A pipelined data-parallel algorithm for ILP

Nuno A. Fonseca Fernando Silva
DCC-FC & LIACC, Universidade do Porto,Portugal
{nf,fds}@ncc.up.pt
Vitor Santos Costa
COPPE/Sistemas & Universidade Federal do Rio de Janeiro,Brasil
vitor@cos.ufrj.br
Rui Camacho
Faculdade de Engenharia & LIACC, Universidade do Porto,Portugal
rcamacho@fe.up.pt

Abstract

The amount of data collected and stored in databases is growing considerably for almost all areas of human activity. Processing this amount of data is very expensive, both humanly and computationally. This justifies the increased interest both on the automatic discovery of useful knowledge from databases, and on using parallel processing for this task. Multi Relational Data Mining (MRDM) techniques, such as Inductive Logic Programming (ILP), can learn rules from relational databases consisting of multiple tables. However, current ILP systems are designed to run in main memory and can have long running times. We propose a pipelined data-parallel algorithm for ILP. The algorithm was implemented and evaluated on a commodity PC cluster with 8 processors. The results show that our algorithm yields excellent speedups, while preserving the quality of learning.

1. Introduction

The amount of data collected and stored in databases is growing considerably in almost all areas of human activity. A paramount example is the explosion of biotech data, where the volume of data doubles every three to six months as a result of automation in biochemistry [3]. In this case, as in many others, processing this amount of data is beyond human analysis and is computationally very expensive. This justifies the increased interest both on the automatic discovery of useful knowledge from databases and on using parallel processing for this task.

Multi Relational Data Mining (MRDM) is a fast growing multi-disciplinary field that deals with knowledge discovery from relational databases consisting of multiple tables. Several techniques in Relational Data Mining have been developed through Inductive Logic Programming (ILP). ILP is a form of supervised relational learning that aims at the induction of first-order logic *theories* (sets of rules) from a given set of examples and from prior knowledge. The expressiveness of first-order logic gives ILP the flexibility and understandability of the induced models. However, ILP systems suffer from significant limitations. First, most ILP systems execute in main memory, limiting their ability to process large databases. Second, ILP systems, like other MRDM algorithms, are computationally expensive. On sizable applications, ILP systems can take several hours, if not days, to return a model.

Previous research on improving the efficiency of ILP systems has largely focused in reducing their sequential execution time, either by reducing search space size (see, e.g., [26]), or by efficiently testing candidate rules (see, e.g., [2, 8]). Parallelism provides an attractive alternative solution, as it may both significantly decrease learning time and allow the original database to be split among different nodes, therefore increasing maximum data size. In most cases the speedup techniques proposed for sequenctial execution are still usable in a parallel setting. Most parallel ILP implementations so far [11] have focused on speeding up execution and targeted shared-memory machines, due to their more convenient programming model. However, distributed memory machines have important advantages, namely regarding data scalability. Unfortunately, to the best of ourt knowledge, previous work on distributed-memory implementations of ILP has only achieved limited performance [19, 14, 6], mainly due to communication and load-balancing issues.

In this paper we propose a novel parallel ILP algorithm

targeted for distributed memory machines. Our algorithm exploits pipelined data-parallelism [16, 17] and is geared to achieve good speedups while preserving the quality of the models. The algorithm takes advantage of data-parallelism by dividing examples evenly between the p processors and starting p searches in parallel. This makes it possible to fit large datasets into main memory. Unfortunately, training on small subsets of the whole data might reduce the quality of learning. We address this problem by streaming the best rules over *every subset* in a pipelined fashion.

In more detail, our key idea is to start p independent searchs, such that each search is divided into p stages. Each step (stage) of the search uses a local subset of the examples. The good rules found in each stage are sent to the next stage and are used as a starting point for a new search, now using the next subset of examples. The search for a good rule eventually traverses all subsets of examples. We expect all different searches to complete at about the same time, giving us balanced parallelism.

We implemented and evaluated this algorithm on a PC cluster with 8 processors using real world applications. Our experiments show that indeed the algorithm does achieve substancial performance improvements in a fully distributed environment, while preserving the quality of the models.

The remainder of the paper is organised as follows. We next introduce some background on ILP and the notation used in this paper. Section 3 presents a widelly used sequential ILP algorithm. Section 4 describes a pipelined data-parallel algorithm for ILP. Section 5 discusses the performance of the proposed algorithm based on an empirical evaluation. In Section 6 we review the related work. Last, we draw some conclusions and present future work.

2. Background

2.1. ILP

Arguably, one of the most expressive and human understandable representation scheme for learning is *if-then* rules. ILP is one of several approaches that learn such rules. In ILP's case, rules can contain variables, and are based on first-order Horn clauses. Sets of first-order Horn clauses can be interpreted as logic programs. This type of learning is thus called Inductive Logic Programming (ILP). For a gentle introduction to the field we refer to [24, 27].

