Mach Learn (2009) 74: 257–279
DOI 10.1007/s10994-008-5094-2

Parallel ILP for distributed-memory architectures

Nuno A. Fonseca · Ashwin Srinivasan · Fernando Silva · Rui Camacho

Received: 16 April 2007 / Revised: 23 November 2008 / Accepted: 25 November 2008 /
Published online: 19 December 2008
Springer Science+Business Media, LLC 2008

Abstract The growth of machine-generated relational databases, both in the sciences and in industry, is rapidly outpacing our ability to extract useful information from them by manual means. This has brought into focus machine learning techniques like Inductive Logic Programming (ILP) that are able to extract human-comprehensible models for complex relational data. The price to pay is that ILP techniques are not efficient: they can be seen as performing a form of discrete optimisation, which is known to be computationally hard; and the complexity is usually some super-linear function of the number of examples. While lit-

Editor: Hendrik Blockeel.

This is an extended version of the paper entitled *Strategies to Parallelize ILP Systems*, published in the *Proceedings of the 15th International Conference on Inductive Logic Programming (ILP 2005)*, vol. 3625 of LNAI, pp. 136–153, Springer-Verlag.

N.A. Fonseca (✉)
Instituto de Biologia Molecular e Celular (IBMC) & CRACS, Universidade do Porto, Rua do Campo Alegre, 823, 4169-007 Porto, Portugal
e-mail: nf@ibmc.up.pt

A. Srinivasan
IBM India Research Laboratory, Block 1, Indian Institute of Technology, Hauz Khas, New Delhi 110 016, India
e-mail: ashwin.srinivasan@in.ibm.com

A. Srinivasan
Department of CSE & Centre for Health Informatics, University of New South Wales, Sydney NSW 2052, Australia

F. Silva
CRACS & Faculdade de Ciências, Universidade do Porto, Rua do Campo Alegre 1021, 4169-007 Porto, Portugal
e-mail: fds@ncc.up.pt

R. Camacho
LIAAD & Faculdade de Engenharia, Universidade do Porto, Rua Dr. Roberto Frias, 4200-465 Porto, Portugal
e-mail: rcamacho@fe.up.pt

tle can be done to alter the theoretical bounds on the worst-case complexity of ILP systems, some practical gains may follow from the use of multiple processors. In this paper we survey the state-of-the-art on parallel ILP. We implement several parallel algorithms and study their performance using some standard benchmarks. The principal findings of interest are these: (1) of the techniques investigated, one that simply constructs models in parallel on each processor using a subset of data and then combines the models into a single one, yields the best results; and (2) sequential (approximate) ILP algorithms based on randomized searches have lower execution times than (exact) parallel algorithms, without sacrificing the quality of the solutions found.

Keywords ILP · Parallelism · Efficiency

1 Introduction

The rapid growth of large-scale scientific and industrial databases has brought into focus the need for methods that can discover trends and predictive patterns in data, and communicate them in a manner designed to provoke insight. For example, public databases describing complete genomes of several hundreds of organisms are now available, yet little is known about the function of most of the coded genes in these databases (for *S. cerevisae*—baker's yeast—an organism that has been the subject of intense study, only 60% of the known genes can be assigned a function with any degree of confidence). Molecular biology is just one field in which there is now vastly greater quantities of data than can be analysed manually. A similar trend is observed in organisations which are besieged by large amounts of machine-generated text data (XML documents for example), with little hope of manually extracting information from them. To bridge this gap, it seems inevitable that machine assistance with analysis and modelling would be needed. Of particular interest here are techniques capable of extracting human-comprehensible models for data. At the cutting-edge of such techniques is Inductive Logic Programming (ILP).

Computer programs now collectively termed 'predictive ILP' systems use as input domain-specific background knowledge (B) and pre-classified sample data (E), both encoded in a subset of first-order logic, and construct, as output, a set of rules (clauses) encoded in first-order logic. These rules allow the assignment of classification labels for new data. The principal attractions of the use of ILP are: (1) the ability to take advantage of accumulated expertise in the form of background knowledge; and (2) the use of first-order logic, which can express complex relational descriptions while being amenable to translation to a human-comprehensible form.

The most successful proof-of-concept applications of ILP have been in the areas of biology and chemistry (King 2004; Srinivasan et al. 1994a; Marchand-Geneste et al. 2002; Turcotte et al. 2001). Other applications include tasks in language processing (Cussens 1997; Tang and Mooney 2001; Boström 2000); environmental data monitoring (Džeroski et al. 2000); music (Tobudic and Widmer 2003); and mathematical discovery (Colton and Muggleton 2003; Todorovski et al. 2004). Despite considerable diversity in the applications of ILP, successful implementations have been relatively uniform, namely, programs that repeatedly examine sets of candidate rules to find the 'best' ones. To a good first approximation, the basic task addressed by most ILP implementations can thus be viewed as a discrete optimisation problem: Given a finite discrete set S and a function $f: S \rightarrow \Re$, find an element $s \in S$ such that $f(s) = \min_{s_i \in S} f(s_i)$. Here f and $f(s_i)$ should more correctly be written as $f(s_i, B, E)$. The optimisation problem can either be posed as finding the best model for all

of the data E (S is then the set of all candidate models), or some subset of them (S is then the set of candidate rules for that subset). The latter corresponds to the task faced by ILP implementations that employ a 'greedy' procedure that constructs a model by finding locally optimal rules for subsets of the data. The restriction to finite S, although not required by ILP theory, is often enforced in practice. Branch-and-bound algorithms can be shown to identify optimal solutions for such optimisation problems (Papadimitriou and Steiglitz 1982). The search executed by a number of prominent ILP systems (for example, Muggleton 1995; Quinlan and Cameron-Jones 1993) can be formulated as instances of such an algorithm.

Despite the promising results cited earlier, current ILP implementations have shortcomings that prevent their routine use in data analysis. These can be broadly categorised into: (a) those that concern the formalism; and (b) those that concern efficiency. In this paper we only attempt to address the issue of efficiency since it will persist even if formalism issues are resolved. Specifically, we intend to investigate the utility, if any, of parallel execution to alleviate some of the main efficiency bottlenecks in an ILP system. The principal efficiency concerns for any ILP implementation are threefold:

1. The space of possible models can be extremely large (in the worst-case, the discrete optimisation problem can be NP-hard). This occurs for real problems where not enough is known to restrict the space via the use of constraints.
2. Datasets can be extremely large. An example of this is provided by various biological databases arising from sequencing animal and plant genomes. Constructing models for such data requires an ILP system to be able to handle millions of data items.
3. Testing the models can be difficult. Botta et al. (2003) show that there is a kind of 'phase-transition' behaviour: namely, circumstances under which evaluation can become extremely hard even if model-spaces and datasets are quite small.

On the face of it, "parallel ILP" would appear to be futile: to achieve polynomial run-times for NP-hard problems would require an exponential number of processors. However there are worthwhile practical reasons for embarking on such an enterprise:

1. The proof-of-concept applications of ILP suggest that ILP techniques are immediately applicable to cutting-edge problems that arise in areas like functional genomics. However, it is also evident from the applications of ILP that the efficiency will have be to significantly improved to handle the vast quantities of information stored in modern databases. Distributed storage and computation could alleviate some of these issues;
2. Stochastic algorithms have been suggested to address each of the three efficiency concerns (Sebag and Rouveirol 1997; Srinivasan 1999; Železný et al. 2002). However, by their nature, these can only guarantee approximate solutions. A parallel implementation of an exact algorithm like branch-and-bound would return exact solutions (parallel stochastic algorithms would still return approximate solutions);
3. There is a wealth of literature on parallel algorithms that can be adapted to ILP. Grama et al. (2003) provide a detailed survey of parallel search algorithms. Considerable research has also been done within the field of logic programming on the parallel evaluation of rules.[1] Much of this relates directly to the problem of model evaluation. There is now an international workshop dedicated to issues arising in parallel and distributed data mining (International Workshop on High Performance and Distributed Mining). Very little has been done to draw these different strands of research into the development of an efficient ILP implementation;

[1] See: http://www.cs.nmsu.edu/~epontell/adventure/paper.html for a good survey.

