A DATA PARALLEL IMPLEMENTATION OF AN INTELLIGENT REASONING SYSTEM*

KEVIN M. LIVINGSTON AND JENNIFER SEITZER

Computer Science Department, University of Dayton, 300 College Park, Dayton, OH 45469-2160
livingkm@flyernet.udayton.edu and seitzer@cps.udayton.edu

Abstract. We present an implementation of a data parallel system. A sequential knowledge-based deductive and inductive system, INDED (pronounced "indeed"), is transformed into a parallel system. In this parallel system the learning algorithm, the fundamental component of the induction engine, is distributed among many processors. The parallel system is implemented with a master node and several worker nodes. The master node is responsible for coordinating the activity of the worker nodes, and organizing the overall learning process. All the worker nodes share the processing of the basic induction algorithms and report their results to the master node. The goal of the data parallel system is to produce, more efficiently, rules that are equal to or better than those produced by the serial system. In this paper, we present the architecture of the parallel version of INDED, and comparison results involving execution speeds and quality of generated rules of the new parallel system to those of the serial system.

Key Words. data parallel, inductive logic programming (ILP), distributed reasoning systems

1. INTRODUCTION

Knowledge discovery in databases has been defined as the nontrivial process of identifying valid, novel, potentially useful, and understandable patterns in data [PSF91]. Because knowledge discovery is time consuming and space intensive, we are experimenting with distributing this endeavor. In this work, we are augmenting a data parallel implementation of system INDED that was described in [ALS00]. The original data parallel implementation involves decomposition and distribution of the background knowledge base, a main input to the induction engine. In this paper, along with the background knowledge base, we are distributing a set of constants among worker nodes. The fundamental operations of the learning algorithm are predicate ranking and position naming. In [ALS00], predicate ranking is performed in parallel among many worker nodes. In this paper, we further the distribution by parallelizing the position naming as well. The code being executed is identical among the worker nodes, however, each worker has a different set of data to process.

2. SYSTEM INDED

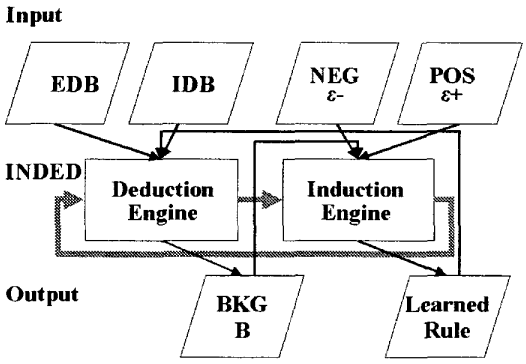
System INDED is a knowledge discovery system that uses inductive logic programming (ILP) [LD94] as its discovery technique. Inductive logic programming (ILP) is a research area in artificial intelligence which attempts to attain some of the goals of machine learning while using the techniques, language, and methodologies of logic programming. Some of the areas to which ILP has been applied are data mining, knowledge acquisition, and scientific discovery [LD94] . The goal of an inductive logic programming system is to output a rule which covers (entails) an entire set of positive observations, or examples, and excludes or does not cover a set of negative examples [Mug92] . This rule is constructed using a set of known facts and rules, knowledge, called domain or background knowledge. In essence, the ILP objective is to synthesize a logic program, or at least part of a logic program using examples, background knowledge, and an entailment relation.

To maintain a database of background knowledge, INDED houses a deduction engine that uses deductive logic programming to compute the current state (current set of true facts) as new rules and facts are procured. The deduction engine is a bottom-up, nonmonotonic reasoning system that computes the state by generating a well-founded model of the current ground instantiation. In essence, the deduction engine is a justification truth maintenance system [Doy79].

The induction engine of INDED symbiotically uses the background knowledge base produced by the deduction engine. In particular, it uses the

* This work is partially supported by National Science Foundation grant 9806184.

background knowledge along with positive and negative example sets to induce a rule that will ultimately be fed back into the knowledge base of the deduction engine in the subsequent iteration.



To perform rule mining, we use a standard hypothesis construction algorithm (learning algorithm) [LD94]. As shown below, the algorithm uses two nested loops. The outer loop controls how many clauses make up the learned rule. The inner loop controls how many predicate expressions constitute each clause (i.e., determines rule length). Continuation of each loop is dictated by coverage of examples as defined in [LD94]. The outer loop continues until all (or a user specified percentage) of the positive examples are covered; the inner loop continues until none (or a user specified percentage) of the negative examples are covered. During execution of the inner loop, a rule of the form:

IF <antecedent> THEN <consequent>

is constructed, where the antecedent is made up of predicate expressions of the form:

predicate(argument$_1$, ... argument$_n$)

For example,

parent(X,Y) ← *father(X,Y)*
parent(X,Y) ← *mother(X,Y)*

is a rule that defines when X is a parent of child Y. Each clause is made up of the predicate expressions *parent(X,Y)* as the head of the clause. Each clause body is made up of one predicate expression: *father(X,Y)* and *mother(X,Y)*, respectively.