ILP systems induce (learn) sets of rules from data. As input, ILP require *training examples* of the concept to learn. Usually, we have positive and negative examples. Often, they benefit from prior knowledge, also named *background knowledge*. Both, examples and background knowledge are usually represented as logic programs. One common case is for examples to be tuples in a table in a database, and the background knowledge to be all other tables on that database.

ILP systems try to to find a consistent and complete *theory*, i.e., a set of rules that *explain* all given positive examples (*completeness*), and do not explain the negative examples (*consistency*). In the real world, examples may be misclassified (*noise*). ILP systems handle noise by relaxing the consistency condition: rules can be accepted even if they do cover some small number of negative examples.

The percentage of correctly classified examples is called *accuracy*. *Training accuracy* is the accuracy observed on the training data, i.e., the data we used for learning. Our real goal, though, is to achieve best *predictive accuracy*, i.e., the accuracy measured on unseen examples. Accuracy of a rule r essentially depends on its *coverage*: the number of positive (*positive cover*) and negative examples (*negative cover*) derivable from r given the background knowledge B. The time needed to compute the coverage of a rule depends primarily on the cardinality of E (i.e., $\mid E^+ \mid$ and $\mid E^- \mid$) and on the theorem proving effort required to evaluate each example using the background knowledge.

2.2. Notation

We will use the following notation in the remainder of the paper. p is the number of processors/workers available. The processors are organized in a master-worker model. We abstract three communication operations: `send`, `broadcast`, and `receive`. The broadcast and send operations are assumed to be non-blocking, while receive is blocking. In our model, workers receive the tasks to execute in messages sent by the master. After receiving a message, a worker executes it. If the task returns some value, it is sent to the master when the task completes.

3. A sequential ILP algorithm

In order to assess our algorithm, we really need to be able to compare ir with a standard ILP search algorithm. A widely implemented technique in ILP is Mode-Directed Inverse Entailment (MDIE) [22]. In MDIE we iterate through a search process to find good rules, and ultimately obtain a set of rules, our *theory*.

3.1. Algorithm

Figure 1 presents an MDIE-style procedure. It induces the theory $Rules\_Learned$ from a set of examples E, background knowledge B, and some constraints C. In a nutshell, the algorithm learns one rule at a time (step 6) using a generalization procedure that performs a search through an ordered space of legal rules. After accepting a rule, all positive examples covered are removed from the training set.

mdie(B,C,E^+,E^-)
Input: Background knowledge (B), constraints (C), positive (E^+) and negative (E^-) examples.
Output: A set of complete and consistent rules ($RulesLearned$).

```
1.  Rules_Learned=∅
2.  Remaining=| E+ |
3.  while Remaining > 0 and stopping condition not met do
4.     e=select an example from E+
5.     ⊥e=build_msh(e,B,E+,C)
6.     R=bestOf(learn_rule(B,C,E+,E−,⊥e))
7.     Rules_Learned=Rules_Learned ∪ {R}
8.     Covered={Examples Covered by R}
9.     E+=E+ \ Covered
10.    Remaining=| E+ |
11. end while
12. return Rules_Learned
```

Figure 1. A learning procedure based on MDIE. The `build_msh()` procedure constructs the most-specific-clause $\perp_e$ that entails the example selected and is within language restrictions provided (C). The search procedure, invoked in step 6, finds the best consistent clause more general than example e by performing a search through the space of legal clauses.

The next rule is then learned from the remaining examples. The process repeats until no positive examples are left or until some other stopping criteria is met (e.g., some time limit).

Steps 4 and 5 show that each rule is obtained by starting from a specific example, and looking for derivable literals. Technically, these literals form a most specific rule $\perp_e$ that entails the example selected and satisfies the language restrictions in C. The most specific rule is also known as *bottom clause*. The step is important because it constrains the search: we will use the literals in the bottom clause to compose the rules. Even so, the number of rules grows exponentially with the size of the *bottom clause*, resulting in a very large, or even infinite, search space.

The procedure shown in Figure 1 is implemented by a number of ILP systems (e.g., [22, 25, 31, 5, 1]). Most often, the algorithms differ in how they perform the `learn_rule()` procedure (step 6). This procedure, outlined in Figure 2, receives a set of examples and prior knowledge and returns a set of consistent rules, that explain some or, ideally, all positive examples. Usually, this procedure returns a single rule, but here we assume it will return a set of good rules and then the best one is chosen.

