`s in a transaction are not visible to other transactions until the `xcommit`.

- Continuation committing. Transactions commits result in writes to tuple space, but not to disk. The `xcommit` operation takes a tuple as a parameter. This tuple is used to save local state—*continuation*. The tuple is only accessible when the `xcommit` operation is finished and is only accessible using the `xrecover` operation.

  Continuation committing is used to make processes resilient to failure without relying on disk. A continuation consists of the live local variables of a process and an indication of which transaction last successfully completed. This information is saved at the end of each transaction by passing a tuple with these variables to the `xcommit` command.

- Checkpoint-protected tuple space. The PLinda runtime server manages the entire tuple space and saves it to disk periodically—**checkpointing**. If the server fails, it restores the latest checkpointed state from disk on recovery and resumes execution from that state—**rollback recovery**.

Also, PLinda replaces the `eval` statement of Linda with the `proc_eval` statement. The `proc_eval` statement takes as its parameters an executable file name and

command line arguments for that executable. This is so that standard Unix calls can be used for process creation. Using these three mechanisms and a process failure detection scheme, PLinda provides a tool-based fault tolerance scheme for Linda programs.

A PLinda program is divided into a sequence of transactions which are executed in an all-or-nothing manner by the runtime system. In this way the programmer need only worry about failures between transactions. To ensure recoverability, each transaction commit statement must include all live variables—those variables which may be used in the rest of the program without first being assigned. The PLinda system automatically detects process failure and reruns the process on a different machine. Using the `xrecover` operation, the programmer may retrieve the continuation of the last committed transaction and continue from that point.

In this way, the PLinda system can support fault tolerance in a heterogeneous environment. The reason is that no stack information is saved, only live variables, but the programmer must write code that can be chopped into transactions. This is a somewhat labor-intensive programming model.

2.4.7 Comparisons

In this section, we examined four systems for parallel programming on networks of workstations. Table 2.3 summarizes the features of these four systems.

Condor does not provide an abstract programming model for writing parallel programs. CALYPSO is easy to program and is efficient for applications which are highly data parallel. But, it does not provide mechanisms for processes to synchronize and communicate under programmer control; these types of mechanisms are needed to efficiently implement some algorithms [46]. Calypso is not fully fault-tolerant because it does not sustain failures of the machine where the compute server is running. Systems

	Condor	CALYPSO	Piranha	Persistent Linda
Parallel programming model	no	yes	yes	yes
Easy to program	yes	yes	no	no
Utilization of idle workstations	yes	yes	yes	yes
Fault tolerant	yes	somewhat	somewhat	yes
Heterogeneity	yes	no	no	yes

Table 2.3: Comparison of Condor, Calypso, Piranha, and Persistent Linda.

such as Piranha and Persistent Linda offer a trade off between high level constructs and efficiency. However, Persistent Linda achieves fault-tolerance and heterogeneity with lower overhead.

2.5 Other Parallel Computing Systems on NOW

2.5.1 Cilk-NOW

Cilk-NOW [6] is a runtime system designed to execute Cilk [5] program in parallel on a network of UNIX workstations. Cilk (pronounced "silk") is a parallel multi-threaded extension of the C language, and all Cilk runtime systems employ a provably efficient thread-scheduling algorithm. A Cilk program contains one or more *Cilk procedures*, and each Cilk procedure contains one or more *Cilk threads*. A Cilk procedure is the parallel equivalent of a C function, and a Cilk thread is a non-suspending piece of a procedure. The Cilk runtime system manipulates and schedules the threads. The runtime system is not aware of the grouping of threads into procedures.

A Cilk thread generates parallelism at runtime by *spawning* a child thread that is

the *initial thread* of a child procedure. After spawning one or more children, the parent thread must additionally spawn a *successor thread* to wait for the values "returned" from the children. The child procedures return values to the parent procedure by sending those values to the parent's waiting successor. Spawning successor and child threads is done with the `spawn_next` and `spawn` keywords, respectively. Sending a value to a waiting thread is done with the `send_argument` statement. Figure 2.8 shows how a function to calculate the Fibonacci numbers is written as a Cilk procedure consisting of two Cilk threads: `Fib` and `Sum`.

```
thread Fib (cont int k, int n)
{
    if (n<2)
        send_argument (k, n);
    else
    {
        cont int x, y;
        spawn_next Sum (k, ?x, ?y);
        spawn Fib(x, n-1);
        spawn Fib(y, n-2);
    }
}

thread Sum (cont int k, int x, int y)
{
    send_argument (k, x+y);
}
```

Figure 2.8: A Cilk procedure to compute the nth Fibonacci number. This procedure contains two threads, `Fib` and `Sum`.

The Cilk-NOW system adaptively executes Cilk programs on a dynamically changing set of otherwise-idle workstations. When a given workstation is not being used by its owner, the workstation automatically joins in and helps out with the execution

of a Cilk program. When the owner returns to work, the machine automatically retreats from the Cilk program. The Cilk-NOW runtime system automatically performs checkpointing, detects failures, and performs recovery while Cilk programs themselves remain *fault oblivious*. That is, Cilk-NOW provides fault tolerance without requiring that programmers code for fault tolerance.

It is important to note that Cilk-NOW provides these features only for Cilk-2 programs which are essentially functional. Cilk-NOW does not support more recent versions of Cilk (Cilk-3 and Cilk-4) that incorporate virtual shared memory, and in particular, Cilk-NOW does not provide any kind of distributed shared memory. In addition, Cilk-NOW does not provide fault tolerance for its I/O facility.

2.5.2 TreadMarks

TreadMarks [10] is a distributed shared memory (DSM) system for networks of UNIX workstations. It has been implemented on most UNIX platforms, including IBM RS-6000, DEC Alpha, HP, SGI, SUN Sparc.

TreadMarks' design focuses on reducing the amount of communication necessary to maintain memory consistency. To this end, it presents a *release consistency* memory model [8] to the user. Release consistency is a *relaxed* memory consistency model that permits a processor to delay making its changes to shared data visible to other processors until certain synchronization accesses occur. Shared memory accesses are categorized either as *ordinary* or as *synchronization* accesses, with the latter category further divided into *acquire* and *release* accesses. Acquires and releases roughly correspond to synchronization operations on a lock, but other synchronization mechanisms can be implemented on top of this model as well. For instance, arrival at a barrier can be modeled as a release, and departure from a barrier as an acquire. Essentially, releases

consistency requires ordinary shared memory updates by a processor p to become visible at another processor p, only when a subsequent release by p becomes visible at q. In the *lazy* implementation of release consistency in TreadMarks [9], the propagation of modifications is postponed until the time of the acquire. At this time, the acquiring processor determines which modifications it needs to see according to the definition of release consistency.

False sharing is a source of frequent communication for many DSM systems. It occurs when two or more processors access different variables within a page, with at least one of the accesses being a write. To address this problem, TreadMarks uses a *multiple-writer* protocol [28]. With multiple-writer protocols two or more processors can simultaneously modify their local copy of a shared page. Their modifications are merged at the next synchronization operation in accordance with the definition of release consistency, thereby reducing the effect of false sharing.

The TreadMarks application programming interface (API) provides facilities for process creation and destruction, synchronization, and shared memory allocation. Shared memory allocation is done through `Tmk_malloc()`. Only memory allocated by `Tmk_malloc()` is shared. Memory allocated statically or by a call to `malloc()` is private to each process. TreadMarks provides two synchronization primitives: barrier and exclusive locks. A process waits at a barrier by calling `Tmk_barrier()`. Barriers are global: the calling process is stalled until all processes in the system have arrived at the same barrier. A `Tmk_lock_acquire` call acquires a lock for the calling process, and `Tmk_lock_release` releases it. No process can acquires a lock while another process is holding it. This particular choice of synchronization primitives is not in any way fundamental to the design of TreadMarks; other primitives can be added later.

The TreadMarks system does not provide adaptive parallelism or fault tolerance.

2.5.3 PVM

PVM [7] stands for Parallel Virtual Machine. It is a software package that allows a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. PVM consists of two parts: a daemon process that any user can install on a machine, and a user library that contains routines for initiating processes on other machines, for communicating between processes, and changing the configuration of machines. The unit of parallelism in PVM is a task (often but not always a Unix process), an independent sequential thread of control that alternates between communication and computation. No process-to-processor mapping is implied or enforced by PVM; in particular, multiple tasks may execute on a single processor. PVM uses a explicit message-passing model where collections of computational tasks, each performing a part of an application's workload using data-, functional-, or hybrid decomposition, cooperate by explicitly sending and receiving messages to one another.

In general, the distributed shared memory approach is more attractive since most programmers find it easier to use than a message passing paradigm, which requires them to explicitly partition data and manage communication. With a global address space, the programmer can focus on algorithmic development rather than on managing partitioned data sets and communicating values.

PVM does not provide adaptive parallelism or fault tolerance.

2.6 Parallel Search Algorithms

Search is a major computational paradigm in Artificial Intelligence. There has been a fair amount of work which has looked at pruning, load-balancing, locality, granularity, and anomalous speedup issues in parallel search applications.

A common search problem concerns exploring a large state-space for a goal state. The state space is usually structured as a tree, with operators that can transform one state "node" to another forming arcs between different states. In a large class of such problems, the computations tend to be unpredictably structured and have multiple solutions. Saletore and Kalé [17] considered the problem of parallel state-space search for a first solution.

In their scheme, every node in the state-space is associated with a *priority bit vector*, which is a sequence of bits of any arbitrary length. Priorities are assigned in a way such that (a) The relative priority of the sibling nodes preserves the left to right order of the sibling nodes; and (b) every descendent of a higher priority node gets higher priority than all the descendents of low priority nodes. This reflects the policy that until there is no prospect of a solution from the left subtree beneath a node, the system should not spend its resources on the right subtrees, unless there are idle processors. That is, if for a time period, if the work available in the left subtree is not sufficient to keep all the processors busy, the idle processors may expand the right subtrees. But as soon as high priority nodes become available in the left subtree, the processors must focus their efforts in that left subtree. Thus, the parallel search behaviour is similar to that of sequential depth-first search. As a result, their algorithm yields consistent linear speedups over sequential depth-first search on a multi-processor system (Sequent Symmetry).

Kumar, Ramesh, and Rao presented many different parallel formulations of depth-first and best-first search algorithms as well [14, 15, 16]. These are all one-solution algorithms. In other words, the goal of the search is one (optimal) solution on the state-space. However, data mining problems require finding all solutions. Furthermore, most of these parallel search techniques require frequent communication among processors, which is not suitable for NOW.

2.7 Domain-Specific Parallelism Frameworks

2.7.1 Multipol

Multipol [11] is a library of distributed data structures designed for irregular applications such as discrete event simulation, symbolic computation, and search problems. Multipol has data structures for both bulk-synchronous applications with irregular communication patterns and asynchronous applications. The Multipol runtime system contains a thread system and a simple producer-consumer synchronization construct for expressing dependencies between threads. The threads also create opportunities for overlapping communication latency with computation, and for aggregating multiple remote operations in large physical messages to reduce communication overhead.

The design and implementation of Multipol target distributed memory architectures, including the Thinking Machines CM-5, Intel Paragon, and IBM SP1. It is also being ported to networks of workstations.

2.7.2 LPARX

LPARX [12] is a parallel programming model for dynamic, non-uniform scientific computations. It provides run-time support for these computations with a data decomposition model for block irregular data structures. The LPARX system was motivated by a number of design considerations. First, LPARX does not rely on architecture-specific facilities, such as fine-grain message passing, and therefore it is portable across a variety of MIMD architectures. Second, LPARX assumes no compiler support and all decisions about data decomposition, the assignment of data to processors, and the communication of data dependencies are made at run-time. Finally, since modern scientific codes often use elaborate data structures which require special treatment when communicated across

address spaces, LPARX provides support for arrays of complicated objects in addition to standard built-in types.

LPARX, implemented as a C++ class library, requires basic message passing support. It is currently running on the CM-5, Paragon, iPSC/860, nCUBE/2, KSR-1, single processor workstations, Cray-90 (single processor), and networks of workstations under PVM.

Chapter 3

E-Dag: Framework for Finding Lattices of Patterns

3.1 Exploration Dag

3.1.1 Modeling Data Mining Applications

In sections 2.2, 2.1, and 2.3, we surveyed three major classes of data mining applications, namely association rule mining, classification rule mining, and pattern discovery in combinatorial databases. In this section, we note the resemblance among the computation models of these three application classes. Table 3.1 is a comparison of specifications of these three classes of applications.

A *task* is the main computation applied on a pattern. Not only are all tasks of any one application of the same kind, but tasks of different applications are actually very similar. They all take a pattern and a subset of the database and count the number of records in the subset that match the pattern. In the classification rule mining case, counts of matched records are divided into c baskets, where c is the number of distinct

	Pattern Discovery	Assoc. Rule Mining	Class. Rule Mining
database	sequences	transaction records	database relation
pattern	partial sequence	itemset	attribute–value condition
good pattern	$occurrence_{pattern}$ $> min_occurrence$	$support_{pattern}$ $> min_support$	$info_gain_{attribute} =$ $max_{sibling_attributes}$ $(info_gain_{attribute})$
task	counting *occurrence* of *pattern* in subset of database	counting *support* of *itemset* over subset of database	building histogram of *pattern* on *classes* over subset of database

Table 3.1: A comparison of specifications of three classes of data mining applications.

classes.

The similarities among the specifications of these applications are obvious, which inspired us to study the similarities among their computation models. As we can see from previous sections, they usually follow a generate-and-test paradigm—generate a candidate pattern, then test whether it is any good. Furthermore, there is some interdependence among the patterns that gives rise to pruning, i.e., if a pattern occurs too rarely, then so will any superpattern. These interdependences entail a lattice of patterns, which can be used to guide the computation.

In fact, this notion of pattern lattice can apply to any data mining application that follows this generate-and-test paradigm. We call this application class *pattern lattice data mining.* In order to characterize the computation models of these applications more concretely, we define them more carefully in Section 3.1.2.

3.1.2 Defining Data Mining Applications

In general, a data mining application defines the following elements.

1. A database $\mathcal{D}$.

2. Patterns and a function len(pattern p) which returns the length of p. The length of a pattern is a non-negative integer.

   Example 3.1.1 Patterns of length k in the three application class are shown below:

sequence pattern discovery	$*C_1C_2 \ldots C_k*$ $C_1, C_2, \ldots, C_k$ are letters
association rule mining	$\{i_1, i_2, \ldots, i_k\}$ $i_1, i_2, \ldots, i_k$ are items
classification rule mining	$(A_1 = v_{1i_1}) \wedge (A_2 = v_{2i_2}) \wedge \ldots \wedge (A_k = v_{ki_k})$ $A_1, A_2, \ldots, A_k$ are attributes, and $v_{1i_1}, v_{2i_2}, \ldots, v_{ki_k}$ are attribute values

   We use $**$, $\{\}$, and $\emptyset$ to represent zero-length patterns in sequence pattern discovery, association rule mining, and classification rule mining, respectively.

3. A function $goodness$(pattern p) which returns a measure of p according to the specifications of the application.

   Example 3.1.2 The goodness of a pattern p in sequence pattern discovery is the occurrence number of p in the set of sequences; the goodness of a pattern p in association rule mining is the support of p over the set of items; the goodness of a pattern p in classification rule mining is the $info_gain$ by partitioning the training set by p.

4. A function $good(p)$ which returns 1 if p is a good pattern or a good subpattern and 0 otherwise. Zero-length patterns are always good.

   Example 3.1.3 In sequence pattern discovery, a pattern p is good if $goodness(p)$ is greater than or equal to some prespecified $min_occurrence$; in association rule mining, pattern p is good if $goodness(p) \geq$ some prespecified $min_support$; in classification rule mining, a pattern p is good if $goodness(p) \geq goodness(p')$, for any p' such that $len(p') = len(p)$.

   Sometimes there are such additional requirements as minimum and/or maximum lengths of good patterns (e.g., in sequence pattern discovery). Without loss of generality, we disregard these requirements in our discussion unless otherwise noted.

The result of a data mining application is the set of all good patterns. If a pattern is not good, neither will any of its superpatterns be. In other words, it is necessary to consider a pattern if and only if all of its subpatterns are good.

Let us define an *immediate subpattern* of a pattern q to be a subpattern p of q where $len(p) = len(q) - 1$. Conversely, q is called an *immediate superpattern* of p.

Example 3.1.4 Immediate subpatterns of a length k pattern p:

sequence pattern discovery	all $(k-1)$-prefixes and all $(k-1)$-suffixes of p
association rule mining	all $(k-1)$-itemsets
classification rule mining	the pattern consisting of the first $k-1$ attribute-value pairs in p

Except for the zero-length pattern, all the patterns in a data mining problem are generated from their immediate subpatterns. In order for all the patterns to be uniquely generated, a pattern q and one of its immediate subpatterns p have to establish a child-parent relationship (i.e., q is a *child pattern* of p and p is the *parent pattern* of q). Except for the zero-length pattern, each pattern must have one and only one parent pattern. For example, in sequence pattern discovery, $*$FRR$*$ can be a child pattern of $*$FR$*$; in association rule mining, $\{2, 3, 4\}$ can be a child pattern of $\{2, 3\}$; and in classification rule mining, $(C = c_1) \wedge (B = b_2) \wedge (A = a_1)$ can be a child pattern of $(C = c_1) \wedge (B = b_2)$.

3.1.3 Solving Data Mining Applications

Having defined data mining applications as above, it is easy to see that an optimal sequential program that solves a data mining application does the following:

1. generates all child patterns of the zero-length pattern;
2. computes $goodness(p)$ if all of p's immediate subpatterns are good;
3. if $good(p)$ then generate all child patterns of p;
4. applies 2 and 3 repeatedly until there are no more patterns to be considered.

Because the zero-length pattern is always good and the only immediate subpatterns of its children is the zero-length pattern itself, the computation starts on all its children, which are all length 1 patterns. After these patterns are computed, good patterns generate their child sets. Not all of these new patterns will be computed—only those whose every immediate subpattern is good will be.

3.1.4 Exploration Dag

We propose to use a directed acyclic graph (dag) structure called *exploration dag* (*E-dag*, for short) to characterize pattern lattice data mining applications. We first describe how to map a data mining application to an E-dag.

The E-dag constructed for a data mining application has as many vertices as the number of all possible patterns (including the zero-length pattern). Each vertex is labeled with a pattern and no two vertices are labeled with the same pattern. Hence there is a one-to-one relation between the set of vertices of the E-dag and the set of all possible patterns. Therefore, we refer to a vertex and the pattern it is labeled with interchangeably.

There is an incident edge on a pattern p from each immediate subpattern of p. All patterns except the zero-length pattern have at least one incident edge on them. The zero-length pattern has an outgoing edge to each pattern of length 1.

Figure 3.1 (Figure 3.2, Figure 3.3, respectively) shows an E-dag mapped from a simple sequence pattern discovery (association rule mining, classification rule mining) application.

3.1.5 A Data Mining Virtual Machine

The input of the proposed *data mining virtual machine* (DMVM) is a data mining application with all the elements defined in Section 3.1.2. The output of the DMVM is the result of the data mining application. In the remainder of this section, we show that the E-dag structure enables us to build an efficient data mining virtual machine.

Fact 1 *All E-DAGs built from the same input are isomorphic.*

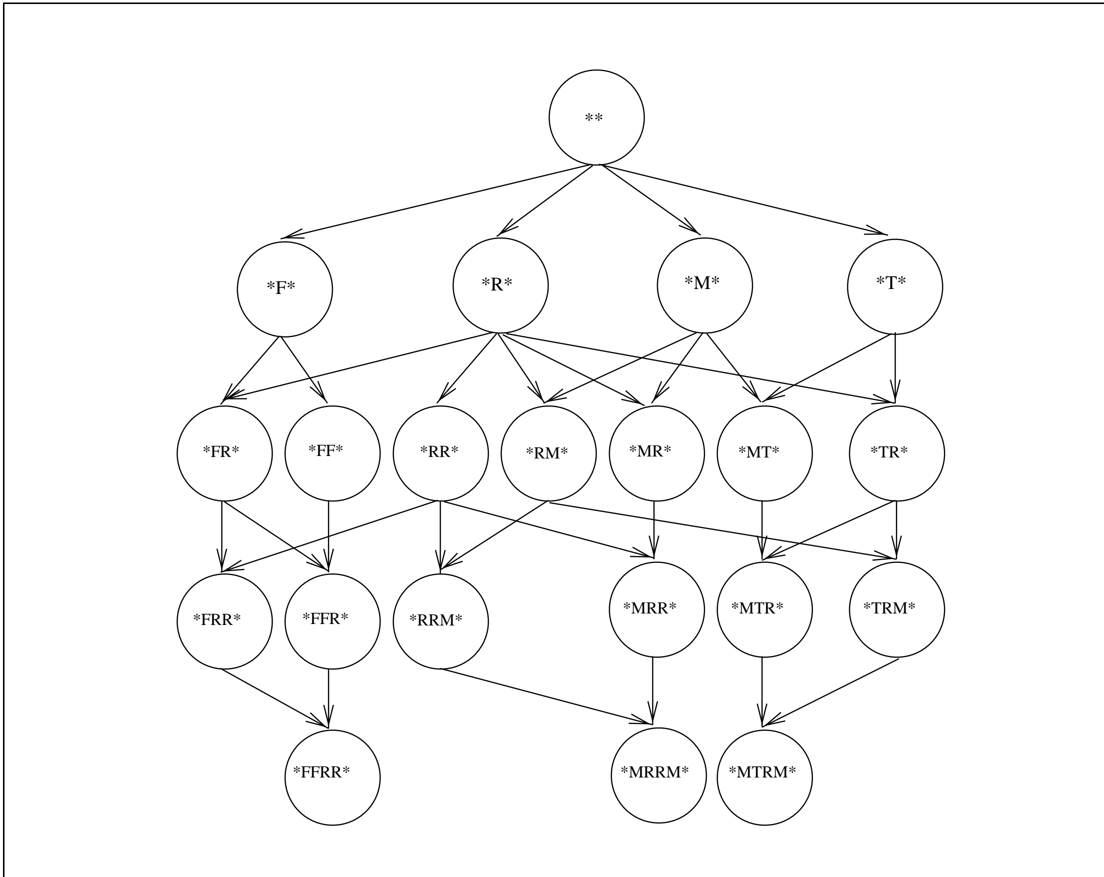


Figure 3.1: A complete E-DAG for a sequence pattern discovery application on sequences FFRR, MRRM, and MTRM.

