A new Model for Scalable θ-subsumption

Hippolyte Leger, Dominique Bouthinon, Mustapha Lebbah, and Hanane Azzag

Universite Paris 13, Sorbonne Paris Cite, L.I.P.N UMR-CNRS 7030 F-93430, Villetaneuse, France
{hippolyte.leger,hanene.azzag,mustapha.lebbah, dominique.bouthinon}@lipn.univ-paris13.fr

Abstract. The θ-subsumption test is known to be a bottleneck in Inductive Logic Programming. The state-of-the-art learning systems in this field are hardly scalable. So we introduce a new θ-subsumption algorithm based on an Actor Model, with the aim of being able to decide subsumption on very large clauses. We use Akka, a powerful tool to build distributed actor systems based on the JVM and the Scala language.

Keywords: θ-subsumption, Inductive Logic Programming, Actor Model, distributed computing, Akka

1 Introduction

θ-subsumption has been introduced by Robinson [23] to replace the logic implication which is undecidable [16]: a clause C θ-subsumes a clause D if and only if there exists a substitution θ such that $C\theta \subseteq D$.

Most of the Inductive Logic Programming (ILP) systems ([18, 22, 26, 19]) use θ-subsumption to check that a hypothesis covers an example. θ-subsumption is at the core of those systems, because the completeness and consistency of new candidate hypotheses require a large amount of subsumption tests against the examples.

θ-subsumption is decidable and equivalent to logical implication when C is not self-resolving and D is not tautological [12]. Unfortunately, the worst case time complexity of θ-subsumption is $(O(|D|^{|C|}))$ even when C and D are Horn clauses and D is fixed and ground. The basic θ-subsumption algorithm, based on SLD-resolution used in Prolog, is inefficient when the predicates are not determinate [18, 14]: there are many ways of mapping with the literals of C onto literals in D leading to a combinatorial explosion.

Many researches have achieved to design efficient θ-subsumption algorithms [11, 14, 25, 17, 5, 6, 15, 24][1]. Nevertheless, the scalable subsumption problem for distributed platforms received far less attention. Although researches have been conducted to make ILP capable of dealing with large data in a parallel environment (for instance [28, 7, 8, 4, 27]), as far as we know, no system focuses on θ-subsumption under modern framework and paradigms.

[1] See for instance [6] for a survey of most of those quoted works.

Our motivation is to design a simple, general model without any specific heuristic or parameters for a scalable subsumption engine that could easily be integrated into relational machine learning systems using distributed big-data platforms. In this paper, we show how to model θ-subsumption than can be run on cloud computers, and present preliminary results.

The rest of this paper is organized as follows. Section 2 introduces preliminaries used in the paper. In section 3 we present our new θ-subsumption modelling based on Actor Model. Section 4 describes the implementation of our model and gives the empirical preliminary results. Last section concludes and suggests future research.

2 Preliminaries

We consider here the θ-subsumption between two clauses C and D. We will use the term θ-subsumes whatever the name of the involved substitution is. We assume that C and D are function-free definite Horn clauses, C may contain variables and constants and D is variable-free (ground). In our context a (ground) substitution is a finite set $\{X_1/v_1, \ldots, X_n/v_n\}$ where X_i is a variable and v_i is a constant (a variable appears only once in a substitution). A substitution is applied to a first order formula to substitute variables with constants. For instance let $\theta = \{X/a, Y/b\}$ then $p(X, Y, Z)\theta = p(a, b, Z)$. Two substitutions θ_1 and θ_2 are not compatible if they assign two distinct values to the same variable. For instance $\theta_1 = \{Y/a\}$ and $\theta_2 = \{Y/b\}$ are not compatible. Conversely, the union of two compatible substitutions is a valid substitution: each variable appears only once. Let us introduce an example of θ-subsumption that will be used throughout this paper:

