Prolog Programming with a Map-Reduce Parallel Construct

Joana Côrte-Real
CRACS & INESC TEC
University of Porto
Rua Campo Alegre 1021/1055
4169-007 Porto, Portugal
jcr@dcc.fc.up.pt

Inês Dutra
CRACS & INESC TEC
University of Porto
Rua Campo Alegre 1021/1055
4169-007 Porto, Portugal
ines@dcc.fc.up.pt

Ricardo Rocha
CRACS & INESC TEC
University of Porto
Rua Campo Alegre 1021/1055
4169-007 Porto, Portugal
ricroc@dcc.fc.up.pt

ABSTRACT

Map-Reduce is a programming model that has its roots in early functional programming. In addition to producing short and elegant code for problems involving lists or collections, this model has proven very useful for large-scale highly parallel data processing. In this work, we present the design and implementation of a high-level parallel construct that makes the Map-Reduce programming model available for Prolog programmers. To the best of our knowledge, there is no Map-Reduce framework native to Prolog, and so the aim of this work is to offer data processing features from which several applications can greatly benefit; the Inductive Logic Programming field, for instance, can take advantage of a Map-Reduce predicate when proving newly created rules against sets of examples. Our Map-Reduce model was comprehensively tested with different applications. Our experiments, using the Yap Prolog system, show that: (i) the model scales linearly up to 24 processors; (ii) a dynamic distributed scheduling strategy performs better than centralized or static scheduling strategies; and (iii) the performance varies significantly with the number of items being sent to each processor at a time. Overall, our Map-Reduce framework presents as a good alternative for both taking advantage of the currently available low cost multi-core architectures and developing scalable data processing applications, native to the Prolog programming language.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming

General Terms

Design, Languages, Performance

Keywords

Map-Reduce, Logic Programming, Implementation

PPDP '13, September 16 - 18 2013, Madrid, Spain. Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2154-9/13/09 ...$15.00.
http://dx.doi.org/10.1145/2505879.2505884.

1. INTRODUCTION

Logic programming (LP) languages, such as Prolog, have been widely used to develop sophisticated, complex applications in diverse areas such as Natural Language Processing, Machine Learning, Deductive Databases, Software Engineering, Program Analysis, among others. Despite the power, flexibility and good performance that LP languages has achieved, one major criticism is that they allow rapid prototyping on small and medium sized problems, but do not scale up to computationally demanding real-world applications, as they often address a variety of complicated issues and manipulate large sets of data.

One possible way of overcoming this problem is to take advantage of the intrinsic parallelism available in LP applications. Many parallel LP systems implemented for shared and distributed memory architectures exist in the literature [5], but most of them are no longer available, maintained or supported. The success of these systems was mainly driven by the fact that parallelism was exploited *implicitly.* However, the lack of control over some of the main factors that often limit performance in parallel systems restricted the interest and applicability of these systems to real-world applications. Our past experience with Prolog applications has shown that, very often, most of the execution time is spent in performing computations that are inherently parallel and independent, and only a small part of the execution time is spent in sequential parts of code. These sequential parts usually correspond to initialization code, code to partitioning the computational activities and/or data into small sub-tasks, and code to aggregate/reduce data from different sub-tasks.

One alternative to address these performance limitations is to delegate the control of parallelism to the user. However, this solution is not elegant, since the user should be allowed to focus on the declarative nature of LP. There are high-level abstract and declarative constructs intrinsically parallelizable which can be used by Prolog programmers and at the same time hide details of parallelization. One such construct is Map-Reduce. Map-Reduce is a programming model dating back to early functional programming and it is designed to transparently manipulate sets or collections of data. One very popular implementation is Google's MapReduce [3], which focuses on processing large amounts of data files in parallel.

The relevance of the Map-Reduce programming model lies in the fact that the map and reduce operations are suitable for expressing a number of classic processing algorithms under a *summation form.* This form allows for

```
map_operation(key, value) -> (key, mapped_value) {
   mapped_value = perform_map_operation(value);
}

reduce_operation(key, set_of(mapped_value)) -> (key, reduced_value) {
   reduced_value = perform_reduce_operation(set_of(mapped_value));
}

aux_aggregator(set_of(key, mapped_value)) -> set_of(key, set_of(mapped_value)) {
   for each key compute
      set_of(mapped_value) = aggregate_by_key(key, set_of(key, mapped_value));
}
```

Figure 1: Pseudo-code for the map and reduce operations

a direct conversion to map and reduce operations, and it has been shown that algorithms such as locally weighted linear regression, expectation maximization and neural networks, amongst others, can be applied successfully to a Map-Reduce framework [2]. The Map-Reduce model is, however, by no means limited to these algorithms, as many possible map and reduce operations can be defined. One needs only to ensure that the operations have no side-effects on data other than that being used in the operation. Furthermore, it is necessary to guarantee that the operations on the data are associative and commutative, so that they can be executed in parallel.

In this work we are interested in a high-level parallel construct that implements the Map-Reduce programming model for Prolog. To the best of our knowledge, there is no Map-Reduce framework native to Prolog. Map-Reduce is an intrinsically parallel model and our motivation lies in the need for a tool to transparently distribute sub-computations in Prolog. Rather than focusing on managing large amounts of data files, we focus on executing the data processing tasks in parallel. We believe this can contribute towards more and simpler data processing support in Prolog, and find it particularly relevant at an age when multi-core processors are increasingly common and inexpensive. Experimental results, using the Yap Prolog system [10], show that our Map-Reduce model can effectively reduce the execution time and scale well up to 24 processors, for a number of different applications, proving itself as a good alternative for taking advantage of the currently available low cost multi-core architectures.

The remainder of the paper is organized as follows. First, we introduce some background about the Map-Reduce programming model and discuss related work. Then, we introduce our high-level Map-Reduce parallel construct for Prolog and describe the most relevant implementation details. Finally, we discuss experimental results and we end by outlining some conclusions.

2. BACKGROUND

The most popular implementation of distributed Map-Reduce is Google's MapReduce [3], developed in the early 2000's and aimed at processing large amounts of data stored in disk. As in the original functional model, it is composed of two elementary operations: the *map* and the *reduce*. The map operation applies a transformation to a set of key/value pairs, resulting in another set of the same size consisting of pairs with the same key but with a *mapped value*. The reduce operation groups all the mapped pairs with the same key and aggregates their values, usually into one – or no – result. Note that the original model of the map and reduce operations is not restricted to a set of key/value pairs. The pseudo-code in Fig. 1 illustrates the map and reduce operations. The *aux_aggregator()* operation aggregates the mapped data by key, in order to ensure that the reduce operation is run with data of the same key only. It was not explicitly mentioned earlier because it is independent of both the data being processed and the map and reduce operations, rendering it autonomous from the remaining program. This operation is merely auxiliary when several different data types are processed simultaneously, and it it is not necessary if all the data in the program concern the same call.