The `learn_rule`() procedure searches the (potentially infinite) space of rules for a rule that optimizes some quality criteria. The search for a rule involves traversing the space of rules, that in turn involves generating and evaluating rules. In order to traverse the space of rules systematically a structure is imposed upon it, i.e., an ordering. The most popular ordering used in ILP is the θ-subsumption, defined by Plotkin [28]. It introduces the syntactic notion of generality. The evaluation of a rule usually requires the computation of its coverage, i.e., computing how many examples the rule explains. The time taken to compute the coverage of a rule depends, primarily, on the number of examples. Thus, scalability problems may arise when dealing with a great number of examples or/and when the computational cost to evaluate a rule is high.

learn_rule$(B,C,E^+,E^-,\perp_e)$
Input: Background knowledge (B), constraints (C), positive (E^+) and negative (E^-) examples, and the bottom clause ($\perp_e$).
Output: A set of rules ($Good$).

```
1.  Good=∅
2.  S=START_RULE
3.  Pick=pickRule(S)
4.  S=S \ {Pick}
5.  NewRule=genNewRule(Pick,C,⊥e)
6.  Val=evalOnExamples(B,E+,E−,C,NewRule)
7.  if is_good(NewRule,C,Val) then
8.     Good=Good ∪ {NewRule}
9.  endif
10. if stop criterium not satisfied then
11.   goto 3
12. endif
13. return Good
```

Figure 2. An example of a generic `learn_rule()` procedure.

4. $P^2 - mdie$: A pipelined data-paralllel algorithm for ILP

This section presents our pipelined data-parallel covering algorithm, p^2 algorithm for short. As the name suggests, the algorithm combines pipelining and data parallelism. The algorithm exploits data-parallelism by dividing the data (set of examples E) by all workers and by learning in parallel on each worker. Pipelining is possible by breaking the process of learning the best rule into a sequence of stages. Each stage performs a search using a subset of examples local to the stage's owner. The good rules found are sent to the next stage (worker) to be used as starting point of a new search using a different subset of examples. Therefore, the search for a rule is incrementally performed using different sets of examples. When the pipeline completes the newly found rules are then sent to the master.

Parallelism is obtained by having the p stages of the pipeline always busy. This is possible because *every stage*

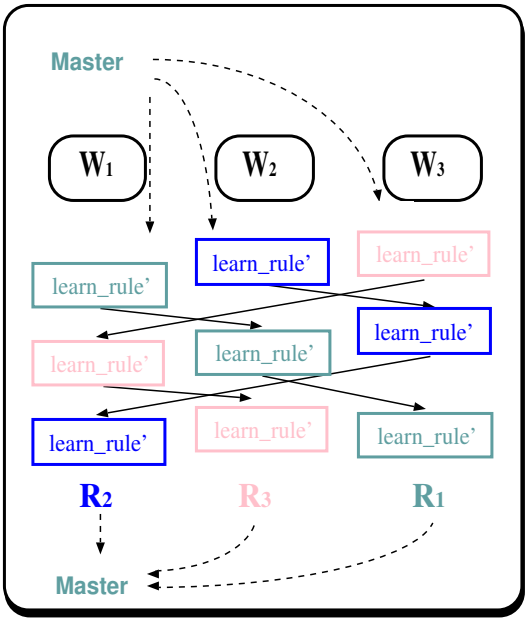

Figure 3. Parallel pipelined rule search with 3 stages

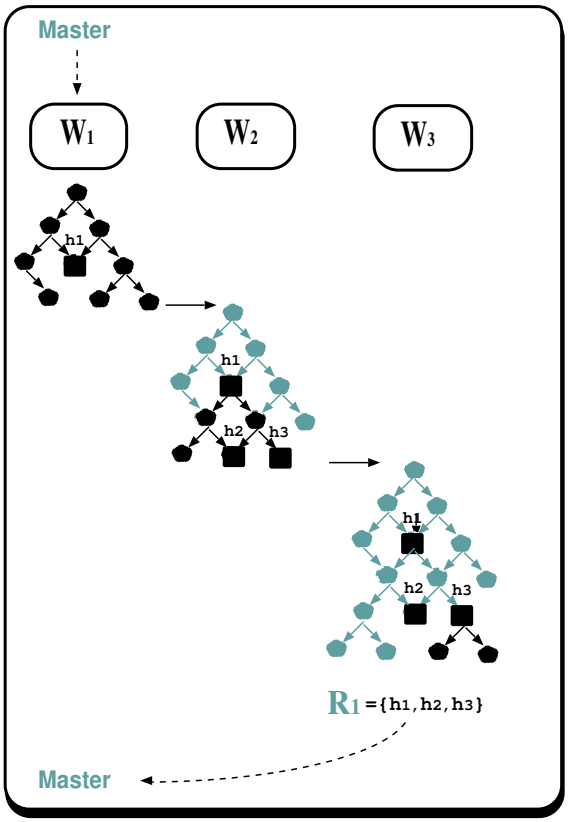

Figure 4. Example of a pipelined rule search. The squared box represent nodes in the search space considered good. The good nodes are used as starting points of the search in the next worker.

