Parallel Inductive Logic in Data Mining

by

Yu Wang

A thesis submitted to the
Department of Computing and Information Science
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
January 2000

Abstract

Data-mining is the process of automatic extraction of novel, useful and understandable patterns from very large databases. High-performance, scalable, and parallel computing algorithms are crucial in data mining as datasets grow inexorably in size and complexity. Inductive logic is a research area in the intersection of machine learning and logic programming, which has been recently applied to data mining. Inductive logic studies learning from examples, within the framework provided by clausal logic. It provides a uniform and very expressive means of representation: All examples, background knowledge as well as the induced theory are expressed in first-order logic. However, such an expressive representation is often computationally expensive. This thesis first presents the background for parallel data mining, the BSP model, and inductive logic programming. Based on the study, this thesis gives an approach to parallel inductive logic in data mining that solves the potential performance problem. Both parallel algorithm and cost analysis are provided. This approach is applied to a number of problems and it shows a super-linear speedup. To justify this analysis, I implemented a parallel version of a core ILP system – Progol – in C with the support of the BSP parallel model. Three test cases are provided and a double speedup phenomenon is observed on all these datasets and on two different parallel computers.

Acknowledgments

I would like to thank my supervisor, Dr. David Skillicorn, for his guide, encouragement and support throughout the course of the thesis. My gratitude also extends to Ken Whelan for his sincere help on my thesis writing. The CISC students, faculty and staff at Queen's were always helpful and a great pleasure to work with.

I would also like to express my gratitude to all my friends in Kingston, for their always-sincere support to my living and study abroad.

Finally, I would like to specially thank my wife – Cao Qun – who gave me her deep love and support during my study, which I has relied on during my entire stay here.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Basis of this thesis. This thesis shows a parallel data-mining algorithm that can be applied to large database mining using inductive logic programming. The central hypothesis of this thesis is that it is necessary and feasible to adopt parallel algorithms in the data mining process. I show that parallelism can be efficiently applied to inductive logic programming (ILP). The powerful knowledge representation and excellent integration with background knowledge of ILP has shown a great value among data-mining algorithms.

What is data mining? The field of data mining is concerned with the theory and processes involved in the representation and efficient extraction of interesting patterns or concepts from very large databases. Most of these concepts are implicit in the database records. Data mining is an interdisciplinary field merging ideas from statistics, machine learning, databases, and high-performance computing.

What is ILP and its role in data mining. Inductive Logic Programming is a relatively new machine learning technique adopted in the data-mining research area. Many researchers have turned to ILP only in the last 5 to 10 years [7]. It is defined as the intersection of machine learning and logic programming, and has grown to become a substantial sub-area of both of them [24]. The success of the subject lies partly in the choice of the core representation language of logic programs. The syntax of logic programs provides modular blocks which, when added or removed, generalize or specialize the program. ILP provides a uniform and very expressive means of representation: All examples, background knowledge, and the induced theory are expressed in first-order logic. Due to this uniform representation, the use of background knowledge fits very well within a logical approach towards machine learning. Theory and background knowledge are of the same form; they are just derived from different sources: theory comes from inductive learning while the background knowledge is provided by the user of the system [6]. Previous experiences [7] showed that some domain knowledge can be best expressed in a first-order logic, or a variant of first-order logic. The use of such domain knowledge is crucial in certain data-mining systems, such as learning drug structure-activity rules [33], because it is essential for achieving intelligent behavior. ILP inherits well-established theories, algorithms and tools from computational logic. Many inductive logic programming systems benefit from using the results of computational logic. There is already a wide range of data-mining applications using ILP algorithms.

Problem with inductive logic in data mining. There exist workable sequential algorithms for data mining, e.g. neural networks [27], association rules [1], decision trees [16], and inductive logic programming [7] that have already been applied to a

wide range of real-world applications. However, exploring useful information from a huge amount of data will require efficient parallel algorithms running on high-performance computing systems. The most obvious (and most compelling) argument for parallelism revolves around database size. The databases used for data mining are typically extremely large, often containing the details of the entire history of a company's standard transactional databases. As these databases grow past hundreds of gigabytes towards a terabyte or more, it becomes nearly impossible to process them on a single sequential machine, for both time and space reasons: no more than a fraction of the database can be kept in main memory at any given time, and the amount of local disk storage and bandwidth needed to keep the sequential CPU supplied with data is enormous. Additionally, with an algorithm that requires many complete passes over the database, which is the case in most ILP algorithms, the actual running time required to complete the algorithm becomes excessive. Because of the use of a more expressive representation, inductive logic programming techniques are often computationally more expensive than their propositional counterparts. This efficiency issue becomes more severe when the dataset is very large. Furthermore, many ILP algorithms have to go through the full dataset many times to get a successful induced concept set. Such an approach seems impractical to solve real-world data-mining jobs. So how to make these ILP algorithms work more effectively and efficiently has become an interesting research topic.

Contribution of this thesis. In this thesis I study the use of inductive logic to generate concepts from very big datasets in parallel. I use p processors to do the data-mining job, each on a subset of the full dataset. A set of concepts is generated from disjoint subsets of the full dataset used for mining. The distributed concept sets

are exchanged and evaluated before merging the valid ones into the final concept set. The final set of concepts is free of conflicts and same as the set of rules developed from the full dataset. In this way the disk I/O access cost for each processor is reduced by a factor of $1/p$.

The algorithm works in this way. First it divides the entire dataset and allocates each subset of data to a processor. Then each processor executes the same sequential ILP algorithm to find its locally-correct concepts. At the end of one step, all these processors exchange their discoveries and evaluate the induced concepts generated in this step. When each processor has collected all the feedback from other processors, it can decide if its locally-correct concepts are globally-correct. If so, it will inform all other processors to add this valid concept to the final concept set and remove the redundant examples covered by this concept. This completes one big step. This loop will continue until all the positive examples are covered by induced concepts.

Since each processor learns concepts independently on its subset, there are some issues that I will explore in this thesis:

- How to secure the accuracy of induced theories on smaller datasets;
- How to deal with negative examples;
- How to reduce communication overhead; and
- How to avoid redundant work by individual processes.

I build a parallel version of a core ILP system – Progol [22] – that shows super-linear speedup in its learning process for a range of data mining problems.

Chapter 2 of this thesis presents the theory and method in inductive logic programming. It reviews several ILP systems and their application in data mining. A particular approach in ILP – Mode-Directed Inverse Entailment (MDIE) [22] – is examined in detail as it is the basis for the parallel version of Progol. The Bulk Synchronous Parallelism (BSP) [8] model is discussed in the latter part of this chapter. A sequential ILP data-mining algorithm and its cost analysis is also provided.

With the theoretical foundations of inductive logic programming in hand, Chapter 3 presents an approach to parallel inductive logic. First a general logical setting for parallel inductive logic programming is given, followed by a detailed discussion of the parallel ILP model. The issues and problems involved in this approach are explored, and a cost analysis is provided.

To examine and support the parallel algorithm discussed in Chapter 3, Chapter 4 presents a parallel ILP system – Parallel Progol. I built this system using the BSP model. It is based on the C version program of Progol implemented by Muggleton. Several test cases are provided and a super-linear speedup phenomenon is explained.

Finally, Chapter 5 summarizes the findings of this thesis and gives a conclusion.

Chapter 2

Inductive Logic Theory and The BSP Model

2.1 Introduction

There are three purposes to this chapter. First, the theory of Inductive Logic Programming (ILP) is briefly introduced. Then a more detailed discussion of one popular ILP approach - Mode-Directed Inverse Entailment (MDIE) - follows. A review of some ILP systems and applications is provided as well. Second, a sequential ILP algorithm based on the MDIE approach is presented, and its cost analysis is provided. Finally, the Bulk Synchronous Parallelism (BSP) model is presented. These three parts form the theoretical foundations of this thesis.

2.2 The Theory of First-order Logic

A first-order language [20] comprises variables, constant symbols, and predicate symbols with their arities. A *term* in a given language is either a variable or a constant. An *atom* is an expression $p(t_1, \cdots, t_n)$, in which $t_1, \cdots, t_n$ are terms, and p is an n-ary predicate symbol. An atom is *ground* if it does not contain any variables. A *literal* is an atom (A) or the negation of an atom (not A). A well-formed-formula is formed using literals and operators such as conjunction, negation, and implication, and the quantifiers $\forall$ and $\exists$. A *sentence* is a closed well-formed-formula (all variables quantified). A *clause* is a disjunction of literals, with all variables universally quantified.

Horn clauses. A definite program clause is a clause containing one positive, and zero or more negative literals. A definite goal is a clause containing only negative literals. A Horn clause is either a definite program clause, or a definite goal. If a definite program clause consists of the positive literal A and the negative literals $\bar{B}_1, \ldots, \bar{B}_n$, then it can be written as

$$A \leftarrow B_1, \ldots, B_n \tag{2.1}$$

where A is called the head of the clause and $B_1, \ldots, B_n$ are called the body literals of the clause [7]. The symbol $\bar{B}_i$ is the negation of B_i.

Model theory. Part of the semantics of first-order logic is a definition of the relation between the terms in the language, and the domain of interest. Each term *refers to* (or *denotes*) an object from this domain. A *pre-interpretation* J of a first-order logic

language L consists of the following: [7]

1. A non-empty set D, called the *domain* of the pre-interpretation.

2. An assignment of each constant in L to an element of D.

A *variable assignment* V with respect to L is a mapping from the set of variables in L to the domain D of J.

An *interpretation* I of a first-order language L consists of the following:

1. A pre-interpretation J, with some domain D, of L. I is said to be *based on* J.

2. An assignment of each n-ary predicate symbol p in L to a mapping I_p from D^n to T, F.

Let ϕ be a formula, and I an interpretation. The formula ϕ is said to be *true under* I if its truth value under I is T. The interpretation I is then said to *satisfy* ϕ. Let $\emptyset$ be a formula, and I an interpretation. The interpretation I is said to be a *model* of $\emptyset$ if I satisfies $\emptyset$. The formula $\emptyset$ is then said to *have* I as a model. For example, let the interpretation I have $D = 1, 2$ as domain, P be a binary predicate interpreted as $\geq$, and let a denote 1 and b denote 2. Then I is a model of the formula $\forall x\ P(x, x)$ since $1 \geq 1$ and $2 \geq 2$. On the other hand, I is not a model of the formula $\forall x \exists y\ (not)\ P(x, y)$, since there is no number n in the domain for which $2 \geq n$ is false [7].

Subsumption of concepts. Computing whether one concept subsumes–is more general than–another is a central facility in all ILP systems. Subsumption is the generality order that is used most often in ILP. The reasons are mainly practical: Subsumption is more tractable and more efficiently implementable than implication [7]. A common method for subsumption computation is based on so-called *θ-subsumption.*

Let C and D be clauses. We say C subsumes D, denoted by $C \succeq D$, if there exists a substitution θ such that $C\theta \subseteq D$ (i.e., every literal in $C\theta$ is also a literal in D). If $C \succeq D$, then there is a substitution θ which maps each $L_i \in C$ to some $M_i \in D$. Examples of subsumption are:

- $C = P(x)$ subsumes $D = P(a) \vee Q(x)$, since $C(x/a) = P(a)$, and $P(a) \subseteq P(a), Q(x)$.
- $C = P(a) \vee P(a)$ subsumes $D = P(a)$.

2.3 Theory of Inductive Logic Programming

What is inductive learning? Inductive learning techniques generalize specific observations into general theories. These theories can be used for explanatory or predictive purposes. Descriptive induction starts from unclassified examples and induces a set of regularities these examples have in common. A typical example in this category is a customer buying-behavior study which discovers that if customers buy sausage and bread that there is a probability of 90 per cent that they will also buy butter. Predictive induction learns from a set of examples that are classified in two or more classes. The aim is to find a hypothesis that classifies the examples in the correct class. A typical example is animal classification. Suppose a robot is instructed to recognize different kinds of animals. Given an example set which contains thousands of animal classification facts, the robot induces useful rules and using such rules it can predict unknown animals. The term inductive logic programming (ILP) was first introduced by Muggleton in 1990 [20]. ILP is concerned with the study of inductive machine learning with the knowledge representation in first-order logic. The goal is

to develop tools and techniques to induce hypotheses from observations (examples) and to synthesize new knowledge from experience [21]:

ILP = Inductive Concept Learning + Logic Programming

Recent studies [6] on this subject shows that ILP is a healthy field but still facing a number of new challenges that it should address.

Why ILP? Data-mining often uses knowledge representation to distinguish different algorithms. Propositional or attribute-value representations use a single table to represent the dataset. Each example or observation then corresponds to a single tuple in a single relation. For all of the attributes the example then has one single value. On the other hand, relational learning and inductive logic programming employ first-order structural representations. For instance, in the learning from interpretations setting, an example corresponds to a set of facts. This generalizes attribute-value representations, as examples now may consist of multiple tuples belonging to multiple tables. A typical example could be a buying behavior analysis for a supermarket. Suppose that the possibility of a male customer buying a particular product is related to how many children he has and if he is a smoker. An example can be expressed in first order logic as: *buy(john, bicycle), man(john), has-children(john,2), not(smoker(john))*. In attribute-value representation we will have to use multiple tuples in multiple tables to express such an example .

Current data-mining systems such as association rule discovery usually deal with numeric values in a relational database, which can be viewed as a propositional or attribute-value representation. If I use a first-order logic representation, I can express

not only the value but also the multi-relationship among those data. By using first-order logic as the knowledge representation for both hypotheses and observations, inductive logic programming may overcome some major difficulties faced by other data-mining systems:

- the use of a limited knowledge-representation formalism
- difficulties in using substantial background knowledge in the learning process.

Previous experiences [7] in expert systems showed that much domain knowledge can be best expressed in a first-order logic, or a variant of first-order logic. Propositional logic has great limitation in certain domains. Most logic programs cannot be defined using only propositional logic. The use of domain knowledge is also crucial because one of the well-established findings of artificial intelligence is that the use of domain knowledge is essential to achieve intelligent behavior. ILP inherits well-established theories, algorithms and tools from computational logic. Many inductive logic programming systems benefit from using the results of computational logic. Background knowledge helps in restricting the hypothesis search and is a key factor for incremental and iterative learning. The concepts generated in each pass are added to the background knowledge. This process will terminate when a predefined accuracy level is reached. Without background knowledge, the hypothesis search space can grow exponentially.