Figure 2 illustrates a very simple Map-Reduce example using a set of squares, triangles and circles, either black or white. The colour of the shapes represents their *key* and the shape itself is the *value*. The mapping process first transforms each shape into the first letter of its name, thus mapping a square to an S, a triangle to a T, and so on. The mapped values are then aggregated by colour, corresponding to the *aux_aggregator()* operation. Finally, the reduce function counts how many T's there are for each group. In the example, the final result is found to be two white T's and one black T.

3. RELATED WORK

Srinivasan *et al.* [13] introduce an approach combining the Prolog inductive logic programming system Aleph [12] and the Hadoop's MapReduce framework [11]. Their aim was to investigate whether Prolog Aleph's engine could be applicable to very large datasets, since the amount of data available for processing has become so large that it could not fit into one machine's memory, and Hadoop's MapReduce was the selected framework for this task. The approach used in this work consisted of two distinct engines, one for running Aleph and the other for running the actual Map-Reduce operations using the Hadoop framework.

Two different sets of map and reduce functions were developed for the combined system, with different aims. The first set was meant to distribute the background knowledge across the Map-Reduce cluster, so as to ensure that the second set of functions – which actually perform the calculation for the given examples – had all the necessary clauses to be

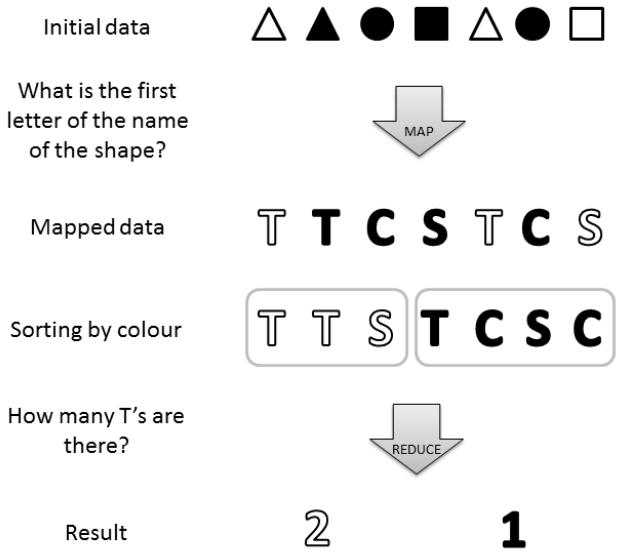


Figure 2: A Map-Reduce example

able to use a greedy algorithm [13]. The Map-Reduce and Aleph engines communicate and the latter transforms examples not yet covered in Map-Reduce queries. When the last reduce operation finishes, the minimum cost clause determined is then returned back to the Aleph engine.

To evaluate the system, the authors have used both synthetic and real-world datasets, with sizes ranging from tens of thousands up to millions, and their results demonstrated that the MapReduce Hadoop framework can be efficiently applied in this context. Still, the size of the dataset must be significant, greater than 500,000, in order to obtain some speed-up. Also, the speed-ups are not nearly linear until datasets of size 5 million, and for datasets smaller than 500,000 the data processing time actually increases when compared with the sequential processing time due to the cost of data communication and disk access in the cluster, amongst other factors.

Orthogonally, in [14] Wielemaker presents and discusses a case study also concerning ILP and the Aleph system, and their parallelization. This case study consisted of performing multi-threaded runs of a randomised local search algorithm – used in the rule-inference process by the Aleph system – and varying the number of threads. Even though the data and task allocation process is not explicitly presented as a Map-Reduce operation, it could easily be transformed into one. The results of this experiment show that the approach obtained under linear speed-ups, reaching a maximum acceleration of approximately 7.5 for 16 threads.

4. MAP-REDUCE FOR PROLOG

In this section we present our high-level Map-Reduce parallel construct for Prolog and discuss the most relevant implementation details.

4.1 Architecture

Most Map-Reduce frameworks described in the literature, if not all, use a master-slave paradigm [3, 4, 6, 7, 8, 9], with the purpose of reducing data processing times by taking advantage of the available computational resources. Given the sometimes huge size of the clusters in which Map-Reduce frameworks are applied, they must be highly fault-tolerant and robust. Amongst other precautions mentioned in the literature, the master node is usually responsible for pinging the slave nodes, as well as backing up the processed data and re-scheduling work in case of slave failure.

Our model's architecture is loosely based on the architecture described in [3] in the sense that it supports clusters of machines and a master-slave paradigm, but it innovates by taking advantage of the parallelism within each machine. We also do not focus on distributed data files. Figure 3 shows how our framework can apply to a generic distributed architecture.

There are three hierarchical levels in this architecture: the *Global Master* (GM), the *Local Masters* (LMs) and the *Slaves* (SLs). The GM controls the flow of communications and first-level scheduling, dispatching data to the LMs. There are as many LMs as machines in the cluster and each LM is responsible for local data scheduling and dispatching among the SLs running on that machine. The SLs execute both map and reduce predicates on their data and return the reduced value to the respective LM. Each LM then performs a reduce operation on all its SLs' reduced values, and similarly the GM executes the last reduce operation of the call. This architecture applies to distributed memory systems composed of multi-core machines.

For shared memory architectures (SMA), our Map-Reduce for Prolog uses multi-threading while for distributed memory architectures (DMA), it uses MPI [1]. In the SMA implementation, the first thread – LM0 – starts as many threads as the number of machine cores. Each thread runs a slave interface, which waits for thread messages from LM0 and carries out the work. In the DMA implementation, processes are started for each machine core or for each distributed computer node. The SLs can be thought of as resources that LMs manage according to different scheduling methods; the SLs do not keep track of how many operations they have executed, and they do not self terminate. Instead, LMs are responsible for their creation, task assignment and termination.

The system requires a set-up time, in which each LM loads any files that may have been requested by the user, so as to have the necessary information to carry out queries. This information is named *background knowledge*; in the case of different LMs, each one can have its own background knowledge. The set-up time is only spent once for each LM and each background knowledge file requested, for the SMA implementation. For the DMA implementation, files need to be read by all LMs. Since the data files are only loaded on LMs during the initialization of the program, this model allows for no communication overheads during runtime. Note that the user is responsible for having a copy of the program source code in each machine, as well as the map and reduce predicates and any other data required to complete the queries.