of the pipeline does the same task, just on a different set of examples. We thus simultaneously start p searches at the p pipeline stages. When a search completes level p it is folded back to level 1. Figure 3 exemplifies this process on a parallel pipeline with 3 workers. The master starts 3 searches, one for each worker. Each worker learns a set of rules, using the local data. At the end of its stage, it passes the good rules (represented as squares in Figure 4) to the next worker in the pipeline. The next worker receives a set of rules found previously and uses them as starting points for a new search using the local data, as exemplified in Figure 4. At the end, the 3 sets of rules found (R_1, R_2, R_3) are sent to the master.

4.1. The algorithm

The $p^2 - mdie$ algorithm is a pipeline data-parallel covering algorithm based on MDIE. Figure 5 outlines the algorithm of the master (the notation used is explained in Section 2.2).

We divide execution into a number of *epochs*. Each epoch is responsible for finding rules that cover a number of positive examples. As in MDIE, epochs are run sequentially, and execution will consist of as many epochs as we need to have all positive examples covered.

Each epoch is managed by a *master*. The master performs three steps. At step 1, the master randomly and evenly partitions the examples into p subsets. Each worker is then notified to load its subset of examples together with the remaining data (prior knowledge, constraints, ...).

We assumed in this algorithm that the data can be shared by all processors, through a distributed file system, and therefore no messages containing background knowledge (B), the constraints (C), and the examples (E^+ and E^-) are exchanged. Obviously, if file sharing is not possible one needs to exchange messages containing the referred data. Example data is loaded only once, hence the transmission cost should be low in both approaches.

The next step for the master is to start p pipelines in parallel, one pipeline per worker. The algorithm for a pipeline stage I proceeds as follows:

1. If first step, select an example e_I and generate the most specific rule $\bot_{e_I}$ from example e_I. This rule constrains the search space (as explained in Section 3). If not first step, $\bot_{e_I}$ is provided by the previous stage.

p^2-mdie(E^+,E^-,B,C,p,w)
Input: set of positive (E^+) and negative (E^-) examples, background knowledge (B), constraints (C), number of processors (p), and the maximum number of consistent rules (w) passed on each stage of the pipeline.
Output: A set of complete and consistent rules ($Rules\_Learned$).

```
1.  Rules_Learned=∅
2.  Partition E+ and E− into p subsets (E1+,E1−),...,(Ep+,Ep−)
3.  broadcast load_examples()
4.  Remaining=| E+ |
5.  while Remaining > 0 or stopping condition met do
6.     for k=1 until p do
7.        send k start_pipeline(w)  /* start pipeline at worker k */
8.     end for
9.     RulesBag=∪_{k=1}^{p} (receive k Rules_k)
10.    broadcast evaluate(RulesBag)
11.    Results=∑_{k=1}^{p} (receive k Result_k)
12.    while RulesBag ≠ ∅ do
13.       R=pickBest(RulesBag,Results)
14.       RulesBag=RulesBag \ {R}
15.       Rules_Learned=Rules_Learned ∪ {R}
16.       broadcast mark_covered(R)
17.       Remaining=Remaining-PosCovered(R,Results)
18.       broadcast evaluate(RulesBag)
19.       Results=∑_{k=1}^{p} (receive k Result_k)
20.       Rules2Delete=notGood(RulesBag,Results)
21.       RulesBag=RulesBag \ Rules2Delete
22.    end while
23. end while
24. return Rules_Learned
```

Figure 5. Procedures of the p^2-mdie algorithm executed by the master node.

2. Perform search using $\perp_{e_I}$.

3. If last step output "good" rules back to the master. Otherwise, output "good" rules and $\perp_{e_I}$ to next worker $(I+1)\%P$.

The last step of the master can be started when the master receives a set of rules from every worker. The master collects all such rules in a bag ($RulesBag$) and then broadcasts the whole set to every worker. This step is necessary in order to appreciate the *global* quality of the rules.

The rules in the bag are then consumed (added to the $Rules\_Learned$) according to the following sequential algorithm, designed to emulate MDIE as closely as possible:

1. select the best rule from the bag, according to predefined criteria, and remove it from the bag;

2. remove the positive examples covered by the rule on all subsets;

3. update the $Remaining$ number of examples by subtracting the positive examples covered by the rule R;

4. reevaluate, in parallel, the value of the rules in the bag (on all subsets) and collect the results;

5. remove the rules from $RulesBag$ that are no longer "good".