4. The rapid increase in the performance of microprocessors has increasingly resulted in the use of 'clusters' of personal computers as low-cost alternatives to special-purpose supercomputers. There is now sufficient hardware and software support for these architectures to expect that they will become commonplace in the near future. It is mandatory that ILP systems be capable of exploiting fully the processing power afforded by such hardware environments.

In this paper, we investigate the effectiveness of parallel implementations of concurrent search through model spaces; concurrent use of sample data; and concurrent evaluation of rules. Each of these has direct relevance to the three efficiency concerns raised earlier (large model spaces, large datasets, and difficult model checking). In particular we wish to evaluate the gains, if any, from using exact algorithms devised for a cluster of processors as opposed to exact or approximate single processor algorithms. While the implementations we have developed can be applied to any field of endeavour, we propose to concentrate on some biochemical problems and an engineering problem that now constitute standard benchmarks for ILP.

2 Predictive ILP

2.1 Specification

We follow closely the specification provided by (Muggleton 1994) for an ILP system designed to construct models (usually called hypotheses in the ILP literature) given background knowledge B and observations (usually called examples in the ILP literature) E. In this specification an ILP algorithm is one that satisfies the following requirements (reproduced with minor changes from Srinivasan and Kothari 2005):

Given:

R1. $B \in \mathcal{B}$: background knowledge encoded as statements in logic.
R2. $E \in \mathcal{E}$: a finite set of examples $= E^+ \cup E^-$ where:
R3. $\mathcal{L}$: a pre-defined language for acceptable clauses.
R4. Examples
$E^+ = \{\underline{e_1}, \underline{e_2}, \ldots, \underline{e_p}\}$ is a set of definite clauses (these are the positive examples);
$E^- = \{\overline{f_1}, \overline{f_2}, \ldots, \overline{f_n}\}$, is an optional set of Horn clauses (these are the negative examples); and
$B \not\models E^+$

Find:

R5. $H = \{D_1, D_2, \ldots, D_k\}$: a set of clauses from the set $\mathcal{D}$ of all possible clauses in $\mathcal{L}$ such that the following conditions are met:
Sufficiency. This consists of:
$S1$. $B \cup H \models E^+$
$S2$. $B \cup \{D_i\} \models e_1 \vee e_2 \vee \cdots \vee e_p$ $(1 \leq i \leq k)$.
Consistency. This consists of:
$C1$. $B \cup H \not\models \Box$; and
$C2$. $B \cup H \cup E^- \not\models \Box$

The requirements $S1$ and $S2$ ensure that H constitutes an explanation for the positive examples and that each hypothesis D_i in H explains at least one positive example. The requirement $C1$ ensures that H does not violate any of the constraints I in B. The requirement

$C2$ is intended to ensure that H does not contain any over-general clauses. Often, implementations do not require clauses to meet this requirement, as some members of E^- are taken to be noisy. This specification is then refined to allow theories to be inconsistent with some negative examples. We will use the phrase "H explains E, given B" to denote that at least $S1$, $S2$ and $C1$ are met. An "acceptable H" is any H that explains E, given B.

2.2 Implementation

The specification does not state how acceptable H's are to be constructed, or, if several H's explain the E, then which of them are to be selected. A number of ILP implementations (such as Muggleton 1995; Muggleton and Feng 1990; Quinlan 1990) find an acceptable H by executing some variant of the greedy cover set procedure shown in Fig. 1, that constructs H incrementally and on each iteration i finds the best clause D_i that can be added to the set H_{i-1} (with $H_0 = \emptyset$). In (Železný et al. 2006) a generic search procedure is described, which provides the template for a range of different implementations. Different procedures are obtained by specific choices made on the choice of refinement operator (general-to-specific, specific-to-general, or both), whether the procedure involves repetitions with different starting points, the kind of pruning employed and so on.

Our interest in this paper is in two particular search algorithms: a deterministic general-to-specific (or "top-down") exhaustive branch-and-bound search, which we denote *DTD*; and the rapid random restart procedure, or *RRR* investigated in (Železný et al. 2006). The latter is a randomised procedure that searches in both general-to-specific and specific-to-general manner (the authors refer to this as a "radial" search), and was found to be substantially faster than *DTD*, with no significant losses in accuracy. The *RRR* procedure performs an exhaustive radial search up to a certain point (time constrained) and then, if no solution is found, restarts to a randomly selected location of the search space. The maximum number of

generalise$(B, E, \mathcal{L}, \rho, f)$: Given background knowledge B; a finite training set $E = E^+ \cup E^-$; a predefined hypotheses language $\mathcal{L}$; a refinement operator ρ; and an utility function f, returns a hypothesis H that explains the E.

1. $i = 0$
2. $E_i^+ = E^+$, $H_i = \emptyset$
3. if $E_i^+ = \emptyset$ return H_i otherwise continue
4. increment i
5. $Train_i = E_{i-1}^+ \cup E^-$
6. $D_i = search(B, H_{i-1}, Train_i, \mathcal{L}, \rho, f)$
7. $H_i = H_{i-1} \cup \{D_i\}$
8. $E_p = \{e_p : e_p \in E_{i-1}^+ \ s.t.\ B \cup H_i \models \{e_p\}\}$.
9. $E_i^+ = E_{i-1}^+ \backslash E_p$
10. Go to Step 3

Fig. 1 An ILP implementation using a greedy cover set procedure. The final set H is constructed by progressively finding the next best clause in Step 6 (this is the clause with the highest utility). The search for this clause is some generic search procedure that returns the best clause that meets the requirements $S1$ and $S2$. We do not describe this further, but refer the reader elsewhere (for example Železný et al. 2006) for such a procedure. It is straightforward to show that the greedy procedure shown here returns an H that satisfies requirement $C1$

restarts and the time constraint (called *maxtime* in Železný et al. 2006) are two parameters of the procedure. In some sense, *DTD* will act as a "strawman", and *RRR* as the more worthy adversary against which the gains, if any, of parallelisation will be measured.

3 Parallel execution of programs

The principal motivation in the parallel execution of a program is usually to improve its execution time. Several tools and platforms have been produced that attempt to simplify the development of parallel programs. Despite such tools, designing efficient parallel algorithms is still a non-trivial task. The main challenges associated with the design of parallel algorithms include minimizing I/O, synchronization and communication, effective load balancing, good data decomposition, and minimizing speculative work.

Before examining the parallel execution of ILP programs, we first describe briefly some common terms found in the literature on parallel algorithms (Grama et al. 2003). The process of dividing a computation into logically, high level, independent smaller parts is called decomposition. Tasks are the smaller parts that result from the decomposition. Tasks can be of arbitrary size, but once defined, they are regarded as indivisible units of computation. Different tasks can have different sizes. A task is said to be independent if its execution does not depend on data produced by others. It is said dependent, otherwise. The simultaneous execution of multiple tasks is the key to exploit parallelism efficiently. The maximum number of tasks that can be executed simultaneously, at any time, in a parallel algorithm determines its degree of parallelism. The granularity of a task is a measure computed as the ratio between the amount of computation done in a parallel task and the amount of communication. The scale of granularity ranges from fine-grained (very little computation per communication-byte) to coarse-grained (extensive computation per communication-byte). The finer the granularity, the greater the limitation on performance, due to the amount of synchronization needed.

The first step to enable parallel execution is to devise a parallel algorithm. Broadly speaking, we start with a sequential algorithm (that is, one that performs some sequential computation) and "parallelize" it by splitting concurrent portions of it as evenly as possible among the available processors, where each processor executes part of the computation and then combine, in some way, the partial results to obtain a final answer. The usual goal in parallelization is to minimize the execution time.

One would expect that increasing the number of processors would result in a proportional decrease of the execution time of a program. In practice, this is rarely observed due to overheads associated with parallelism. There are three major sources of overheads: interprocess communication, idling (for example, as a result of synchronization points in the computation or load imbalance), and extra computation (scheduling, packing/unpacking of messages, and so on).

The efficiency of a sequential algorithm is usually evaluated in terms of its execution time, which can be highly dependent on the size of the input. The efficiency of a parallel algorithm depends on more factors, namely the number of processors used, the amount and speed of inter-process communication, and also on the size of the input.

A number of performance metrics have been devised to be used in the study of parallel algorithms performance (Grama et al. 2003). 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. The speedup S achieved by a parallel algorithm measures the performance

improvement obtained by solving a problem in parallel. S 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. This is because many algorithms are essentially sequential in nature. However, 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).