The Map-Reduce predicates are user-defined but follow a specific pre-defined signature. The map predicate has two arguments, the first being an element from the list of values to be mapped and the second the *mapped result*. The reduce predicate also has two arguments, the first being a list of Prolog terms to be reduced and the second the *reduced result*.

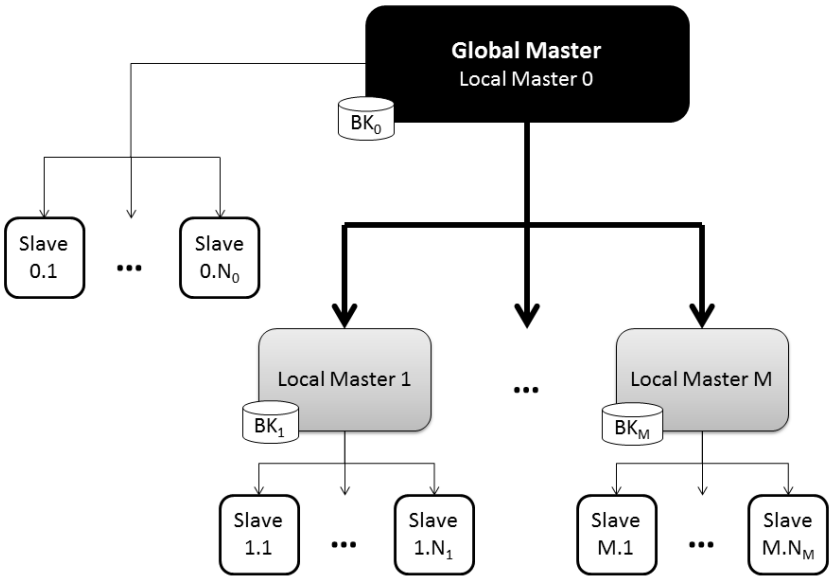


Figure 3: Framework architecture

Each Map-Reduce call receives as arguments the names of predicates to be used to map and reduce data. As such, the user can specify several different predicates and use them indiscriminately in different Map-Reduce calls without having to re-initialize the system. The Map-Reduce predicate also requires a data array as input. This array can be created by the user, or it can be loaded from a file automatically. Our framework includes predicates capable of creating an array of data from a given file. The positions in the array contain the respective line of the file, in the form of a generic Prolog term. We consider this to be a flexible approach, since the user can use data from any other source he/she requires, as long as he/she makes it available to the system under this structure.

One of the main goals of this implementation is to provide a flexible system, which supports both heavy computations across several machines and lighter iterative runs of Map-Reduce possibly executing on one machine alone. We have designed a transparent architecture divided in three functional modules as follows:

Initializer Creates a communication grid encompassing the LMs and the SLs, and loads the data for each LM.

Map-Reduce This module is composed of the master and the slave files. Only one of these files is used at any given time, according to the entity's hierarchical level. The slave version executes the map and reduce predicates, while the master version performs reduce operations and implements communication protocols.

Terminator Terminates the communication grid created by the Initializer and frees the allocated memory.

Additionally, user-defined files are required in order to specify the several map and reduce predicates to be used. The fact that this information is specified as Prolog predicates allows the user to easily reconfigure them – including system architecture and map and reduce predicates; it is also possible to run distinct Map-Reduce calls simultaneously.

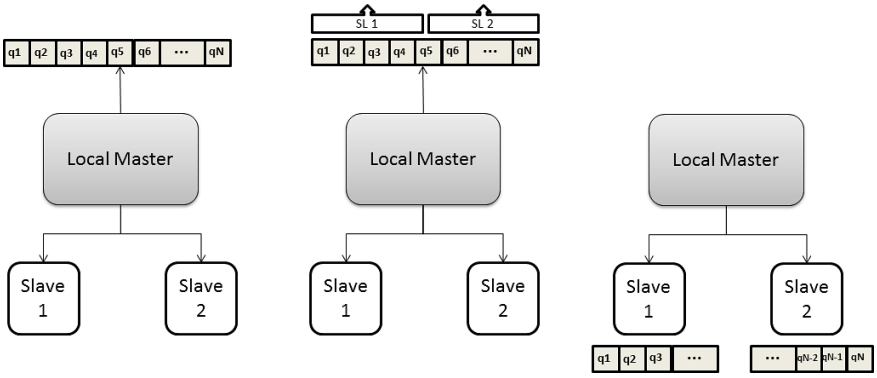
4.2 Scheduling Methods

Most parallel and distributed Map-Reduce systems are not very concerned with the efficiency of the scheduling strategies, rather with their redundancy and fault-tolerance strategies. Conversely, and since Map-Reduce for Prolog is an implementation for more modest computing capabilities, we concern ourselves with the speed-up that this construct achieves, when compared to executing the Map-Reduce call sequentially. It is then crucial to have a scheduling method which allows for good performance in parallel, and bearing this in mind we developed four scheduling methods: (i) *single-step scheduling*; (ii) *static scheduling*; (iii) *dynamic scheduling* and (iv) *workpool scheduling*.

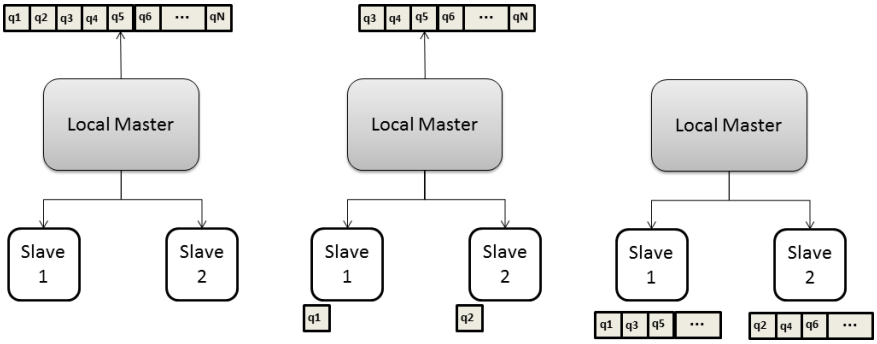
Figure 4 depicts the interaction between LMs and SLs for each type of scheduling. This interaction can obviously extrapolate to GMs and LMs, respectively. All figures depict three stages of the scheduling algorithm, temporally from left to right, and the explanatory text is presented below.

