A Hybrid MapReduce Model for Prolog

Joana Côrte-Real, Inês Dutra, and Ricardo Rocha
CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{jcr, ines, ricroc}@dcc.fc.up.pt

***Abstract*—Interest in the MapReduce programming model has been rekindled by Google in the past 10 years; its popularity is mostly due to the convenient abstraction for parallelization details this framework provides. State-of-the-art systems such as Google's, Hadoop or SAGA often provide added features like a distributed file system, fault tolerance mechanisms, data redundancy and portability to the basic MapReduce framework. However, these features pose an additional overhead in terms of system performance. In this work, we present a MapReduce design for Prolog which can potentially take advantage of hybrid parallel environments; this combination allies the easy declarative syntax of logic programming with its suitability to represent and handle multi-relational data due to its first order logic basis. MapReduce for Prolog addresses efficiency issues by performing load balancing on data with different granularity and allowing for parallelization in shared memory, as well as across machines. In an era where multicore processors have become common, taking advantage of a cluster's full capabilities requires the hybrid use of parallelism.**

I. Introduction

MapReduce is a programming model developed by Google in the early 2000's [6] aimed at processing large amounts of data. As the name suggests, it is composed of two elementary operations: *map* and *reduce*, which are based on primitives originally introduced in functional programming languages such as Lisp. The MapReduce model was primarily aimed at application onto a large set of machines linked together - also known as a *cluster* - with the purpose of efficiently dividing data among disks in the cluster.

The relevance of the MapReduce 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* [4]. This form allows for a direct conversion to map and reduce operations, and it has been shown by [4] that algorithms such as locally weighted linear regression, expectation maximization and neural networks, amongst others, can be applied successfully to a MapReduce framework.

Whilst these algorithms can be useful, the MapReduce model is by no means limited to them, as many possible map and reduce operations can be defined. One needs only to ensure that the operations have no collateral 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 and thus benefit from the inherent speeding up of the process.

As such, 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 a *mapped* value. The reduce operation groups all the mapped pairs with the same key and aggregates their values to one - or no - result.

978-1-4799-4833-8/14/$31.00 © 2014 IEEE

There is an auxiliary aggregation operation used to separate data from different *keys* and it is independent of both the data being processed and the map and reduce operations, rendering it autonomous from the remaining program; this operation allows the user to run different kinds of data on the same MapReduce call and group them using a key. This feature is specific to Google's MapReduce implementation and it is not included in all implementations presented in the literature because the size of the available resources may not justify burdening the framework with another mandatory operation since there are different ways to obtain key-like functionality by using different abstractions.

One might wonder about the relevance of creating a MapReduce framework for Prolog, since there are already several portable and flexible implementations for other programming languages in the literature. However, Prolog is a fundamental infrastructure for many research areas which are difficult to support in functional, imperative or object-oriented languages, such as Natural Language Analysis, Machine Learning or Inductive Logic Programming (ILP). ILP is the preferred target application for this work because it requires intensive and iterative processing of large amounts of data so as to infer rules applicable to it. As such, a hybrid MapReduce construct would be a valuable tool to make this process simpler and more efficient.

II. Related Work

There are presently several MapReduce implementations described in the literature [5]–[12], and in this document the most relevant to our work will be briefly introduced.

The HDFS or Hadoop Distributed File System [2] is a fault-tolerant distributed file system, which is designed to run on low-cost hardware. Its purpose is to meet the requirements of applications which need to manipulate large datasets and it was designed with a batch processing methodology in mind, as opposed to iterative data processing. In [11] Hadoop is compared to other approaches of large-scale data analysis and the overall task processing time was found to be 3.2 times slower than the second slower approach tested [11].

Twister [7] presents an architecture different from other MapReduce frameworks since it provides efficient support for iterative MapReduce calls. Unlike most systems in the literature, it uses a publish/subscribe messaging protocol and attempts to reduce the amount of communication data to a minimum by increasing the granularity of the map operation. This approach presents slightly faster results than Hadoop in the situations described in [7].

SAGA [8] is a high level API which executes operations on distributed systems, supporting various architectures like clusters, clouds or grids. Unlike the two previous frameworks,

SAGA is implemented natively in C++, as opposed to Java, and the MapReduce model was recently introduced into it. This approach is slower than most others due to its portability; the fact that it is not optimized for one distributed system only has a cost in terms of efficiency.