A parallel algorithm is said to be scalable if the speedup increases with the number of processors or if it maintains the execution time fixed while increasing the number of processors and size of the problem proportionally.

The memory architecture of parallel computers play a major role on the type of parallel programs one can write and execute. In shared-memory computers, all the memory is viewed and globally addressable by all processors. In distributed memory computers, each processor can only access its own local memory and there is no provision for a global address space. Several inter-process communication (IPC) techniques exist for the exchange of data between processes. Processes may be running on one computer or on two or more computers connected by a network. The main methods for IPC are message passing, synchronization, shared memory, and remote procedure calls (RPC). The IPC method used may vary accordingly to the parallel computer architecture: it is selected based on the bandwidth and latency of communication between the processes, and the type of data being communicated.

3.1 Parallel execution for ILP

Based on the principal performance bottlenecks for ILP systems identified in Sect. 1, we classify the coarse and fine-grained options for parallel ILP execution into three main strategies:

Search. We can distinguish here between parallel execution of multiple searches, and the parallel execution within a search. The granularity of the latter is substantially finer than the former.

Data. In this, individual processors are provided with subsets of the examples prior to invoking the search procedure in Fig. 1. We distinguish between two forms of parallel execution, with different communication requirements. In the first, each processor completes its search and returns the best clause. The set of all clauses are then examined in conjunction and a final result constructed by re-computing the utility of each clause using all the data. In the second approach, as each processor finds a good clause, its utility in the final set is re-computed using all the data. The granularity of the second approach is finer than the first, but it has the advantage that all clauses found will also be in the final set of clauses (in both cases, recursive clauses cannot be identified reliably).

Evaluation. The search procedure invoked in Fig. 1 evaluates the utility of a clause. This usually requires its "coverage", which means determining the subset of E entailed by the D_i given B and H_{i-1}. A coarse-grained strategy involves partitioning E into blocks. The blocks are then provided to individual processors, which compute the examples covered in

Table 1 Parallel ILP systems reported in the literature

Parallelism	Architecture	
	Shared-memory	Distributed-memory
Search	Dehaspe and De Raedt (1995) Ohwada and Mizoguchi (1999) Ohwada et al. (2000) Wielemaker (2003)	No reports
Data	Skillicorn and Wang (2001) Graham et al. (2003)	Matsui et al. (1992) Clare and King (2003) Blaták and Popelínský (2006)
Evaluation	Ohwada and Mizoguchi (1999) Graham et al. (2003)	Matsui et al. (1992) Konstantopoulos (2003)

the block. The final coverage is obtained by the union of examples entailed in each block. There is a similarity to the coarse data-parallelism strategy described above. There too processors are provided with subsets of the data. There, the subsets are used to identify different clauses. Here, the subsets are used to evaluate a given set of clauses. A fine-grained strategy would involve determining subsets of literals in each D_i that can be evaluated independently (this could be identified, for example, using the "cut" transformation described in Santos Costa et al. 2003). Each such independent subset is then evaluated on a separate processor and the final result obtained by the intersection of examples entailed by the subsets.

It should be evident, but nevertheless worth noting here, that the three strategies above are not mutually exclusive. In fact, a parallel algorithm may exploit more than one. Furthermore, it should also be evident that the parallel algorithms can be classified in many different ways. For instance, we could also classify the parallel algorithms regarding their *correctness*, i.e., do they produce the same solution (correct) as the corresponding sequential algorithm. However, here we will focus our classification of previous work on parallel ILP systems on the three strategies mentioned above, along with the hardware architecture. Table 1 shows a brief summary of the entries that follow.

Dehaspe and De Raedt (1995). This is the first parallel ILP system that we are aware of. The system is a parallel implementation of Claudien, an ILP system capable of discovering general clausal constraints. The strategy is based on the 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 sets of clauses found in each processor are combined. The system was evaluated on a shared-memory computer with two datasets and exhibited a linear speedup up to 16 processors.

Ohwada and Mizoguchi (1999). This implements an algorithm based on inverse entailment (Muggleton 1995). The implementation uses a parallel logic programming language and explored the parallel evaluation of clause coverage, and two strategies for search parallelisation (parallel exploration of independent hypotheses and parallel exploration of each refinement branch of a search space arising from each hypothesis). The 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 sublinear speedup in all strategies, although parallel coverage testing appeared to yield the best results.

Ohwada et al. (2000). This implements an algorithm that explores the search space in parallel. The set of nodes to be explored is dynamic and implemented using contract-net communication (Smith 1980). Their paper investigated two types of inter-process communication, with results showing near-linear speedups on a 10 processor machine.

Wielemaker (2003). This implements a parallel version of a randomised search found in the Aleph system. The parallel implementation executes concurrently several randomised local searches using a multi-threaded version of the SWI Prolog engine. Experiments examined performance as the number of processors was progressively increased. Near-linear speedups were observed up to 4 processors, however the speedup was not sustained as the number of processors increased to 16.

Skillicorn and Wang (2001). This implements a parallel version of the Progol algorithm (Muggleton and Firth 2001) by partitioning the data and applying a sequential search algorithm to each partition. Data are partitioned by dividing the positive examples among all processors and by replicating the negative examples at each processor. Each processor then performs a search using its local data to find the (locally) best clause. The true utility of each clause found is then re-computed by sharing it among all processors. Note that this algorithm exploits the parallelization strategies identified (parallel search, data and evaluation) mentioned above, thus being an example that the strategies are not mutually exclusive. Experiments with three datasets suggest linear speedups on machines with 4 and 6 processors.

Matsui et al. (1992). This evaluates and compares two algorithms based on data parallelism and parallel evaluation of refinements of a clause (the paper calls this parallel exploration of the search space, although it really is a parallelisation of the clause evaluation process). The two strategies are used to examine the performance of a parallel implementation of the FOIL (Quinlan and Cameron-Jones 1993) system. Experiments are restricted to a small synthetic data set (the "trains" problem Michalski 1980) and the results show poor speedups from parallelisation of clause evaluation. Data parallelism showed initial promise, with near linear speedups up to 4 processors. Above 4 processors, speedup was found to be sub-linear due to increased communication costs.

Clare and King (2003). This describes PolyFarm: a parallel ILP system specifically designed for the discovery of first-order association rules on distributed memory machines. Data are partitioned amongst multiple processors and the system follows a master-worker strategy. The master generates the rules and reports the results and workers perform the coverage tests of the set of rules received from the master on the local data. Counts are aggregated by a special type of worker that reports the final counts to the master. No performance evaluation of the system is available.

Graham et al. (2003). This implements a parallel ILP system, using the PVM message passing library. Parallelisation is achieved by partitioning the data and by parallel coverage testing of sets of clauses (corresponding to different parts of the search space) on each processor. Near-linear speedups are reported up to 16 processors on a shared memory machine.

Konstantopoulos (2003). This investigates a data parallel version of a deterministic top-down search implemented within the Aleph ILP system (2003). The parallel implementation uses the MPI library and performs coverage tests in parallel on multiple machines. This strategy is quite similar to that reported in Graham et al., with the caveat that testing is restricted to one clause at a time (Graham et al. look at sets of clauses). Results are not promising, probably due to the over-fine granularity of testing one clause at a time.

Blaták and Popelínský (2006). This describes dRap: a parallel ILP system specifically designed for the discovery of first-order association rules on distributed memory machines.

Table 2 Summary of speedups reported of parallel ILP systems. The numbers in parentheses refer to the number of processors. Neither Clare & King nor Blaták & Popelínský report any speedups

Parallelism	Speedup	
	Shared-memory	Distributed-memory
Search	Dehaspe & De Raedt: Linear (16)	No reports
	Ohwada & Mizoguchi: 2–3 (6)	
	Ohwada et al.: 8 (10)	
	Wielemaker: 7 (16)[†]	
Data	Wang & Skillicorn: Linear or better (6)	Matsui et al.: 4 (15)[‡]
		Clare & King: –
	Graham et. al: linear (16)	Blaták & Popelínský: –
Evaluation	Ohwada & Mizoguchi: 4 (6)	Matsui et al.: 1 (15)
	Graham et al.: 5 (8)	Konstantopoulos: none

[†] linear up to 4 processors

[‡] linear up to 5 processors

Data are partitioned amongst multiple processors and the system follows a master-worker strategy. The master generates the partitions and each worker then executes a sequential first-order association rules learner. The master collects the rules found by the workers and then redistributes the rules by all the workers to compute the support on the whole dataset. No performance evaluation of the system is reported.

