Parallel computation of the reachability graph of petri net models with semantic information

Citation for published version (APA):

DOI:
10.1002/spe.2438

Document status and date:
Published: 01/05/2017

Document Version:
Accepted manuscript including changes made at the peer-review stage

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Parallel Computation of the Reachability Graph of Petri Net Models with Semantic Information

Eduardo González-López de Murillas¹, Javier Fabra², Pedro Álvarez², Joaquín Ezpeleta²

¹Architecture of Information Systems (AIS) group in the Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands (e-mail: e.gonzalez@tue.nl)
²Institute of Engineering Research (I3A) in the Department of Computer Science and Systems Engineering, University of Zaragoza, Spain (e-mail: {jfabra, alvaper, ezpeleta}@unizar.es)

SUMMARY

Formal verification plays a crucial role when dealing with correctness of systems. In a previous work the authors proposed a class of models, the Unary RDF Petri Nets (U-RDF-PN), which integrated Petri nets and (RDF-based) semantic information. The work also proposed a model checking approach for the analysis of system behavioural properties that made use of the net reachability graph. Computing such a graph, specially when dealing with high level structures as RDF graphs, is a very expensive task that must be considered. This paper describes the development of a parallel solution for the computation of the reachability graph of U-RDF-PN models. Besides that, the paper presents some experimental results when the tool was deployed in cluster and cloud frameworks. The results not only show the improvement in the total time required for computing the graph, but also the high scalability of the solution, which make it very useful thanks to the current (and future) availability of cloud infrastructures. Copyright © 0000 John Wiley & Sons, Ltd.

KEY WORDS: Petri nets, Reachability graph, Semantics, Parallel computing, High-performance computing, Cloud computing.

1. INTRODUCTION

From a technological point of view, Service Oriented Computing (SOC) has become the backbone of information systems for business companies. Independently of the specific way services are implemented (own host system, service providers, cloud infrastructures, etc.), services are fundamental elements for developing (added value) distributed applications for heterogeneous environments. The use of semantic techniques was a big help in automating the process of describing, discovering and integrating provided services in order to develop new business processes [1].

Independently of the technological infrastructures used for their deployment, current information systems are complex enough to make necessary the use of methodologies allowing not only to help in understanding and designing the system, but also in studying its possible behaviours so that a set of required properties could be analysed (system analysis) or, even, imposed (system synthesis and control). Petri nets [2] are a family of formalisms well suited for the modelling and analysis of concurrent/distributed systems which have been extensively used in different domains where workflows were a very natural way of dealing with modelling, analysis and implementation (see [3] for a general overview, and [4] for a specific view from the point of view of business processes or [5, 6, 7, 8] for semantic business process).

For the work on the domain of concurrent/distributed systems that also include semantic information, [9] presented a class of high level Petri nets, named Unary RDF Petri Nets (U-RDF-PN), as a formalism integrating the control flow view of Petri nets and semantic technologies to
deal with data (at arc and transition notation and at marking level). In order to analyse U-RDF-PN systems, the Computation Tree Logic (CTL) based model checker COMBAS was developed [10].

The temporal logic model checker presented in [9] required the computation of the model reachability graph. Reachability graph generation is known to be a complex problem from a computational point of view. This is because of the state explosion problem: the system can have a very big number of states (even be infinite).

In the case of a finite state space, different techniques have been developed trying to alleviate that problem. In some cases, specific components or properties of the models are used to reduce the number of states to be computed and stored. This is the case of stubborn sets [11] or symmetries [12]. In other cases, special data structures, such as the binary decision diagrams [13] or some kind of variants, are used for very efficient computation and storage [14, 15]. Other authors have focused on the definition of non-standard semantics in order to reduce the transition system and, therefore, the complexity of the reachability graph. This is the case of the LOTOS formal specification language, used to verify temporal properties is model checking [16].

In the case of non-finite reachability graphs, some techniques have been developed based on the concept of coverability graph [17, 18]. These techniques avoid the infiniteness of the graph, still providing with the capacity of analyzing a wide set of behavioral properties.

An alternative way, looking for a gain in computer power, is the development of parallel implementations. Different architectures, either based on shared memory [19, 20] or in distributed memory [21], have been proposed. Also, some effort has been done to support parallel programming languages and to ease the parallelization of existing code [22]. Efficient implementations of binary decision diagrams have been proposed in [23, 24] providing engineers with a powerful technology for the implementation of parallel space generators and analysers.

The nature of the model in [9] makes each state to be quite complex. A state is composed of two parts. On the one hand, the distribution of tokens in places, where each token is an RDF graph. On the other hand, since transitions can have postconditions, a state also has a logic formula, expressed according to the smt-lib standard [25], stating some relations among the model parametric values. This specificity and complexity makes that usual techniques for reachability graph construction and analysis were not applicable, making the implementation of new tools necessary. A working prototype was presented in [10]. However, we soon realized that space requirements for storing states and computing power for calculating them required a much more efficient and scalable solution.

Considering previous parallel solutions we concluded that a shared memory architecture would lack of the scalability and performance we were looking for. We wanted to be able to scale and adapt the tool for problems of different sizes. Choosing a shared memory approach would establish a (short) limit in the set of problems we could deal with, being such limit related to the capabilities of computing hardware. Therefore, a parallel implementation that could take advantage of the computing clusters at our disposal as well as the cloud-based resources any user or organization can hire if necessary, on demand and under a pay-per-use approach, was proposed as a valid and suitable approach.

As has been mentioned before, previous work exists on parallelizing or distributing the computation of reachability graphs. However, none of these solutions considers U-RDF-PNs as a modeling formalism. To support this special kind of high level Petri nets was the motivation to develop our own implementation. Further on, during the experimentation phase we encountered several challenges. First, the distribution and allocation of computational resources on distributed environments was not trivial. Second, it was necessary to adapt to the configuration particularities of each of these platforms. Third, many of these execution environments can be unstable due to several reasons: hardware failure, conflict with co-located processes from other users, lack of disk space, network errors, etc. Handling all this instability poses a big challenge that none of the existing solutions in the literature is able to tackle at the moment. Therefore, fault tolerance became one of the main goals when developing our solution.

