Mach Learn (2012) 86:141–168
DOI 10.1007/s10994-011-5245-8

Data and task parallelism in ILP using MapReduce

Ashwin Srinivasan · Tanveer A. Faruquie · Sachindra Joshi

Received: 26 July 2010 / Accepted: 28 February 2011 / Published online: 10 May 2011

Abstract Nearly two decades of research in the area of Inductive Logic Programming (ILP) have seen steady progress in clarifying its theoretical foundations and regular demonstrations of its applicability to complex problems in very diverse domains. These results are necessary, but not sufficient, for ILP to be adopted as a tool for data analysis in an era of very large machine-generated scientific and industrial datasets, accompanied by programs that provide ready access to complex relational information in machine-readable forms (ontologies, parsers, and so on). Besides the usual issues about the ease of use, ILP is now confronted with questions of implementation. We are concerned here with two of these, namely: can an ILP system construct models efficiently when (a) Dataset sizes are too large to fit in the memory of a single machine; and (b) Search space sizes becomes prohibitively large to explore using a single machine. In this paper, we examine the applicability to ILP of a popular distributed computing approach that provides a uniform way for performing data and task parallel computations in ILP. The MapReduce programming model allows, in principle, very large numbers of processors to be used without any special understanding of the underlying hardware or software involved. Specifically, we show how the MapReduce approach can be used to perform the coverage-test that is at the heart of many ILP systems, and to perform multiple searches required by a greedy set-covering algorithm used by some popular ILP systems. Our principal findings with synthetic and real-world datasets for both data and task parallelism are these: (a) Ignoring overheads, the time to perform the computations concurrently increases with the size of the dataset for data parallelism and with the size of the search space for task parallelism. For data parallelism this increase is roughly in proportion to increases in dataset size; (b) If a MapReduce implementation is used as part of an ILP system, then benefits for data parallelism can only be expected above some minimal

A. Srinivasan is also an Adjunct Professor at the School of Computer Science and Engineering, University of New South Wales; a Visiting Professor at the Computing Laboratory, Oxford.

A. Srinivasan (✉)
Faculty of Mathematics and Computer Science, South Asian University, New Delhi 110067, India
e-mail: ashwin.srinivasan@wolfson.oxon.org

T.A. Faruquie · S. Joshi
IBM Research—India, New Delhi 110070, India

dataset size, and for task parallelism can only be expected above some minimal search-space size; and (c) The MapReduce approach appears better suited to exploit data-parallelism in ILP.

Keywords ILP · MapReduce · Parallelism · Scaling-up

1 Introduction

As is the case with any engineering discipline, it is useful to classify research in the area of Inductive Logic Programming (ILP) broadly into conceptual, implementation, and application categories. The nomenclature of the first and last suggests their purpose: conceptual research is concerned with the development of the abstract, formal basis for the field; and application research is concerned with establishing its experimental, or empirical results. Working lockstep since 1991, ILP research in these two categories have been used to advance a scientific case for the role of first-order learning in constructing models for data in complex domains. The rapid growth and size of machine-generated scientific and industrial databases, along with the availability of information in an electronic form of complex relations (from ontologies, parsers, and so on) have, however turned the spotlight onto ILP implementations. The question confronting ILP is now not a scientific one of whether ILP is applicable to the analysis of such data—let us accept, for the moment that the framework is sufficiently powerful for this to be so—but whether an ILP tool can actually perform such an analysis efficiently. Here are some examples of the growth of data in some prominent application areas:

Biology. Automated genome sequencing techniques have made available millions of nucleotide sequences comprising entire genomes of organisms, that is increasing at an extremely rapid rate. The number of sequences in one of the three primary sources of such data, GenBank, (from the U.S. NIH) is about 100 million (containing about 100 billion base-pairs, or 25 gigabytes of data). These are now augmented by taxonomic databases, protein sequence databases, gene expression data from microarray experiments, metabolic pathway data and so on. The emerging field of Integrative Biology aims to use all these different sources of information to construct models for entire biological systems.

Text. It is now believed that a majority of data stored in organisations will be in the form of unstructured data (that is, data that do not have a data model in the conventional database sense of the term), much of it in the form of text. Estimates put this data as comprising up to 80% of an organisation's data volume, amounting to several terabytes of data each year. Clearly not all of this is useful, but it is there and in such large amounts and is a challenge for any analysis method devised to extract information.

Telecommunications. Telecommunication companies are facing significant data analysis challenges as they increase the number and types of services they provide. Data stored from telephone calls ("call data records" or CDRs) for an organisation can be nearly a terabyte a day, and have to be stored for months or years together to meet governmental regulations. Analysis techniques will thus have to be able to handle 10s to 100s of terabytes of data.

The immediate implication for data analysis tools, including ones in ILP, is that they have to have some mechanism for dealing with data volumes that range from 10s of megabytes at present, to 1000s of megabytes or more in the future. Along with this increase in data

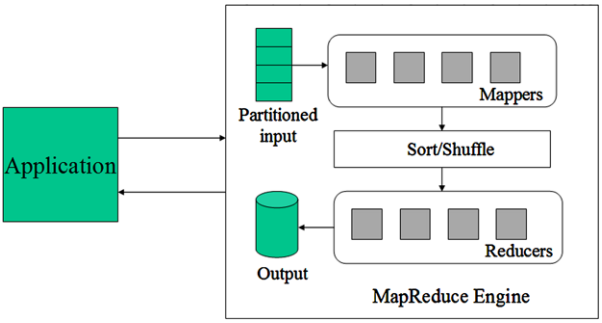

Fig. 1 Parallel computation using the MapReduce model

volume, is the availability in an electronic form of diverse kinds of information that may be potentially relevant for analysis. An ILP system applied to text analysis now has access to semantic lexicons like Wordnet (Miller et al. 1990), dictionaries, parsers, grammars and so on (Specia et al. 2007). These sources of information increase the size of the search space for an ILP system

Our interest in this paper is to examine the following question: is there a single, simple approach to perform effective data and task parallel computation in ILP? By "single" we mean that the same conceptual approach can be used for both kinds of parallelisation, and by "simple" we mean that the ILP system developer can use the approach with practically no knowledge of the underlying hardware, or of a number of book-keeping details associated with distributed computing.

Recently, an extremely simple parallel computing approach has been found to have applications to a wide-range of data-intensive tasks (Dean and Ghemawat 2008), including several propositional machine learning techniques (Chu et al. 2006). Although lacking the sophistication of an RDBMS, the so-called "MapReduce programming model" or "MapReduce framework" allows the analysis of very large datasets in a massively parallel manner. The system builder concentrates only on the logical definition of two functions *map* and *reduce*, loosely motivated by counterparts from functional programming. Once these functions are defined, the implementation uses the former to compute, concurrently, values for each data item; and the latter to "reduce" the values computed by the concurrent "maps" to a single value. The architecture of a system using the MapReduce programming model is broadly as shown in Fig. 1. We refer to a specific implementation of this programming model as a MapReduce implementation. There are several MapReduce implementations available such as Twister,[1] GridGain[2] and Hadoop.[3] These implementations differ in specific infrastructure that is supported by them. Recently a high level programming interface called SAGA has been proposed that provides the ability to create a MapReduce application in an infrastructure independent way (Miceli et al. 2009).

In this paper, we investigate the use of a popular MapReduce implementation (Hadoop: http://hadoop.apache.org/) by a well-known ILP engine (Aleph: Srinivasan 1999) to construct clauses. We examine its use in data parallelism using synthetic and real-world datasets that range from 10s of thousands to millions of data items. Our intent in these experiments is two-fold: (1) To investigate the performance of the MapReduce approach to evaluate

[1] http://www.iterativemapreduce.org/.

[2] http://www.gridgain.com/.

[3] http://hadoop.apache.org/.

clause utility as dataset size increases; and (2) To examine whether an ILP engine coupled with a MapReduce implementation can be used to evaluate clauses efficiently on very large datasets. For task parallelism, again using synthetic and real-world datasets with search-space size ranging from 10s of thousands to millions we examine the scaling performance of the ILP-MapReduce combination, and the efficiency of using the combination to search the space completely (but not exhaustively). In both cases, "efficiency" will be measured against performing the corresponding activity sequentially (on a single fast processor).

It is relevant to ask why ILP should be concerned with MapReduce. Two general reasons have already been mentioned, namely: the widespread application of the MapReduce model to very large datasets, and the abstraction (and, as will be seen later, the use of polymorphism) of a range of parallel computation tasks by definitions of two simple functions. To these, we add a third, namely, the recent demonstration that the MapReduce approach can be used by wide range of propositional learning algorithms (Chu et al. 2006). There are significant differences between propositional learning and ILP—the incorporation of multi-relational background knowledge, and the inter-dependence of examples are two important ones—making it unclear whether the advantages of using MapReduce in propositional learning carry over to the first-order setting. It is the intent of experiments here to shed some light on this.

The rest of the paper is organised as follows. In Sect. 2, we discuss some background material and related work. In Sect. 3, we provide logical definitions of *map* and *reduce* functions and discuss the effectiveness of the MapReduce approach under different scenarios. In Sect. 4, we describe a MapReduce implementation for both data and task parallelism. We discuss empirical evaluation of our proposed MapReduce implementations in Sect. 5 and present concluding remarks in Sect. 6. Finally, we provide details on some other ways of using MapReduce in ILP in the Appendix.

2 Background and related work

2.1 Clause-set identification with ILP

In this paper we focus on ILP systems that are concerned with identifying a set of clauses, although the techniques we develop are applicable to other kinds of ILP systems. We use a variant to the partial specification provided by Muggleton (1994) in which we seek to identify, using a cost function, clauses that entail a set of examples. More formally, given some language $\mathcal{L}$; a set of clauses representing background knowledge B; examples E consisting of some non-empty subset of "positive" examples $E^+ \subseteq E$ s.t. $B \not\models E^+$; a cost function f that returns the cost of a clause, given B and E; we want to identify a set of clauses H such that for each $e_i \in E^+$, there is an $h_i \in H$ s.t. $B \cup \{h_i\} \models e_i$. A greedy procedure for this is in Fig. 2. We note that at least two modifications are needed to realise a practical implementation: (1) Deciding logical entailment is hard, and some simpler relation has to be used in its place; and (2) The search procedure may not examine all clauses satisfying even this simpler relation. ILP implementations usually opt for the relation of subsumption in the sense described by Plotkin (1971), which ensures that clauses found still satisfy the entailment requirement. Complete search of the set of clauses subsuming an example, given B may also not be tractable, and search in Step 1 is restricted to some maximum size k. It

$ilp(B, E, f)$: Given background knowledge B; examples E with positive examples $E^+ \subseteq E$ and $B \not\models E^+$; and a real-valued cost function f, return a set of clauses H such that for each $e_i \in E^+$, $\exists h_i \in H$ s.t. $B \cup \{h_i\} \models e_i$.