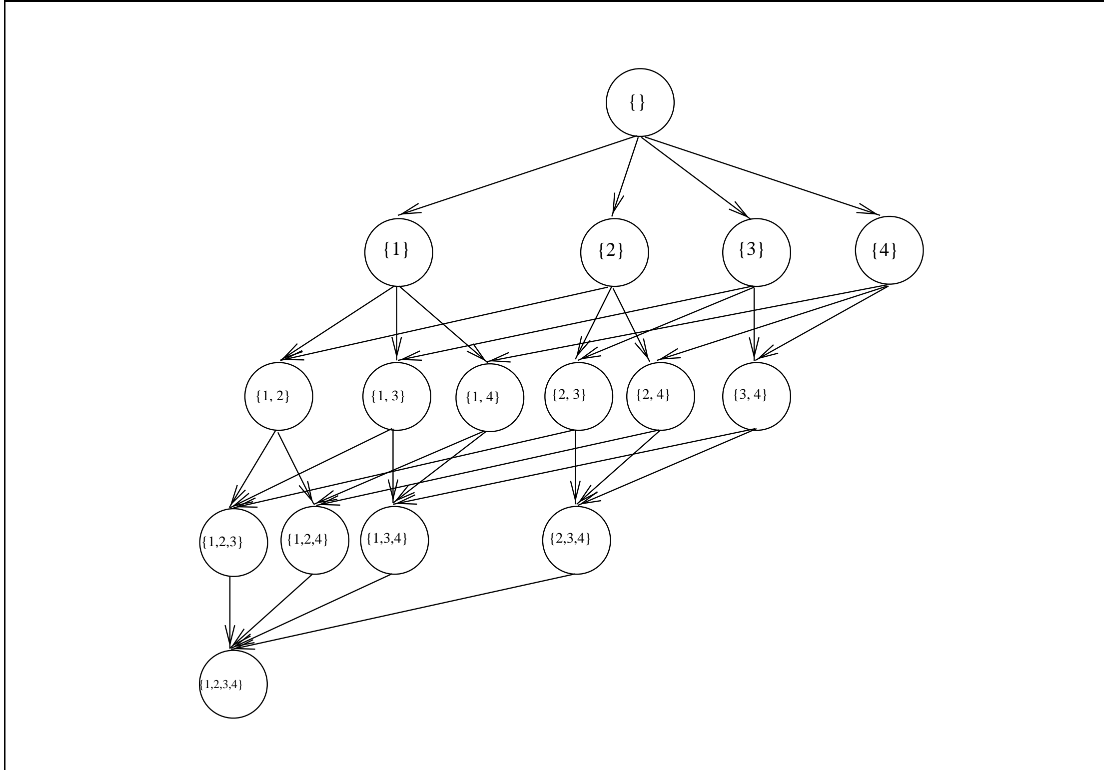


Figure 3.2: A complete E-DAG for an association rule mining application on the set of items {1, 2, 3, 4}.

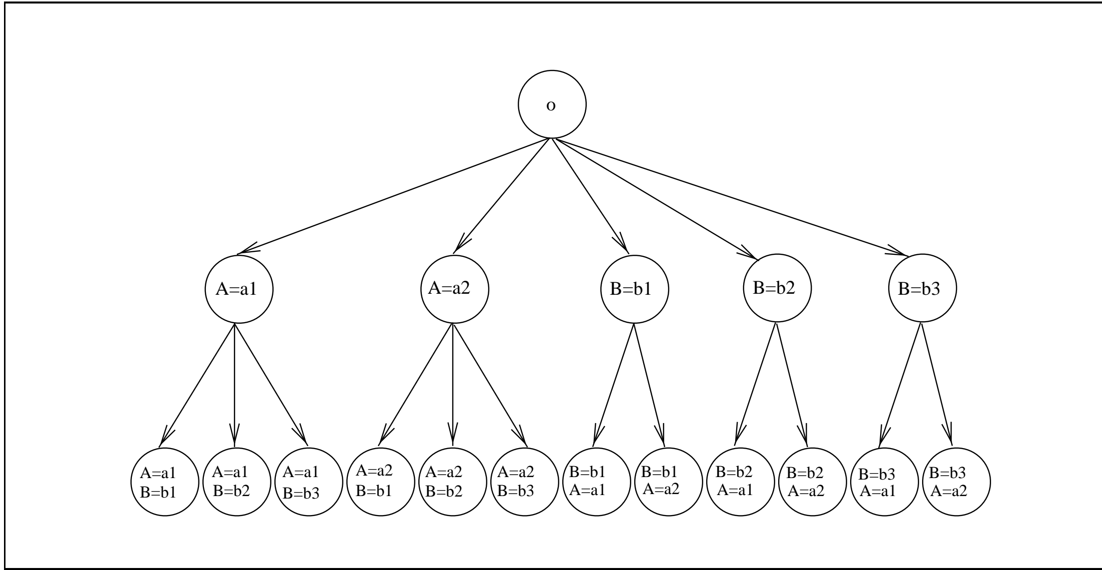


Figure 3.3: A complete E-DAG for a classification rule mining application on a simple database with attributes A (possible values $a1$ and $a2$), and B (possible values $b1$, $b2$, and $b3$).

This guarantees that the following discussion applies on all E-dags built for a single application.

Definition 1 *In an* E-dag traversal (EDT)*, a vertex n is visited only if all vertices that have an incident edge on n have been visited.*

Fact 2 *There is a plenty of pruning in an E-dag traversal.*

If a pattern p is not good, neither is any of its superpatterns. Therefore, there is no need to build the complete E-dag before we perform an E-dag traversal. Instead, an E-dag is lazily constructed—vertices are generated only when it is necessary to look at the patterns on them—while an E-dag traversal is performed on the E-dag on the fly.

For an E-dag $\mathcal{E}$ built from a data mining application $\mathcal{A}$ and a sequential program $\mathcal{P}$ that solves $\mathcal{A}$, an E-dag traversal of $\mathcal{E}$ is said to be *equivalent* to an execution of $\mathcal{P}$

if and only if (1) the results (good patterns) from the E-dag traversal are the same as those from the execution of $\mathcal{P}$, and (2) the same potential patterns are tested (in other words, all patterns that the function $goodness(p)$ would have been applied on are the same in the E-dag traversal as in the execution of $\mathcal{P}$).

Theorem 1 *For a (i) data mining application $\mathcal{A}$, (ii) an E-dag $\mathcal{E}$ built from $\mathcal{A}$, and (iii) a sequential program $\mathcal{P}$ that solves $\mathcal{A}$ and that is optimal, an E-dag traversal of $\mathcal{E}$ is equivalent to an execution of $\mathcal{P}$ on input $\mathcal{A}$.*

Because $\mathcal{P}$ is optimal, an execution of $\mathcal{P}$ will test the least number of potential patterns. Until it has tested *all* length k patterns, it will not test *any* pattern whose length is greater than k, because the goodness result of a length k pattern may make testing (some) patterns of greater length unnecessary. Therefore, an execution of $\mathcal{P}$ will test all length 1 patterns first, and then test all length 2 patterns, and so on, until there are no more patterns to be considered.

According to the definition of E-dag traversal, all patterns of length 1 are considered before any pattern of length 2 are, as is true in an execution of $\mathcal{P}$. Because the same result would be obtained on each length 1 pattern, exactly the same length 2 patterns will be considered in an E-dag traversal as in an execution of $\mathcal{P}$. Similarly, exactly the same length 3 patterns will be tested. This goes on until all length k patterns are not good (in other words, there are no more patterns to be considered.) Because an E-dag traversal and an execution of $\mathcal{P}$ test exactly the same length 1 patterns, exactly the same length 2 patterns, $\ldots$, exactly the same length k pattern all the patterns they test are the same. And because each test yields the same result, the same good patterns are found. Therefore, an E-dag traversal of $\mathcal{E}$ is equivalent to an execution of $\mathcal{P}$. In fact, for every two patterns of different length, the goodness function would have applied in

the same order in an E-dag traversal of $\mathcal{E}$ as in an execution of $\mathcal{P}$. □

At the end of the computation, the resultant E-dag gives the results of the data mining application. For a sequence pattern discovery application, motifs on the vertices of the resultant E-dag are active motifs. For an association rule mining application, all itemsets on the vertices of the resultant E-dag are frequent itemsets (these are answers to phase I in association rule mining, but phase II is rather straightforward and is much less time-consuming). For a classification rule mining application, a simple transition from the resultant E-dag gives the decision tree. For an attribute–value pair $A = a$ on a node n, move A up into n's parent p (if there is not an attribute name in p already), and make a the label along the path from p to n. Having done this to all the nodes except the root of the tree, the root and each of the internal nodes will have an attribute name. Finally label the leaves with the names of the appropriate classes they belong to.

3.2 A Parallel Data Mining Virtual Machine

3.2.1 A Parallel Data Mining Virtual Machine

We can now define a *parallel data mining virtual machine* (PDMVM). The input of the PDMVM is a data mining application with all the elements defined in Section 3.1.2. The output of the PDMVM is the result of the data mining application. The input and the output of the PDMVM are exactly the same as those of the DMVM presented in Section 3.1.5.

Definition 2 *A* parallel E-dag traversal (PEDT) *is an E-dag traversal done in parallel.*

Imagine that we have an unlimited number of *visiting workers* each of which can individually visit any vertex of an E-dag. If, at the beginning of the EDT, there are n_1

patterns of length 1, n_1 visiting workers can visit these patterns in parallel. After all length 1 patterns have been visited, there are, say, n_2 patterns of length 2 to be visited; then n_2 visiting workers can visit these patterns in parallel; so on and so forth.

A PEDT is done in a manner that no two patterns of different length on which $goodness(p)$ would have applied in a different order in an EDT of $\mathcal{E}$ than in a PEDT of $\mathcal{E}$.

Lemma 1 *The result of a PEDT is the same as the result of an EDT.*

This follows immediately from the definition of PEDT. □

Theorem 2 *For (i) a data mining application $\mathcal{A}$, (ii) an E-dag $\mathcal{E}$ built from $\mathcal{A}$, and (iii) a sequential program $\mathcal{P}$ that solves $\mathcal{A}$ and that is optimal, a PEDT of $\mathcal{E}$ is equivalent to an execution of $\mathcal{P}$ on input $\mathcal{A}$.*

The definition of PEDT and Theorem 1 guarantee that a PEDT and an execution of $\mathcal{P}$ test exactly the same potential patterns. And Lemma 1 guarantees that the result of a PEDT is the same as the output of an execution of $\mathcal{P}$. □

3.2.2 PLinda Implementation of PDMVM

In this section, we describe an implementation of the parallel data mining virtual machine on the Persistent Linda system.

A parallel E-dag traversal PLinda program (PLED) consists of a PLED master and a PLED worker. A PLED worker plays the role of a visiting worker described above. Figure 3.4 (figure 3.5, respectively) is the pseudo-PLinda code of a PLED master (worker).

The function $child_pattern(node)$ generates all child patterns of the pattern on

```
child_pattern(zero-length pattern);
while !done do {
    xstart;
    in(?node, ?score);
    if good(node) then {
        child_pattern(node);
    } else if (task_sent == task_done) then {
        done=1;
    }
    xcommit;
}
out(poison tasks);
```

Figure 3.4: Pseudo-PLinda code of a PLED master.

```
while !done {
    xstart;
    in("task", ?node);
    if !(poison task) {
        out(node, goodness(node));
    } else {
        done=1;
    }
    xcommit;
}
```

Figure 3.5: Pseudo-PLinda code of a PLED worker.

the input *node*, but it outs only those children whose immediate subpatterns are all good into tuple space. As a result, a PLED worker can take only patterns that are known to be necessary to be considered. Work tuples are of the form `("task", node)`. Goodness of a node is outed by a worker in a result tuple of the form `(node, score)`. The function *good*(*node*) will determine if *node* is a good pattern according to goodness of all of its immediate subpatterns.

3.3 Optimal Implementation of PDMVM

Our PLinda implementation of the parallel data mining virtual machine generates the correct result for any data mining application because it faithfully implements a parallel E-dag traversal. But what would be an optimal implementation of a parallel E-dag traversal on networks of workstations? Would an optimal implementation have an equivalent execution as the optimal sequential program on any input? In the distributed shared memory model, with the presence of complicated communication and synchronization costs, the first question is unanswerable in general. And the answer to the second question is *no*.

So what is our approach to optimality? In order to answer this question, we first introduce a tree structure that is closely related to the E-dag structure. The tree structure is called *exploration tree* (*E-tree*, for short).

3.3.1 Exploration Tree

An E-tree is a tree transformed from an E-dag. The transformation is very simple. If we make edges from all patterns to their non-child immediate superpatterns dashed, all the vertices in the E-dag and all the solid edges constitute the E-tree. The root of

the E-tree is the zero-length pattern. Figure 3.6, Figure 3.7, and Figure 3.8 are E-trees transformed from the corresponding E-dags in Section 3.1.4.

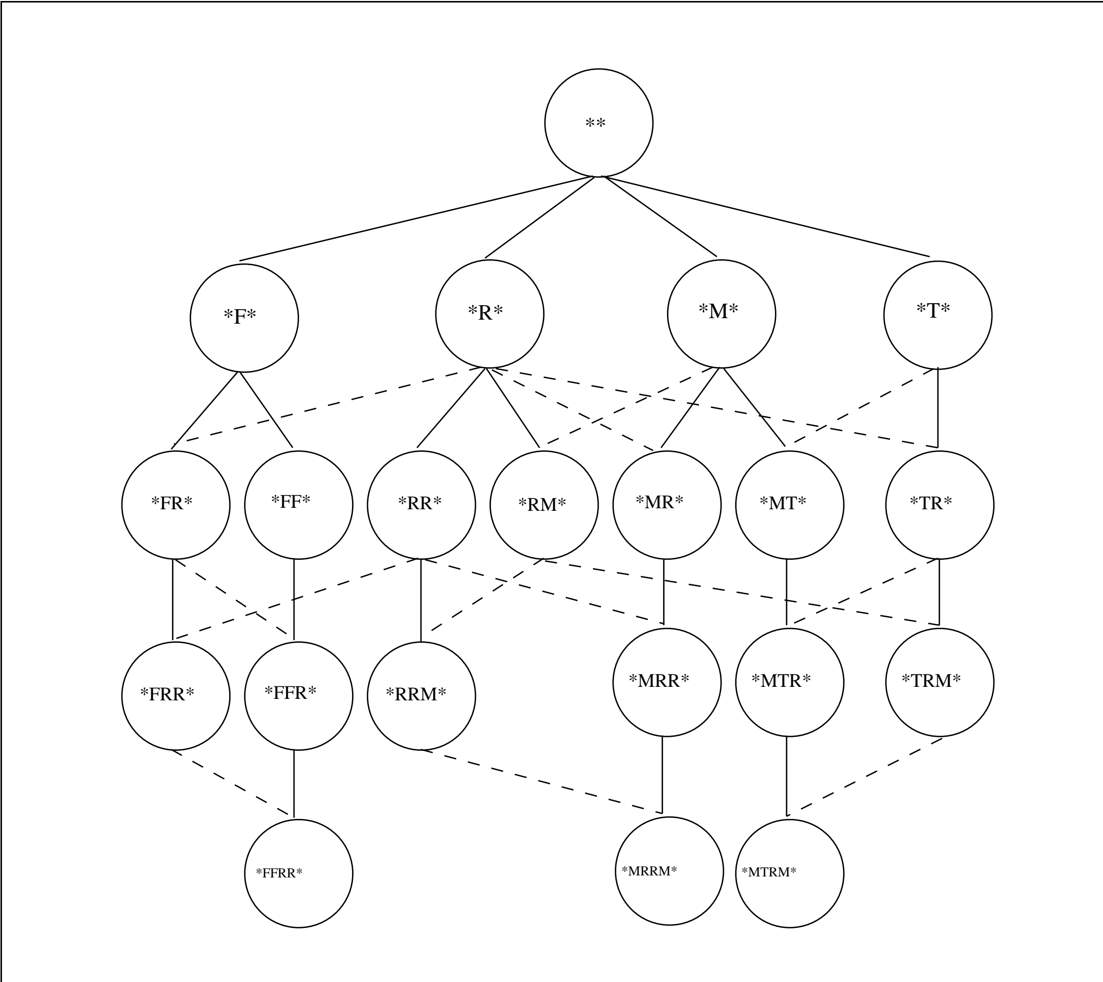


Figure 3.6: An E-tree for a sequence pattern discovery application on sequences FFRR, MRRM, and MTRM.

3.3.2 E-tree Traversal

Definition 3 *In an* E-tree traversal (ETT), *a node of a tree is visited only if its parent has been visited and is good.*

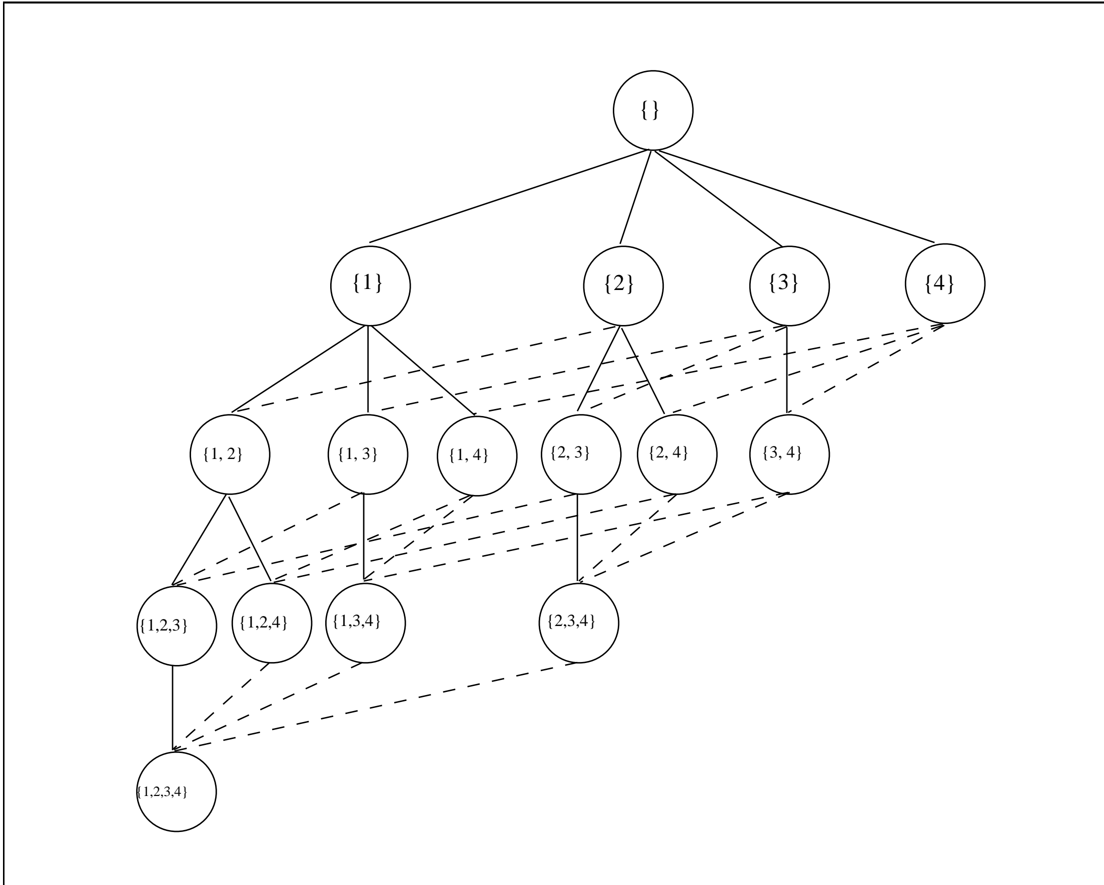


Figure 3.7: An E-tree for an association rule mining application on the set of items {1, 2, 3, 4}.

