Strategies to Parallelize ILP Systems

Nuno A. Fonseca[1], Fernando Silva[1], and Rui Camacho[2]

[1] DCC-FC & LIACC, Universidade do Porto
R. do Campo Alegre 823, 4150-180 Porto, Portugal
{nf, fds}@ncc.up.pt
[2] Faculdade de Engenharia & LIACC, Universidade do Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
rcamacho@fe.up.pt

Abstract. It is well known by Inductive Logic Programming (ILP) practioners that ILP systems usually take a long time to find valuable models (theories). The problem is specially critical for large datasets, preventing ILP systems to scale up to larger applications. One approach to reduce the execution time has been the parallelization of ILP systems. In this paper we overview the state-of-the-art on parallel ILP implementations and present work on the evaluation of some major parallelization strategies for ILP. Conclusions about the applicability of each strategy are presented.

Key words: Parallelism, Scaling-up

1 Introduction

There are two major motivations for using ILP. First, ILP provides an excellent framework for learning in multi-relational domains. Second, the theories learned by general purpose ILP systems are in a high-level formalism often understandable and meaningful for the domain experts. We believe that these two reasons mostly explain the success of ILP systems in several well known industrial and scientific relevant problems [1,2,3,4]. The success usually comes at a price, and in the case of ILP systems the price is long execution times. For complex applications, ILP systems can take several hours, even days, to return a theory.

Research on reducing the execution time of ILP systems has deserved plenty attention in the last years. The proposed approaches are very diverse, ranging from new algorithms (see e.g., [5,6,7]), reducing the number of hypotheses generated (see e.g., [8,9,10]), to efficiently testing candidate hypotheses (see e.g., [11,12]), just to mention a few. A quite different line of research to reduce the execution time of ILP systems is through parallelization. This has been pointed out as a promising approach to improve efficiency by several researchers [13,14,15].

In this paper we survey the current state-of-the-art research on parallel ILP. The many implementations described in the literature are succinctly presented together with reported results. A comparison of the algorithms based only on

their reported results is hard since they were observed on different systems, datasets, and platforms. We thus implemented three parallel algorithms, that accomplish the main parallelization strategies that we have identified, and studied their performance using three well known applications and the same test environment (i.e., the same underlying ILP system and the same parallel architecture). The three parallel implementations were evaluated on a distributed memory architecture.

The remainder of this paper is organized as follows. Section 2 provides some background on parallelism and describes a generic ILP algorithm. Section 3 describes the main strategies to parallelize ILP systems and in Section 4 a survey of the parallel ILP implementation is made. In Section 5 is made an evaluation of three parallel algorithms. In Section 6 we present the conclusions.

2 Background

In this section we start by providing a small introduction to parallelism and then describe a generic sequential covering algorithm.

2.1 Parallelism

By expressing parallelism in an algorithm one aims to improve its performance. However, designing efficient parallel algorithms is still a difficult task as there are many factors that can influence efficient parallel execution, for example balancing the work among the different the available processors and controlling communication costs in a distributed parallel architecture.

In order to clarify the discussion about parallel algorithms in ILP, we shall first briefly define common terms. A *task* is typically a program (or set of instructions) that is executed by a processor. *Parallel tasks* are tasks whose computations are independent of each other, so that all such tasks can be performed simultaneously with correct results. The maximum number of tasks that can be executed simultaneously at any time in a parallel algorithm, determines the *degree of parallelism* of the application program. The granularity of a task measures the ratio between the time a task takes to be executed and the corresponding overhead time required to schedule that task. The higher the ratio (*coarse-grain parallelism*) the better to scale up parallel execution.

A sequential algorithm is usually evaluated in terms of its execution time (sometimes expressed as a function of the size of its input data). The execution time of a parallel algorithm depends on the number of processors used, interprocess communication speed, and size of the input data.

One would expect that increasing the number of processors results in a proportional decrease of the execution time of a program, but this is rarely observed due to overheads associated with parallelism. There are three major sources of overheads: interprocess communication, idling, and extra computation.

A number of performance metrics have been devised to be used in the study of parallel algorithms performance [16]. The *serial runtime* (T_S) of a program

is the time elapsed between the beginning and the end of its execution on a sequential computer. The *parallel runtime* (T_P) is the elapsed time from the beginning of the parallel computation until it ends. Speedup (S) is the most often used measure when studying the performance of parallel algorithms. It captures the relative benefit of solving a problem in parallel and is defined as the ratio between the time taken to solve a problem on a single processor and the time required to solve the same problem on a parallel computer with p identical processors.

$$S = \frac{T_S}{T_P}$$

Theoretically, the speedup can never exceed the number of processors p. In practice, a speedup greater than p, called *super-linear speedup*, is sometimes observed. This happens when the work performed by a sequential algorithm is greater than its parallel version or due to hardware features that slowdown the sequential algorithm (for instance, as a result of using slower memory, i.e., disk).