Results reported by these papers are summarised in Table 2. The principal points that emerge are these:

1. Most of the effort has been focused on shared-memory machines where the communication costs are lower than for distributed-memory machines.
2. Speedups observed on shared-memory machines are higher than those observed on distributed memory ones. Maximum disparity is observed with parallel execution of the coverage tests: this is undoubtedly due to the fact that communication costs are high for distributed-memory machines, and the granularity of the task is finer than other forms of parallelism.

Despite the apparently discouraging results observed to date on distributed-memory machines, we believe that a further investigation is warranted for several reasons. First, the results are not obtained from a systematic effort to investigate the effect of the different kinds of parallelism. That is: results that are available are obtained from a mix of fine and coarse-grained parallelisation, on differently configured networks and with different communication protocols. Second, the availability and parallelism of shared-memory architecture machines continues to be substantially lower than distributed-memory ones (for example, distributed-memory "clusters" comprised of 10s or 100s of machines are relatively easy, and cheap, to construct). There is, therefore, practical interest in examining if significant speedups are achievable on distributed-memory architectures. In this paper, we present a systematic empirical evaluation of coarse-grained search, data and evaluation parallelisation for such architectures using a well-established network of machines (a Beowulf cluster) and a widely accepted protocol for communication (an implementation of the Message Passing Interface, or MPI (Message Passing Interface Forum 1994), that can be used by applications running in heterogeneous distributed-memory architectures).

4 Empirical evaluation

In this section we intend to undertake an empirical investigation into the effect of coarse-grained parallelisation implemented on a distributed-memory architecture. The investigation will examine the implementation of parallel algorithms for each of the three categories identified, namely: search, data and evaluation. In each case, we examine the speedup obtained with an efficient parallel implementation. Specifically, our goals are to estimate the speedups obtained with each form of parallelisation; and to determine if these are an improvement over those obtainable with a sequential randomised technique (to date, these randomised techniques have reported some of the most significant speedups on benchmark data).

4.1 Materials

4.1.1 Data and background knowledge

Data for the experiments are from three well-known biochemical problems in the ILP literature: (1) identifying mutagenic nitroaromatics (Srinivasan et al. 1994a, 1994b) (specifically, the "regression-friendly" subset consisting of 188 compounds); (2) identifying chemical carcinogens (Srinivasan et al. 1997; King and Srinivasan 1996); and (3) the inhibition of E. coli dihydrofolate reductase by pyrimidines (King et al. 1992). Furthermore, we used a well known engineering problem in the ILP literature: the problem of learning rules for determining the resolution of a Finite Element mesh (Dolšak et al. 1997), that corresponds to learning rules that determine the number of elements on an edge. We refer the reader to the references cited for details of the datasets and the background knowledge available for each problem.

Besides the obvious differences, the problems have a number of distinguishing features from an ILP viewpoint:

1. The mutagenesis problem is known to have good short clauses (with four literals or fewer) that entail the observed data; and the carcinogenesis problem is known not to have any good short clauses that entail the data (this is purely based on chemical structure: see Dehaspe et al. 1998). To the best of our knowledge, the pyrimidines problem has only been addressed using a specific-to-general ILP system (Golem: Muggleton and Feng 1990). This system has a propensity to construct long clauses. It is not known therefore if there are any short clauses that are equally effective in explaining the observed data.
2. The number of examples constituting the observational data vary from a few hundreds (188 for mutagenesis and 337 for carcinogenesis) to a few thousands (2788 for pyrimidines and 3119 for mesh).
3. The background knowledge varies from determinate (pyrimidines and mesh); to moderately non-determinate (mutagenesis); to highly non-determinate (carcinogenesis). The amount of background knowledge also varies: it comprises about 2000 ground facts for the pyrimidines and mesh problems; about 10,000 ground facts for mutagenesis; and about 20,000 ground facts for carcinogenesis.

4.1.2 Algorithms and machines

All algorithms follow a standard master-worker scheme in which one processor is designated as the "master" and the remaining are "workers". Workers wait for requests from the master: after receiving a message from the master, a worker executes the task contained in

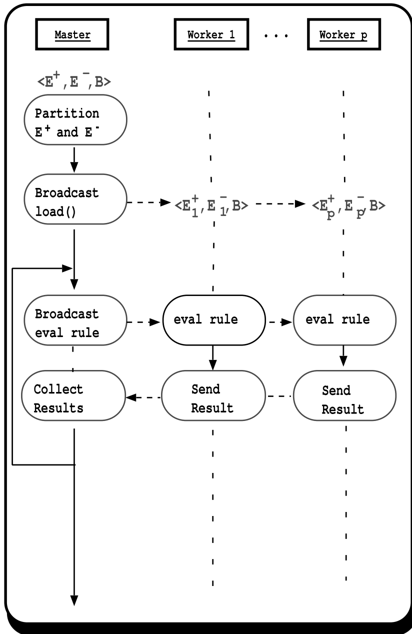
it. Task results are sent to the sender (master) after task completion (a simplified diagrammatic view of the messages exchanged between a master and worker for each of the parallel implementations can be found in Fig. 2).

We concentrate on coarsely-parallel strategies for each of search, data and evaluation parallelisation. Specifically, we have implemented the following within the April ILP system (Fonseca et al. 2006):

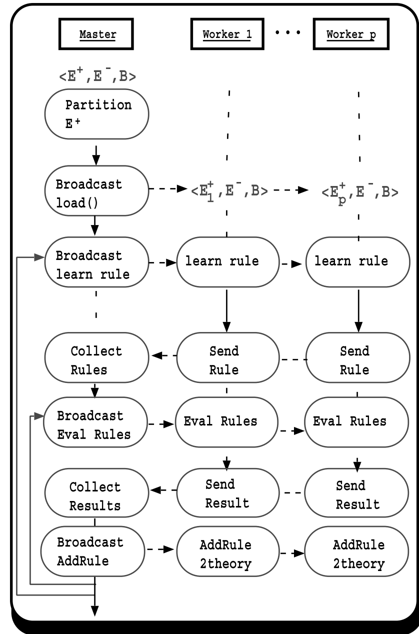
Search. Parallel version of the *RRR* algorithm described in (Železný et al. 2006). We will call it *SP–RRR* (the prefix SP stands for "search parallel"). In the parallel version, the master's algorithm is similar to the sequential version with the following differences: (i) the master replicates the data to all workers in the beginning of the execution; (ii) the searches are performed in parallel by the workers, after receiving the initial (random) rule from the master, and the good rules found are sent to the master; the number of searches performed is limited by *maxtries*; if the number of workers is lower than *maxtries*, the master starts one search on each worker and then spawns new searches on workers that complete their previous assignment. The master stops spawning searches to workers whenever it reaches the *maxtries* count or upon receiving a rule from a worker, case in which it waits for all workers to complete before proceeding to the next step; (iii) the master selects the best rules of all received rules; (iv) the removal of examples covered by the best rule (steps 8 and 9 in Fig. 1) is performed in parallel on all workers. Note that *SP–RRR may* not return the same solution as the sequential *RRR* due, mainly, to the different order by which the rules are retained in the final set. The granularity of the parallel tasks of this algorithm varies accordingly to the value of the *maxtime* parameter: the bigger the value, the larger the granularity of the parallel task.

Data. Two data procedures based on data parallelism were implemented. We call these two procedures *DP–LR* and *DP–LT* (DP stands for "data parallel"; LR for "learn rule" and LT for "learn theory"). *DP–LR* was implemented as described by Wang & Skillicorn (in Skillicorn and Wang 2001) and summarised in Sect. 3.1. We now describe *DP–LT* that is a more general procedure than *DP–LR*. *DP–LT* starts by partitioning the set of examples (both positive and negatives: recall the Wang & Skillicorn variant only partitions the positive examples, the negative examples are replicated). The *DP–LT* procedure induces p sets of rules in parallel, using the standard sequential algorithm on each subset. We still need to obtain a single set of rules from the p rule-sets. The combination of the sets of rules into a single rule set can be made using several strategies. In order to make the comparison with the sequential algorithm clearer, a simple strategy was selected, very similar to the one used by the sequential algorithm. The rules are ordered using some metric (in our implementation we used coverage). The best rule is selected for retention in the final set and the remaining rules are reevaluated and reordered (the rules that are no longer considered acceptable are discarded). Then the best rule is selected for retention and process is repeated for the remaining rules. The solution returned by these procedures may not be the same as the sequential version. It is straightforward to see that *DP–LT* procedure has the largest granularity of all procedures here described.