Single-step scheduling is used as a base case. It takes the total number of items and distributes them evenly across slaves in just one step. One block of items goes to one slave, another to the second slave and so on, ensuring every SL is assigned the same number of queries, approximately. In stage two of Fig. 4(a), the method of dividing data is depicted, and in stage three the division is completed.

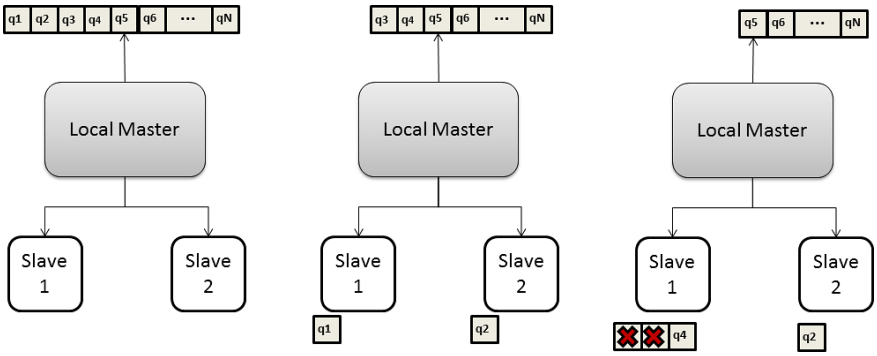
Static scheduling consists of dividing the M data items in *chunks* of N elements and distribute them in a round-robin fashion by all the slaves. It differs from single-



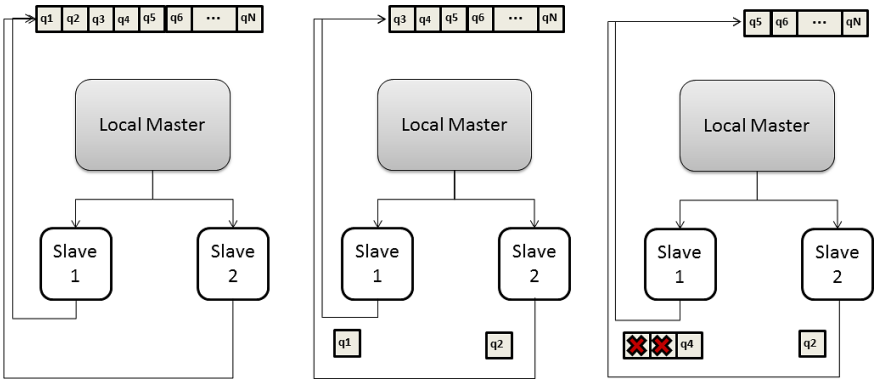
(a) Single-step scheduling



(b) Static scheduling



(c) Dynamic scheduling



(d) Workpool scheduling

Figure 4: Scheduling methods

```
init_communicator(-Comm).
init_communicator(-Comm,+NoCores).
end_communicator(+Comm).

data_from_file(+Filename,-DataArray).

map_reduce(+Comm,+MapPred,+ReducePred,+DataArray,-Result).
map_reduce(+Comm,+MapPred,+ReducePred,+DataArray,-Result,+Scheduling).
map_reduce(+Comm,+MapPred,+ReducePred,+DataArray,-Result,+Scheduling,+NoElements).

map(+Value,-MappedValue).
reduce(+ListOfValues,-ReducedValue).
```

Figure 5: High-level Prolog predicates for Map-Reduce

step scheduling because the queries are distributed in several small chunks, in turns. Figure 4(b) shows that it first attributes a chunk to each slave and from then all the data is distributed alternately by the slaves.

Dynamic scheduling is more adaptive than the previous methods, but also more demanding on the LM in terms of computation time. At first, it also attributes a chunk of data to each slave, in order, but then the LM waits for a reply from one of the SLs, informing that it is free. This algorithm behaves differently from static scheduling because, as shown on stage three of Fig. 4(c), the LM waits for a reply from one of the SLs. The LM then attributes further work to the free SL and waits again. Ultimately, and if the data granularity is low, the dynamic scheduling converges towards static scheduling, since all SLs take the same time to complete the same number of queries.

Workpool scheduling is similar to the dynamic one, but implements a pool of work that is consumed on demand of idle slaves. As depicted in Fig. 4(d), the SLs have access to a pool of work that is filled by the LM with chunks of data to be processed. The SLs remove one chunk of work when they are finished with their current one, until the pool is empty. The LM is not responsible for distributing the work between SLs, and this can be computationally less taxing on the LM entity. However, the access to the workpool is heavily competed for, and more so with a growing number of SLs.

4.3 User Interface

The Map-Reduce for Prolog user interface is composed of six predicates, as illustrated in Fig. 5.

To initialize the system, both `init_communicator/1` and `init_communicator/2` predicates can be used: if no `NoCores` argument is provided, the Map-Reduce for Prolog determines the number of cores in the machine and starts the corresponding number of slaves. The predicate then returns the slave's information in the `Comm` argument. The `end_communicator/1` predicate should be used to terminate the communication grid and free memory.

The `data_from_file/2` predicate can be used to consult a file and load its lines, as Prolog terms, into an array. The use of this predicate is optional, since the user may build an array from other sources and pass it as argument to the *map_reduce()* call. This predicate supports three levels of customization. The most basic form – `map_reduce/5` – uses the standard scheduling options. The `map_reduce/6` and `map_reduce/7` allow the user to select a scheduling method and the number of elements per chunk for that method, if applicable. These predicates can be called iteratively and with different map and reduce operations, and they return only the final result.

Finally, the `map/2` and `reduce/2` are not part of the interface *per se*, but they are included in the description for completeness and also because even though they are user-defined, their signature must match the one in Fig. 5. These predicates define the specific map and reduce operations and their names are passed as arguments to the `map_reduce/5` predicate. This allows for great flexibility, since the user can define several predicates prior to execution, as well as specify different behaviours according to the machine the predicates are running in, for instance.

Due to the MPI communication protocol usage, the interface differs between shared memory and distributed memory architectures. The predicates for the distributed memory version do not contain the `Comm` argument, since the program is run as an MPI executable, meaning that the communication grid must be configured in the MPI protocol, outside the Map-Reduce for Prolog interface. For distributed memory systems, it is assumed that the grid has been configured and is running, and that a copy of the relevant files has been placed in every machine in the cluster. It is also not possible to change the scheduling method to workpool, since the SLs behaviour is radically different from the one exhibited in the other three scheduling methods. Other than that, the interface is very similar in both cases, and the configuration options are common to both cases. Note that the user can abstract from the details of the parallel implementation and machine architecture as we provide interfaces with different levels of transparency.