This paper describes the proposed parallel solution as well as some experimental results to measure how power increases when new computing resources are integrated, showing that the
solution scales properly. In order to also show the flexibility of the tool, it has been used in a cluster environment as well as in the Google Cloud infrastructure. Section 2 describes the main elements involved in U-RDF-PN. Sections 3 and 4 describe the design and implementation of the tool while Section 5 show the experimental results obtained when it has been deployed in a Condor-based cluster and in the Google Cloud. Finally Section 6 introduces some conclusions and future work lines that could be followed in order to extend/improve the current research.

2. THE PROBLEM

Let us briefly describe the formalism of Unary-RDF-Petri nets (U-RDF-PN) and then state the problems that appeared when trying to compute the reachability graph. In the next section we will present the solutions proposed to deal with the problem of computing such graph using a parallel implementation.

2.1. U-RDF Petri nets

We are assuming the reader is familiar with (high level) Petri nets, and knows about its main definitions and properties. In this section we are going to informally describe the specific elements of the subclass of the U-RDF Petri nets.

Petri nets are a technology widely used in the world of workflows [26, 27]. As usual in Petri nets, a system state is modeled by means of the net marking while system evolutions are represented as the enabling and firing of transitions.

In order to be able to incorporate semantic information to Petri net models, we developed the class of Unary-RDF-Petri nets, U-RDF-PN [9]. In our approach, RDF is used to describe information about the inputs and outputs of each task represented in the workflow, as well as information about the data flows which are carried out through the workflow.

U-RDF-PN belongs to the family of high level Petri nets and share the notions of place, arc, transition and marking. Semantics have been incorporated in the tokens of the nets (tokens are RDF graphs with information related to the object the token refers to), in arc inscriptions (which are RDF graph-patterns) and, finally, in transitions (incorporating preconditions, postconditions and guards related to the transition input and output arc inscriptions).

Figure 1-a) sketches the main elements that define the structure of a Petri net: transitions and related arcs and places, with their inscriptions, as well as the tokens, whose distribution in places define the marking [2]. In the figure, transition corresponds to the invocation to a task called align_warp.

As usual in Petri nets, a system state is modelled by means of the net marking while system evolutions are represented as the enabling and firing of transitions [28]. Tasks will be modelled by means of transitions, as sketched in Figure 1, as follows:

1. Tokens in places will be RDF graphs, as well as the tokens produced by transition firings.
2. Transition input arcs will correspond to RDF graph patterns in the transition input arcs. At a given state, a token in an input place entailing the RDF graph pattern in the corresponding arc is a possible binding for firing the transition.
3. Transitions can have attached guards. A guard is a boolean expression involving URIs, literals and input variables (variables used in RDF graph patterns attached to the input arcs). Among the possible graph candidates, only those satisfying the guard can be used to enable the transition.
4. A transition is enabled at a given marking when candidate tokens are found for every input arc so that a binding is possible (which means that the same variable in different input arcs must correspond to the same value in the different input candidate tokens) and so that the guard attached to the transition is made true for the chosen values.
5. The RDF graph pattern attached to the output arcs represent task outputs, while the postcondition mapping associates to a transition its corresponding postcondition, establishing the relation between the input and output task values.
Figure 1. a) Scheme of a transition and related arcs and places in a U-RDF-PN b) The model after transition firing
2.2. Computation of the reachability graph

For a given Petri net, the reachability graph is the graph containing the set of reachable markings (obtained after the sequential firing of transitions from the initial marking) as nodes and an arc joining two states \( m_1 \) and \( m_2 \) if \( m_2 \) is reachable from \( m_1 \) by firing a transition. Since the transition will be fired for a concrete binding, the binding itself is associated to the arc.

The generation of the reachability graph is based on the classical algorithm used for computation of the reachability graph in Petri nets [2].

The algorithm is quite simple. Initially, the reachability graph only contains the initial marking, being this marking the only marking on a stack of pending (to be processed) markings. Then an iterative process is applied as follows: take the marking in the top of the stack; compute the possible firings from that marking, as well as the marking reached for each possible firing (possible successor markings); if the new marking is not in the until-now computed reachability graph, add such marking and state transition to the graph, and also push this marking into the stack. The process terminates when the stack is empty. The method relies on two main functions. The first one is the function that, for a given marking, computes the set of enabled transition firings, as well as the markings reached in case of such firings. The second one, the function that, given two markings, determines whether they are equivalent or not.

In the case of U-RDF-PN, the functions involved in the reachability graph computation must work with RDF annotations: the function looking for enabling of transitions, the function for firing a transition and, finally, the function checking for the equivalence of two markings. As shown in [9], evaluating these functions is a quite time-consuming task and then, computing the reachability graph of U-RDF-PN, can become a very hard task. In fact, the model uses the simply entailment, which, as proved in [29], is NP-complete when patterns contain blank nodes (in our case this can occur when looking for transition enabling) and is in P when the target graph does not contain blank nodes (in our case this occurs when comparing reachable markings for equivalence). Therefore, besides the inherent cost associated to the number of possible reachable states, U-RDF-PN must consider the cost of looking for possible bindings for transition enabling when building the reachability graph.

This is the reason for looking for a parallel implementation that alleviates the problem. Given the high dependency between operations in the Reachability Graph generation problem, the distribution of the algorithm requires a central coordinator that handles task assignment and controls the finalization of the computation. That is why a master-worker architecture has been chosen for the implementation. The following sections describe the solution we have implemented and the results obtained.

