From sequential to Parallel Inductive Logic Programming

Rui Camacho

LIACC, Rua do Campo Alegre, 823, 4150 Porto, Portugal
FEUP, Rua Dr Roberto Frias, 4200-465 Porto, Portugal
rcamacho@fe.up.pt
http://www.fe.up.pt/∼rcamacho
tel: +351 22 508 1849 fax: +351 22 508 1443

Abstract. Inductive Logic Programming (ILP) has achieved considerable success in a wide range of domains. It is recognized however that efficiency is a major obstacle to the use of ILP systems in applications requiring large amounts of data. In this paper we address the problem of efficiency in ILP in three steps: i) we survey speedup techniques proposed for sequential execution of ILP systems; ii) we survey different ways of parallelizing an ILP system and; ii) adapt and combine the sequential execution speedup techniques in the parallel implementations of an ILP system. We also propose a novel technique to partition the search space into independent sub-spaces that may be adequately searched in parallel.

keywords: Inductive Logic Programming, Parallel Automatic Data Analysis

1 Introduction

Inductive Logic Programming (ILP) has achieved considerable success in a wide range of domains. It is recognized however that efficiency is a major obstacle to the use of ILP systems in applications requiring large amounts of data. Relational Data Mining applications are such an example where efficiency is very important.

To address the problem of efficiency in ILP we first look at improvements in their sequential execution. Techniques have been proposed that substantially improve the sequential execution time. We then investigate the approaches currently adopted for the parallel execution of ILP systems. To gain the most we look at the combination of the two approaches. Some of the sequential improvements may be directly used in a parallel context whereas some others have to be adapted or partially used. The gains with the proposed combinations dependent on the parallel strategy used and can involve an increase in the communication costs.

In this paper we also propose a novel technique for parallel execution of ILP systems. Our proposal is inspired on a Logic Programming and-parallelism technique of independence of goals. It consists in the partitioning of the search space into independent sub-spaces that can therefore be searched in parallel.

A parallel implementation of an ILP algorithm may: i) improve the quality of the solutions found by searching more space in the same time of the sequential execution and/or; process larger datasets distributing the examples among the computing nodes (loading all of then in a single node may be impossible in some cases) or; get the same solution of the sequential execution much faster.

The rest of the paper is organised as follows. Section 2 presents the ILP framework. Sequential execution speedups are surveyed in Section 3. Section 4 surveys the current parallel execution approaches. In Section 5 we identify which and how sequential speedup techniques can be adapted for a parallel execution setting. In Section 6 we present our approach to define and search independent sub-spaces. Finally we draw the conclusions in Section 7.

2 The ILP framework

The objective of an ILP system is the induction of logic programs from a set of examples (E) and background knowledge (B). Examples are of two kinds: positive examples (E^+) are instances of the target concept and; negative examples (E^-) are not instances of the target concept. Background knowledge is any kind of information the domain expert thinks relevant for defining the target concept. Both examples and background knowledge are usually represented as logic programs. An ILP system induces a logic program where positive examples succeed and negative examples fail.

By means of defining a generalization relation over the set of clauses, ILP maps the induction process into a graph search. Thus, learning can be seen as searching for a correct theory. The states in the search space (designated as *hypothesis space*) are concept descriptions (hypothesis) and the goal is to find one or more states satisfying some quality criterion. The search space may be "generated" by the transitive closure of a refinement operator [4] starting on the most general clause. Refinement operators generalize or specialize hypotheses, thus generating more hypotheses.

Each hypothesis generated during the search is evaluated to determine their quality. A widely used approach to score a hypothesis is by measuring its coverage. The coverage of an hypothesis h is the number of positive (*positive cover*) and negative examples (*negative cover*) derivable from $B \wedge h$.

A major problem in ILP is that the search space is, in general, very large and in most cases infinite. In some domains the coverage computation is computationally very expensive due to: a large number of examples or; proving each example is computationally hard (high non-determinism in the background knowledge predicates)[5].

To constrain the size of the search space clauses arguments are assigned a type and I/O sign. These constraints are specified by means of *mode declarations*[4].