2.2 Generic ILP Algorithm

A plethora of rule learning algorithms [17], ILP algorithms included, use a variant of the generic *covering algorithm* (also called *separate-and-conquer*). An example of a generic covering algorithm is presented in Figure 1. This algorithm learns one rule at a time using some generalization procedure that performs a search through an ordered space of legal rules. After finding a rule, all covered positive examples are separated (removed) from the training set and the next rule is learned from the remaining examples. Rules are learned until no positive examples are left or some other stopping criteria is met.

---

covering(E^+, E^-, B)
Input: set of positive (E^+) and negative (E^-) examples, and background knowledge (B)
Output: A set of rules ($RulesLearned$)
1. $Rules\_Learned = \varnothing$
2. `while` $E^+ \neq \varnothing$ `do`
3. $R = learn\_rule(E^+, E^-, B)$
4. $Rules\_Learned = Rules\_Learned \cup \{R\}$
5. $B = B \cup \{R\}$
6. $E^+ = E^+ \setminus \{Examples\ Covered\ by\ R\}$
7. `end while`
8. `return` $Rules\_Learned$

Fig. 1. A generic covering algorithm. `learn_rule`() should return a (the best) rule that explains a subset of the positive examples(E^+).

Most ILP systems use some variant of the generic covering algorithm. The main difference between the existing ILP systems and algorithms that use a variant of this covering algorithm (e.g., [18,19,20]) concerns the `learn_rule()` procedure (step 3). Given a set of examples and prior knowledge, the procedure returns a consistent rule (clause) that explains some or all positive examples. This procedure is the most time consuming and will be described next in more detail.

---

learn_rule(E^+, E^-, B)
Input: set of positive (E^+) and negative (E^-) examples, and background knowledge (B)
Output: The "best" rule
1. $Good = \varnothing$
2. $S = START\_RULE$
3. $Pick = pickRule(S)$
4. $NewRule = genNewRule(Pick)$
5. $Val = evalOnExamples(NewRule)$
6. `if` $is\_good(NewRule, Val)$ `then` $Good = NewRule$`endif`
7. $S = S \setminus \{Pick\}$
8. `if` $stop\_criterium\_satisfied$ `then return` $bestOf(Good)$ `endif`
9. `goto` 3

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

The `learn_rule`() procedure, as described in Figure 2, searches the (potentially infinite) hypothesis space for a rule that optimizes some quality criteria. At each node of the search one rule is generated and evaluated. 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 large number of examples or/and when the computational cost of evaluating a rule is high.

3 Strategies for Parallelizing ILP Systems

Parallel algorithms aim to divide the work among the available processors so that a solution is achieved as fast as possible. The main difficulty faced by implementors is how to efficiently divide the work. Ideally, one would want to divide the computation and data evenly, and, at the same time, minimize the communication among processors, striving for a coarse-grained parallelism.

We classify the strategies to parallelize ILP systems described in the literature into four main approaches: parallel exploration of independent hypotheses [21]; parallel exploration of the search space [22,21,23,24]; parallel coverage test [21,25,26]; parallel execution of an ILP system over a partition of the

data [27,22,25]. Surely, one could consider other views, however, we consider that these cover the main approaches to parallelize an ILP system. A parallel algorithm may not fit solely in a single strategy, but may combine several. Each strategy is next described in detail.

3.1 Parallel Exploration of Independent Hypotheses

Parallel exploration of independent hypotheses is performed as follows. Let n be the number of classes of the target predicate. Learning each class value is an independent task and can be done in parallel. This procedure requires that each processor owns a replica of the whole data.

Parallel exploration of independent hypotheses has a major drawback: it is not a general approach. It is adequate only for applications where the target predicate is composed by several independent predicates. Learning a definition of the target predicate can be seen as learning several sub-concepts, corresponding each subconcept to a class value. Since the induction of sub-concepts is inherently independent, it can be easily performed in parallel, For instance, consider the task of learning a predicate that classifies emails into categories such as `priority(+Email,-Priority)`, where $Priority \in \{low, medium, high\}$. The task of learning can thus be divided into 3 subtasks, one learning task for `priority(+Email,low)`, other for `priority(+Email,medium)`, and `priority( +Email, high)`.

The degree of parallelism of this strategy corresponds to the number of subconcepts. The granularity is very high, since the learning of each subconcept corresponds to calling an ILP system to learn n sub-concepts independently.

3.2 Parallel Exploration of the Search Space

The search for a hypothesis involves traversing the generalization lattice in some way (e.g., top-down, bottom-up, bidirectional). The search space can be divided and explored in parallel by each processor to find a hypothesis.

The degree of parallelism and granularity of this strategy depends on the approach adopted to divide the search space.

3.3 Data Parallelism

Data parallelism consists in partitioning the data in subsets, assigning each subset of data to a processor. Each processor applies an algorithm (or part of an algorithm, e.g., coverage test) or the whole sequential ILP algorithm, on its local data. Generally, data partitioning is usually performed in the beginning of the execution. This happens because it is expensive to reassign the examples during execution, i.e., perform load-rebalancing.