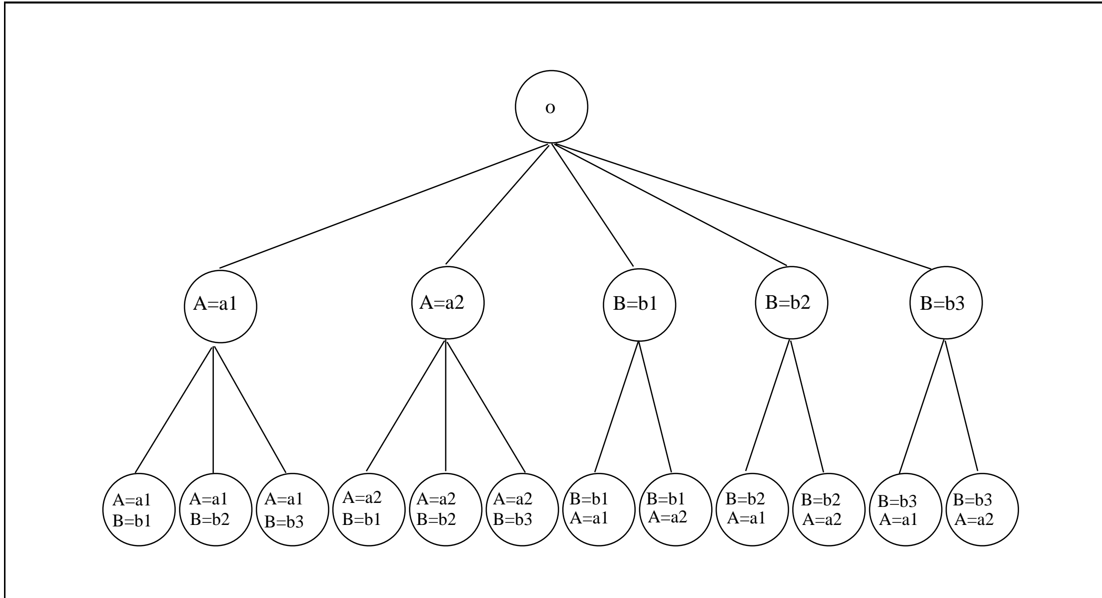


Figure 3.8: An E-tree for a classification rule mining application on a simple database with attributes A (possible values $a1$ and $a2$), and B (possible values $b1$, $b2$, and $b3$).

An E-tree traversal is different from an E-dag traversal in that a pattern can be considered even if not all of its subpatterns are good. Because of this, there may be wasted work done on some nodes. For the same reason, however, it does not require a synchronization at each level of the tree, which gives rise to good load balancing when it is done in parallel.

Opportunities for pruning are still apparent in an E-tree traversal. Once a pattern p is found not good, the whole subtree rooted at p is pruned. Therefore, we should still take advantages of lazy construction of E-trees when we perform E-tree traversals.

Lemma 2 *The result of an ETT is the same as the result of an EDT.*

It is easy to see that all good patterns produced in an EDT are also produced in an ETT since all patterns ever considered in an EDT are also considered in an ETT. If

any subpattern of p is not good, neither is p. So if a pattern is considered even if not all of its subpatterns are good, it will eventually be pronounced not good and hence does not produce more patterns to be considered. Therefore, no more good patterns will be produced in an ETT than in an EDT. Thus Lemma 2 holds. □

Definition 4 *A* parallel ETT (PETT) *is an ETT done in parallel.*

If the root of the E-tree has n_1 children, then n_1 visiting workers can traverse them in parallel. Once any of these pattern is found good, then all its children can be visited by the same number of visiting workers in parallel. This applies recursively until the end of computation.

Because each branch of any subtree can proceed independently, it is certainly possible that some visiting workers are working at a lower level in the E-tree than some other visiting workers. Furthermore, it is possible that there are two patterns of different length on which $goodness(p)$ would have been applied in a different order in an EDT of $\mathcal{E}$ than in a PETT of $\mathcal{E}$.

Lemma 3 *The result of a PETT is the same as the result of an ETT.*

All the good patterns produced by an ETT are produced by a PETT. Similar to the proof of Lemma 2, no more good patterns are produced by a PETT than by an ETT. Therefore, the above lemma holds. □

Theorem 3 *For a data mining application $\mathcal{A}$, an E-dag $\mathcal{E}$ built from $\mathcal{A}$, and a sequential program $\mathcal{P}$ that solves $\mathcal{A}$ and that is optimal, the result of a PETT of $\mathcal{E}$ is the same as the output of $\mathcal{P}$ on input $\mathcal{A}$.*

This follows from Theorem 1, Lemma 2, and Lemma 3. □

3.3.3 PLinda Implementation of PETT

A parallel E-tree traversal PLinda program (PLET) also consists of two parts: a PLET master and a PLET worker. Figure 3.9 (figure 3.10, respectively) is the pseudo-PLinda code of a PLET master (worker).

The function *termination*(*node*) detects if the computation has completed. It works as follows.

1. Mark *node* as pruned (in this case no descendants of *node* will be visited);
2. Check if all siblings of *node* are pruned; if so, mark the parent of *node* as pruned;
3. If the root is pruned, the computation has completed.

3.3.4 Optimal PLinda Implementation of PDMVM

The optimal PLinda implementation of the parallel data mining virtual machine is a combination of PLED and PLET.

We observe that one can benefit more from the pruning due to a non-good subpattern in early stages of an E-dag traversal than in later stages. One possibility is that the optimal PLinda program will start with PLED, in the middle of which, will switch to PLET. When to switch is the crucial question in this scheme. The answer will depend on the particular environment on which the program is running and the data mining application itself.

Another possibility for the optimal program is a hybrid from PLED and PLET, in which, when a visiting worker becomes free, it visits the non-pruned node as high as possible in the E-tree. If the result is not good, then it prunes all its superpatterns. This algorithm would be optimal under the assumptions of free network bandwidth and free access to shared storage.

```
child_pattern(root);
while !done {
    xstart;
    in("pruned", ?node);
    if termination(node) then {
        out(poison tasks);
        done=1;
    }
    xcommit;
}
```

Figure 3.9: Pseudo-PLinda code of a PLET master.

```
while !done {
    xstart;
    in("task", ?node);
    if !(poison task) then {
        compute goodness(node);
        if good(node) then {
            child_pattern(node);
        } else {
            out("pruned", node);
        }
    } else {
        done=1;
    }
    xcommit;
}
```

Figure 3.10: Pseudo-PLinda code of a PLET worker.

Theorem 4 *For (i) a data mining application* $\mathcal{A}$*, (ii) an E-dag* $\mathcal{E}$ *built from* $\mathcal{A}$*, and (iii) a sequential program* $\mathcal{P}$ *that solves* $\mathcal{A}$ *and that is optimal, the result of an optimal PLinda E-dag traversal of* $\mathcal{E}$ *is the same as the output of an execution of* $\mathcal{P}$ *on input* $\mathcal{A}$*.*

Since the optimal PLinda program is a combination of PLED and PLET, this theorem follows from Theorem 2 and Theorem 3. □

3.4 Summary

We proposed the E-dag framework for finding pattern lattices based on analysis of computation models of three classes of data mining problems. We described how an E-dag traversal would solve the data mining problem the E-dag represents and proved that an E-dag traversal is equivalent to any optimal sequential program that solves the same problem. E-dag construction and traversal can be done efficiently in parallel. However, network latency can slow down parallel E-dag traversal if there is a fair amount of inter-process communication. In an E-tree traversal, much communication is eliminated by giving up some pruning opportunities. We observe that an optimal parallel program is some form of combination of E-dag and E-tree traversal.

Chapter 4

Biological Pattern Discovery

4.1 Biological Pattern Discovery

Biological pattern discovery problems are computationally expensive. A possible technique for reducing the time to perform pattern discovery is parallelization. Since each task in a biological pattern discovery application is usually time consuming by itself, we might be able to use networks of workstations (NOW).

Finding active motifs in sets of protein sequences and in multiple RNA secondary structures are two examples of biological pattern discovery. We will demonstrate that, using the E-dag framework, it is easy to parallelize these applications and it is efficient to run the parallel programs in PLinda on NOW.

4.1.1 Discovery of Motifs in Protein Sequences

Biologists represent proteins as sequences made from 20 amino acids, each represented as a letter. Figure 4.1 is the sequence representation of a real protein named "CG2A_DAUCA G2/MITOTIC-SPECIFIC CYCLIN C13-1 (A-LIKE CYCLIN)". If two sequences are

```
APSMTTPEPASKRRVVLGEISNNSSAVSGNEDLLCREFEVPKCVAQKKRKRGVKEDVGVD
FGEKFDDPQMCSAYVSDVYEYLKQMEMETKRRPMMNYIEQVQKDVTSNMRGVLVDWLVEV
SLEYKLLPETLYLAISYVDRYLSVNVLNRQKLQLLGVSSFLIASKYEEIKPKNVADFVDI
TDNTYSQQEVVKMEADLLKTLKFEMGSPTVKTFLGFIRAVQENPDVPKLKFEFLANYLAE
LSLLDYGCLEFVPSLIAASVTFLARFTIRPNVNPWSIALQKCSGYKSKDLKECVLLLHDL
QMGRRGGSLSAVRDKYKKHKFKCVSTLSPAPEIPESIFNDV
```

Figure 4.1: Sequence representation of a real protein named "CG2A_DAUCA G2/MITOTIC-SPECIFIC CYCLIN C13-1 (A-LIKE CYCLIN)".

almost the same or share very similar subsequences, it may be that the common part may perform similar functions via related biochemical mechanisms [40, 45, 51, 56]. Thus, finding a frequently occurring subsequences in a set of protein sequences is an important problem in computational biology.

Consider a database of imaginary protein sequences $\mathcal{D}$={FFRR, MRRM, MTRM, DPKY, AVLG} and the query "Find the patterns P of the form $*X*$ where P occurs in at least 2 sequences in $\mathcal{D}$ and the size of P $|P| \geq 2$." (X can be a segment of a sequence of any length, and $*$ represents a variable length don't care.) The good patterns are $*$RR$*$ (which occurs in FFRR and MRRM) and $*$RM$*$ (which occurs in MRRM and MTRM).

Pattern discovery in sets of sequences concerns finding commonly occurring subsequences (sometimes called *motifs*). The structures of the motifs we wish to discover are regular expressions of the form $*S_1 * S_2 * \ldots$ where $S_1, S_2, \ldots$ are *segments* of a sequence, i.e., subsequences made up of consecutive letters and $*$ represents a variable length don't care (VLDC). In matching the expression $*S_1 * S_2 * \ldots$ with a sequence S, the VLDCs may substitute for zero or more letters in S. Segments may allow a specified number of mutations; a mutation is a insertion, a deletion, or a mismatch.

We use terminology proposed in [Wang *et al.*, 1994]. Let $\mathcal{S}$ be a set of sequences.

The occurrence number of a motif is the number of sequences in $\mathcal{S}$ that match the motif within the allowed number of mutations. We say the occurrence number of a motif P with respect to mutation i and set $\mathcal{S}$, denoted $occurrence_no^i_{\mathcal{S}}(P)$, is k if $*P*$ matches k sequences in $\mathcal{S}$ within at most i mutations, i.e., the k sequences contain P within i mutations. Given a set $\mathcal{S}$, we wish to find all the active motifs P where P is within the allowed Mut mutations of at least $Occur$ sequences in $\mathcal{S}$ and $|P| \geq Length$, where $|P|$ represents the number of the non-VLDC letters in the motif P. (Mut, $Occur$, $Length$ and the form of P are user-specified parameters.)

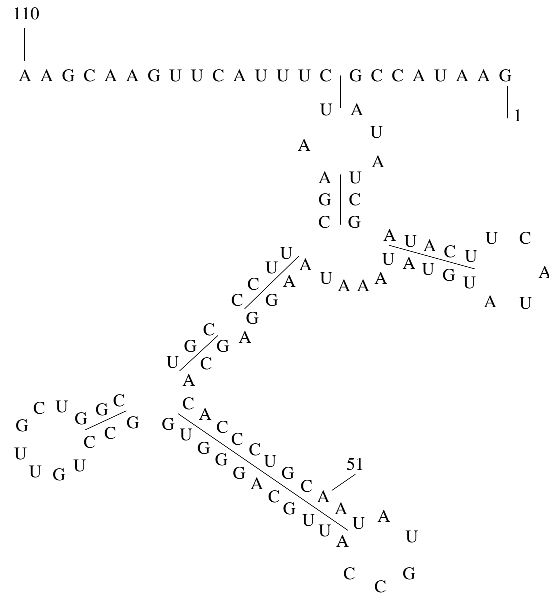
4.1.2 Discovery of Motifs in RNA Secondary Structures

Finding approximately common motifs (or *active motifs*) in multiple RNA secondary structures helps to predict secondary structures for a given mRNA [Zuker, 1989; Le *et al.*, 1989] and to conduct phylogenetic study of the structure for a class of sequences [Shapiro and Zhang 1990]. Adopting the RNA secondary structure representation previously proposed in [Shapiro and Zhang, 1990], we represent both helical stems and loops as nodes in a tree. Figure 4.2 illustrates a RNA secondary structure and its tree representation. The structure is decomposed into five terms: stem, hairpin, bulge, internal loop and multi-branch loop. In the tree, H represents hairpin nodes, I represents internal loops, B represents bulge loops, M represents multi-branch loops, R represents helical stem regions (shown as connecting arcs) and N is a special node used to make sure the tree is connected. The tree is considered to be an ordered one where the ordering is imposed based upon the 5' to 3' nature of the molecule. This representation allows one to encode detailed information of RNA by associating each node with a property list. Common properties may include sizes of loop components, sequence information and energy.

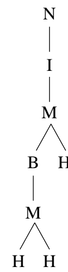
We consider a motif in a tree T to be a connected subgraph of T, viz., a subtree U of T with certain nodes being cut at no cost. (Cutting at a node n in U means removing n and all its descendants, i.e., removing the subtree rooted at n.) The dissimilarity measure used in comparing two trees is the *edit distance*, i.e., the minimum weighted number of insertions, deletions and substitutions (also known as relabelings) of nodes used to transform one tree to the other [Shapiro and Zhang, 1990; Wang *et al.*, 1994]. Deleting a node n makes the children of n the children of the current parent of n. Inserting n below a node p makes some consecutive subsequence of the children of p become the children of n. For the purpose of this work, we assume that all the edit operations have unit cost.

Consider the set $\mathcal{S}$ of three trees in Figure 4.3(a). Suppose only exactly coinciding connected subgraphs occurring in all the three trees and having size greater than 2 are considered as active motifs. Then $\mathcal{S}$ contains two active motifs shown in Figure 4.3(b). If connected subgraphs having size greater than 4 and occurring in all the three trees within distance one are considered as active motifs, i.e., one substitution, insertion or deletion of a node is allowed in matching a motif with a tree, then $\mathcal{S}$ contains two active motifs shown in Figure 4.3(c).

We say a tree T contains a motif M within distance d (or M approximately occurs in T within distance d) if there exists a subtree U of T such that the *minimum* distance between M and U is less than or equal to d, allowing zero or more cuttings at nodes from U. Let $\mathcal{S}$ be a set of trees. The occurrence number of a motif M is the number of trees in $\mathcal{S}$ that contain M within the allowed distance. Formally, the occurrence number of a motif M with respect to distance d and set $\mathcal{S}$, denoted $occurrence_no^d_{\mathcal{S}}(M)$, is k if there are k trees in $\mathcal{S}$ that contain M within distance d. Given a set $\mathcal{S}$ of trees, we wish to find all the motifs M where M is within the allowed distance *Dist* of at least *Occur*



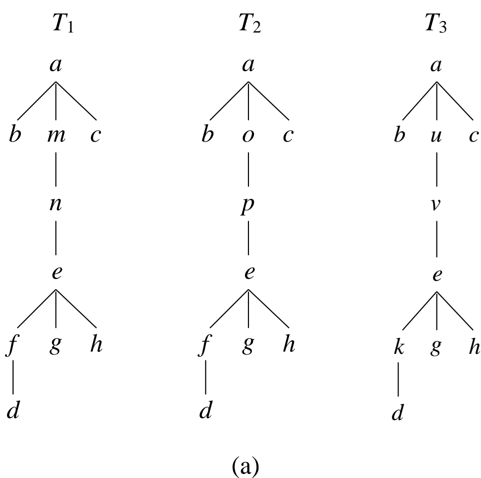
(a)



(b)

Figure 4.2: Illustration of a typical RNA secondary structure and its tree representation. (a) Normal polygonal representation of the structure. (b) Tree representation of the structure.

trees in $\mathcal{S}$ and $|M| \geq Size$, where $|M|$ represents the size, i.e., the number of nodes, of the motif M. ($Dist$, $Occur$ and $Size$ are user-specified parameters.)



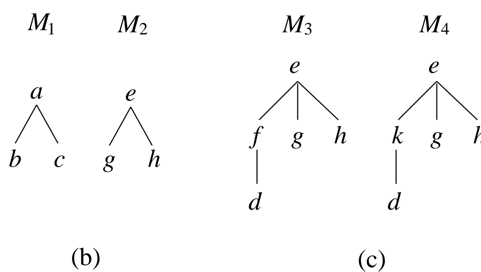


Figure 4.3: (a) The set $\mathcal{S}$ of three trees (these trees are hypothetical ones used solely for illustration purposes). (b) Two motifs exactly occurring in all three trees. (c) Two motifs approximately occurring, within distance 1, in all three trees.

4.2 Parallel Pattern Discovery in PLinda

4.2.1 Applying the E-dag Model

The computation models of the two pattern discovery applications fit well into the E-dag framework. Their key elements are listed in Table 4.1.

	Pattern Discovery in Protein Sequences	Pattern Discovery in RNA Secondary Structures
database	a set of protein sequences	a set of trees (representing RNA secondary structures)
pattern	partial sequence	subtree
good pattern	$occurrence_{pattern} > min_occurrence$	$occurrence_{pattern} > min_occurrence$
task	counting occurrences of *partial sequences* in subset of database	counting occurrences of *subtrees* in subset of database

Table 4.1: A comparison of specifications of two biological pattern discovery applications.

4.2.2 PLinda Implementation

We have implemented parallel sequence pattern discovery in PLinda. Two parallel E-tree traversal programs in PLinda have been implemented. In the first program, the master produces a work tuple for each top level pattern. Each worker then takes one work tuple and finishes the computation on the whole subtree. and puts the results in the tuple space. The master reads the results and concludes the program. This approach requires little communication between processes. It is *optimistic* in that it does not provide consideration for load-balancing. When computation on some of the branches are significantly more than that on the other branches, some workers may continue work for a long time while other workers wait. This is not efficient when a large number of workers have to wait during the whole computation. Figure 4.4 (Figure 4.5 respectively) shows the pseudo-PLinda code of the master (worker).

The function $child_pattern(node)$ generates all child patterns of the pattern on

```
child_pattern(root);
xstart;
    for (i=0; i<num_of_tasks; i++) {
        pl_in("done", ?j);
    }
xcommit;
out(poison tasks);
```

Figure 4.4: Pseudo-PLinda code of optimistic parallel sequence pattern discovery master.

```
while (!done) {
    xstart;
    in("task", ?node);
    if !(poison task) then {
        push node onto stack;
        while(stack not empty) {
            pop node from stack;
            compute goodness(node);
            if good(node) then {
               push child_pattern(node) onto stack;
            }
        }
        pl_out("done", node);
    } else {
        done=1;
    }
    xcommit;
}
```

Figure 4.5: Pseudo-PLinda code of optimistic parallel sequence pattern discovery worker.

the input *node*. Work tuples are of the form `("task", node)`. The function *good*(*node*) will determine if *node* is a good pattern according to goodness of all of its immediate subpatterns.

Our second program is *load-balanced*. The master produces top level tasks as in the optimistic version but the workers can generate work tuples themselves and put them in the tuple space. When the computation requires that more patterns to be looked at because a good pattern is found, the worker who has that branch will generate appropriate work tuples so that other idle workers can help. This approach adds some overhead to the computation in that there is some delay between the time a work tuple is created and the time when the work tuple is taken by another worker or even the same worker. Figure 4.6 (Figure 4.7, respectively) shows the pseudo-PLinda code of the master (worker).

The function *termination*(*node*) detects if the whole computation has completed. It works as follows.

1. Mark *node* as pruned (in this case no descendants of *node* will be visited);
2. Check if all siblings of *node* are pruned; if so, mark the parent of *node* as pruned;
3. If the root is pruned, the computation has completed.

4.3 Experimental Results

The protein sequence set we used in our experiments is *cyclins.pirx*. It has 47 sequences; the average length of a sequence is about 400. The E-tree for this data set has 20 top level patterns and 397 second level patterns. A single task takes anywhere from several seconds to several minutes to compute while the average is between 20 and 30 seconds.

```
child_pattern(root);
while !done {
    xstart;
    in("pruned", ?node);
    if termination(node) then {
        out(poison tasks);
        done=1;
    }
    xcommit;
}
```

Figure 4.6: Pseudo-PLinda code of load-balanced parallel sequence pattern discovery master.

```
while !done {
    xstart;
    in("task", ?node);
    if !(poison task) then {
        compute goodness(node);
        if good(node) then {
            child_pattern(node);
        } else {
            out("pruned", node);
        }
    } else {
        done=1;
    }
    xcommit;
}
```

Figure 4.7: Pseudo-PLinda code of load-balanced parallel sequence pattern discovery worker.

The machines we used in all our experiments (unless otherwise noted) are Sun Sparc 5 workstations with 32 MB RAM connected on a LAN.

