

Computing Answer Sets for Monadic Logic Programs via MapReduce

Ilias Tachmazidis, Grigoris Antoniou and Wolfgang Faber

University of Huddersfield, UK

Abstract. In this paper, the applicability of the MapReduce framework for parallel computation for monadic logic programs is studied. In monadic programs all predicates are of arity one. Two different approaches are suggested: the first is based on parallelizing on constants, the second parallelizes on predicates. In each case, a method is provided that makes use of MapReduce and calls standard ASP solvers as backends via an abstract API. For each method, we provide a theoretical analysis of its computational impact.

1 Introduction

Answer Set Programming (ASP) is a powerful formalism for knowledge representation and common sense reasoning [1]. Encouraged by the availability of efficient inference engines, such as Gringo and Clasp [4], DLV [10], GtT [7], Cmodels [11], ASP has found several practical applications in various domains, including data integration [9], semantic-based information extraction [14, 15], e-tourism [19], workforce management [20], and many more. As a matter of fact, these ASP systems are continuously enhanced to support novel optimization strategies, enabling them to be effective over increasingly larger application domains.

Parallelization has been one of the proposed optimization techniques for ASP. We mention two previous approaches: The system *clasp* [5] uses the MPI (Message Passing Interface), but it needs a preliminary grounding step, as it accepts only ground or propositional programs. The system described in [18] uses POSIX threads on shared memory for parallelized grounding. To our knowledge, so far no system has been proposed that exploits parallelism and avoids full grounding. In this paper we describe a first attempt to do so by means of the MapReduce framework, but we restrict our focus to monadic ASP programs, in which each predicate is of arity one.

In a broader setting, parallelization in general has been identified as a prominent approach for large scale reasoning [3] the computation among nodes. Two approaches, namely rule partitioning and data partitioning [21] were proposed in the literature.

In the case of rule partitioning, the computation of each rule is assigned to a node in the cluster. Thus, the workload for each rule (and node) depends on the structure and the size of the given rule set, which could possibly prevent

balanced work distribution and high scalability. An experimental evaluation reported in [21] shows sub-linear but monotonic speedups. However, the used rule sets were small, and thus, only a small number of processors could be utilized. On the other hand, for the case of data partitioning, data is divided in chunks with each chunk assigned to a node, allowing more balanced distribution of the computation among nodes.

Parallel reasoning, based on data partitioning, has been studied extensively. WebPIE [26] implements forward reasoning under RDFS and OWL *ter Horst* semantics over the MapReduce framework [2] scaling up to 100 billion triples. In [6] authors present RDFS inference scaling up to 512 processors with the ability to process, entirely in-memory, 20 billion triples.

MARVIN [17] implements the SpeedDate method, presented in [8], where authors pointed out and addressed the scalability challenge posed by the highly uneven distribution of Semantic Web data. In particular, MARVIN follows the *divide-conquer-swap* strategy in which triples are being swapped between nodes in the cluster in order to achieve balanced workload.

FuzzyPD [12,13] is a MapReduce based prototype system allowing fuzzy reasoning in OWL pD^* with scalability of up to 128 process units and over 1 billion triples. In [27] authors present an approach for finding all justifications of an entailment in OWL pD^* , which has the ability to scale to over 1 billion triples. Description logic in the form of $\mathcal{EL}+$ have been studied in [16]. The authors parallelize an existing algorithm for $\mathcal{EL}+$ classification by converting it into MapReduce algorithms, while experimental evaluation was deferred to future work.

The aforementioned approaches are mainly focused on monotonic reasoning. However, there has been work on nonmonotonic reasoning as well. Specifically, in [24] authors presented the first step towards scalable reasoning, using the MapReduce framework, over defeasible logic with unary predicates. Experimental evaluation showed that defeasible reasoning is feasible over billions of facts.

In [25] the authors extended their approach by performing parallel defeasible reasoning for predicates of arbitrary arity. In particular, it has been presented that defeasible reasoning, under the assumption of stratification, can be performed over millions of facts and has the potential to scale up to billions of facts. Results in [23] indicate that this approach is applicable to RDF data, showing good scalability properties and being able to handle a benchmark dataset of 1 billion triples.