Example 1

$$
\begin{array}{lllll}
C = t(X) & \leftarrow p(X,Y,Z) & \wedge q(Z,T) & \wedge r(T,T,U). & \\
D = t(a) & \leftarrow p(a,b,c) & \wedge q(c,e) & \wedge r(e,e,g) & \wedge \\
 & \quad p(a,b,d) & \wedge q(d,f) & \wedge r(f,f,g) & \wedge \\
 & & & \quad r(e,f,g). &
\end{array}
$$

Example 1 shows that $C\theta \subseteq D$ both for $\theta = \{X/a, Y/b, Z/c, T/e, U/g\}$ and $\theta = \{X/a, Y/b, Z/d, T/f, U/g\}$. Let us consider the following properties that will be used in our model:

Property 1. C θ-subsumes D if and only if

1) there exists a substitution α only referring to the variables of $head(C)$, such that $head(C)\alpha = head(D)$ and,
2) there exists a substitution μ only referring to the variables of $body(C)\alpha$, such that $body(C)\alpha\mu \subseteq body(D)$.

Proof. $C\theta \subseteq D$ is equivalent to $head(C)\theta = head(D)$ and $body(C)\theta \subseteq body(D)$. $\Rightarrow$: Assume $C\theta \subseteq D$, then θ can be split into three substitutions: α only referring the variables of $head(C)$, μ only referring the variables of $body(C)\alpha$ and $\gamma = \theta \backslash (\alpha \cup \beta)$. $\Leftarrow$: Assume 1) and 2) hold, then $C\theta \subseteq D$ with $\theta = \alpha \cup \mu$. $\square$

Example 1 we have $head(C)\alpha = head(D) = t(a)$ with $\alpha = \{X/a\}$, and $body(C)\alpha = \{p(a, Y, Z), q(Z, T), r(T, T, U)\}$. For $\mu = \{Y/b, Z/c, T/e, U/g\}$ we have $body(C)\alpha\mu \subseteq body(D)$. Thus C θ-subsumes D with $\theta = \alpha \cup \mu$.

Property 2. Let $A = \{a_1, \ldots, a_n\}$ be a conjunction of literals and B be a conjunction of ground literals. Then A θ-subsumes B if and only if there exists a set of compatible substitutions $\{\mu_1, \ldots, \mu_n\}$ such that $a_i\mu_i \in B$ $(1 \leq i \leq n)$.

Proof. $A\mu \subseteq B \Leftrightarrow \{a_1\mu, \ldots, a_n\mu\} \subseteq B \Leftrightarrow$ there exists a set of compatible substitutions $\{\mu_1, \ldots, \mu_n\}$ with $\mu = \mu_1 \cup \ldots \cup \mu_n$ and where μ_i refers to the variables of a_i, such that $\{a_1\mu_1, \ldots, a_n\mu_n\} \subseteq B \Leftrightarrow a_i\mu_i \in B$ $(1 \leq i \leq n)$. □

Let us consider $A = body(C)\alpha$ and $B = body(D)$ mentioned above. We notice that $A\mu \subseteq B$ with $\mu = \mu_1 \cup \mu_2 \cup \mu_3$ where $\mu_1 = \{Y/b, Z/c\}$, $\mu_2 = \{Z/c, T/e\}$ and $\mu_3 = \{T/e, U/g\}$.

So, according to properties 1 and 2, the subsumption problem of two clauses C and D can be modelled as:

1. seek a substitution α only referring the variables of $head(C)$ such that $head(C)\alpha = head(D)$,
2. if step1 succeeds: (let $body(C)\alpha = \{a_1, \cdots, a_n\}$) find a set of compatible substitutions $\{\mu_1, \cdots, \mu_n\}$, where μ_i only refers the variables of a_i, such that $a_i\mu_i \in body(D)$ $(1 \leq i \leq n)$,
3. if step 2 succeeds: output the substitution $\theta = \alpha \cup \mu_1 \cup \cdots \cup \mu_n$.

Step 1 is very easy to check, so the new model of θ-subsumption we introduce in the next section focuses on step 2.