4.3.1 Load-balanced vs. Optimistic

We ran our experiments with two sets of parameter settings, which are listed in Table 4.2. The efficiencies of the load-balanced program and the optimistic program are compared in figure 4.8 and figure 4.9. The efficiency of a parallel execution is defined as follows.

$$\text{Efficiency} = \frac{\text{speedup}}{\text{number of machines}} \times 100\%,$$

where

$$\text{speedup} = \frac{\text{sequential running time}}{\text{parallel running time}}.$$

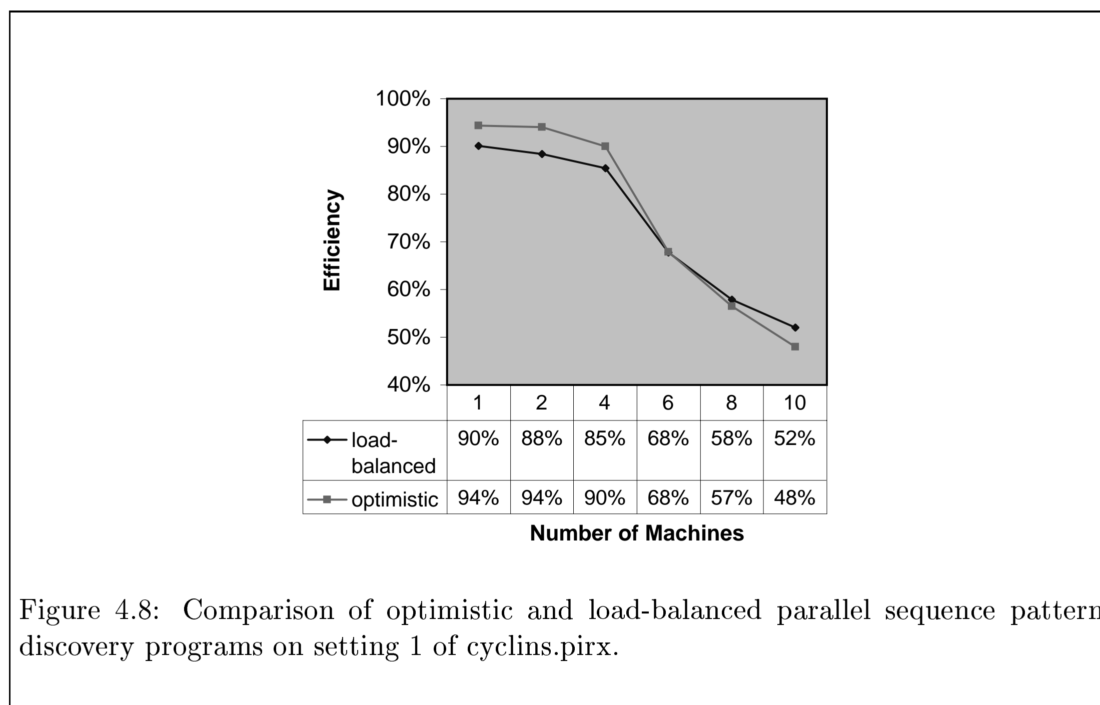


Figure 4.8: Comparison of optimistic and load-balanced parallel sequence pattern discovery programs on setting 1 of cyclins.pirx.

The experimental results confirm our analysis in the previous section. While the overhead in the load-balanced version makes it slower than the optimistic version when

Setting Number	Minimum Length	Minimum Occurrence	Maximum Mutations	Number of Motifs	Sequential Time (sec.)
Setting 1	12	5	0	3	1134
Setting 2	16	12	4	65	1299

Table 4.2: Parameter settings and sequential program results of cyclins.pirx.

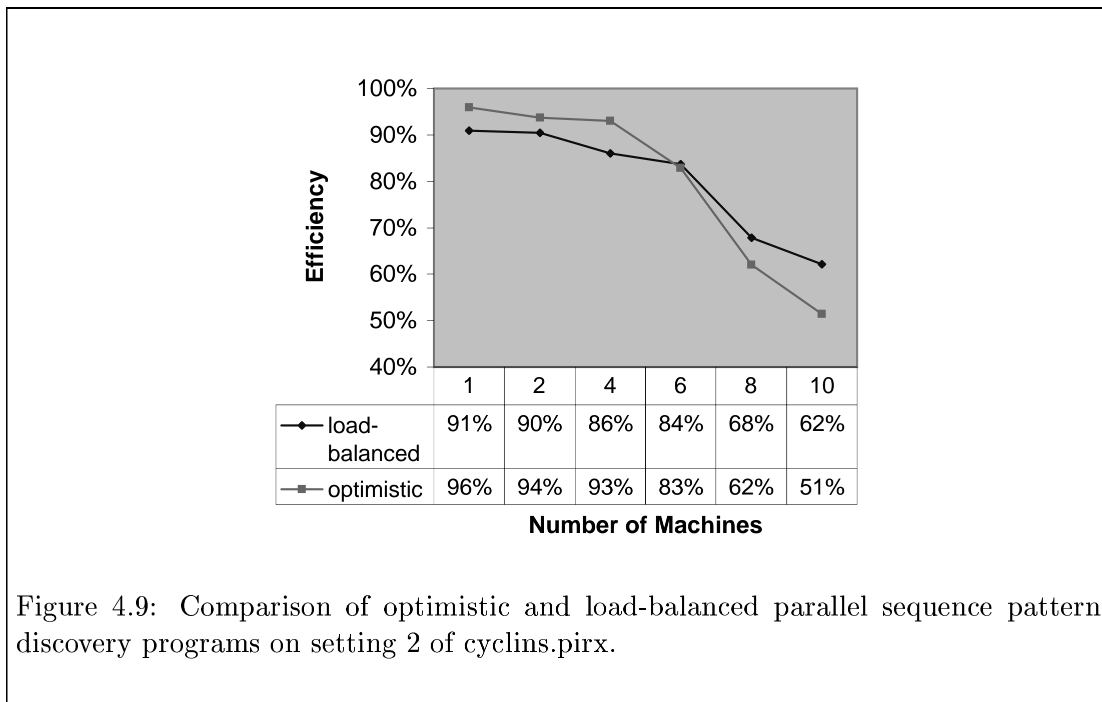
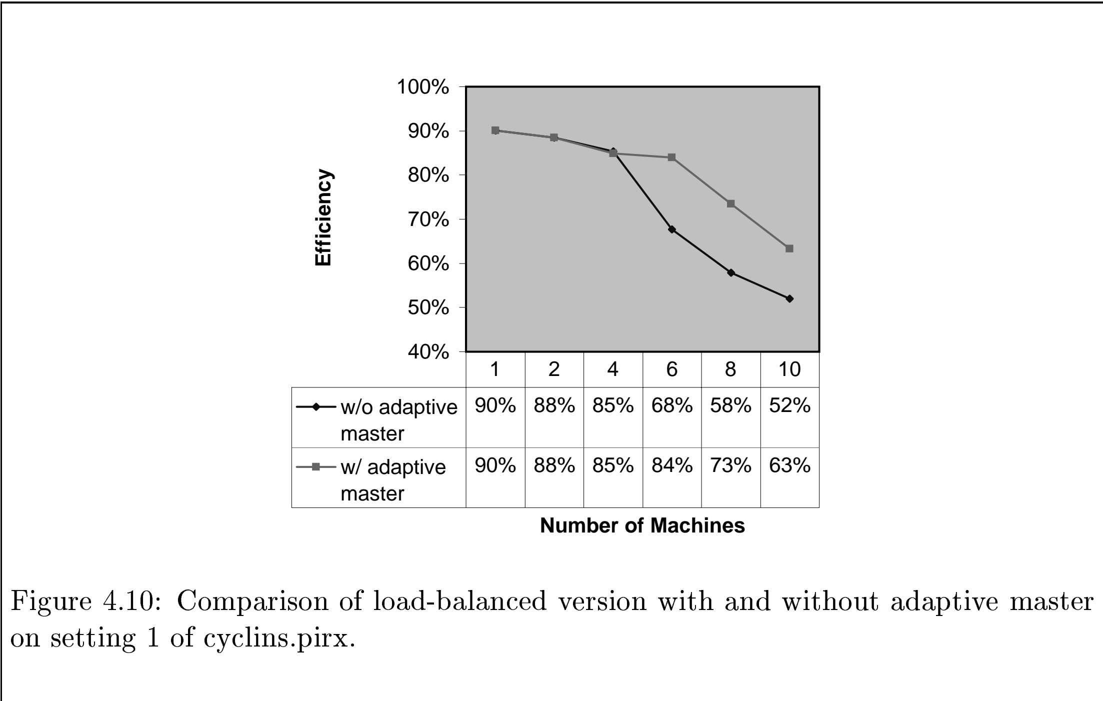


Figure 4.9: Comparison of optimistic and load-balanced parallel sequence pattern discovery programs on setting 2 of cyclins.pirx.

the number of machines is small (6 or less), it outperforms the optimistic version when there are more machines (8 and 10).

4.3.2 Adaptive Master



	1	2	4	6	8	10
w/o adaptive master	90%	88%	85%	68%	58%	52%
w/ adaptive master	90%	88%	85%	84%	73%	63%

Figure 4.10: Comparison of load-balanced version with and without adaptive master on setting 1 of cyclins.pirx.

Because of the limited number of initial tasks, when the number of workers increases, the chances of load imbalance also increases (even for the load-balanced version of the program). To alleviate this problem, the master can execute in the E-dag traversal mode for several levels and then generate the initial tasks for workers. For example, when the master goes to the second level in our experiments, it produced 387 work tuples. Our experience shows that when there are 6 or more machines, the master going to the second level is much more efficient than just going to the first level.

Therefore, the *adaptive master* determines the appropriate level for the initial

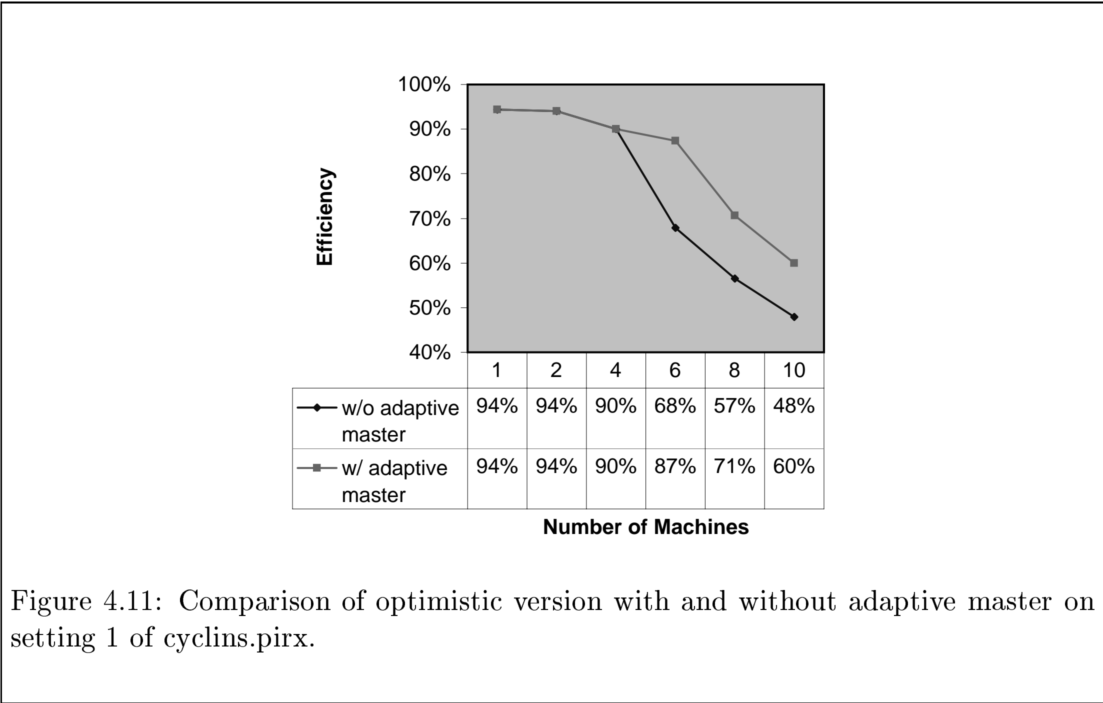


Figure 4.11: Comparison of optimistic version with and without adaptive master on setting 1 of cyclins.pirx.

tasks based on the number of workers. In our experiments, when the number of workers is 5 or less, the adaptive master goes down to only the first level, and when there are 6 or more machines, it goes down to the second level. Figure 4.10 to figure 4.13 show the improvements of efficiencies due to the adaptive master.

4.3.3 Experiments on a Large Network

We have put our load-balanced version with adaptive master to the test on a LAN of about 50 Sun Sparc workstations (they are not identical machines). Figure 4.14 shows the experiment results of the program running on 5, 10, 15, 20, 25, 30, 35, 40, and 45 machines. These experiments were run after 5pm at a major research lab. Good speedup is achieved even when as many as 45 machines joined the computation. For 15 and less machines, the speedup is particularly good.

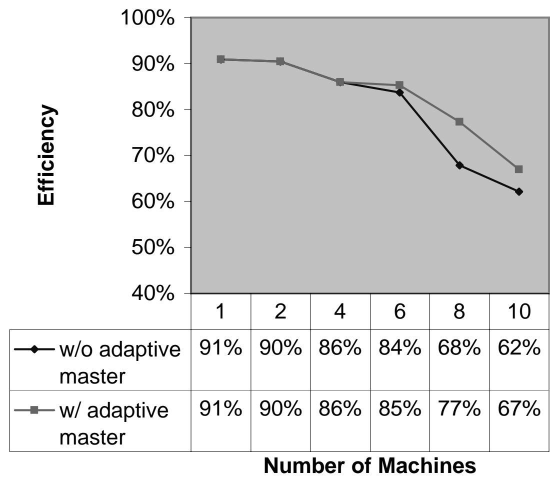


Figure 4.12: Comparison of load-balanced version with and without adaptive master on setting 2 of cyclins.pirx.

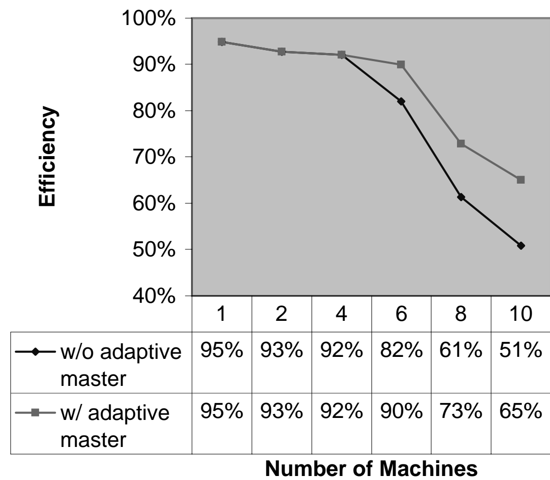


Figure 4.13: Comparison of optimistic version with and without adaptive master on setting 2 of cyclins.pirx.

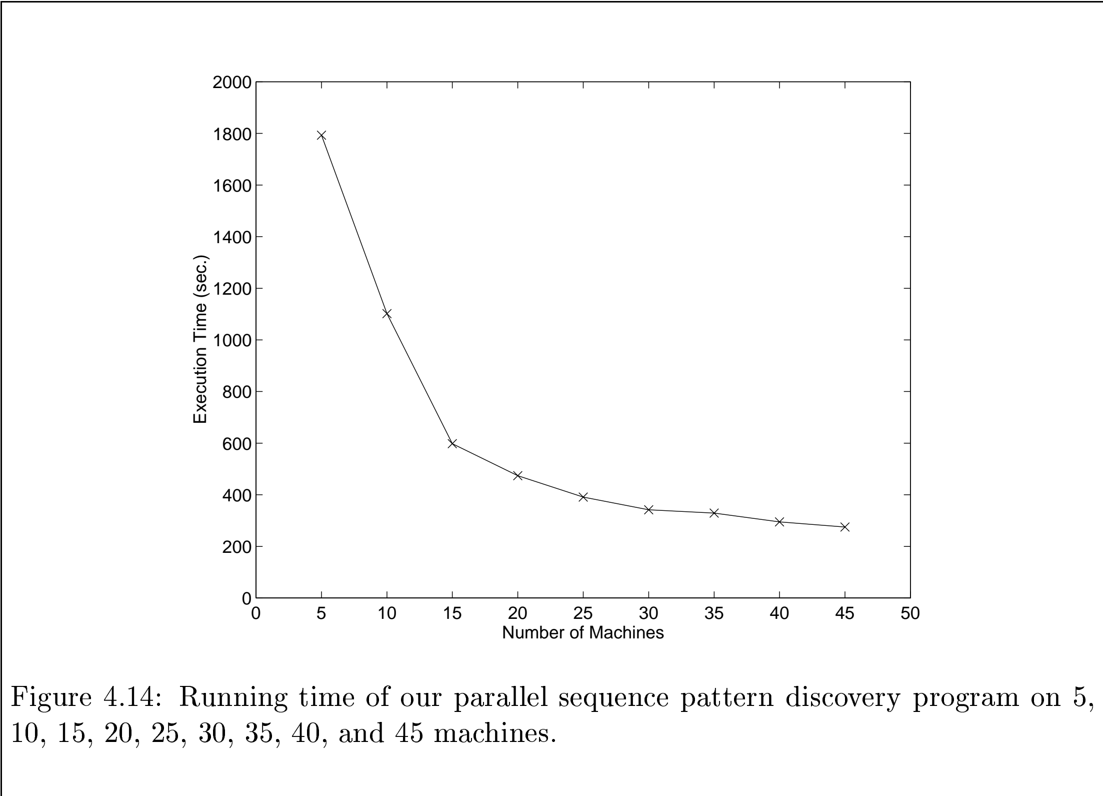


Figure 4.14: Running time of our parallel sequence pattern discovery program on 5, 10, 15, 20, 25, 30, 35, 40, and 45 machines.

Chapter 5

NyuMiner: Classification Trees by Optimal Sub-K-ary Splits

5.1 Introduction

The problem of learning classification trees is formulated as follows: A *training set* is a set of *data elements*, each of which is associated with particular values of a number of *independent variables* (or *attributes*). Each data element also takes a particular value of a *dependent variable* (this value is said to be the *class* of the data element). Each independent variable may be *categorical* (a finite set of distinct values, no ordering among values) or *numerical* (integer or real values, all values are ordered). The task is to develop a classifier in the form of a tree to predict the classes of further, unclassified data elements. For example, in a data set of potential heart disease patients, the record for each patient may show the age (integer), the weight (real), the blood pressure ("high", "medium", or "low"), and the test result ("positive" or "negative") of the patient. In this data set, blood pressure is a categorical attribute, age and weight are numerical

attributes, and test result is the dependent variable. A classification tree built from this data set can be used to predict whether a new patient has heart disease or not.

A classification tree is a decision tree. From each interior node n of a classification tree, the edges pointing to the children of n are associated with conditions (e.g. age > 50) on an independent variable. These conditions are mutually exclusive. At the leaves a decision (e.g. "positive") is suggested about the dependent variable. Building a classification tree is a top-down process in which each node is expanded by *splitting*—partitioning the data elements in a node into several sub-nodes that are more "pure". A set of data elements is *pure* if every element has the same value of the dependent variable. An *impure* set is one in which there is an equal distribution of values of the dependent variable among the data elements. At each tree node, each independent variable is considered a potential splitting variable. For each potential splitting variable, a best split is found from potentially an infinite number of possible splits. The best of these best splits is chosen to partition the data elements into sub-nodes. This goes on until all leaves of the tree are pure.

There are many criteria to measure the goodness of a split, but the most commonly used functions belong to the family of *impurity measures.* They include such well-known functions as the Gini index used in CART [24] and C4.5's information gain and gain ratio [65]. The difference between a node n's impurity and the weighted sum of its children's impurities (their *aggregate impurity*) measures the goodness of the split. The bigger the difference, the better the split. Many classification algorithms limit each split to a fixed number of branches. For example, CART and SLIQ [59] allow only binary splits for both categorical and numerical variables. C4.5 allows only binary splits for numerical variables and only fixed m-way split for categorical variables where m is the cardinality of the category. Allowing a variable to appear more than once on a path

from the roof to a leaf can compensate to some extent by providing a way of eventually splitting the variable into finer partitions, but this damages the intelligibility of the tree. And unfortunately, the repetitive binarization of a variable cannot guarantee an optimal multi-way split even if each binary split is optimal [37]. *NyuMiner* is a classification tree algorithm which selects an optimal multi-branch split with respect to any given impurity function and any given maximum number of branching at every tree node.

The rest of this chapter is organized as follows. In section 5.2, we discuss related work. In section 5.3, we describe the algorithm to find optimal sub-K-ary splits. In the next section, we discuss the use of cross validation for pruning and offer an alternative to pruning—rule selection. In section 5.5, we compare NyuMiner to C4.5 and CART in terms of classification accuracy on 7 benchmark data sets. In this section, we also present the results on complementarity tests on the same benchmark data sets. In section 5.6, we describe a real world application of NyuMiner—predicting foreign exchange rate movement.

5.2 Related Work

There has been some work on how to obtain an optimal split for numerical variables. Let us first present the problem with an example used in [33]. There are a set of 27 data elements as shown in Figure 5.1. The variable being considered is a numerical variable of which each data element has a value between 0 and 9. Each data element belongs to one of 3 different classes "A", "B", and "C".

The standard technique is to sort the data elements in an ascending (or descending) order by values of the variable in question and then consider each adjacent pair of data elements as a potential *cut point* for a split. Clearly, it is impossible to divide

Item	1	2	3	4	5	6	7	8	9	10	11	12	13	...
Class	A	A	A	A	B	B	B	B	B	C	C	C	B	...
Value	0	0	0	1	1	1	1	2	2	3	3	3	4	...

...	14	15	16	17	18	19	20	21	22	23	24	25	26	27
...	B	B	C	A	A	A	C	C	C	C	C	C	C	C
...	4	4	4	5	5	6	7	7	7	8	8	9	9	9

Figure 5.1: A set of 27 data elements with values of a numerical variable showing.

the data between two data elements that have the same value. Thus, we can collapse all data elements with the same value into one basket and consider only cut points in between baskets. The data elements in Figure 5.1 are thus grouped into the 10 baskets in Figure 5.2.