In [22] the authors studied parallel reasoning, based on data partitioning, over negative rules, namely over rules containing negative subgoals. In particular, stratified semantics of logic programming, equivalent to the well-founded semantics for stratified programs were considered. Experimental results showed that stratified semantics can be applied to billions of facts.

To the best of our knowledge, this is the first work to study the applicability of the MapReduce framework for parallel computation of monadic logic programs under the semantics of answer set programming. As mentioned earlier, in this paper we deal with monadic programs, namely programs where all predicates are

of arity one. Our main contribution consists of two approaches, the first is based on parallelizing on constants and the second on parallelizing on predicates. Each approach makes use of MapReduce while calling standard ASP solvers as backends via an abstract API. For each method, we provide a theoretical analysis of its computational impact.

The rest of the paper is organized as follows. Section 2 introduces briefly the MapReduce framework and answer set programming. A description of single variable (nomadic) programs and the intuition behind the proposed parallelization techniques is provided in Section 3. Parallel reasoning based on constants is described in Section 4, while Section 5 provides the description for the predicate based approach. Further issues regarding parallelization are discussed in Section 6. We conclude in Section 7.

2 Preliminaries

2.1 MapReduce Framework

MapReduce is a framework for parallel processing over huge datasets [2]. Processing is carried out in a map and a reduce phase. For each phase, a set of user-defined map and reduce functions are run in parallel. The former performs a user-defined operation over an arbitrary part of the input and partitions the data, while the latter performs a user-defined operation on each partition.

MapReduce is designed to operate over key/value pairs. Specifically, each *Map* function receives a key/value pair and emits a set of key/value pairs. Subsequently, all key/value pairs produced during the map phase are grouped by their key and passed to reduce phase. During the reduce phase, a *Reduce* function is called for each unique key, processing the corresponding set of values.

Let us illustrate the *wordcount* example. In this example, we take as input a large number of documents and calculate the frequency of each word. The pseudo-code for the *Map* and *Reduce* functions is provided in Algorithm 1.

Consider the following documents as input:

Doc1: "Hello world."
Doc2: "Hello MapReduce."

During map phase, each map operation gets as input a line of a document. *Map* function extracts words from each line and emits pairs of the form $\langle w, "1" \rangle$ meaning that word w occurred once ("1"), namely the following pairs:

$\langle \text{Hello}, 1 \rangle \quad \langle \text{world}, 1 \rangle \quad \langle \text{Hello}, 1 \rangle \quad \langle \text{MapReduce}, 1 \rangle$

MapReduce framework will perform grouping/sorting resulting in the following intermediate pairs:

$\langle \text{Hello}, \langle 1, 1 \rangle \rangle \quad \langle \text{world}, 1 \rangle \quad \langle \text{MapReduce}, 1 \rangle$

During the reduce phase, the *Reduce* function sums up all occurrence values for each word emitting a pair containing the word and the frequency of the word. Thus, the reducer with key:

Algorithm 1 Wordcount example

```
map(Long key, String value) :  
  // key: position in document  
  // value: document line  
  for each word  $w$  in  $value$   
    EmitIntermediate( $w$ , "1");  
  
reduce(String key, Iterator values) :  
  // key: a word  
  // values : list of counts  
  int count = 0;  
  for each  $v$  in  $values$   
    count += ParseInt( $v$ );  
  Emit(key, count);
```

Hello will emit <Hello, 2>
world will emit <world, 1>
MapReduce will emit <MapReduce, 1>

2.2 Answer Set Programming

For a definition of syntax and semantics of ASP, we refer to [1]. We will next focus on monadic, or single-variable programs.