1. $i := 1$
2. $H_0 := \emptyset$
3. $E^- := E - E^+$
4. $E_0^+ := E^+$
5. $E_0 := E_0^+ \cup E^-$
6. while $(E_{i-1}^+ \neq \emptyset)$
7. begin
   (a) Let $S_i = \{h_j$: for each $e_j \in E_j^+$ $h_j = search(e_j, B, E_{i-1}, f)\}$
   (b) Let $h_i \in S_i$ be a minimal cost clause in S_i. That is, $h_i = \min_{h \in S_i} f(h, B, E_{i-1})$
   (c) $H_i = H_{i-1} \cup \{h_i\}$
   (d) ${E_i}^+ = E_{i-1}^+ - \{e_i : e_i \in E_{i-1}^+$ and $B \cup \{h_i\} \models e_i\}$
   (e) $E_i = E_i^+ \cup E^-$
   (f) increment i
8. end
9. return H_{i-1}

Fig. 2 A greedy procedure to identify clauses which entail a set of examples E^+. In Step 7a search(e_j, B, E_{i-1}, f) returns a minimal cost clause h s.t. $B \wedge h \models e$. On each iteration, each example not yet entailed (the set E_{i-1}^+ in Step 7a is used to find a clause that entails it using a procedure *search*. The clause with the lowest cost is selected, the examples it entails are removed, and the procedure iterates

is not hard to see that even with this restriction, if $|E^+| = N$, then the number of clauses examined by the procedure is still $O(N^2)$.[4]

The cost function $f(h_i, B, E)$ returns some estimate of the utility of the element h_i. For reasons that will become apparent shortly, we are especially interested in the case when f is *data-decomposable* in the following sense. Suppose the sets $E_1, E_2, \ldots, E_n$ be any partition of the examples E. Then, for $h \in \mathcal{H}$, by "f is data-decomposable" we will mean $f(h, B, E)$ is some function of $\Phi(g(h, B, E_1), g(h, B, E_2), \ldots, g(h, B, E_n))$, where Φ is a function denoting the iterated operation of a binary function ϕ defined on the $g(h, B, E_i)$. An example of ϕ is the binary addition function $+$. Φ would then compute $\sum_{i=1}^n g(h, B, E_i)$, and f would be some function of $\sum_{i=1}^n g(h, B, E_i)$.

At this point in the paper, we note that the dominant flavours of ILP—"learning from interpretations" and "learning from entailment"—differ in at least one important way. In the former, examples are sets of statements that are independent of each other (technically, each example is a Herbrand interpretation containing all ground facts that are true for a particular object). An example in the learning from entailment setting is a statement (usually a ground fact) about one or more objects that may be related to each other and to objects in other examples given the background knowledge B. As we shall see, this possible dependence of examples requires some attention before using the MapReduce framework.

[4]On each iteration $i > 0$, the greedy procedure described here first finds the set S_i of minimal-cost clauses possible with each $e \in E_{i-1}+$ and then selects the best (that is, lowest cost) amongst this set. An ILP system like Progol uses a randomised variant of this procedure, in which S_i is restricted to a singleton set by first randomly selecting a single example from E_{i-1}^+. This makes the procedure $O(N)$, but clearly clause-set identified will vary with repetitions of the procedure.

2.2 Scaling-up ILP

We are concerned with using techniques that allow ILP systems to deal with very large datasets and search spaces. In this paper, large datasets will mean that the set of examples E given to an ILP system is usually too large to store in a single machine's memory. Large search spaces will mean that it will require excessive amounts of time to explore the space of possible clauses for identifying a low-cost set of clauses, even using a greedy procedure such as the one in Fig. 2.

It would appear that sampling theory provides adequate machinery to address issues of dealing with large datasets and search spaces, and their use has appeared in the ILP literature for both the problems (Srinivasan 1999, 2000; Cardoso and Zaverucha 2006; Zelezny et al. 2002). Recently, sampling based methods have also been proposed for scaling relational algorithms for multi-level frequent pattern discovery (Appice et al. 2010). Despite these efforts, there are some reasons to investigate methods that handle each of these problems exactly. First, some ILP procedures, like those concerned with finding first-order association rules, are formulated over the entire data. Secondly, sampling guarantees often only hold if the samples drawn satisfy some constraints: for example, of being the result of uniform random sampling of all of the data or the hypothesis space. This cannot always be ensured. So what has been done so far, without resorting to sampling? The answers lie in overall size reduction, or concurrent computation (parallelisation).

Parallel computation using some *ad hoc* partitioning of the entire data was first investigated (Wang and Skillicorn 2000) with a variant in Fonseca et al. (2008), but the results are not guaranteed to be the same as using all of the data. Provided the limitations imposed by Datalog are acceptable, the natural approach for an ILP system to address exact computations with large datasets is to store the data in a relational database (Ho et al. 2005). The principal difficulty is that it requires a substantial understanding of the relational database management system (RDBMS). Specifically, much depends on devising appropriate data indexing and partitioning. Exact data-parallel computation for the task of determining first-order association rules has been addressed in Clare and King (2003). This can be seen as a special case of the MapReduce approach we examine in this paper, in which the authors have developed special-purpose software specifically for the purpose of learning association rules by distributed processing of the example data by several worker nodes ("mappers" in the MapReduce context), and collating their results ("reducer" in MapReduce). Clearly, with the advent of general-purpose MapReduce software, this becomes unnecessary.

A survey and empirical investigation of exploiting data and task parallelism in several existing ILP techniques on some real-world datasets was conducted in Fonseca et al. (2008). In this, the authors describe three kinds of concurrent computation in ILP namely: search, data and evaluation parallelisation. For each of these three categories, different methods for concurrent computation were used along with the MPI message-passing communications protocol to communicate between several machines. In order to understand these results in the context of this paper, we first clarify that the terminology of data and task parallelisation of this paper is to be understood in the manner employed by general-purpose parallel computation. Search-parallelism in Fonseca et al. (2008) is thus a form of task parallelism here. But Fonseca's data-parallelism (concurrent identification of clause-sets using subsets of data) and evaluation-parallelism (concurrent computation of the examples entailed by a clause) are characterised as data parallelism here as they both involve concurrent computation over data instances. Thus, the results in Fonseca et al. (2008) can be re-phrased as follows: task parallelism is effective, but data parallelism, when used for clause evaluation, is not. These results should be assessed with the following caveats. First, experiments were conducted

only on datasets of very moderate data sizes. Secondly, although the MPI protocol has become a kind of *de facto* standard for communication between multiple machines, its use in ILP requires either that ILP systems provide libraries for using the protocol, or the user requires some knowledge of the procedures provided by the interface. Finally, although not especially serious, there was no single conceptual basis for data and task parallelism. That is, different procedures have to be developed specifically for each kind of parallelism.

On the question of complete (but not necessarily exhaustive) exploration of large search spaces, techniques have been proposed that search through hypotheses spaces partitioned using some syntactic measure (for example, the number of occurrences of a predicate symbol in any hypothesised clause (Camacho 1994) or a "language-number" in De Raedt and Bruynooghe (1991)). The principal difficulty here, of course, is the need for coming up with a way of meaningfully partitioning the hypothesis space. If complete exploration of the search space is not required, then a range of heuristics that examine smaller portions of the search-space are available. These include techniques like the greedy search methods employed by Quinlan (1990) and Blockeel and De Raedt (1998), the use of negative examples to reduce the search-space (first reported in Muggleton and Feng (1990)), and the use of statistical techniques for dimensionality reduction (Srinivasan and Kothari 2005).

2.3 Scaling-up propositional learning using MapReduce

The use of the MapReduce framework for data paralleisation for propositional machine learning algorithms has been examined in Chu et al. (2006). They show that the algorithms that fit the statistical query model (Kearns 1998) can be written in a certain "summation form" which can easily be exploited by the MapReduce framework.[5] They show linear speed up with number of cores for several machine learning algorithms such as k-means, logistic regression and SVM by using the MapReduce framework. ILP does not necessarily satisfy an important requirement of the approach in Chu et al. (2006), namely that data instances used for learning are independent of each other. As we mentioned earlier, this does not hold when learning from entailment.

To the best of our knowledge, there has been no work done on using MapReduce for task parallelisation of propositional learning algorithms. To some extent, this is not surprising since scaling-up in learning algorithms of this kind has largely come to mean handling very large datasets. An additional dimension of complexity is added with ILP, in which model construction can also involve independent searches through very large search spaces. In this paper, we show that such tasks can be executed in parallel by the MapReduce framework.

3 *map*, *reduce* and MapReduce

In this section, we provide logic programming definitions of the basic relations underlying the MapReduce approach. We note that a reconstruction using functional programming of MapReduce is available in Lämmel (2007). Nevertheless, the logic-based approach here is probably more familiar to the ILP-oriented reader: at any rate, no such description exists in the literature.

[5]The original motivation of the statistical query model in Kearns (1998), namely a framework for learning with noise using an oracle that returns a probability distribution over classes is not the important feature here. Rather, it is the fact that the utility of a model can be evaluated by summing over its performance over individual data items that is relevant, since this can be computed independently by mappers in the MapReduce setting.

```
% map: ((* → **) × [*]) → [**]
map(_F, [], []).

map(F, [X|Xs], [FX|FXs]):-
        apply(F, X, FX),
        map(F, Xs, FXs).
```

```
% reduce: ((* × * → *) × [*] × *) → *
reduce(F, [], X0):-
        identity(F, X0).
reduce(F, [X|Xs], R):-
        reduce(F, Xs, R0),
        apply(F, X, FX),
        apply(FX, R0, R).
```

Fig. 3 Logical definitions of the higher-order functions *map* and *reduce* available with most functional programming languages. The definitions are in the Prolog language, which is untyped—the intended types of the functions are shown here as a comment (the line following the "%" sign). We assume here that the function F provided as an argument to reduce/3 is a binary associative function with a unique identity element. This element is returned by identity/2. apply/3 takes a function F and an argument X and returns the functional term FX. For example, an implementation of this for the "squaring" function may be: apply($sqr, X, sqr(X)$). Multiple calls to apply/3 (as in the definition of reduce/3 above) simply achieves the equivalent of currying in functional programming

As is done in Lämmel (2007), we distinguish between the *map* and *reduce* higher-order functions of functional programming, and their recent incarnation within the MapReduce programming approach for data parallel computing (Dean and Ghemawat 2008). In the former, *map* is higher-order function that takes two arguments: a function, of type $x \rightarrow y$ and a list of x values, and returns a list of y values (the names x and y are arbitrary). *reduce* is a special case of a generic *fold* function that represents an iterated binary operation. For simplicity, we will assume here that the binary operation employed by *reduce* is associative and has a unique identity element (neither of these constraints are necessary for the general *fold* operation). *reduce* takes two arguments: a binary function defined on arguments of type x, a list of x values. The result is of type x. Example definitions of these two higher-order functions in the Prolog language are shown in Fig. 3 (these follow Naish and Sterling 2000).