The proposed method is assuming the URDF-PN model has a finite reachability graph. In the case of URDF-PN models, as in any other high level Petri net formalism, proving the finiteness of the reachability graph of a given model is not an easy task. There are two elements whose combination could influence that question. On the one hand, the structure of the net, which could lead to places with an infinite number of (either identical or different) tokens. On the other hand, the post-condition mapping associated to transitions, which, together with the domains of the variables involved in the input and output patterns of transitions, could allow the generation of an infinite sequence of different tokens.

There are some usual sufficient conditions preventing the previous circumstances to occur. For instance, the use of finite domains for variables, as well as the use of mapping having finite images for the case of infinite variable domains, allows to avoid having an infinite number of different tokens [12, 30]. With respect to the possibility of having an infinite number of repeated tokens in the net, usually the study of the supporting ordinary Petri net model (the Petri net model resulting when arc and transition inscriptions are removed [9]) can give sufficient conditions to avoid such situation. This is the case, for instance, of acyclic or bounded Petri nets (Petri nets whose places are covered for p-invariants, which establish upper bounds to the number of tokens in places, for instance).
3. TOWARDS A PARALLEL IMPLEMENTATION

As explained in previous sections, reachability graph computation is a highly expensive process in both memory and time. We point at developing a distributed program for the computation of the reachability graph in a parallel way, so it could be deployed in either cluster, grid or cloud infrastructures. This would open the possibility of dealing with complex systems. The development of the solution was a long process, which will be explained in greater detail in the following sections. However, for the sake of clarity, a brief overview of the final architecture is provided below.

Our objective became to implement a parallel version of the COMBAS [10] reachability graph generator and to deploy it in the HERMES cluster. HERMES is a cluster hosted by the Aragón Institute of Engineering Research (I3A) ∗. In general terms, HERMES consists of 1308 cores and 2.56 TB of RAM. At the moment of this writing, all the general purpose processors in HERMES use a 64-bits Intel hardware architecture and a Scientific Linux, version 5.5, as the operating system. Cluster nodes are externally connected using 10-Gigabit routers and internally connected using a Gigabit link, allowing high-speed data transfers. The cluster is managed by means of the HTCondor middleware †, version 7.8.4.

Let us now describe the main concepts behind marking storage, signatures and comparison processes. Markings (states) of a U-RDF-PN model are RDF graphs, composed of RDF triples. Two markings are considered as equal if they are equivalent in terms of RDF graphs: if each one entails each other.

Entailment [31] uses inference for a given ontology, so that two non equal graphs can be equivalent. For instance, given a graph:

\[ A = \{ :a :isA :Book, :b :isA :Book \} \]

and a graph


they are clearly different. However, if we consider one ontology in which

\[ :Book :isA :Publication \]

inference application to A and B will give the same equivalent graph:


We have used this property to generate a canonical form for each marking as follows: duplicated triples are removed, being the remaining ones then sorted in alphabetical order; then, all the sorted triples are concatenated in a single string, on which a hash function is applied, obtaining a long integer as the hash of the marking. Therefore, the signature of each state is a data structure that contains the following fields: the (unique) name of the state, the hash of the semantic representation of the state, the additional parametric data of the state in SMT-lib format [25] and, finally, a list of nodes containing a copy of such state. This nodes list is stored in the form of \( \text{key} - \text{value} \) pairs, where the key represents the index associated to each worker node, and the value is the name used to store the state in such node.

Doing so, the signature of a marking with several hundreds of KBytes is compressed to a few bytes, as can be seen in the example below:

```json
{ name: "state-125", semantic-hash: 1652659789226454879, parametric-data: ["(= $a$ true)" , "(= $b$ 9)" , "(= $c$ 3)"], nodes: ["1"=>"state-125", "3"=>"state-211", "5"=>"state-789"]}
```

∗ Aragón Institute of Engineering Research (I3A)
† http://research.cs.wisc.edu/htcondor/
Comparing the hash of two markings becomes comparing two long integers. However, knowing whether two markings are equivalent is a more complex process that is carried as a specific job (and will be detailed later on Section 4). Selection of the hash function is non trivial, since collisions generate an important penalty in efficiency. In our case, since the semantic representation of a state is canonicalized into a single string, the hasCode method of the standard String Java Class is used. Given a string $s$, the hash is computed using the following formula: $hashCode(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$, where $n$ is the string length.

Figure 2 depicts the high-level view of the system architecture, based on a master-worker approach. The configuration is composed of a single master and a set of workers, whose number is specified at deployment time by the user.

On the top-level, the user accesses the system through a Web service-based system interface. The user must provide the system with the net to be processed as well as the number of nodes to be used for the computation (for master and worker processes). The system interface directly connects to a deployment component, which coordinates with a scheduler component in order to set-up in the used computing infrastructure the required master-worker configuration.

The framework has been designed to exploit the computational power of clusters, grids and clouds, trying to reduce the network communication as much as possible. The process is divided in small jobs which are performed by the workers. Each worker iteratively execute two types of jobs:

- **State job**: For a given reachable state, the worker must compute all of its successors. Then, for each of them it computes the signature, which is then transferred to the master if necessary. A more detailed description will be given on next section.
- **Comparison job**: whenever two states signatures collide, the master detects this and generates a comparison job between both states. Therefore, the worker assigned with this task must compare their full representation and determine if they correspond to equivalent states or not.

The master node is the one in charge of distributing jobs and controlling the start-up and termination of the process. The workers request jobs to the master, perform the task and return the results back.
Let us establish the relationship between nodes, states and signatures. Not only computation tasks will be distributed among workers, but also the set of states: each worker will store (some of) the markings it computes. The master is the only process that has a global view of the set or computed reachable markings: when a worker obtains a possible new marking, it computes a signature for it, which is sent to the master. The master will then decide what to do with such marking, telling the worker to store it if it is new (no other marking exists with the same signature) or telling him to do some additional work. The objective of doing so has two main objectives. On the one hand, to distribute markings among different resources, making possible dealing with bigger systems, as markings are quite big structures whose size can vary between a few Kbytes to hundreds of them (depending on the nature of the net as well as the used annotations). On the other hand, to improve the system performances, reducing as much as possible moving complete markings among different resources.