A problem arises when a sequential ILP algorithm is applied to a subset of the training data: the hypotheses may be locally consistent and complete, but they may not be globally consistent. A solution to this problem may involve sharing the locally good hypotheses among all processors to obtain a global view.

Another problem, that results from partitioning the set of positive examples, is the impossibility of learning recursive rules. The only solution to this problem is the replication of the set of positive examples through all processors while dividing the set of negative examples.

The degree of parallelism of this strategy depends on the size of the data. The granularity depends on the algorithm applied to the dataset and size of the data.

3.4 Parallel Coverage Tests

The time to compute a hypothesis coverage depends on the cardinality of E^+ and E^-. Each example can be independently tested to determine if it is entailed by a rule h and the background knowledge B. The parallel coverage test strategy consists in performing the coverage test in parallel, i.e., for each example $e \in E$ the coverage test $(B \wedge h \vdash e)$ is performed in parallel.

The degree of parallelism depends on the number of examples evaluated in parallel by each processor. The granularity in this strategy is, relatively, low. However, the granularity can be enlarged either by increasing the number of examples of each processor or/and by evaluating several rules in parallel instead of a single one.

4 Parallel ILP Systems

We next survey the parallel ILP implementations, focusing on the strategy used and results reported.

The first parallel ILP system we are aware of is Claudien [27]. The algorithm followed a strategy based on parallel exploration of the search space where each processor keeps a pool of clauses to specialize, and shares part of them to idle processors (processors with an empty pool). In the end, the p set of clauses found are combined and returned as the solution. One should note that Claudien follows a non-monotonic setting of ILP instead of the usual normal ILP setting. The parallel system was evaluated on a shared-memory computer with two datasets and exhibited a linear speedup up to 16 processors.

Matsui et al. [22] evaluated and compared data parallelism (background knowledge and the set of examples) and, what they called, parallel exploration of the search space. The later approach consisted in evaluating, in parallel, the refinements of a clause, therefore, corresponding to a strategy based on parallel coverage tests. The two strategies were implemented in the FOIL [19] system and were evaluated on a distributed memory computer using the *trains* dataset [28]. The results of the search space parallel approach showed very low speedups. The reason pointed out by the authors for the poor results was 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 (background knowledge and the set of examples) 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.

Ohwada and Mizoguchi [21] implemented an algorithm (based on Inverse Entailment) using a logic programming parallel language that explored three types of parallelism: parallel coverage tests; parallel exploration of independent hypotheses; and parallel exploration of the search space (each processor followed a branch of the search space). The parallel system was applied to three variants of an email classification dataset and the experiments performed evaluated each strategy. The results on a shared-memory parallel computer showed a non linear speedup in all strategies. The strategy that appears to show better results, on average, was the parallel coverage tests.

Ohwada et. al [23] implemented an algorithm that explores the search space in parallel. The job allocation (set of nodes to be explored) was dynamic and was implemented using contract-net [29] communication. The parallel system was evaluated on two datasets and showed an almost linear speedup on a ten-processor parallel machine.

Wang and Skillicorn [25] parallelized the Progol [30] system by partitioning the data and applying a sequential algorithm to each partition. The data partitioning consisted in dividing the positive examples among all processors and by replicating the negative examples. Each processor applies the sequential algorithm on its local data to find a locally good clause. Such clause is then shared among all processors to evaluate its quality on the whole training set. If a processor considers that a clause is globally good then it exchanges this information with all processors, so that all processors may add the clause to the local theory and remove the examples explained by the clause. It is important to point out that this algorithm is not complete in relation to the sequential algorithm, i.e., the theory found by the parallel algorithm may be different to the one found with the sequential algorithm. The evaluation of the algorithm focused on speedup and did not allow the assessment of the impact on accuracy. They reported double and linear speedups in their experiments with three datasets. The experiments were performed on shared-memory machines (with 4 and 6 processors).

Graham et al. [26] implemented a parallel ILP system, using the PVM [31] message passing library. They employ data partition and parallel coverage tests of parts of the search space on each processor. They reported an almost linear speedup up to 16 processors on a shared memory machine.

Konstantopoulos [32] implemented a data parallel version of the Aleph [20] system using MPICH [33] MPI [34] library. His algorithm performs the coverage tests evaluation in parallel. This approach, although very similar to the one of Graham et al., it only evaluates in parallel a single clause at a time while Graham et al. evaluates a set of clauses. The smaller granularity of the parallel tasks, in Konstantopoulos' approach, is, probably, the main reason for the poor results presented.

Wielemaker [24] implemented a parallel version of Aleph for shared memory machines. The strategy adopted was parallel exploration of the search space. The algorithm exploits parallelism by executing concurrently several randomized local searches [6] and was implemented on top of the Aleph system. The

implementation was evaluated on the Carcinogenesis [4] dataset. The Aleph system was configured to perform 16 random restarts, and made 10 moves per restart, on each processor. The reported speedups (e.g., 7 on 16 processors) can be considered low when compared to other shared memory implementations. In spite of the results, this is an interesting proposal that could accomplish better results if the granularity of the tasks is enlarged. This can be easily accomplished by increasing the number of moves or the number of restarts done by each thread.