3 θ-subsumption based on an Actor Model

The general problem addressed in this section is: given a (usually non-ground) conjunction $A = \{a_1, \ldots, a_n\}$ and a ground conjunction B, find a set of compatible substitutions $\{\mu_1, \ldots, \mu_n\}$ such that μ_i refers only the variables of a_i and $a_i\mu_i$ belongs to B $(1 \leq i \leq n)$. To solve this problem we introduce an original subsumption procedure based on an Actor Model.

3.1 Actor Model

The Actor Model has been used as the theoretical basis for several practical implementations of concurrent systems [13]. It was motivated by the prospect of highly parallel computing machines communicating via a high-performance communications network [2, 1]. An actor is a computational entity linked to some other actors forming a graph, automata or network. Actors may modify private state, but can only affect each other through asynchronous messages. The message-driven framework of the Actor Model is well adapted to represent the θ-subsumption process introduced in the next section.

3.2 New θ-subsumption modelling based on an Actor Model

To solve the θ-subsumption problem between two conjunctions A and B we start by building an actor network from A. Then we send the atoms of B to the network which outputs the first (or all) substitution(s) ensuring the θ-subsumption and a message "end". If there is no possible substitution it will only produce the message "end".

Building the network The network (actually a directed graph) is made of four types of actors as illustrated Figure 1:

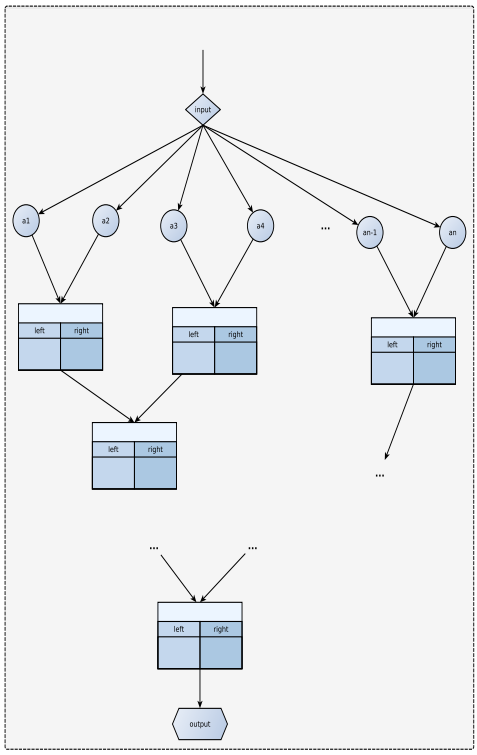

Fig. 1. Actors network built from a conjunction $A = \{a_1, \ldots, a_n\}$.

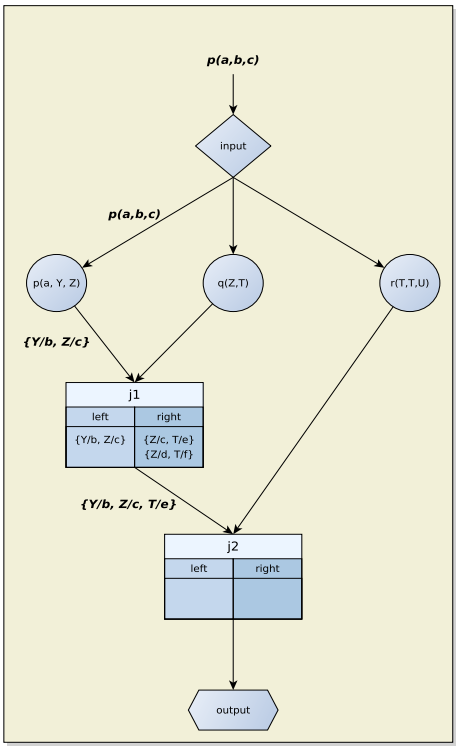

Fig. 2. Actors network built from the conjunction $A = \{p(a, Y, Z),\ q(Z, T),\ r(T, T, U)\}$. It illustrates how the message *p(a,b,c)* circulates in the network.

the input actor is the single input of the network. Each message it receives is a ground atom from B.

the substitution actor (represented by a circle): each one is associated with an atom a_i of A and is devoted to build substitutions from ground atoms it receives.

the join actor (represented by a rectangle) has two parents and is devoted to join the compatible substitutions provided by his parents (like a beta node in a RETE network [9]). It has two internal memories (*left* and *right*) to store the substitutions provided by its left and right parents.

the output actor is the single output of the network. It receives the final substitutions (if they exist) establishing the θ-subsumption between A and B.