Thus, *map* essentially applies a function to a list of values, and *reduce* computes a single value from a list of values. Ensuring appropriate type constraints are satisfied, we can therefore construct a new function that is the composition that "map's" a function to a list of values, and reduces this list to a single value. Further, unfolding the definition for *map* above, it is evident that the computation performed by *map* is logically equivalent to a conjunction of apply/3 calls. As long as the definition of apply/3 has no side-effects, it has been long understood that this conjunction can be evaluated concurrently. These two aspects—function composition and concurrent mapping—form the inspiration for the MapReduce procedure for distributed computing described in Dean and Ghemawat (2008).

Logically, the description of MapReduce in Dean and Ghemawat (2008) involves functions that have more complex type-definitions than the *map* and *reduce* we have just considered. In addition, the final computation is not just a composition of two, but three functions: we direct the reader to Lämmel (2007) for a reverse-engineering of Dean and Ghemawat (2008), that performs a plausible discovery of the types of the functions, and their definitions. We are concerned here with specialised variants of these functions. For the purposes of the paper, we will use the term "dictionary item" to mean a (*dictionary_key*, *dictionary_value*) pair, and a "dictionary entry" to mean a (*dictionary_key*, *list_of_dictionary_values*) pair. A "dictionary" will simply be a list of dictionary entries. The function definition *mapwithkey* is identical to *map*, with differences being in the types of the arguments. The function passed as argument deals with dictionary items, and has type $(k1, v1) \rightarrow (k1, v2)$. *mapwithkey* uses this function to map a list of dictionary items into another such list. We will simply pass the function provided to *mapwithkey* as an argument to the *map* function and assume that the difference in types can be handled by appropriate modification of the apply/3 function. The function *reducebykey* operates on

```
% mapreduce: (((*, **) → (*, * * *)) × (* * * × * * * → * * *) × [(*, **)]) → [(*, [* * *])]

mapreduce(MapFn, ReduceFn, DictItems, Result):-
    mapwithkey(MapFn, DictItems, NewItems),
    groupbykey(NewItems, [], Dictionary),
    reducebykey(ReduceFn, Dictionary, Result).

% mapwithkey: (((*, **) → (*, * * *)) × [(*, **)]) → [(*, * * *)]
% reducebykey: ((* × * → *) × [(**, [*])] → [(**, [*])]
mapwithkey(F, DictItems, Result):-
    map(F, DictItems, Result).

reducebykey(_F, [], []).
reducebykey(F, [(K, Vs)|KVs], [(K, [R])|Rs]):-
    reduce(F, Vs, R),
    reducebykey(F, KVs, Rs).

% groupbykey: ([(*, **)] × [(*, [**])] → [(*, [**])]
groupbykey([], D, D).
groupbykey([(K, V)|Items], D, Dictionary):-
    insertdict((K, V), D, D1),
    groupbykey(Items, D1, Dictionary).
```

Fig. 4 Logical definition of a map-reduce implementation

an entire dictionary. The list of dictionary values for each key in the dictionary is reduced to a single one by an application of the function *reduce* (given earlier). We need one additional helper function, *groupbykey* that inserts the dictionary items from *mapwithkey* into an existing (possibly empty) dictionary. The result is used by *reducebykey*. Prolog definitions of these three functions are shown in Fig. 4, along with the composition *mapreduce* (we do not show the functions needed for dictionary manipulations).

We note that the MapReduce implementation described in Dean and Ghemawat (2008) and the one used by us (Hadoop) are both more general than the definition in Fig. 4. The principal restrictions in the latter are: (a) *mapwithkey* returns dictionary items with the same keys as its input (the implementation in Dean and Ghemawat (2008) does not require this); and (b) *reducebykey* returns dictionary entries which have lists with a single value for each key (the implementation allows a set of values). Thus, all computations performed by *mapreduce* in Fig. 4 can be performed by the implementation.[6]

3.1 Effectiveness of the MapReduce approach

Let us distinguish first between the computation that can be done in parallel by the mappers in a MapReduce implementation, and the computation that needs to be done serially. We adopt the terminology of Shi (1996) for the following: t_s, the processing time, ignoring overheads, for the serial part (on a single processor), and $t_p(P, M)$, the processing time of the parallel part, including overheads, on P processors using M mappers. Further, by convention, we take $t_p(1, 0)$ to be the time taken to perform the parallel part on a single processor without any overheads for parallelisation. Thus, $T(1, 0) = t_s + t_p(1, 0)$, will be the total processing time of both parts using a single processor without using a MapReduce implementation, which for this paper will simply mean the ILP engine in isolation. $T(P, M)$,

[6]But the converse is not true, because of the use of multiple keys allowed by the implementations. The reader may be concerned that this does not conform to the relationship required in Hoare (1993) between a specification S and an implementation I (namely, $I \models S$). *mapreduce* thus does not constitute a specification of the implementation in Dean and Ghemawat (2008) or of Hadoop. Nevertheless, it is sufficient for our purposes that *mapreduce* can be implemented correctly by these programs.

is the same quantity with the ILP engine using a MapReduce implementation with access to P processors and M mappers. We are interested in estimating the speedup $S = \frac{T(1,0)}{T(P,M)}$.

Let us assume that the P processors perform the parallel computation by executing the M mappers concurrently, and that M is sufficient to perform the parallel computation. Let $r_1, r_2, \ldots, r_M$ be the processing time required by each of the M mappers. Then, ignoring overheads of using a MapReduce implementation, $t_p(1,0) = \sum_{i=1}^{M} r_i$. Without loss of generality, let us take the r_i to be in non-decreasing order. That is, $r_M = \max(r_1, r_2, \ldots, r_M)$. Assuming that t_s is unaffected by the use of a single or multiple processors, $S = \frac{t_s + \sum_{i=1}^{M} r_i}{t_s + r_M}$. If $t_s \ll r_M$, then we are able to simplify this further to $S = 1 + \frac{\sum_{i=1}^{M-1} r_i}{r_M}$. Since $\sum_{i=1}^{M-1} r_i < (M-1)r_M$, it is evident that we will always get sublinear speedups, with high values being obtained when $r_1, \ldots, r_m$ are roughly equal to each other. Lower values follow, on the other hand, if one of the processors starts to take disproportionately longer time than the others. For example, if $r_M = 2r_{M-1}$, then $S < M/2$. We would therefore expect the MapReduce implementation to be effective under the following circumstances: (a) reduction time is negligible (t_s is small compared to $t_p(P, M)$); and (b) mapper times are roughly comparable ($t_p(P, M)$ is not dominated by a single value).

In practice, even if these conditions are satisfied, speedups will be reduced by overheads of three kinds: (1) Resource files needed for the parallel computation will have to be distributed to each of the P processors. This is proportional to the number of P; (2) It will not always be possible to execute all the M mappers required concurrently. For example, if the processors are only able to execute m mappers at once, although r_M will remain the same, in the MapReduce implementation M mappers are executed by multiple (serial) invocations of m concurrent mappers. The associated overheads is dependent on the number of invocations M/m, which is proportional to M for a fixed m; and (3) Additional bookkeeping is needed prior to the sequential computation required by the reducer to collect results from each of the mappers. This is usually a sort operation that is proportional to $M \log(M)$. Thus, $t_p(P, M) \approx r_M + k_1 P + k_2 M + k_3 M \log(M)$. More generally, we expect $t_p(P, M) = r_M + t_o(P, M)$, where $t_o(P, M)$ increases with increase in P or M. This way of decomposing the total computation time of a job in a MapReduce implementation is similar to Miceli et al. (2009).

4 Parallel computation in ILP using MapReduce

We are now in a position to examine the use of the MapReduce approach of Dean and Ghemawat (2008)—as defined by the *mapreduce* function in Fig. 4—in ILP. Using Fig. 2 as a template, there are at least two areas where this can be used: (1) Task parallelism, by performing the different *search* calls in Step 7a concurrently using all of the data; and (2) Data parallelism, by performing concurrent cost computations within any one *search* call in Step 7a, using subsets of the data. We consider these in reverse order, since the MapReduce approach has largely been devised for data parallel computation. In both cases, the architecture used for the hybrid ILP-MapReduce engine will be identical, as shown in Fig. 5. What will be different is the information communicated back and forth between the ILP and MapReduce engines, and the definitions of *map* and *reduce* used to accomplish each kind of parallelisation.

4.1 Data parallel computation

Consider the evaluation of the cost function $f(h, B, E_{i-1})$ in Step 2 of Fig. 2. We will assume that the true cost h will require evaluation on the entire set of examples E (rather

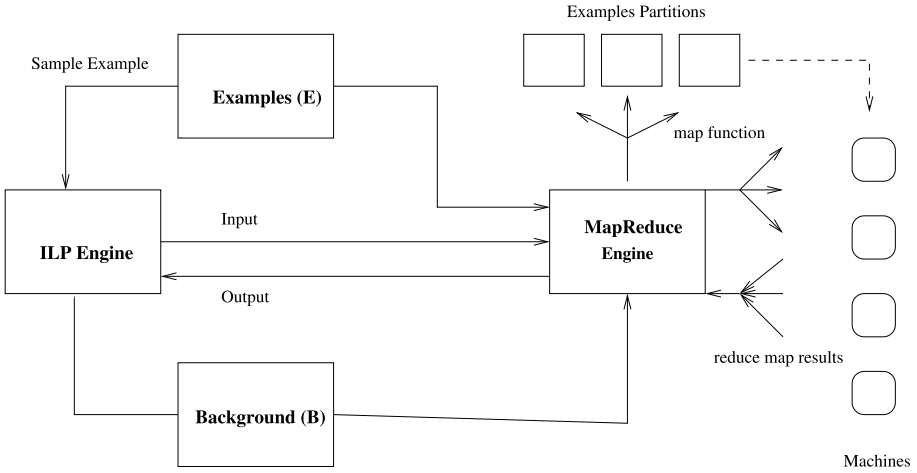

Fig. 5 Generic architecture of the ILP-MapReduce hybrid

than simple E_{i-1}). For convenience, we will also restrict ourselves to classification problems in which the examples are pre-classified into one of 2 classes (for convenience, denoted here as "+" and "−"). Each element of E is therefore a pair (x, c), where x is a data instance and c is its class value. It has been shown in Lavrač et al. (1999) that a number of cost functions for ILP can be re-formulated as functions of the 2×2 confusion matrix arising from using the clause h in conjunction with B to classify the E.

In this section, we employ a MapReduce implementation to achieve one kind of data parallelism in ILP, namely, the parallel computation of the confusion matrix associated with a clause h in Step 7b.[7] This can then be used by the ILP engine to compute $f(h, B, E)$. Specifically, it is our intention to employ an ILP engine that implements the procedure in Fig. 2 along with a MapReduce engine in the manner shown in Fig. 6. That is, each mapper m receives a clause h, background knowledge B and some subset of examples E_m. Each mapper m computes the confusion matrix on E_m for h given B. The reducer computes the final confusion matrix for h using the matrices computed by each of the mappers.