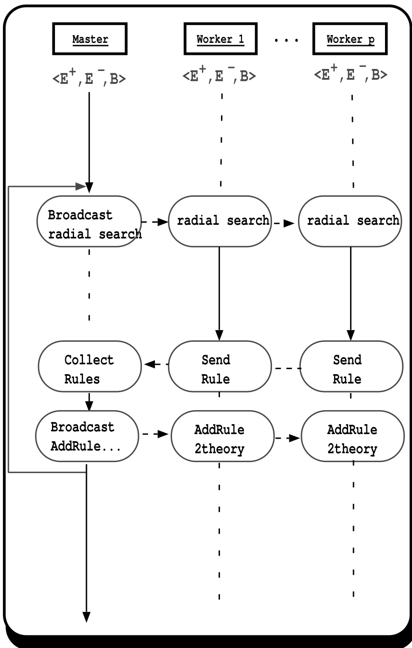
Evaluation. A parallel version of the procedure implemented in Konstantopoulos (Konstantopoulos 2003 and Sect. 3.1) with two important differences: (1) we use asynchronous message passing communication for all operations involving the sending of a message (Konstantopoulos only uses synchronous message passing operations); and (2) our implementation used LAM MPI as opposed to the MPICH MPI implementation used by Konstantopoulos. In the procedure, examples are divided evenly amongst the p processors. Coverage of each rule found in the search is evaluated in parallel on the p subsets and the final coverage computed by the union of examples covered in each subset. We call the



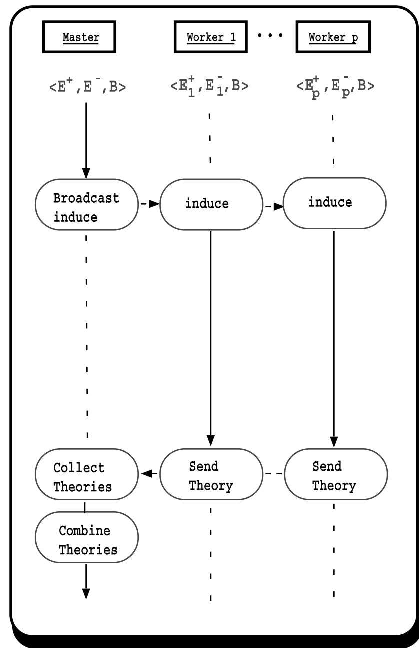
a) *EP-CT*



b) *DP-LR*



c) *SP-RRR*



d) *DP-LT*

Fig. 2 Simplified schemes of the messages exchanged by the parallel algorithms. *Solid lines* represent the execution flow, *horizontal dashed lines* represent message passing between the processes, and *vertical dashed lines* represent idleness. The algorithms are ordered by the granularity of their parallel tasks, from the finest-grained to the most coarse-grained

Table 3 ILP settings used in the experiments

Application	*i*-depth	Nodes	Noise	MinAcc	CL
Mesh	4	20,000	1%	85%	8
Carc	4	50,000	5%	70%	10
Mut	3	700	5%	70%	4
Pyr	3	20,000	1%	85%	10

procedure *EP–CT* (EP stands for "evaluation parallel" and CT for "coverage test"). Note that *EP–CT* returns the same solution as the sequential algorithm whilst the previous procedures *may* not return the same solution as the sequential algorithm due, mainly, to the different order by which the rules are found and retained in the final set.

In all cases, we will be comparing the parallel implementations against the randomised bi-directional search *RRR*; and, for completeness, the systematic top-down search *DTD*.

All parallel implementations used the Prolog language (YAP compiler version 5.1); and LAM (Squyres and Lumsdaine 2003) MPI for the communication layer. 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.

The experiments were performed on a Beowulf Cluster composed of 4 nodes. Each node is a dual processor computer with 2 GB of memory, running the Fedora Core 6 distribution of GNU/Linux.

4.1.3 Settings

The settings of the ILP system were tuned so that the *DTD* runs did not take more than two hours to complete (except for the **Mut** application). Table 3 shows the main settings used for each application. The parameter *nodes* specifies an upper bound on the number of rules (nodes-restriction) generated while searching for a rule. The *i*-depth (Muggleton and Feng 1992) corresponds to the maximum depth of a literal with respect to the head literal of the rule. The parameter *CL* defines the maximum length that a rule may have, i.e., the maximum number of literals in a clause. *MinAcc* specifies the minimum accuracy that a rule must have in order to be accepted as good. Finally, the *noise* parameter defines the maximum percentage of negative examples that a rule may cover in order to be accepted.

For the *RRR* algorithm we ignored the parameter *nodes* and used the following settings: $restarts = 100$, $maxtime = 20$ s. The choices for these parameters, although admittedly arbitrary, were the same used previously in the literature (Srinivasan 2000; Železný et al. 2002).

4.2 Method

Our method is straightforward:

- For each problem area (mesh, mutagenesis, carcinogenesis, and pyrimidines):
  - With 1 processor:
    1. Construct the best possible theories using the systematic top-down search *DTD* and the randomised bi-directional search *RRR*.
    2. Estimate the accuracies of the theories constructed and the times taken to construct the theories. For simplicity, we will call these "sequential accuracy" and "sequential time".

– With 2, 4, 6 and 8 processors:

  1. Construct the best possible theories using the search-parallel procedures (*SP–RRR*); data-parallel procedures (*DP–LR* and *DP–LT*); and the evaluation-parallel procedure (*EP–CT*) described earlier.
  2. Estimate the accuracies of the theories constructed and the times taken to construct the theories. For simplicity, we will call these "parallel accuracy" and "parallel time".

– Compare the estimates of sequential accuracy and time against the corresponding estimates of parallel accuracy and time.

The following details are relevant:

(a) Estimates of accuracy and times for theory construction were obtained using 5-fold cross validation;
(b) For each problem and procedure, constructing the "best possible theory" requires optimal values for some parameters. The principal parameters of interest for an ILP algorithm are usually: the maximum number of literals in any clause found by the ILP system and the minimum accuracy for a clause to be acceptable. We have used values from the original reports in the literature (references provided above in *Data and Background Knowledge*' section), under the assumption that these are likely to be near-optimal;
(c) Constructing theories with a search procedure results in cross-validated estimates of accuracy and time for theory construction. The standard error in the accuracy estimate is obtained using the contingency table from the cross-validation. Strictly speaking, this is incorrect, but it has been found to be an acceptable approximation (Breiman et al. 1984).
(d) A quantitative analysis for significant differences in accuracies should account for the fact that all theories are tested on the same sample. The appropriate statistical test for this is McNemar's test for changes (Everitt 1992), that tests the null hypothesis that the proportion of examples correctly classified by a pair of theories is the same. Differences in accuracy will be deemed significant only if on completing the calculations for McNemar's test, the resulting probability of the observed differences occurring by chance is ≤ 0.05. Comparisons of times for theory construction will simply be the speedups (if any) observed over the time taken by the *DTD* procedure.

4.3 Results

Tables 4 and 5 summarise the accuracies and speedups obtained with the sequential and parallel procedures (the complete set of results is in Appendix A). We note first that the sequential randomised procedure *RRR* attains comparable accuracies to the systematic search procedure *DTD* in substantially less time. This confirms observations made elsewhere in the literature (Železný et al. 2006).