If we do not consider the input and output actors the network is a full (proper) binary tree where the substitution actors are the leaves, all the other nodes being join actors. A single join actor, the root of the tree, is linked with the output actor (Algorithm 1 presents the way we build the whole network).

Algorithm 1 Building the actors network

```
buildNetwork(A)                    /* A = {a_1, ..., a_n} is a conjunction of n literals */
begin
  create the output actor out ;
  buildTree(out, n) ;                    /* build the tree of join and substitution actors */
  S = {s_1, ..., s_n} ← leaves(out) ;  /* get the substitution actors of the tree of root root */
        /* the leaves of the tree are stored during the building, so this operation is in O(1). */
  create the input actor in ;
  for i ← 1 to n do
        link s_i with in ;
        set a_i as internal label of s_i ;
  end for
  return in ;
end.

buildTree(j, n)                    /* j: join (or output) actor of the preceding level */
                                   /* n: number of leaves still to consider */
begin
  if n = 1 then
        create a substitution actor a and store it ;        /* a new leaf for the tree */
  else
        create a join actor a ;
        buildTree(a, n/2 + n mod 2) ;
        buildTree(a, n/2) ;
  end if
  link a to j ;
end.
```

The network contains $2n + 1$ nodes: $2n - 1$ nodes in the tree (n substitution actors and $n - 1$ join actors) plus the input and output nodes) so the time complexity to build the network is $O(n)$.

The θ-subsumption process Let us illustrate how such a network is used to check the subsumption through the network presented in Figure 2 built from the conjunction $A = body(C)\alpha = \{p(a, Y, Z),\ q(Z, T),\ r(T, T, U)\}$ where C is the clause given in Example 1. We also assume that the atoms of $B = body(D)$ $= \{p(a, b, c),\ p(a, b, d),\ q(c, e),\ q(d, f),\ r(e, e, g),\ r(f, f, g),\ r(e, f, g)\}$ are sent to the network:

- When a ground atom b of B is provided to the *input actor*, this actor sends b to all *substitution actor*s associated with the atoms of A built from the same predicate as b. ex) the *input actor* receives $b = p(a, b, c)$, then it sends $p(a, b, c)$ to the actor $p(a, Y, Z)$.
- When a *substitution actor* associated with an atom a_i of A receives a ground atom b it checks if there exists a substitution μ_i such that $a_i\mu_i = b$. If μ_i exists the actor sends it to the single *join actor* with which it is linked. ex) $b = p(a, b, c)$ and $a_i = p(a, Y, Z)$ then the substitution $\mu_i = \{Y/b, Z/c\}$ is sent to the *join actor* j_1, if $b = p(e, b, c)$ no substitution is sent. The worst case complexity to check if there exists μ_i with $a_i\mu_i = b$ is $O(v.ln(v))$ where v is the common arity of a_i and b ($ln(v)$ is the worst case complexity to access any argument of a_i and b).
- When a *join actor* receives a substitution μ from its left (right) parent it first stores it in its left (right) internal memory. Then, it joins μ with each compatible substitution δ found in its right (left) memory and sends $\mu \cup \delta$ to its single successor. ex) the *join actor* j_1 receives $\mu = \{Y/b, Z/c\}$ from its left parent, then it stores it in its left memory. Assume the right memory of j_1 contains the substitutions $\delta_1 = \{Z/c, T/e\}$ and $\delta_2 = \{Z/d, T/f\}$. Thus $\mu \cup \delta_1 = \{Y/b, Z/c, T/e\}$ is sent to the successor of j_1, while $\mu \cup \delta_2 = \{Y/b, Z/c, Z/d, T/f\}$ is not considered because it is not a valid substitution. The complexity to check that μ and δ are compatible is $O(|\mu|.ln(|\mu)|)$ (we assume that μ and δ have approximately the same size). So the number of operations made by a join actor when it receives a message μ is in $O(m.|\mu|.ln(|\mu|))$ where m is the current size of the right (left) memory.
- When the *output actor* receives a substitution μ it displays μ as a solution ($A\mu \subseteq B$). If we do not want any other solutions, the process terminates, otherwise the actor waits for other substitutions.

