FlowCert : Probabilistic Certification for Peer-to-Peer Computations

Sébastien Varrette, Jean-Louis Roch
Laboratoire ID-IMAG (UMR 5132)
Projet APACHE (CNRS/INPG/INRIA/UJF),
51, av. Jean Kuntzmann
38330 Montbonnot Saint-Martin, FRANCE
FRANCE
{Sebastien.Varrette,Jean-Louis.Roch}@imag.fr

Franck Leprévost
Université du Luxembourg
Faculté de Droit, Economie et Finance
162 A, Avenue de la Faïencerie
L-1511 Luxembourg, LUXEMBOURG
LUXEMBOURG
Franck.Leprevost@univ.lu

Abstract

Large scale cluster, Peer-to-Peer computing systems and grid computer systems gather thousands of nodes for computing parallel applications. At this scale, it raises the problem of the result checking of the parallel execution of a program on an unsecured grid. This domain is the object of numerous works, either at the hardware or at the software level. We propose here an original software method based on the dynamic computation of the data-flow associated to a partial execution of the program on a secure machine. This data-flow is a summary of the execution : any complete execution of the program on an unsecured remote machine with the same inputs supplies a flow which summary has to correspond to the one obtained by partial execution.

1. Introduction

Large scale distributed platforms, such as grid and Peer-to-Peer computing systems, gather thousands of nodes for computing parallel applications. The use of remote resources in parallel processingis the object of numerous research, particularly in the field of security.

We assume the disposal of a large scale distributed platform where a secure system architecture such as Globus [4, 5] is deployed. Allocation of the resources to the application is performed almost transparently to the user; the user submits its parallel application described as a set of tasks together with their dependencies. To increase security, the system provides strong authentication and secure communications. Yet, even on such a secured environment, the outputs results of the program which is executed on a remote resource (also called *worker* henceforth) may be modified with no control of the client application. In all this paper, a task is said *forged* (or *faked*) when its output results are different than the results it would have delivered if executed on an equivalent resource but under the full control of the client. Task forgery may of course occur when the remote resource is the victim of a Trojan horse that emulates the behavior of a correct system to the outside. However there are more pernicious situations (see §6). Result checking consists then to detect (and eventually correct) tasks forgeries.

A strong certification for the execution of a program on a remote architecture requires that this architecture has a specific secure hardware[8]; various works propose hardware solutions in this frame[13]. Nevertheless, this hardware approach is not suitable for peer-to-peer computing where the hardware is composed of "standard" platforms. As a consequence, several studies propose software solutions that supply intermediate certifications in the case of independent tasks (§2). We propose a probabilistic algorithm for intermediate certification : it ensures that the probability of non-detection of forgery is lesser than an arbitrary threshold ϵ fixed by the user.

More precisely, we consider a peer-to-peer application composed with n tasks (or *jobs*) with dependencies: the inputs of those tasks can be produced by other tasks and their outputs can eventually be consumed by other tasks. Since all workers are anonymous in a peer-to-peer platform, we assume that the result of a given task is forged with a probability $q \in]0, 1[$ and the forgeries between two distinct tasks are assumed independent: this hypothesis is reasonable as it introduces no restriction on the kind of sabotage that may be performed. Also, the distribution of errors is modelled as a Bernoulli distribution $\mathcal{B}(n, q)$. We first show (§3) that the error threshold ϵ can be reached by partial dupli-

cation of only $N_{\epsilon,q} = \frac{\ln \epsilon}{\ln(1-q)}$ randomly chosen tasks on trusted machines (oracles); this quantity is quickly negligible to n. From this result, an original software approach based on the dependencies analysis of the parallel program to execute is proposed. This approach offers two advantages. First, the knowledge of the graph of tasks dependencies - which can be generated before the execution of the tasks that composed it - supplies a partial summary of the program execution. The additional cost of this generation is limited and allows to check the (partial) result conformity of any task without duplication (§4). Secondly, when the forgery of a task t is detected, the knowledge of the graph allows to invalidate the successors tasks of t while pursuing the partial certification of the other tasks. Therefore, we obtain a dynamic algorithm of certification that avoids the complete re-execution of the program (§5). Finally, §6 expounds a distributed software architecture implementing this approach and its advantages against classical attacks, while experimental evaluations are developed in §7.

2. Context of software approaches

Since result checking is performed by software, there is no absolute guarantee that the results are correct. Then, the objective is to minimize the certification cost while ensuring an arbitrary small probability of certification error. Basically, software certification consists of adding informations to the execution to accept/refuse the result(s) of the jobs.