The consumption of rules ends when there are no more rules in the bag or other criteria is verified. At this point the epoch is over. This is a main difference to the sequential algorithm, since several rules can be added to the theory in a single epoch as opposed to a single rule in the sequential algorithm. Epochs repeat until have all positive examples covered.

start_pipeline(W)
Input: The maximum number of consistent rules W passed on each stage of the pipeline.
E^+ and E^- are, respectively, the local set of positive and negative examples, and B the background knowledge, and C the set of constraints

```
1.  e=select an example from E+
2.  ⊥e=build_msh(e,B,C)
3.  learn_rule'(⊥e,1,W,∅)
```

evaluate_rules($Rules$)
Input: Set of rules ($Rules$)

```
1.  Stats=∅
2.  foreach rule in Rules
3.     Stats=Stats ∪ evalOnExamples(rule)
4.  endfor
5.  send master Stats
```

load_examples()

```
1.  me=processor id
/* Loads the subset of examples and remaining data */
2.  <B,C,E+,E−>=load(me)
```

mark_covered(R)
Input: Rule (R)

```
1.  B=B ∪ { R }
2.  E+=E+ \ {e | e ∈ E+ ∧ B ⊢ e}
```

Figure 6. p^2-mdie - A pipelined data-parallel covering algorithm based on MDIE (worker view).

Pipelining is exploited by the `learn_rule'()` procedure, shown in Figure 7. It is similar to the `learn_rule()` described previously with the following differences. First, it includes a parameter to indicate the set of starting points of the search. Secondly, the procedure has a parameter that indicates the current pipeline stage. When the pipeline starts this value is set to 1, and is incremented when the rules found are passed to the next stage in the pipeline. Thirdly, the procedure outputs a set consisting of the "best" W rules instead of the single "best" rule. The W can be seen as a parameter that defines the pipeline width. After performing the search, the best W rules are selected and sent to: i) the

learn_rule'($\perp_e$,$Step$,w,S)
Input: current pipeline step ($step$), pipeline width (w), initial set of rules S.
Output: A set of rules ($Good$).
Local data: Background knowledge (B), constraints (C), positive E^+ and negative E^- examples.

```
1.  Good=S
2.  Pick=pickRule(S)
3.  S=S \ {Pick}
4.  NewRule=genNewRule(Pick,C,⊥e)
5.  Val=evalOnExamples(B,E+,E−,C,NewRule)
6.  if is_good(NewRule,C,Val) then
7.     Good=Good ∪ {NewRule}
8.  endif
9.  if stop criterium not satisfied then
10.   goto 3
11. endif
12. if is_last_step(step) then
13.   send master Good
14.   return
15. endif
16. NextWorker=next_worker()
17. send NextWorker learn_rule'(⊥e,step+1,w,Good)
```

Figure 7. A pipelined `learn_rule` procedure. `next_worker()` computes the identifier of the next worker on the pipeline.

master if the pipeline has reached the end, i.e., current stage value is p; or ii) to the next stage of the pipeline (i.e., next worker).

The granularity of the parallel tasks are medium or high, and depend primarily on the number of examples on each partition and the complexity of the background knowledge. The granularity of the tasks executed in parallel are very similar, leading to balanced computations, hence simplifying processor load balancing.

As computation evolves and the rules are learned, the examples explained by them are removed from the subsets of examples. This may lead to unbalanced subsets, i.e., their sizes may vary considerably, which in turn may result in unbalanced computations. A possible solution to cope with this problem, with a considerable cost in message communication, could be the repartitioning of examples always before starting the pipelines. However, we did not considered this approach mainly because the high communication cost of repartitioning.

One drawback of the algorithm is that it is not possible to learn recursive rules. This results from having partitioned the set of positive examples. We are aware of this problem, but, at this stage, we did not consider it important since in most real world applications very few concepts seem to have recursive definitions.

4.2. Implementation

We implemented the ILP pipeline data-parallel covering algorithm based on MDIE as described in the previous section. The implementation was done in the context of the April [12] ILP system. April combines ideas and features from several systems, such as Progol [22] et seq. [23, 31], Indlog [5], CILS [1], into a single coherent system. April aims at maximum flexibility through a very modular design. April is implemented using the Prolog language. The Prolog engine used was the Yap prolog system [7]. We chose Yap because it is one of the fastest available Prolog systems [9].

April was configured to perform a top-down breadth-first search to find a rule. The rules generated during a search are evaluated using a heuristic that relies on the number of positive and negative examples. The rules in the bag were ordered based on their global coverage, i.e., the aggregate coverage on all subsets.

We used for the communication layer LAM [4, 30] MPI. A YAP module was developed in C to act as the interface between Prolog and LAM/MPI libraries.

It is interesting to note that the proposed algorithm is also applicable to other ILP systems that are based on the $mdie$ algorithm.