III. MAPREDUCE FOR PROLOG

The MapReduce framework presented here extends the original MapReduce for Prolog [5]. This framework is native to the Yap Prolog system [13] and its API is declarative in nature. It implements a master-slave architecture and it supports both iterative runs using the same workers and several scheduling methods for the data, which can be selected by the user at runtime.

The motivation for the work presented here lies in the need for a transparent tool which can support explicit parallelism in heterogeneous environments and whose results present speed-ups even for small datasets.

Our aim with this work is to improve on the previous implementation of this framework so as to execute in shared and distributed memory interchangeably from the user's point of view. We believe this contributes towards more and simpler data processing support in Prolog, and find it particularly relevant at an age when multi-core processors and heterogeneous clusters are increasingly common and inexpensive.

A. Hybrid Model

The model's architecture is loosely based on the architecture described in [6] in the sense that it supports clusters of machines, but it innovates by taking advantage of the parallelism within each machine. Figure 1 shows how our framework can apply to a generic number of processors running on top of a distributed architecture. Conceptually, each MapReduce computation will be performed on a *communicator*; this communicator is an abstraction representing a set of teams - each composed of a set of workers - and its resources are automatically managed by the system. Figure 1 depicts three processes and two communicators, *Comm A* (in blue) composed of two teams (*A0* in the Main Process and *A1* in Process 1), with two and three workers respectively, and *Comm B* (in orange), also composed of two teams (*B1* and *B2*), including two and three workers. Each process also includes a controller thread that handles the MPI communications between processes; in addition to that, the main process runs the thread the user should interact with.

Communicators must be created and destroyed by the user and they have an unique identifier name. Their resources can belong to different processes, but the data files used by them must be accessible by all these processes as well. Along with files, the user must also make the map and reduce predicates available to all processes; this can be done by making them available in a file before runtime, or by taking advantage of the `send_script/2` feature of this MapReduce implementation to run code remotely, in a different communicator.

Unlike earlier implementations of MapReduce for Prolog, this hybrid implementation allows for non-blocking MapReduce calls, making it possible to place several calls at one time - albeit in different communicators - and to retrieve the result at a later stage in the computation. This new feature makes the implementations much more flexible and allows for further concurrency to be taken advantage of since the main thread is not forced to wait for the termination of a MapReduce computation in order to proceed with other tasks.

The interface provides the possibility to test whether a MapReduce computation result has arrived in a non-blocking way. Should the computation be dependent on this result, there is also a predicate which uses non-busy waiting to wait for this result. However, the retrieval must be made by the same process which made the MapReduce call in the first place.

Orthogonally to this, each process in the cluster is assumed to be running on a separate machine (represented as a grey box in Figure 1) and so a *controller* thread is created to manage communication details; there is only one such thread per process. This thread is responsible for the creation and deletion of *teams* - or sets of worker threads belonging to one communicator (represented as boxes and octagons in Figure 1, respectively); the controller also manages all MPI communication among different processes and maintains control data structures pertaining to the *workers* attached to the process. The controller is thus the only entity to change the global database of a process, except for the case where the main thread is also running (typically process 0). This allows for a significant decrease in synchronization points, since it is guaranteed that the memory will only be written on by one thread.

Presently, all teams have a dynamic work scheduling strategy, meaning that for each team a queue is maintained by the controller as part of the communicator data structures, and regularly filled with more work as it becomes empty when the workers remove tasks. In addition, the system implements a steal work strategy inside a communicator, so should a local queue be empty, the controller will attempt to steal work from other processes where the same communicator has active teams. Whilst this can be an efficient strategy to guarantee good load balancing, its communication complexity can increase with the number of processes. As such, it was decided to organize these processes in an unidirectional ring, which also helps to guarantee termination using the Chandy and Lamport ring termination algorithm [3].

Overall, preliminary experiments of hybrid MapReduce for Prolog show that it can correctly create and destroy communicators, as well as terminate MapReduce calls placed in different orders and configurations of processes and communicators.

B. File System

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 MapReduce possibly executing on one machine alone. We have designed a transparent architecture divided in five functional modules as follows:

- Setup — contains the predicates required to initialize a communicator, as well as to delete it and guarantee that all communication is terminated.
- Data — can be used for loading data files to arrays of data. The use of this module is optional, since the user can load the data differently.
- Main — is composed of user level predicates required to make MapReduce calls and retrieve their result. This file should be used only on the process which will be running the main computation.