PolyFarm [35] 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 [35].

Strategy	**Arch.**	**Speedup/#procs.**	**Work**
Parallel exploration of the search space	Shared Memory	linear/16	[27]
		3/6	[21]
		7/16	[24]
		8/10	[23]
Parallel exploration of independent hypotheses	Shared Memory	2/6	[21]
Parallel coverage tests	Distributed Memory	1/15	[22]
		no	[32]
	Shared Memory	4/6	[21]
		5/8	[26]
Data Parallelism	Distributed Memory	4/15 (linear upto 5)	[22]
		not reported	[35]
	Shared Memory	linear and super-linear/6	[25]
		5/8	[26]

Table 1. Summary of the parallel ILP implementations and reported results

Table 1 summarizes the survey by presenting for each parallelization strategy the implementations made, targeted computer architecture, and reported results. The first observation concerns the fact that the majority of the parallel implementations were made for shared memory architectures, where the cost of data transmission is very low when compared to distributed memory architectures. In spite of the high cost of the communication, parallel ILP systems targeted for distributed memory computers may still achieve good speedups (see e.g., [22]). The results reported are generally good on all strategies except the parallel coverage test. The results reported with this strategy differ considerably

if the target architecture is shared memory or distributed memory. The poor results of the latter can be explained by the higher communication cost not being compensated by the granularity of the tasks.

Even though most implementations just described were for shared memory machines, we share the view of the recent work reported [32,26], that is, to target distributed memory architectures when parallelizing ILP systems, therefore favoring coarse grain approaches.

5 An Evaluation of Parallelization Strategies

In the previous section we summarized current state-of-the-art research on parallel ILP algorithms. It is hard to compare the results of the referred implementations since they were observed on different systems, platforms, and datasets. We implemented on a distributed memory architecture three parallel algorithms based on the most general strategies, namely parallel exploration of the search space, parallel coverage tests, and data parallelism. No algorithm was implemented based on parallel exploration of independent hypotheses because, as discussed before, this strategy is not applicable to all applications. By implementing these strategies on the same platform, using the same techniques to distribute work among the processing units, and the same applications, we were able to make a fair comparison.

5.1 Parallel algorithms

We started with a sequential implementation of the April [36] ILP system. The main loop of April's algorithm is similar to the covering algorithm presented in Section 2.2. For simplicity of presentation, all algorithms follow a master-worker scheme. In the beginning of the execution the worker enters a loop and waits for requests from the master. The master shares one processing unit with one of the workers.

The parallel algorithms were implemented using the Prolog language. For the communication layer we used LAM [37] MPI. LAM is a high-quality open-source implementation of the Message Passing Interface (MPI) specification, that can be used for applications running in heterogeneous clusters or in grids. Since the development was made in Prolog and LAM does not provide a native YAP Prolog interface, we had to develop a Prolog module for YAP, using the C language, to act as an interface between Prolog and LAM/MPI libraries.

The implemented algorithms are next described. For each algorithm, we refer the reader to Figure 3 for a schema of the messages exchanged between the master and the workers.

Parallel Coverage Tests (*pct*) A clause is dispatched for a processor to be evaluated on the local subset of examples. The master algorithm is similar to the covering algorithm of Section 2.2 with two main changes: first, the examples

are divided evenly among the processors in the beginning of the execution (this could be done in the first line of the `covering` algorithm) and are then loaded by each worker; secondly, line 5 of the `learn_rule` is changed to

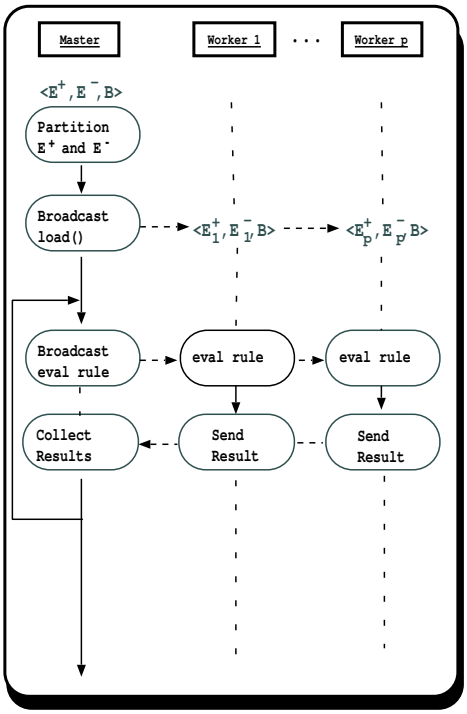
```
broadcast(evalOnExamples(NewRule))
Val = collectAndCombine()
```

where `broadcast()` is a procedure that sends a command to all processors to be executed, each processor executes the command and returns the result to the master. This corresponds to each slave evaluating a rule against its local set of examples and then returning the coverage value. The master collects and combines the coverage information using the `collectAndCombine()` procedure. This algorithm is basically the algorithm implemented by Konstantopoulos [32]. However, there are two main differences at the implementation level: i) we used asynchronous message passing communication for all operations involving the sending of a message, while Konstantopoulos only used synchronous message passing operations; ii) our implementation was done with LAM as opposed to the MPICH platform used by Konstantopoulos.