Class	AAA	ABBB	BB	CCC	BBBC	AA	A	CCC	CC	CCC
Value	0	1	2	3	4	5	6	7	8	9

Figure 5.2: The set of 27 data elements grouped into 10 baskets by value.

Class Label	A	M	B	C	M	A	A	C	C	C
Value	0	1	2	3	4	5	6	7	8	9

Figure 5.3: The 10 baskets with class labels.

If all data elements in a basket belong to the same class, we label the basket with the class symbol. If a basket has a mixed class distribution, we label the basket with an "M". The resultant baskets with their class labels are shown in Figure 5.3. Finally, by combining adjacent baskets having the same class labels (except for adjacent "M" baskets), we get the 7 baskets in Figure 5.4. The points between adjacent baskets are called *boundary points* [36] [1].

[1]Definition of boundary points: A value T in the range A is a boundary point if and

Class Label	A	M	B	C	M	A	C
Values	0	1	2	3	4	5,6	7-9

Figure 5.4: 7 baskets divided by boundary points.

Fayyad and Irani proved that optimal binary splits always fall on these boundary points when using *average class entropy* as the impurity measure. The average class entropy for a binary split T on set S

$$E(T) = \frac{|S_1|}{|S|} Ent(S_1) + \frac{|S_1|}{|S|} Ent(S_2),$$

where

$$Ent(S) = -\sum_{i=1}^{J} P(C_i, S) \log(P(C_i, S)),$$

where J is the number of classes and $P(C_i, S)$ stands for the proportion of data elements in S of class C_i. This led them to propose a greedy top-down method of recursively applying binary splitting to a numerical variable in order to obtain the optimal split on that variable [37]. Although this scheme has been reported to give good results in practice, no guarantee can be given to the resulting multi-branch split induced by recursive optimal binary splitting. More recently, Elomaa and Rousu [33] proved that only boundary points need to be inspected in order to find the optimal multi-way split of a numerical variable. Their proof relies on Fayyad and Irani's theorems and assumes average class entropy as the impurity measure.

---

only if in the sequence of examples sorted by the value of A, there exist two examples $e_1, e_2 \in S$, having different classes, such that $A(e_1) < T < A(e_2)$; and there exists no other example $e' \in S$ such that $A(e_1) < A(e') < A(e_2)$.

5.3 Optimal Sub-K-ary Splits

To formalize the problem, let us assume that we are to select a split at a node that has a set s of N data elements. A sub-K-ary split is a split that partitions s into K or fewer baskets. Our goal is to find optimal sub-K-ary splits for both numerical and categorical variables for any given K with respect to any given impurity measure.

Definition 5 *An* impurity function *is a function ϕ defined on the set of all J-tuples $(p_1, p_2, \ldots, p_J)$ where J is the number of classes, $0 \geq p_j \geq 1, j = 1, 2, \ldots, J$, and $\sum_{j=1}^{J} p_j = 1$. ϕ has the following properties:*

1. *ϕ achieves its maximum only at the point $(\frac{1}{J}, \frac{1}{J}, \ldots, \frac{1}{J})$;*
2. *ϕ achieves its minimum only at the points $(1, 0, \ldots, 0)$, $(0, 1, \ldots, 0)$, $\ldots$, $(0, 0, \ldots, 1)$;*
3. *ϕ is a symmetric function of $p_1, p_2, \ldots, p_J$;*
4. *ϕ is strictly concave in the sense that for $r + s = 1$, $r \geq 0$, $s \geq 0$,*

   $$\phi(rp_1 + sp'_1, rp_2 + sp'_2, \ldots, rp_J + sp'_J) \geq r\phi(p_1, p_2, \ldots, p_J) + s\phi(p'_1, p'_2, \ldots, p'_J)$$

   and the equality holds only when

   $$p_j = p'_j, 1 \leq j \leq J.$$

The last property of impurity functions implies that if either partition in a binary split has a different class distribution from that of the other partition (or from that of the original set), the split makes the original set more "pure". Popular impurity measures such as the Gini index and the information gain all satisfy this definition.

Suppose a split S partitions a set of N data elements into k subsets $s_1, s_2, \ldots, s_k$. The impurity of a subset s_i

$$I(s_i) = \phi(\frac{c_{i1}}{n_i}, \frac{c_{i2}}{n_i}, \ldots, \frac{c_{iJ}}{n_i}),$$

where c_{ij} is the number of elements of class j in s_i, $1 \leq j \leq J$, n_i is the total number of elements in subset s_i, and $\sum_{i=1}^{k} n_i = N$. The aggregate impurity of all partitions in the split S

$$I(S) = \sum_{i=1}^{k} \frac{n_i}{N} I(s_i).$$

Definition 6 *S is an* optimal K-ary split *if and only if there does not exist another K-ary split S' and $I(S') < I(S)$.*

Definition 7 *S is an* optimal sub-K-ary split *if and only if*

1. *$I(S) \leq I(S_k)$ for any optimal k-ary split S_k, $1 < k \leq K$;*
2. *There does not exist a split S' such that $I(S') \leq I(S)$ and S' partitions the data elements into fewer baskets than S does.*

In other words, an optimal sub-K-ary split is one with the fewest branches among all sub-K-ary splits having the least aggregate impurity.

Lemma 4 *If two partitions in a split S are merged into one forming split S', then $I(S) \leq I(S')$.*

Proof. Let us consider a k-ary split S and a $(k-1)$-ary split S' resulted from merging two partitions in S into one (Figure 5.5). Without loss of generality, we assume that partitions $k-1$ and k in S merged into partition $(k-1)'$ in S'. We have,

$$I(S) = \frac{n_1}{N} I(s_1) + \frac{n_2}{N} I(s_2) + \ldots + \frac{n_{k-1}}{N} I(s_{k-1}) + \frac{n_k}{N} I(s_k)$$

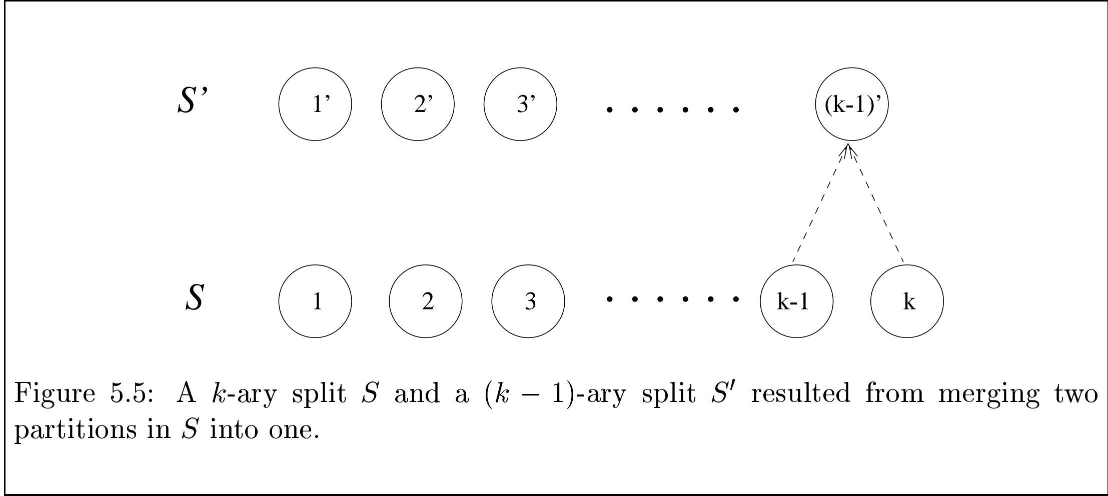


Figure 5.5: A k-ary split S and a $(k-1)$-ary split S' resulted from merging two partitions in S into one.

and

$$I(S') = \frac{n_{1'}}{N} I(s_{1'}) + \frac{n_{2'}}{N} I(s_{2'}) + \ldots + \frac{n_{(k-1)'}}{N} I(s_{(k-1)'}).$$

According to Property 4 of impurity functions,

$$I(s_{(k-1)'}) \geq \frac{n_{k-1}}{n_{(k-1)'}} I(s_{k-1}) + \frac{n_k}{n_{(k-1)'}} I(s_k).$$

Thus,

$$\begin{aligned} I(S') &\geq \frac{n_{1'}}{N} I(s_{1'}) + \frac{n_{2'}}{N} I(s_{2'}) + \ldots + \frac{n_{(k-1)'}}{N} [\frac{n_{k-1}}{n_{(k-1)'}} I(s_{k-1}) + \frac{n_k}{n_{(k-1)'}} I(s_k)] \\ &= \frac{n_{1'}}{N} I(s_{1'}) + \frac{n_{2'}}{N} I(s_{2'}) + \ldots + \frac{n_{k-1}}{N} I(s_{k-1}) + \frac{n_k}{N} I(s_k) \\ &= I(S). \end{aligned}$$

This holds for every pair of S and S' for every $k > 1$. Therefore, the lemma is true. □

Let us look at the example introduced in the previous section again. When the data elements are sorted and grouped into the 10 baskets of Figure 5.3, we can think of these 10 baskets as partitions in a split S_0. By inspection, S_0 is an optimal 10-way split. In fact, only basket 1 and 4 contribute to the aggregate impurity of the split. The other baskets all have zero impurity. Thus, when we combine adjacent baskets with the same

class labels (that are not "M"'s) into one basket, which results into the 7-way split S_1 of Figure 5.4, the aggregate impurity does not change. And from Lemma 4 we know that there does not exist another 7-way split with less aggregate impurity. Furthermore, according to Property 4 of impurity functions, any more merging of partitions will increase the aggregate impurity because any merge now will involve partitions with different class distributions. Therefore, S_1 is an optimal sub-K-ary split for all $K \geq 7$. In general, we have the following theorem.

Theorem 5 *If S_1 is the split whose cut points are all boundary points and only boundary points, B is the number of baskets in S_1, and N is the total number of data elements before splitting, then S_1 is an optimal sub-K-ary split for all $B \leq K \leq N$.*

5.3.1 Numerical Variables

In practice, it is usually necessary to have an upper bound K on the number of partitions each split can have. If $K \geq B$, S_1 is an optimal sub-K-ary split. If $K < B$, we have to use a dynamic programming technique to find an optimal sub-K-ary split for numerical variables.

Let $I(k, j, i)$ denotes the minimum aggregate impurity that results when baskets j through i are partitioned into k intervals. We define that

$$I(k, 1, i) = \min_{1 \leq j < i} [\frac{\sum_{l=1}^{l \leq k-1} n_l}{N} I(k-1, 1, j) + \frac{n'}{N} I(1, j+1, i)],$$

where n_l is the number of elements in partition l of the split that minimizes $I(k-1, 1, j)$, n' is the number of elements in baskets $j+1$ through i, and $\sum_{l=1}^{l \leq k-1} n_l + n' = N$.

Thanks to Theorem 5, the set of data elements can first collapse to form the B-way split S_1. Then, the optimal K-ary split is the one that minimizes $I(K, 1, B)$. The optimal sub-K-ary split is the one that minimizes $I(k, 1, B)$ for all $k \leq K$. This

leads to an algorithm with an asymptotic time complexity of $O(KB^2)$. In the worst case ($B = N$), the complexity is $O(KN^2)$.

5.3.2 Categorical Variables

Categorical variables can be treated exactly the same way as numerical variables if we pretend values in the category are ordered. B can simply be the number of distinct values in the category. Therefore, the optimal sub-K-ary split can be obtained by finding an optimal sub-K-ary split for every possible order of the values and choosing the one with the least aggregate impurity. The time complexity of this algorithm is $O(B! \times KB^2)$. Normally B is small, but when it is big, the running time may be a concern.

This can be improved by the following optimization. Let the set of distinct values in the category be V (so $B = |V|$). If all data elements having a value v belong to the same class c, we label v with c's class symbol. Otherwise, we label v with an "M" (for "mixed"). Let s_c denotes the set of distinct values that are labeled with c's class symbol. It is easy to prove that in an optimal sub-K-ary split, all values in s_c must be in one basket. We then use a logical value v_c to represent all values in s_c. Let V_L denotes the set of all logical values and "M"-labeled values. Clearly, $|V_L| \leq |V|$ and the equality holds only when no logical value represents more than one real value. Therefore, using V_L instead of V in the above algorithm can reduce the running time (by reducing B) without compromising the correctness of the algorithm.

5.4 To Prune or Not to Prune

Growing a full-sized tree is not the end of a classification tree algorithm. Because the trees have been grown to "fit" the training data, they have to be pruned so that they work better on new, unclassified data. CART uses a technique called *minimal cost complexity pruning with V-fold cross validation*, which is proved very effective. A flavor of NyuMiner, *NyuMiner-CV*, adopts this pruning technique.

5.4.1 Minimal Cost Complexity Pruning

Cost complexity is a measure of the resubstitution error of a tree further penalized by the size of the tree. The first step is to grow a maximum sized tree T_{max} by letting the splitting procedure continue until all leaves are pure. Let us use $\widetilde{T}$ to denote the set of terminal nodes (leaves) in a tree T.

Definition 8 *For any subtree $T \leq T_{max}$, define its complexity as $|\widetilde{T}|$, the number of terminal nodes in T. Let $\alpha \geq 0$ be a real number called the complexity parameter and define the cost complexity measure $R_\alpha(T)$ as*

$$R_\alpha(T) = R(T) + \alpha|\widetilde{T}|,$$

where $R(T)$ is the resubstitution error estimate[2] *of T.*

Thus, $R_\alpha(T)$ is a linear combination of the cost of the tree and its complexity. For any node $t \in T$, denote by $\{t\}$ the sub-branch of T_t consisting of the single node $\{t\}$. Set

$$R_\alpha(t) = R(t) + \alpha.$$

[2]The resubstitution error estimate of a classification tree is the number of training data misclassified by the classification tree.

For any branch T_t, define

$$R_\alpha(T_t) = R(T_t) + \alpha|\widetilde{T}_t|.$$

As long as

$$R_\alpha(T_t) < R_\alpha(t),$$

the branch T_t has a smaller cost complexity than the single node $\{t\}$. But at some critical value of α, the two cost complexities become equal. At this point the sub-branch t is smaller than T_t, has the same cost complexity, and is therefore preferable. To find this critical value of α, solve the inequality

$$R_\alpha(T_t) < R_\alpha(t),$$

getting

$$\alpha < \frac{R(t) - R(T_t)}{|\widetilde{T}_t| - 1}.$$

Let T_1 denote the smallest subtree of T_{max} satisfying

$$R(T_1) = R(T_{max}).$$

Define a function $g_1(t)$, $t \in T_1$, by

$$g_1(t) = \begin{cases} \frac{R(t)-R(T_t)}{|\widetilde{T}_t|-1} & t \notin \widetilde{T_1} \\ +\infty & t \in \widetilde{T_1} \end{cases} \tag{5.1}$$

Then define the *weakest link* $\overline{t_1}$ in T_1 as the node such that

$$g_1(\overline{t_1}) = \min_{t \in T_1} g_1(t)$$

and set

$$\alpha_2 = g_1(\overline{t_1}).$$

The node $\overline{t_1}$ is the weakest link in the sense that as the parameter α increases, it is the first node such that $R_\alpha(\{t\})$ becomes equal to $R_\alpha(T_t)$. Then $\{\overline{t_1}\}$ becomes preferable to $T_{\overline{t_1}}$, and α_2 is the value of α at which the equality occurs.

Define a new tree $T_2 < T_1$ by pruning away the branch $T_{\overline{t_1}}$, that is,

$$T_2 = T_1 - T_{\overline{t_1}}.$$

Now, using T_2 instead of T_1, find the weakest link $\overline{t_2}$ in T_2 and define

$$T_3 = T_2 - T_{\overline{t_2}}.$$

Continuing this procedure, we get a sequence of subtrees of decreasing sizes

$$T_1 > T_2 > T_3 > \ldots > \{t_0\}$$

where t_0 is the root of all trees. The problem now is to select one of these as the optimum-sized tree.

In V-fold cross validation, the original learning sample L is divided by random selection into V subsets, $L_v, v = 1, 2, \ldots, V$, each containing the same number of data elements (as nearly as possible). Then the vth learning sample is

$$V^{(v)} = L - L_v, v = 1, 2, \ldots, V,$$

so that $V^{(v)}$ contains the fraction $(V-1)/V$ of the total data elements. V auxiliary trees are then grown together with the main tree grown on L. The vth auxiliary tree is grown using the learning sample $V^{(v)}$.

For each value of the complexity parameter α, let $T(\alpha), T^{(v)}(\alpha), v = 1, 2, \ldots, V$, be the corresponding minimal cost complexity subtree of T_{max}, $T_{max}^{(v)}$. For each v, the trees $T_{max}^{(v)}$, $T^{(v)}(\alpha)$ have been constructed without ever seeing the data elements in L_v. Thus, the data elements in L_v can serve as an independent test set for the tree $T^{(v)}(\alpha)$.

Put L_v down the tree $T_{max}^{(v)}, v = 1, 2, \dots, V$. Fix the value of the complexity parameter α. For every value of v, i, j, define

$$N_e^{(v)} = \text{the number of data elements misclassified by } T^{(v)}(\alpha),$$

and set

$$N_e = \sum_{v=1}^{V} N_e^{(v)},$$

so N_e is the total number of misclassified test data.

The idea now is that for V large, $T^{(v)}(\alpha)$ should have about the same classification accuracy at $T(\alpha)$. Therefore, the cross validation error estimate

$$R^{CV}(T(\alpha)) = \frac{1}{N} N_e.$$

Although α may vary continuously, the minimal cost complexity trees grown on L are equal to T_k for $\alpha_k \leq \alpha \leq \alpha_{k+1}$. Set

$$\alpha_k' = \sqrt{\alpha_k \alpha_{k+1}}$$

so that α_k' is the geometric midpoint of the interval such that $T(\alpha) = T_k$. Then

$$R^{CV}(T_k) = R^{CV}(T(\alpha_k')).$$

That is, $R^{CV}(T_k)$ is the estimate gotten by putting the test set L_v through the trees $T^{(v)}(\alpha_k')$. Now the rule for selecting the right sized tree is: Select the tree T_{k_0} such that

$$R^{CV}(T_{k_0}) = \min_k R^{CV}(T_k).$$

Breiman *et al.* argue from their empirical studies that V should be set to about 10. Thus, we use 10-fold cross validation for CART and NyuMiner-CV in all our experiments in this chapter.

5.4.2 Multiple Incremental Sampling and Rule Selection

Another flavor of NyuMiner, *NyuMiner-RS*, offers an alternative to pruning by combining *multiple incremental sampling* and *rule selection*.

Multiple incremental sampling is like the *windowing* technique used in C4.5. It starts with a randomly selected subset of training set, which is used to build an initial classification tree. This tree is then used to classify the remaining data elements (those that were not selected); usually with the result that some are misclassified. A selection of these "difficult" data elements is then added to the initial training set. This enlarged training set is used to build a second tree, which is again tested on the remaining data elements. The cycle is repeated until a tree from the current training set correctly classifies all the remaining data elements or all data elements are used to build the tree. The final training set, usually a small portion of the data elements, can be thought of as a screened set of data elements that contains all the ones needed to guide the tree-building.

Different initial training sets generally lead to different initial trees and quite frequently to different final trees. Like the windowing technique, NyuMiner-RS generate several alternate trees from different initial training sets. Unlike the windowing technique, NyuMiner builds a *classifying rule list* from these trees instead of using any single tree to classify unseen data. The technique for building a classifying rule list is called rule selection which is inspired by PRIM [39], where a sequence of induced subregions of the independent variable space can be regarded as an ordered list that are used to classify. Every node in a classification tree can be thought of as a classifying rule. Each rule r has a *condition*, a *decision class*, a *confidence*, and a *support*. The condition of r is the conjunction of variable-value pairs obtained by traversing from

the root to the node representing r and the decision class of r is the majority class of these data elements. The confidence of r $Conf(r)$ is the percentage of data elements of the majority class over all data elements in the node. The support of r $Supp(r)$ is the percentage of data elements in the node over all data elements in the training set. (The concepts of confidence and support of a classification rule are very similar to those of an *association rule*).

Let R be the set of all rules (from all alternate trees) whose confidence is greater than or equal to some threshold C_{min} and whose support is greater than or equal to some threshold S_{min}. Let us define a partial order on R.

Definition 9 *A rule r is ordered higher than another rule r' ($r > r'$) if and only if $Conf(r) > Conf(r')$ and $Supp(r) > Supp(r')$.*

Thus, if we sort the rules in R in descending order, we have a classifying rule list. Given a test case, we find the first rule whose condition matches the test case. If there are several matching rules that have the same order, we choose the one with the highest confidence. The decision class of the chosen rule is the classification of the test case. Note that C_{min} should be greater than the confidence of the rule represented by the root (call this the *plurality rule*). And S_{min} should be greater than $\frac{1}{N}$ where N is the total number of data elements in the training set.

5.5 Comparing NyuMiner to C4.5 and CART

We used seven benchmark data sets in our experiments to compare NyuMiner to C4.5 and CART. Most of these data sets are from the UCI Machine Learning Database Repository. However, we re-prepared each data set in the way described below. The C4.5 program we used is the release 8 of the program distributed with Quinlan's book

"C4.5: Programs for Machine Learning". We used the implementation of CART in Version 2.1 of the IND package by Buntine [23].