3 Single variable programs

In this section, we present a solution for single variable ASP programs. Consider the following program:

$$\begin{aligned} male(X) &\leftarrow human(X), \text{ not } female(X). \\ female(X) &\leftarrow human(X), \text{ not } male(X). \end{aligned}$$

and the following facts:

$$human(pat) \quad human(mary) \quad female(mary)$$

The name *Pat* may either refer to a male (Patrick) or a female (Patricia), while *Mary* is a well known female name. Thus, this program has two answer sets, namely:

$$\begin{aligned} &\{human(pat), human(mary), male(pat), female(mary)\} \\ &\{human(pat), human(mary), female(pat), female(mary)\} \end{aligned}$$

A typical ASP solver would first ground the given program and then compute the aforementioned answer sets. However, this process can be improved for programs with one variable. The key observation here is the fact that for each

rule, literals in both head and body are instantiated using the same constant (as X is the only variable). Since this is true for every rule, we can consider the following program for each constant separately:

$$\begin{aligned} male &\leftarrow human, \text{ not } female. \\ female &\leftarrow human, \text{ not } male. \end{aligned}$$

Thus, reasoning is performed for *pat* and *mary* separately. There are two answer sets for *pat*, namely $\{human(pat), male(pat)\}$ and $\{human(pat), female(pat)\}$, and one answer set for *mary*, namely $\{human(mary), female(mary)\}$.

Such reasoning process reduces both reasoning complexity and space requirements for answer sets storage. In general, we consider the following parameters:

- k , which is the number of predicates.
- m , which is the number of constants.
- $n = k * m$, which is the number of ground literals.

For the program containing the variable X , there are 2^n potential answer sets, since we need to consider every ground literal. However, by performing reasoning for each constant separately, the number of potential answer sets is reduced to $2^k * m$. In addition to this complexity reduction, we can speed up the process by performing reasoning, either on each constant or on each subset of predicates, in parallel applying the MapReduce framework.

4 Parallel reasoning based on constants

In this section, we provide a solution based on the MapReduce framework. Our solution is based on the concept of grouping given facts based on their constants and performing reasoning for each constant separately, as described in Algorithm 2. Consider the following program:

$$\begin{aligned} male &\leftarrow human, \text{ not } female. \\ female &\leftarrow human, \text{ not } male. \end{aligned}$$

The *Map* function reads facts of the form $predicate(constant)$ and emits pairs of the form $\langle constant, predicate \rangle$. Functions $getConstant(value)$ and $getPredicate(value)$ extract the constant and the predicate respectively from each *value* (given fact).

Given the facts: $human(pat)$, $human(mary)$ and $female(mary)$, *Map* function will emit the following pairs :

$$\langle pat, human \rangle \quad \langle mary, human \rangle \quad \langle mary, female \rangle$$

Note that, reasoning is performed for each constant (*pat*, *mary*) separately, and in isolation. Therefore, the MapReduce framework will group/sort the pairs emitted by *Map*, resulting in the following pairs:

$$\langle pat, human \rangle \quad \langle mary, \langle human, female \rangle \rangle$$

Algorithm 2 Parallel inference based on constants

```
map(Long key, String value) :  
    // key: position in document (irrelevant)  
    // value: document line (a fact)  
    constant = getConstant(value);  
    predicate = getPredicate(value);  
    EmitIntermediate(constant, predicate);  
  
reduce(String key, Iterator values) :  
    // program: propositional program stored in memory  
    // key: constant  
    // values : list of predicates (facts)  
    Set facts =  $\emptyset$ ;  
    for each  $v$  in values  
        facts.add(v);  
    ASPSolver solver = ASPSolver.getSolver();  
    solver.reason(facts, program);  
    Emit(key , solver.getResults());
```

Each *Reduce* function has to perform, in parallel, reasoning on an ASP solver (we assume any available solver that can be imported as an external library). First, *Reduce* function reads all the *values* (predicates) and adds them to *facts*, which is a set of predicates. We use a generic API defining functions such as *ASPSolver.getSolver()*, which provides an instance of the available solver. Subsequently, *solver.reason(fact, program)* performs reasoning based on the given predicates and the in-memory propositional program. Calculated answer sets are extracted from the solver using *solver.getResults()*.

Thus, during the reduce phase the reducer with key:

pat, will emit $\langle \text{pat}, \langle \{\text{human}, \text{male}\}, \{\text{human}, \text{female}\} \rangle \rangle$
mary, will emit $\langle \text{mary}, \{\text{human}, \text{female}\} \rangle$

Note that in case there is no answer set for at least one constant, then no answer set exists for the given ASP program.