Logical settings of ILP. Inductive Logic Programming is a research field that investigates the inductive construction of concepts from examples and background knowledge. Deductive inference derives consequences E from a prior theory T [21] . Thus if T says that all flying objects are birds, E might state that a particular flying

object is a bird. Inductive inference derives a general belief T from specific beliefs E. After observing one or more flying objects T might be the conjecture that all flying objects are birds. In both deduction and induction, T and E must be consistent and

$$T \models E \tag{2.2}$$

where $\models$ is the symbol of *logical implication.* Within ILP it is usual to separate the elements into examples (E), background knowledge (B), and hypothesis (H). These have the relationship

$$B \wedge H \models E \tag{2.3}$$

where B, H and E are each logic programs. E can be separated into E^+ and E^-.

Normal semantics. Here all examples, background theory and induced concepts are (well-formed) logical formulae. The problem of inductive inference is as follows. Given background (prior) knowledge B and an example set $E = E^+ \wedge E^-$ in which E^+ is positive example set and E^- is negative example set, the objective is to find a hypothesis such that completeness and consistency conditions hold:

- Completeness: background knowledge and induced theory cover all the positive examples.
- Consistency: background knowledge and induced theory do not cover any negative examples.

In most ILP systems, background theory and hypotheses are restricted to being definite. The special case of the definite semantics, where the evidence is restricted to true and false ground facts (examples), will be called the example setting. The

example setting is the main setting of ILP. It is employed by the large majority of ILP systems [7].

Learning from positive data. Some datasets contain only positive data. How to learn from positive data has been a great concern over recent years. When learning from only positive data, predictive accuracy will be maximized by choosing the most general consistent hypothesis, since this will always agree with new data. However, in applications such as grammar learning, only positive data are available, though the grammar, which produces all strings, is not an acceptable hypothesis. Algorithms to measure generality and positive-only compression have been developed [7].

2.4 Theory of MDIE

Introduction. Muggleton has demonstrated that a great deal of clarity and simplicity can be achieved by approaching the problem from the direction of model theory rather than resolution proof theory. My research and experiment on parallel ILP is largely based on a core MDIE algorithm - Progol. So I will introduce the theory of MDIE [22] here. Let us now consider the general problem specification of ILP in this approach. That is, given background knowledge B and examples E find the simplest consistent hypothesis H (where simplicity is measured relative to a prior distribution) such that

$$B \wedge H \models E \tag{2.4}$$

In general B, H and E can be arbitrary logic programs. Each clause in the simplest H should explain at least one example, since otherwise there is a simpler H' which will do. Consider then the case of H and E each being single Horn clauses. This can

now be seen as a generalised form of absorption and rearranged similarly to give

$$B \wedge \bar{E} \models \bar{H} \tag{2.5}$$

where $\bar{E}$ is the negation E and $\bar{H}$ is the negation of H. Let $\overline{\perp}$ be the (potentially infinite) conjunction of ground literals which are true in all models of $B \wedge \bar{E}$. Since $\bar{H}$ must be true in every model of $B \wedge \bar{E}$ it must contain a subset of the ground literals in $\overline{\perp}$. Therefore

$$B \wedge \bar{E} \models \overline{\perp} \models \bar{H} \tag{2.6}$$

and for all H

$$H \models \perp \tag{2.7}$$

A subset of the solutions for H can be found by considering the clauses which θ-subsume $\perp$.

Definition of Mode. In general $\perp$ can have infinite cardinality. I can use mode declarations to constrain the search space for clauses which θ-subsume $\perp$. A mode declaration has either the form modeh(n,atom) or modeb(n,atom) where n, the recall, is either an integer, $n \geq 1$, or * and atom is a ground atom. Terms in the atom are either normal or place-marker. A normal term is either a constant or a function symbol followed by a bracketed tuple of terms. A place-marker is either +type, -type or #type, where type is a constant. If m is a mode declaration then a(m) denotes the atom of m with place-markers replaced by distinct variables. The sign of m is positive if m is a modeh, and negative if m is a modeb. For instance the following are mode declarations.

```
modeh(1,plus(+int,+int,-int))
modeb(*,append(-list,+list,+list))
```

modeb(1,append(+list,[+any],-list))

The recall is used to bound the number of alternative solutions for instantiating the atom.

The most-specific clause. Certain MDIE algorithms, e.g. Progol, search a bounded sub-lattice for each example e relative to background knowledge B and mode declarations M. The sub-lattice has a most general element $\top$ which is the empty clause $\odot$, and a least general element $\bot_i$ which is the most specific element such that

$$B \wedge \bot_i \wedge \bar{e} \vdash_h \odot \tag{2.8}$$

where $\vdash_h \odot$ denotes derivation of the empty clause.

Refinement Operator in MDIE. When generalising an example e relative to background knowledge B, MDIE algorithm constructs $\bot_i$ and searches from general to specific through the sub-lattice of single-clause hypotheses H such that $\odot \preceq H \preceq \bot_i$. This sub-lattice is bounded both above and below. The search is therefore better constrained than other general-to-specific searches in which the sub-lattice being searched is not bounded below. For the purposes of searching a lattice of clauses ordered by θ-subsumption I need a proper refinement operator.

The refinement operator in MDIE is designed to avoid redundancy and to maintain the relationship $\odot \preceq H \preceq \bot_i$ for each clause H. Since $H \preceq \bot_i$, it is the case that there exists a substitution θ such that $H\theta \subseteq \bot_i$. Thus for each literal l in H there exists a literal l' in $\bot_i$ such that $l\theta = l'$. Clearly there is a uniquely defined subset $\bot_i\ (H)$ consisting of all l' in $\bot_i$ for which there exists l in H and $l\theta = l'$. A non-deterministic approach to choosing an arbitrary subset S' of a set S involves maintaining an index k.

For each value of k between 1 and n, the cardinality of S, I decide whether to include the kth element of S in S' . Clearly, the set of all series of n choices corresponds to the set of all subsets of S. Also for each subset of S, there is exactly one series of n choices. To avoid redundancy and maintain θ-subsumption of $\perp_i$, MDIE's refinement operator maintains both k and θ.

Sequential cover algorithm. Based on the theory introduced above, there is a generalized sequential cover algorithm used for MDIE systems, e.g. Progol.

- Select example. Select an example to be generalized. If none exists, stop, otherwise proceed to the next step.

- Build most-specific-clause. Construct the most specific clause that entails the example selected, and is within language restrictions provided. This is usually a definite program clause with many literals, and is called the *bottom clause.* This step is sometimes called the *saturation* step.

- Search. Find a clause more general than the bottom clause. This is done by searching for some subset of the literals in the bottom clause that has the *best* score.

- Remove redundant examples. The clause with the best score is added to the current theory, and all examples made redundant are removed. This step is sometimes called the *cover removal* step.

2.5 Existing ILP systems and applications

This section gives an overview of some core ILP systems, from which we can see that ILP is not only an academic research topic: it has been used in a wide range of machine learning and data-mining applications.

FOIL. FOIL [33] is a system for learning intensional concept definitions from relational tuples. It has been recently applied to web mining [19]. The induced concept definitions are represented as function-free Horn clauses, optionally containing negated body literals. The background knowledge predicates are represented extensionally as sets of ground tuples. FOIL employs a heuristic search strategy which prunes vast parts of the hypothesis space. It is a top-down, non-interactive, batch single-predicate learning algorithm. As its general search strategy, FOIL adopts a covering approach. Induction of a single clause starts with a clause with an empty body which is specialised by repeatedly adding a body literal to the clause built so far. It learns clauses of theory one by one. Each new clause C that the system constructs should be such that C, together with current theory and the background knowledge implies some positive examples that are not implied without C, while C together with the positive examples and background knowledge implies no negative examples. It adds this clause to the current theory and removes the derived positive example from example set. It then constructs another clause, adds it to current theory and so on, until all positive examples can be derived.

Among the candidate literals, FOIL selects one literal to be added to the body of the hypothesis clause. The choice is determined by an information gain heuristic. FOIL's greedy search strategy makes it very efficient, but also prone to exclude the

intended concept definitions from the search space. Some refinements of the hill-climbing search alleviate its short-sightedness, such as including a certain class of literals with zero information gain into the hypothesis clause, and a simple backtracking mechanism.

GOLEM. GOLEM [33] is a "classic" among empirical ILP systems. It has been applied successfully to real-world problems such as protein structure prediction and finite element mesh design. GOLEM copes efficiently with large datasets. It achieves this efficiency because it avoids searching a large hypothesis space for consistent hypotheses like, for instance, FOIL, but rather constructs a unique clause covering a set of positive examples relative to the available background knowledge. The principle is based on the relative least general generalisations (rlggs) [7]. GOLEM embeds the construction of rlggs in a covering approach. For the induction of a single clause, it randomly selects several pairs of positive examples and computes their rlggs. Among these rlggs, GOLEM chooses the one which covers the largest number of positive examples and is consistent with the negative examples. This clause is further generalised. GOLEM randomly selects a set of positive examples and constructs the rlggs of each of these examples and the clause obtained in the first construction step. Again, the rlgg with the greatest coverage is selected and generalised by the same process. The generalisation process is repeated until the coverage of the best clause stops increasing. GOLEM conducts a postprocessing step, which reduces induced clauses by removing irrelevant literals. In the general case, the rlgg may contain infinitely many literals. Therefore, GOLEM imposes some restrictions on the background knowledge and hypothesis language which ensure that the length of rlggs grows at worst polynomially with the number of positive examples. The background knowledge of GOLEM

is required to consist of ground facts. For the hypothesis language, the determinacy restriction applies, that is, for given values of the head variables of a clause, the values of the arguments of the body literals are determined uniquely. The complexity of GOLEM's hypothesis language is further controlled by two parameters, i and j, which limit the number and depth of body variables in a hypothesis clause.

LINUS. LINUS [33] is an ILP learner which incorporates existing attribute-value learning systems. The idea is to transform a restricted class of ILP problems into propositional form and solve the transformed learning problem with an attribute-value learning algorithm. The propositional learning result is then re-transformed into the first-order language. On the one hand, this approach enhances the propositional learners with the use of background knowledge and the more expressive hypothesis language. On the other hand, it enables the application of successful propositional learners in a first-order framework. As various propositional learners can be integrated and accessed via LINUS, LINUS also qualifies as an ILP toolkit offering several learning algorithms with their specific strengths. LINUS can be run in two modes. Running in class mode, it corresponds to an enhanced attribute-value learner. In relation mode, LINUS behaves as an ILP system. Here, I focus on the relation mode only. The basic principle of the transformation from first-order into propositional form is that all body literals which may possibly appear in a hypothesis clause (in the first-order formalism) are determined, thereby taking into account variable types. Each of these body literals corresponds to a boolean attribute in the propositional formalism. For each given example, its argument values are substituted for the variables of the body literal. Since all variables in the body literals are required to occur also as head variables in a hypothesis clause, the substitution yields a ground fact. If it is a true

fact, the corresponding propositional attribute value of the example is true, and false otherwise. The learning results generated by the propositional learning algorithms are retransformed in the obvious way. The induced hypotheses are compressed in a postprocessing step.

The papers [33] and [5] summarize practical applications of ILP:

Learning drug structure-activity rules. The research work carried out by the Oxford machine learning group has shown that ILP can construct rules which predict the activity of untried drugs, given examples of drugs whose medicinal activity is already known. These rules were found to be more accurate than statistical correlations. More importantly, because the examples are expressed in logic, it is possible to describe arbitrary properties of, and relations between, atoms and groups. The logical nature of the rules also makes them easy to understand and can provide key insights, allowing considerable reductions in the numbers of compounds that need to be tested.

Learning rules for predicting mutagenesis. The problem here is to predict the mutagenicity of a set of 230 aromatic and heteroaromatic nitro compounds. The prediction of mutagenesis is important as it is relevant to the understanding and prediction of carcinogenesis. Not all compounds can be empirically tested for mutagenesis, e.g. antibiotics. The compounds here are more heterogeneous structurally than any of those in other ILP datasets concerning chemical structure activity. The data here comes from ILP experiments conducted with Progol. Of the 230 compounds, 138 have positive levels of log mutagenicity. These are labelled *active* and

constitute the positive examples: the remaining 92 compounds are labelled *inactive* and constitute the negative examples. Of course, algorithms that are capable of full regression can attempt to predict the log mutagenicity values directly.

Learning rules for predicting protein secondary structure. Predicting the three-dimensional shape of proteins from their amino acid sequence is widely believed to be one of the hardest unsolved problems in molecular biology. It is also of considerable interest to pharmaceutical companies since a protein's shape generally determines its function as an enzyme.

Inductive Learning of Chess Rules Using Progol. Computer chess programs can be thought of as having two parts, a move generator and an algorithm for evaluating the strength of generated moves. The move generator effectively gives the computer information concerning the rules of chess.

The structured method used here is slightly larger, involving the splitting of the problem into some 40 sub-problems, creating a structure some 15 levels deep. With structured induction, clauses learned in an earlier part of the process are appended to the background knowledge to enable the learning of subsequent clauses.

First-Order Learning for Web Mining. Two real-world learning problems that involve mining information from the web with first-order learning using FOIL have been demonstrated [19]. The experiment shows that, in some cases, first-order learning algorithms learn definitions that have higher accuracy than statistical text classifiers. When learning definitions of web page relations, they demonstrate that first-order learning algorithms can learn accurate, non-trivial definitions that necessarily

involves a relational representation.

Other ILP applications mentioned in [33] are:

- Learning rules for finite element mesh design.
- Learning diagnostic rules for qualitative models of satellite power supplies.
- Learning qualitative models of the U-tube system.
- Learning qualitative models for functional genomics.

2.6 Sequential ILP Algorithm.

In this section, I analyze a general MDIE ILP algorithm and provide a cost analysis of this algorithm. Chapter 3 discusses how to parallelize this sequential algorithm and analyze its cost. In Chapter 4, I will discuss how to implement a parallel Progol system based on the parallel approach introduced in Chapter 3, and provide some test cases as examples to support the cost analysis.