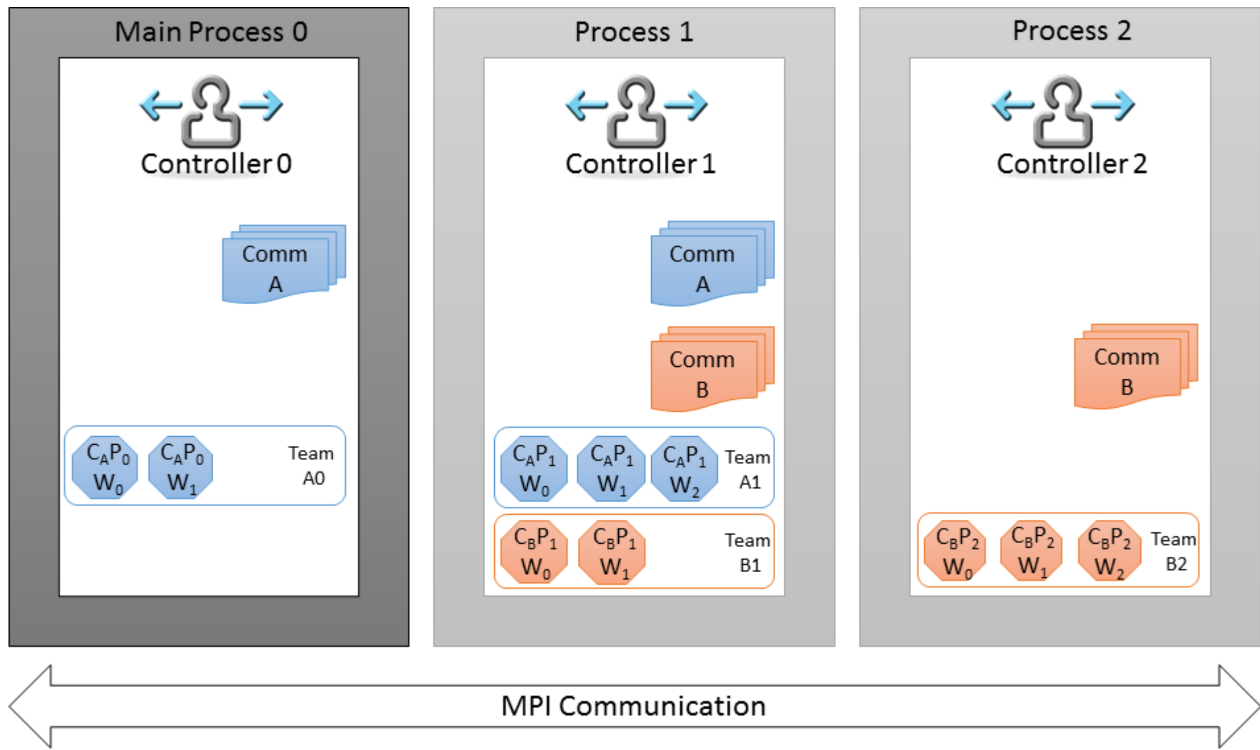

Fig. 1. Schematics of MapReduce for Prolog model

Control must be called on all machines/processes not running the main computation and contains predicates pertaining scheduling techniques and management of workers.

Worker contains the lower level predicates which perform the execution of the map and reduce operations defined by the user.

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.

C. User Interface

Hybrid MapReduce for Prolog user interface is composed of seven predicates, as illustrated in Figure 2. The last two predicates `map/2` and `reduce/2` are user-defined but their signature must follow the one presented in Figure 2.