For using the MapReduce setting, we need to define two functions, *map* and *reduce*, which we will do as follows:

The map function g_D. Although we are really concerned with $g_D(h, B, E_m)$, we will write this as $g_D(E_m)$ since h and B are constant across all mappers:

$$g_D(E_m) = \begin{bmatrix} |TP| & |FP| \\ |FN| & |TN| \end{bmatrix}$$

where, $TP = \{(x, c) : (x, c) \in E_m \text{ and } c = + \text{ and } B \wedge h \vdash x\}$, $FP = \{(x, c) : (x, c) \in E_m \text{ and } c = - \text{ and } B \wedge h \vdash x\}$, $FN = \{(x, c) : (x, c) \in E \text{ and } c = + \text{ and } B \wedge h \nvdash x\}$, and $TN = \{(x, c) : (x, c) \in E \text{ and } c = - \text{ and } B \wedge h \nvdash x\}$. Here $\vdash$ means "derives",

[7] This corresponds to what is called evaluation-parallelism in Fonseca et al. (2008). Data-parallelism in that paper refers to parallel construction of intermediate clause-sets using subsets of data. For completeness, we demonstrate that this can be formulated in the MapReduce framework in the Appendix.

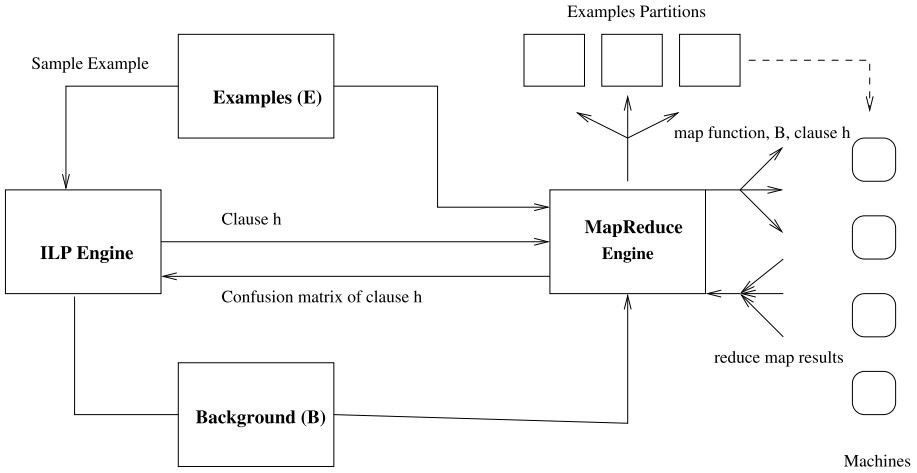

Fig. 6 Using a MapReduce implementation in conjunction with an ILP engine for data parallel cost computation

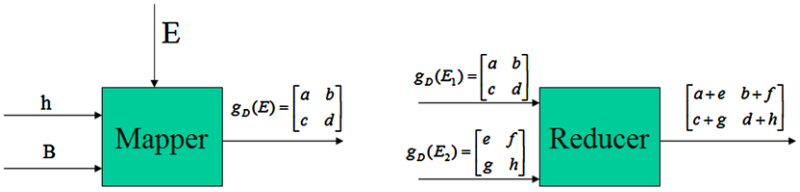

Fig. 7 Map and Reduce functions for data parallel cost computation. E_1 and E_2 are disjoint sets of examples

The reduce function ϕ_D. We need to define a binary operation that is associative and has a unique identity element. Let $E_1, E_2, \ldots, E_n$ be any partition of the E_m (that is, $\bigcup E_i = E_m$ and $E_i \cap E_j = \emptyset$ for $i \neq j$). It is straightforward to show that $g_D(E_m) = g_D(E_1) \oplus g_D(E_2) \cdots \oplus g_D(E_n)$, where $\oplus$ denotes matrix addition. We let $\phi_D = \oplus$, which is clearly associative and has a unique identity element.

We note that $f(h, B, E) = cost(g_D(h, B, E)) = cost(\bigoplus_i g_D(E_i))$, is a special case of the kind of data-decomposable function mentioned earlier. A diagrammatic view of the mapper and reducer for this form of data-parallel computation is shown in Fig. 7.

We now turn to the logical view of Fig. 6 using the *mapreduce*/4 predicate in Sect. 3. Let h denote the clause for which we need to compute the confusion matrix. We assume a helper function *cartesian* that, given the set $\{h\}$ and a set of examples E, returns the Cartesian product $\{h\} \times E$, which forms the list of dictionary items over which the *map* function is applied. The *map* function g_D maps each such dictionary item of the form (h, e), where e is any example, to another with the same key (h) and a 2×2 matrix as its value. The list of dictionary items computed by g_D are converted into a dictionary by *groupbykey* in which all matrices associated with the clause h are grouped together. These are then summed over by the reduction function using the binary matrix addition function $\oplus$. The result is the full confusion matrix for h (the f value can be obtained by the ILP implementation by mapping the function *cost* over this confusion matrix. This would give a list containing the dictionary

Fig. 8 Using *mapreduce*/4 to perform the cost computation within an ILP system. The background knowledge B is not shown, with the understanding that it is the same for all cost computations, and will be used by the *map* function g_D

```
% costmapreduce: H × E (Clause h and mapper examples E)
%                 × (H, E) → (H, M) (the function g_c)
%                 × M × M → M (the function φ_c)
%                 → [(H, M)] (the result)
costmapreduce(Clause, E, MapFn, ReduceFn, Cost, DictCMatrix):-
    cartesian(Clause, E, DictItems),
    mapreduce(MapFn, ReduceFn, DictItems, DictCMatrix).

apply(g_D, (Clause, Example), (Clause, Matrix)):-
    g_D((Clause, Example), (Clause, Matrix)).

apply(φ_D, (Clause, Matrix), (Clause, Result)):-
    identity(φ_D, I),
    φ_D(Matrix, I, Result).

apply((Clause, Matrix1), (Clause, Matrix2), (Clause, Matrix)):-
    φ_D(Matrix1, Matrix2, Matrix).
```

item of the form (h, f) where f is the cost of h). The principal predicate definitions are shown in Fig. 8, including a definition for *apply* that uses a polymorphic type definition.

4.1.1 An ILP-specific issue

We turn now to an ILP-specific issue that although apparent from the logical view, requires further clarification before the MapReduce framework can be used for data parallelism.

We note first that the *map* function is clearly identical across all machines, which are also provided the background knowledge required for performing the coverage test of a clause on the set of examples allocated to that machine. Traditionally, the two dominant ILP variants have differed on how examples are represented. Within the "learning from interpretations" setting, all information relevant to an example are kept together and separately from the background knowledge. Within the "learning from entailment" setting, information relevant to an example is obtained from the background knowledge using meta-data about the example. This usually means that what constitutes background knowledge in the latter setting is different (and bulkier) than in the former. Current MapReduce implementations are better suited to settings where all the information for an example are kept together, since it makes the distribution of the examples easier. It also defeats the purposes of distributing the data, if very large amounts of background knowledge have to be provided to each *map* function. Thus, as it stands, MapReduce implementations appear to be best suited for ILP problems that are naturally formulated within the learning from interpretations setting. However, for generality problems within the learning from entailment setting also need to be addressed. We achieve this here by first "saturating" each example e (in the sense described by Muggleton (1995)) to obtain all the relevant information from the background knowledge B. Each example e is then represented by its saturated version—in this paper, represented by the clause $\bot_{\mathcal{L}}(B, e)$ that conforms to language restrictions in $\mathcal{L}$. These saturated examples are distributed by the MapReduce implementation. This pre-computation of saturated clauses is also been performed for different reasons in Fonseca et al. (2008).

4.2 Task parallel computation

We now describe how the same MapReduce implementation can be used to achieve a form of task parallelism for ILP. Specifically, we demonstrate its use in the parallel construction of

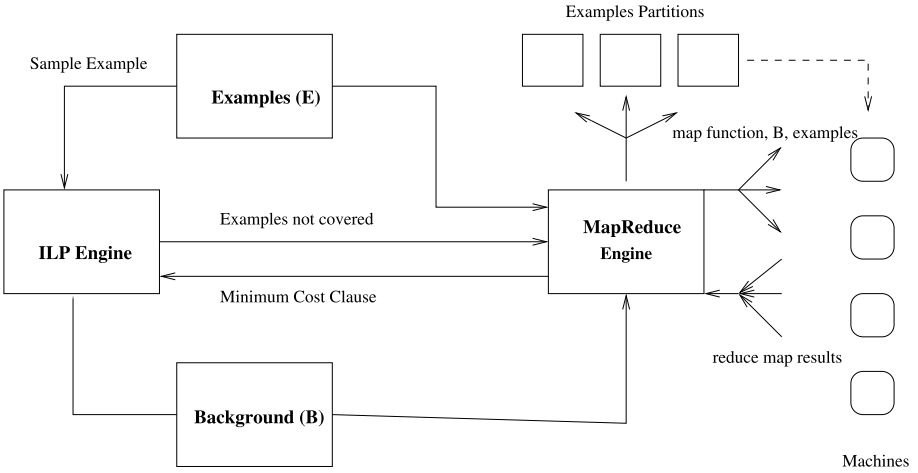

Fig. 9 Using a MapReduce implementation in conjunction with an ILP engine for task parallel search

the clause-set S_i on each iteration i of the greedy procedure in Step 7a of Fig. 2.[8] Assuming each invocation of the *search* function in that step is independent of each other, it is evident that these can be performed in parallel by different mappers. Further, rather than simply put these results together (using a union operation), the reducer can in fact be used to perform the selection in Step 7b. These points are indeed correct. However, there are some peculiarities to the greedy implementation in Fig. 2, requiring changes to what is communicated to each mapper. First, in an iteration i, clauses are obtained using the set E_i, which contains the set of positive examples E^+_{i-1} remaining to be covered. Secondly, to be useful each clause in iteration i has to cover at least one of these examples. This means that each mapper requires, along with the background knowledge B, the examples E_{i-1} and a set of positive examples that it needs to cover. Clearly, to minimise work, we would need to partition the E^+_{i-1} amongst the mappers. The combined use with an ILP engine is therefore as in Fig. 9.

For the logical definition, we assume the presence of a helper function *selectexamples* that selects each of the k examples in E^+_{i-1} and creates a trivial list of dictionary items $[(e_1, e_1), (e_2, e_2), \ldots, (e_k, e_k)]$. The map and reduce functions then are as follows:

The map function g_T. When applied to a dictionary item (e_j, e_j), the map function returns the dictionary item $(e_j, (f_j, h_j))$, where $h_j = search(e_j, B, E_{i-1}, f)$, and $f_j = f(h_j, B, E_{i-1})$.

The reduce function ϕ_T. The reduce function is applied to dictionary entries of the form $(e_j, (f_j, h_j))$. We will take this to be the iterated application of the binary operation ϕ_T, defined as follows:

$$\begin{aligned}\phi_T((e_i, (f_i, h_i)), (e_j, (f_j, h_j))) &= (e_i, (f_i, h_i)) \quad \text{if } f_i < f_j \\ &= (e_j, (f_j, h_j)) \quad \text{otherwise}\end{aligned}$$

[8] In Fonseca et al. (2008), task parallelism is employed to perform multiple random restarts of a search procedure that returns a clause-set on each restart. The greedy procedure we use here is the basis for a number of popular ILP implementations. Nevertheless, for completeness, a MapReduce formulation of a multiple random restart search is in the Appendix.