An usage example is presented in Fig. 6. The `map1/2` predicate verifies whether a given query is true and the `reduce/2` predicate sums all the numbers in a list, which calculates how many queries are true for `map1/2`. The `map2/2` predicate is an example of a generic computation, and the purpose of that Map-Reduce call is to determine the number of odd numbers. This example is intended to be illustrative of a map operation native to Prolog, but there are many other possible applications for the simple but powerful framework we provide, such as run `map_reduce/5` calls in a

```
example(Result) :-
   init_communicator(Comm,8),
   data_from_file('queries.pl',MyArray),
   map_reduce(Comm,map1,reduce,MyArray,Result1),
   do_something(Result1,MyArray,MyNewArray),
   map_reduce(Comm,map2,reduce,MyNewArray,Result),
   end_communicator(Comm).

map1(Query,1) :- call(Query), !.
map1(_,0).

map2(N,Val) :- Val is N mod 2.

reduce([],0) :- !.
reduce([H|T],Val) :- reduce(T,AuxVal), Val is H+AuxVal.
```

Figure 6: Map-Reduce example

loop, or define map and reduce operations so as to apply the Naïve Bayes algorithm on a dataset, as described in [2], amongst others.

5. EXPERIMENTAL SETTINGS

Our testing environment consisted of two shared memory machines, used both independently and as a cluster. Their technical specifications are the same: four six-core AMD Opteron 8425 processors, 2.1 GHz (totalling 24 cores), with 64 GB RAM, 1.5 TB HD and running Red Hat Enterprise Linux in 64-bit mode.

For distributed memory architectures, our implementation uses MPI to communicate between machines. Because the Yap Prolog system currently does not support MPI and threads simultaneously, it is necessary to adapt the Map-Reduce construct in one of two ways: (i) use each machine as uni-core, and have it run one slave only; or (ii) when setting up the MPI grid, take into account the number of cores in each machine and start a slave for each core before runtime. In the experiments that follow, we used the latter approach. We are aware of the limitations this presents, both in usability and performance of the system. In particular, both adaptations remove the possibility of having two scheduling levels. Even though we have, obviously, no way to compare two-level scheduling with single scheduling at this time, we believe there could be a significant difference performance-wise.

Four datasets of different characteristics were selected to validate the Map-Reduce for Prolog implementation. Two of them are composed of data native to Prolog, as well as background knowledge files (data files specified by the user) which must be consulted during execution. The other two consist of integers, and simple operations are performed on them. Table 1 summarises this information.

We next describe the map and reduce operations applied to these datasets:

ODD the map operation verifies whether a number is odd and the reduce operation counts how many odd numbers there are in the dataset. Code implementing these operations can be found in Fig. 6.

PROB the map operation assigns a partition of the probabilistic space to an occurrence and the reduce operation counts the total number of occurrences in each partition. This can be used to calculate conditional probabilities so as to implement a step of a Bayesian network, for instance.

Table 1: Data type and background knowledge file size

Dataset	Data type	Background knowledge
ODD	Arithmetic	–
PROB	Probabilistic	–
MAMMO	Prolog facts	91.2 MB
BLOG	Prolog facts	1.5 GB

MAMMO/BLOG the map and reduce operations applied to these datasets are similar and the reduce operation is also the one reported in Fig. 6. The map operation verifies whether a term is true based on rules specified in the background knowledge files (which differ according to the dataset) and the reduce operation counts how many terms were covered by that rule.

6. EXPERIMENTAL RESULTS

Tests were run for both the shared memory and the distributed memory implementations, across the two machines in the cluster, using different numbers of queries (300,000, 600,000 or 1,200,000 queries were posed for each test). We also performed experiments with the four different scheduling strategies for a fixed number of queries (dataset size) and fixed number of items sent to each slave (chunk size). Experiments varying the dataset and chunk sizes were performed for 1, 2, 4, 8, 16 and 24 slaves.

6.1 Loading Initial Data Files

Table 2 contains the set-up time spent loading the queries files and the background knowledge, when applicable, for each dataset and query number. This time is only spent on the first run of the Map-Reduce for Prolog and it was recorded in seconds. In shared memory, the time of thread creating and termination is not taken into account, since it is negligible. For distributed memory, the termination time

is also negligible. Note that the set-up time for distributed memory is highly dependent on the number of running slaves and on the machines' hard drive: if the files being loaded are shared between several processes, the set-up time could be slightly increased.

Table 2: Set-up times (in seconds) for varying dataset sizes

Dataset	300,000	600,000	1,200,000
ODD	2.35	4.20	7.79
PROB	24.03	47.51	95.49
MAMMO	30.18	34.00	41.76
BLOG	377.47	381.90	387.11

6.2 Sequential Execution Times

Tables 3 and 4 show the overall time (*walltime*), in milliseconds, of a Map-Reduce call for each dataset. Note that the corresponding times between SMA and DMA vary significantly. This can be justified by the fact that MPI runs processes (and not threads), which are managed at kernel level, and thus more efficiently. Also, Yap's sequential version run in MPI processes uses simpler data structures than the multi-threaded version, making it significantly faster when loading data.

Table 3: Sequential execution times (in milliseconds) for SMA and varying dataset sizes

Dataset	300,000	600,000	1,200,000
ODD	240	485	956
PROB	479	968	2,016
MAMMO	1,238	2,194	4,623
BLOG	824	1,872	3,783

Table 4: Sequential execution times (in milliseconds) for DMA and varying dataset sizes

Dataset	300,000	600,000	1,200,000
ODD	226	453	905
PROB	376	733	1,447
MAMMO	707	1,413	2,829
BLOG	573	1,148	2,294

6.3 Varying Scheduling Strategies

Figure 7 plots the seven scheduling methods made available by Map-Reduce for Prolog for each dataset. The results presented here do not take into account the set-up times described in Table 2. The aim of these plots is to demonstrate the variation of the performance of the scheduling methods according to the type of data and also with the implementation used. The data used to plot these graphs was obtained by running five trials of each Map-Reduce call and calculating their average value. Finally, the data from dataset BLOG is incomplete in the distributed memory instances because memory constraints did not allow for running sixteen instances of this application on the cluster. Each machine of the cluster is equipped with 64GB RAM and each Yap instance requires about 7GB of RAM memory to load the background knowledge of this particular application. As such, it is possible to run 8 such processes on each machine, but 12 would require at least 84 GB RAM.