The `init_interface/0` and `end_interface/0` predicates respectively initialize and end the MPI environment. These predicates must be called once at the beginning and end of the execution and are mainly an abstraction to the `MPI_Init()` and `MPI_Finalize()` functions.
Predicates `create_communicator/3` and `delete_communicator/1` are, as expected, used to create and delete communicators; these predicates then remotely run code in different processes should the communicator own resources in more than one process. They are also responsible for creating local workers and a work queue.
MapReduce calls can be made using `map_reduce_call/5` predicates; this predicate does not require the last argument and if it is not provided, no code is executed remotely prior to

```
init_interface.
end_interface.

create_communicator(+CommunicatorName,+
    ListOfResources,+DataFile),
detele_communicator(+CommunicatorName).

map_reduce_call(+CommunicatorName,+
    MapPredicate,+ReducePredicate,+
    DataList,+PreScript).

map_reduce_get_result(+CommunicatorName,-
    Result,+PostScript).
map_reduce_get_result_nonblocking(+
    CommunicatorName,-Result,+PostScript).

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

Fig. 2. Hybrid MapReduce for Polog interface

the call. The `DataList` interval must be provided as a list of indices and the `MapPredicate` and `ReducePredicate` are the names of the user defined map and reduce operations and must follow the signature also presented in Figure 2.
The `map_reduce_get_result/3` predicate can be used to retrieve the result of a MapReduce execution in that communicator and also does not require the last argument. There is also a non-blocking version of this predicate, which unifies `Result` with the result for the operation should it be completed and fails otherwise.

IV. Current and Future Work

As an initial experiment, it was decided to parallelize the coverage of examples in the ILP Aleph; example coverage is executed in the assessment of every hypothesis generated by the system and thus seems like a good starting point. The overall time an ILP system spends evaluating hypothesis can represent a large percentage of the execution time and so the performance of the system can be improved by parallelizing this code section. However, each hypothesis evaluation takes in fact a rather short time; the significant execution time spent in coverage is due to the frequency with which this operation is executed. This presents an added challenge to the framework: the overheads of parallelization must be small enough so that each coverage step speeds-up individually.

Additionally, assessing the coverage of a hypothesis has no side-effects in other tasks, and so the adaptation process should be straightforward: in this particular section of code, positive and negative example evaluation is independent from any data except the hypothesis being tested. This makes the task particularly suited for use with our MapReduce for Prolog framework. The map operation verifies whether the hypothesis explains the example and the reduce operations counts how many examples were explained. The final reduce will thus return the coverage for a given hypothesis.

Despite the alterations to Aleph being fairly straightforward, our preliminary experiments showed no speed-up. After analysing the code in order to rule out conceptual issues, it was decided to run a performance analysis of the code using gperftools [1]. Results showed that the slowdown was due to issues in the Yap Prolog system's access to the internal database, namely to the *Atom Table* data structure, now being handled.

Yap's Atom Table stores atoms of all types, whether they are array names, predicate names, simple atoms and so on. Each entry in the Atom hashtable is a linked list, in which each object is a different atom, or *Atom Entry*. If a new atom is hashed to that entry, it is inserted in the beginning of the linked list, directly between the hash table and the previously first atom entry in the list.

An atom entry can then have different sets of *properties* according to what sort of entity it represents. Different properties for the same atom are also stored in a linked list connected to the functor property (*FuncProp*) of that atom. As is the case with the atom entry linked list, new properties are always inserted in the beginning of the property list as well. In the particular case of predicates, and since they are attached to the functor property of the atom in most cases, there is still another level to be considered: that of predicate properties (*PredProp*). This linked list stores predicates with the same name and arity, but of different modules. Each of these levels of the Atom Table is protected by a *pthread_rwlock* and so the attempt to access the same predicate repeatedly causes a large overhead due to the configuration of these structures.

The work undertaken thus far is, however, promising. Once the concurrent database access of the Yap Prolog system has been resolved, we believe that our proof of concept will indeed scale the coverage evaluation time in proportion to the number of cores. We are looking forward to also testing this approach for several processes, since there are interesting data distribution issues to be addressed there, as well as performance and communication issues.

V. Conclusion

The hybrid MapReduce for Prolog framework presented is a flexible and versatile implementation which can take advantage of both shared and hybrid memory parallelism in non-structured machine clusters with different configurations. Its architecture and interface are described, as well as the methodology for some preliminary experiments. The first stage of this proof of concept was to apply the hybrid model to the ILP system Aleph, and the preliminary results show some performance problems related to the underlying Prolog system. Once these issues are overcome, this tool is expected to speed-up Aleph's hypotheses evaluation process.

Acknowledgment

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 PTDC/EEI-SII/2094/2012 (FCOMP-01-0124-FEDER-029010). Joana Côrte-Real is funded by the FCT grant SFRH/BD/52235/2013.

References

[1] gperftools: Fast, multi-threaded malloc() and nifty performance analysis tools. https://code.google.com/p/gperftools/.

[2] D. Borthakur. The Hadoop Distributed File System: architecture and design. *Hadoop Project Website*, 11:21, 2007.

[3] K. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[4] Cheng-Tao Chu, S. Kyun Kim, Yi-An 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.

[5] J. Côrte-Real, I. Dutra, and R. Rocha. Prolog Programming with a Map-Reduce Parallel Construct. In T. Schrijvers, editor, *Proceedings of the International Symposium on Principles and Practice of Declarative Programming (PPDP 2013)*, pages 285–295, Madrid, Spain, September 2013. ACM Press.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] S. Plimpton and K. Devine. MapReduce in MPI for Large-scale Graph Algorithms. *Parallel Computing*, 37(9):610–632, 2011.

[13] 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.