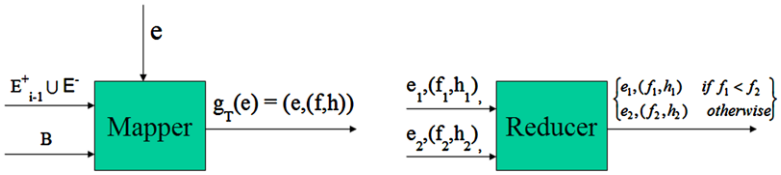

Fig. 10 Map and Reduce functions for task-parallel computation

Fig. 11 Using *mapreduce*/4 to perform concurrent search and selection in one iteration of the greedy search procedure in Fig. 2 (Steps 7a and 7b)

```
% searchmapreduce: E (examples E)
%                   × (E, N) → (E, (ℜ, H)) (the map function g)
%                   × (ℜ, H) × (ℜ, H) → (ℜ, H) (the function φ)
%                   → [(E, (ℜ, H))] (the result)
searchmapreduce(E, MapFn, ReduceFn, Result):-
        selectexamples(E, DictItems),
        mapreduce(MapFn, ReduceFn, DictItems, Result).

apply(g_T, (Example, Example), (Example, (Clause, Cost))):-
        g_T((Example, Example), (Example, (Clause, Cost))).

apply(φ_T, (E, ClauseCost), (Example, Result):-
        identity(φ_T, I),
        φ_T((E, ClauseCost), I, Result).

apply((E1, ClauseCost1), (E2, ClauseCost2, (E, ClauseCost)):-
        φ_T((E1, ClauseCost1), (E2, ClauseCost2), (E, ClauseCost)).
```

Clearly ϕ_T is associative. We define a unique identity element (*false*, $(\infty, \square)$) (where $\square$ denotes the empty clause, and is taken by convention to be the only clause with infinite cost).

mapreduce with these functions results in a list $[(e_i, (f_i, h_i)]$ containing the example e_i that resulted in the cost-minimal clause h_i, along with its score f_i. Thus, the map step performs Step 7a and the reduce step performs Step 7b of Fig. 2. A diagrammatic view of the mapper and reducer for this form of task-parallel computation is shown in Fig. 10.

The complete definition using these functions is in Fig. 11 (again, *apply* requires polymorphic types).

It is feasible, of course, to employ a MapReduce engine along the lines of the previous section to perform data parallel computations required during the search conducted by each mapper. This is beyond the scope of this paper, in which we are concerned with an investigation of data and task parallelisation in isolation.

5 Empirical evaluation

5.1 Aims

We intend to investigate the effectiveness of using MapReduce for data and task parallel computation in ILP, as embodied by computing clause matrices and performing parallel search. The question we examine is the following:

Question. How does the performance of a hybrid ILP-MapReduce implementation change as the size of data or size of search space increases?

We obtain empirical answers to this question using synthetic problems of varying data and search sizes (in the case of the former, examples ranging from few thousands to millions; and the latter, search space varying from few thousands of nodes to millions). The experiments are intended to provide some insight into how processing time $t_p(P, M)$ changes as size changes for a given P and M; and how speedup $S = T(1, 0)/T(P, M)$ changes as size, P and M vary. In fact, to make matters simpler, we vary only size and P: M is fixed at $2P$ for $P > 0$ (this is the default option for the MapReduce implementation we have used). We then examine S values on real datasets to determine the extent to which results from the synthetic data are indicative of real performance.

It is important to note here that we are not interested in comparisons against a "pure" ILP implementation that can store the data in its entirety in the random access memory of a single machine and perform search on that machine. Our experiments are meant to inform the use of ILP where the datasets and search spaces are sufficiently large to require some form of parallel processing: the question being studied here is how well the MapReduce model fits that requirement.

5.2 Materials

Data Data for experiments are in two categories:

Synthetic. We use the "Trains" problem posed by R. Michalski. For data parallelism, four datasets of sizes varying from 100 000 examples to 5 million examples are generated. For this we use S.H. Muggleton's random train generator[9] that defines a random process for generating examples. A single draw from the random process gives us one example. We can generate a dataset having E examples by drawing as many times from this random process. Each example is a random train with the number of carriages in the train following a multinomial distribution. We use a distribution that is biased towards generating long trains. For task parallelism, we restrict ourselves to a dataset of 40 positive and 40 negative examples. Multiple synthetic problems are generated by varying the number of search nodes from $k = 5\,000$ to 100 000 for each search. For the trains problem with a single clause target (this is the case with the problem as originally posed), k nodes and N positive examples, the total search space is kN. Thus, the synthetic experiments examine search spaces of sizes ranging from 200 000 to 4 million.

Real. For data parallelism, we examine three real-world datasets: (a) *HIV*. This is a Prolog-representation of the atom-bond structure of molecules in the NCI-HIV dataset consisting of approximately 42500 compounds; (b) *Yeast.* This is the data provided for prediction of gene function in the KDD cup competition of 2001; and (c) *Zinc.* This is a free database of commercially available compounds for virtual screening (Irwin and Shoichet 2004). We use a subset of clean-drug-like dataset that contains 2.3 million compounds. For task parallelism, we use 3 datasets commonly used in the ILP literature, namely: the mutagenesis dataset containing 188 compounds (King et al. 1996) (125 of which are positive); the carcinogenesis dataset containing 337 compounds (King and Srinivasan 1996) (182 of which are positive); and the DSSTox dataset (Muggleton et al. 2008) containing 550 compounds (194 of which are positive). Of these datasets, "HIV" and "Zinc" are problems of learning from interpretations, and "Yeast" is a problem of learning from entailment.

A summary of all datasets used is in Fig. 12.

[9] http://www.doc.ic.ac.uk/~shm/Software/GenerateTrains/.

Dataset	Size
Trains_D_10K	≈ 10 MB
Trains_D_100K	≈ 100 MB
Trains_D_500K	≈ 500 MB
Trains_D_1M	≈ 1 GB
Trains_D_5M	≈ 5 GB

Dataset	Size
HIV	≈ 45 MB
Yeast	≈ 800 MB
Zinc	≈ 7 GB

(a) Synthetic and real data used for evaluation of data parallelism

Dataset	Search space (nodes)
Trains_T_1K	40 000
Trains_T_5K	200 000
Trains_T_10K	400 000
Trains_T_50K	2 000 000
Trains_T_100K	4 000 000

Dataset	Search size (nodes)
Mut (188)	1.25×10^2–7.8×10^8
Carc	1.82×10^2–1.6×10^9
DSSTox	1.94×10^2–2.4×10^9

(b) Synthetic and real data used for evaluation of task parallelism

Fig. 12 Datasets used for experimental evaluation. We are only able to estimate an upper bound on the size of the search space for the real data sets in (**b**), since we do not know beforehand precisely which clauses would be found on each iteration of the greedy search in Fig. 2. The upper bound is taken to be kN^2 where k is the maximum number of nodes in the search with any one example, and N is the number of positive examples (the bound follows from assuming that no clauses are found on any of the iterations of the greedy search). The lower bound follows trivially from the assumption that no more than a single node is examined for each positive example, and that the greedy procedure terminates after 1 iteration. In the tabulation above and the experiments in the paper, $k = 50\,000$

Algorithms and machines We distinguish here between three separate sets of procedures:

1. The ILP engine, equipped with a search procedure that is concerned with identifying clauses. For this, we use the ILP engine Aleph (Srinivasan 1999);
2. The MapReduce engine, that is concerned with data distribution, application of *map* and *reduce* functions in accordance with the definitions given in the main paper. For this we use Hadoop[10] which is an open source implementation of MapReduce; and
3. Definitions of the functions *map* (g_D and g_T) and *reduce* (ϕ_D and ϕ_T). We implement the *map* function as a set of clauses in the Prolog language. The *reduce* function is simply a counting program that is implemented using standard programs provided by the Unix operating system. We use the generic API provided by Hadoop Streaming, that allows mappers and reducers can be written in almost any language. Both kinds programs receive their inputs and write their outputs in standardised ways.

Our choice of Aleph is based on three factors: (a) It is a program that we are intimately familiar with; (b) Our own familiarity with Aleph notwithstanding, it remains perhaps one of the most widely used ILP systems; and (c) It has the flexibility to emulate a number of different ILP systems. We are thus able to conduct experiments by both learning from entailment and interpretations. The specific aspects of the ILP engine we have elected to parallelise, namely, clause evaluation and clause-set identification are still at the heart of many ILP engines. Replacement of Aleph with any other ILP system is straightforward, since all our communication with the MapReduce implementation is through the rudimentary mechanism of text files.

We have access to two separate clusters of machines, which in effect gives us access to 16 cores on each cluster. Cluster1 actually consists of 16 quad-core processors each equipped

[10]See: http://hadoop.apache.org/.

with up to 250 GB of external disk storage. Cluster2 consists of 8 dual-core processors, each equipped with up to 600 GB of disk storage. However, with Cluster1, we were operating in a multi-user environment, in which the MapReduce engine is only allowed access to 1 core per processor for any task; and not all of the disk storage. Cluster2 operated in a single-user environment, allowing access to all the cores and disk storage. However, its configuration make it substantially slower than Cluster1 (we provide an estimate of the difference later in the paper). Requirements of disk storage space force us to perform experiments on the real data on Cluster2. At any rate, in either case, the total number of cores is the same (16), and we will henceforth use the term "processor" and "core" interchangeably to mean nominally, a single-core processor. In all cases, we use the default setting for the MapReduce engine that restricts the number of mappers to twice the number of cores.

5.3 Method

Preparatory work for the experiments was as follows. A set of approximately 5000 legal clauses were generated. We obtained the coverage-testing time of these clauses on a single machine over a dataset containing 100 000 examples generated randomly using the random train generator. In order to control for the role of clause complexity on the coverage-test, we consider 2 classes of clauses as follows. Members of a "simple" class were obtained by selecting 100 clauses with the lowest coverage-testing time (R_l). Similarly, 100 clauses with the highest coverage-testing time (R_h) were selected as representative of the class of complex clauses. Two different experiments were conducted to investigate data parallelism, using a ILP-MapReduce hybrid along with relevant background knowledge and the logical definitions for *map* and *reduce* described in Sect. 4.1:

DP Scale. This investigates increases in the time to compute the coverage of a clause, with a fixed number of processors, as data size increases:

For synthetic datasets D = Trains_D_10K ... Trains_D_5M
(a) With clause-complexity $R = R_l, R_h$
  i. Select a clause C with coverage-testing time R.
  ii. Record the time to obtain the confusion matrix of C using data D using the MapReduce engine with a fixed number of processors (> 1).
(b) For each value of R, let T_R be the average time to obtain the coverage of a clause with complexity R.