3 Performance improvements for Sequential Execution

The general idea of ILP systems being slow is based on essentially two items of the technique: the number of nodes to be searched and; the amount of computation effort used to evaluate each hypothesis (node). Therefore, to get the most on improving the sequential execution efficiency one has to tackle both items.
To address the first item a system may take advantage of techniques to avoid the generation of useless hypothesis that is, redundancy avoidance techniques. One approach is to profit from special purpose declarations such as transitivity and commutativity declarations or expert provided declarations to avoid redundancy.
Techniques to reduce the computation effort when evaluating an hypothesis include the following ones. *Lazy evaluation* of examples [1] aims at the evaluation of the minimum number of examples necessary to decide if an hypothesis needs to be specialised, accepted or abandoned. *Query packs* [6] avoid recomputation of goals (literals) that are common among a parent clause and its direct refinement clauses. This is achieved by making the evaluation of a clause and its refinements together. *Exact transformations* of queries [2] transform clauses into equivalent ones that are more efficiently evaluated by a theorem prover. The transformation depends on characteristics of the clauses. *User defined refinement operators* allow the expert to constrain the clause refinements to only the ones he thinks useful. *Caching* [8] avoids recomputation of previously obtained results.

4 Parallel execution

As pointed out by Page [3] parallelization of an ILP system is a very promising line of research to overcome the efficiency bottleneck. The strategies used so far to parallelize ILP systems can be categorized as: parallel exploration of the search space [9]; parallel coverage test [9]; and parallel execution of an ILP system over a partition of the data [10]. Most of the techniques referred in the previous section are still applicable in a parallel execution setting and therefore substantial improvement on efficiency may be gained through the combination of the results of both lines of research.
Since the ILP induction step is performed as a search through a graph (generalisation lattice) we can assign different parts of the search graph to different nodes in a parallel execution setting. When evaluating individual hypothesis (their coverage) we may use a parallel executing theorem proving (parallel Prolog engine, for example). Data partitioning is performed by distributing the set of examples among all processors. This approach is of capital importance for applications where the dataset does not fit into the memory of a single processor node.

5 From Sequential to Parallel execution

In this section we investigate which sequential execution speedups can be directly used in a parallel execution setting and which ones have to be *adapted*.

Techniques such as *caching*, *query packs*, the *exact transformations* and the *user defined refinement operators* can be directly used in any node of a parallel execution approach without any overhead of communication or computational cost. Redundancy avoidance using special purpose declarations may be directly used when data splitting is adopted in a parallel setting. In such approach each node searches the same space of the sequential case, only the computational effort to evaluate an hypothesis is reduced. However in the case of hypothesis space split there may be extra communication among the nodes to avoid global redundancy. If no communication is made different nodes may produce redundant clauses in the overall set of produced clauses.

We distinguish two cases of lazy evaluation: lazy evaluation of negative examples and; *total laziness*. In the first case only negative examples are lazily evaluated whereas in the second the positive examples are evaluated only after the lazy evaluation of the negatives and only if the clause is consistent with the negatives. In the lazy evaluation of the negatives the system just evaluates examples until either it used all of them or it has surpassed the allowed number of negatives to be covered (the noise level). If the noise level is zero then lazy evaluation may be directly used in any parallel setting approach. We can still use lazy evaluation of the negatives directly and without any restrictions in an *hypothesis space split* parallel setting. When data is split and the noise level is not zero we may still take advantage of lazy evaluation by adopting the global noise level as the threshold for negative lazy evaluation at each processor node. Only with extra communication effort would we achieve a full lazy evaluation of negatives making sure that the global computation of the negative cover of the current hypothesis evaluates the minimum number of examples. *Total laziness* may be implemented in a parallel setting with some extra communications. After (lazily) evaluating the negatives the processor nodes have to establish if the clause is consistent or not. In case the clause is not consistent then the positive examples are not evaluated. A summary of the results of this section are shown in Figure 1.

technique	**hypothesis space split**	**data split**
exact transformation	no	no
caching	no	no
query packs	no	no
redundancy avoidance	extra communication	no
lazy evaluation:		
negs	no	extra communication (when noise > 0)
total	no	extra communication

Fig. 1. Extra effort in implementing sequential execution speedups in a parallel setting.

6 Independent Sub-spaces

We propose a novel approach to parallel execution of ILP systems that is inspired on a technique from Logic Programming (LP) and-parallelism. We use the LP result, together with the expert provided mode declarations (Shapiro [4]) to define sub-spaces that are searched independently (in parallel). Each sub-space is searched in an individual node and contains only clauses that can be generated using the subset of mode declarations assigned to that processor node. The procedure at each processor node searches a sub-space in a "normal ILP" fashion. Hypotheses are generated, evaluated and refined. Evaluation of hypotheses is done using a theorem proving to establish which examples are explained by each hypothesis. The technique we are proposing requires a post-processing step in order to attain completeness of the induction process. In the post-processing phase clauses from different sub-spaces have to be combined and evaluated in order to keep the search complete. The advantage of the proposal is that the coverage of clauses in the post-processing phase is the intersection of the coverages of the clauses being merged. This results from the fact that the literals of each original clause constitute a set of independent goals as was proved by the LP result. If coverage of an hypothesis is stored in a bit array the *bitwise AND* operation implements the coverages intersection and is a very efficient computer operation when compared with theorem proving the examples.