The parallel version of *RRR*, *SP–RRR*, shows good speedups with the **Carc** application, while in the **Mut** we observe a slowdown. A distinctive characteristic of *SP–RRR*, that can explain the slowdown observed in **Mut**, is that it makes more searches (restarts) than *RRR*. Although both *RRR* and *SP–RRR* stop *restarting* as soon as a clause is found, in the case of *SP–RRR* it must wait for the restarts being run in parallel to terminate, thus potentially causing some overhead in the execution. Aborting the execution of these restarts is not an

Table 4 Classification accuracies of theories. The first two rows represent sequential procedures, and the remainder parallel ones. The parallel procedures are in groups corresponding to those that perform search parallelisation (prefixed by "SP"), data parallelisation ("DP") and evaluation parallelisation ("EP"). The accuracy shown for any parallel procedure is the highest of the values obtained with 2, 4, 6 and 8 processors. There are no significant statistical differences in the accuracies of sequential and parallel procedures.

Procedure	Application			
	Mesh	Mutagenesis	Carcinogenesis	Pyrimidines
DTD	90.2	76.6	55.2	88.5
RRR	89.7	78.7	55.8	83.7
SP–RRR	89.5	78.8	54.6	83.6
DP–LR	43.8	79.3	56.1	80.6
DP–LT	91.1	84.0	58.7	87.7
EP–CT	90.2	76.6	55.2	88.5

Table 5 Speedups for constructing theories, measured against the time for theory construction by *DTD*, (shown normalised to 1.00). That is, for any procedure, the speedup is the ratio of the time taken by *DTD*to the corresponding time taken by that procedure: an entry greater than 1 indicates that the procedure is faster than *DTD*. The speedup shown for any parallel procedure is the highest of the values obtained with 2, 4, 6 and 8 processors

Procedure	Application			
	Mesh	Mutagenesis	Carcinogenesis	Pyrimidines
DTD	1.00	1.00	1.00	1.00
RRR	4.06	2.08	23.45	7.13
SP–RRR	6.18	0.70	139.39	7.30
DP–LR	5.34	7.82	2.72	4.77
DP–LT	3.10	5.69	9.89	4.47
EP–CT	0.46	3.02	1.09	0.55

easy and costless option on a distributed execution environment[2]. As stated earlier, the granularity of the *SP–RRR* algorithm depends on the value of the *maxtime* parameter. Therefore, increasing the value of this parameter results in longer searches (restarts), thus increasing the granularity of the parallel computations.

The results reported on *DP–LR* are quite different from the ones previously reported in (Skillicorn and Wang 2001). Wang et al. reported super-linear speedups (up to 6 processors) while here we report sub-linear speedups in most of the cases. This can be explained by the fact that Wang et al. ran the experiments in a shared memory machine whereas we ran them on a distributed memory machine (a Beowulf cluster) with unavoidable communication overheads. We can also observe that *DP–LR* achieves a considerable worst accuracy on **Mesh** than that obtained by the other algorithms. The theories found by *DP–LR* on **Mesh** are composed by more specific and lengthier rules than the ones found by *DTD* suggesting that the algorithm is overfitting while learning in the subset of positive examples.

[2]Both procedures, *SP–RRR* and *RRR*, stop the generation of clauses in a restart after the time limit *maxtime* is reached. However, the time limit does not stop the process of checking how many examples an already generated clause covers. Therefore, a restart can take more than *maxtime*. The extra time will depend greatly on the cost to evaluate the examples.

Table 6 Spearman's rank correlation between accuracy and the number of processors. The entries are calculated using accuracies obtained with 2, 4, 6 and 8 processors. We expect all entries to have a value of 0.00 if accuracies are uncorrelated with the number of processors

Procedure	Application			
	Mesh	Mutagenesis	Carcinogenesis	Pyrimidines
SP–RRR	−0.77	−0.25	0.00	−0.60
DP–LR	−0.80	+0.80	+0.60	+0.60
DP–LT	−0.63	−0.31	+0.94	−0.54
EP–CT	0.00	0.00	0.00	0.00

Table 7 Spearman's rank correlation between speedup and the number of processors. The entries are calculated using speedups obtained with 2, 4, 6 and 8 processors. We expect all entries to have a value of 0.00 if speedups are uncorrelated with the number of processors

Procedure	Application			
	Mesh	Mutagenesis	Carcinogenesis	Pyrimidines
SP–RRR	+1.00	+1.00	+1.00	−0.40
DP–LR	+0.80	+0.80	+0.40	+1.00
DP–LT	+1.00	−0.40	+1.00	+1.00
EP–CT	−0.55	+0.80	−1.00	−0.40

We assess the performance of the parallel procedures by considering three questions:

1. Is there any relation between accuracy, speedup and parallelisation?
2. What are the comparative efficacies of search, data or evaluation parallelism for distributed memory architectures?
3. Does parallelisation give substantial gains over the best sequential approach?

Tables 6 and 7 show estimates of the pairwise association between accuracy and speedup with parallelisation. The estimates are obtained using rank correlations of the values obtained with each of the former quantities with 2, 4, 6 and 8 processors. We are able to test the hypotheses $H_{0,acc}$ that accuracy is uncorrelated with the number of processors and $H_{0,speedup}$ that speedups are uncorrelated with the number of processors. This translates to expected rank correlations of 0.0 in both cases (accuracy versus processors, and speedup versus processors). Assuming the entries in Tables 6 and 7 are observations of independent, identically distributed random variables, we use a simple sign test to obtain approximate probabilities of observing the values shown, if the null hypotheses held. Ignoring ties and denoting values below the expected value as "tails" and those above as "heads", we observe 7 tails and 4 heads in Table 6. The corresponding values are 5 and 11 Table 7. The corresponding binomial probabilities are 0.3 and 0.1, which suggest that we are clearly unable to reject $H_{0,acc}$, but can reject $H_{0,speedup}$ with some caution. We are therefore, able to formulate the first of our findings:

F1. The data are consistent with accuracies being unaffected by parallelisation; and provide some evidence of speedups increasing with parallelisation.

We turn now to the second question, namely, the comparative efficacies of search, data and evaluation parallelisation. Based on the discussion just preceding, we assume accuracies are unaffected by parallelisation and that the question is concerned with speedups only (specifically, on the maximum speedups possible). Both the best-case speedups in Table 5

Table 8 Data and background knowledge characteristics of the applications considered. Here $|E^+|$ and $|E^-|$ refer to the numbers of positive and negative examples; $|B|$ refers to the number of background relations; *AET* is the average estimated time to evaluate if a single example is entailed by a clause along with the background knowledge

Application	$\|E^+\|$	$\|E^-\|$	$\|B\|$	*AET* (µs)
Mesh	2841	290	29	12
Mutagenesis	125	63	21	20846
Carcinogenesis	182	155	38	305
Pyrimidines	1394	1394	45	35

Table 9 The effect of changing the average clause evaluation time on speedups obtained with search and evaluation parallelisation

Procedure	Speedup on mutagenesis	
	AET = 20846 µs	*AET* = 163 µs
SP–RRR	0.70	3.68
EP–CT	3.02	1.43

and the more detailed break-up in Table 11 (see Appendix A) show that while the consequences of data parallelisation are relatively uniform across the four problems—there is always a speedup—the mutagenesis data appear to affect the other two forms of parallelisation in different ways. It is only here that search parallelisation is slower than deterministic search, and, for all practical purposes, evaluation parallelisation is faster. To investigate this further, we examine some quantitative differences between the four problems: these are tabulated in Table 8.

We note the average time to evaluate an example is markedly different for the mutagenesis data (the difference is 2 to 3 orders of magnitude, while being within 1 order of magnitude along other dimensions). There are many reasons why times for evaluation could be high: the background knowledge may be very non-determinate or time-consuming to execute; there may be many literals in the clause; the order of the literals may be inappropriate; the evaluation performs a subsumption test that is inherently difficult, and so on. Here, a large proportion of the long time for evaluation is due to the use of intensional definitions—in effect, a logic program—in the background knowledge for detecting chemical groups (ring structures, methyl groups, alcohols, and so on). Definitions encoded in this form are slow to execute, and for small datasets can be pre-computed into a table of ground facts (this has been done, for instance, for the carcinogenesis data). An alternative to pre-computing is to dynamically ground the intensional definitions by means of tabling (Rocha et al. 2005). Using an extensional version of the background knowledge alters the *AET* for mutagenesis from 20846 µs to 163 µs. The effect on search and evaluation parallelisation is shown in Table 9 and in more detail in Table 12 (see Appendix A)

These results form the basis for the following findings:

F2. On average, we expect evaluation parallelisation to be the least effective of the three forms of parallelisation. We expect search parallelisation to result in higher speedups than data parallelisation unless clause evaluation times are very high.
F3. Data parallelisation appears to be generally effective in that it always results in some speedup.
F4. Evaluation parallelisation is only effective when clause evaluation times are very high.