We use the network by sending all atoms of B to the *input actor*. To ensure that the activity of the network stops we send a specific *end-message* to the *input actor* when the latest atom of B has been sent. The *end-message* is broadcasted through the network to the *output actor* which terminates the process. Notice that when a *join actor* receives the *end-message* from one of its parents it waits for another *end-message* to come from its other parent before sending it to its successor and finally terminating itself.

Let us show that every substitution produced by the network is correct:

Property 3. Let A be a conjunction of literals and B be a conjunction of ground literals. Let $net(A)$ be the actor network built from A. If we send the atoms of

B to the *input actor* of $net(A)$, then every subsumption μ arriving to the *output actor* satisfies $A\mu \subseteq B$.

Proof. The behaviors of *substitution actors* and *join actors* are defined to produce only valid substitutions, so only valid substitutions circulate in the network, as a consequence μ is a valid substitution. Moreover, according to the tree structure of the network (minus the *input actor*) $\mu = \mu_1 \cup \ldots \cup \mu_n$ where μ_i $(1 \leq i \leq n)$ is a substitution produced by the *substitution actor* a_i when it receives a ground atom b of B, with $a_i\mu_i = b$. Thus, for $1 \leq i \leq n$, we have $a_i\mu_i \in B$ and consequently $A\mu \subseteq B$ (property 2). □

Now let us prove that every substitution satisfying the θ-subsumption between A and B is produced by the network (only if we do not stop the process when the first solution is seen):

Property 4. Let A be a conjunction of literals and B be a conjunction of ground literals. Let $net(A)$ be the actor network built from A. If we sent all atoms of B to the *input actor* of $net(A)$ then the *output actor* will receive every substitution μ such that $A\mu \subseteq B$.

Proof. All atoms of B are sent to the *input actor* which distribute them to the appropriate *substitution actors* $a_i s$. Therefore, each actor a_i produces all the substitutions μ_i such that $a_i\mu_i \in B$. Moreover, each *join actor* produce all the unions of compatible substitutions coming from its parents. So the *output actor* receives all compatible unions $\mu_1 \cup \ldots \cup \mu_n$. According to property 3, each one of them is a solution. □

4 Experiments

We first describe the implementation of the Actor Model introduced in section 3.2. Then we describe the data-set used in our experiments, and finally we present some performance analysis.

4.1 Actors implementation

Several programming languages implement the Actor Model. In this work we use the Akka [3] toolkit which is integrated to Scala [20], [21], a multi-paradigms (mainly functional) programming language built on the Java Virtual Machine. The parallelism in an Akka actor system relies on the dispatcher that is used. A dispatcher is a process that optimizes the workload and exchange of messages between different actors, according to the available computational resources (threads). The two main parameters of the dispatcher are the *parallelism-max* value, and the executor *throughput*. The *parallelism-max* value limits the number of threads to be used for computation. The executor *throughput* value sets the maximum number of messages of an actor to be processed by a thread before it jumps to the next actor in the execution queue. The higher the *throughput*, the more time will threads spend on each actor of the queue.

4.2 Dataset

We ran our implementation on the same piece of data with several distinct parameters values. The goal is to observe how the Actor Model behaves when running on different parallelism configurations and ultimately to have an idea of how our model would be able to scale up to a distributed environment. The dataset were generated to be similar in form and complexity to the instances of the Phase Transition problem [10]. The subsumer consists of a single clause with 30 literals, 4 distinct predicate symbols with an arity of 2 or 3. There are 8 distinct variable symbols. The subsumee example has the same properties, but holds 200 literals instead of 30. It is built to be subsumed by the hypothesis. Note that we have run the test to only find the first possible solution.