The LP result we used is based on the observation that when executing a conjunction of goals $G_1, ..., G_n$, failure of a goal G_i will result in attempting to generate more solutions for goals earlier in the sequence. This effort is useless if these solutions do not alter the computation of G_i. The and-parallelism exploits the notion of *goal independence* by partitioning the goals in a clause into classes of independent goals. It will execute each class in a separate processor node. In our approach the classes of independent goals are generated using the mode declarations.

In pure logic programs, goals depend on each other because they share variables. Given a function $vars(T)$ that returns all variables in the term T, two literals G_i and G_j are said to *share*, that is the relation $Shares(G_i, G_j)$ holds, when: $i = j \ \vee \ vars(G_i) \cap vars(G_j) \neq \emptyset$ The definition ensures that the relation $Shares$ is reflexive and symmetric. Its transitive closure $Linked$, defined as the smallest transitive relation that is a superset of $Shares$, is an equivalence relation. If two goals are in the same equivalence class, we will say they may be *dependent*, otherwise we will say they are *independent*. We would like to divide the original clause into several conjunctions of independent goals and execute them separately. Using the mode declarations which specify types and I/O signs for the predicates arguments we define the sub-spaces assuring that clauses belonging to different sub-spaces do not share variables.

Unfortunately, checking whether goals are independent is quite expensive. Ideally, we would like to do so only once when we compile a newly induced clause, not for every time we run the clause against an example. Our proposal addresses that concern and uses the mode declarations to guarantee the independence of the body literals of two clauses belonging to any different sub-spaces.

7 Conclusions

We have surveyed the techniques to improve the performance of sequential execution of ILP systems. We have also surveyed the different approaches taken to execute ILP systems in a parallel framework. We have proposed in the paper ways of adapting some of the sequential execution speedups in a parallel execution setting. We have also proposed a novel technique to execute ILP systems in a parallel environment. The proposal is inspired on a Logic Programming and-parallelism result and consists in establishing a partition of the search space into independent sub-spaces.

Acknowledgments

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, FCT* and *Programa POSI*.

References

1. Camacho, Rui, As lazy as it can be, 8th Proceedings of the Scandinavian Conference on AI, ed. Biornar Tessen et al. IOS press, 47-58, (2003).
2. V. Costa and A. Srinivasan and R. Camacho and H. Blockeel and B. Demoen and G. Janssens and J. Struyf and H. Vandecasteele and W. Van Laer, Query Transformations for Improving the Efficiency of ILP Systems, Journal of Machine Learning Research, (2002)
3. Page, David, ILP: Just Do It, Proceedings of the 10th International Conference on Inductive Logic Programming, Springer-Verlag, LNAI, vol 1866, (2000).
4. Shapiro, E.Y., Algorithmic Program Debugging, The MIT Press, (1983).
5. M. Botta and A. Giordana and L. Saitta and M. Sebag, Relational learning: hard problems and phase transitions, 178–189, Proc. of the 6th Congress AI*IA, LNAI 1792, Springer-Verlag, (1999).
6. Blockeel, H. and Dehaspe, Luc and Demoen, B. and Janssens, G. and Ramon, J. and Vandecasteele, H. Executing Query Packs in ILP, Proceedings of the 10th International Conference on Inductive Logic Programming, LNAI, vol 1866, (2000).
7. A. Srinivasan. A study of two sampling methods for analysing large datasets with ILP. *Data Mining and Knowledge Discovery*, 3(1):95–123, 1999.
8. James Cussens, Part-of-speech disambiguation using ILP, Oxford University Computing Laboratory, PRG-TR-25-96, (1996).
9. Hayato Ohwada and Fumio Mizoguchi. Parallel execution for speeding up inductive logic programming systems. In *Lecture Notes in Artificial Intelligence*, number 1721, pages 277–286. Springer-Verlag, 1999.
10. 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.
11. S. Muggleton. Optimal layered learning: A PAC approach to incremental sampling. In *Proceedings of the 4th Conference on Algorithmic Learning Theory*, Springer-Verlag, 1993.