Finally, Figure 3 provides a detailed overview of both master and workers first shown in Figure 2. On the one hand, a master is composed of three main storage units: a hash table containing the states signatures, and two queues that store the jobs with the status which are marked as pending-to-be-processed and pending-to-be-compared (to-process and to-compare queues, respectively). These three storage units are used as inputs for the states signature comparer component, whose process will be detailed later on this section. On the other hand, the job dispatcher coordinates with the workers in order to provide them with jobs that are queued on the to-process or on the to-compare queues.

Both master and workers expose their functionalities by means of REST [32] interfaces. Therefore, both master and workers (even among themselves) directly interact using these interfaces. Trying to improve performance, when possible these REST interactions are complemented with direct socket connections, thus avoiding the necessity of managing the processing REST messages. This way, the proposed solution can be deployed on network-independent nodes as well as on local/private networks (improving the latency and response times in this last case).

On the other hand, workers compose of four different storage units: a local queue that contains the jobs to be processed by that node as well as a local queue to store the states pending to be compared (to-process and to-compare queues, respectively); a local storage for states that have been processed and, finally, a local instance of a Virtuosso Triple Store (including its semantic reasoner).

From the worker’s perspective, the job retriever component first requests jobs to the master, and puts these jobs on the to-process local queue. A similar process is achieved by the comparison job retriever component. Then, the state successor generator component processes them using the Virtuosso Triple Store and saves the resulting states on the local storage unit. The state successor generator also interacts externally with the master in order to generate the next state to compute. The stored states are then compared by the states comparator component, that uses the to-compare queue to complete the process.

Interaction among workers is carried out by the communication between the state successor generator and the states comparator components of the nodes. Nodes organize themselves in order to conduct the computation/comparison process.

With respect to checkpointing, both master and workers contain a local checkpointer component that will perform the checkpointing process as described in Section 5.2.

Finally, we must remark that each component that contains the MapDB‡ label shares the characteristics of this data structures implementation library, this is, an HD file-based persistence and a memory caching mechanism.

‡MapDB http://www.mapdb.org/
4. SYSTEM DESCRIPTION

Let us now describe how the reachability graph is obtained. The process is composed of a set of stages, depicted in Figure 4.

The **Deployment and Start-up stage** consists of the deployment and start-up of the execution over the chosen infrastructure. Figure 5 depicts the steps performed during the deployment phase. First, the user provides the system with the net willing to be processed as well as the number of nodes in which the computation will be deployed (step 1). We are assuming that resources are homogeneous, which allows us to consider all workers are equivalent, avoiding this way the necessity of managing load balancing and other related aspects.

The user specifications are sent through the Systems Interface component. This information is then passed to the Deployment component (the **deployer**, step 2), which connects to the Scheduler in order to get a deployment plan. A deployment plan consists of the specification of the nodes that will be required, as well as the category of each node (master/worker) and endpoint information (IP, port, service, etc.) to connect with these nodes (step 3). These nodes are then deployed on the computing infrastructure (step 4).
Figure 4. Stages that are followed during the computation of a reliability graph.

After that, both master and worker nodes send their endpoint information to the Deployment component (step 5), which is in charge of broadcasting the list of nodes and endpoints to all nodes (step 6). Then, the computing process starts. After that, the deployer terminates (step 7).

The general RG computing process is composed of three different processes: the states computation process, the states comparison process and, finally, the checkpointing process.

Figure 6 shows the states computation process interactions among master and workers. The states computation process is started by the master node that creates job specifications (to be done by workers) in order to compute the different states of the RG, starting from the initial marking (step 1).
A worker has two main processes. The one that is taking job specifications from the master and putting them into a local queue to process; and a second one, which is processing the jobs in the queue (step 2). This last process retrieves a job from the local queue and looks for the corresponding state in the local repository. In case the state is not found, it asks for it to the proper worker (step 3). The list of suitable workers is specified in the signature.

The worker looks for the possible transitions the state enables and computes the successors the firing of such transitions would lead to. For each one of these successors, the signature is computed, being the resulting signatures sent to the master node (step 4). The master will then check if these signatures exist in the states signature table (step 5). If a signature does not exist in that table, it is inserted into it. Otherwise, a comparison job specification is generated and sent to the worker.

Code listings 1 and 2 sketch the pseudocode of the most representative parts of the computation process in both worker and master nodes, respectively.

The states comparison process starts when the worker receives a comparison job. A comparison job is a different type of job, which uses two signature structures, one for each state to be compared, lets say $S_a$ and $S_b$. Any worker can make use of such signatures to retrieve the complete graph describing each state in order to be compared. Figure 7 depicts the main interaction at this stage.

Once the worker node has received the comparison job (step 1), it checks whether $S_a$ and $S_b$ are in its local storage (step 2). The local state storage of each worker is implemented as a MapDB HashMap, making possible a fast access to states (as they are cached in memory), maintaining persistence (as the HashMap is stored on disk) and allowing for fast check-pointing, as the whole HashMap is stored in the disk as a single file (instead of dealing with thousands of very small files).

The comparison algorithm performs the following actions. It first compares the semantic hash of $S_a$ and $S_b$. If they are different, the algorithm returns false. Otherwise, it compares the canonical representation of $S_a$ and $S_b$. In case their representations are different, each state is looked for on the local repository. If it is not found there, the state is requested to a worker from the list of suitable workers that was specified on the signature. This way the canonical representation is obtained as a string; the method returns false (step 3).

In case the canonical representations is equal, the algorithm returns the result of the comparison of the parametric equivalence between $S_a$ and $S_b$. The comparison is performed using an instance of the SMTsolver (CVC) [33], initializing the description of the variables, introducing the statements of both $S_a$ and $S_b$ and performing an equivalence check [10].