We have run the same subsumption test (with the same subsumer and subsumee) multiple times with executor throughput values ranging from 1 to 1000 and the parallelism-max parameter ranging from 1 to 4 (which means that the number of dedicated threads ranged from 1 to 10). The program was built on the SBT 2.3.12, Scala 2.11.7 and Akka 2.4.4 versions. All the computing was done on an Intel Core i7-5600U CPU running at 2.6GHz with 8GB of RAM and a 64 bits Linux system.

4.3 Performances

throughput	pmax	comp. time (s)
1	1 [1]	3296
	2 [3]	619
	3 [7]	499
	4 [10]	605
10	1 [1]	3214
	2 [3]	427
	3 [7]	269
	4 [10]	329
100	1 [1]	3620
	2 [3]	650
	3 [7]	317
	4 [10]	148
1000	1 [1]	7371
	2 [3]	1271
	3 [7]	403
	4 [10]	411

Fig. 3. Mean computation time for each different dispatcher configuration (pmax means parallelism max). The value between brackets shows the effective number of threads used for the corresponding configuration.

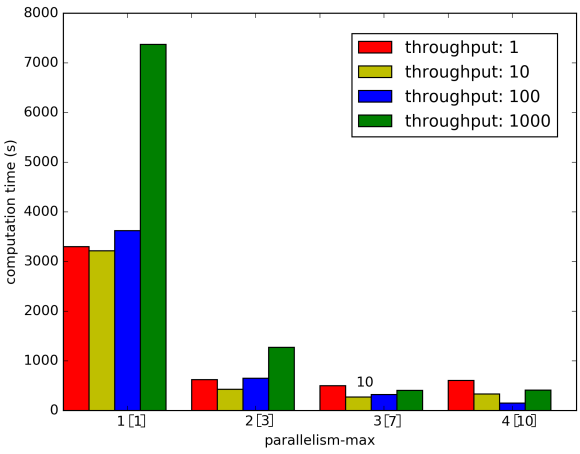

Fig. 4. Running time for each different dispatcher configuration. The value between brackets shows the effective number of threads used for the corresponding configuration.

In Figure 3 and Figure 4, we can see that both parameters have an impact on the running time. The increase of the parallelism-max value has a significant impact. Please keep in mind that the number of actors is directly linked to the number of literals the hypothesis (subsumer) holds. This means that for a bigger clause, the cost-efficiency of parallelism would be higher.

We used a monitoring tool called Kamon to track the number of messages that each actor are storing in their mailbox (because they are busy processing other messages) throughout time. Figure 5 shows the bottom-end topology of our actor system running on the afore-mentioned dataset. The other actor were cut out of the figure because they completed all of their tasks within several seconds of execution. Since our monitoring tool allows us to collect statistical metrics every 10 seconds, no relevant information about them could be collected in such a short time-period. Figure 6 shows the metrics collected while running the program on the same dataset, with a throughtput value of 1000 and the parallelism-max parameter set to 4. We can see that within the first 20 seconds, the actor A processes several messages. These are the messages coming from the right branch. It is then idle, because the upstream actors from the left branch are not sending any messages yet. The actor D shuts down after 30 seconds, leaving only three actors up and running. Some substitution candidates eventually reach the end of the workflow after 3 minutes. Then the actor A finds a solution substitution and the whole system shuts down. In this case, actor B appears as a bottleneck of our actor-system, cumulating over 200,000 messages at once. A possible optimization is to focus the parallelism effort on this specific actor to speed-up the whole workflow. Of course, the location of this bottleneck within the topology is highly dependent on the data. Thus, in order to achieve this optimization, we would need to programatically identify the actors whose mailbox sizes reach a certain threshold.