5.5.1 Description of Data Sets

Data Set	Description
diabetes	Predicting whether a patient has diabetes based on data recorded on possible diabetes patients (several weeks' to months' worth of glucose, insulin, and lifestyle data per patient and a description of the problem domain).
german	Predicting whether annual income exceeds $50K based on census data of Germany.
mushrooms	Predicting whether a mushroom is poisonous or edible based on attributes of physical characteristics.
satimage	From multi-spectral values of pixels in 3x3 neighbourhoods in a satellite image, classifying the central pixel in each neighbourhood.
smoking	Predicting attitude towards restrictions on smoking in the workspace (prohibited, restricted, or unrestricted) based on bylaw-related, smoking-related, and sociodemographic covariates.
vote	To classify a Congressman as a Democrat or a Republican based on the votes for each U. S. House of Representatives Congressman on the 16 key votes identified by the Congressional Quarterly Almanac.
yeast	Predicting the cellular localization sites of proteins.

Table 5.1: Descriptions of the 7 benchmark data sets.

Table 5.1 has a brief description for each of the seven benchmark data sets. Relevant statistical features such as number of numerical and categorical attributes, number of classes, and percentage of missing values of each data set are listed in Table 5.2.

Data Set	Total Number of Cases	% of Cases with At Least One Missing Value	% of Missing Values Over All Values	Variables			Number of Classes
				Catego-rical	Nume-rical	Total	
diabetes	768	0.0%	0.0%	0	8	8	2
german	1000	0.0%	0.0%	13	7	20	2
mushrooms	8124	30.5%	1.4%	22	0	22	2
satimage	6434	0.0%	0.0%	0	36	36	7
smoking	2854	0.0%	0.0%	10	3	13	3
vote	435	46.7%	5.8%	16	0	16	2
yeast	1483	0.0%	0.0%	0	8	8	10

Table 5.2: Statistical features of the 7 benchmark data sets.

5.5.2 Comparison of Accuracy

For each data set, we first joined the original training set and testing set to form a complete set. We then randomly divided each complete set into almost-equal subsets while maintaining the same class distribution in both subsets: We first partition data elements into baskets according to their class values. Suppose we are to order the data elements in a basket and there are n possible permutations of ordering. We randomly select a number k between 1 and n. Then from the kth permutation, the odd-indexed data elements go to the first subset and the even-indexed data elements go to the second subset (the two subsets might not have exactly the same number of data elements). Repeat this for all baskets.

The first subset is used as training set and the second as testing set. We prepared 10 training–testing set pairs for each data set. A classifier's classifying accuracy on a data set is the average accuracy over the 10 pairs. Table 5.3 compares the classifying accuracies of the four classifiers (C4.5, CART, NyuMiner-CV, NyuMiner-RS) on each data set.

NyuMiner-RS scored the highest on 5 of the 7 data sets and NyuMiner-CV scored the highest on 3 data sets (the numbers do not add up to 7 because of the tie on mushrooms). While NyuMiner-CV uses exactly the same pruning technique as in CART, its accuracy is either higher or the same on all 7 data sets. This is expected because of the optimal multi-way splits. However, it is interesting to note that the difference is not much, which may imply that: 1. Although not optimal in general, binary splits may be very effective in practice; 2. At least sometimes, pruning plays a more significant role than the quality of splits.

Data Set	Plurality Rule	C4.5	CART	NyuMiner-CV	NyuMiner-RS
diabetes	65.1%	73.6%	73.0%	73.8%	**74.4%**
german	60.0%	72.0	72.0	**72.3%**	71.8%
mushrooms	51.8%	**100.0%**	**100.0%**	**100.0%**	**100.0%**
satimage	23.8%	85.0%	84.9%	85.2%	**86.8%**
smoking	69.5%	67.1%	69.5%	69.5%	**69.6%**
vote	61.4%	94.7%	94.7%	94.7%	**95.2%**
yeast	31.2%	54.6%	56.0%	**56.3%**	55.5%

Table 5.3: Comparison of classification accuracies of C4.5, CART, NyuMiner-CV, and NyuMiner-RS.

NyuMiner-RS generates the same splits as NyuMiner-CV does, but it did better on 4 of the 7 data sets and was very close or equal to NyuMiner-CV in the other 3 data sets. This demonstrates that multiple incremental sampling plus rule selection is a viable alternative to pruning.

5.5.3 Complementarity Tests

We compared C4.5, CART, and NyuMiner-RS' decisions on the testing set of each benchmark data set. They are said to "all agree" on a test case when all three give the

same classification. Table 5.4 summarized the results on the 7 testing sets.

Data Set	Total Test Cases	All Agree			Not All Agree		
		Subtotal	Coverage	Accuracy	Subtotal	Coverage	% of At Least One Correct
diabetes	384	337	87.8%	**76.3%**	47	12.2%	100.0%
german	500	356	71.2%	**81.2%**	144	28.8%	100.0%
mushrooms	4062	4062	100.0%	**100.0%**	0	0.0%	N/A
satimage	3137	2513	80.1%	**94.8%**	624	19.9%	87.0%
smoking	1425	1303	91.4%	**69.7%**	122	8.6%	99.2%
vote	217	213	98.2%	**96.7%**	4	1.8%	100.0%
yeast	738	432	58.5%	**67.4%**	306	41.5%	77.5%

Table 5.4: Complementarity test results on the benchmark data sets.

The table shows that when the three classifiers agree, the classification has a higher accuracy than any one of them alone (except in the case of "mushrooms" where there is no room for improvement). For "german", "satimage", and "yeast" especially, the improvement is substantial.

The data sets that have more than three classes are "satimage" (7) and "yeast" (10). It is useful to note that for these data sets, when not all three classifiers agree, the likelihood that at least one of them is correct (87.0%, 77.5%) is significantly higher than their respective plurality rule accuracy (23.8%, 31.2%).

NyuMiner generates classification trees with different structures from those generated by C4.5 and CART. When a data set has numerical variables, their tree structures can be drastically different. This partially explains the complementarity among NyuMiner, C4.5, and CART.

5.6 Application: Making Money in Foreign Exchange

5.6.1 Preparing Data

We started with a list of daily exchange rate between Japanese Yen and U. S. Dollar for the past 27 years (1971–1997). For each day, we generated the following 10 values from the rate data: 1. Percentage change of today's rate over yesterday's (**one**, for shorthand); 2. Percentage change over the day before yesterday (**two**); 3. Percentage change over three days ago (**three**); 4. Percentage change over four days ago (**four**); 5. Percentage change over five days ago (**five**); 6. Average percentage change in the last five days (a business week) (**average**); 7. Weighted average percentage change in the last five days (**weighted**); 8. Percentage change over a month ago (**month**); 9. Percentage change over six months ago (**six-month**); 10. Percentage change over a year ago (**year**).

For the dependent variable, we used the movement of tomorrow's rate with respect to today's rate (1 for up, -1 for down). Thus, we have a data set for predicting tomorrow's movement of the exchange rate between Japanese Yen and U. S. Dollar based on historical rates. We did the same for four other pairs of currencies. The five prepared data sets are listed in Table 5.5[3].

We then divided each data set into equal halves. The first half covers roughly data from 1972–1984 and the second half covers roughly data from 1985–1997.

[3]Due to unavailability of original rates, every data set does not have the same number of data elements.

Currency Pair	Data Set	Number of Data Elements
Japanese Yen vs. U. S. Dollar	yu	5904
Deutcshe Mark vs. U. S. Dollar	du	6076
Japanese Yen vs. Deutcshe Mark	yd	6162
French Franc vs. U. S. Dollar	fu	6344
U. S. Dollar vs. Great Britain Sterling	up	6419

Table 5.5: Descriptions of foreign exchange data sets.

5.6.2 Selecting Rules

We used the first half of each data set to build classification trees. When we used the second half to test the trees, all four classifiers (C4.5, CART, NyuMiner-CV, and NyuMiner-RS) did a poor job (accuracies range from 49% to 52%). Complementarity tests turned out to be not so useful either.

Fortunately, it is acceptable for a classifier to be non-decisive on most days because foreign exchange traders do not trade every day anyway. This inspired us to use the rule selection technique to choose just the best rules from each tree and use only these rules in predicting the rate movement.

Let us use the **yu** data set as an example to illustrate how this is done. Figure 5.6 shows the complete first level and a partial second level of a tree that NyuMiner-RS built. On each intermediate node, the first line is the splitting attribute, the second line is the decision of that node, and the third line is the confidence and support (in that order) of the rule represented by the node. The edges from an intermediate node to its children are associated with ranges of the splitting attribute on the intermediate node. These ranges are shown in basis points (a basis point is one-hundredth of one percent).

We chose the confidence threshold C_{min} to be 80% and the support threshold

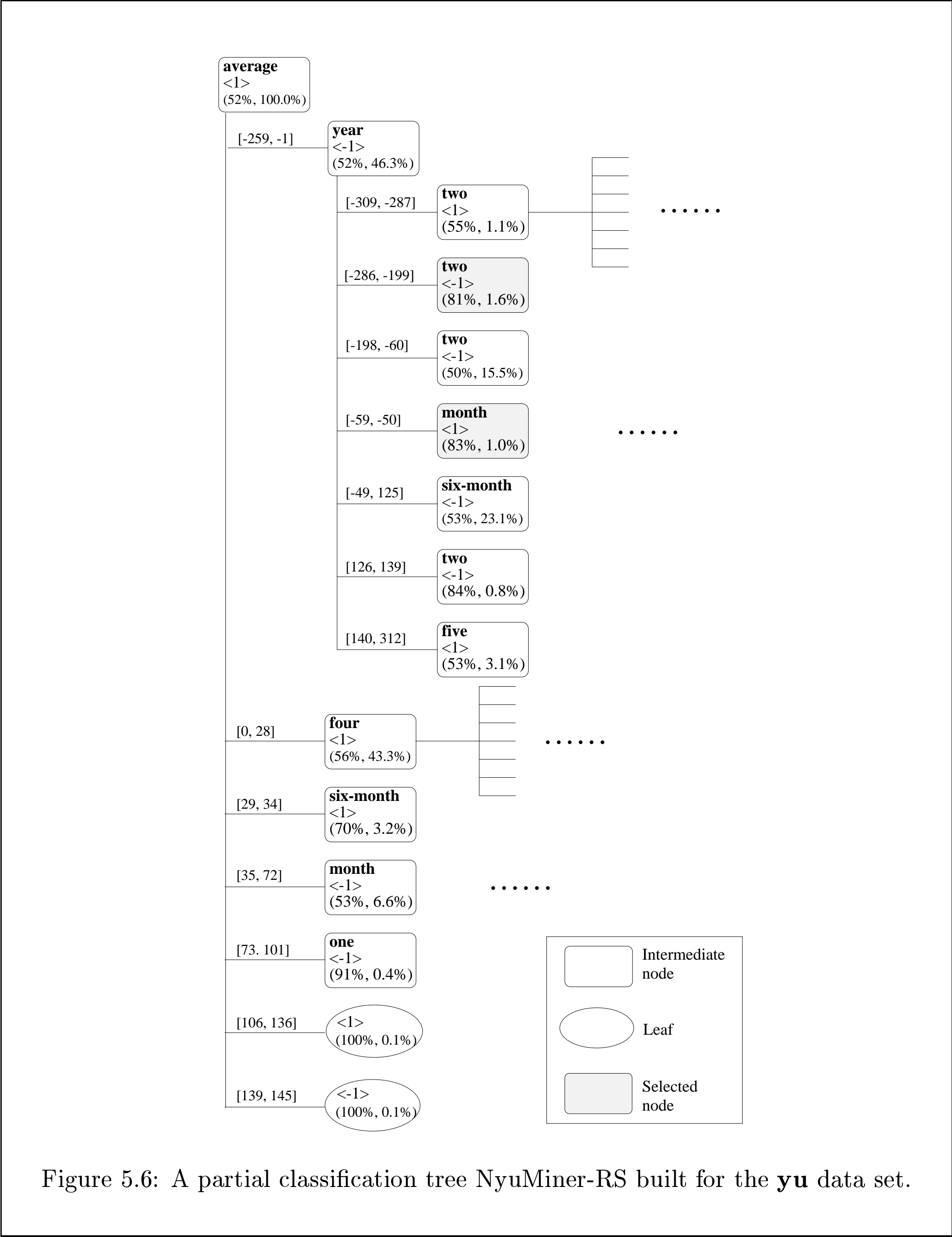


Figure 5.6: A partial classification tree NyuMiner-RS built for the **yu** data set.

S_{min} to be 1%. Only two rules are over both thresholds. They are

$$\textbf{average} \text{ in } [-2.59\%, -0.01\%] \;\&\; \textbf{year} \text{ in } [-2.86\%, -1.99\%] \Longrightarrow -1$$

and

$$\textbf{average} \text{ in } [-2.59\%, -0.01\%] \;\&\; \textbf{year} \text{ in } [-0.59\%, -0.50\%] \Longrightarrow 1.$$

The first rule applies to 94 days on the **yu** testing set; on 54 out these 94 days the rule predicts correctly. The second rule applies to 80 days on the testing set and the rule predicts correctly on 45 days. Combined, 99 out of 174 days on the testing set are correctly classified, with a accuracy of 56.9%.

The C_{min} (80%) and S_{min} (1%) we chose is not the only combination that would generate good results. In fact, it is not a combination that would generate the best results on the **yu** data set. Rather, this combination is appropriate because it successfully sifts out the rules we are looking for that have high accuracy and enough support. And this combination happens to produce rules that cover 100–200 days on the testing set for each pair of currencies. In order to be consistent, we used this combination for all five data sets. The number of rules that were selected and the number of days they cover for each data set are shown in table 5.6. Rules from each data set cover between 110 and 180 days for the 13 years in the testing set. This translates roughly to one trade per month. On these selected days, NyuMiner-RS's classification accuracy increases to 56.9% to 62.5%.

5.6.3 Making Money

So how do we make money if we were told about these rules in 1985? Suppose we use the simplest strategy:

1. For every pair of currencies, start with an amount of money in either currency;

Data Set	# of Rules Selected	# of Days Covered	Accuracy on Days Covered	1000 Currency Units in 13 Years				
				In First Currency	% Gain	In Second Currency	% Gain	Average % Gain
yu	2	174	56.9%	1128	12.8%	1025	2.5%	7.7%
du	2	112	62.5%	1062	6.2%	1029	2.9%	4.5%
yd	3	164	57.3%	1063	6.3%	1069	6.9%	6.6%
fu	5	159	57.9%	1044	4.4%	1124	12.4%	8.4%
up	4	135	60.6%	1112	11.2%	1081	8.1%	9.7%

Table 5.6: Money made in foreign exchange.

2. On the days that are covered by our rules, if the predicted movement is to our advantage (e.g. we hold Japanese Yens and the rule predicts that Yen goes up again U. S. Dollar tomorrow), we convert all our money to the other currency and convert back the next day;

3. Keep our positions on all other days.

Even with this simple strategy, we can be quite successful in the past 13 years (see Table 5.6).

Chapter 6

Parallel Classification Tree Algorithms

In this chapter, we explore data parallelism in classification tree algorithms. In Section 6.1, we describe how to explore data parallelism in cross validation and present the parallel NyuMiner-CV algorithm. In Section 6.2, we describe how to explore data parallelism in the windowing technique and the multiple incremental sampling technique and present Parallel C4.5 and the parallel NyuMiner-RS algorithm. We also give experimental results on PLinda implementations of these three parallel classification tree algorithms.

6.1 Parallelism in Cross Validation

CART and NyuMiner-CV use minimal cost complexity pruning with V-fold cross validation to find the best pruned tree. In a V-fold cross validation, V auxiliary trees have to be grown besides a main tree, making the total number of trees to grow $V+1$. These

$V+1$ trees are grown in exactly the same way but with $V+1$ different training sets. They are clearly candidates for data partitioning.

6.1.1 Parallel NyuMiner-CV

Parallel NyuMiner-CV builds the main tree and V auxiliary trees concurrently, each as a parallel task. The master divides the original training set into V subsets and produces V learning sets, each of size $(V-1)/V$. It then goes on to build a tree based on the whole training set—the main tree. The V auxiliary trees are to be built by the workers from the V learning sets produced by the master. Figure 6.1 (Figure 6.2, respectively) shows the pseudo-PLinda code of the master (worker).

To measure the performance of Parallel NyuMiner-CV, we ran the program on 1, 2, 3, 4, 5, and 6 machines. Because growing the main tree (including resubstitution error estimation) uses more data than each auxiliary tree and requires a lot more book-keeping, it takes significantly longer to build the main tree than an auxiliary tree. Our experience with the **yeast** and the **satimage** data sets shows that growing the main tree takes roughly the same time it takes to grow 4 auxiliary trees. Therefore, in our experiments, we run NyuMiner-CV with 4-fold, 8-fold, 12-fold, 16-fold, and 20-fold cross validation on 2, 3, 4, 5, and 6 machines, respectively. The main tree is grown by the master and each worker grows 4 auxiliary trees. Sequential running times are listed in Table 6.1. Figure 6.3 and figure 6.4 show the running time and speedup results on the **yeast** and the **satimage** data sets. Because we run NyuMiner-CV with 4-fold cross validation on 2 machines, the speedup for a 2 machine run is the 4-fold cross validation sequential running time divided by the 2 machine parallel running time. Speedups for 3, 4, 5, and 6 machines are calculated similarly.

In our experiments with the **yeast** data set, each parallel task takes approx-

```
xstart;
    partition training set into V subsets;
    for (i=0; i<V; i++) {
        pl_out("learning set", i, set[i]);
    }
xcommit;
xstart;
    build the main tree;
xcommit;
xstart;
    for (i=0; i<V; i++) {
        pl_in("alpha_list", ?j, ?alpha[i]);
    }
xcommit;
out(poison tasks);
pick the right alpha;
```

Figure 6.1: Pseudo-PLinda code of Parallel NyuMiner-CV master.

```
while (!done) {
    xstart;
    in("learning set", ?i, ?set);
    if !(poison task) then {
        build tree on set;
        pl_out("alpha_list", i, alpha);
    } else {
        done=1;
    }
    xcommit;
}
```

Figure 6.2: Pseudo-PLinda code of Parallel NyuMiner-CV worker.

imately 15 seconds with small variation (± 3 seconds). Each parallel task for the **satimage** data set takes approximately 120 seconds, also with small variation (± 20 seconds). Since the total number of tasks is 4 times the number of workers and all tasks are of similar size, each worker will mostly likely execute 4 tasks. Therefore, work load is normally balanced among workers.

V in V-fold CV	0	4	8	12	16	20
yeast	53	108	153	181	216	249
satimage	470	980	1499	1880	2302	2723

Table 6.1: Sequential running time (sec.) of NyuMiner-CV on the data sets **yeast** and **satimage**. ($V = 0$ means no cross validation.)

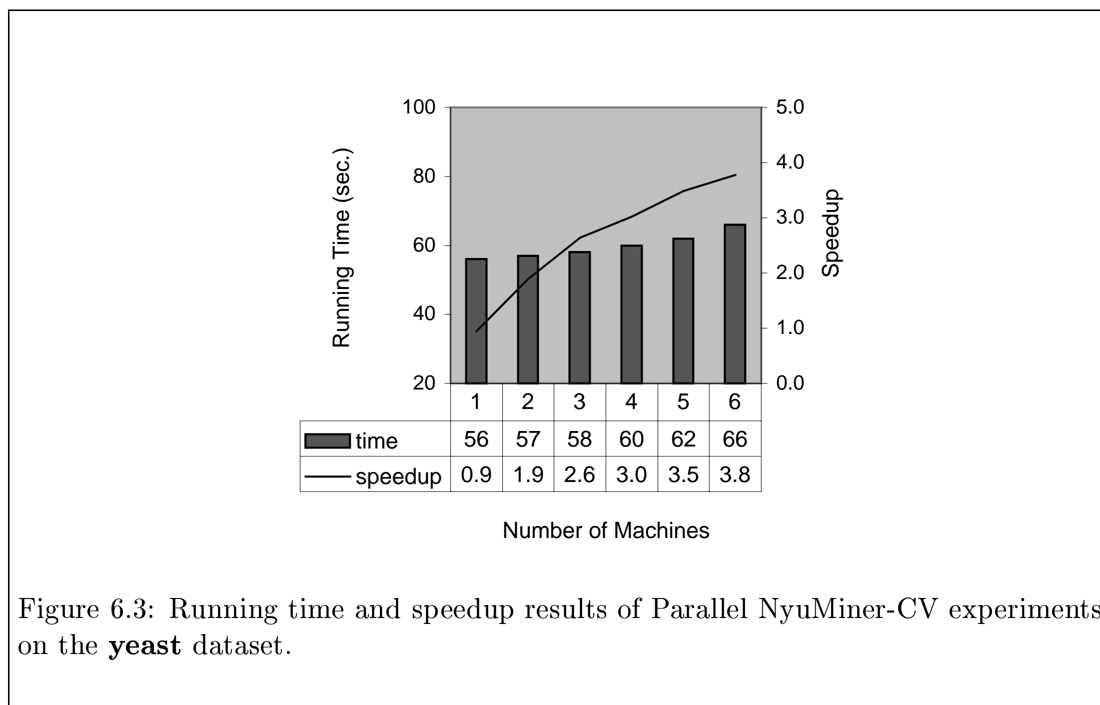


Figure 6.3: Running time and speedup results of Parallel NyuMiner-CV experiments on the **yeast** dataset.