These results show that Map-Reduce for Prolog achieves nearly linear speed-ups, for both shared and distributed memory, and for all the different datasets tested. The distributed memory implementation has proved to be consistently faster than the shared memory one. This is to be expected since Yap is not yet finely tuned for thread support. In fact, this could explain the somewhat under achieving results for the dataset BLOG in shared memory. The BLOG dataset requires intensive use of the Yap atom table, whose synchronization is centralized. Since this table is shared between all slaves in a process, it can cause a significant overhead. In the future, it would be interesting to de-centralize access to shared data structures in Yap and assess the effect of that change in our approach's SMA execution times.

The excellent results of the DMA implementation are also partially due to the very low network traffic MapReduce for Prolog generates. Since data is loaded as an array, the work scheduled for a SL is described as an interval of that array. As such, the information *to* an SL is composed of two integers representing the array indices and the names of the map and reduce predicates, and the information *from* an SL is merely the reduce value of corresponding to that interval. In our particular setup, the two machines used as a cluster are physically linked to the same network bus, and so our results for distributed memory may not be representative of the most typical cluster architecture.

From Fig. 7, we can also observe that globally the most efficient scheduling methods are the workpool (SMA-POOL) and the dynamic scheduling (SMA-DYNAMIC or DMA-DYNAMIC). If the data's granularity was negligible, the dynamic algorithm would tend to static scheduling, with slightly worse performance due to the small wait caused by the master only sending work when the slave is already free. In the workpool strategy, the slaves are responsible for their own work management, thus making it even more efficient than the dynamic scheduling. However, and to ensure compatibility between both Map-Reduce for Prolog versions, we will adopt the dynamic scheduling method as the default strategy, since it displays the best behaviour for distributed memory and a close second for shared memory.

6.4 Load Balancing

In order to assess load balancing in the different scheduling methods, the CPU time of each slave was measured and plotted in Fig. 8. This test was run for 1.2 million queries and for sixteen slaves, with the exception of DMA-BLOG, in which case it was only possible to use eight slaves due to memory constraints. The y-axis of Fig. 8 denotes the maximum deviation between slaves, as a percentage of the average walltime of the respective run. As before, each Map-Reduce call was run 5 times and all values presented are calculated from the averages of those runs.

From Fig. 8 it becomes evident that static scheduling is generally more efficient for datasets PROB and ODD and dynamic scheduling for datasets MAMMO and PROB. This

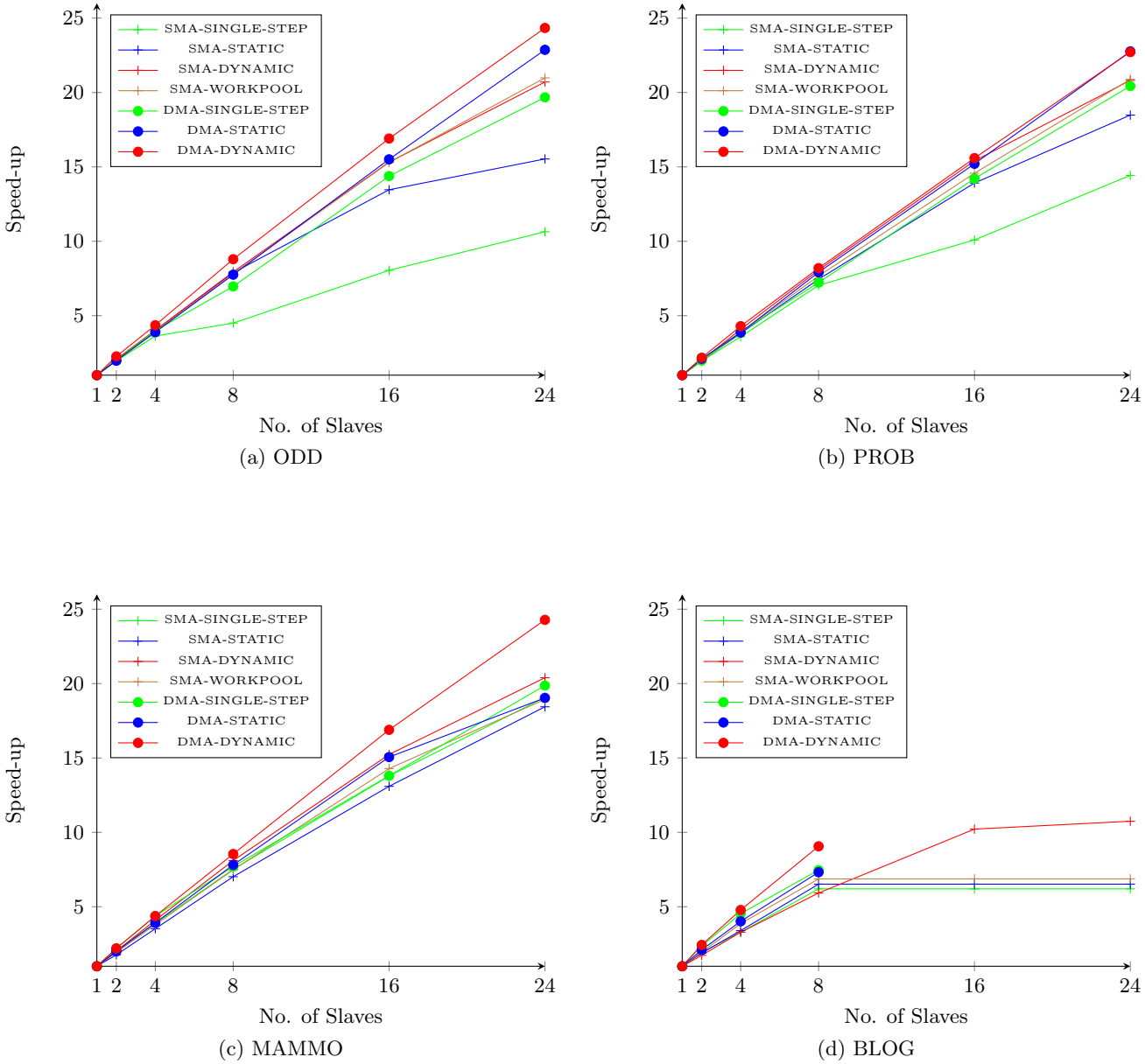


Figure 7: Comparison of scheduling methods (600,000 queries and 1,000 elements per chunk)