5 Conclusion and perspectives

We have created an original θ-subsumption model, which shows a great scalability potential and proves parallelization of the θ-subsumption test worthy of interest. Notice we have investigated a model of θ-subsumption founded on Mapreduce in the Spark framework [29]. It proved to be inadequate due to the fact that Mapreduce is not designed to deal with problems where we check the existence of a solution. On contrary, we show that Actor Modeling is indeed effective at reducing the running time, thus we can construct ILP models that efficiently use big data technologies. Due to a lack of time we did not compare the implementation of our model with classical state-of-the-art solutions like Subsumer [24], Django[17] or Resumer[15]. We must also refine our model by applying the major ILP optimizations, like clause partitioning and linked variables analysis.

We have just implemented a distributed version of our model and started our first distributed experiment using the Grid'5000 testbed, supported by a

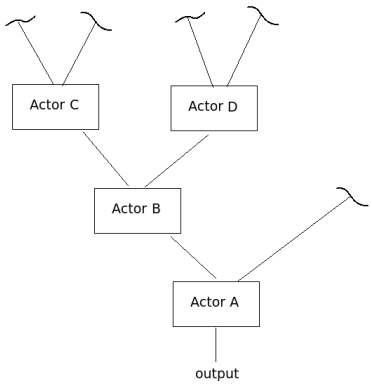

Fig. 5. The bottom-end of the actor-system

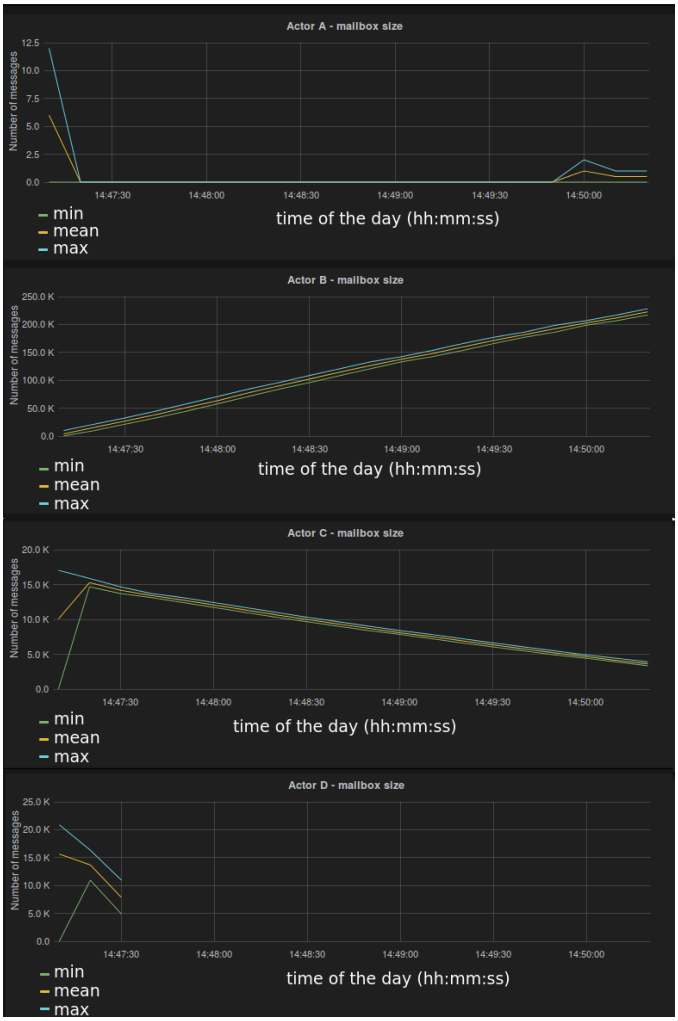

Fig. 6. Metrics collected for the actors of the bottom-end of the system.

scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). Although promising, our distributed implementation is not yet mature enough to discuss the results.

We also consider using different topologies of the actor network in order to reduce the bottle-neck phenomenon we discussed earlier. Finally, we are investigating another Actor Model for scalable θ-subsumption where the actors are the subsumee's literals instead of the subsumer's literals. This would lead to an increasing number of actors but an important reduction of the workload for each individual actor.