2.6.1 A Sequential ILP Algorithm

In order to give a parallel approach to ILP data mining, first I need to know the general steps involved in a sequential ILP data-mining algorithm. As shown in Section 2.4, a mode-directed approach can provide a much simpler and convenient way in inductive concept learning. So I provide a general ILP data-mining procedure based on mode-directed inverse entailment(MDIE). The whole sequential ILP data-mining procedure consists of a loop structure: In each cycle, some concepts are learnt and

```
repeat
  if there is still a positive e in E not covered by H and B
        select an example e in E
        search for a good concept H that covers e
        add H to background knowledge B
        retract redundant examples that covered by H
  end if
end repeat
```

Figure 2.1: Sequential ILP Algorithm

some positive examples that are covered by the new induced concepts are retracted from the dataset. The loop will come to an end when all positive examples are covered by the final induced concept set and no positive examples are left in the dataset. Figure 2.1 gives a general sequential ILP approach.

Several issues in the above algorithm need to be addressed.

How to select an example? The example selection procedure can be random. The example selection can also be based on the sequence order in the dataset: ILP picks up one positive example after another in the order in which they are located in the dataset. A more sophisticated approach is to pick up an example according to its score. The score of an example is determined by its properties, i.e., an example gets a high score when its occurrence in the whole dataset is more frequent than other examples. In this way, ILP can possibly induce the most important concepts first. When one concept is generalized and it covers more positive examples in the dataset, the dataset shrinks more quickly after each loop, thus improving the performance of the whole learning procedure. Though this approach seems plausible in a sequential

algorithm, it can be potentially problematic in parallel approach. In parallel learning, the last approach will increase the chance that two or more processors select the same example and thus waste time inducing the same concept.

How to generalize concepts from examples? It is the most important task of the whole job that distinguishes different ILP systems. An induced concept set is too strong if it wrongly covers some negative examples and thus makes it inconsistent. On the other hand, a concept set is too weak if it cannot cover all the positive examples and thus makes it incomplete. A concept set is overly general if it is complete with respect to positive example set E^+ but not consistent with respect to negative concept set E^-. A concept set is overly specific if it is consistent with respect to E^- but not complete with respect to E^+. An ILP system is meant to search in the hypothesis space and find a concept set that is neither too strong nor too weak.

The two basic techniques in the search for a correct concept set are specialization and generalization. If the current concept set together with the background knowledge contradicts the negative examples, it needs to be weakened. That is, I need to find a more specific theory, such that the new theory and the background knowledge are consistent with respect to negative examples. This is called specialization. On the other hand, if the current theory together with the background knowledge does not cover all positive examples, I need to strengthen the theory: I need to find a more general theory such that all positive examples can be covered. This is called generalization. Note that a theory may be too strong and too weak at the same time, so both specialization and generalization are needed.

To achieve the above goal, I introduce two approaches here.

- Top-Down. Start with a theory Σ such that $\Sigma \cup B$ is overly general, and specialize it.
- Bottom-Up. Starts with a theory Σ such that $\Sigma \cup B$ is overly specific, and generalize it.

In MDIE, a most-specific clause is formed at the first phase when generating a hypothesis from an example. Then it searches the hypothesis space from general to this most specific clause to find a *good* concept.

What does *good* mean? During the search in the hypothesis space, an ILP system will generate and evaluate some candidate concepts. I need to determine which concept is better than other candidates. In practice, a score can be assigned to each candidate concept. The candidate concept with the highest score will be the right one. Then there comes a problem: How is the score decided? One way to decide the score is to calculate it from a few parameters using a function $f(y, n, c)$ which gives each induced candidate hypothesis H a score based on:

- y: the number of positive examples covered by H
- n: the number of negative examples wrongly covered by H
- c: the conciseness of H, which is generally measured by the number of literals in H

For example, f could be:

$$f(y, n, c) = y + c - n \tag{2.9}$$

The candidate with the highest score is added to the final set of induced theory. When generating a candidate H' from the example e, the ILP algorithm generally searches in the hypothesis space to find the candidates. In order to give a score $f(y, n, c)$ to each candidate H', the ILP algorithm has to look through the entire dataset.

2.6.2 Cost of sequential ILP data-mining algorithms.

The ILP data-mining algorithm described above has the property that its global structure is a loop, extracting more concepts through each iteration. Suppose this loop executes k_s times. I can describe the sequential complexity of this algorithm with a formula:

$$cost_s = k_s [STEP(nm) + ACCESS(nm)] \tag{2.10}$$

where $STEP$ gives the cost of a single iteration of the loop, and $ACCESS$ is the cost of accessing the dataset in one step; n is the number of objects in the dataset and m is the size of each example. To give a more specified cost model for sequential ILP algorithm, there is another formula:

$$\begin{aligned} cost_s = k_s [&SEL(nm) + \varepsilon(GEN(nm) + EVA(nm)) \\ &+ RET(nm)] \end{aligned}$$

where SEL gives the cost of selecting one example from the dataset; GEN gives cost of generating one candidate hypothesis from the selected example; and EVA gives the cost of evaluation of candidate hypothesis and giving it a score. Usually this step involves accessing the dataset once. RET gives the cost of retracting redundant

positive examples already covered by the newly induced concept. ε gives the number of candidate hypothesis generated in each step. Please notice that *EVA* and *RET* involve data access so they will dominate costs for large datasets.

The cost of *SEL* varies in different implementations, from only one data access in random or sequential selection to entire dataset access in some more sophisticated algorithms. I assume sequential or random selection is adopted in the ILP algorithm. Also the cost of *GEN* varies in different ILP inductive learning algorithms. In MDIE it involves first building the most specific clause and then searching from hypothesis space to construct each candidate hypothesis. It is the most significant computational cost in one step.

The value of ε depends on the hypothesis search space and search algorithm. Most ILP algorithms will search in the hypothesis space from general to specific or vice versa to get satisfied concepts. To reduce the search space some algorithms adopt language bias such as MODE declaration in MDIE. Also some heuristic search algorithms will help to reduce the value of ε. Since this value determines the number of passes through the entire dataset in each step, it is critical to the performance of ILP data-mining system.

The *EVA* cost usually involves one pass through the entire dataset to give each candidate hypothesis a score. In the same way, the cost of *RET* also involves one pass through the entire dataset to remove redundant examples.

If after each loop η ($0 \leq \eta \leq 1$) examples remain not covered by the newly-induced concept, I can give a more accurate formula:

$$cost_s = k_s[SEL(nm * \eta^i) + \varepsilon(GEN(nm) + EVA(nm * \eta^i)) + RET(nm * \eta^i)]$$

where after each big step a fraction of $(1 - \eta)$ examples are removed from the dataset and the work in next step is reduced by a factor of η.

2.7 Introduction to the BSP model

Introduction. I have discussed the sequential ILP algorithm above. Now I come to the point of how to make it work in parallel, and how to speed up its learning process. At this point, a parallel computing model is needed. Bulk Synchronous Parallelism (BSP) [31] provides a model for the parallel system. I can perform cost analysis based on BSP cost equations without having to implement different kinds of systems [8]. In traditional message-passing systems, a programmer has to ensure, explicitly, that no conflict will occur when one data item is being accessed by two or more processes. Though some systems can provide deadlock control, concurrency control or remote data access control, these mechanisms introduce cost overhead. It is hard to establish a cost model with the great variety of the memory access patterns and network architecture.

A parallel complexity measure that is correct to within a constant factor is needed. Such a measure must take into account costs associated with the memory hierarchy and accurately reflect the costs of communication, whether explicitly, as in message-passing programs, or implicitly, as in shared-memory programs.

Bulk Synchronous Parallelism (BSP) is a parallel programming model that divides computation and communication into separate phases. Such phases are called supersteps. A superstep consists of a set of independent local computations, followed by a global communication phase and a barrier synchronisation. Writing programs with the BSP model enables their costs to be accurately determined from a few simple architectural parameters. Contrary to general belief, the structure imposed by BSP does not reduce performance, while bringing considerable benefits from an application-building perspective.

BSP programming. Supersteps are an important concept in BSP. A BSP program is simply one which proceeds in phases, with the necessary global communications taking place between the phases. This approach to parallel programming can be applied to both distributed systems and shared-memory multiprocessors. BSP provides a consistent, and very general, framework within which to develop portable software for scalable computing.

A BSP computation consists of a sequence of supersteps, where each superstep is a sequence of steps carried out on local data, followed by a barrier synchronisation at which point any non-local data accesses take effect. Requests for non-local data, or to update non-local data locations, can be made during a superstep but are not guaranteed to have completed until the synchronisation at superstep end. Such requests are non-blocking; they do not hold up computation.

The programmer's view of the computer is that it has a large and universal accessible memory. To achieve scalability it will be necessary to organise the calculation in such a way as to obviate the bad effects of large latencies in the communication network.

By separating the computation on local data from the business of transferring shared data, which is handled by lower level software, I ensure that the same computational code will be able to run on different hardware architectures from networked workstations to genuinely shared-memory systems.

The superstep structure of BSP programs lends itself to optimization of the data transfers. All transfers in a superstep between a given pair of processors can be consolidated to form larger messages that can be sent with lower (latency) overheads and so as to avoid network contention. The lower level communications software can also exploit the most efficient communication mechanisms available on the actual hardware. Since this software is application-independent, the cost of achieving the efficiency can be spread over many applications.

BSP cost model. I need to identify the key parameters of a BSP parallel system that determine its performance [8]. Obviously the number of processors and computational speed of each are key parameters. If I define a step to be the basic unit of calculation, then I can denote the speed as s steps/sec.

I can also see that the capacity and speed of the communications network is a vital element. For ease of comparison between systems, I will measure the performance of the communications network in units of the computing speed. The cost of carrying out a barrier synchronisation of the processors, for example, can be measured in terms of the number of steps that could have been performed in the time taken to synchronise. This lets us contrast a system with fast synchronisation, in which relatively few steps can have been executed during the time it takes to synchronise, with one which has much worse performance relative to its computational power. In general I can expect better overall performance from a system with low values of this parameter.

Similarly when I estimate the communications throughput of the network linking the processors, I look at the cost in steps for each word of data transmitted. This gives the ratio of the computing power to the communication power of the system. The lower this figure is, the better the balance between compute power and communications power, and the easier it is to get scalable performance.

I therefore arrive at the following four parameters [31], which extensive research has shown to be sufficient:

- p = number of processors
- s = processor speed (number of steps per second)
- l = the cost, in steps, of achieving barrier synchronisation (depends on network latency)
- g = the cost, in steps per word, of delivering message data

Note that all are based on the bulk properties of the system. The values are determined by actual measurement using suitable benchmarks that mimic average computation and communication loads.

The speed s is measured as the actual rate at which useful calculation is done; it is not the peak performance figure quoted in the manufacturer's data sheet.

The value of g is calculated from the average cost of transferring each word of messages of all sizes in the presence of other traffic on the network. It is not based on the manufacturer's claimed bisection bandwidth. It is not measured from single point-to-point transfers but measures the sustainable speed that will be experienced by real application code. The g value can be approximated by calculating (total number of local operations by all processors per second)/(number of words delivered

by the communications system per second) The value g enables you to estimate the time taken to exchange data between processors. If the maximum number of words arriving at any one processor during a single such exchange is h, then I estimate that up to gh steps can have been executed during the exchange.

Another advantage of the simple structure of BSP programs is that the modeling of their performance is much easier than for message passing systems, for example. In place of the random pair-wise synchronisation that characterises message passing, the superstep structure in BSP programs makes it relatively easy to derive cost models (i.e. formulae that give estimates for the total number of steps needed to carry out a parallel calculation, including allowance for the communications involved).

Cost models can be used to determine the appropriate algorithmic approach to parallelisation. They enable us to compare the performance of different approaches without writing the code and manually measuring the performance. And they provide predictors for the degree of scaling in performance that is to be expected on any given architecture for a given problem size.

Cost models have proved to be very useful guides in the development of high quality parallel software.

Oxford BSPlib. Like many other communications libraries, BSPlib adopts a Single Program Multiple Data (SPMD) programming model. The task of writing an SPMD program will typically involve mapping a problem that manipulates a data structure of size N into p instances of a program that each manipulate an N/p sized block of the original domain. The role of BSPlib is to provide the infrastructure required for the user to take care of the data distribution, and any implied communication necessary to manipulate parts of the data structure that are on a remote process.

bsp-begin	Start of SPMD code
bsp-end	End of SPMD code
bsp-init	Simulate dynamic processes
bsp-abort	One process stops all
bsp-nprocs	Number of processes
bsp-pid	Find my process identifier
bsp-time	Local time
bsp-sync	Barrier synchronization
bsp-push-reg	Make area globally visible
bsp-pop-reg	Remove global visibility
bsp-put	Copy to remote memory
bsp-get	Copy from remote memory
bsp-set-tagsize	Choose tag size
bsp-send	Send to remote queue
bsp-qsize	Number of messages in queue
bsp-get-tag	Getting the tag of a message
bsp-move	Move from queue
bsp-hpput	Unbuffered communication

Table 2.1: BSPlib Operation

An alternative role for BSPlib is to provide an architecture-independent target for higher-level libraries or programming tools that automatically distribute the problem domain among the processes. I use BSPlib to develop the parallel ILP system and do the cost analysis.

Table 2.1 is a list of BSPlib operations:

Summary. In this chapter I introduced basic knowledge of ILP and MDIE. A sequential ILP algorithm was discussed, and it cost analysis was provided. To implement a parallel ILP algorithm, the BSP model is introduced. In the next chapter I will discuss a parallel ILP algorithm using the BSP model and based on the sequential algorithm introduced in this chapter.

Chapter 3

Parallel Inductive Logic in Data Mining

The general task of inductive logic programming is to search a predefined subspace of first-order logic for hypotheses, together with background knowledge, that explain examples. However, due to the expressiveness of knowledge representation such a search is usually computationally expensive. Most ILP systems have to pass over the entire example set many times to find a successful induced theory H among other candidate induced concepts, which in turn increases the computation cost tremendously. When such ILP systems are to be applied to real-world data-mining tasks, the expensiveness of algorithm seems to be a big obstacle. Thus, how to speed up the learning process of ILP algorithm has become a practical and critical issue. In this section, I present and discuss a parallel approach that shows a linear or super-linear speed up on some applications for traditional sequential ILP algorithms. Important issues in this approach are discussed in detail. A cost analysis of the parallel ILP algorithm is provided as well.

3.1 Reason, possibility and approaches of parallel ILP in data mining

As I mentioned above, there are some reasons why parallelism in ILP data-mining is needed. The first and most obvious reason concerns the data size. The databases used for data mining are typically extremely large. As these databases grow past hundreds of gigabytes towards a terabyte or more, it becomes nearly impossible to process them on a single sequential machine running a single sequential algorithm. Another reason for the need of parallelism is the expensiveness of ILP systems. This expensiveness comes from two aspects:

- The powerful and expressive knowledge representation in ILP requires more computation power than propositional data-mining.
- Searching the entire dataset many times to find a successful hypothesis H among candidate concepts increases the disk access greatly. Therefore, disk (I/O) access is one of the most serious bottlenecks for sequential ILP systems.