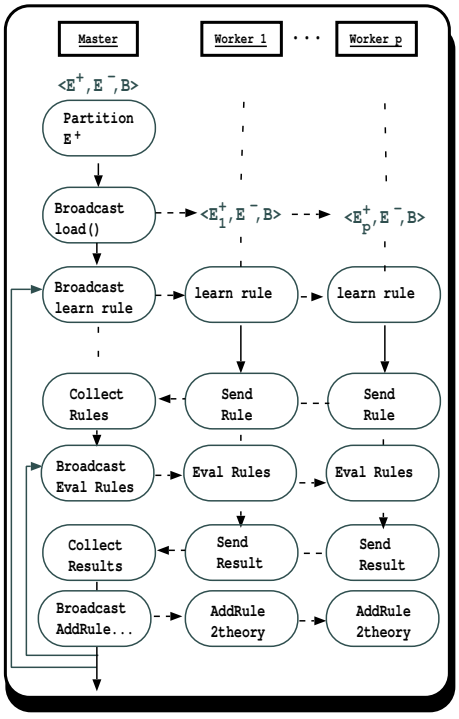
Data Parallel Learn Rule (*dplr*) This algorithm is based on the Wang et al. [25] algorithm mentioned in the previous section but it is next described in more detail.

```
Rules_Learned = ∅
< (E_1^+, E^-), ..., (E_p^+, E^-) >=partition  E^+  into p subsets
broadcast(load_files)
while ∪_{k=1}^{p} E_k^+ ≠ ∅ do
  RulesBag=collect(broadcast(learn_rule()))
  while RulesBag ≠ ∅ do
    Results=collect(broadcast(eval(RulesBag)))
    R=pickBest(RulesBag)
    RulesBag=RulesBag \ {R}
    Rules_Learned=Rules_Learned ∪ {R}
    collect(broadcast(addRule2Theory(R)))
  end while
end while
return Rules_Learned
```

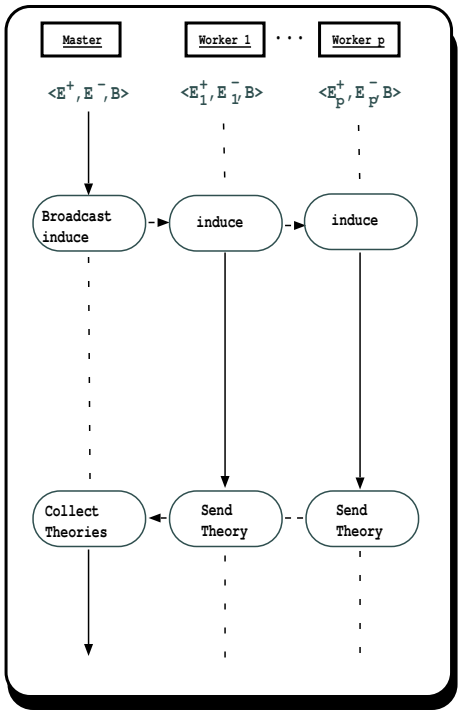
The algorithm consists of 1) dividing the set of positive examples among all processors and replicating the negative examples; 2) learning p rules in parallel (line 3 of the covering algorithm) starting at different points of the search space (using different seeds), where p is the number of processors available; 3) exchanging rules found among all processors to obtain their coverage values on the whole training set; 4) selecting a rule and mark examples covered on all subsets.



a) pct



b) dplr



c) dpilp

Fig. 3. Simplified schemes of the messages exchanges by the parallel algorithms. Solid lines represent the execution flow, horizontal dashed lines message passing between the processes, and vertical dashed lines idleness. The algorithms are ordered by the granularity of their parallel tasks, from the finest-grained to the most coarse-grained.

The `AddRule2theory(R)` performs steps 5 and 6 of the covering algorithm of Section 2.2, i.e., adds the rule to the background knowledge, marks the examples locally covered, and returns the number of examples locally covered.

Note that the first algorithm described in this section returns the same solution as the sequential algorithm, whereas this algorithm *may* not return the same solution due mainly to the order by which the rules are found and added to the theory.