5 Parallel reasoning based on predicates

An alternative solution for parallel reasoning, using the MapReduce framework, is based on the idea of grouping given facts according to the set of predicates they form for each constant and performing reasoning on each set of predicates separately. Consider the following program:

$\text{male} \leftarrow \text{human}, \text{not female}.$
 $\text{female} \leftarrow \text{human}, \text{not male}.$

Algorithm 3 Calculating sets of predicates per constant

```
map(Long key, String value) :  
    // key: position in document (irrelevant)  
    // value: document line (a fact)  
    constant = getConstant(value);  
    predicate = getPredicate(value);  
    EmitIntermediate(constant, predicate);  
  
reduce(String key, Iterator values) :  
    // key: constant  
    // values : list of predicates (facts)  
    Set facts =  $\emptyset$ ;  
    for each  $v$  in  $values$   
        facts.add(v);  
    Emit(facts, key);
```

Our first step is to calculate the sets of predicates for each constant, as depicted in Algorithm 3. Given the facts: *human(pat)*, *human(mary)* and *female(mary)*, *Map* function will emit the following pairs :

<pat, human> <mary, human> <mary, female>

Note that, the calculation of corresponding set of predicates for each constant (pat, mary) is performed separately, and in isolation. Therefore, the MapReduce framework will group/sort the pairs emitted by *Map*, resulting in the following pairs:

<pat, human> <mary, <human, female>>

Each *Reduce* function has to calculate and emit, in parallel, the corresponding sets of predicates. Thus, *Reduce* function reads all the *values* (predicates) and adds them to *facts*, which is a set of predicates. Finally, both the set of predicates (*facts*) and the constant (*key*) are emitted.

During the reduce phase the reducer with key:

pat, will emit <human, pat>
mary, will emit <{human, female}, mary>

Our next step is to perform reasoning for each set of predicates, as described in Algorithm 4. The *Map* function reads input of the form <set of predicates, constant> and emits pairs of the form <predicates, constant>. Functions *getPredicates(value)* and *getConstant(value)* extract the set of predicates and the constant respectively from each *value* (group of predicates per constant).

Given as input the pairs: <human, pat> and <{human, female}, mary>, *Map* function will emit the following pairs :

<human, pat> <{human, female}, mary>

Algorithm 4 Parallel inference based on predicates

```
map(Long key, String value) :  
    // key: position in document (irrelevant)  
    // value: document line (a fact)  
    predicates = getPredicates(value);  
    constant = getConstant(value);  
    EmitIntermediate(predicates, constant);  
  
reduce(String key, Iterator values) :  
    // program: propositional program stored in memory  
    // key: predicates  
    // values : list of constants  
    ASPSolver solver = ASPSolver.getSolver();  
    solver.reason(key, program);  
    for each v in values  
        Emit(v, solver.getResults());
```

Note that, reasoning is performed for each set of predicates separately, and in isolation. Therefore, the MapReduce framework will group/sort the pairs emitted by *Map*, resulting in the following pairs:

$$\langle \text{human}, \text{pat} \rangle \quad \langle \{\text{human}, \text{female}\}, \text{mary} \rangle$$

Each *Reduce* function has to perform, in parallel, reasoning on an ASP solver (we assume any available solver that can be imported as an external library). We use a generic API defining functions such as *ASPSolver.getSolver()*, which provides an instance of the available solver. Subsequently, *solver.reason(fact, program)* performs reasoning based on the given set of predicates and the in-memory propositional program. Finally, *Reduce* function reads all the *values* (constants) and emits, for each constant, the calculated answer sets. Calculated answer sets are extracted from the solver using *solver.getResults()*.

Thus, during the reduce phase the reducer with key:

$$\text{human}, \text{ will emit } \langle \text{pat}, \langle \{\text{human}, \text{male}\}, \{\text{human}, \text{female}\} \rangle \rangle \\ \{\text{human}, \text{female}\}, \text{ will emit } \langle \text{mary}, \{\text{human}, \text{female}\} \rangle$$

Note that in case there is no answer set for at least one set of predicates, then no answer set exists for the given ASP program.

6 Final remarks

Let us elaborate on the advantages of each parallelization technique and the challenges coming from programs containing rules with multiple variables. Practically, the parallelization technique that is more suitable in each case depends on the given program. Recall that the number of potential answer sets is reduced to $2^k * m$, where k is the number of predicates and m is the number of constants.