Code listings 3 and 4 sketch the pseudocode of the most representative parts of the computation process in both master and worker nodes, respectively.
As a result of the comparison process in a worker node, a boolean value is obtained. This value, along with the signature of the state B, is sent back to the master (step 4), so this state ($S_b$) can be added to the general table if the state was different of $S_a$, or discarded on the contrary (step 5).

When the reachability graph computation process finishes, the resulting graph is stored in RDF format in the database, and it can be obtained through the SPARQL interface of the RDF store [9].

```java
while (j = request_computation_job()) {
    s = get_state_local(j.get_signature());
    if (s == null) {
        s = get_state_remote(j.get_signature());
    }
    sc_list = compute_successors(s);
    for (sc : sc_list) {
        sg = compute_signature(sc);
        send_signature_to_master(sg);
    }
}
```

Listing 1: Computation process (Worker)

```java
while (sga = wait_for_signature()) {
    sgb_list = states_table.search_by_semantic_hash(sga.get_semantic_hash());
    if (sgb_list == null) {
        queue_computation_jobs.add(sga);
    }
    else {
        queue_computation_jobs.add(sgb, sga);
    }
}
```

Listing 2: Computation jobs process (Master)

```java
while (cr = wait_for_comparison_result()) {
    sga = cr.get_signature_a();
    sgb = cr.get_signature_b();
    equivalent_states = cr.get_comparison_result();
    if (equivalent_states == false) {
        states_table.insert_signature(sgb);
        queue_computation_jobs.add(sgb);
    }
}
```

Listing 3: Comparison jobs process (Master)

```java
while (j = request_comparison_job()) {
    sga = j.get_signature_a();
    sgb = j.get_signature_b();
    sa = get_state_local(sga);
    if (sa == null) {
        sa = get_state_remote(sga);
    }
    sb = get_state_local(sgb);
    if (sb == null) {
        sb = get_state_remote(sgb);
    }
    equivalent_states = false;
    if (sga.get_semantic_hash() == sgb.get_semantic_hash()) {
        if (sa.get_canonical() == sb.get_canonical()) {
            equivalent_states = compare_parametric_equivalence(sa, sb);
        }
    }
    send_comparison_result_to_master(sga, sgb, equivalent_states);
}
```

Listing 4: Comparison states process (Worker)

5. EXPERIMENTATION AND RESULTS

Let us now report the experimental results obtained for three different cases: the first two ones correspond to the same implementation on the HERMES cluster: with and without applying error recovery strategies. The third one corresponds to a deployment in a cloud environment (the Google Cloud Platform [34]). For the experiments, the reachability graph of two different configurations of the First Provenance Challenge (FPC) scientific workflow [35], enhanced with semantic information was computed.

The First Provenance Challenge§ is an experiment from the area of Functional Magnetic Resonance Imaging (fMRI). Its aim is to create population-based brain atlases from the fMRI Data

§http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge
PARALLEL COMPUTATION OF THE REACHABILITY GRAPH OF A SEMANTIC PETRI NET

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Execution time (mins)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>859.58</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>75.03</td>
<td>11.46</td>
</tr>
<tr>
<td>20</td>
<td>37.64</td>
<td>22.84</td>
</tr>
<tr>
<td>50</td>
<td>15.24</td>
<td>56.40</td>
</tr>
<tr>
<td>100</td>
<td>7.64</td>
<td>112.44</td>
</tr>
<tr>
<td>150</td>
<td>5.47</td>
<td>157.20</td>
</tr>
<tr>
<td>200</td>
<td>4.30</td>
<td>199.68</td>
</tr>
</tbody>
</table>

Table I. Execution times and speedup of the RG generation for FPC-2pckts experiment on HERMES

Center’s archive (currently, this archive is hosted in a cloud-based environment in the neuroimaging data repository at NITRC\(^5\)) of high resolution anatomical data.

Figure 8 depicts the workflow of this example. As shown, the workflow specifies the set of processing steps to be carried out over the input in order to obtain the final result. In each stage there is a specific procedure which process the input data and generates the output, which is then used as an input for the next stage. The input of the workflow are a set of four brain images (Anatomy Image 1 to 4) and a single reference brain image (Reference Image), which is used with each previous brain image as input for the different stages of the workflow. For each image, besides the pixel image itself there is the metadata information (Anatomy header 1 to 4 and Reference Header). As a result, images are processed and the atlas of the images on each coordinate X, Y and Z are obtained.

Let us now briefly depict the different stages of the workflow. The first stage requires as input parameters a brain image, the reference image and also the metadata information attached to the brain image. At this stage, the align\_warp process compares the reference image to determine how the new image should be warped, this is, the position and shape of the image adjusted, to match the reference brain. As an output, this procedure generates the optimal warp parameter set which defines the spatial transformation to be performed in the next stage (Warp Params 1 to 4). At second stage, the transformation of the image is performed by the reslice process using each warp parameter set. This creates a new version of the original brain image with the configuration defined in the warp parameter set. The output is a resliced image. These first two stages can be executed in parallel. Once all the execution flows (one per each input image) have finished, all the re-sliced images are averaged into one single image using the softmean procedure at the third stage. As a result, an atlas image and its attached metadata information are generated.

Then, the averaged image is sliced for each dimension (x, y and z) to give a 2D atlas along a plane in that dimension, taken through the center of the 3D image. The output is an atlas data set, which is then converted into a graphical atlas image using the ImageMagick utility convert (fourth and fifth stages, respectively).

The First Provenance Challenge is a good example which reflects the traditional elements in scientific computing scenarios, and which will properly introduce our approach.