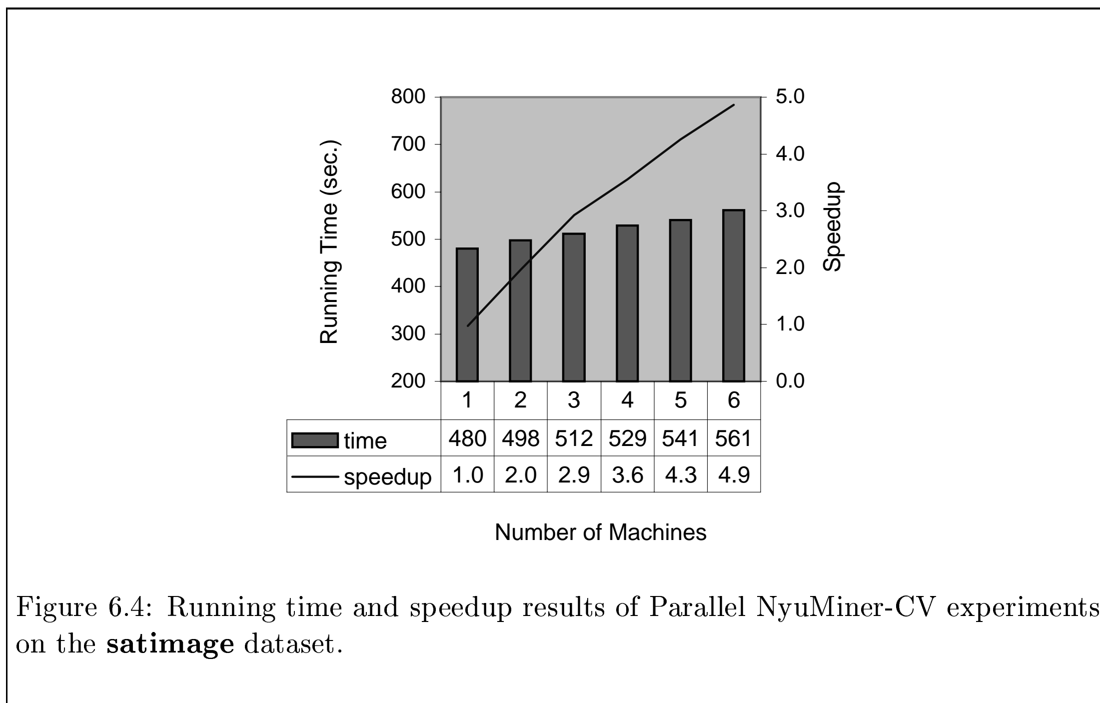


Figure 6.4: Running time and speedup results of Parallel NyuMiner-CV experiments on the **satimage** dataset.

6.2 Parallelism in Multiple Incremental Sampling

The multiple incremental sampling technique and the windowing technique require building trees from a small initial training set multiple times. The number of trees to grow is called the number of *trials* in C4.5. This is equivalent to building multiple trees from different initial training sets at the same time. This makes it natural to parallelize C4.5 and NyuMiner-RS using data partitioning.

6.2.1 Parallel C4.5

We implemented the windowing technique in parallel so that each PLinda worker builds a tree from a randomly sampled initial training set. We used two datasets to compare the running time of Parallel C4.5 to that of the original C4.5. The results are shown in Figure 6.5 and Figure 6.6. The sequential running times are shown in Table 6.2.

As in our parallel NyuMiner-CV experiments, parallel task grain-size is relatively large (approximately 10 seconds for the **smoking** data set and 200 seconds for the **letter** data set) and load-balancing is normally achieved.

Trials	1	2	4	6	8	10
smoking	8.8	15.7	31.7	46.4	60.4	74.0
letter	205	407	866	1284	1689	2165

Table 6.2: Sequential running time (sec.) of C4.5 on the data sets **smoking** and **letter**.

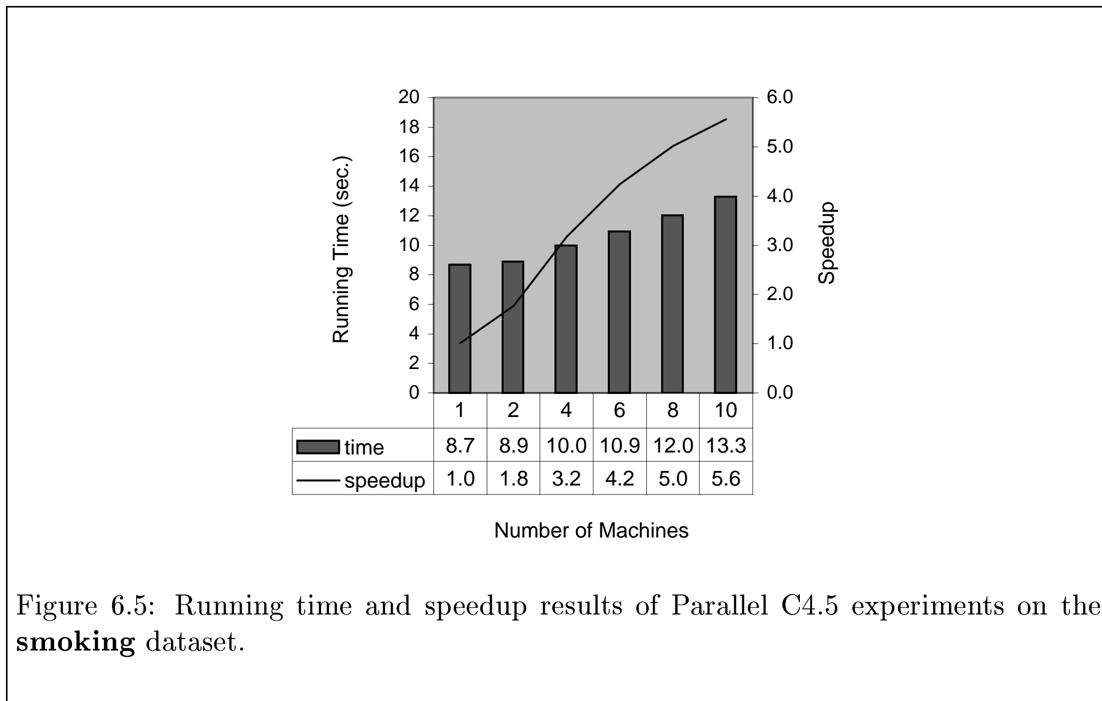


Figure 6.5: Running time and speedup results of Parallel C4.5 experiments on the **smoking** dataset.

Note that for the **letter** dataset, PC4.5 sometimes achieve super-linear speedup against the sequential program. The reason for this is as follows. C4.5 is memory intensive, especially when the windowing technique is used. The **letter** dataset is relatively large with 20000 cases, 17 attributes and 26 classes. Building one tree for

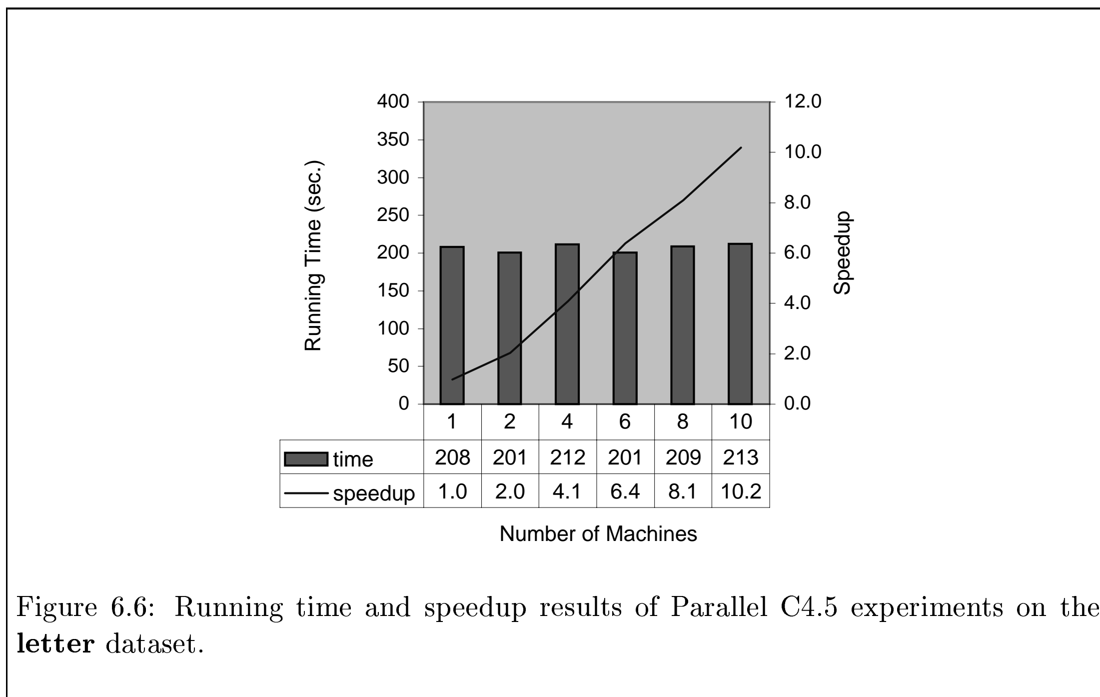


Figure 6.6: Running time and speedup results of Parallel C4.5 experiments on the **letter** dataset.

the **letter** data set takes about 14 MB memory space. The machines we use all have 32 MB RAM. Because all intermediate trees have to be kept until the end of computation, there is going to be some paging during the sequential execution if more than 2 trials are used. In PC4.5, only one tree grows on each machine and thus paging is eliminated. This explains the super-linear speedup.

6.2.2 Parallel NyuMiner-RS

Parallel NyuMiner-RS implements multiple incremental sampling and rule selection in parallel. Each tree from a different initial training set is built in a different process. We used the datasets **yeast** and **satimage** to compare the performance of Parallel NyuMiner-RS to the sequential version. The sequential running times are shown in Table 6.3. The experiment results are show in Figure 6.7 and Figure 6.8. For the **yeast**

data set, the parallel task grain-size is approximately 50 seconds and for the **satimage** data set approximately 600 seconds. Again, because the variation in grain-sizes is small, work load is normally balanced.

Number of Trees	1	2	4	6	8	10
yeast	51	104	166	214	324	391
satimage	573	1148	2414	3231	4446	5825

Table 6.3: Sequential running time (sec.) of NyuMiner-RS on the data sets **yeast** and **satimage**.

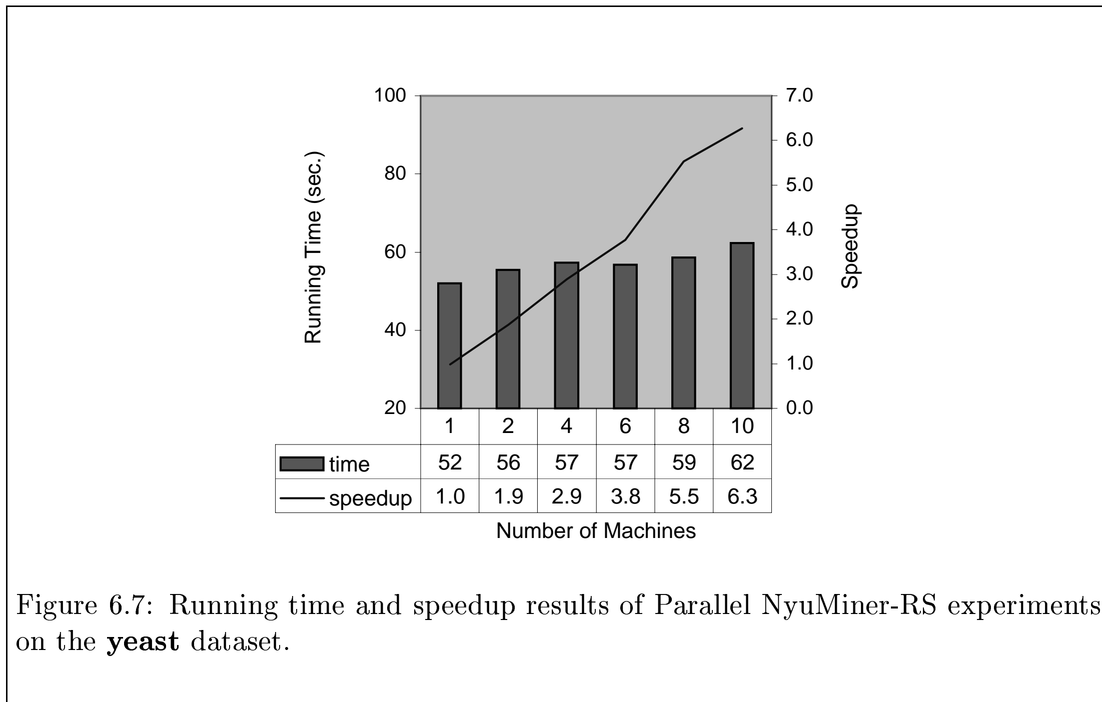
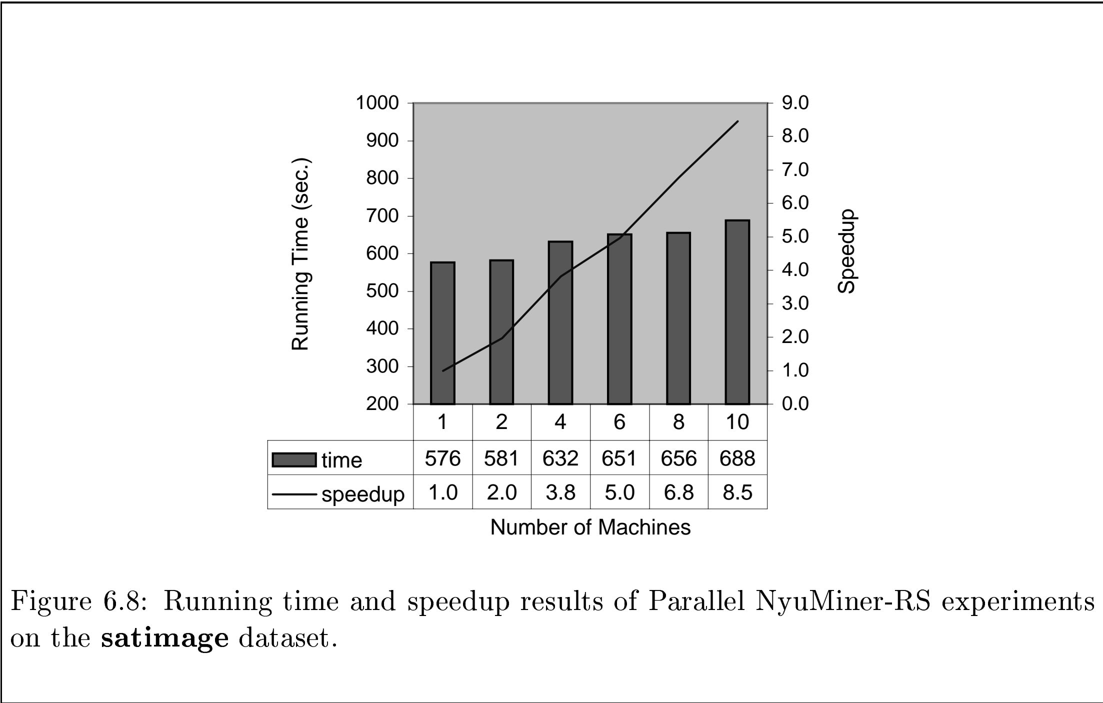


Figure 6.7: Running time and speedup results of Parallel NyuMiner-RS experiments on the **yeast** dataset.



	1	2	4	6	8	10
time	576	581	632	651	656	688
speedup	1.0	2.0	3.8	5.0	6.8	8.5

Figure 6.8: Running time and speedup results of Parallel NyuMiner-RS experiments on the **satimage** dataset.

6.3 Summary

As demonstrated by the three parallel classification tree algorithms presented in the chapter, it is easy to explore data parallelism in data mining applications. Good performance is achieved using the same software architecture as with task partitioning programs. Because of higher main memory efficiency, sometimes super-linear speedup can be achieved.

Chapter 7

Software Architecture

Throughout the previous chapters, we have provided a set of PLinda templates for writing parallel data mining programs. These templates not only help data mining programmers to understand the parallelism in data mining algorithms, but also give them good starting points for their programs. The effectiveness of these templates are already demonstrated in our own implementations of several data mining applications. In this chapter, we discuss the fault-tolerance of PLinda programs developed from these templates. We also describe how to run PLinda programs on a network of workstations.

7.1 Fault-Tolerance

The distributed computing environment that exists in many organizations usually consists of a number of networked workstations. When using multiple workstations on a network as a virtual parallel machine, any network fault or individual workstation failure will "break" the virtual machine. To address this problem, PLinda provides a fault-tolerant execution environment.

7.1.1 Parallel Virtual Machine in PLinda

The current PLinda prototype allows the user to use networks of workstations as one fault-tolerant parallel virtual machine whose processors are only idle workstations. However, the current implementation assumes that the user runs the server and user interface processes on machines dedicated to the parallel application. That is, the server and user interface processes are not migrated during execution.

The parallel virtual machine consists of a pool of processors which are running or failed. A PLinda daemon is created on each workstation. Idle workstations are treated as running processors and busy or failed workstations are treated as failed processors. Only running processors participate in computation. When a running processor fails (because of either owner activity or a real processor failure), its daemon immediately destroys all the running client processes on the machine. Then, the server is either informed of the simulated failure or it detects the real failure. The retreated processes are then restarted on other running processors. In this way, PLinda programs can run in a manner that is fault-tolerant and does not disturb the owners of the workstations at all (we have tested this empirically).

7.1.2 PLinda's Fault-Tolerance Guarantee

PLinda's transaction mechanism *guarantees that a completed PLinda computation, with or without failures, achieves the same final state as a failure-free execution of the associated Linda program*, where the associated Linda program is identical to the PLinda program but without the transaction and recovery statements [49].

7.1.3 PLinda Data Mining Programs Are Fault-Tolerant

Our data mining programs written in PLinda has this null hypothesis: *there is no special accommodation for existence of failure*. The correctness of our programs is assured by the PLinda fault-tolerance guarantee.

7.2 Running PLinda Programs

All of our parallel data mining programs run as normal PLinda programs. This section describes how to start the PLinda environment on a network of workstations, how to start a PLinda program, and how to observe and control execution behaviors of a running PLinda program.

7.2.1 How to Start the PLinda User Interface

The PLinda user interface can be started in the `~/plinda` directory by typing `plm`. The menu panel of the user interface will pop up (see Figure 7.1). The PLinda kernel will create some files and subdirectories in the directory where it is run from. Therefore, it is important that the system be run from the directory `~/plinda`.

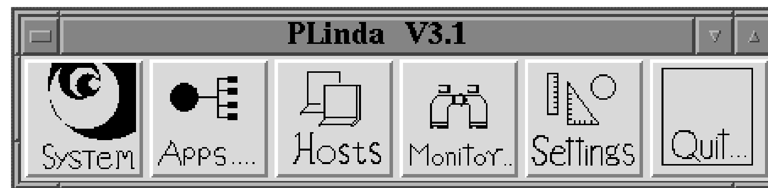


Figure 7.1: A snapshot of PLinda runtime menu window.

7.2.2 How to Start the PLinda Server

Before we start the PLinda server, we need to set some configuration parameters for the server. These parameters can be defined in the `~/.plindarc` file but they can be changed in the "Settings" window (see Figure 7.2). After these parameters are set, we can use the "System" window to start the server (see Figure 7.3). This windows allows you to specify the name and type of the machine on which you want to start the server. The default values are read from the file `~/.plindrc`. You can also change the working directory and display host, although normally they need not to be changed. Check the radio button "Boot" and then click "Apply" to start the server.

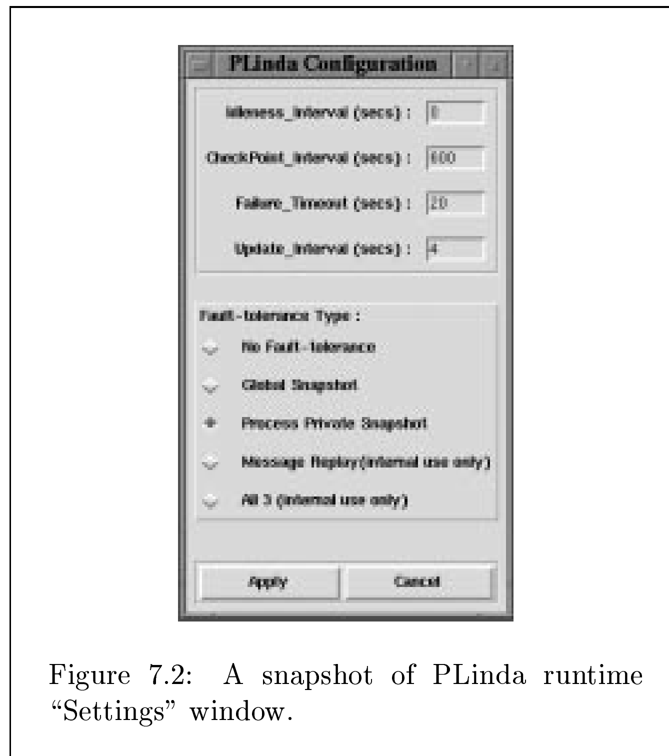


Figure 7.2: A snapshot of PLinda runtime "Settings" window.