Data Parallel ILP (*dpilp*) This algorithm starts by dividing the set of examples (positive and negatives) among all processors. It then induces p theories in parallel, using the covering algorithm on each subset, and then combines the p theories found using the whole training set. The combination of the theories (i.e., rules that compose the theories) can be made using several strategies (e.g., [38]). In order to make the comparison with the sequential algorithm more clear, we chose a simple strategy, very similar to the one used by the sequential algorithm. The rules are ordered using a metric (coverage in our implementation). The best rule is added to the theory and the remaining rules are reevaluated and reordered. The process is repeated while there are good rules to add to the theory. Like the previous algorithm, the solution returned by this algorithm may not be the same as the sequential version. It is obvious that this algorithm has the largest granularity of the three algorithms.

5.2 Materials

We used 3 ILP applications in the experiments. Table 2 characterizes the datasets used, in terms of number of examples (positive and negative) as well as background knowledge size (i.e., number of relations used in the learning task). AET is the average time required to test if an example is explained by a rule. This value is presented in microseconds and was estimated by dividing the sequential execution time by the number of examples evaluated during execution. This estimative of the cost of evaluating an example is a useful indicator when one considers the use of a parallelization strategy based on parallel coverage tests.

Dataset	$\mid E^+ \mid$	$\mid E^- \mid$	$\mid B \mid$	AET (μs)
Carcinogenesis [4]	162	136	38	305
Mesh [2]	2272	223	29	46
Mutagenesis [1]	114	57	21	20846

Table 2. Datasets Characterization.

The experiments were performed on a Beowulf Cluster composed by 8 nodes. Each node is a dual processor computer with 2GB of memory, and running the Linux Fedora OS. We used the YAP Prolog system version 4.5. The ILP

system was configured to perform breadth-first search to find a rule. The search was guided using a heuristic that relies on the number of positive and negative examples.

We used 3-fold cross validation. The evaluation was focused on training time speedup and accuracy. We measured the accuracy because two of the implemented parallel algorithms may produce theories different from the ones obtained with the sequential version. The accuracy variation is the ratio between the predictive accuracy observed when using P processors and the predictive accuracy observed when using a single processor.

Dataset	i-depth	Nodes	Noise	Minacc	CL
Carc	4	20000	5%	-	10
Mesh	4	10000	10%	85%	8
Mut	2	500	25%	70%	4

Table 3. Settings

We tuned the settings so that the sequential runs would not take more than one hour to complete (except for the **Mut** dataset). Table 3 shows the main settings used for each dataset. The parameter *nodes* specifies an upper bound on the number of rules generated during a search for a rule. The *i*-depth [39] corresponds to the maximum depth of a literal with respect to the head literal of the rule. *MinAcc* specifies the minimum accuracy that a rule must have in order to be considered good. The parameter *CL* defines the maximum length that a rule may have. Finally, the *noise* parameter defines the percentage of negative examples that a rule may cover in order to be accepted.

5.3 Results

Table 4 presents the execution time (in seconds) and speedups observed, on each dataset and algorithm, for 1, 2, 4, 6, 8 and 16 processors. Some runs were not performed for one of two reasons: i) no speedup would be achieved; ii) the subset of data associated to each processor becomes too small (for the *dplr* or *dpilp* algorithms).

The effects on execution time of the parallel coverage tests approach (*pct*) show quite different behaviors. In the **Carc** and **Mesh** datasets the parallel version is slower than the sequential one, while in the **Mut** dataset a considerable speedup is observed. Since the **Mut** dataset has less examples than the other two, we can only deduct that the higher cost of evaluating an example (see Table 2) is the main reason for the speedups. However, when the subset becomes too small, as is the case for 16 processors, we stop obtaining gains.

The poor results with the **Carc** and **Mesh** suggest that the distribution of the work, and consequent parallel evaluation of the examples, is not compensated by the cost of message passing. Clearly, this fine-grain approach to parallelize ILP

Dataset		1	2	4	8
Carc	T	416	554	999	-
	S		**0.75**	**0.42**	**-**
Mesh	T	717	948	912	901
	S		**0.76**	**0.79**	**0.80**
Mut	T	8,022	4,565	2,502	-
	S		**1.75**	**3.20**	**-**

a) *pct*

Dataset		1	2	4	8	16
Carc	T	416	311	180	530	-
	S		**1.34**	**2.31**	**0.79**	**-**
Mesh	T	717	1,904	1,347	608	592
	S		**0.38**	**0.53**	**1.18**	**1.21**
Mut	T	8,022	6,339	3756	-	-
	S		**1.26**	**2.13**	**-**	**-**

b) *dplr*

Dataset		1	2	4	8	16
Carc	T	416	347	129	81	-
	S		**1.20**	**3.23**	**5.17**	**-**
Mesh	T	717	260	207	165	164
	S		**2.75**	**3.46**	**4.35**	**4.37**
Mut	T	5,865	6,339	3,756		
	S		**1.26**	**2.13**	**-**	**-**

c) *dpilp*

Table 4. Execution time (T) and speedup (S)

system seems only suited for datasets with a complex background knowledge, where the cost of evaluating an example is high, or for very large datasets (as the number of examples is concerned) where the parallel evaluation of the examples on a subset outweighs the parallel overhead. A way of increasing the granularity of the parallel task is to evaluate in parallel a set of rules, as proposed in [35], instead of evaluating a single rule.