Parallel ILP data-mining requires dividing the task, so that processors can make useful progress towards a solution as fast as possible. From the sequential ILP algorithm I can see that the disk access is one of the most significant bottleneck. Therefore, how to divide the access to the dataset and minimize communication between processors are important to the total performance.

In general, there are three different approaches [29] to parallelizing data mining. They are:

- Independent Search. Each processor has access to the whole dataset, but each heads off into a different part of the search space, starting from a randomly chosen initial position.

- Parallelize a sequential data-mining algorithm. There are two forms within this approach. One approach is that the set of concepts is partitioned across processors, and each processor examines the entire dataset to determine which of its local concepts is globally-correct. The other approach is to partition the dataset by columns, and each processor computes those partial concepts that hold for the columns it can see. Regular exchanges of information of concepts are required in both approaches to determine which partial concepts are globally-correct.

- Replicate a sequential data-mining algorithm. Each processor works on a partition of the dataset and executes the sequential algorithm. Because the information it sees is only partial, it builds entire concepts that are locally correct, but may not be globally correct. Such concepts are called as approximate concepts. Processors exchange these approximate concepts to check if they are globally-correct. As they do so, each learns about the parts of the dataset it cannot see.

Independent search is simple and works well for minimization problems. However, it does not divide the dataset, so it cannot reduce the disk access. Therefore, it is not suitable for problems with huge dataset. Parallelized approaches try to reduce both the amount of memory each processor uses to hold concepts and the fraction of the dataset that each processor must access. But its fine-grained parallelism requires too much extra communication.

The replicated approach is often the best way for parallelizing ILP data-mining applications. It has two significant advantages: First, it necessarily partitions the dataset and so spreads the disk access. Second, the size of induced concepts that must be exchanged between phases is small, so communication is cheap. Previous work in *using the BSP cost model to optimize parallel neural network training* [27] shows that the replicated approach gives the best performance improvement among all these three approaches introduced above. I adopt this approach in the parallel ILP algorithm for its simplicity and possibility of a *double* speedup. I will discuss double speedup in the following sections.

The following shows the possibility of adopting parallelism in ILP data mining.

- Due to the nature of data mining, there are lots of similarities and redundancies within the large dataset. Therefore, it is plausible to induce correct theories from a small subset of the full data.

- In most ILP systems, the whole concept-learning process consists of a loop structure. After each loop, a successful hypothesis is found and added to the final induced theory set. The learning process stops when all the positive examples have been explained by the induced concepts. The size of the induced hypothesis during each phase is small compared to the dataset. So it is plausible to let p processes induce concepts from a subset. At the end of each phase, these p processes exchange the locally-induced concepts and determine the valid (globally-correct) concepts after evaluation.

3.2 Logical Settings Of Parallel ILP

In this section I discuss the logical setting of the division of the ILP task into subtasks that can be handled concurrently by multiple processes executing a common sequential ILP algorithm. I try to explore a parallel approach to obtain an algorithm with a speedup proportional to the number of processors over the best available sequential algorithm.

A central issue in designing a computer system to support parallelism is how to break up a given task into subtasks, each of which will be executing in parallel with the others. In general, ILP starts with an initial background knowledge B and some examples E. The aim is to induce a hypothesis H that, together with background knowledge B, explains the examples E.

A partition $T_1, \cdots, T_n$ of an ILP-task $T = (B, E)$ is a set of ILP tasks. $T_i = (B, E_i)$ such that $E_i \subset E$ for all i, and that $(\cup_{i=1}^{n} E_i) = E$. The partition $T_1, \cdots, T_n$ of an ILP-Task T is valid if and only if the union $\cup_{i=1}^{n} H_i$ of partial hypothesis H_i obtained by applying a common sequential ILP algorithm A to task T_i is equivalent to the solution hypothesis H obtained by applying algorithm A to task T. Completeness and consistency of parallel ILP can be expressed as follows:

Completeness:

$$B \cup (\bigcup_{i=1}^{n} Hi) \models E^{+} \leftrightarrow \begin{cases} B \cup H_1 \models E_1^{+} \\ B \cup H_2 \models E_2^{+} \\ \cdots \\ B \cup H_n \models E_n^{+} \end{cases}$$

Consistency:

$$B \cup (\bigcup_{i=1}^{n} Hi) \cup E^{-} \not\models \odot \leftrightarrow \begin{cases} B \cup H_1 \cup E_1^{-} \not\models \odot \\ B \cup H_2 \cup E_2^{-} \not\models \odot \\ \cdots \\ B \cup H_n \cup E_n^{-} \not\models \odot \end{cases}$$

I will explore and explain in an intuitive way why in the parallel approach the completeness and consistency hold.

3.3 An Approach to Parallel ILP Using the BSP Model

Based on the sequential algorithm I discussed in Chapter 2, I give a parallel ILP algorithm based on the replicated approach discussed above. There are two significant reasons for using the replicated approach. First, it partitions the entire dataset and so spreads the data access cost across processors. Second, the data that must be exchanged between phases is small, so communication is cheap. The size of a concept generated by each processor in one step is around 10^2 characters in the test cases. If the value of g is 4.1 flops/32 bit word and there are 4 processors, then the communication cost per total exchange equals 1600 flops. It is quite small compared to the local computation cost or data access cost which are hundreds of times bigger. Therefore, though the replicated approach is not particularly novel, it is perhaps the best way to increase performance in ILP data-mining tasks.

I divide the full dataset into p subsets and allocate each subset to one processor.

```
divide dataset into p subsets
repeat
  for all processors i
     if there is still an e in E_i
       select e in E_i
       form a set of good concepts H_i that covers e
       total exchange H_i(i = 1, 2..p)
       evaluate H_i(j = 1, 2 ..., p)
       total exchange evaluation result of H_i
       find the successful H_i with globally good score
       total exchange which are the valid H_i
       add all valid H_i into B
       retract redundant examples that covered by H_i
     end if
  end for
end repeat
```

Figure 3.1: Parallel ILP Algorithm

Each processor executes the same (or similar) sequential ILP data-mining algorithm introduced above on its local subset of data. At certain synchronization points, all these processors exchange their local induced concepts and evaluate them. Only globally-correct concepts will be left, and added to the final concept set. Figure 3.1 gives the parallel algorithm.

In this approach, each processor works on its subset to find a locally-correct concepts set H_i in each step. The measure $f(y, n, c)$ in each processor is based on its own subset of data. In order to know whether this locally-correct concept set is also globally-correct and to find the successful H in the set, it is necessary to find a way of learning the general knowledge of the whole dataset. To do so, all p processors perform a total exchange after all the processors reach the synchronization point

when they have found their locally-correct concept $H_i's$. After the total exchange, each processor gets all the H_i induced by peer processors. Each processor gives every $H_i(i = 1, 2, ..p)$ a score $f(p, n, c)$ based on its local knowledge from the subset of the full data. Then there will be a second total exchange: the evaluation result of $H_i's$ will be exchanged among p processors. In this way each processor learns the whole dataset and can give a global score to its local candidate concept H_i. With the third phase of total exchange the valid $H_i's$ are added to each processor's final concept set and redundant examples are retracted.

The whole computation in the approach consists of a sequence of supersteps, where each superstep is a sequential ILP computation carried out on local data, followed by a barrier synchronization at which point all induced concepts in this step are exchanged and evaluated. The cost of such a phase is described by an expression of the form:

$$Cost = \underset{\text{processes}}{\text{MAX}}\ w_i + \underset{\text{processes}}{\text{MAX}}\ h_i\, g \tag{3.1}$$

where w_i is the number of instructions executed by processor i. The value of h_i is the size of the concepts exchanged between processors. This cost model is derived from BSP, which I introduced in Chapter 2. The system parameters of s, l, g can be obtained from the Oxford BSPlib. Notice that both terms are in the same units: time. This avoids the need to decide how to weight the cost of communication relative to computation, and makes it possible to compare algorithms with different mixes of computation and communication.

3.4 Potential problems with this approach

3.4.1 Accuracy of induced theory on smaller dataset

Since I use p processors to do the data mining job on a subset of the full dataset, a set of concepts will be generated from disjoint subsets of the full dataset used for mining. Given p disjoint subsets of the full dataset there will be p sets of concepts generated by each processor. Each subset of data resides on a distinct processor. The distributed concept sets must be totally exchanged and evaluated before merging the valid ones into the final concept set. The final set of concepts should be free from conflicts and same as the set of rules developed from the full dataset.

There is a question as to how to ensure that the individual concepts generated by each processor which are locally-correct are also globally-correct. If each processor spends a lot of time only to find unwanted concepts, there will be no performance improvement from parallelism.

Any concept acceptable on the full dataset will be acceptable on at least one disjoint subset of the full data [7]. This suggests that a concept set created by merging sets of acceptable concepts contain concepts that would be found on the full dataset. Earlier work [7] has found that the merged set of concepts contained the same concepts as found by learning on the full dataset. If there are enough representative examples for each class in each of p disjoint partitions, the concepts found in the parallel version will have high accuracy.

In the approach to parallel data-mining an important question is how large p can be before communication costs begin to slow the concept generation process significantly. But the more important question is how to determine a p for which the accuracy of the resultant concept set is acceptable. There is a tradeoff between accuracy and speed. The use of more processors promises that each can learn faster on a smaller subset of data at the usual cost of communication overhead. However, there is a second accuracy cost that will be paid when at some point p becomes too large and it is therefore hard to maintain in each subset the representative examples of the full data set. Previous work [27] done by Owen Rogers in parallel neural network mining shows that correct concepts can be generated from a small subset of the entire data but have taken much less processing to discover. When the subset size reaches some size bound, however, the concepts generated becomes less accurate and hence do not help. That means in the parallel algorithm I can divide the dataset into smaller subsets and at the same time keep the induced concepts accurate enough to show a significant performance increase, provided the size of each subset is greater than that size boundary.

3.4.2 Dealing with Negative Examples

There is always a problem with dealing with negative examples, that is, how to make sure one concept induced by one processor is consistent with all other subsets? If one concept which is locally consistent can be easily rejected by other processors, there will be a severe cost efficiency issue with this approach. In fact, the problem may be not as serious as it appears to be. There are several reasons:

- Negative examples in real-world applications are usually rare among the entire dataset. Hence it is reasonable to assume that the chances that one locally consistent concept is also globally consistent are high. Even though there are some cases that some locally consistent concepts are rejected by other processors, the total negative cost is not too high and can be tolerated compared to the speedup gained.

- Since the number of negative examples is small compared to positive examples, I can keep a duplicate copy of the entire negative example set on each processor. In this way all locally-consistent concepts are also globally-consistent at the cost of some redundancy. This is the approach I adopted in the test cases. There are some negative examples in test case 2 -the chess move learner. Since the size of negative examples is not too big compared to positive ones (about 10 per cent), I duplicate all the negative examples across the processors. Though there is redundancy in the local subset of data, the overall performance increase is still obvious and a double speedup is observed.

- There are some effective learning algorithms that can learn from only positive data. There is no consistent issue when learning from positive data, which is the case in test cases 1 and 3.

3.4.3 Communication Overhead

There is a concern that at certain stages the number of processors becomes too large and the communication cost is too big. However, the communication cost is not a big problem in the parallel approach.

- First, the size of the data to be exchanged between processors is small. Since only the induced concepts are exchanged and the size of an induced concept – usually a logical clause – is quite small, the communication cost of exchanging such small size concepts is not a big concern in the approach.

- Second, I have to maintain a reasonable amount of data in each subset to ensure that there are enough representative examples. This, in turn, keeps p from growing too big.

- Third, since each processor performs the same sequential algorithm and the size of each subset is similar, it is reasonable to predict that the time spent on local computation on each of the p processors is comparable. Therefore the synchronization model need not be a big performance concern here in this approach.

From the analysis above I can draw a conclusion that the communication overhead is small compared to the local computation cost saved. This conclusion is supported by the test cases. In the test cases, the size of induced concepts is around 100 characters. The value of g is 4.1 flops/32 bit Word. The value of l is 118 flops. There are three total communications within one big step, and there are 4 processors working in parallel. So I get the communication cost in one big step: 3 *(4*100*4.1 + 118) = 5274 flops. The CPU speed is 10.1 Mflops. Then the cost of communication in one big step is around 0.0005 second. The cost of local computation and disk access cost in one step is greater than 1 second in the test cases. It is easy to get the conclusion that the communication overhead in the parallel ILP algorithm is not a big issue.

3.4.4 Redundant Work by Individual Processors

There is a debate over how to ensure that different processors do their part of the job as there will not be too much time wasted doing redundant work. Such a situation is likely to happen when the full dataset contains similar and/or redundant examples. Since one subset might contain the same or similar examples in another subset, there is a chance that the two processors on these two subsets select the same example in one particular step and do a redundant induction. If there are many redundant examples in the subsets, such redundancy might become a serious problem, affecting overall performance. I found by experiment that this problem is not as serious as it seems. The reasons are:

- First, if the selection process chooses an example randomly or by sequence order, the chances of two or more processors selecting the same example are small in a big and randomly-distributed dataset.

- Second, when one processor induces a valid (globally-correct) hypothesis from one example, this hypothesis will be updated into all processors induced theory set and all examples covered by this hypothesis will be retracted from each processor's example subset. Such a mechanism will eliminate the chance of redundant work done by different processors in different steps.

- Third, even if there are still some cases that two or more processors select the same example in the same step, it is not a great factor in the overall performance. In the test cases, such redundancy occurs in some big steps. But there is still obvious performance improvement in parallel approach.

In the experiment I found such chances are small even though the datasets contained many redundant and similar examples.

3.5 Cost Analysis and Argument for a Double Speedup

The parallel approach mentioned above is structured in a number of phases, each of which involves a local computation, followed by an exchange of data between processors. In this approach it is straightforward to tell when computation will dominate memory access, and the memory access cost is predictable. The cost model presented above is likely to produce accurate estimates of running times on existing parallel computers. Because the cost model depends only on high level properties of algorithms, it can be applied to an algorithm in the abstract.

The basic structure of the parallel algorithm is:

- Partition data into p subsets, one per processor.
- Repeat

  Execute the sequential ILP algorithm on each subset.