5. Experiments and Results

The proposal of using pipelined data-parallelism in covering algorithms aims, primarily, at the reduction of the execution time. A variation in predictive accuracy could be expected as a side effect of data partitioning. The question is to know if the quality of the model is significantly changed. To answer this question and to evaluate the speedup of the algorithm we performed several experiments using a set of ILP applications.

5.1. Materials

For the experiments we used 3 ILP applications. Table 1 characterizes the datasets using the number of positive and negative examples. The carcinogenesis [32], and pyrimidines [18] datasets are applications from molecular biology. The mesh [10] dataset is an application from the area of mechanical engineering.

Dataset	$\|E^+\|$	$\|E^-\|$
carcinogenesis	162	136
mesh	2840	278
pyrimidines	848	764

Table 1. Datasets Characterization

5.2. Method

The experiments were performed on an Beowulf Cluster composed with 4 nodes. Each node is a dual processor computer with 2GB of memory and runs the Linux Fedora OS. We used the YAP Prolog system, version 4.5, and the April ILP system [12], version 0.9. Its important to point out that our proposal is applicable to other ILP systems based on the $mdie$ algorithm. For each dataset we tuned the settings of the ILP system so that the sequential runs would not take more than two hours. We did this by imposing a threshold on the number of rules that can be generated on each search.

The performance of our $p^2 - mdie$ algorithm, described in Section 4, was evaluated using 5-fold cross validation, i.e., the data was dividided into 5 subsets (folds) of (approximately) equal size. Then, for each run one fold was set aside for testing while the remaining were joined and used for learning. The values presented are the average of the five runs, with the standard deviations in parenthesis.

5.3. Results

We ran the algorithm with two different pipeline **widths**: in the first case, no limit was imposed on the width of the pipeline, i.e., *all* good rules were passed to the next stage of the pipeline; in the second case we imposed a limit of 10 rules between stages (the limit is based on the maximum number of processors for our tests). Note that the number of good rules found at each stage of the pipeline can be quite high. For instance, in the Mesh dataset is usual to have some thousand rules at the end of one pipeline. Therefore, the ammount of communication exchanged between stages can be quite large.

The run time and speedups observed in the experiments are summarised in Table 2 and Table 3, respectively. Sequential running times for all three cases are on the order of the thousands of seconds. Using two processors results in a speedup ranging from 20% to almost 90%. Speedups improve at 4 processors, and get close to linear and, in some cases, superlinear for 8 processors. Note that accuracy is about the same throughout, as shown in Table 6.

Two factors may explain the superlinear performance. First, the division of the data among more processors speeds up the evaluation of each rule in a subset, since there are less examples to test. Second, an increase on the number of processors may also result on an increase of the number of rules each epoch generates, therefore reducing the number of epochs. Table 5 shows this to be indeed the case. The reduction in the number of epochs while keeping or reducing the time to process an epoch explains the speedups.

Dataset	Width	2	4	8
carcinogenesis	nolimit	1.63	3.47	11.12
	10	1.19	3.48	10.22
mesh	nolimit	1.32	3.11	4.90
	10	1.21	4.22	10.85
pyrimidines	nolimit	1.89	3.71	4.70
	10	1.72	3.82	9.00

Table 2. Average speedup observed for 2, 4 , and 8 processors for a pipeline width of 10 and unlimited.

Dataset	Width	1	2	4	8
carcinogenesis	nolimit	3,231	1,976	930	290
	10	-	2,704	927	316
mesh	nolimit	4,412	3,335	1,419	900
	10	-	3,639	1,045	407
pyrimidines	nolimit	5,704	3,013	1,536	1,215
	10	-	3,202	1,493	633

Table 3. Average execution time (in seconds) observed for 2, 4 , and 8 processors

We observe that the best speedups were obtained when constraining the width of the pipeline. We believe that the explanation is that the wider pipeline results in more data being exchanged between processors. Table 4 does show that the communication is usually much higher when the width of the pipeline is unconstrained. Namely, both Mesh and Pyrimidines have the speedup for 8 processors to be below linear for unlimited pipeline width, but superlinear when the pipeline is constrained. In both cases the am-

Dataset	Width	2	4	8
carcinogenesis	nolimit	3	13	33
	10	2	7	18
mesh	nolimit	80	105	1,074
	10	46	92	552
pyrimidines	nolimit	12	105	1,166
	10	6	55	593

Table 4. Average communication exchanged (in MBytes) for 2, 4 , and 8 processors.

Dataset	Width	2	4	8
carcinogenesis	nolimit	19	12	8
	10	20	13	10
mesh	nolimit	309	150	65
	10	315	149	64
pyrimidines	nolimit	241	122	64
	10	250	126	64

Table 5. Average number of epochs for 2, 4 , and 8 processors for a pipeline.

mount of communication exchanged when using 8 processors grows more 10 times when compared with 4 processors. The large amount of messages being transfered should affect performance.