Parallel reasoning based on predicates has the advantage that reasoning for each set of predicates is performed only once. However, for programs containing few predicates, parallelization may be affected as reasoning process can be speeded up by a factor of 2^k . On the other hand, for reasoning based on constants, given that the amount of constants is most likely to be larger than the amount of available nodes in the cluster, parallelization comes with higher scalability. Clearly, the best approach depends on the number of predicates and constants, and their distribution within the given program.

For programs containing rules with more than one variable, parallelization is more complex. It is currently part of an ongoing study to determine whether reasoning can be partitioned in order to be able to take as input disjoint sets of literals, and compute independent answer sets. Facts are most likely to be included in several overlapping subsets, while potential answer sets may be overlapping as well, resulting in the re-computation of certain parts of the reasoning process. Further investigation is required in order to devise a scalable and efficient mechanism for parallel ASP reasoning.

7 Conclusion and Future Work

In this paper, we studied the parallel computation for monadic logic programs, namely programs where all predicates are of arity one, and proposed algorithms based on the MapReduce framework. More specifically, two different approaches were suggested: one based on parallelizing on constants and the other based on parallelizing on predicates. In each case, a method is provided utilizing MapReduce while calling standard ASP solvers as back-ends via an abstract API. A theoretical analysis of the computational impact of each method was provided as well.

In future work, we intend to implement our approach and test it empirically. Currently this is somewhat hampered by a requirement of the Hadoop framework: the ASP solver to be called should be written in Java, and its source should be available. So far, we were unable to find an existing solver implemented in Java with available source code. We will therefore have to settle with creating our own solver implementation, which will require some effort.

We also plan to study the feasibility of extending the approach to programs with predicates of greater arity. However, at this stage it is very unclear whether the MapReduce framework can be applicable in such a setting at all, and if so, how.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6. pp. 10–10. USENIX Association, Berkeley, CA, USA (2004), <http://dl.acm.org/citation.cfm?id=1251254.1251264>

3. Fensel, D., van Harmelen, F., Andersson, B., Brennan, P., Cunningham, H., Valle, E.D., Fischer, F., Huang, Z., Kiryakov, A., il Lee, T.K., Schooler, L., Tresp, V., Wesner, S., Witbrock, M., Zhong, N.: Towards LarKC: A Platform for Web-Scale Reasoning. In: ICSC. pp. 524–529 (2008)
4. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam answer set solving collection. *AI Communications* 24(2), 107–124 (2011)
5. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schnor, B.: Cluster-based asp solving with *clasp*. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. Lecture Notes in Computer Science, vol. 6645, pp. 364–369 (2011)
6. Goodman, E.L., Jimenez, E., Mizell, D., Al-Saffar, S., Adolf, B., Haglin, D.J.: High-performance computing applied to semantic databases. In: Antoniou, G., Grobelnik, M., Simperl, E.P.B., Parsia, B., Plexousakis, D., Leenheer, P.D., Pan, J.Z. (eds.) ESWC (2). Lecture Notes in Computer Science, vol. 6644, pp. 31–45. Springer (2011)
7. Janhunen, T., Niemelä, I., Seipel, D., Simons, P., You, J.H.: Unfolding Partiality and Disjunctions in Stable Model Semantics. *ACM Transactions on Computational Logic* 7(1), 1–37 (Jan 2006)
8. Kotoulas, S., Oren, E., van Harmelen, F.: Mind the data skew: distributed inferencing by speeddating in elastic regions. In: Rappa, M., Jones, P., Freire, J., Chakrabarti, S. (eds.) WWW. pp. 531–540. ACM (2010)
9. Leone, N., Gottlob, G., Rosati, R., Eiter, T., Faber, W., Fink, M., Greco, G., Ianni, G., Kalka, E., Lembo, D., Lenzerini, M., Lio, V., Nowicki, B., Ruzzi, M., Staniszki, W., Terracina, G.: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD 2005). pp. 915–917. ACM Press, Baltimore, Maryland, USA (Jun 2005)
10. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic* 7(3), 499–562 (Jul 2006)
11. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR’05, Diamante, Italy, September 2005, Proceedings. LNCS, vol. 3662, pp. 447–451. Springer Verlag (Sep 2005)
12. Liu, C., Qi, G., Wang, H., Yu, Y.: Large scale fuzzy pD* reasoning using mapreduce. In: Proceedings of the 10th international conference on The semantic web - Volume Part I. pp. 405–420. ISWC’11, Springer-Verlag, Berlin, Heidelberg (2011), <http://dl.acm.org/citation.cfm?id=2063016.2063043>
13. Liu, C., Qi, G., Wang, H., Yu, Y.: Reasoning with Large Scale Ontologies in fuzzy pD* Using MapReduce. *IEEE Comp. Int. Mag.* 7(2), 54–66 (2012)
14. Manna, M., Ruffolo, M., Oro, E., Alviano, M., Leone, N.: The HiLeX System for Semantic Information Extraction. *Transactions on Large-Scale Data and Knowledge-Centered Systems* (2011), to appear
15. Manna, M., Scarcello, F., Leone, N.: On the complexity of regular-grammars with integer attributes. *Journal of Computer and System Sciences (JCSS)* 77(2), 393–421 (2011)
16. Mutharaju, R., Maier, F., Hitzler, P.: A MapReduce Algorithm for EL+. In: Description Logics (2010)
17. Oren, E., Kotoulas, S., Anadiotis, G., Siebes, R., ten Teije, A., van Harmelen, F.: Marvin: Distributed reasoning over large-scale Semantic Web data. *J. Web Sem.* 7(4), 305–316 (2009)