Software approaches for the certification of execution can be divided in two categories:

1. *"simple checkers"* [2] : for some computations, the time required to carry out the computation is asymptotically greater than the time required, given a certain result, to determine whether or not that result is correct. This is possible thanks to a post-condition the results have to verify.

   Whereas this approach seems to be very simple and elegant, it is often impossible to automatically extract such post-condition on any program.

   Furthermore, let assume that the execution of the job J is parallelized on m machines. The detection of the forgery of the final result (that is only checked) does not supply any information on the peer(s) responsible for the forgery. Yet, the knowledge of the dependency graph provides a partial post-condition to any program as it will be described in §4.

2. *duplication* [12, 7] : this approach is based on several executions of each task (or job) on many resources. Among the jobs, some are dedicated to Global Computation - i.e. sequential tasks within the parallel program - while the others (the *oracles*) check these tasks by duplication i.e. re-execution.

Figure 1 describes the general principle of the approach by complete duplication. An important additional cost is generated by this model : the cost of the certification corresponds to the number of oracles queries which is equal to the size of the batch. If the jobs are assumed independent, C. Germain and N. Playez in [7] suggest to limit the additional cost by using a sequential test of Wald [15] at the level of the tester. Yet, this approach is limited as it assumes that the program is composed of independent tasks. We propose now an extension of this approach for the case of tasks with dependencies. The following section expounds how to limit the number of queries to oracles with a probability of certification error bounded by an arbitrary chosen threshold.

3. Certification by partial execution

A peer-to-peer application G_0 composed of n tasks with dependencies is considered. The purpose is to certify some or all outputs of the program.

Our approach is to provide a probabilistic certificate by analogy to the Miller-Rabin Monte-Carlo test of composition (see [9] p. 139). That test considers that a number is prime if the probability of non-detection of composition is small enough. Similarly, we consider that the results to certify are correct if the probability of non-detection of forgery results is small enough. Hence, we will define a *Monte-Carlo test of forgery*. This test is based on duplication of randomly chosen tasks on trusted machines (called *oracles*); communications and computations on oracles are assumed as totally reliable. Thus, if oracles are used in an hypothesis test and if α is the risk of first kind (false alarm) in the oracle answer, then it is assumed that $\alpha = 0$.

The problem is to decide whether or not G_0 contains forged tasks, with a risk of second kind (false negative or non-detection) $\beta \leq \epsilon$, where ϵ is an arbitrary threshold fixed by the user. Let H_0 be the event "G_0 does not contain any forged tasks" and $H_1 = \overline{H}_0$ ("G_0 contains at least a forged task"). Let G be a subset of k uniformly chosen tasks in G_0. These tasks will be submitted to oracles (as described in figure 1). The tester takes one of the following decisions: "ACCEPT" (no tested task was detected forged) or "REJECT" (at least one task was detected faked).

The next proposition states that if the number of tasks is large enough, then a partial duplication of

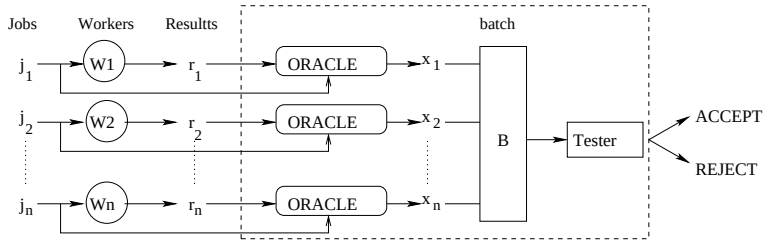


Figure 1. Approach by complete duplication. Let r_i be the result of a job j_i. This result is submitted to a reliable oracle which re-executes the job, compares the result of the duplication with r_i and provides a binary result x_i (0=correct, 1=default). In a Global Computation composed of n independent jobs, the vector $[x_1, ..., x_n]$ is a *batch* submitted to a tester that determines if the batch is accepted or not.

only $N_{\epsilon,q} = \frac{\ln(\epsilon)}{\ln(1-q)}$ tasks, is sufficient to guarantee a given quality of certification (the risk of second kind is bounded by the arbitrary threshold ϵ). Note that $N_{\epsilon,q}$ does not depend on the number n of tasks.