1. For each value of R, plot the average time T_R obtained (Y) against the size of the dataset (X).

DP Speedup. This investigates the variation, as data size and the number of processors change, in speedup $S = T(1,0)/T(P,M)$ in obtaining the coverage of a set of clauses in an iteration of the greedy procedure in Fig. 2:

For processors (correctly, cores) $P = P_1, P_2, \ldots$ and $M = 2P$
  For synthetic datasets D = Trains_D_10K ... Trains_D_5M
    i. Use the ILP engine to generate a set of clauses S_D constituting the search-space for the best clause that entails at least one positive example in D given background knowledge B.
    ii. Obtain the confusion matrix of each $c \in S_D$ using the MapReduce engine with P processors. Let the total time for this be $T(P,M)$.
    iii. Obtain the confusion matrix of each $c \in S_D$ using the ILP engine with a single processor. Let the total time for this be $T(1,0)$.

iv. Find the ratio $S = T(1,0)/T(P,M)$.
(To ensure comparability, we determine a set S_D that can be used across all of the D.)

Analogous experiments were conducted for task parallelism as follows, this time using the logical definitions for *map* and *reduce* described in Sect. 4.2:

TP Scale. The performance of the ILP-MapRreduce hybrid as the search space increased was investigated as follows:

For synthetic datasets D = Trains_T_5K ... Trains_T_100K
(a) Record the time T_D to obtain a hypothesis using the steps in Fig. 2 for problem D using the ILP-MapReduce engine with a fixed number of processors (> 1).
1. For each D, plot the time T_D to obtain the hypothesis (Y) against the size of the search space (X).

TP Speedup. We investigate the variation of speedup $S = T(1,0)/T(P,M)$ with size and processors as follows:

For processors (correctly, cores) $P = P_1, P_2, \ldots$ and $M = 2P$
For synthetic datasets D = Trains_T_5K ... Trains_T_100K
i. Use the ILP-MapReduce engine to generate a hypothesis using the steps in Fig. 2 for D with P processors. Let the total time for this be $T(P,M)$.
ii. Use the ILP engine to generate a hypothesis using the steps in Fig. 2 for D with a single processor. Let the total time for this be $T(1,0)$.
iii. Find the ratio $S = T(1,0)/T(P,M)$.

The following additional points are relevant:

1. Although the greedy procedure in Fig. 2 is deterministic—in the sense that given a set of examples and background knowledge, repetitions always return the same hypothesis—parallel environments are still subject random effects. Processors may not always be idle, network traffic may vary, and so on. Correctly therefore, the entire set of experiments above should be repeated several times, and average values obtained for all quantities. This is computationally expensive, especially for very large search-spaces for task-parallelism. Using limited repetitions (5) of experiments for data-parallelism and smaller search-spaces for task-parallelism, we find that the variation in values is small for our hardware environment. We report average values wherever available, with the caveat that for large values of search-spaces, the numbers reported for task-parallelism have a greater degree of uncertainty.
2. Our principal interest is to construct hypotheses quickly on real-world datasets. We also conduct **DP Speedup** and **TP Speedup** on the non-synthetic datasets described in Sect. 5.2 to examine the extent to which results on the synthetic data are indicative of real performance.

5.4 Results

Results on synthetic data relevant to data parallelism are shown in Figs. 13 and 14. The corresponding results for task parallelism are in Figs. 15 and 16. The principal details in these experimental observations are these:

Finding 1. Benefits of using the MapReduce implementation are evident only if data and search sizes are above some minimal threshold. While the precise value of this size will be problem-dependent, provided sufficiently large number of processors are available

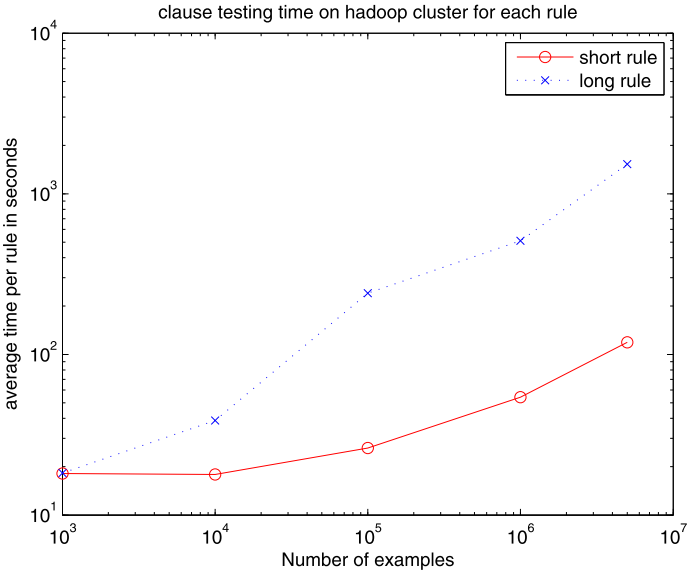

Fig. 13 *DP Scale Results*. Mean coverage time per clause in seconds using MapReduce implementation for data parallelism on synthetic data, with a fixed number of processors (here, 16). "Simple" means clauses with low coverage testing time (R_l) and "complex" means clauses with high coverage testing time (R_h), as described in the text

Fig. 14 *DP Speedup Results*. Data parallelism speedups as a function of number of processors (correctly, cores) and data size. We use abbreviations for the datasets: "10K" stands for "Trains_D_ 10K" and so on

Processors	Dataset				
	10K	100K	500K	1M	5M
4	0.19	0.97	2.71	2.48	2.97
8	0.19	0.89	2.74	3.12	5.23
16	0.19	0.82	3.97	5.88	13.21

Fig. 15 *TP Scale Results*. Results for task parallelism on synthetic data using the ILP-MapReduce hybrid with a fixed number of processors (correctly, cores. Here, this is equal to 16)

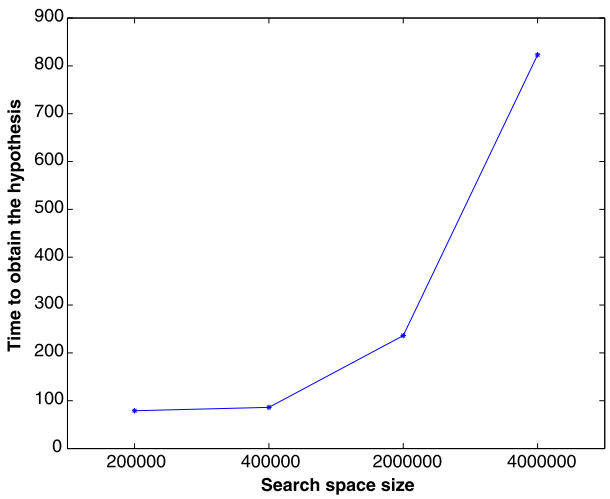

Fig. 16 *TP Speedup Results*. Task parallelism speedups as a function of number of processors (correctly, cores) and search size. We use abbreviations for the datasets: "1K" stands for "Trains_T_ 1K" and so on

Processors	Dataset				
	1K	5K	10K	50K	100K
4	0.03	0.21	0.69	2.67	2.73
8	0.03	0.20	0.61	2.95	2.86
16	0.03	0.20	0.59	2.98	2.98

(16 in our case), our synthetic experiments suggest the data threshold is approximately 125 megabytes; and the search-space threshold is about 700 000 nodes.

Finding 2. Ignoring overheads, the time to obtain the confusion matrix for a clause using a MapReduce engine grows with the size of the dataset. Similarly, as the size of the search space increases, so does the time to obtain the best clause. In themselves, these observations are unsurprising. Interest lies in *how* time grows as size increases. For clauses with a given complexity, the increase in time for data parallelism is roughly linear with data size (after sizes are above some minimal size). The increase in time for task parallelism is non-linear with increase in search space (that is, increases in time are disproportionate to increases in search size: the most likely reason for this will be discussed shortly).

Finding 3. Even when speedups are obtained, the speedups for task-parallelism appear to be well below the maximum value (equal to the number of mappers, which should be at least the same as the number of processors).

We now investigate reasons for each of the observations in greater detail:

Why are benefits only apparent after a certain size? The answer to this lies in the overheads associated with the MapReduce implementation. Recall that the time for parallel execution requiring P processors with M mappers is $t_p(P, M) = r_M + t_o(P, M)$ where t_o computes the overheads as a monotonic function of P and M, and r_M is the time taken by the slowest mapper (Sect. 3.1); and speedup $S = \frac{t_s + \sum_{i=1}^{M} r_i}{t_s + r_M + t_o(P,M)}$. At small sizes—for example, 10K for data parallelism and 1K for task parallelism—we first note that $S < 1$. A little manipulation will show that speedups less than 1 follow whenever $t_o(P, M) > \sum_{i=1}^{M-1} r_i$. It is the case that this condition holds at small data and task sizes for the problems examined here. As data size increases, M increases and r_i may increase and the condition ceases to hold at some point. As search size increases, r_i increases (but M stays constant) and, once again, the condition ceases to hold at some point. Some of the effects of the overheads can be lessened, at least for data parallelism, by reducing the need to repeatedly transfer resource files each time a clause is to be evaluated. If several clauses can be evaluated at once (this is a feature of some ILP systems like HYPER and Tilde), then this cost is in some sense, amortized. Evidence for this is seen from Fig. 17.

Why do the times for task parallelism increase non-linearly with search size? This is an artifact of the particular search performed by the ILP engine we have chosen. The Aleph program performs a general-to-specific search. Increasing the size of the search space results, usually, in increasing the specificity of clauses examined (to a first approximation, the number of literals in clauses increases). In turn, this usually increases the time taken to obtain the clause utility. Aleph determines utility using a subsumption test, whose time increases non-linearly with increase in the number of literals in the clause. Thus, the corresponding time for a search conducted by a mapper also increases non-linearly. We would not expect to see this behaviour with an ILP system that performed a specific-to-general search.