18. Perri, S., Ricca, F., Sirianni, M.: Parallel instantiation of asp programs: techniques and experiments. *Theory and Practice of Logic Programming* 13(2), 253–278 (2013)
19. Ricca, F., Alviano, M., Dimasi, A., Grasso, G., Ielpa, S.M., Iiritano, S., Manna, M., Leone, N.: A Logic-Based System for e-Tourism. *Fundamenta Informaticae* 105(1–2), 35–55 (2010)
20. Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., Leone, N.: Team-building with answer set programming in the Gioia-Tauro seaport. *Theory and Practice of Logic Programming* (2011), to appear, doi:10.1017/S147106841100007X
21. Soma, R., Prasanna, V.K.: Parallel Inferencing for OWL Knowledge Bases. In: *ICPP*. pp. 75–82. IEEE Computer Society (2008)
22. Tachmazidis, I., Antoniou, G.: Computing the Stratified Semantics of Logic Programs over Big Data through Mass Parallelization. In: Morgenstern, L., Stefaneas, P.S., Lévy, F., Wyner, A., Paschke, A. (eds.) *RuleML. Lecture Notes in Computer Science*, vol. 8035, pp. 188–202. Springer (2013)
23. Tachmazidis, I., Antoniou, G., Flouris, G., Kotoulas, S.: Scalable Nonmonotonic Reasoning over RDF Data Using MapReduce. In: *SSWS+HPCSW* (2012)
24. Tachmazidis, I., Antoniou, G., Flouris, G., Kotoulas, S.: Towards parallel non-monotonic reasoning with billions of facts. In: Brewka, G., Eiter, T., McIlraith, S.A. (eds.) *KR. AAAI Press* (2012)
25. Tachmazidis, I., Antoniou, G., Flouris, G., Kotoulas, S., McCluskey, L.: Large-scale Parallel Stratified Defeasible Reasoning. In: Raedt, L.D., Bessière, C., Dubois, D., Doherty, P., Frasconi, P., Heintz, F., Lucas, P.J.F. (eds.) *ECAI. Frontiers in Artificial Intelligence and Applications*, vol. 242, pp. 738–743. IOS Press (2012)
26. Urbani, J., Kotoulas, S., Massen, J., van Harmelen, F., Bal, H.: Webpie: A Web-scale parallel inference engine using MapReduce. *Web Semantics: Science, Services and Agents on the World Wide Web* 10(0) (2012), <http://www.websemanticsjournal.org/index.php/ps/article/view/222>
27. Wu, G., Qi, G., Du, J.: Finding all justifications of OWL entailments using TMS and Mapreduce. In: Macdonald, C., Ounis, I., Ruthven, I. (eds.) *CIKM*. pp. 1425–1434. ACM (2011)