Proposition 1. *Let n be the number of tasks of the program; If the probability of tasks forgery is lesser than q,*

then $\forall \epsilon > 0$, $\exists n_0 / n > n_0$: it is sufficient to check $N_{\epsilon,q} = \frac{\ln(\epsilon)}{\ln(1-q)}$ tasks uniformly chosen to have $\beta = \mathcal{P}(ACCEPT|H_1) \leq \epsilon$.

Proof. If T_i is the number of tasks that have been detected forged in a set $\mathcal{G}$ after i tests, then T_i follows the binomial law $\mathcal{B}(i, q)$. Let k be the number of tasks uniformly chosen among the n tasks of the program for checking. We have :
$\mathcal{P}(H_1) = 1 - \mathcal{P}(H_0) = 1 - \mathcal{P}(T_n = 0) = 1 - (1-q)^n$ and $\mathcal{P}(ACCEPT) = \mathcal{P}(T_k = 0) = (1-q)^k$. Now, if the tester answers "REJECT", then at least one task of G_0 is forged. Hence, $\beta = 1 - \frac{\mathcal{P}(REJECT \cap H_1)}{\mathcal{P}(H_1)} = 1 - \frac{\mathcal{P}(REJECT)}{\mathcal{P}(H_1)}$. Then,

$$\beta \leq \epsilon \iff \frac{(1-q)^k - (1-q)^n}{1-(1-q)^n} \leq \epsilon$$
$$\iff k \geq \frac{\ln[(1-q)^n(1-\epsilon)+\epsilon]}{\ln(1-q)} = f_{\epsilon,q}(n).$$

Now, for $n > 0$, $f_{\epsilon,q}(n)$ is a non-decreasing and positive function, and $f_{\epsilon,q}(n) \xrightarrow{n \to +\infty} N_{\epsilon,q} = \frac{\ln(\epsilon)}{\ln(1-q)}$. Consequently, $\beta \leq \epsilon$ as long as $k \geq N_{\epsilon,q}$. Figure 2 exhibits the evolution of $f_{\epsilon,q}(n)$ when n is increasing. We can see that it quickly tends to the constant value $N_{\epsilon,q}$. □

To illustrate the proposition, the following experiment has been set up. A program composed of $n = 10^6$ tasks is considered. The probability of tasks forgery is raised by $q = 0.01$. The maximum values for the risk of second kind is bounded by $\epsilon = 5\%$. In each experiment, the number of tasks uniformly chosen to check

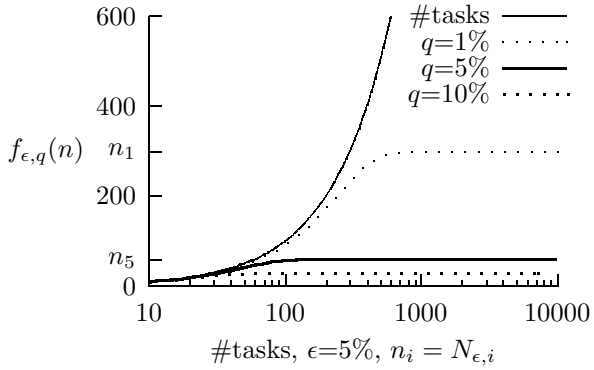


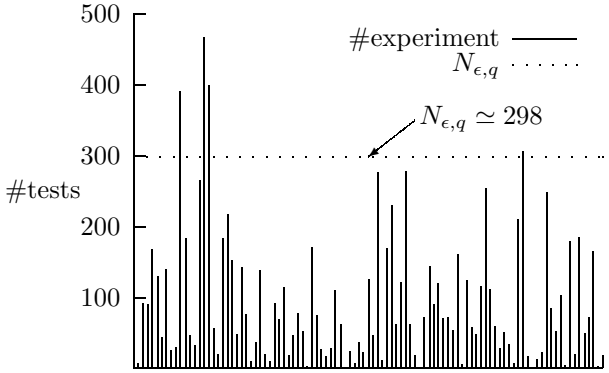
Figure 2. Minimum number of tasks to check w.r.t. the total number of tasks n leading to $\beta \leq \epsilon$.

before the first detection of forgery is computed. Figure 3 illustrates this algorithm repeated 100 times. In practice, as n is large enough (large size programs are considered), $\min\{N_{\epsilon,q}, n\} = N_{\epsilon,q} = o(n)$ tasks will be checked. This value is very small regarding the total number of tasks; in the context of the experiment described in fig. 3, $N_{\epsilon,q} \simeq 298$. Thus, the additional cost required for the certification is quickly negligible. This behavior is illustrated by the experimentations done in §7 (figure 9).