The impact of the *dplr* algorithm on the execution time is variable. In the **Carc** dataset a speedup is observed up to four processors and is nonexistent for eight processors. Interestingly, in the **Mesh** dataset, although we do not get a speedup for two and four processors, we observe a decrease in the execution time as the number of processors increases. A small speedup is observed with 8 and 16 processors.

One should note that the order by which the rules are found and added to the theory is a crucial factor to the execution time since it conditions the amount of the hypotheses space traversed. Recall that each worker gets a subset of E^+ but all E^-. If one of the workers does not find a globally "good" rule using its local subset, it will have to do a more extensive search. This may happen when a rule has an accuracy bellow the threshold, in the subset of the data where is being generated (thus not being considered good), and is above the threshold if the whole dataset is considered We observed that the final set of rules found by the *dplr* algorithm is far bigger than the set found by the sequential algorithm and that the rules are also lengthier. This suggests that the algorithm is unable to find a small number of simple rules.

The *dplr* is a master-worker implementation of the algorithm described by Wang et al. [25]. The results reported here are quite different from the ones pre-

Dataset	Alg.	2	4	8	16
Carc	dplr	+5%	+14%	-1%	-
(56%)	dpilp	+4%	-5%	+10%	-
Mesh	dplr	-25%	-25%	-	-
(72%)	dpilp	+17%	+21%	+23%	+25%
Mut	dplr	-2%	-13%	-	-
(86%)	dpilp	-1%	-12%	-	-

Table 5. Variation on predictive accuracy

viously reported. In [25] super-linear speedups (up to 6 processors) were reported while the speedups we found are not even linear or, in some cases, inexistent. The reason for this is two-fold. First, Wang et al. run the experiments in a shared memory machine while we run on a distributed memory machine (a cluster, in fact). Second, in the experiments performed by Wang et al. no good rules were lost while learning because they did not define parameters, such as minimum accuracy or minimum coverage. These parameters are used when considering if a rule is good or not. However, when dealing with real world applications, these parameters are often used to make the learning process more tractable and to discard rules with very low coverage that may represent "overfitting".

Since the *dplr* is not complete, when compared to the sequential algorithm, the predictive accuracy of the theory found may vary. In Table 5 we can see that predictive accuracy is affected negatively by the use of this algorithm. The reason for this is also related to loosing "good" rules while looking for a rule in the subsets. The theories found by the algorithm are composed by much more specific and lengthier rules than the ones found by the sequential version.

The results obtained with the *dpilp* algorithm are clearly the best ones. This algorithm not only provides a consistent speedup but can also improve the theory predictive accuracy. *dpilp* differs from *dplr* in the amount of negative examples. In *dpilp* a worker gets a percentage of the total negatives whereas in *dplr* each worker gets a percentage of the positives but all of the negatives. *dpilp* also needs much less communication among the processors. This confirms the theory that greater task granularity results in bigger speedups.

6 Final Remarks

This paper has two main contributions: first, it surveys the state-of-the-art on parallel ILP implementations; secondly, the performance impact of three parallel algorithms on a distributed memory computer is studied using real world applications.

The parallel ILP algorithms described in the literature were grouped into four main approaches: parallel exploration of independent hypotheses; parallel exploration of the search space; parallel coverage test; parallel execution of an ILP system over a partition of the data. Parallel exploration of independent

hypotheses is not a general approach since it is only adequate for applications where the target concept is composed by several independent subconcepts. However, when this approach is applicable it can be combined with other approaches to learn the subconcepts.

Three algorithms were implemented based on the three more generic strategies: parallel exploration of the search space; parallel coverage test; parallel execution of an ILP system over a partition of the data. The results show that a good approach to parallelize ILP systems in a shared-memory computer is one of the simplest to implement: divide the set of examples into p subsets; run the ILP system in parallel on each subset; combine the theories found. This approach not only reduces the execution time but can also improve predictive accuracy.

We have also noticed a significant difference between shared and distributed memory machines. In shared memory machines the communication overhead is significantly reduced and strategies, like *dlpr*, may give super-linear speedups. However, in distributed memory machines, where the communication costs are higher, fine-grained strategies are severely penalized.

A natural extension of this work is to perform a larger experimental evaluation over a greater number of datasets. This could provide us more insights about the applicability of each strategy. It would also be interesting to extend the evaluation of the strategies to shared memory architectures.

Acknowledgments We are thankful to the anonymous referees for their valuable comments. The work presented in this paper has been partially supported by project APRIL (Project POSI/SRI/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.

References

1. A. Srinivasan, S. Muggleton, R.D. King, and M.J.E. Sternberg. Mutagenesis: Ilp experiments in a non-determinate biological domain. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, pages 217–232, 1994.
2. 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.
3. Muggleton S., King R.D., and Sternberg M.J.E. Predicting protein secondary structure using inductive logic programming. *Protein Engineering*, (5):647–657, 1992.
4. 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.
5. Lappoon R. Tang, Raymond J. Mooney, and Prem Melville. Scaling up ilp to large examples: Results on link discovery for counter-terrorism. In *Proceedings of the KDD-2003 Workshop on Multi-Relational Data Mining (MRDM-2003)*, pages 107–121, 2003.