Why are speedups so modest for task parallelism? This is an artifact of the data we have chosen. We will concentrate here on low speedups for high search sizes (for moderate to

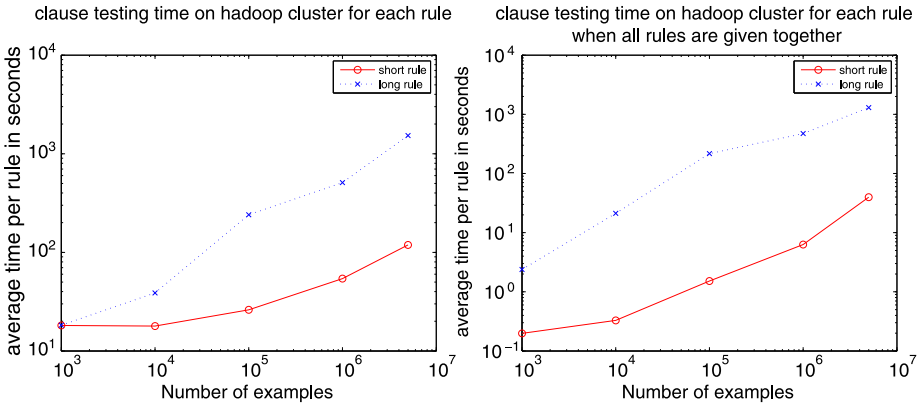

Fig. 17 Mean coverage time per clause in seconds using MapReduce implementation for data parallelism, when: clauses are provided one-at-a-time to the MapReduce engine as before (*left*); and a set of clauses are provided (*right*)

Fig. 18 Variation of the time taken by the slowest mapper for task parallelism, against the total time taken by the remaining mappers. Dataset abbreviations are as before

Dataset	$A = r_M$ (secs)	$B = \sum_{i=1}^{M-1} r_i$ (secs)	$A : B$
1K	0.17	5.42	0.03
5K	1.05	23.55	0.04
10K	9.43	65.85	0.14
50K	192.69	601.31	0.32

small search sizes, overheads of using the MapReduce implementation account for much of the low values). For the synthetic data, as size increases, r_M—the time for the slowest mapper—progressively begins to take disproportinately more time (see Fig. 18). Recall from Sect. 3.1 that this situation results in progressively lower speedups. Why does this happen? It appears that there is one very "difficult" data instance in the synthetic dataset that entails a hypothesis space whose complex clauses are very complex indeed. Of course, this kind of situation could well arise with a real dataset, and is indicative of the complexities that can arise in an ILP problem.

How do these results compare with prior work? We comment first on work in parallel ILP. For data parallelism no benefits were observed in Fonseca et al. (2008), but we have to proceed with caution here. Performance benefits reported in Fonseca et al. (2008) for data parallelism were obtained differently to the method used in this paper. There, speedup is computed over the time to identify an entire theory, using a randomised greedy procedure, with and without parallel coverage computation. Speedups reported here are for identifying a single clause in such a theory. Further, we have employed a range of synthetic and real data for our estimates, while results in Fonseca et al. (2008) were on some well-known real datasets. For some kinds of imbalanced problems, it is quite possible that speedups are observed for identifying most of the clauses in a theory—as we have observed here—but not for the theory overall (as was observed in Fonseca et al. 2008). The time to identify the very last clause, for example, may dominate the total time. To this extent, the results here are not inconsistent with Fonseca et al. (2008). The findings here do however suggest the possibility that the negative results in Fonseca et al. (2008) might have been an artifact of dataset size. That is, had

Dataset	Size (MB)	Speedup		Ratio (C1:C2)
		C1	C2	
100K	100	0.82	0.24	3.42
500K	500	3.97	0.88	4.51
1M	1000	5.88	1.56	3.77
5M	5000	13.21	2.27	5.82

Dataset	Size (nodes)	Speedup		Ratio (C1:C2)
		C1	C2	
1K	4×10^4	0.03	0.04	0.75
5K	2×10^5	0.20	0.20	1.00
10K	4×10^5	0.59	0.69	0.86
50K	2×10^6	2.98	2.92	1.02
100K	4×10^6	2.98	3.01	0.99

Fig. 19 Performance differences between the clusters used. Here "C1" stands for Cluster1 and "C2" stands for Cluster2. The speedups are obtained utilising 16 processors (correctly, cores), and the synthetic data used generated for (**a**) data (*left*); and (**b**) task parallelism (*right*)

the experiments in Fonseca et al. (2008) been conducted on significantly larger sized datasets—such as the ones we have used here—then benefits might have become evident. Performance benefits for task parallelism observed here were also apparent in Fonseca et al. (2008). It is difficult to state categorically whether the overheads on search size are the same in both cases, since no controlled experiments were performed in Fonseca et al. (2008). At any rate, we can state that for significantly large search spaces, the MapReduce approach and that using the MPI protocol show the same qualitative behaviour.

Comparing against work in using the MapReduce approach with propositional learning, we find that the linear scale-up observed in Chu et al. (2006) is reflected here only after some minimal data size. This initial overhead is either absent or not observed in propositional machine learning. We have already mentioned that we are not aware of any attempts at using MapReduce implementations to accomplish task parallelism in propositional learning.

The results from the synthetic data serve as the basis of what we can expect when using the ILP-MapReduce hybrid with real-life data, based on the sizes of data and search spaces. We recall first that, for logistic reasons, we conduct experiments on real data on a different cluster (Cluster2) to that used for synthetic data (Cluster1). We first provide an estimate of the difference in speeds for the different clusters using the synthetic datasets (Fig. 19). These results suggest that Cluster2 is somewhere between 3 to 6 times slower than Cluster1 for data parallelism, and approximately the same speed as Cluster1 for task parallelism.[11] Further, speedups initially increase for task parallelism, and then saturate for either cluster around 3 (probably for reasons explained above, namely, that the time taken by the slowest mapper begins to dominate).

We are now in a position to use the results obtained from synthetic data to make rough predictions on what can be expected with real data (Fig. 20) and the corresponding results are in Fig. 21. It is evident that the observed values are within the range we expect.

Finally, we turn to some issues that are of practical relevance:

Generality of the approach. We have already enumerated earlier the three principal reasons for our choice of Aleph (see Sect. 5.2 on "Algorithms and Machines"). Concerning gen-

[11]The difference in performance between the two clusters is largely due to differences in overheads in communication between mappers (communication overheads for Cluster2 are substantially higher). This affects performance for data parallelism much more than for task parallelism: in the former there is substantial communication between mappers as the performance estimates of a clause are the result of the computation performed by multiple mappers.

Dataset	Size (MB)	Speedup expected
HIV	45	0.07–0.16
Yeast	800	0.85–1.71
Zinc	7000	> 2.2

Dataset	Size (nodes)	Speedup expected
Mut(188)	$125–7.8 \times 10^8$	$\approx 0–3.0$
Carc	$182–1.6 \times 10^9$	$\approx 0–3.0$
DSSTox	$194–2.4 \times 10^9$	$\approx 0–3.0$

Fig. 20 Speedup expected on real world datasets using MapReduce on Cluster2 for (**a**) data (*left*); and (**b**) task parallelism (*right*)

Fig. 21 Speedup obtained on real world datasets using MapReduce on Cluster2 for (**a**) data (*left*); and (**b**) task parallelism (*right*)

Dataset	Speedup
HIV	0.176
Yeast	1.40
Zinc	2.87

Dataset	Speedup
Mut(188)	2.11
Carc	2.16
DSSTox	2.44

erality, there are two questions that arise: (1) How Aleph-specific is the ILP-MapReduce engine used here?; and (2) Can the MapReduce approach be used for other data and task-parallel computations that arise in ILP? The implementation we have used deliberately couples the ILP engine to the MapReduce implementation very loosely: communication is through the (admittedly slow) mechanism of text files. Neither Aleph nor the use of text-files as the method of communication is necessary: any ILP engine that can send and receive information shown diagrammatically in Figs. 6 and 9 can be used. The specific aspects we have elected to parallelise—clause evaluation and greedy clause-set identification—are at the heart of many ILP engines, but not the only kinds of computation for which a MapReduce approach can be used. Appendix shows how the MapReduce techniques can be used to implement other kinds of data and task parallel computations in ILP.

ILP-specific load balancing. The MapReduce implementation only performs one kind of load-balancing: a task that is running slowly on a processor (correctly, a mapper) may be stopped and re-started on a faster processor, if one becomes available. As such, this does not "solve" ILP-specific imbalances that may arise in data or task parallelism. It is quite possible, for example, that checking coverage of just a single example could prove to be extremely time-consuming (see Botta et al. 1999), and skew the time taken to compute the overall coverage of a clause. As we have described it, the MapReduce implementation may progressively allocate this computation to faster processors, it will not automatically perform any optimisations of the kind described in Costa et al. (2003), or that may be done by an RDBMS. The essential reasons for the imbalance will therefore remain (similar arguments hold for task-parallelism). There is, of course, nothing preventing us from exploring the use of a MapReduce engine designed to address this issue better. For example, each mapper could be provided a subset of literals in a clause, and compute the sets of examples covered and not covered. In this case, "slower subsets" would get assigned to a faster processors.

6 Concluding remarks

We recall the principal question with which we embarked on this paper: is there a single, simple approach that can be used by any ILP system to perform effective data and task parallelisation? To this we are now able to give a qualified answer concerning the widely-used

MapReduce approach. It is a uniform approach that is certainly simple, and can be used by any ILP engine with practically no significant implementation effort. The qualifications arise when examining its effectiveness. Our results suggest that it will provide an effective form of parallelisation when: (a) problem sizes (for example, data or search-space size) are very large; and (b) dynamic overheads can be minimised. The latter costs are not evident from descriptions of the MapReduce approach available in the literature, and will increase the implementation burden on developing an ILP engine that can properly utilise the parallelisation technique. If these two conditions can be met, then there are significant benefits of using a MapReduce engine like Hadoop, which removes several other complexities associated with distributed computing (like fault-tolerance, redundancy, load-balancing and so on). MapReduce is not, however, a panacea for all forms of parallelisation, nor is it an alternative to a full-scale relational database system. The limitations of the approach have been described in detail in Pavlo et al. (2009). Nevertheless, for the easy scaled-up ILP engines, the approach seems to be acceptable. Even here, we need to add a caveat: our results with synthetic data suggest that for ILP, the framework is better suited for data parallelism, rather than task parallelism.

There are a number of ways in which the work here can be extended. First, nothing we have proposed here is restricted to the specific ILP engine we have selected (Aleph). It is clearly of interest to investigate improvements in performance with other ILP engines. Communication between the ILP and MapReduce engines can also be improved from the rudimentary approach we have employed here (that of text files). Secondly, at various points in the paper, we have compared results we have obtained against those obtained with an implementation with the same ILP engine using the MPI protocol. In fact, this kind of comparison is not completely appropriate. Recent research has shown how the MapReduce approach can be implemented using corresponding functions in the MPI protocol (Hoefler et al. 2009). Our principal goal here is not to advocate the specific MapReduce implementation we have used here: rather it is to emphasise that several kinds of parallelism can be achieved by definitions of two simple functions. Provided these functions can be implemented without any user-involvement, we are relatively unconcerned by the underlying implementation—it may well be the case that an MPI-based implementation of MapReduce may be more efficient than the Hadoop-based one we have used here, and implementations such as those proposed in Miceli et al. (2009) would be better suited for scaling-up ILP for problems where data and background are even geographically distributed. Thirdly, although we have chosen to study data- and task-parallelism in isolation here, it is clearly of interest to combine them. This can be done in a straightforward manner by setting up dependencies amongst jobs (the MapReduce implementation we have used allows this). Finally, we have not used any form of sampling here. It is quite likely that substantial gains would be possible by incorporating data reduction by sampling within an ILP-MapReduce engine.

We turn now to the broader issue addressed in this paper, namely, the use of ILP as a tool for data analysis in an era where both domain-specific data and background information are increasing at a rapid rate; and multi-processor machines are becoming the norm. In such a setting, the emergence of ILP engines that exploit the concurrency within their procedures would appear to be a natural development. Yet, this has not been the case: implementations continue to sequential programs handling modest-sized problems. We believe that a point will soon be reached where large-scale multi-processor implementations will become a necessity if ILP is to fulfil its potential as one of the most powerful automated methods available for scientific and industrial data analysis. Investigations such as the one in this paper are aimed at preventing the loss of kingdom for the want of a nail.