5.1. First experiment: HERMES cluster

The first experiment was executed in the HERMES cluster, choosing the FPC for the case of two packages of images (FPC-2pckts experiment). The reachability graph contains \(108^2 = 11664\) states. In average, each state required 8 Kbytes of disk space, while its signature required 1.3 Kbytes. Table I and Figure 9 show the execution results when using between 1 and 200 computation nodes. As it can be seen, the method is really scalable, obtaining very good speed-up results.

The speedup value gives an insight of the performance gain obtained with the parallel version of the algorithm, as the number of computing nodes is increased. It is interesting to remark that, from 10 to 150 nodes, the speedup is higher than the number of nodes. This special phenomenon is called

\(^5\)http://www.nitrc.org
super linear speedup, and despite seeming illogical obtaining a reduction of the execution time in a factor greater than the number of processors involved, its explanation can be found in the cache effect. Not only the computational power of processors is joined, but also the caches and storage space present in the memory hierarchies of the used processors. Thus, joining all the RAM from the involved nodes, less disk access is needed for the same state space.

However, despite the promising speedup obtained, different problems arose when addressing larger problems:
• Memory usage: during the computation, large data structures are used. The size of such structures varies with the size of the problem and the number of states generated. For small experiments, this does not represent a limitation. However, when computing hundreds of thousands or millions of states, RAM availability can become a real problem.

• Random failures of the underlying infrastructure: nodes shutdown, Condor failures, RAM starvation due to coexistent processes, etc. These problems, out of control for the application, can provoke failures of some of the instances of the distributed computation. Due to the high dependency between nodes of the considered distribution strategy, the failure of a single worker node will surely trigger a failure effect domino of the executing processes.

• Network failures: whether caused by the network infrastructure, congestion or destination hosts unavailability (due to node shutdowns), these failures can have a similar effect to the ones in the previous item, also generating a generalized shutdown.

Due to the problems mentioned above, the execution of larger experiments would not be successfully achievable. To overcome such limitations, a second version was developed, including some fault tolerance techniques and failure recovering strategies.

5.2. Addition of checkpointing techniques

Failures looked to happen in a random manner, making it impossible to predict which nodes, when, or how many of them would fail. Such uncertainty meant we had to find a trade-off between the time and resources spent on failure counter-measures and the ones wasted when failures appear. Due to the way Condor works, when a job fails because of a node shutdown, the job is restarted from the initial state, which means that the computations done until the failure moment will be lost. Since the considered problem is really demanding from the computational point of view, we envisaged to apply some recovery strategy so that those results obtained until the failure instant could be recovered, avoiding the necessity of recomputing them. Checkpointing represents a good way to achieve this goal. Checkpointing is a concept that can be implemented in very different manners:

• Condor-based checkpointing: Condor provides a checkpointing mechanism [36] for jobs. The mechanism is restricted to the Standard universe, which only allows for the execution of mono process jobs. Given that the proposed architecture requires of multithreading, this approach did not solve our problem.

• Java Virtual Machine checkpointing: Several tools exist [37] which perform checkpointing of a Java Virtual Machine instance in a configurable way. This could look as a quite straightforward solution to overcome the problem, without requiring any code modification. It also allows synchronizing checkpoints of several instances. However, this kind of checkpointing pauses the execution of the virtual machine, creating a dump of memory, which
would interfere with ongoing communications, file writes and other operations. All that would lead to an inconsistent state even if all the checkpoints are performed at the same time in a synchronized fashion.

- Local asynchronous checkpointing: If every node performs checkpoints asynchronously, they could store different states at different time instants. Given the fact that the computations on each node are dependent on the states stored in other nodes, a roll-back operation could lead to an inconsistent state. Such inconsistency can cause a failure when a node asks for a state which was generated in a different node, but which is not present any longer because it was computed after the last checkpoint in such node. This excludes asynchronous checkpointing as a valid solution.

As a consequence, the solution adopted was to design a checkpointing system adapted to the considered distributed architecture. It was implemented at the application level and uses synchronization techniques to ensure consistency between individual node checkpoints. The presence of a shared file system in HERMES made this task easier, so the checkpoints of each node could be stored in a persistent and common location, making files available for resuming processes after a failure. A checkpointing coordinator process, executed by the master node, will monitor heartbeats from each node; it will also coordinate the checkpoint procedure using a two-phase commit protocol (2PC).

The checkpointing method has to periodically stop the computation of new reachable states in the workers, so that they can save their current state (basically, the current set of reachable states each one has). Given that this process pauses the computation flow, the checkpointing period will certainly affect the overall computation time. Its choice represents a trade-off between information loss in case of failure, and computation time penalty generated by the checkpointing. In the context of an environment very prone to failure, a shorter period could benefit the execution time. However, in a stable environment, a longer period would avoid spending too much time on unnecessary checkpoints. Based on the experiments we conducted, we set an hour as an adequate period.

The checkpointing process involves both workers and master nodes. The implemented process is as follows. As a first step, the master sends a checkpointing signal to all workers. When a worker receives the signal it stops retrieving new jobs from the master. Its aim is now to consume local jobs (those jobs stored in its local queue). When the queue has been cleaned up the worker notifies the master that it is ready for checkpointing, and waits for the master’s answer. Once the master has received the ready messages from every worker, it sends the go for checkpoint message. A worker, upon reception of that signal, saves and compresses the set of states he is storing, as well as an execution log (with verbose debug information for diagnostic purposes), and also an MD5 hash of the resulting file is generated. On his side, the master also saves and compresses his state (general job queue, comparison queue, generated states counter and execution log file), also generating the corresponding MD5 hash. Once all the workers signal completion of the saving phase, the master verifies the checkpoints and resumes the computation. The resulting tar.gz files are stored in a common shared file system (via nfs, ftp ... services) in a specific directory, so it could be retrieved in case a rollback would be necessary.

In the first version of the technique, workers stored states as single text files in a common directory. It soon became clear that it was a bad idea, as the archiving of a directory containing thousands of small text files took tens of minutes. The situation got worse as the number of states increased. To deal with that drawback, the saving phase was redesigned so as to use one single database file per node, which is known to provide much better performance [38]. This allowed to skip the archiving step and directly proceed with the compression of a larger file, which is much more efficient.