  Exchange information about what each processor learned with the others.

So the cost has the following general form:

$$cost_r = k_r[STEP(nm/p) + ACCESS(nm/p) + COMM(p, r)] \qquad (3.2)$$

where k_r is the number of iterations required by the parallel algorithm, r is the size of the data about candidate concepts generated by each processor, $COMM$ is the cost

of total exchange and evaluation between the processors of these candidate concepts.

It is reasonable to assume that:

$$STEP(nm/p) = STEP(nm)/p \tag{3.3}$$

$$ACCESS(nm/p) = ACCESS(nm)/p \tag{3.4}$$

First, if I assume that k_s and k_r are of comparable size, I get

$$cost_r \approx cost_s/p + k_r COMM(p, r) \tag{3.5}$$

We expect an almost linear speedup. To make the above formula more specific according to parallel ILP algorithm, I get

$$\begin{aligned} cost_r = k_r[&SEL(nm/p)) + \varepsilon(GEN(nm/p) + EVA(nm/p)) \\ &+ 3(rpg + l) + p * EVA(nm/p)) + RET(nm/p)] \end{aligned}$$

where *SEL* gives the cost of selecting one example from the dataset; *GEN* gives cost of generating one candidate hypothesis from the selected example; *EVA* gives the cost of evaluation of candidate hypothesis and giving it a score; and *RET* gives the cost of retracting redundant positive examples already covered by the newly induced concept. ε gives the number of candidate hypothesis generated in each step. The symbol $rpg + l$ is the cost of a total exchange of candidate concepts between processors; since there are three total exchange in the parallel algorithm, the overall communication cost should be $3(rpg + l)$. Since each processor will get and evaluate p candidate concepts generated from p processors, the cost of evaluation *EVA* should

be multiplied by a factor of p.

It is reasonable to assume:

$$EVA(nm/p) = EVA(nm)/p \tag{3.6}$$

$$RET(nm/p) = RET(nm)/p \tag{3.7}$$

since the value of *GEN(nm)* is usually much smaller than the value of *EVA(nm)* when the dataset is big, I get

$$cost_r \approx cost_s/p + k_r(3rpg + l) \tag{3.8}$$

If k_s and k_r are of comparable size, I get a $p-$fold speedup except for a communication overhead.

In this approach, each processor induces the concepts from its own subset of data independently. So it is likely that the concepts induced by different processors are different. Frequent exchange of these concepts will improve the rate to which concepts are induced. One processor will learn concepts induced by other processors during the total exchange phase. Therefore we might actually expect that $k_r \ll k_s$. This phenomenon is called *double* speedup. The interesting phenomenon of double speedup occurs in the test examples. Each processor learns, in a condensed way, what every other processor has learned from its data, whenever communication phases take place. This information has the effect of accelerating its own learning and convergence. The overall effect is that k_r is much smaller than k_s would have been, and this in turn leads to a double speedup. If each subset maintains the characteristics of the entire

dataset, there is much chance that the locally-correct concepts will be also globally-correct. If the algorithm selects an example randomly, the chances that two or more processors working on the same example are small. All these arguments suggest a much quicker learning process, which is observed in the test cases.

Suppose that the first phase of the sequential algorithm requires work (computation) w, but that the work in the subsequent phases can be reduce by a multiplicative factor α. Then the sequential algorithm has a computation cost of the form

$$(1 + \alpha + \alpha^2 + \alpha^3 + \alpha^4 + \cdots)w \tag{3.9}$$

The parallel algorithm, say, using four processors takes less time overall. The first parallel phase takes time w, but the second phase takes only $\alpha^4 w$, and so on. This reduction is a function of w, which in turn is a function of the size of the dataset. Then the parallel algorithm has a computation cost of the form

$$(1 + \alpha^4 + \alpha^8 + \alpha^{12} + \cdots)w \tag{3.10}$$

If $\alpha = 0.9$, then $cost_r/cost_s \approx 0.39$; if $\alpha = 0.1$, then $cost_r/cost_s \approx 0.90$. This analysis is optimistic in that I assume the reduction is independently additive and the communication overhead is not included in this calculation. However, it provides an explanation why double speedup occurs in the experiments.

Summary. In this chapter I proposed a parallel ILP algorithm, which is based on the sequential algorithm introduced in Chapter 2. The related issues in this parallel approach are discussed in detail, which are:

- Accuracy of induced theory on smaller dataset.

- Dealing with negative examples.
- Communication overhead.
- Redundant work by individual processes.

A cost analysis is provided using the BSP cost model. A possibility of double speedup phenomenon is discussed. A parallel ILP system based on this parallel approach will be discussed in next chapter. Some test cases will be provided, and the cost analysis will be given.

Chapter 4

Parallel Progol

To make the arguments in Chapter 2 more concrete, I developed some programs to show how parallel ILP works and give a performance analysis. Since Progol is a core MDIE ILP system and has drawn much research interests in recent years, I decided to parallelize the Progol system. The source code of Progol in C is freely available from the Oxford University machine learning group web site. I implement a parallel version of CProgol – PCProgol – that induces concepts in parallel on several processors with the support of Oxford BSPlib. To show how PCProgol works, I developed three test cases in this chapter. They all show a super-linear speed up relative to the number of processors.

4.1 Parallel Progol Algorithm

According to the general parallel ILP approach discussed in Chapter 3, I divide the example set into several subsets, each of which is saved to a file. All the same background knowledge and mode declarations are included in each file. Multiple processes

forall processor i
start:if E_i = empty return B
let e be the first example in E_i
construct the most specific clause $\perp$ for e
construct hypothesis H_i from $\perp$
propagate H_i to all other processes
evaluate H_j $(j = 1, 2, \cdots, p)$ in E_i
propagate evaluation results to all processors
decide if H_i is valid
propagate validation result of H_i to all other processors
let $B = B \bigcup H_i \bigcup \cdots \bigcup H_n$
let $E' = e : e \in E \, and \, B \models e$
let $E = E - E'$
goto start

Figure 4.1: Parallel Progol Algorithm

work in parallel to generalize the examples. Each process works on a partition of the dataset and executes the sequential CProgol program. By doing so, the search space is reduced by $1/p$ while the induced hypotheses remains the same.

The concept induced in one process is correct locally. But I have to make sure that it is also globally-correct. Since the information each process sees is partial, a mechanism must be provided to let each process have the knowledge of the entire dataset in some sense.

Figure 4.1 is the algorithm of PCProgol. It provide a way to check if a locally-correct concept is also globally-correct. For process i, B is the background knowledge, E_i is its subset of examples.

How to divide the dataset. In my approach, I divide the entire positive example set into p subsets and allocate one subset to each processor. In many cases, the size of positive examples is much bigger than the size of background knowledge and negative examples.

I have to find a way to deal with negative examples. To make sure the locally consistent concepts are also globally consistent, I keep a copy of the negative example set on each processor.

How to evaluate H. When inducing concepts on one processor, PCProgol uses several parameters to give the induced concept H a score relative to the local subset. An H with the highest score will be the locally-correct concept induced in this step. The score f of a candidate concept s is defined as follows:

$$f = Y - (N + C + R) \tag{4.1}$$

where

- Y = the number of positive examples correctly deducible from s
- N = the number of negative examples incorrectly deducible from s
- C = the length of concept s
- R = the number of further atoms to complete the induced concept

R is calculated by inspecting the output variables in the clause and determining whether they have been defined.

So f is a measure of how well a concept s explains all the positive examples, with preference to the shorter ones. The evaluation process will go through the entire

dataset once to give a candidate s a score f. In the worst case, it will consider all the clauses in order and the algorithm will look through the entire dataset many times to find a correct concept.

When all the processors have found their locally-correct concept H_i, they come to a synchronization point. At this point, each processor sends its locally-correct concept H_i to all other processors. After total exchange, each processor has a copy of all the concepts induced by all the processors during this step. Each processor evaluates these concepts and gives a score f to each H_i relative to its subset of data. Then there is a second round of total exchange - exchange of the score f. When one processor has collected all the scores from other processor for its H_i, it can give its H_i a global score and then decide if it is valid or not. So the total exchange of information provides a way for each processor to evaluate its locally-correct concepts against the whole dataset. Once the validation is made by all processors, there comes the third phase of total exchange. During this communication phase, each processor tells other processors whether its H_i is globally valid. If so, all processors will update their background knowledge with this H_i and delete redundant examples already covered by it. More than one globally-correct concept is usually induced in one big step.

4.2 Test Cases

Experiment platform. BSP can support both shared-memory and distributed-memory computing environments. In my experiment I built and ran PCProgol on two different machines. One is a 4-processor shared-memory SUN machine. The platform is :

Example	Number	k_s	k_r (4-process)	k_r (6-process)
Animal Classifier	4000	9	2	2
Chess Move Learner	4000	23	4	3
Game Ending Problem	2000	12	4	4

Table 4.1: Test cases and sequential performance

- Name: teaspoon.cs.queensu.ca
- Model: SUN Enterprise Server 3000.
- Processors: four Sparc processors, each one operating at 50 MHz and has a Sparc floating point processor.

The other is a 6-processor shared-memory SUN machine. The platform is :

- Name: zeus.caslab.queensu.ca
- Model: SUN Enterprise Server 3500.
- Processors: six UltraSparcII processors, each one operating at 336 MHz and has a Sparc floating point processor.

Though this program is developed and tested on SMP machines, this parallel approach can be transparently adapted for distributed-memory computing environments with the support of BSP.

There are three example sets provided in this chapter to test parallel Progol. They are shown in Table 4.1. The first test case is an animal classifier. In this case animal classification information is given as positive examples. The background knowledge is provided to describe the properties of one particular animal. The program tries to

form some general rules to classify an animal according to its properties. There are 4000 examples in this test case which contains some redundancy and similar examples.

The second test case is a chess move learner program. It learns legal chess moves. The moves of the chess pieces
Pieces = (King, Queen, Bishop, Knight and Rook) are learned from examples. Each example is represented by a triple from the domain
Piece * (Original-Rank * Original-File) * (Destination-Rank * Destination-File)
There are 4000 examples in this test case.

The third test case is a chess game-ending problem. It tries to form a rule to decide whether a chess ending with White King, White Rook and Black King is illegal when White is to move. Example positions are defined as
illegal(WKRank, WKFile, WRRank, WRFile, BKRank, BKFile)
There are 2000 examples in this test case.

The source file Types describes the categories of objectives in the world under consideration. Modes describes the relationship between objects of given types, and the form these atoms can take within a clause. The Examples section contains all the positive and negative examples.

Example 1: Animal Classifier

Types

Type provides information about the type of the object.

```
animal(dog).        animal(dolphin).
class(mammal).      class(fish).
covering(scales).   covering(feathers).
habitat(land).      habitat(water).
...
```

Modes

For the head of any general rule defining *class* I give the following head mode declarations

```
:- modeh(1,class(+animal,#class))?
```

which means class may have 2 arguments of type animal and class. A + sign indicates that the argument is an input variable. A # sign denotes a constant.For atoms in the body of a general rule, body mode declarations are given as follows:

```
:- modeb(1,has-gills(+animal))?
:- modeb(1,hascovering(+animal,#covering))?
:- modeb(1,haslegs(+animal,#nat))?
:- modeb(1,homeothermic(+animal))?
```

Examples

I give some examples of what animal belongs to what class.

```
class(eagle,bird).      class(bat,mammal).
class(dog,mammal).      class(bat,mammal).
class(eagle,bird).      class(ostrich,bird).
....
```

Background knowledge

```
hascovering(dog, hair). hascovering(dolphin, none).  ...
```

Example 2: Chess Move Learner

Types

```
piece(king).        piece(queen).
...
```

Modes

```
:- modeh(1,move(#piece,pos(+file,+rank),pos(+file,+rank)))?
:- modeb(1,rdiff(+rank,+rank,-nat))?
:- modeb(1,fdiff(+file,+file,-nat))?
```

Examples

There are some negative examples in this case. $:-$ is for negative examples.

```
move(king,pos(b,7),pos(c,6)).
move(bishop,pos(g,3),pos(e,1)).
move(queen,pos(e,6),pos(h,3)).
:- move(pawn,pos(g,3),pos(c,5)).
:- move(king,pos(h,2),pos(e,2)).
:- move(king,pos(e,2),pos(a,5)).
...
```

Background knowledge

The only background predicate used is symmetric difference, i.e.

diff(X,Y) = absolute difference between X and Y

Symmetric difference is defined separately on Rank and File.

```
rdiff(Rank1,Rank2,Diff) :-
rank(Rank1), rank(Rank2), Diff1 is Rank1-Rank2, abs(Diff1,Diff).
fdiff(File1,File2,Diff) :-
file(File1), file(File2), project(File1,Rank1), project(File2,Rank2), Diff1 is Rank1-Rank2,
abs(Diff1,Diff).
abs(X,X) :- X ≥ 0.
abs(X,Y) :- X < 0, Y is −X.
```

Example 3: Game Ending Problem

Types

```
rf(X) .- nat(X), 0=<X, X=<7.
```

Modes

```
:- modeh(1,illegal(+rf,+rf,+rf,+rf,+rf,+rf))?
:- modeb(1,adj(+rf,+rf))?
```

Examples

```
illegal(5,5,4,6,4,1).      illegal(5,6,7,5,7,5).
illegal(3,2,4,6,6,6).      illegal(2,1,6,1,2,0).
illegal(3,0,2,3,4,0).      illegal(6,2,5,1,6,1).
...
```

4.3 Test Results

For each test case I did the following experiments:

- run sequential algorithm on teaspoon
- run sequential algorithm on zeus
- run parallel algorithm on teaspoon with 4 processes
- run parallel algorithm on zeus with 4 processes
- run parallel algorithm on zeus with 6 processes

I collected the corresponding data, which are shown in the tables of this chapter. From this data I calculated the double speedup phenomenon observed in these 3 test cases, i.e., $p * cost_r < cost_s$ where $cost_r$ is the cost of one process in parallel version, and $cost_s$ is the cost of sequential version.