2.1 Learning Algorithm

Note: all generated rules defining a target are returned as one collective set, at one time.

Input: Target example sets $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$ and background knowledge *B*
Output: A set of intensional rule(s) of learned hypothesis *H*

Begin Algorithm 2.1

While there exists $e \in \mathcal{E}^+$ yet to be covered Do:
Create another intensional rule with head *H*:
Create the rule body as follows:
While there exists $e \in \mathcal{E}^-$ covered Do:
Append the highest ranking predicate expression (positive or negative literal) to the rule body.
Name positions for appended predicate.
Add the newly created rule to the rule set *H*
Return rule set *H*

End Algorithm 2.1

3. PARALLEL INDED

Our goals for parallelizing the induction engine include producing rules faster than the serial implementation, and producing more accurate rules than the serial implementation. The data parallel implementation should enable us to obtain rules faster, and thus potentially explore more of the state space and generate a more accurate rule.

3.1 System Architecture

We are using an eleven node a cluster of workstations (COW). The number of workstations can vary, and generally there is one for the master node with the others serving as worker nodes. Each workstation has its own single processor and local memory. The machines are all connected via a 10Mb Ethernet LAN, with a router separating five of the machines from the others. A Network File System (NFS) shared directory is mapped to each workstation, allowing the distributed induction engine to access files containing the global background knowledge and examples in a centralized location. We are using Message Passing Interface (MPI) as the coordinating software for the cluster [MPI96].

A parallel program executed through the MPI interface creates an instance of the program in the memory of each node. However, all nodes do share the same file space, allowing data transfer and sharing easily through files. MPI can pass messages from one node to another, and facilitate synchronization of parallel code. This synchronization can help avoid common problems, such as race conditions. Under this system, all nodes

run the same code, and it is the responsibility of the implementation to ascertain the role of the node and perform that role. In our implementation, the nodes of the cluster operate in the master-worker paradigm so much synchronization is accomplished by master to worker interaction.

3.2 Parallel Induction Engine

Our motivation is to quickly explore much of the search space and obtain the best rule possible. As indicated in the above algorithm, every rule discovered by INDED is constructed by systematically appending chosen predicate expressions to an originally empty rule body. In order to choose the best predicate expression during clause construction, the predicate expressions are ranked. This ranking takes place by employing various algorithms, each of which designates a different search strategy. The highest ranked expressions are chosen. The variables in this chosen predicate are then ranked as well so that names can be assigned to the variable positions to indicate the implicit correspondence to the variable names in the rule head. For example, without position naming:

parent(X,Y) ● *father(U,W).*

indicates that there is no necessary relation between objects instantiating the first position variables of predicate expressions *parent* and *father*, *X* and *U*, and, likewise, there is no necessary relation between *Y* and *W*. Whereas, employing position naming renders

parent(X,Y) ← *father(X,Y).*

and indicates that the same object instantiating the first position variable in *parent* is the same object as that instantiating the first position variable in *father*.

3.3 Data Parallel Implementation of Inductive Engine

The parallel version of system INDED runs on a Beowulf cluster as mentioned above. Each worker node runs INDED when invoked by a master MPI node. Each worker executes by running on a partial background knowledge base which has been created from the original produced by the deduction engine. The master node takes responsibility for coordination and organization of all tasks, and performs any serial portions of the code. Currently, the worker nodes are limited in their functionality, and wait for input from the master node before performing their work and terminating.

Every rule discovered by INDED is constructed by systematically appending chosen predicate expressions to an originally empty rule body. This choice of predicate expression is determined by the predicate ranking algorithm. The chosen predicate expression then must have variable names assigned to its argument positions. We have distributed both of these activities.

Each worker node receives all facts for a given predicate symbol of the background knowledge base. A worker node will rank an entire predicate at a time, and then report the rank of that predicate symbol to the master node. The worker will then proceed to its next assigned predicate or signals completion. The master waits for information from the workers and stores this in a global data structure. After the worker data is collected, the master node begins formation of the target rule by choosing the best ranked predicate expression. At this point, the chosen predicate must have variable names assigned to its argument positions.

The position naming algorithm assigns variable names to each of the variable positions in the chosen predicate. This activity is far more complex than predicate ranking. For predicate ranking, in previous work [ALS00], the background knowledge base was broken up into (approximately) equally sized portions and delegated amongst the workers. Here, an additional decomposition of work is performed by the master assigning one constant (object) at a time to each worker node, which will search the example sets. The worker then returns the number of occurrences of that constant in each position of the target predicate expression (the expression defined by the positive and negative examples). This quantity is used in overall position ranking for ultimate position naming of the chosen predicate expression.

3.3.1 Parallel Position Naming Algorithm

Note: two algorithms are outlined here, one for the master node and one for all other worker nodes.