This version was tested on the HERMES cluster with the FPC workflow for the case of three packages of images (FPC-3pckts), which has \(10^8^3 = 1259712\) reachable states. The results of the experiment are shown in Table II and Figure 10.

As it can be seen, the impact of the checkpointing procedure in the whole process is rather low, compared to the saved time in case of failure given that the absence of checkpoints would require restarting from the beginning, losing all the achieved results until that point. Looking at Figure 11 we
Table II. Execution times (mins) of the RG generation for FPC 3pckts experiment on HERMES with checkpoint

<table>
<thead>
<tr>
<th>Nodes</th>
<th>100</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>RG Computation time</td>
<td>2106.58</td>
<td>1249.15</td>
</tr>
<tr>
<td>Total checkpointing time</td>
<td>199.58</td>
<td>196.99</td>
</tr>
<tr>
<td><strong>Total time</strong></td>
<td><strong>2306.16</strong></td>
<td><strong>1446.14</strong></td>
</tr>
<tr>
<td>Number of failures</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Time wasted by failures</td>
<td>180</td>
<td>60</td>
</tr>
<tr>
<td><strong>Total time with failures</strong></td>
<td><strong>2486.16</strong></td>
<td><strong>1506.14</strong></td>
</tr>
<tr>
<td>Number of checkpoints</td>
<td>32</td>
<td>19</td>
</tr>
<tr>
<td>Average checkpointing time</td>
<td>6.24</td>
<td>10.37</td>
</tr>
<tr>
<td>Checkpointing impact on total time (%)</td>
<td>8.65</td>
<td>13.62</td>
</tr>
<tr>
<td>Checkpointing impact on total time with failures (%)</td>
<td>8.03</td>
<td>13.08</td>
</tr>
</tbody>
</table>

Figure 10. Comparison of execution times per number of nodes for RG generation of PFC-3pckts on HERMES with checkpoinings

see that with 100 workers failures occur after 300, 1500 and 1750 mins approximately, which would have caused the loss of 1750 mins in absolute time and the restart of the procedure. The technique acquires much more interest when the experiment takes a time longer than the maximum period observed without failures (which will surely require restarting computations as the results of some failures). Figure 11 shows the evolution, with respect to time, of the number of computed states, when 100 and 200 worker nodes are used. Short steps in the chart correspond to checkpointing processes (time consumption without node computation), and long horizontal pauses correspond to failures (time to detect failure and restart computation from the last checkpoint).

5.3. Cloud-based deployment

The general structure of the proposed solution had as one of its aims to allow any organization being able to analyse their models despite not having a cluster framework at their disposal (which will be the most usual case). Currently, cloud-based infrastructures are a very adequate mean for getting on-demand computing and storage resources with a high scalability. This was one of the reasons for deploying the experiments on the cloud. A second reason was going further in the study of the system scalability with respect to the number of processes involved, going beyond the maximum number of nodes that we could use in HERMES (200 nodes). We chose the Google Cloud Platform infrastructure [34], but the experiment could have been deployed in any public or private cloud.
Two different executions of the PFC-3pckts experiment were carried out, with 400 and 800 workers respectively. To do so, 51 (machine) instances were requested, one for the master and fifty for the workers. The master node was executed in a "n1-standard-4" machine with 4 cores and 15 GB of RAM. The workers were executed in "n1-highcpu-16" machines, hosting each machine 8 workers for the first experiment (400 workers) and 16 workers for the second one (800 workers). Every machine used a disk of 40 GB with Debian 7.6, Linux kernel 3.2.0-4-amd64 and Oracle Java SE JRE 1.7.0_67, sharing all the instances the same local network. The "gcloud compute" command-line tool, part of the Google Cloud SDK [39], was used to automate the deployment of the whole set of machines. The deployment phase (getting the required resources and starting computing processes) took 25 minutes.

The results obtained from the experiments are shown in Table III. As it can be seen, the number of computed states per time unit scales in a reasonable way. Using 400 worker nodes the execution time was 315 minutes. On the other hand, using twice the number of worker nodes (800 nodes), the time was 221 minutes, which represents a reduction of about a 30% on the execution time. This is so because in the last case there was an under-use of the set of processors. No failures were registered, as the nodes were fully dedicated to the experiment, and no external events or users activities could affect the computation process. Therefore, the checkpointing process could be avoided in order to obtain some better results. However, as it can be easily noticed, the impact of checkpointings was not really significant on the overall time (about 4% in both cases).
Table IV. Monetary cost (in $USD) of the RG generation for FPC-3pkts experiment on Google Compute Engine with checkpointing.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>400</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total time (mins)</td>
<td>314.72</td>
<td>220.14</td>
</tr>
<tr>
<td>n1-standard-4 machines</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Cost of n1-standard-4 machine per hour</td>
<td>$0.177</td>
<td>$0.177</td>
</tr>
<tr>
<td>Total cost of n1-standard-4 machines</td>
<td>$0.93</td>
<td>$0.65</td>
</tr>
<tr>
<td>n1-highcpu-16 machines</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Cost of n1-highcpu-16 machine per hour</td>
<td>$0.448</td>
<td>$0.448</td>
</tr>
<tr>
<td>Total cost of n1-highcpu-16 machines</td>
<td>$117.50</td>
<td>$82.19</td>
</tr>
<tr>
<td>GBs of disk per machine</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>Cost of 1GB of disk per month</td>
<td>$0.04</td>
<td>$0.04</td>
</tr>
<tr>
<td>Total cost of disk</td>
<td>$0.58</td>
<td>$0.40</td>
</tr>
<tr>
<td>Total cost of experiment</td>
<td>$119.01</td>
<td>$83.24</td>
</tr>
</tbody>
</table>

Figure 12 depicts the comparison of execution times per number of nodes for the generation of the reachability graph in the First Provenance Challenge problem on Google Cloud with checkpointing.