Finally, we consider the third question, which is whether parallelisation is any better than the best sequential approach (here, *RRR*). From the best-case speedups in Table 5 it is clear that the sequential approach is slower than at least one parallel procedure on all three problems. As to the more specific question of whether any *particular* form of parallelisation is consistently better than the sequential approach, the answer appears to be "yes", only if average evaluation time is not too high. If this situation holds, then results with search parallelisation (that is, *SP–RRR*) are consistently better than the best sequential results. While there is still a 1 in 4 chance that *SP–RRR* is actually slower than *RRR*, there is nevertheless some evidence here for the following finding:

F5. If clause evaluation times are not very high, then we expect search parallelisation to be more effective than the best sequential approach.

5 Concluding remarks

We surveyed the state-of-the-art on parallel ILP implementations and studied the performance of five parallel algorithms on a distributed memory computer using four ILP benchmark applications. The parallel algorithms implemented exploit the three main strategies to parallelize ILP systems identified.

The principal findings of interest are: (1) Parallelisation on distributed memory architectures does seem to be useful (measured by speedup over a deterministic search strategy on a sequential architecture); (2) Of search, data and evaluation parallelisation, the last seems to be the least effective. Of search and data parallelisation, the former is more effective if clause evaluation times are not high. In general, the latter always yields some speedup over the deterministic search strategy, but not necessarily over the fastest sequential search method reported in the literature; and (3) If clause evaluation times are not high, then search parallelisation yields speedups over both a deterministic search, and the fastest sequential search method reported in the literature.

A natural extension of this work is to perform a larger experimental evaluation on larger clusters and number of applications. It would also be interesting to extend the evaluation of the algorithms to shared memory architectures. This paper has been restricted to studying each form of parallelisation in isolation. Combinations of the different methods need also to be investigated. Finally, communication is always an important factor on the performance of parallel algorithms. Although absolute numbers were not our goal here, improvements in the communication layer implementation should yield higher speedups.

Acknowledgements The authors would like to acknowledge the anonymous reviewers for the insightful and detailed comments that greatly improved the paper. The work presented in this paper has been supported by funds granted to *LIACC* through the *Programa Operacional "Ciência, Tecnologia, Inovacão" (POCTI) and Programa Operacional Sociedade da Informacão (POSI) do Quadro Comunitário de Apoio III (2000–2006)*. This work has been also partially supported by project ILP-Web-Service (PTDC/EIA/70841/2006). Nuno Fonseca is funded by FCT grant SFRH/BPD/26737/2006.

Appendix A: Results tables

Table 10 Estimates of the classification accuracies of theories. The first two rows represent sequential procedures, and the remainder parallel ones. The parallel procedures are in groups corresponding to those that perform search parallelisation (prefixed by "SP"), data parallelisation ("DP") and evaluation parallelisation ("EP"). The accuracies shown for the parallel procedures were obtained using 2, 4, 6 and 8 processors

Procedure	Application			
	Mesh	Mutagenesis	Carcinogenesis	Pyrimidines
DTD	90.3	76.6	55.2	88.5
RRR	89.8	78.7	55.8	83.7
SP–RRR	89.8 89.7 89.7 89.7	78.8 78.8 78.7 78.8	54.6 54.6 54.6 54.6	83.4 83.6 83.2 83.3
DP–LR	43.8 41.5 40.5 40.6	77.7 78.2 79.3 79.2	52.8 54.6 52.8 56.1	79.8 77.7 80.6 80.0
DP–LT	91.0 90.6 90.8 90.6	84.0 75.0 78.3 78.3	55.8 58.4 58.4 58.7	86.9 87.7 86.7 87.3
EP–CT	90.3 90.3 90.3 90.3	76.6 76.6 76.6 76.6	55.2 55.2 55.2 55.2	88.5 88.5 88.5 88.5

Table 11 Estimates of the time for constructing theories. The times are shown as speedups measured against the time for theory construction by *DTD* (for any procedure, the speedup is the ratio of the time taken by *DTD* to the corresponding time taken by that procedure. Thus, an entry greater than 1 indicates that the procedure is faster than *DTD*). The speedups shown for the parallel procedures were obtained using 2, 4, 6 and 8 processors. The *DTD* times are presented in seconds

Procedure	Application			
	Mesh	Mutagenesis	Carcinogenesis	Pyrimidines
DTD	5,987 (s)	31,979 (s)	2,509 (s)	7,393 (s)
RRR	4.06	2.08	23.45	7.13
SP–RRR	3.12 5.67 5.92 **6.18**	0.19 0.37 0.55 **0.70**	54.54 104.54 125.45 **139.39**	6.90 **7.30** 7.01 6.53
DP–LR	1.86 4.47 **5.34** 5.15	4.63 5.43 5.39 **7.82**	2.64 1.78 2.28 **2.72**	0.84 1.70 2.82 **4.77**
DP–LT	0.35 1.81 1.93 **3.10**	**5.69** 2.37 2.20 5.52	1.15 2.77 3.45 **9.89**	1.29 2.12 3.87 **4.47**
EP–CT	0.34 **0.46** 0.34 0.32	1.58 2.96 2.90 **3.02**	**1.09** 0.75 0.70 0.65	0.53 0.42 **0.55** 0.41

Table 12 Estimates of the time for constructing theories for the extensional version of the mutagenesis dataset. The times are shown as speedups measured against the time for theory construction by *DTD*. The speedups shown for the parallel procedures were obtained using 2, 4, 6 and 8 processors. The *DTD* time is presented in seconds. We point out that with the same settings as the other version of the dataset the *DTD* execution time was only 3 seconds. We therefore increased the search space (50 fold) in order to be able to evaluate the performance of the two parallel algorithms

Procedure	Mutagenesis
DTD	2,601 (s)
RRR	4.61
SP–RRR	3.63 3.67 3.68 3.65
EP–CT	1.43 1.34 1.26 1.04

References

Blaták, J., & Popelínský, L. (2006). dRAP: a framework for distributed mining first-order frequent patterns. In *Proceedings of the 16th conference on inductive logic programming* (pp. 25–27). Berlin: Springer.

Boström, H. (2000). Induction of recursive transfer rules. In J. Cussens & S. Džeroski (Eds.), *Lecture notes in computer science: Vol. 1925. Learning language in logic* (pp. 237–246). Berlin: Springer.

Botta, M., Giordana, A., Saitta, L., & Sebag, M. (2003). Relational learning as search in a critical region. *Journal of Machine Learning Research*, *4*, 431–463.

Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and regression trees*. Belmont: Wadsworth.

Clare, A., & King, R. D. (2003). Data mining the yeast genome in a lazy functional language. In *Proceedings of the fifth international symposium on practical aspects of declarative languages* (pp. 19–36).

Colton, S., & Muggleton, S. (2003). ILP for mathematical discovery. In *Proceedings of the 13th international conference on inductive logic programming* (pp. 93–111).

Cussens, J. (1997). Part-of-speech tagging using Progol. In *Proceedings of the 7th international workshop on inductive logic programming* (pp. 93–108).

Dehaspe, L., & De Raedt, L. (1995). Parallel inductive logic programming. In *Proceedings of the MLnet familiarization workshop on statistics, machine learning and knowledge discovery in databases*.

Dehaspe, L., Toivonen, H., & King, R. D. (1998). Finding frequent substructures in chemical compounds. In *Proceedings of the fourth international conference on knowledge discovery and data mining (KDD-98)* (pp. 30–36). Menlo Park: AAAI Press.

Dolšak, B., Bratko, I., & Jezernik, A. (1997). Application of machine learning in finite element computation. In *Machine learning, data mining and knowledge discovery: methods and applications*. New York: Wiley.

Džeroski, S., Demšar, D., & Grbović, J. (2000). Predicting chemical parameters of river water quality from bioindicator data. *Applied Intelligence*, *13*(1), 7–17.

Everitt, B. S. (1992). *The analysis of contingency tables* (2nd ed.). London: Chapman and Hall.