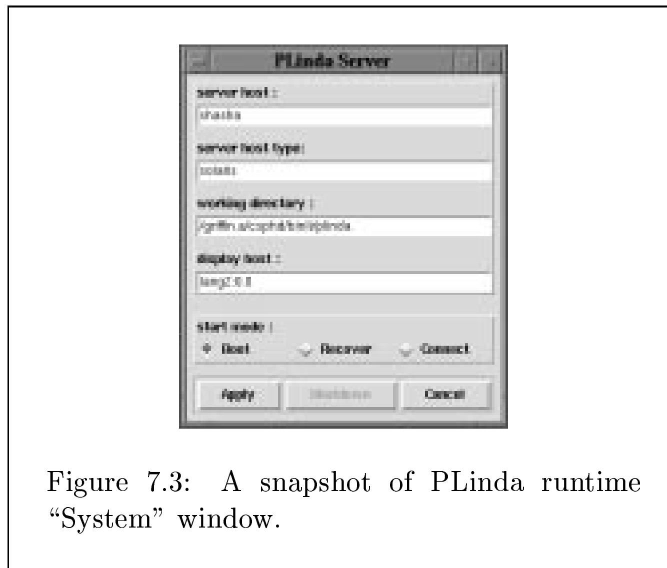


Figure 7.3: A snapshot of PLinda runtime "System" window.

7.2.3 How to Add Hosts

The user must supply the file `~/.plinda.hosts` with the names of workstations, their architecture types, and the user account name for each machine he will use. Click the "Hosts" button on the menu bar and a window with host information will pop up (see Figure 7.4). It lists all the machines in the `~/.plinda.hosts` file. Check the radio button besides a host name to select the host and click the "Add" button to add it to the PLinda host pool. Multiple hosts can be selected and added together. To delete a machine from the PLinda host pool, select the host and click on the "Delete" button.

7.2.4 How to Select a PLinda Program to Run

The environment variable $PLINDA_HOME points to the PLinda installation directory. All PLinda executables should be located in the directory $PLINDA_HOME/lib/ $ARCHTYPE (where $ARCHTYPE may be `sunos, solaris, linux,` etc.). Ones to be started from the interface should have a `.exe` extension. The PLinda kernel will use

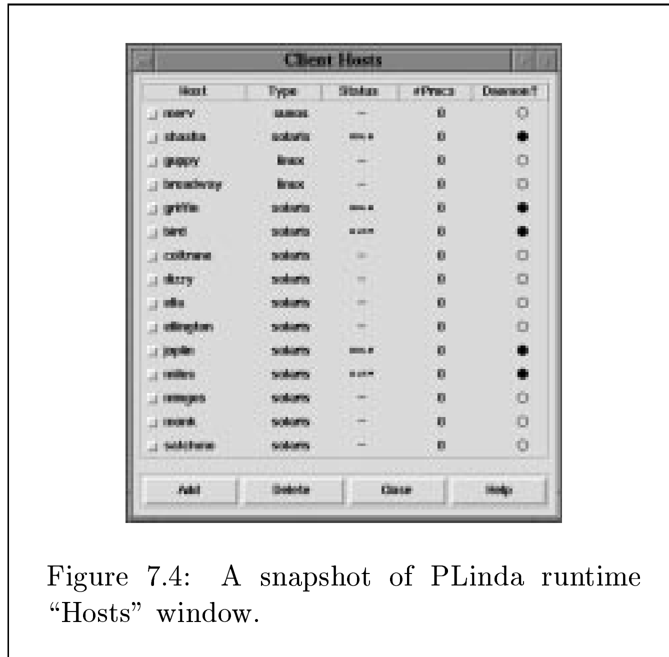


Figure 7.4: A snapshot of PLinda runtime "Hosts" window.

the Unix `rsh` command to run processes on remote machines. Therefore, it is necessary to be able to access those machines from within an executable. To facilitate this the user needs to have an `.rhosts` file on each remote machine listed in the `.plinda.hosts` file. The file must define all machines and alternate login names from which the PLinda user interface will be run.

To select a PLinda program to rum, click on the "Apps" button on the menu bar. This brings up regular file selection interface that starts with the directory `~/plinda/lib` (see Figure 7.5). You can go to the directory (e.g. `sunos, solaris`, or `linux`) which has the right executable and start the program by clicking on the "Select" button. You must compile source code for each of the types of architectures you plan to use in your virtual machine host pool. The runtime system assumes you have compiled code for all those types of machines.

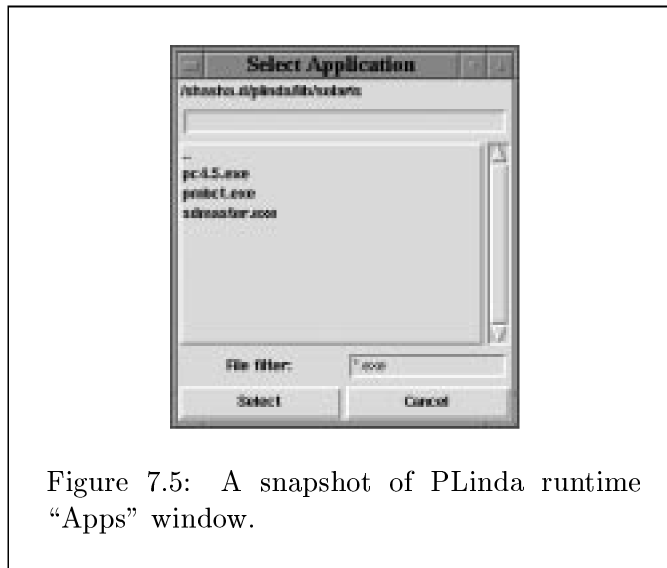


Figure 7.5: A snapshot of PLinda runtime "Apps" window.

7.2.5 How to Observe and Control Execution Behaviors

When you start an application, two additional windows should pop up. The first one is the "Process Watch" window (you can also get it by clicking on the "Monitor" button on the menu bar) (see Figure 7.6). This window shows you all running PLinda processes with process id (pid), the name of the host the process is on, etc. The "Process Watch" window is refreshed every `Update_Interval` seconds, where `Update_Interval` can be defined either in `plinda.defaults` or by changing the value in the dialog box that pops up when you click the "Settings" button on the menu bar. The second window is an `xterm` used as an I/O shell for the PLinda master. Figure 7.7 shows the complete screen when a PLinda program is running.

One important piece of information about a process is its status. When a PLinda process is spawned, its status is "DISPATCHED". When a PLinda process is waiting for a tuple, its status is "BLOCKED"; most of the time its status is "RUNNING". When a process goes from "RUNNING" to "BLOCKED", there is a middle stage called

Process Watch Window

pid	status	host	line#	file	exec
10	BLOCKED	shasha	202	master.C	pmbct.exe
16	BLOCKED	shasha	83	worker.C	pmbct_worker
14	BLOCKED	griffin	83	worker.C	pmbct_worker
17	READY	griffin	83	worker.C	pmbct_worker
13	BLOCKED	bird	83	worker.C	pmbct_worker
12	RUNNING	joplin	83	worker.C	pmbct_worker
15	RUNNING	joplin	83	worker.C	pmbct_worker
18	RUNNING	joplin	83	worker.C	pmbct_worker
11	BLOCKED	miles	83	worker.C	pmbct_worker

Kill | Migrate | Suspend | Resume | View | Trace | Close | Help

Figure 7.6: A snapshot of PLinda runtime "Monitor" window.

Figure 7.7: A screen snapshot of PLinda environment.

"READY". But "READY" holds for a very short period of time and will be rarely seen in the "Process Watch" window. When the PLinda server re-spawns a failed process, the status of the process is "FAILURE_HANDLED".

The user can "Kill", "Migrate", "Suspend", or "Resume" processes by clicking on appropriate buttons in the "Process Watch" window. These functions are mostly used to test the fault-tolerance of a PLinda program.

7.3 Software Available on the WWW

All of our data mining software are freely available, including source code. For the PLinda software, please download from

http://merv.cs.nyu.edu:8001/~binli/plinda/.

For all data mining software, please download from

http://merv.cs.nyu.edu:8001/~binli/datamining/.

Installation instructions and user guides are on the web pages as well.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Data mining is becoming more and more of a interest for "ordinary" people (who do not have easy access to super-computers). In this thesis, we have argued that to make data mining practical for them, data mining algorithms have to be efficient and data mining programs should not require dedicated hardware to run. On these fronts, we can conclude from this thesis that:

- Parallelization is a viable solution to efficient data mining;
- Task parallelism and data parallelism exist in most data mining algorithms;
- The E-dag framework for task partitioning can be used to easily and efficiently parallelize most data mining applications;
- Classification tree algorithms can readily take advantages of data parallelism.

We can also conclude from this thesis that networks of workstations are a suitable platform for efficient parallel data mining. PLinda is a suitable computing system for

parallel data mining on NOW. PLinda software being able to automatically utilize idle workstations makes parallel data mining on NOW virtually free. Parallelization frameworks coupled with PLinda templates for data mining make free efficient data mining realistic.

In software practice, normal programmers seldom write parallel programs, although they usually know the parallelism in their applications. The computer science community has failed to provide them with language constructs and programming environments that support concurrency and are easy to use. In some sense, this thesis work fills the gap between data mining programmers and a parallel computing system. This gap needs to be filled for other application programmers and other parallel computing paradigms. To achieve this, frameworks like ours need to be developed.

We presented a new classification tree algorithm—NyuMiner. NyuMiner generally achieves better classification accuracy than other classification tree algorithms. Because of the usually different tree structures generated by NyuMiner, it can be used to complement other classifiers to obtain higher classifying confidence than one classifier used alone. NyuMiner's advantages of being able to find finer ranges in numerical attributes can be very useful in some classification applications.

8.2 Future Work

Our overall plan for future research in parallel data mining and data mining in general are outlined below.

1. The E-dag framework for task partitioning and the data partitioning technique can be applied to many other data mining applications, such as market basket analysis, frequent episode discovery, etc. Many applications fit the pattern lattice

paradigm and thus are suitable for the E-dag framework. Data partitioning is especially beneficial to applications that require a very large amount of data.

2. Further theoretical study and experimenting on parallel E-dag and E-tree traversals should reveal an effective cost model for choosing parallelization strategies for different applications.

3. NyuMiner is a promising algorithm to obtain better classification trees. To make it more database-friendly, we need to develop additional modules for it to interface directly with database systems. Visualizing resultant trees and making it possible to consult the trees interactively will make NyuMiner easier to use.

4. A challenge for data mining in general is how to develop a data mining model so that growth and change in data requires minimal additional mining. We need to tackle this challenge in the context of free parallel data mining.

Bibliography

[1] T. Brown, K. Jeong, B. Li, D. Shasha, S. Talla, and P. Wyckoff. PLinda User Manual. Technical Report 729, Department of Computer Science, New York University, December, 1996.

[2] Bin Li, Dennis Shasha, and Jason T. L. Wang. A Framework for Biological Pattern Discovery on Networks of Workstations. Chapter 11, *Pattern Discovery in Biological Data: Tools, Techniques, and Applications*, ed. Jason T. L. Wang, Bruce A. Shapiro, and Dennis Shasha, Oxford University Press, 1997.

[3] Bin Li and Dennis Shasha. Free Parallel Data Mining. *SIGMOD '98*, Seattle, WA, June 1998.

[4] Bin Li, Chin-Yuan Cheng, and Dennis Shasha. NyuMiner: Classification Trees by Optimal Sub-K-ary Splits. *The Fourth International Conference on Knowledge Discovery and Data Mining*, New York City, August 1998, submitted.

[5] Robert D. Blumofe. An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[6] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations. In *Proceedings of the USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems*, pages 133-147, Anaheim, California, January 6-10, 1997.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM 3 User's Guide and Reference Manual. Oak Ridge National Laboratory, Oak Ridge, TN 37831, 1994.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[9] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[10] P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 94 Usenix Conference*, pp. 115-131, January 1994.

[11] C.-P. Wen, S. Chakrabarti, E. Deprit, Chih-Po Wen, A. Krishnamurthy, and K. Yelick. Runtime Support for Portable Distributed Data Structures. Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers, May 1995.

[12] Scott R. Kohn and Scott B. Baden. A Robust Parallel Programming Model for Dynamic Non-Uniform Scientific Computations. In *Proc. 1994 Scalable High Performance Computing Conference*, Knoxville, Tennessee, May 23-35, 1994.

[13] R. Parsons and D. Quinlan. A++/P++ Array Classes for Architecture Independent Finite Difference Computations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, Sunriver, Oregon, April 1994.

[14] V. Kumar and V. N. Rao. Parallel Depth First Search on Multiprocessors. Part I: Implementation. *International Journal of Parallel Programming*, vol 16, no. 2, pages 479-499, December 1987.

[15] V. Kumar and V. N. Rao. Parallel Depth First Search on Multiprocessors. Part II: Analysis. *International Journal of Parallel Programming*, vol 16, no. 2, pages 501-519, December 1987.

[16] V. Kumar, V. N. Rao and K. Ramesh. Parallel Best-First Search of State-Space Graphs: A Summary of Results (1988). In *Proceedings of the 1988 National Conf. on Artificial Intelligence (AAAI-88)*, August 1988.

[17] V. A. Saletore and L. V. Kale. Consistent Linear Speedups to a First Solution in Parallel State-Space Search. In *Proceedings of the 1990 National Conf. on Artificial Intelligence (AAAI-90)*, July 1990.

[18] R. Agrawal, T. Imielinski, and A. Swami. Mining Association Rules Between Sets of Items in Large Databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., May 26-28, 1993.

[19] R. Agrawal and R. Srikant. Fast Algorithm for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Databases*, Santiago, Chile, August 29-September 1, 1994.

[20] B. Anderson and D. Shasha. Persistent Linda: Linda + transactions + query processing. In J. P. Banatre and D. Le Metayer, editors, *Research Directions in High-Level Parallel Programming Languages*, volume 574, pages 93–109. Springer-Verlag Lecture Notes in Computer Science, June 1992.

[21] A. Baratloo, P. Dasgupta, and Z. Kedem. CALYPSO: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms. In *Proceedings of the 4th IEEE Intl. Symp. on High Performance Distributed Systems*, 1995.

[22] Carel A. van den Berg and Martin D. Kersten. An Analysis of a Dynamic Query Optimization Scheme for Different Data Distributions. In J. Freytag, D. Maier, and G. Vossen, editors, *Advances in Query Processing*, pages 449–470. Morgan-Kaufmann, 1994.

[23] W. Buntine and R. Caruana. Introduction to IND Version 2.1 and Recursive Partitioning. NASA Ames Research Center TR# FIA-91-28, October, 1991.

[24] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth International Group, Belmont, CA, 1984.

[25] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recoverability in Database Systems*, Addison-Wesley Publishing Company, 1987.

[26] Robert Bjornson. Linda on Distributed Memory Multiprocessors. Ph.D. Thesis, Department of Computer Science, Yale University, 1992.

[27] N. Carriero. Implementing Tuple Space Machines. Ph.D. Thesis, Department of Computer Science, Yale University, 1987.

[28] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October, 1991.

[29] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[30] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs, A First Course*, MIT Press, 1991.

[31] Philip K. Chan and Salvatore J. Stolfo. Experiments in Multistrategy Learning by Meta-Learning. In *Proceedings of the second international conference on information and knowledge management*, pages 314–323, Washington, D.C., 1993.

[32] Philip K. Chan and Salvatore J. Stolfo. Towards Scalable and Parallel Inductive Learning: A Case Study in Splice Junction Prediction. Columbia University, CUCS-032-94, 1994.

[33] T. Elomaa and J. Rousu. Finding Optimal Multi-Splits for Numerical Attributes in Decision Tree Learning. NeuroCOLT Technical Report Series, NC-TR-96-041, Department of Computer Science, University of Helsinki, Finland, March 1996.

[34] U. M. Fayyad and K. B. Irani. The Attribute Selection Problem in Decision Tree Generation. In *Proceedings of AAAI-92*, pages 104–110, 1992.

[35] Fayyad, Piatetsky-Shapiro, Smyth, and Uthurusamy (eds.). *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press, 1995.

[36] U. Fayyad and K. Irani. On the Handling of Continuous-valued Attributes in Decision Tree Generation. *Machine Learning*, 8: 87–102, 1992.

[37] U. Fayyad and K. Irani. Multi-interval Discretization of Continuous-valued Attributes for Classification Learning. In *Proc. Thirteenth International Joint Conference on Artificial Intelligence*, pages 1022–1027, San Mateo, CA: Morgan Kaufmann, 1993.

[38] Jerome H. Friedman. Data Mining and Statistics: What's the Connection? URL: http://stat.stanford.edu/~jhf/dm-stat.ps.Z.

[39] Jerome H. Friedman and Nicholas I. Fisher. Bump Hunting in High-Dimensional Data. URL: ftp://stat.stanford.edu/pub/friedman/prim.ps.Z.

[40] K. A. Frenkel. The Human Genome Project and Informatics. *Communications of the ACM*, 34(11):41–51, November 1991.

[41] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*, Morgan Kaufman Publishers, Inc., 1993.

[42] Jiawei Han and Yongjian Fu. Discovery of multiple-level association rules from large databases. In *Proceedings of the 21th International Conference on Very Large Databases*, Zurich, Switzerland, September, 1995.

[43] M. Holsheimer and M. L. Kersten. Architectural support for data mining. Centrum voor Wiskunde en Informatica (CWI), CS-R9429, Netherlands, 1994.

[44] L. C. K. Hui. Color set size problem with applications to string matching. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Combinatorial Pattern Matching, Lecture Notes in Computer Scienc*, Volume 644, pages 230–243, Springer-Verlag, 1992.

[45] L. Hunter, editor. *Artificial Intelligence and Molecular Biology.* AAAI Press/The MIT Press, Menlo Park, CA, 1993.

[46] Ian Foster. Task Parallelism and High-Performance Languages. *IEEE Parallel and Distributed Technology Systems and Applications*, pages 27–36, 1994.

[47] Karpjoo Jeong and Dennis Shasha. Persistent Linda 2: a transactioncheckpointing approach to fault-tolerant Linda. In *Proceedings of the 13th Symposium on Fault-Tolerant Distributed Systems*, IEEE, October, 1994.

[48] Karpjoo Jeong. Fault-tolerant Parallel Processing Combining Linda, Checkpointing, and Transactions. Ph.D. Thesis, Courant Institute of Mathmatical Sciences, New York University, January, 1996.

[49] K. Jeong, D. Shasha, S. Talla, and P. Wyckoff. An Approach to Fault Tolerant Parallel Processing on Intermittently Idle, Heterogeneous Workstations. *The Twenty-Seventh International Symposium on Fault-Tolerant Computing*, pages 11–20, Seattle, Washington, June 24–27, 1997.

[50] George H. John. *Enhancements to the Data Mining Process.* Ph.D. Thesis, Department of Computer Science, Stanford University, March 1997.

[51] N. Kamel, M. Delobel, T. G. Marr, R. Robbins, J. Thierry-Mieg, and A. Tsugita. Data and Knowledge Bases for Genome Mapping: What Lies Ahead? Panel Persentation in the 17th International Conference on Very Large Data Bases, Barcelona, Spain, September 1991.

[52] D. Kaminsky. Adaptive Parallelism with Piranha. Ph.D. Thesis, Department of Computer Science, Yale University, 1994.

[53] Martin L. Kersten. Goblin: A DBPL Designed for Advanced Database Applications. In *2cn International Conference on Database and Expert Systems Applications, DEXA'91*, Berlin, Germany, August 1991.

[54] G. M. Landau and U. Vishkin. Fast Parallel and Serial Approximate String Matching. *Journal of Algorithms*, Volume 10(2), pages 157–169.

[55] J. Leichter. Shared Tuple Memories: Shared Memories, Buses and LAN's—Linda Implementation Across the Spectrum of Connectivity. Ph.D. Thesis, Department of Computer Science, Yale University, 1989.

[56] R. J. Lipton, T. G. Marr, and J. D. Welsh. Computational Approaches to Discovering Semantics in Molecular Biology. In *Proceedings of the IEEE*, 77(7):1056–1060, July 1989.

[57] M. Litzkow, M. Livny, and M. W. Mutka. Condor—a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June, 1988.

[58] J. Makulowich. *Data Mining Developments Gain Attention.* Washington Technology (a biweekly supplement to Washington Post), Oct. 23, 1997.

[59] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A Fast Scalable Classifier for Data Mining. *EDBT-96*, Avignon, France, March 1996.

[60] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, Volume 23, pages 262–272, 1976.

[61] Andreas Mueller. Fast Sequential and Parallel Algorithms for Association Rule Mining: A Comparison. Dept. of Computer Science, Univ. of Maryland, CS-TR-3515, August, 1995.

[62] Jong Soo Park, Mink-Syan Chen, and Philip S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 175–186, San Jose, CA, 1995.

[63] J. R. Quinlan. Induction of decision trees. *Machine Learning*, Volume 1, pages 81–106, 1986.

[64] J. R. Quinlan. Decision Trees and Multi-valued attributes. In J. E. Hayes, D. Michie, J. Richards, editors, *Machine Intelligence*, Volume 11, pages 305–318, Oxford University Press, 1988.

[65] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[66] Ashok Sarasere, Edward Omiecinsky, and Shamkant Navathe. An Efficietn Algorithm for Mining Association Rules in Large Databases. In *Proceedings of the 21st International Conference on Very Large Databases*, Zurich, Switzerland, September, 1995.

[67] Ramakrishnan Srikant and R. Agrawal. Mining Generalized Association Rules. In *Proceedings of the 21th International Conference on Very Large Databases*, Zurich, Switzerland, September, 1995.

[68] J. T. L. Wang, G.-W. Chirn, T. G. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial Pattern Discovery for Scientific Data: Some Preliminary

Results. In *Proceedings of the 1994 ACM SIGMOD International Conference on the Management of Data*, Minneapolis, Minnesota, 1994.

[69] J. T. L. Wang, G.-W. Chirn, T. G. Marr, B. A. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 115–125, Minneapolis, Minnesota, May 1994.