Input: Target example sets $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$ and chosen predicate *P*

Output: Variable names for each position in predicate *P*

Begin Master Node Algorithm 3.3.1.a

For each constant $c \in P$ Do:
Instruct a worker to rank *c* using Algorithm 3.3.1.b
For each position $p \in P$ Do:
Collect worker responses for *p*
Name *p* with name of position with highest value
Return predicate *P* with named variables

End Master Node Algorithm 3.3.1.a

Input: Target example sets $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$ and constant C
Output: Number of occurrence of C in each position in $\mathcal{E}$

Begin Worker Node Algorithm 3.3.1.b
For each position $p \in \mathcal{E}$ Do:
  For each constant $c \in p$ Do:
    Count number of occurrences $c = C$
      (if occurrence in $\mathcal{E}^+$ inc count
      otherwise if in $\mathcal{E}^-$ dec count)
  Return counts for p
  (returns are done as soon as the are calculated)
End Worker Node Algorithm 3.3.1.b

4. CURRENT STATUS AND RESULTS

We have developed a framework for transforming a serial inductive logic programming system into a data parallel system. A prototype induction engine which parallelizes the predicate ranking algorithm and position naming algorithm has been developed to run on the LAM implementation of MPI. At this time there has been no major increase in performance speed for the parallel version verses the serial version, and in fact, on average, the serial system still performs better. For one data set, the serial implementation execution times averaged a half second faster than the parallel implementation. However, when the minimum execution times were taken for that same data set, which has some complexity as a position naming problem, it was found that the data parallel implementation performed two hundredths of a second faster. It should be noted that both of these minimum execution times were approximately seven tenths of a standard deviation off of their respective averages. This is the first time the data parallel implementation out-performed the serial implementation, and it does not appear to be an anomaly. This result gives us great hope that when we experiment with larger data sets, the reduction in time will be substantial.

5. FUTURE WORK

There is much future work for this project. One goal is to obtain large data sets in other domains, especially the domain of Lyme disease diagnosis. Another goal is to redesign the overall induction engine so that it is more suited for parallel operation. Along these lines an interesting side effect of the data parallel implementation is that there is no need for a central store or for one node to contain the entire knowledge base. This implementation can greatly reduce the amount of memory and storage necessary to operate system INDED, allowing it to provide the same functionality on cheaper systems. We expect that this will prove especially useful for analyzing large problem domains such as Lyme disease diagnosis. Also, a more efficient means of resource management and communication will need to be investigated and implemented. Currently, the master node tells the workers what to do; only then will they do work. Consequently, in some cases, workers can sit idle waiting for a directive from the master node.

6. CONCLUSION

The absence of time reduction could be due to several reasons. The data sets used for experimentation are still relatively simple and to some degree, trivial. It will be necessary to find larger data sets with which to test the data parallel version of INDED. A significant amount of overhead is also added to facilitate communication between processors. It should also be noted that the parallel implementation is thus far still in its prototype stages, and is highly dependent on internal representations and algorithms which are more suited to serial implementation as opposed to a parallel one. These implementation problems provide extra overhead and work for the data parallel system, which is not encountered by the serial system. As we continue to parallelize more parts of INDED, however, we anticipate that the data parallel INDED will exhibit vast temporal improvements over the serial one.

7. REFERENCES

[ALS00] Lee A. Adams, Kevin M. Livingston, and Jennifer Seitzer. An Implementation of a Parallel Machine Learner. Proceedings of the Eleventh Midwest Artificial Intelligence and Cognitive Science Conference (MAICS'2000); Pages 47-52, .AAAI Press.

[Doy79] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231--272, 1979.

[LD94] Nada Lavrac and Saso Dzeroski. *Inductive Logic Programming*. Ellis Horwood, Inc., 1994.

[MPI96] MPI primer / developing with lam, 1996.

[Mug92] Stephen Muggleton, editor. *Inductive Logic Programming*. Academic Press, Inc, 1992.

[PSF91] Piatetsky-Shapiro and Frawley, editors. *Knowledge Discovery in Databases*, chapter Knowledge Discovery in Databases: An Overview. AAAI Press/ The MIT Press, 1991.

AUTHORS:

Kevin Livingston is currently a student at the University of Dayton, where he will graduate with a BS in Computer Science in December of 2000. He has been a student of Dr. Seitzer's since January 1999 studying both Artificial Intelligence and Networking, and has been involved with the Learning with Reason Research Group since August of 1999. Kevin's main area of focus within Computer Science is naturally Artificial Intelligence, and will continue to be his focal point for any postgraduate education.

Dr. Jennifer Seitzer is an Assistant Professor at the University of Dayton where she teaches both undergraduate and graduate students courses in artificial intelligence and networking. She is director of the Learning with Reason Student Research Group under partial funding from the NSF. Dr. Seitzer is a supervisor of the Digital Networking Lab where the Beowulf cluster for this research is housed. Dr. Seitzer is also the faculty co-advisor for UD's chapter of the ACM.