Proposition 1 directly leads to a simple algorithm for error detection: either the test ends after $N_{\epsilon,q}$ successful checks or else an error as been detected.

Figure 3 represents also the value of $N_{\epsilon,q}$ under the hypotheses of the experiment. It can be seen that this algorithm for error detection would have failed 4 times. With this experiment, we have consequently $\beta = 4\%$. This value has to be compared with the threshold ϵ fixed to 5%: the relation $\beta \leq \epsilon$ is verified. This is a general behavior.

The last algorithm enables the detection of forgery by the execution of random chosen tasks. Yet, it re-



Context : #tasks=10^6;q=1%;ϵ=5%

Figure 3. Number of necessary tests before detection of error in 100 successive experiments.The value of $N_{\epsilon,q}$ is also represented in this context.

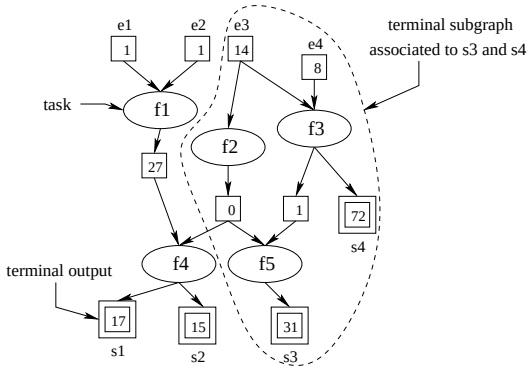


Figure 4. Instance of a data-flow graph associated to the execution of five tasks $\{f_1, ..., f_5\}$. The input parameters of the program are $\{e_1, ..., e_4\}$ whereas the outputs (i.e the results) are $\{s_1, ..., s_4\}$.

quires that we have the possibility to test tasks independently to each other. Consequently, the inputs/outputs of the tasks have to be identified. This is possible thanks to dependencies analysis. This concept is introduced in the following section.

4. Data-Flow analysis

A parallel program will be represented by a bipartite direct acyclic graph G: the first class of vertices is associated to the tasks whereas the second one represents the parameters of the tasks (either inputs or outputs according to the direction of the edge). A *terminal output* indicates a leaf parameter with eventually a result value to certify.

Previous approaches (§2) delimited independent jobs $\{j_i\}_{i\geq 0}$ executed on workers, and certified the computation of j_i by checking the reliability of the worker where j_i is executed; we rather consider a set of terminal outputs $\mathcal{S} = \{s_1, ..., s_m\}$ to certify. $\mathcal{S}$ can contain all or part of the last outputs of the program. Those outputs are independent and their values result from the execution of jobs on one or several worker(s). Let $G_{\mathcal{S}} \subset G$ be the subgraph restricted to the ancestors of the vertices in $\mathcal{S}$. $G_{\mathcal{S}}$ is called the *terminal subgraph associated to* $\mathcal{S}$. Figure 4 illustrates those notions. The computation of the terminal subgraph $G_{\mathcal{S}}$ can be done in linear time $O(|G_{\mathcal{S}}|)$ [3]. $G_{\mathcal{S}}$ supplies then a set of tasks to certify. Modelling an execution by a data-flow graph is part of many parallel programming languages such as Jade [11] or Athapascan [6]. We propose to adapt such language in order to generate the subgraph $G_{\mathcal{S}}$.

4.1. Deterministic tasks re-execution hypothesis

We exhibit two hypothesis H1 and H2 on a macro data-flow parallel program that ensure deterministic results based on any individual tasks re-execution. Those hypothesis are verified by most peer-to-peer applications:

H1 : all synchronizations between tasks are explicitly described in the data-flow graph;

H2 : tasks are deterministic; any execution of a task with same input delivers the same result.

4.2. A Generic Partial Post-Condition

A complete execution of the program supplies then a graph $G_{\mathcal{S}}$ in which all the parameter values are explicit, as in fig.4. This graph is called the **execution track** of the program. Now let's consider the same graph where all the parameters values are symbolic, except the input parameters of the program ($\{e_1, ..., e_4\}$ in fig.4). This partial graph is a summary of the execution track called the **certification track**. It only describes the tasks to execute and their dependencies. Hypothesis assumed in §4.1 ensure that the certification track is deterministic. It has to be generated on reliable resources (oracles) and verify the following properties:

Proposition 2. *The certification track is a summary of the execution track; a partial execution is sufficient to generate it and any correct execution of the program (with the same inputs) on a remote unsecured worker supplies an execution track which summary has to correspond to the certification track.*

By the way, a partial post-condition that can be applied to *any* program is defined. Even if it does not allow to certify the reliability of the computation (more precisely of the set of terminal outputs $\mathcal{S}$), it makes it possible to control the general structure of the executed DAG. Besides, once this post-condition is verified, the execution track can be used to certify the set of terminal outputs $\mathcal{S}$ to detect (and eventually correct) attacks which do not change the structure of the DAG. This is the subject of the following section.