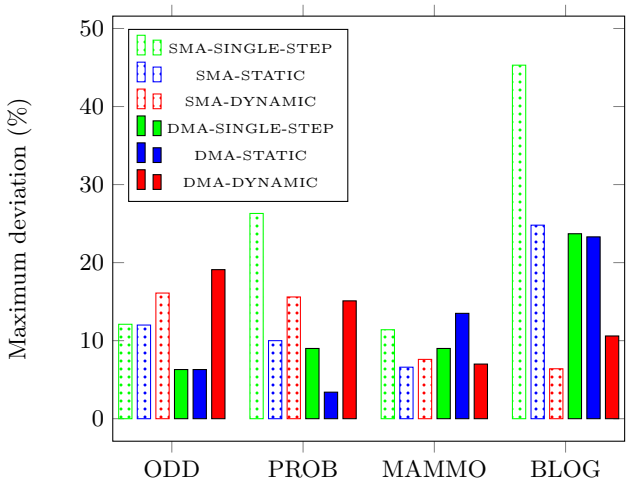


Figure 8: Load balancing for different scheduling methods (1,200,000 queries and 1,000 elements per chunk)

is caused by the data granularity of the datasets native to Prolog; queries can take variable times to succeed or fail, which can contribute to load imbalance. The fact that the SMA is consistently slower than DMA, and more so for single-step scheduling, can be justified by the fact that the communication between threads is slower than between MPI nodes due to synchronization issues in the Yap Prolog system; this would cause a significant detachment between the reception of the first data in each slave. This effect becomes more evident when the slaves are only processing a large block of data, at once.

6.5 Varying Chunk Sizes

Figure 9 depicts the effect of varying the size of the chunks in the two best performing scheduling methods. The time is given in milliseconds and it is an average of five consecutive and equal Map-Reduce runs.

For all four datasets used for testing, there appears to be an optimum number of queries to minimize execution time. In our methodology, when testing scheduling methods using chunks, we have used queries of size 1,000 precisely to obtain the fastest result possible when assessing other parameters. 1,000 elements per chunks is a somewhat empirical choice, however, because even though the curves all demonstrate a tendency towards a minimum around that point, it would require testing every single value to ensure that 1000 is in fact the best choice.

6.6 Varying Data Sizes

Figure 10 depicts the behaviour of dynamic scheduling, for each dataset, with varying queries size and 1,000 elements per chunk.

In general, these results show that DMA seems to be immune to variations on the dataset size. On the other hand, for SMA, these results show a generic tendency to obtain better speed-ups as we increase the dataset size and the number of slaves, which confirms the good scalability of our Map-Reduce for Prolog framework.

We believe all these tests consider and evaluate the most relevant features of Map-Reduce for Prolog. They demonstrate that our construct can scale efficiently, and that it can manage data with different granularity. We provide a flexible user interface, which allows for adapting the scheduling method to the data type, should the user wish to do so. The results are good for both shared and distributed memory implementations, making Map-Reduce for Prolog a flexible and agile Map-Reduce implementation for modest computing capabilities, whose focus is data native to Prolog.

7. CONCLUSIONS AND FURTHER WORK

A Map-Reduce parallel construct was designed and implemented in the Yap Prolog system. This construct provides an elegant way of implementing many applications in the summation form in Prolog, with the advantage of being intrinsically parallelizable. Two parallel implementations of the Map-Reduce are provided: a multithreaded and a message passing. In contrast to the Google's MapReduce implementation, whose focus is on distributed processing of data stored in disk, our implementation focuses on parallelization of the map and reduce operations where the data is already in memory.

We tested our implementation with four applications and evaluated how different scheduling strategies and chunk sizes can affect performance and concluded that: (i) our Map-Reduce construct can have linear speed-ups up to 24 processors; (ii) a dynamic distributed scheduling strategy, in general, performs better than centralized or static strategies; (iii) the performance varies significantly with the number of items being sent to each processor at a time; and (iv) our Map-Reduce model is a good alternative for taking advantage of the currently available low cost multi-core architectures.

One of the limitations of performance is related to the data synchronization used in the Yap implementation. Work is in progress to decentralize the access to data structures in order to further improve performance. We have also been studying best ways of executing Map-Reduce in the hybrid distributed shared-memory multi-core architectures.

Acknowledgments

This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within project LEAP (FCOMP-01-0124-FEDER-015008).

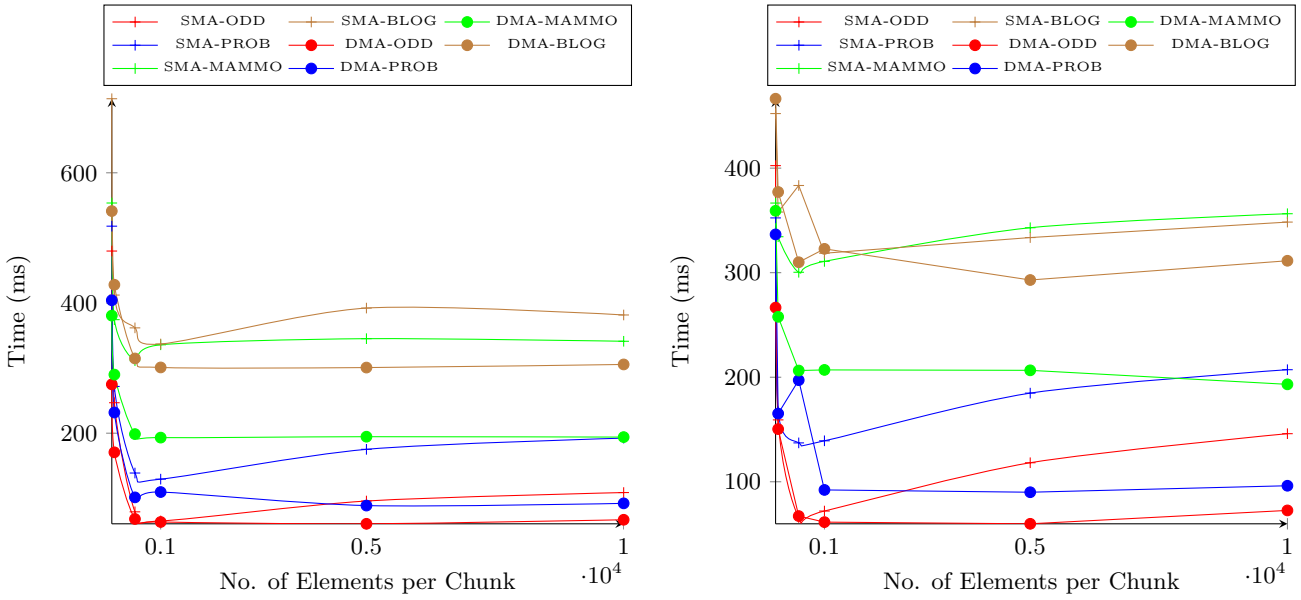
8. REFERENCES

[1] Message Passing Interface Forum.