Acknowledgements During the course of this work A.S. was supported by a Ramanujan Fellowship, Department of Science and Technology, Government of India; and was a Visiting Scientist at IBM Research, India. The work here originated from some discussions with Indrajit Bhattacharya.

Appendix: Some other ways of using MapReduce in ILP

A.1 Clause-set identification using subsets of data

There have been attempts reported in the ILP literature (Fonseca et al. 2008; Wang and Skillicorn 2000) of clause-set identification by constructing (intermediate) clause-sets on subsets of the examples (the subsets need not necessarily be disjoint), and constructing a single final set of clauses from these intermediate clause-sets (this is called "data parallelism" in Fonseca et al. (2008)). It is straightforward to formulate this within the MapReduce framework: individual mappers are assigned the task of identifying an intermediate clause-sets using a subsets of examples. The reducer then combines the intermediate clause-sets. We illustrate this with a k-partition $E_1^+, E_2^+, \ldots, E_k^+$ of the positive examples E^+ (that is, $\bigcup E_i^* = E^+$ and $E_i^+ \cap E_j^+ = \emptyset$ for $i \neq j$). We assume the presence of a helper function *splitdata* that creates a list of k dictionary items $(\alpha, E_1), (\alpha, E_2), \ldots, (\alpha, E_k)$ where $E_i = E_i^+ \cup E^-$ and α is some constant. The map and reduce functions are as follows:

The map function g. When applied to a dictionary item (α, E_i), the map function returns the dictionary item (α, H_i), where H_i is the intermediate clause-set identified by an ILP engine given the subset E_i (along with background knowledge B and cost function f which is taken to be the same for all mappers).

The reduce function ϕ. The reduce function is applied to dictionary entries of the form $(\alpha, [H_1, H_2, \ldots, H_k])$. We will take this to be the iterated application of the binary operation ϕ, defined as $\phi(H_i, H_j) = H_i \cup H_j$. That is, the final clause-set is simply the union of all the intermediate clause-sets. Clearly ϕ is associative with $\emptyset$ as an identity element.

A.2 Clause-set identification using multiple random restarts

In the paper we examined the parallel construction of clauses on each iteration of the greedy procedure in Fig. 2. There are other ways to identify clause-sets: in Zelezny et al. (2002), it was shown that a performing several randomised searches was usually better than a single deterministic search. In addition, in Fonseca et al. (2008), it was shown that conducting these randomised searches in parallel was more effective than conducting them sequentially. Here, we consider identifying of k iterations of a procedure, each of which consists of r randomised searches conducted in parallel, using e_i examples selected from E. On an iteration i, the best clause h_i (that is, with the lowest cost) from these r parallel searches is selected, and the next iterate is commenced. We are able to use *mapreduce* function described in the paper to perform in parallel both the r randomised searches, and the k iterations. We assume the presence of a helper function *randomsample* that samples k examples from E and creates a list of $r \times k$ dictionary items $[(e_1, 1), (e_1, 2), \ldots, (e_1, r), \ldots, (e_k, 1), (e_k, 2), \ldots, (e_k, r)]$. The map and reduce functions are as follows:

The map function g. When applied to a dictionary item (e_i, j), the map function returns the dictionary item $(e_i, (f_j, h_j))$, where h_j is the best clause returned by the jth randomised restarted search using example e_i, background B and examples E (see Zelezny et al. 2002 for details). Since B, E and the cost function f are constant in all cases, we

will omit them in the specification of the map function for clarity. g will therefore be taken to be a function of e_i and j, with the understanding that B, E and f are contained in the definition. Thus, $g(e_i, j) = rrr(e_i, j, B, E)$ where rrr denotes the randomised search function described in Zelezny et al. (2002).

The reduce function ϕ. The reduce function is applied to dictionary entries of the form $(e_i, [(f_1, h_1), (f_2, h_2), \ldots, (f_r, h_r)])$. We will take this to be the iterated application of the binary operation ϕ, defined as follows:

$$\begin{aligned}\phi((f_i, h_i), (f_j, h_j)) &= (f_i, h_i) \quad \text{if } f_i < f_j \\ &= (f_j, h_j) \quad \text{otherwise.}\end{aligned}$$

Clearly ϕ is associative, and has a unique identity element $(\infty, \square)$ (where $\square$ denotes "false", and is taken by convention to be the only clause with infinite cost).

References

Appice, A., Ceci, M., Turi, A., & Malerba, D. (2010). A parallel, distributed algorithm for relational frequent pattern discovery from very large data sets. In *Intelligent data analysis*.

Blockeel, H., & De Raedt, L. (1998). Top-down induction of first-order logical decision trees. *Artificial Intelligence*, *101*, 285–297.

Botta, M., Giordana, A., Saitta, L., & Sebag, M. (1999). Relational learning: Hard problems and phase transitions. In *LNAI: Vol. 1792. Proceedings of the sixth congress AI*AI* (pp. 178–189). Berlin: Springer.

Camacho, R. (1994). Learning stage transition rules with Indlog. In S. Wrobel (Ed.), *Proceedings of the fourth international inductive logic programming workshop*. Gesellschaft für Mathematik und Datenverarbeitung MBH, GMD-Studien Nr. 237.

Cardoso, P. M., & Zaverucha, G. (2006). Comparative evaluation of approaches to scale up ilp. In *International conference on inductive logic programming* (pp. 37–39).

Chu, C. T., Kim, S. K., Lin, Y. A., Yu, Y., Bradski, G. R., Ng, A. Y., & Olukotun, K. (2006). Map-reduce for machine learning on multicore. In *NIPS* (pp. 281–288).

Chu, C.-T., Kim, S. K., Lin, Y.-A., Yu, Y., Bradski, G. R., Ng, A. Y., & Olukotun, K. (2006). Map-reduce for machine learning on multicore. In *NIPS* (pp. 281–288). Cambridge: MIT Press.

Clare, A., & King, R. D. (2003). Data mining the yeast genome in a lazy functional language. In *Proceedings of the 5th international symposium on practical aspects of declarative languages* (pp. 19–36).

Costa, V. S., Srinivasan, A., Camacho, R. C., 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*(8), 465–491 (Aug).

De Raedt, L., & Bruynooghe, M. (1991). Clint: a multistrategy interactive concept-learner and theory revision system. In *Proceedings of the 1st international workshop on multistrategy learning* (pp. 175–191).

Dean, J., & Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107–113.

Fonseca, N., Srinivasan, A., Silva, F., & Camacho, R. (2008). Parallel ILP for distributed-memory architectures. *Machine Learning Journal*. doi:10.1007/s10994-008-5094-2

Hoare, C. A. R. (1993). Programs are predicates. *ICOT Journal*, *38*, 2–15.

Hoefler, T., Lumsdaine, A., & Dongarra, J. (2009). Towards efficient mapreduce using mpi. In *Proceedings of the 16th European PVM/MPI users' group meeting on recent advances in parallel virtual machine and message passing interface* (pp. 240–249).

Irwin, J. J., & Shoichet, B. K. (2004). Zinc—a free database of commercially available compounds for virtual screening. *Journal of Chemical Information and Modeling*, *45*(1), 177–182.

Kearns, M. (1998). Efficient noise-tolerant learning from statistical queries. *Journal of the ACM*, *45*, 983–1006.

King, R. D., Muggleton, S. H., Srinivasan, A., & Sternberg, M. J. E. (1996). Structure-activity relationships derived by machine learning: the use of atoms and their bond connectivities to predict mutagenicity by inductive logic programming. In *Proc. of the national academy of sciences* (Vol. 93, pp. 438–442).

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.

Ho, E. K. Y., Knobbe, A., & Malik, R. (2005). Ilp 2005 challenge: the safarii mrdm environment. In *Proceedings late-breaking papers track ILP* (p. 2005).

Lämmel, R. (2007). Google's mapreduce programming model—revisited. *Science of Computer Programming*, *68*(3), 208–237.

Miceli, C., Miceli, M., Jha, S., Kaiser, H., & Merzky, A. (2009). Programming abstractions for data intensive computing on clouds and grids. In *Proceedings of the 2009 9th IEEE/ACM international symposium on cluster computing and the grid, CCGRID'09* (pp. 478–483).

Miller, G. A., Beckwith, R., Fellbaum, C., Gross, D., & Miller, K. J. (1990). Introduction to wordnet: an on-line lexical database. *International Journal of Lexicography*, *3*(4), 235–244.

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*, *13*, 245–286.

Muggleton, S. H., & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the first conference on algorithmic learning theory*. Tokyo: Ohmsha.

Muggleton, S. H., Almeida Santos, J. C., & Tamaddoni-Nezhad, A. (2008). Toplog: ILP using a logic program declarative bias. In *ICLP* (pp. 687–692).

Lavrač, N., Flach, P., & Zupan, B. (1999). Rule evaluation measures: a unifying view. In *Proceedings of ninth international workshop on ILP* (Vol. 1634, pp. 174–185). Berlin: Springer.

Naish, L., & Sterling, L. (2000). Stepwise enhancement and higher-order programming in prolog. *Journal of Functional and Logic Programming*, 4. MIT Press, March.

Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S., & Stonebraker, M. (2009). A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on management of data, SIGMOD'09* (pp. 165–178).

Plotkin, G. D. (1971). *Automatic methods of inductive inference*. Ph.D. thesis, Edinburgh University, August.

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

Shi, Y. (1996). Re-evaluating Amdahl's law and Gustafson's law. Available at http://www.cis.temple.edu/shi/docs/amdahl/amdahl.html.

Specia, L., Srinivasan, A., Ramakrishnan, G., & Nunes, M. (2007). Word sense disambiguation with ILP. In *LNAI: Vol. 4455. Proceedings of the sixteenth international conference on inductive logic programming* (pp. 409–423).

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. (1999). The Aleph manual. Available at http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.

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, Oxford.

Srinivasan, A., & Kothari, R. (2005). A study of applying dimensionality reduction to restrict the size of a hypothesis space. In *LNAI: Vol. 3625. Proceedings of the fifteenth international conference on inductive logic programming (ILP2005)* (pp. 348–365). Berlin: Springer.

Fonseca, N. A., Camacho, R., Rocha, R., & Costa, V. S. (2008). Compile the hypothesis space: do it once, use it often. *Fundamenta Informaticae*, *89*, 45–67 Special issue on multi-relational data mining.

Wang, Y., & Skillicorn, D. (2000). Parallel inductive logic for data mining. In *Proceedings on distributed and parallel knowledge discovery, KDD2000*. New York: ACM.

Zelezny, F., Srinivasan, A., & Page, C. D. (2002). Lattice-search runtime distributions may be heavy-tailed. In *LNAI. Proceedings of the twelfth international conference on inductive logic programming (ILP2002)*.