The graph presented in Figure 13 provides a more detailed view of the performance in both experiments. Horizontal steps in the chart correspond to checkpointing processes (time consumption without node computation).

It is evident that duplicating the number of workers improves the performance. However, the total execution time is not reduced to 50% of the long run but a 70% approximately. This is probably due to the difference between the “resources per worker” rate. That is, the number of physical cores and amount of RAM memory available for each worker instance. Both experiments (400 and 800 workers) were executed on the same underlying infrastructure, same settings and same amount of resources. That means that each worker in the 400 nodes experiment gets twice as much resources (2 cores and 1.8 GB of RAM) as each worker in the 800 nodes experiment (1 core and 900 MB approximately). This generates a certain overhead given that each of the 50 machines handling the workers had to deal with double number of processes, network connections and disk I/O operations. Also, the Master node had to take care of double number of workers opening connections, requesting new jobs and returning results.

One of the main reasons behind the decision of adopting such configuration is the difficulty to allocate more than 50 machines of such characteristics in a single Zone in Google Cloud. Usually usage quotas apply, which are quite strict by default and its increment requires contacting Google Customer Support directly. By default no more than 24 CPUs per zone can be allocated. However, our plans were to launch 50 machines with 16 cores each, plus another machine with 4 cores. That sums up to 804 cores. After some email exchange with customer support the CPU quota was increased to 900 cores per zone.

Also, the associated monetary cost\(^{1}\) of deploying such big number of high performance machines was a big impediment. Both experiments used 50 instances of n1-highcpu-16 machines, one n1-standard-4 machine and 40GB of disk per instance. The combined cost sums up to $202.25 as can be seen in Table IV.

Despite such inconveniences, the main motivation to perform this experiment was to prove the scalability of this distributed approach to the generation of reachability graphs, and show that cloud systems represent a very adequate infrastructure for the problem we are dealing with in this paper. Therefore, it seemed as a great opportunity to test the approach using a number of resources beyond the limits found in clusters and grids available to us at the moment of performing this study.

There is no reason to directly compare the absolute results obtained with Hermes and those obtained with the cloud-based version, since the experiments have been executed under very

\(^{1}\)Google Compute Engine Pricing https://cloud.google.com/compute/pricing
different conditions. However, we can analyse the different behaviour of both deployments. Hermes is a cluster, shared with tens of other users, with a high usage rate and no strict execution quotas. Therefore, in such environment, it is very likely that our jobs will find themselves affected by other user’s processes. When processes do not behave responsibly in terms of resource usage, the computation nodes would reach such a saturation level that very often they would collapse and crash. This lead to failures and process restarts, which was one of the main reasons to implement the checkpointing strategy described in previous sections.

The collision of our processes with other user’s processes, when competing for the shared resources, explain the irregularities observed in the shape of lines in Figure 11. Also, the long steps (around minute 300, 1500 and 1800) observed in the 100 workers experiment represent the global failures and restarts of the whole computation from the last checkpoint. Smaller steps are due to the checkpoint phases performed every 60 minutes, which required the computation to stop for a few minutes.

The main difference with the Cloud based experiment is the absence of failures in the last one. Also the checkpoints are much faster given that the checkpoint files are not transmitted anywhere, given that each computation instance has its own persistent disk which is not removed even in case of failure.
6. CONCLUSIONS

The construction of the reachability graph in Petri nets is a classical problem that in the case of U-RDF-PN requires working with RDF annotations. This introduces a new problem, as the evaluation of the functions used to compute the reachability graph involve RDF graphs, as well as entailment operators, which are very expensive from a computational point of view. As a result, the whole process becomes a much more complex task from both time and computing resources point of view.

During the literature review, it became clear that the existing solutions on computation of reachability graphs do not support U-RDF-PN nets. In addition to this, despite the existence of some parallel and distributed implementations, none of them was fault tolerant. This was a critical requirement due to the fact that the instability we faced when dealing with most of the execution environments. All these reasons motivated us to develop our own implementation.

In this paper we have shown a parallel implementation that solves the problem. The proposed approach is based on a master-worker architecture that can be deployed on most computing paradigms (from grid to cluster or cloud). The architecture of both master and worker components has been depicted as well as their implementations.

The results should go deeper in different directions. The experiments have been executed in cluster and cloud computing infrastructures. As shown, both cases allow building the reachability graph in a reasonable amount of time. However, the comparison between cluster and cloud has opened a new research line: the use of dynamic and adaptive checkpointing strategies. In the experiments, a checkpointing process was periodically executed. However, the results have shown that the time between checkpoints may/should vary, and the implementation should be able to adapt itself in order to get a good trade-off between robustness and efficiency. For instance, when using clouds with on-demand instances, checkpointing impact could be minimized, as no evictions or hardware failures may be expected. However, when working in highly-used clusters or in cloud systems using pre-emptive instances (as the case of Spot Instances of Amazon EC2 [40], for instance), checkpointing period is a very important parameter. From the deployment point of view, we are looking at the use of other techniques such as map-reduce [41]. The reachability graph building and processing can be effectively managed using this technique, but the introduction of semantics would require to analyse dependencies between nodes and other complex aspects that may limit the use of this approach.

A different direction of improvement seeks at the implemented algorithm. It would be interesting to study whether classical techniques used in Coloured Petri nets for obtaining a more compact graph (such as symmetries [12]) could be adapted. It would also be interesting to remove the constraint of the reachability graph being finite adapting techniques used for the computation of the coverability graph [42, 17, 18].

ACKNOWLEDGEMENTS

This work has been funded by project TIN2014-56633-C3-2-R from the Spanish Government and project JIUZ-2015-TEC-04 from the University of Zaragoza.

REFERENCES