References

1. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
2. Gul Agha. An overview of actor languages. In *Proceedings of the 1986 SIGPLAN Workshop on Object-oriented Programming*, OOPWORK '86, pages 58–67, New York, NY, USA, 1986. ACM.
3. Jamie Allen. *Effective Akka*. O'Reilly Media, Inc., 2013.
4. Annalisa Appice, Michelangelo Ceci, Antonio Turi, and Donato Malerba. A parallel, distributed algorithm for relational frequent pattern discovery from very large data sets. *Intell. Data Anal.*, 15(1):69–88, 2011.
5. Stefano Ferilli, Nicola Fanizzi, Nicola Di Mauro, and Teresa M. A. Basile. Efficient theta-Subsumption under Object Identity. In F. Esposito and D. Malerba, editors, *AI*IA Workshop (Apprendimento Automatico)*, 2002.
6. Stefano Ferilli, Nicola Mauro, Teresa M. A. Basile, and Floriana Esposito. *AI*IA 2003: Advances in Artificial Intelligence: 8th Congress of the Italian Association for Artificial Intelligence, Pisa, Italy, September 2003. Proceedings*, chapter A Complete Subsumption Algorithm, pages 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
7. Nuno A. Fonseca, Fernando Silva, and Rui Camacho. *Strategies to Parallelize ILP Systems*, pages 136–153. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.
8. Nuno A. Fonseca, Ashwin Srinivasan, Fernando Silva, and Rui Camacho. Parallel ilp for distributed-memory architectures. *Machine Learning*, 74(3):257–279, 2009.
9. Charles L. Forgy. Expert systems. chapter Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, pages 324–341. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990.
10. Attilio Giordana and Lorenza Saitta. Phase transitions in relational learning. *Machine Learning*, 41(2):217–251, 2000.
11. G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *J. ACM*, 32(2):280–295, April 1985.
12. Georg Gottlob. Subsumption and implication. *Information Processing Letters*, 24(2):109 – 111, 1987.
13. Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

14. Jrg-Uwe Kietz and Marcus Lübbe. An efficient subsumption algorithm for inductive logic programming. In *Proceedings of the 11th International Conference on Machine Learning*, pages 130–138, 1994.
15. Ondrej Kuzelka and Filip Zelezn. A restarted strategy for efficient subsumption testing. *Fundam. Inform.*, 89(1):95–109, 2008.
16. John W. Lloyd. *Foundations of Logic Programming, 1st Edition*. Springer, 1984.
17. Jérôme Maloberti and Michèle Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55(2):137–174, 2004.
18. S Muggleton and C Feng. Efficient induction of logic programs. pages 368–381, 1990.
19. Stephen Muggleton, José Santos, and Alireza Tamaddoni-Nezhad. *Inductive Logic Programming: 19th International Conference, ILP 2009, Leuven, Belgium, July 02-04, 2009. Revised Papers*, chapter ProGolem: A System Based on Relative Minimal Generalisation, pages 131–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
20. Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.
21. Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.
22. Luc De Raedt and Maurice Bruynooghe. A theory of clausal discovery. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*, pages 1058–1063, 1993.
23. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.
24. Jose Santos and Stephen Muggleton. Subsumer: A Prolog theta-subsumption engine. In Manuel Hermenegildo and Torsten Schaub, editors, *Technical Communications of the 26th International Conference on Logic Programming*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 172–181, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
25. Tobias Scheffer, Ralf Herbrich, and Fritz Wysotzki. *Inductive Logic Programming: 6th International Workshop, ILP-96 Stockholm, Sweden, August 26–28, 1996 Selected Papers*, chapter Efficient θ-subsumption based on graph algorithms, pages 212–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.
26. Ashwin Srinivasan. The aleph system. 1987.
27. Ashwin Srinivasan, Tanveer A. Faruquie, and Sachindra Joshi. Data and task parallelism in ilp using mapreduce. *Machine Learning*, 86(1):141–168, 2012.
28. Yu Wang and David Skillicorn. Parallel inductive logic for data mining. In *In Workshop on Distributed and Parallel Knowledge Discovery, KDD2000*. ACM Press, 2000.
29. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.