6. F. Železný, A. Srinivasan, and D. Page. Lattice-search runtime distributions may be heavy-tailed. In S. Matwin and C. Sammut, editors, *Proceedings of the 12th International Conference on Inductive Logic Programming*, volume 2583 of *LNAI*, pages 333–345. Springer-Verlag, 2003.
7. A. Srinivasan. A study of two probabilistic methods for searching large spaces with ilp. Technical Report PRG-TR-16-00, Oxford University Computing Laboratory, 2000.
8. Rui Camacho. Improving the efficiency of ilp systems using an incremental language level search. In *Annual Machine Learning Conference of Belgium and the Netherlands*, 2002.
9. A. Srinivasan, R.D. King, and M.E. Bain. An empirical study of the use of relevance information in inductive logic programming. *JMLR*, 2003.
10. Nuno Fonseca, Vitor Santos Costa, Rui Camacho, and Fernando Silva. On avoiding redundancy in Inductive Logic Programming. In Rui Camacho, Ross D. King, and Ashwin Srinivasan, editors, *Proceedings of the 14th International Conference on Inductive Logic Programming*, volume 3194 of *Lecture Notes in Artificial Intelligence*, pages 132–146, Porto, Portugal, September 2004. Springer-Verlag.
11. 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.
12. V.S. Costa, A. Srinivasan, R. Camacho, H. Blockeel, and W. Van Laer. Query transformations for improving the efficiency of ilp systems. *JMLR*, 2002.
13. Luc De Raedt. A perspective on inductive logic programming. In *The logic programming paradigm - a 25 year perspective*, pages 335,346. Springer-Verlag, 1999.
14. David Page. ILP: Just do it. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *LNAI*, pages 3–18. Springer-Verlag, 2000.
15. David Page and Ashwin Srinivasan. Ilp: a short look back and a longer look forward. *J. Mach. Learn. Res.*, 4:415–430, 2003.
16. Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2nd edition, 2003.
17. Johannes Fürnkranz. Separate-and-conquer rule learning. *Artificial Intelligence Review*, 13(1):3–54, February 1999.
18. S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
19. 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.
20. Ashwin Srinivasan. Aleph manual, 2003.
21. Hayato Ohwada and Fumio Mizoguchi. Parallel execution for speeding up inductive logic programming systems. In *LNAI*, number 1721, pages 277–286. Springer-Verlag, 1999.
22. 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.
23. Hayato Ohwada, Hiroyuki Nishiyama, and Fumio Mizoguchi. Concurrent execution of optimal hypothesis search for inverse entailment. In J. Cussens and A. Frisch, editors, *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *LNAI*, pages 165–173. Springer-Verlag, 2000.
24. Jan Wielemaker. Native preemptive threads in swi-prolog. In *ICLP*, pages 331–345, 2003.

25. Y. Wang and D. Skillicorn. Parallel inductive logic for data mining. In *Workshop on Distributed and Parallel Knowledge Discovery, KDD2000*, Boston, 2000. ACM Press.
26. James Graham, C. David Page, and Ahmed Kamal. Accelerating the drug design process through parallel inductive logic programming data mining. In *Proceeding of the Computational Systems Bioinformatics (CSB'03)*. IEEE, 2003.
27. L. Dehaspe and L. De Raedt. Parallel inductive logic programming. In *Proceedings of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, 1995.
28. R.S. Michalski. Pattern recognition as rule-guided inductive inference. In *Proceedings of IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 349–361, 1980.
29. R.G. Smith. "The contract net protocol: High-level communication and control in a distributed problem solver". *IEEE Trans. Computers*, 29(12):1104–1113, Dec 1980.
30. Stephen Muggleton and John Firth. Relational rule induction with cprogol4.4: A tutorial introduction. In Saso Dzeroski and Nada Lavrac, editors, *Relational Data Mining*, pages 160–188. Springer-Verlag, September 2001.
31. Pvm:parallel virtual machine. http://www.csm.ornl.gov/pvm/.
32. Stasinos 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.
33. 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, September 1996.
34. Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.
35. Amanda Clare and Ross D. King. Data mining the yeast genome in a lazy functional language. In *PADL*, pages 19–36, 2003.
36. Nuno Fonseca, Fernando Silva, Rui Camacho, and Vitor S. Costa. Induction with April - A preliminary report. Technical Report DCC-2003-02, DCC-FC & LIACC, Universidade do Porto, 2003.
37. Jeffrey M. Squyres and Andrew 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.
38. Ronaldo Cristiano Prati and Peter Flach. Roccer: an algorithm for rule learning based on roc analysis. In *Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'2005)*, 2005.
39. S. Muggleton and C. Feng. Efficient induction in logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.