According to the formulae 3.8–3.11 derived from Chapter 3, the cost of selecting an example, generating a hypothesis from the most specific clause $\perp$, evaluating a candidate concept and retracing redundant examples in one subset should be $1/p$ of the sequential algorithm. Table 4.3 shows *SEL()*, *EVA()*, *RET()* values in the sequential algorithm. The first column shows the test case number and parameter name; the second and third columns show the values on teaspoon and zeus. Since zeus is a faster machine than teaspoon, the values on zeus are smaller. Table 4.4 shows *SEL()*, *EVA()*, *RET()* values in the parallel algorithm. The first column shows the test case number and parameter name; the second and third column show the values on teaspoon and zeus with 4 processes; the last column shows the values on zeus

with 6 processes. Variances were typically within 20 per cent. Please refer to Table 4.5 for detailed information. The test results shown in Table 4.3 and Table 4.4 do not totally match the above analysis in my experiment. I suppose this is due to the workload of the machine that is for public use, and the disk access time is affected by the hardware architecture. These values depend on size of dataset and machine speed. So they vary little among each big step. I repeat the experiments on the same machines four times during different time of the day to collect data. The result shown in Table 4.3 and Table 4.4 is the average value. The value of *GEN()* varies in different test cases depending on how a candidate concept is generated from $\perp$. When the dataset is big, the cost of *GEN()* is small compared to the disk access cost *EVA()*. The value of $\varepsilon(GEN(nm/p) + EVA(nm/p))$ is the most significant local computation cost in each big step.

Though the cost analysis given in these examples is in terms of execution time, it is easily adapted to the number of instruction cycles with the system parameters provided by BSPlib. Then the cost analysis can be applied universally and independent of particular machine architecture.

BSP system parameters on teaspoon and zeus are shown in Table 4.2. With the system parameters in hand, I can give the optimistic communication cost. The size of data r in one total communication is around 100 words. There are three total communications in one big step. The value of g on teaspoon is 4.1 flops/word, p is 4, s is 10.1 Mflops, and l is 118 flops. So $3 * (rpg + l)$ = 3*(100*4*4.1 + 118) flops = 5274 flops = 0.0005274 s. The value of g on zeus is 3.17 flops/word, p is 4, s is 44.5 Mflops, and l is 697.4 flops. So $3 * (rpg + l)$ = 3*(100*4*3.17 + 697.4) flops = 5896

flops = 0.00013 s.

Parameter	teaspoon	zeus	zeus
Number of processes	4	4	6
BSP parameter s	10.1 Mflops	44.5 Mflops	
BSP parameter l	118 flops	499 flops	
BSP parameter g	4.1 flops/32bit word	3.17 flops/32bit word	

Table 4.2: System Information

Value	teaspoon	zeus
Test Case 1: *SEL(nm)*	0.04 s	0.01 s
Test Case 1: *EVA(nm)*	0.60 s	0.13 s
Test Case 1: *RET(nm)*	0.60 s	0.13 s
Test Case 2: *SEL(nm)*	0.04 s	0.01 s
Test Case 2: *EVA(nm)*	1.51 s	0.13 s
Test Case 2: *RET(nm)*	1.51 s	0.13 s
Test Case 3: *SEL(nm)*	0.04 s	0.01 s
Test Case 3: *EVA(nm)*	0.60 s	0.07 s
Test Case 3: *RET(nm)*	0.60 s	0.07 s

Table 4.3: Mean SEL, EVA, and RET Values in Sequential Algorithm

Value	teaspoon	zeus	zeus
Processes	4	4	6
Test case 1 :*SEL(nm/p)*	0.02 s	0.01 s	0.01 s
Test case 1 :*EVA(nm/p)*	0.30 s	0.06 s	0.04 s
Test case 1 :*RET(nm/p)*	0.30 s	0.06 s	0.04 s
Test case 2 :*SEL(nm/p)*	0.02 s	0.01 s	0.01 s
Test case 2 :*EVA(nm/p)*	1.20 s	0.12s	0.08s
Test case 2 :*RET(nm/p)*	1.20 s	0.12 s	0.08 s
Test case 3 :*SEL(nm/p)*	0.02 s	0.01 s	0.01 s
Test case 3 :*EVA(nm/p)*	0.40 s	0.08 s	0.05 s
Test case 3 :*RET(nm/p)*	0.40 s	0.08 s	0.05 s

Table 4.4: Mean SEL, EVA, and RET Values in Parallel Algorithm

4.3.1 Test Result of Animal Classification.

Table 4.5 shows the test results in sequential algorithm. The concepts induced in each big step are shown. The value of *GEN(nm)+EVA(nm)* shows the cost to generate and evaluate one candidate concept, which is the most significant computational cost. The values shown in the table are average values. The range and number of data nodes I collected are also shown. The value of ε shows the number of candidate concepts generated in each step. The cost of each big step should be roughly equal to $\varepsilon\,(GEN(nm) + EVA(nm))$. The sequential algorithm takes 9 steps to generate all the rules.

In the parallel approach with 4 processors, the 4000 examples are divided into 4 subsets. Four processors induce the concept set on their subset of data in parallel. The number of big steps is reduced to 2. The test results on both machines is shown in Table 4.6. The concepts induced by different processors in one big step are shown in the table. The value of $\varepsilon\,(GEN(nm/p) + EVA(nm/p))$ shows the cost of local

Parameters	Value on teaspoon	Value on zeus
Big Step 1	class(A,fish) :- has-gills(A), hascovering(A,none).	
GEN(nm)+EVA(nm)	0.50s (0.48-0.52s, 31 nodes)	0.20s (0.16-0.24s, 31 nodes)
ε	31	31
examples retracted	863	863
subtotal	15.35s s	6.44 s
Big Step 2	class(A,reptile) :- habitat(A,land), habitat(A,water).	
GEN(nm)+EVA(nm)	0.72s (0.63-0.85s, 99 nodes)	0.13s (0.09-0.15s, 99 nodes)
ε	99	99
examples retracted	78	78
subtotal	72.84 s	13.12 s
Big Step 3	class(A,mammal) :- habitat(A,caves).	
GEN(nm)+EVA(nm)	0.94s (0.62-0.95s, 163 nodes)	0.13s (0.09-0.15s, 163 nodes).
ε	163	163
examples retracted	78	78
subtotal	160.22 s	21.16 s
Big Step 4	class(A,reptile) :- hascovering(A,scales), habitat(A,land).	
GEN(nm)+EVA(nm)	0.74s (0.63-0.93s, 57 nodes)	0.13s (0.09-0.15s, 57 nodes)
ε	57	57
examples retracted	156	156
subtotal	43.81 s	7.36 s
.....		
Big Step 7	class(A,bird) :- hascovering(A,feathers), habitat(A,land).	
GEN(nm)+EVA(nm)	0.54s (0.42-0.64s, 163 nodes)	0.13s (0.09-0.15s, 163 nodes)
ε	163	163
examples retracted	549	549
subtotal	89.04 s	15.34 s
Big Step 8	class(A,bird) :- hascovering(A,feathers).	
GEN(nm)+EVA(nm)	0.46s (0.38-0.56s, 99 nodes)	0.08s (0.06-0.10s, 99 nodes)
ε	99	99
examples retracted	156	156
subtotal	47.02 s	8.35 s
Big Step 9	class(A,mammal) :- hascovering(A,hair).	
GEN(nm)+EVA(nm)	0.52s (0.41-0.78s, 163 nodes)	0.10s (0.06-0.12s, 163 nodes)
ε	163	163
examples retracted	156	156
subtotal	87.38 s	15.62 s
Total cost	604.45 s	107.75 s

Table 4.5: Test case 1: result of sequential algorithm

computation on the processor which takes the longest time in one big step. The value of 3(rpg + l) shows the measured communication cost. In the parallel approach with 6 processors, the 4000 examples are divided into 6 subsets. The number of big steps is also 2. The test results is shown in Table 4.7.

Parameters	Value on teaspoon	Value on zeus
Big Step 1:	concept induced	
process 1 process 2 process 3 process 4	class(A,mammal) :- hascovering(A,hair). class(A,fish) :- has-gills(A), hascovering(A,none) . class(A,reptile) :- haslegs(A,4), habitat(A,water). class(A,bird) :- hascovering(A,f eathers).	
ε (*GEN(nm/p)+EVA(nm/p)*) 3(rpg + l) examples retracted subtotal	163*0.43s = 72.23 s 0.21 s 3765 75.69 s	163*0.06s = 9.98 s 0.02 s 3765 10.32 s
Big Step 2:	concept induced	
processes 1-4	class(A,reptile) :- not(has-gills(A)), hascovering(A,scales).	
ε (*GEN(nm/p)+EVA(nm/p)*) 3(rpg + l) examples retracted subtotal	57*0.41s = 23.65 0.2017 s 235 26.44 s	57*0.04s = 2.39s 0.01 s 235 2.71 s
Total parallel algorithm cost	105.86 s	13.05 s

Table 4.6: Test case 1: results of 4-process parallel algorithm

In the parallel algorithm with four processors, each processor induces a different concept in the first step. So at the end of the first big step, each process has learnt four valid concepts. In the second big step four processes induce only one concept. In parallel algorithm with six processors, there are five concepts induced in the first step. In the second big step six processes induce only one concept. Though there is redundancy, the overall performance is still greatly improved. Table 4.8 shows the

Big Step 1:	concept induced
process 1	class(A,mammal) :- hascovering(A,hair).
process 2	class(A,fish) :- has-gills(A), hascovering(A,none) .
process 3	class(A,reptile) :- haslegs(A,4), habitat(A,water).
process 4	class(A,bird) :- hascovering(A,f eathers).
process 5	class(A,reptile) :- hascovering(A,scales), habitat(A,land).
process 6	class(A,fish) :- has-gills(A), hascovering(A,none).
ε (*GEN(nm/p)*	
+ *EVA(nm/p)*)	163*0.05s = 9.68 s
3(rpg + l)	0.03 s
examples retracted	3765
subtotal	10.34 s
Big Step 2:	concept induced
processes 1-6	class(A,reptile) :- not(has-gills(A)), hascovering(A,scales).
ε (*GEN(nm/p)*	
+ *EVA(nm/p)*)	57*0.04s = 2.40s
3(rpg + l)	0.02 s
examples retracted	235
subtotal	2.91 s
Total parallel	
algorithm cost	13.25 s

Table 4.7: Test case 1: results of 6-process parallel algorithm

results of test case one. The costs of sequential algorithm and parallel algorithm with four and six processors on both machines are compared. The row $cost_r$ shows the average cost of one processor in parallel algorithm. The row $cost_s$ shows the average cost in sequential algorithm. A double speedup phenomenon is observed in this test case on both machines with different processor number, which is shown as $p * cost_r < cost_s$.

Parameters	teaspoon(4-process)	zeus(4-process)	zeus(6-process)
Number of examples	4000	4000	4000
k_r	2	2	2
k_s	9	9	9
$cost_r$	105.86 s	13.05 s	13.25 s
$cost_s$	604.65 s	107.75 s	107.75 s
$cost_r * p$	423.44 s	52.5 s	79.50 s

Table 4.8: Test case 1: comparison of sequential and parallel algorithm

4.3.2 Test Result of Chess Move Learner.

Table 4.9 shows the test results for the sequential algorithm. The concepts induced in each big step are shown. The value of *GEN(nm)+EVA(nm)* shows the cost to generate and evaluate one candidate concept. The values shown in the table are average values. The range and number of data nodes I collected are also shown. The value of ε shows the number of candidate concepts generated in each step. The cost of each big step should be roughly equal to $\varepsilon\,(GEN(nm)+EVA(nm))$. The sequential algorithm takes 23 steps to generate all the rules.

In the parallel approach with 4 processors, the 4000 examples are divided into 4 subsets. Four processors induce the concept set on their subset of data in parallel. The number of big steps is reduced to 4. The test results on both machines is shown in Table 4.10. The concepts induced by different processors in one big step are shown in the table. The value of $\varepsilon\,(GEN(nm/p) + EVA(nm/p))$ shows the cost of local computation on the processor which takes the longest time in one big step. The value of 3(rpg + l) shows the measured communication cost. In the parallel approach with 6 processors, the 4000 examples are divided into 6 subsets. The number of big steps is further reduced to 3. The test results is shown in Table 4.11.

This test case shows the scalability of the parallel algorithm. The parallel algorithm with six processors induces concepts in a quicker way than with four processors. So the total cost of the parallel algorithm with six processors is less than the cost with four processors. Table 4.12 shows the results of test case two. The costs of sequential algorithm and parallel algorithm with four and six processors on both machines are compared. Though there is redundancy, i.e. the concepts induced in last big step

Parameters	Value on teaspoon	Value on zeus
Big Step 1	move(bishop,pos(A,B),pos(C,D)) :- rdiff(B,D,2), fdiff(A,C,2).	
GEN(nm)+EVA(nm)	1.57s (0.90-1.81s, 22 nodes)	0.13s (0.09-0.15s, 22 nodes)
ε	22	22
examples retracted	159	159
subtotal	37.75 s	2.86 s
Big Step 2	move(queen,pos(A,B),pos(C,D)) :- rdiff(B,D,7), fdiff(A,C,7).	
GEN(nm)+EVA(nm)	1.51s (1.16-1.96s, 22 nodes)	0.12s (0.09-0.15s, 22 nodes)
ε	22	22
examples retracted	14	14
subtotal	35.99 s	2.66 s
Big Step 3	move(bishop,pos(A,B),pos(C,D)) :- rdiff(B,D,1), fdiff(A,C,1).	
GEN(nm)+EVA(nm)	1.49s (1.25-2.03s, 22 nodes)	0.12s (0.09-0.15s, 22 nodes)
ε	22	22
examples retracted	214	214
subtotal	35.52 s	2.59 s
Big Step 4	move(rook,pos(A,B),pos(C,B)) :- fdiff(A,C,5).	
GEN(nm)+EVA(nm)	1.59s (1.31-1.87s, 34 nodes)	0.12s (0.09-0.15s, 34 nodes)
ε	34	34
examples retracted	72	72
subtotal	59.12 s	4.04 s
Big Step 5	move(queen,pos(A,B),pos(A,C)).	
GEN(nm)+EVA(nm)	1.63s (1.43-1.96s, 34 nodes)	0.12s (0.09-0.15s, 34 nodes)
ε	34	34
examples retracted	502	502
subtotal	62.12 s	4.23 s
Big Step 6	move(rook,pos(A,B),pos(A,C)).	
GEN(nm)+EVA(nm)	1.57s (1.34-2.21s, 34 nodes)	0.12s (0.08-0.14s, 34 nodes)
ε	34	34
examples retracted	556	556
subtotal	57.84 s	4.00 s
.....		
Big Step 23	move(bishop,pos(A,B),pos(C,D)) :- rdiff(B,D,6), fdiff(A,C,6).	
GEN(nm)+EVA(nm)	0.52s (0.41-0.78s, 22 nodes)	0.10s (0.06-0.12s, 22 nodes)
ε	22	22
examples retracted	14	14
subtotal	29.85 s	2.29 s
Total cost	1062.95 s	77.93 s