Fonseca, N. A., Silva, F., & Camacho, R. (2006). April—an inductive logic programming system. In *Lecture notes in artificial intelligence: Vol. 4160. Proceedings of the 10th European conference on logics in artificial intelligence (JELIA06)* (pp. 481–484), Liverpool, 2006. Berlin: Springer.

Graham, J., Page, D., & Kamal, A. (2003). Accelerating the drug design process through parallel inductive logic programming data mining. In *Proceeding of the computational systems bioinformatics (CSB'03)*. New York: IEEE.

Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *Introduction to parallel computing* (2nd ed.). Reading: Addison-Wesley.

King, R. D. (2004). Applying inductive logic programming to predicting gene function. *AI Magazine*, *25*(1), 57–68.

King, R. D., & Srinivasan, A. (1996). Prediction of rodent carcinogenicity bioassays from molecular structure using inductive logic programming. *Environmental Health Perspectives*, *104*(5), 1031–1040.

King, R. D., Muggleton, S., & Sternberg, M. J. E. (1992). 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 *Proceedings of the national academy of sciences* (Vol. 89, pp. 11322–11326).

Konstantopoulos, S. K. (2003). 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.

Marchand-Geneste, N., Watson, K. A., Alsberg, B., & King, R. D. (2002). A new approach to pharmacophore mapping and QSAR analysis using inductive logic programming. Application to thermolysin inhibitors and glycogen phosphorylase B inhibitors. *Journal of Medicinal Chemistry*, *45*, 399–409 (Erratum: *Journal of Medicinal Chemistry*, *46*, 653).

Matsui, T., Inuzuka, N., Seki, H., & Itoh, H. (1992). Comparison of three parallel implementations of an induction algorithm. In *8th international parallel computing workshop* (pp. 181–188), Singapore.

Michalski, R. S. (1980). Pattern recognition as rule-guided inductive inference. In *Proceedings of IEEE transactions on pattern analysis and machine intelligence* (pp. 349–361).

Message Passing Interface Forum. (1994). *MPI: a message-passing interface standard* (Technical Report UT-CS-94-230). University of Tennessee, Knoxville, TN, USA.

Muggleton, S. (1994). Inductive logic programming: derivations, successes and shortcomings. *SIGART Bulletin*, *5*(1), 5–11.

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing, Special Issue on Inductive Logic Programming*, *13*(3–4), 245–286.

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the 1st conference on algorithmic learning theory* (pp. 368–381), Ohmsma, Tokyo, Japan.

Muggleton, S., & Feng, C. (1992). Efficient induction in logic programs. In S. Muggleton (Ed.), *Proceedings of the 2nd international workshop on inductive logic programming* (pp. 281–298). New York: Academic Press.

Muggleton, S., & Firth, J. (2001). Relational rule induction with CProgol4.4: a tutorial introduction. In S. Džeroski & N. Lavrač (Eds.), *Relational data mining* (pp. 160–188). Berlin: Springer.

Ohwada, H., & Mizoguchi, F. (1999). Parallel execution for speeding up inductive logic programming systems. In *Lecture notes in artificial intelligence: Vol. 1721. Proceedings of the 9th international workshop on inductive logic programming* (pp. 277–286). Berlin: Springer.

Ohwada, H., Nishiyama, H., & Mizoguchi, F. (2000). Concurrent execution of optimal hypothesis search for inverse entailment. In J. Cussens & A. Frisch (Eds.), *Lecture notes in artificial intelligence: Vol. 1866. Proceedings of the 10th international conference on inductive logic programming* (pp. 165–173). Berlin: Springer.

Papadimitriou, C. H., & Steiglitz, K. (1982). *Combinatorial optimisation*. Edgewood-Cliffs: Prentice-Hall.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning Journal*, *5*(3), 239–266.

Quinlan, J. R., & Cameron-Jones, R. M. (1993). FOIL: a midterm report. In P. Brazdil (Ed.), *Proceedings of the 6th European conference on machine learning* (Vol. 667, pp. 3–20). Berlin: Springer.

Rocha, R., Fonseca, N. A., & Santos Costa, V. (2005). On applying tabling to inductive logic programming. In *Lecture notes in artificial intelligence: Vol. 3720. Proceedings of the 16th European conference on machine learning, ECML-05* (pp. 707–714). Berlin: Springer.

Santos Costa, V., Srinivasan, A., Camacho, R., Blockeel, H., Demoen, B., Janssens, G., Struyf, J., Vandecasteele, H., & Van Laer, W. (2003). Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research*, *4*, 465–491.

Sebag, M., & Rouveirol, C. (1997). Tractable induction and classification in first order logic via stochastic matching. In *Proceedings of the 15th international joint conference on artificial intelligence* (pp. 888–893). San Mateo: Morgan Kaufmann.

Skillicorn, D. B., & Wang, Y. (2001). Parallel and sequential algorithms for data mining using inductive logic. *Knowledge and Information Systems*, *3*(4), 405–421.

Smith, R. G. (1980). The contract net protocol: high-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, *29*(12), 1104–1113.

Squyres, J. M., & Lumsdaine, A. (2003). A component architecture for LAM/MPI. In *Lecture notes in computer science: Vol. 2840. Proceedings, 10th European PVM/MPI users' group meeting*, Venice, Italy, 2003. Berlin: Springer.

Srinivasan, A. (1999). A study of two sampling methods for analysing large datasets with ILP. *Data Mining and Knowledge Discovery*, *3*(1), 95–123.

Srinivasan, A. (2000). *A study of two probabilistic methods for searching large spaces with ILP* (Technical Report PRG-TR-16-00). Oxford University Computing Laboratory.

Srinivasan, A. (2003). The Aleph manual. Available from http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph.

Srinivasan, A., & Kothari, R. (2005). A study of applying dimensionality reduction to restrict the size of a hypothesis space. In *Proceedings of the 15th international conference on inductive logic programming* (pp. 348–365).

Srinivasan, A., Muggleton, S., King, R. D., & Sternberg, M. J. E. (1994a). Mutagenesis: ILP experiments in a non-determinate biological domain. In S. Wrobel (Ed.), *GMD-Studien: Vol. 237. Proceedings of the 4th international workshop on inductive logic programming* (pp. 217–232).

Srinivasan, A., Muggleton, S., King, R. D., & Sternberg, M. J. E. (1994b). Mutagenesis: ILP experiments in a non-determinate biological domain. In S. Wrobel (Ed.), *GMD-Studien: Vol. 237. Proceedings of the 4th international workshop on inductive logic programming* (pp. 217–232).

Srinivasan, A., King, R. D., Muggleton, S., & Sternberg, M. J. E. (1997). Carcinogenesis predictions using ILP. In S. Džeroski & N. Lavrač (Eds.), *Proceedings of the 7th international workshop on inductive logic programming* (Vol. 1297, pp. 273–287). Berlin: Springer.

Tang, L. R., & Mooney, R. J. (2001). Using multiple clause constructors in inductive logic programming for semantic parsing. In *EMCL '01: proceedings of the 12th European conference on machine learning* (pp. 466–477). London, UK, 2001. Berlin: Springer.

Tobudic, A., & Widmer, G. (2003). Relational IBL in music with a new structural similarity measure. In *Proceedings of the 13th international conference on inductive logic programming* (pp. 365–382).

Todorovski, L., Ljubič, P., & Džeroski, S. (2004). Inducing polynomial equations for regression. In *Proceedings of the 15th European conference on machine learning* (pp. 441–452).

Turcotte, M., Muggleton, S. H., & Sternberg, M. J. E. (2001). Automated discovery of structural signatures of protein fold and function. *Journal of Molecular Biology*, *306*, 591–605.

Wielemaker, J. (2003). Native preemptive threads in SWI-Prolog. In C. Palamidessi (Ed.), *Lecture notes in artificial intelligence: Vol. 2916. Proceedings of the 19th international conference on logic programming* (pp. 331–345). Berlin: Springer.

Železný, F., Srinivasan, A., & Page, D. (2002). Lattice-search runtime distributions may be heavy-tailed. In S. Matwin & C. Sammut (Eds.), *Lecture notes in artificial intelligence: Vol. 2583. Proceedings of the 12th international conference on inductive logic programming* (pp. 333–345). Berlin: Springer.

Železný, F., Srinivasan, A., & Page, D. (2006). Randomised restarted search in ILP. *Machine Learning*, *64*(1–3), 183–208.