[2] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for Machine Learning on Multicore. In *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, 2007.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: Twister: A Runtime for Iterative MapReduce. In *ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.



(a) Dynamic scheduling

(b) Static scheduling

Figure 9: Effect of chunk size variation (1,200,000 queries)

[5] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

[6] C. Miceli, M. Miceli, S. Jha, H. Kaiser, and A. Merzky. Programming Abstractions for Data Intensive Computing on Clouds and Grids. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 478–483. IEEE Computer Society, 2009.

[7] S. Pallickara, J. Ekanayake, and G. Fox. Granules: A Lightweight, Streaming Runtime for Cloud Computing with Support, for Map-Reduce. In *International Conference on Cluster Computing and Workshops*, pages 1–10. IEEE Computer Society, 2009.

[8] S. Papadimitriou and J. Sun. DisCo: Distributed Co-clustering with Map-Reduce: A Case Study Towards Petabyte-Scale End-to-End Mining. In *International Conference on Data Mining*, pages 512–521. IEEE Computer Society, 2008.

[9] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *ACM International Conference on the Management of Data*, pages 165–178. ACM, 2009.

[10] V. Santos Costa, R. Rocha, and L. Damas. The YAP Prolog System. *Journal of Theory and Practice of Logic Programming*, 12(1 & 2):5–34, 2012.

[11] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Symposium on Mass Storage Systems and Technologies*, pages 1–10. IEEE Computer Society, 2010.

[12] A. Srinivasan. *The Aleph Manual*, 2004.

[13] A. Srinivasan, T. A. Faruquie, and S. Joshi. Data and task parallelism in ILP using MapReduce. *Machine Learning*, 86(1):141–168, 2012.

[14] J. Wielemaker. Native Preemptive Threads in SWI-Prolog. In *International Conference on Logic Programming*, number 2916 in LNCS, pages 331–345. Springer-Verlag, 2003.

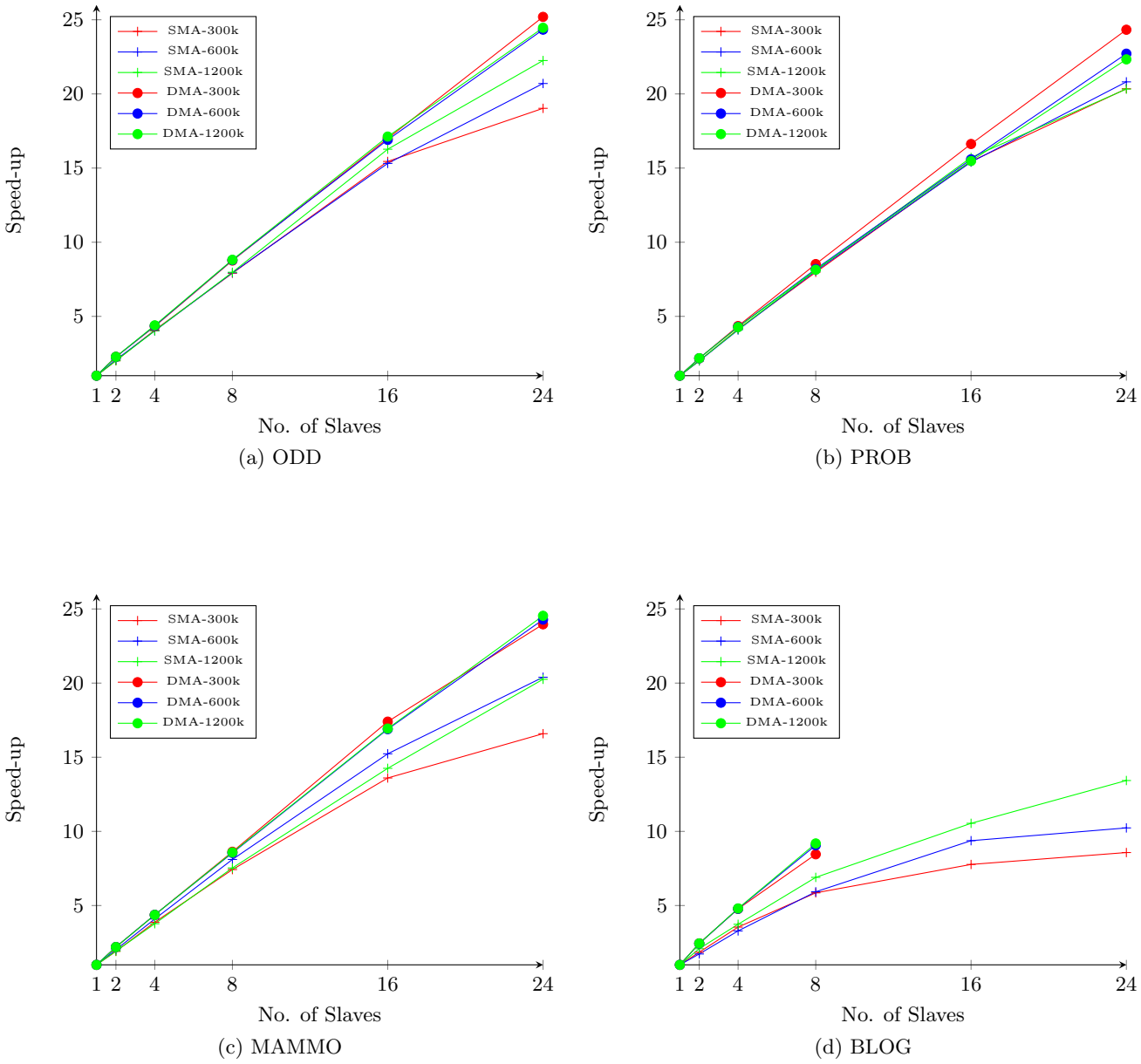


Figure 10: Effect of variation of queries size with dynamic scheduling (1,000 elements per chunk)