Table 4.9: Test case 2: results of sequential algorithm

Parameters	Value on teaspoon	Value on zeus
Big Step 1:	concept induced	
process 1	move(king,pos(A,B),pos(C,D)) :- rdiff(B,D,1), fdiff(A,C,1).	
process 2	Invalid	
process 3	move(bishop,pos(A,B),pos(C,D)) :- rdiff(B,D,E), fdiff(A,C,E).	
process 4	move(rook,pos(A,B),pos(C,B)).	
ε (*GEN(nm/p)+EVA(nm/p)*)	51*0.56 s = 29.37 s	51*0.12s = 6.27 s
3(rpg + l)	0.0998 s	0.01 s
examples retracted	848	848
subtotal	32.60 s	6.58 s
Big Step 2:	concept induced	
process 1	move(queen,pos(A,B),pos(C,D)) :- rdiff(B,D,E), fdiff(A,C,E).	
process 2	move(queen,pos(A,B),pos(C,B)).	
process 3	move(knight,pos(A,B),pos(C,D)) :- rdiff(B,D,1), fdiff(A,C,2).	
process 4	Invalid	
ε (*GEN(nm/p)+EVA(nm/p)*)	49*1.50s = 73.98	22*0.12s = 2.62s
3(rpg + l)	0.0406 s	0.03 s
examples retracted	1720	1720
subtotal	77.67 s	2.76 s
Big Step 3:	concept induced	
process 1	move(queen,pos(A,B),pos(A,C)).	
process 2	move(knight,pos(A,B),pos(C,D)) :- rdiff(B,D,2), fdiff(A,C,1).	
process 3	move(king,pos(A,B),pos(A,C)) :- rdiff(B,C,1).	
process 4	move(rook,pos(A,B),pos(A,C)).	
ε (*GEN(nm/p)+EVA(nm/p)*)	34*1.68s = 56.42 s	29*0.11s = 3.22 s
3(rpg + l)	0.2706 s	0.03 s
examples retracted	1344	1344
subtotal	59.54 s	3.29 s
Big Step 4:	concept induced	
processes 1-4	move(king,pos(A,B),pos(C,B)) :- fdiff(A,C,1).	
ε (*GEN(nm/p)+EVA(nm/p)*)	34*1.65s = 56.42 s	28*0.08s = 2.33s
3(rpg + l)	0.404 s	0.03 s
examples retracted	88	88
subtotal	58.35s s	2.38 s
Total parallel algorithm cost	237.84 s	15.02 s

Table 4.10: Test case 2: results of 4-process parallel algorithm

Big Step 1:	concept induced
process 1	move(king,pos(A,B),pos(C,D)) :- rdiff(B,D,1), fdiff(A,C,1).
process 2	move(queen,pos(A,B),pos(C,D)) :- rdiff(B,D,E), fdiff(A,C,E).
process 3	move(bishop,pos(A,B),pos(C,D)) :- rdiff(B,D,E), fdiff(A,C,E).
process 4	move(rook,pos(A,B),pos(C,B)).
process 5	move(king,pos(A,B),pos(C,D)) :- rdiff(B,D,1), fdiff(A,C,1).
process 6	move(king,pos(A,B),pos(C,B)) :- fdiff(A,C,1).
ε (*GEN(nm/p)* + *EVA(nm/p)*)	51*0.07s = 3.73 s
3(rpg + l)	0.04 s
examples retracted	2180
subtotal	4.08 s
Big Step 2:	concept induced
process 1	move(queen,pos(A,B),pos(A,C)).
process 2	move(queen,pos(A,B),pos(C,B)).
process 3	move(knight,pos(A,B),pos(C,D)) :- rdiff(B,D,1), fdiff(A,C,2).
process 4	move(rook,pos(A,B),pos(A,C)).
process 5	move(queen,pos(A,B),pos(A,C)).
process 6	move(knight,pos(A,B),pos(C,D)) :- rdiff(B,D,2), fdiff(A,C,1).
ε (*GEN(nm/p)* + *EVA(nm/p)*)	22*0.067s = 1.51s
3(rpg + l)	0.04 s
examples retracted	1720
subtotal	1.67 s
Big Step 3:	concept induced
processes 1-6	move(king,pos(A,B),pos(A,C)) :- rdiff(B,C,1).
ε (*GEN(nm/p)* + *EVA(nm/p)*)	29*0.047s = 1.36 s
3(rpg + l)	0.05 s
examples retracted	88
subtotal	1.99 s
Total parallel algorithm cost	7.74 s

Table 4.11: Test case 2: results of 6-process parallel algorithm

are th same, the overall performance is still greatly improved. The row $cost_r$ shows the average cost of one processor in parallel algorithm. The row $cost_s$ shows the average cost in sequential algorithm. A double speedup phenomenon is observed in this test case on both machines with different processor number, which is shown as $6 * cost_6 < 4 * cost_4 < cost_s$.

Parameters	teaspoon (4-process)	zeus (4-process)	zeus (6-process)
Number of examples	4000	4000	4000
k_r	4	4	3
k_s	23	23	23
$cost_r$	237.84 s	15.02	7.74 s
$cost_s$	1062.95 s	77.93 s	77.93 s
$cost_r * p$	951.36 s	60.08 s	46.44 s

Table 4.12: Test case 2: comparison of sequential and parallel algorithm

4.3.3 Test Result of Chess Game Ending Illegal Problem

Table 4.13 shows the test results for the sequential algorithm. The concepts induced in each big step are shown. The value of *GEN(nm)+EVA(nm)* shows the cost to generate and evaluate one candidate concept. The values shown in the table are average values. The range and number of data nodes I collected are also shown. The value of ε shows the number of candidate concepts generated in each step. The cost of each big step should be roughly equal to $\varepsilon\,(GEN(nm)+EVA(nm))$. The sequential algorithm takes 12 steps to generate all the rules.

In the parallel approach with 4 processors. the 2000 examples are divided into 4 subsets. Four processors induce the concept set on their subset of data in parallel. The number of big steps is reduced to 4. The test results on both machines is shown in Table 4.14. In the parallel approach with 6 processors, the 2000 examples are divided into 6 subsets. The number of big steps is also. The test results is shown in Table 4.15. As in test case 1 and 2, different processors induce some redundant concepts. But the overall performance is improved.

Table 4.16 shows the results of test case three. The costs of sequential algorithm and parallel algorithm with four and six processors on both machines are compared. The row $cost_r$ shows the average cost of one processor in parallel algorithm. The row $cost_s$ shows the average cost in sequential algorithm. A double speedup phenomenon is observed in this test case on both machines with four processors. However, the parallel algorithm with six processors does not show such a phenomenon, though $cost_r$ with six processors is less than $cost_r$ with four processors. This is partly due to the example set which, in this test case, does not show enough scalability. So

Parameters	Value on teaspoon	Value on zeus
Big Step 1	illegal(A,A,B,C,B,D) :- adj(A,B), adj(A,C).	
GEN(nm)+EVA(nm)	0.70s (0.50-1.89s, 248 nodes)	0.08s (0.06-0.15s, 248 nodes)
ε	248	248
examples retracted	16	16
subtotal	178.01 s	20.79 s
Big Step 2	illegal(A,B,C,D,C,A) :- adj(D,A), adj(B,C).	
GEN(nm)+EVA(nm)	0.50s (0.43-0.85s, 633 nodes)	0.07s (0.05-0.15s, 633 nodes)
ε	633	633
examples retracted	20	20
subtotal	336.95 s	42.33 s
Big Step 3	illegal(A,B,C,D,E,D) :- adj(A,B), adj(A,C).	
GEN(nm)+EVA(nm)	0.57s (0.39-0.95s, 212 nodes)	0.07s (0.05-0.15s, 212 nodes).
ε	212	212
examples retracted	120	120
subtotal	121.62 s	15.05 s
Big Step 4	illegal(A,B,C,D,E,F) :- adj(A,E), adj(B,F).	
GEN(nm)+EVA(nm)	0.69s (0.63-0.93s, 248 nodes)	0.08s (0.06-0.15s, 248 nodes)
ε	248	248
examples retracted	680	680
subtotal	171.47 s	19.89 s
.....		
Big Step 12	illegal(A,B,C,D,E,D).	
GEN(nm)+EVA(nm)	0.37s (0.20-0.78s, 165 nodes)	0.04s (0.02-0.06s, 165 nodes)
ε	165	165
examples retracted	96	96
subtotal	61.14 s	7.31 s
Total cost	1239.88 s	150.58 s

Table 4.13: Test case 3: results of sequential algorithm

Parameters	Value on teaspoon	Value on zeus
Big Step 1:	concept induced	
process 1	illegal(A,B,C,D,E,F) :- adj(E,A), adj(B,F).	
process 2	Invalid	
process 3	illegal(A,B,C,D,C,E).	
process 4	illegal(A,B,C,D,C,E) :- adj(C,E).	
ε (*GEN(nm/p)+EVA(nm/p)*)	121*0.44 s = 53.49 s	51*0.12s = 7.54 s
3(rpg + l)	0.093 s	0.04 s
examples retracted	1305	1305
subtotal	56.13 s	6.58 s
Big Step 2:	concept induced	
process 1	illegal(A,B,C,D,D,D) :- adj(A,C).	
process 2	illegal(A,B,C,C,D,C) :- adj(A,D), adj(C,D).	
process 3	illegal(A,B,C,D,E,D) :- adj(A,C), adj(D,E).	
process 4	illegal(A,B,C,D,E,D) :- adj(A,C), adj(D,E).	
ε (*GEN(nm/p)+EVA(nm/p)*)	267*0.26s = 70.56	267*0.03s = 8.31s
3(rpg + l)	0.262 s	0.02 s
examples retracted	153	153
subtotal	73.95 s	8.71 s
Big Step 3:	concept induced	
process 1	illegal(A,B,A,B,C,D).	
process 2	illegal(A,B,C,D,E,D) :- adj(D,E).	
process 3	illegal(A,B,C,D,E,D) :- adj(D,E).	
process 4	illegal(A,B,C,D,E,D) :- adj(D,E).	
ε (*GEN(nm/p)+EVA(nm/p)*)	357*0.25s = 91.81 s	357*0.03s = 11.04 s
3(rpg + l)	0.216 s	0.03 s
examples retracted	195	195
subtotal	92.52 s	11.35 s
Big Step 4:	concept induced	
process 1	illegal(A,B,C,D,E,D).	
process 2	illegal(A,B,C,D,E,D) :- adj(A,C).	
process 3	illegal(A,B,C,D,E,D).	
process 4	illegal(A,B,C,D,E,D).	
ε (*GEN(nm/p)+EVA(nm/p)*)	268*0.24s = 65.28 s	268*0.03s = 7.82s
3(rpg + l)	0.20 s	0.04 s
examples retracted	348	88
subtotal	68.89 s	8.02 s
Total parallel algorithm cost	293.07 s	36.14 s

Table 4.14: Test case 3: results of 4-process parallel algorithm

Big Step 1:	concept induced
process 1	Invalid
processes 2-4	illegal(A,B,C,D,C,E).
process 5	illegal(A,B,C,D,E,D) :- adj(D,C), adj(C,E).
process 6	illegal(A,B,C,D,E,D) :- adj(D,C).
ε (*GEN(nm/p)+EVA(nm/p)*)	267*0.03s = 8.78 s
3(rpg + l)	0.03 s
examples retracted	680
subtotal	9.27 s
Big Step 2:	concept induced
process 1	illegal(A,B,C,D,E,D) :- adj(A,C), adj(E,D).
process 2	illegal(A,B,C,D,E,D) :- adj(B,D).
process 3	illegal(A,B,C,D,E,D) :- adj(A,C), adj(D,E).
process 4	illegal(A,B,C,D,E,B) :- adj(A,E).
process 5	illegal(A,B,C,A,D,E) :- adj(A,D), adj(B,E).
process 6	illegal(A,B,C,D,E,B) :- adj(A,E).
ε (*GEN(nm/p)+EVA(nm/p)*)	248*0.03s = 6.24s
3(rpg + l)	0.01 s
examples retracted	720
subtotal	6.66 s
Big Step 3:	concept induced
processes 1,3,4	illegal(A,B,C,D,D,E) :- adj(A,D), adj(B,E).
process 2	illegal(A,B,C,D,E,D).
process 5	illegal(A,B,C,D,E,D) :- adj(A,C).
process 6	illegal(A,B,C,D,E,D) :- adj(E,D).
ε (*GEN(nm/p)+EVA(nm/p)*)	215*0.02s = 4.23 s
3(rpg + l)	0.01 s
examples retracted	340
subtotal	4.55 s
Big Step 4:	concept induced
process 1	illegal(A,B,A,B,C,D).
process 2	illegal(A,B,A,B,C,D) :- adj(B,D).
process 3-6	illegal(A,B,A,B,C,D).
ε (*GEN(nm/p)+EVA(nm/p)*)	462*0.017s = 7.66 s
3(rpg + l)	0.009 s
examples retracted	260
subtotal	7.97 s
Total parallel algorithm cost	28.44 s

Table 4.15: Test case 3: results of 6-process parallel algorithm

processors waste time doing redundant work when processors becomes large.

Parameters	teaspoon (4-process)	zeus (4-process)	zeus (6-process)
Number of examples	2000	2000	2000
k_r	4	4	4
k_s	23	23	23
$cost_r$	293.07 s	36.14	28.44 s
$cost_s$	1239.88 s	150.58 s	150.58 s
$cost_r * p$	1172.28 s	144.56 s	170.64 s

Table 4.16: Test case 3: comparison of sequential and parallel algorithm

4.3.4 Summary.

Super-linear speedup is observed in all these test cases. I might expect the parallel implementation using p processors to take time $cost_s/p$ if I ignore the communication overhead. But here in the experiment it executes even faster due to the information exchange between processors and reduction of subsequent work. There is a big performance improvement with a small p. Though I did the experiments on a p-processor SMP machine, I believe that the parallel ILP algorithm is scalable given that each subset of data is still big enough to induce correct concepts. And the double speedup phenomenon will be observed with a larger processor set.