On the other hand, using the unlimited pipeline width results in better performance in carcinogenesis. The reason is that in this application, the amount of data transferred between stages is quite small, suggesting that stages are not finding enough rules to cause a significant communication overhead.

Table 5 presents the average number of epochs for the three benchmarks. We can notice that in all cases there is a significant reduction in epochs as we increase the number of processors, indicating that we do benefit from running the pipelines in parallel.

Table 6 reports the averaged accuracy for each dataset. We use the paired t-test to detect significance. Our conclusion, up to a 98% confidence level in, was that accuracy did not significantly change for most runs except on the cases marked as '*'. Accuracy improved in these cases, as compared with the sequential algorithm.

We believe the speedup results are quite good, overall, especially when compared with other distributed ILP implementations [11]. Constraining the pipeline width leads to increased speedups, without affecting the quality of the models.

6. Related Work

Several approaches have already been implemented to parallelize ILP systems. Most implementations where done for shared memory multi-computers, where data transmission is very low when compared to distributed memory computers. In spite of the higher cost in communication, distributed memory computers are capable of scaling better (e.g., just add another computer) and are cheaper (consider the widespread use of Beowulf clusters). Thus, it is not a surprise that research and implementation are currently focusing on the development of algorithms and their implementations for distributed memory architectures (see e.g., [19, 14, 6]).

PolyFarm [6] is a parallel ILP system for the discovery of association rules targeted to distributed memory architectures. The system follows a master-worker scheme. The master generates the rules and reports the results. The workers perform the coverage tests of the set of rules received from the master on the local data. The counts are aggregated by a special type of worker (Merger) that reports the final counts to the master. No empirical evaluation of the system was presented in [6].

Konstantopoulos [19] implemented a data parallel version of the Aleph [31] system using MPICH [15] MPI [13] library. The algorithm performs coverage tests in parallel. This approach was very similar to the one of Graham et al. [14] with the difference that in [19] only a single clause is evaluated in parallel, while in [14] a set of clauses are evaluated. The smaller granularity of the parallel tasks may be the justification for the poor results presented by Konstantopoulos.

Matsui et al. [20] evaluated and compared data parallelism (background knowledge and the set of examples) and, what they called, parallel exploration of the search space. The search space parallel approach consisted in evaluating in parallel the refinements of a clause without doing data partitioning. The three strategies were implemented within the FOIL [29] system and were evaluated on a distributed memory computer using the *trains* dataset [21]. Unfortunately, parallelizing the search resulted in low speedups. The authors observed that the size of the divided tasks may not be all the same, hence reducing the efficiency. The other two approaches based on data parallelism showed a linear speedup up to 4 processors. The speedup decreased above 4 processors as a result of an increase in communication due to the exchange of the training set.

7. Concluding Remarks

This paper describes the first pipelined data-parallel algorithm for multi-relational learning. The algorithm exploits data parallelism by evenly dividing the set of examples by the available processors and parallel-pipelining in the search for good rules. The empirical evaluation of the algorithm shows excellent results, in some cases even superlinear speedups. We believe that our work is a step towards having MRDM systems to tackle large scale data mining tasks.

Our algorithm has several interesting characteristics. It fosters scalability on the number of examples, resulting from partitioning the set of examples and then incrementally learn partially correct rules on each subset. The granu-

Dataset	Width	1	2	4	8
carcinogenesis	nolimit	61.73 (8.30)	58.12 (8.11)	59.05 (7.98)	56.67 (8.61)
	10	-	59.35 (7.64)	54.89 (8.49)	59.63 (8.74)
mesh	nolimit	63.76 (2.35)	61.96 (2.54)	63.79 (2.50)	*70.72 (2.44)
	10	-	63.41 (2.30)	*69.05 (2.24)	*74.95 (2.20)
pyrimidines	nolimit	76.47 (1.38)	76.18 (1.34)	74.64 (1.31)	74.89 (1.24)
	10	-	74.93 (1.44)	74.43 (1.31)	75.61 (1.24)

Table 6. Average predictive accuracy observed for 2, 4 , and 8 processors, with the standard deviations in parenthesis.

larity of the tasks executed in parallel are very similar, leading to balanced computations, and we believe therefore simplifying processor load balancing. The granularity of the parallel tasks are medium or high, and depend primarily on the number of examples on each subset. It is possible to control the granularity of data exchanged between the processors in the pipeline (i.e., by limiting the number of good rules passed on each stage of the stream). The experimental results show that the algorithm does not affect the quality of the model.