Chapter 5

Conclusion

In this thesis I studied the use of inductive logic to generate concepts from very big datasets in parallel. I use p processors to do the data-mining job, each on a subset of the full dataset. A set of concepts are generated from disjoint subsets of the full dataset used for mining. The distributed concept sets are total exchanged and evaluated before merging the valid ones into the final concept set. The final set of concepts is free of conflicts and has accuracy equivalent to a set of rules developed from the full dataset. The disk I/O access cost for each processor will be reasonably reduced by $1/p$.

Since each processor learns concepts independently on its subset, there are some issues that I have discussed in this thesis:

- How to secure the accuracy of induced theory on smaller datasets;
- How to deal with negative examples;
- How to reduce communication overhead; and

- How to avoid redundant work by individual processes.

I presented a parallel ILP data-mining algorithm using the BSP model and gave its cost analysis. I implemented a parallel version of a core ILP system – Progol – using C with the support of Oxford BSPlib. I developed several different test cases to show typical speedup. With all the test results, a double speedup phenomenon was observed which greatly improved the performance of ILP data-mining algorithm.

From the analysis of the parallel ILP data mining algorithm and the test results of parallel Progol, I can draw the conclusion that the benefits of the performance of parallel computing for ILP data mining is obvious. Though the cost measures in the implementation is not complete accurate and the parallel version of Progol has its limitation, they are expressive enough to show that even modestly parallel implementations of ILP algorithm can achieve significant performance gains. The following is what I discovered in my study:

First, inductive logic programming employs first-order structural representations, which generalizes attribute-value representations, as examples now may consist of multiple tuples belonging to multiple tables. These representations can succinctly represent a much larger class of concepts than propositional representations and have demonstrated a decided advantages in some problem domains [19]. By using first-order logic as the knowledge representation for both hypotheses and observations, inductive logic programming may overcome some major difficulties faced by other data-mining systems. ILP inheritates well-established theories, algorithms and tools from computational logic. Background knowledge helps in restricting the hypothesis search and is a key factor for incremental and iterative learning.

Second, the BSP model provides a simple way to implement a parallel ILP data-mining system and gives a relative accurate cost model based on counting computations, data access, and communication. Based on BSP model, I have confidence to give a relative accurate cost analysis to the test results.

Third, replicated implementation is shown to be a simple, yet powerful, approach to parallel ILP system design. Independent search is simple and works well for minimization problems. However, it does not divide the dataset, so it cannot reduce the disk access. Therefore, it is not suitable for problems with huge dataset. The fine-grained parallelism in parallelized approaches requires more communication, so I do not use this approach in out parallel ILP data-mining algorithm. The replicated approach is often the best way for parallelizing ILP data-mining applications. Previous work in [27] shows that the replicated approach gives the best performance improvement among all these three approaches introduced above. It gives a way for the algorithm to exploit collective knowledge quickly. The parallel algorithm exchanges information after each phase. The knowledge gained by one processor in a step will be exchanged with all other processors during the end of that step. In this way, once the algorithm has found a concept that can explain part of the data, it does not need to examine that part again. So there is less work for the next phase. A double speedup phenomenon is observed in this parallel algorithm, as shown is Table 5.1, 5.2, 5.3.

Double Speedup 1

Example	Sequential cost	Processors	Parallel cost	Parallel cost*4
Animal	604.65s	4	105.86s	423.44s
Chess Move	1062.95s	4	237.84s	951.36s
Game Ending	1239.88s	4	293.07s	1172.28s

Table 5.1: Double speedup on teaspoon with 4 processors

Double Speedup 2

Example	Sequential cost	Processors	Parallel cost	Parallel cost*4
Animal	105.86 s	4	13.05 s	52.50 s
Chess Move	77.93 s	4	15.02 s	60.08 s
Game Ending	150.58 s	4	36.14 s	144.56 s

Table 5.2: Double speedup on zeus with 4 processors

Double Speedup 3

Example	Sequential cost	Processors	Parallel cost	Parallel cost*4
Animal	105.86s	6	13.25 s	79.50 s
Chess Move	77.93 s	6	7.74 s	44.64 s
Game Ending	150.58 s	6	28.44 s	170.64 s

Table 5.3: Double speedup on zeus with 6 processors

Finally, though my test results are obtained from 4 and 6 processor SMP machines, it is reasonable to assume the scalability of this parallel approach to modest number of processes. Since the communication overhead is small, the parallel ILP algorithm will work well with more processors provided that each subset of data on one processor is big enough to induce accurate concepts.

Bibliography

[1] R. Agrawal and J. Shafer. Parallel mining of association rules: Design, implementation and experience. Technical Report RJ10004, IBM Research Report, February 1996.

[2] M. Besch and H.W. Pohl. How to simulate artificial neural networks on large scale parallel computers exploiting data parallelism and object orientation. Technical Report TR-94022, GMD FIRST Real World Computing Laboratory, November 1994.

[3] M. Besch and H.W. Pohl. Flexible data parallel training of neural networks using MIMD computers. In *Third Euromicro Workshop on Parallel and Distributed Processing*, January 1995.

[4] P.S. Bradley, U.M. Fayyad, and O.L. Mangasarian. Mathematical programming for data mining: Formulations and challenges. *INFORMS Journal of Computing*, 11:217–238, 1999.

[5] I. Bratko and S. Muggleton. Applications of inductive logic programming. *Communications of the ACM*, 38(11):65–70, 1995.

[6] S.H.N. Cheung. Data mining: From statistics to inductive logic programming. Technical report, Department of Computer Science, Erasmus University of Rotterdam, November 1996.

[7] S.H.N. Cheung. *Foundations of Inductive Logic Programming.* Springer, 1997.

[8] J.M.D. Hill D.B. Skillicorn. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, November Fall, 1997.

[9] P. Finn, S. Muggleton, D. Page, and A. Srinivasan. Pharmacophore discovery using the inductive logic programming system Progol. *Machine Learning*, 30:241–271, 1998.

[10] P. Frasconi, M. Gori, and G. Soda. Daphne: Data parallelism neural network simulator. *International Journal of Modern Physics C*, 1992.

[11] Y. Freund and R. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the 13th International Conference on Machine Learning*, pages 148–156, 1996.

[12] V. Gaede and O. Günther. Survey on multidimensional access methods. Technical Report ISS-16, Institut für Wirtschaftsinformatik, Humboldt Universität zu Berlin, August 1995. **www.wiwi.hu-berlin.de/~gaede/survey.rev.ps.Z.**

[13] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[14] G.H. Golub and C.F. van Loan. *Matrix Computations.* Johns Hopkins University Press, 3rd edition, 1996.

[15] G. Gonnet. Unstructured data bases or very efficient text searching. In *ACM Principles of Database Systems*, pages 117–124, Atlanta, Georgia, 1983.

[16] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 47–57, June 1984.

[17] E.-H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *ACM-SIGMOD Internation Conference on Management of Data*, May 1997.

[18] M.V. Joshi, G. Karypis, and V. Kumar. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In *Proceedings of IPPS/SPDP'98*, pages 573–580, 1998.

[19] K. Nigan M. Craven, S. Slattery. First-order learning for web mining. In *proceedings of the 10th European Conference on Machine Learning*, 1998.

[20] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[21] S. Muggleton. Inductive logic programming: theory and method. *Journal of Logic Programming*, 19:20, 1994.

[22] S. Muggleton. Inverse entailment and Progol. *New Generation Computing Systems*, 13:245–286, 1995.

[23] S. Muggleton. Inductive logic programming: issues, results and the LLL challenge. *Artificial Intelligence*, 114(1–2):283–296, December 1999.

[24] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[25] D.A. Pomerleau, G.L. Gusciora, D.L. Touretzky, and H.T. Kung. Neural network simulation at Warp speed: How we got 17 million connections per second. In *IEEE International Conference on Neural Networks*, July 1988.

[26] J.R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan-Kaufmann, 1993.

[27] R.O. Rogers and D.B. Skillicorn. Using the BSP cost model to optimize parallel neural network training. *Future Generation Computer Systems*, 14:409–424, 1998.

[28] J. Schafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proceedings of VLDB22*, Mumbai, India, 1996.

[29] D.B. Skillicorn. Strategies for parallel data mining. *IEEE Concurrency*, 7(4):26–35, October 1999.

[30] H. Toivonen. Discovery of frequent patterns in large data collections. Technical Report A-1996-5, Department of Computer Science, University of Helsinki, 1996.

[31] L.G. Valiant. *Oxford Parallel - BSP Model.* World Wide Web, 1997. **http://oldwww. comlab.ox.ac.uk/oucl/oxpara/bsp/bspmodel.htm** .

[32] N.B. Šerbedžija. Simulating artificial neural networks on parallel architectures. *Computer*, 29, No.3:56–63, 1996.

[33] I. Weber. *ILP* systems on the ilp-net systems repository. Technical report, Department of Computer Science, University of Stuttgart, Germany, 1996.

[34] M. Whitbrock and M. Zagha. An implementation of backpropagation learning on GF11, a large SIMD parallel computer. *Parallel Computing*, 14:329–346, 1990.

Appendix A

PCProgol Implementation

Oxford BSPlib is the platform used to implement the parallel version of CProgol. I made the necessary modifications to CProgol 4.4 to make it work in parallel.

At the beginning of the main() function, I call bsp_begin(int process_number) to start p processes. The number of processor can be modified as a parameter. Each process needs to be allocated to a processor. If more than one process is allocated to one processor the performance will be greatly affected due to the barrier synchronization. Each process will get its process ID by bsp_pid(). In this way I can tell which process is inducing concepts. At the end of the program bsp_end() is called to terminate the program. Function c_doall() will perform all the induction procedures describe in the parallel algorithm.

The main function starts:

```
main() {
    // BSP Begin, X = number of processes = number of processors
    bsp_begin(X);
    // get my process ID
    pid = bsp_pid();
```

```
    // Analyze command line parameters
    checkargs(argc,argv,envp);
    // Initialise built-in predicates
    I_init();
    // Begin induction process
    c_doall(fileroot_in,fileroot_out);
    // close all files
    c_close();
    return(1l);
    // BSP End
    bsp_end();
}
```

In CProgol, the big loop structure is implemented in the procedure c_sat(). I modified the big loop structure in c_sat() to make it work in parallel on several processors. c_sat() is the core procedure which does top-down search, asserts result if compressive and does theory reduction. The whole structure in the PCProgol will be made clear once I introduce the function of c_sat().

c_sat() first declares local variables. Some of these variables are used for BSP communication. A function cputime() is called to record computation and communication cost.

```
PREDICATE
  c_sat(cclause,nex)
  //DECLARE LOCAL VARIABLES
  //Start recording computation time
  start = cputime();
```

Once all the local variables are allocated, bsp_push_reg() is called to register necessary variable for communication. A synchronization function bsp_sync() is then called to make it happen.

```
/*register variables for BSP communication*/
     bsp_push_reg(concept,sizeof(char)*MAXMESS*X);
/*synchronization point*/
     bsp_sync();
```

ct_sat() and cl_symreduce() are then called for generating the most specific clause for the example selected.

```
//generate the most specific clause
    if (hypothesis = ct_ sat(cclause,atoio,otoa,&head))
        cl_symreduce(&hypothesis,atoio,head),
        outlook=r_outlook(hypothesis,head,otoa,atoio);
        vdomains=r_vdomains(otoa,atoio);
        if(verbose>=2)
         fprintf(tty_file- >file,'Most specific clause is:');
         cl_print(hypothesis);
```

Function r_search() will search in the hypothesis space to find a locally-correct hypothesis. If a successful hypothesis is found, then bsp_put() is called to send this hypothesis to all other processes. It is followed by a synchronization function call.

```
// search for locally-correct concept
      r_search(&hypothesis,atoio,otoa,outlook,vdomains,fnex);
      if(hypothesis&& !L_EMPTYQ(hypothesis))
       cl_unflatten(hypothesis);
       if(verbose>=1)
        fprintf(tty_file- >file,'Result of search is:');
        cl_print(hypothesis);

       result=TRUE;

      else
       fprintf(tty_file- >file,'[No compression]');
       result=FALSE;

      // propagate hypothesis to other processes
      for (i = 0;i<X;i++)
       bsp_put(i,hypothesis,receive,0,sizeof(char)*MAXMESS);

      bsp_sync();
```

The size of data in the first round of total exchange is a character string. Its size is defined by the macro MAXMESS to be 40 characters in PCProgol. After the global synchronization, each process gets all the hypotheses generated in this step. Then it will perform the evaluation. It will get the number of positive examples covered and

the number of negative examples wrongly covered by one hypothesis relative to its local example set.

Once each processes get all the p and n values for all the hypothesis, the score f will be calculated for that hypothesis. There will be a second-round total exchange. This time only the integer value of f are exchanged.

```
// use BSP model to get other processes' hypotheses and evaluate them
      for(int i=0; i<x,i++)
        if (pid ! = i)
          ITEM c1,call=d_gcpush(c1=i_copy(re_hyp[i]));
          LIST *end=cl_push(re_hyp[i]);
          PREDICATE negq=(PSYM(HOF((LIST)I_GET(re_hyp[i]))));
          p[i]=(int)cl_pcoverage(call,L_GET(*end));
          if (r_posonly())
            n[i] = (int) cl_dcoverage(call,fnex);
          else
            n[i] = (int) cl_ncoverage(negq,call,L_GET(*end));

        f[i] = get_score(p[i],n[i])4;

      for(i = 0;i<X;i++)
        if(pid != i )
          for(j=0;j<X;j++) bsp_put(i, f[i], rec_f[j],0,sizeof(int));
      bsp_sync();
```

According to f value from each processes, one process can decide if the hypothesis generated in this step is valid or not. And then during the third round communication the boolean value of validate will be total exchanged.

```
// Test if the hypothesis is valid or not.
     Validate = validate_test(f[pid]);
// propagate validation result to other processes
     for( i=0;i<4;i++)
       if(pid!=i)
         bsp_put(i,validate,go,pid*sizeof(int),sizeof(int));
     bsp_sync();
```

After the third round communication, all the valid hypothesis induced in this step will be updated to background knowledge and all the redundant examples will be retracted from each subsets.

```
for(i=0;i<X,i++)
   if(pos&&cover) c_updbsamp(psym,cclause[i]);
     cl_assert(cclause1,FALSE,TRUE,TRUE,FALSE,(ITEM)NULL);
     i_delete(cclause1);
     d_treduce(psym);
```

The total communication cost is small because the size of data to be exchanged is small.