In the future, we plan to experiment on a variety of other applications. We are particularly interested in applications where ILP has performed badly so far due to lack of resources, such as some large biological and web-mining applications.

Acknowledgments

The work presented in this paper has been partially supported by project APRIL (Project POSI/S-RI/40749/2001) and funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *Programa POSI*. Nuno Fonseca is funded by the FCT grant SFRH/BD/7045/2001. Support for this research was partially provided by U.S. Air Force grant F30602-01-2-0571. Vítor Santos Costa was visiting the University of Wisconsin-Madison while this research took place.

References

[1] S. Anthony and A. M. Frisch. Cautious induction: An alternative to clause-at-a-time induction in inductive logic programming. *New Generation Computing*, 17(1):25–52, January 1999.

[2] H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.

[3] C. Brown. Biotech data's big bang. EE Times online, April 2005. http://www.eetimes.com/news/latest/showArticle.jhtml?-articleID=161501131.

[4] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[5] R. Camacho. *Inducing Models of Human Control Skills using Machine Learning Algorithms*. PhD thesis, Department of Electrical Engineering and Computation, Universidade do Porto, 2000.

[6] A. Clare and R. D. King. Data mining the yeast genome in a lazy functional language. In *PADL*, pages 19–36, 2003.

[7] V. S. Costa, L. Damas, R. Reis, and R. Azevedo. *YAP Prolog User's Manual*. Universidade do Porto, 1989.

[8] V. S. Costa, A. Srinivasan, and R. Camacho. A note on two simple transformations for improving the efficiency of an ILP system. *LNCS*, 1866, 2000.

[9] B. Demoen and P.-L. Nguyen. So Many WAM Variations, So Little Time. In *LNAI 1861, Proceedings Computational Logic - CL 2000*, pages 1240–1254. Springer-Verlag, July 2000.

[10] B. Dolsak, I. Bratko, and A. Jezernik. *Machine Learning, Data Mining and Knowledge Discovery: Methods and Applications*, chapter Application of machine learning in finite element computation. John Wiley and Sons, 1997.

[11] N. A. Fonseca, R. Camacho, and F. Silva. Strategies to Parallelize ILP Systems. In *To appear in Proceedings of the 15th International Conference on Inductive Logic Programming*, volume 3625 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2005.

[12] N. A. Fonseca, R. Camacho, F. Silva, and V. S. Costa. Induction with April: A preliminary report. Technical Report DCC-2003-02, DCC-FC, Universidade do Porto, 2003.

[13] M. P. I. Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[14] J. Graham, C. D. Page, and A. Kamal. Accelerating the drug design process through parallel inductive logic programming data mining. In *Proceeding of the Computational Systems Bioinformatics (CSB'03)*. IEEE, 2003.

[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message

passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[16] W. D. Hillis and G. L. S. Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[17] C.-T. King, W.-H. Chou, and L. M. Ni. Pipelined data parallel algorithms: Part I - Concept and modeling. *IEEE Transactions on Parallel And Distributed Systems*, 1(4), 1990.

[18] R. King, S. Muggleton, and M. Sternberg. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. In *Proc. of the National Academy of Sciences*, volume 89, pages 11322–11326, 1992.

[19] S. K. Konstantopoulos. A data-parallel version of aleph. In *Proceedings of the Workshop on Parallel and Distributed Computing for Machine Learning, co-located with ECML/PKDD'2003*, Dubrovnik, Croatia, September 2003.

[20] T. Matsui, N. Inuzuka, H. Seki, and H. Itoh. Comparison of three parallel implementations of an induction algorithm. In *8th Int. Parallel Computing Workshop*, pages 181–188, Singapore, 1998.

[21] R. Michalski. Pattern recognition as rule-guided inductive inference. In *Proceedings of IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 349–361, 1980.

[22] S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

[23] S. Muggleton. Learning from positive data. In S. Muggleton, editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming*, volume 1314 of *LNAI*, pages 358–376. Springer-Verlag, 1996.

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

[25] S. Muggleton and J. Firth. Relational rule induction with cprogol4.4: A tutorial introduction. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*, pages 160–188. Springer-Verlag, September 2001.

[26] C. Nédellec, C. Rouveirol, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in ILP. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 82–103. IOS Press, 1996.

[27] S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *LNAI*. Springer-Verlag, 1997.

[28] G. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[29] J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, volume 667, pages 3–20. Springer-Verlag, 1993.

[30] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in LNCS, Venice, Italy, September / October 2003. Springer-Verlag.

[31] A. Srinivasan. Aleph manual, 2003.

[32] A. Srinivasan, R. D. King, S. Muggleton, and M. J. E. Sternberg. Carcinogenesis predictions using ILP. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297, pages 273–287. Springer-Verlag, 1997.