5. Checking algorithm with error correction

In the algorithm defined in §3, tasks have to be checked on secure oracles. Thus, an *elementary oracle* is defined as a task checker operating in a secure environment. Its running is illustrated in figure 5.

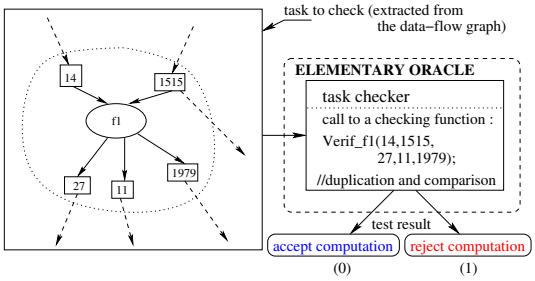


Figure 5. Running of an elementary oracle

The execution track $G_{\mathcal{S}}$ allows to identify the tasks of the program and to access to their execution context (such as the values of inputs/outputs parameters). Thanks to the input parameters of a task $f1$, an elementary oracle can perform a re-execution of $f1$ by calling the associated checking function (`Verif_f1` in fig.5). The results from this re-execution have to match the previous output parameters already extracted from the execution track; otherwise, the task has been forged. In the sequel, $\mathcal{O}_e(t)$ indicates this operation. As in fig.5, $\mathcal{O}_e(t) = 1$ if t has been forged.

In a certification with an arbitrary fixed threshold $\epsilon > 0$, the execution track $G_{\mathcal{S}}$ is submitted to an oracle $\mathcal{O}$ which decides whether the values of the terminal outputs included in $\mathcal{S}$ are correct or not, with respect to the relation $\beta \leq \epsilon$. Yet, if a forged task t is detected, the knowledge of the graph allows to invalidate the successor tasks of t:the related sub-graph has to be replayed and the partial certification of the other tasks can be continued in parallel. Therefore, a *dynamic parallel certification algorithm* is defined and allows to *correct* the forgeries. This algorithm is detailed in Algorithm 1. As the partial post-condition defined in §4.2 ensures the general structure of the program for any execution (the tracks are supposed deterministic), the relation $G_{\mathcal{S}} = G_C \cup G_F$ is satisfied along the recursive calls to the procedure Check.

Let C be the certification cost i.e. the number of operations. If no forgery is detected, $C \leq \min\{N_{\epsilon,q}, |G_{\mathcal{S}}|\}$. Otherwise, in the case of error correction and if a certification is obtained after d detections then the additional cost for the full certification is $\leq (d+1)\min\{N_{\epsilon,q}, |G_{\mathcal{S}}|\}$. Of course, if the tasks of G_F are to be re-executed, the unavoidable cost of this duplication has to be added to C.

Algorithm 1: Dynamic parallel certification algorithm with error correction

Data : $G_{\mathcal{S}}$: execution track to certify **Result :** $\mathcal{O}(G_{\mathcal{S}})$

Check($\emptyset$,$G_{\mathcal{S}}$);

Procedure `Check`

Input: G_F : subgraph of forged tasks and their successors,
G_C : the rest of the graph ($G_C \cap G_F = \emptyset$)

$G = G_C \cup G_F$, $TasksChecked = 0$;
repeat
  Pick up a new task t uniformly chosen;
  if $(t \in G_C)$ *OR* `IsEndOfExecution(`G_F`)`
  then
    if $\mathcal{O}_e(t) == 1$ **then**
      //Detection of a forgery;
      $G_F = G_F \cup$ `Successors(`t`)`;
      $G_C = G \backslash G_F$;
      //G_F must be re-executed;
      `LaunchExecution(`G_F`)`;
      //Checking tasks of G_C can continue;
      //while G_F is being executed;
      `Check(`G_F,G_C`)`;
    else
      $TasksChecked$+=1;
until $TasksChecked == \min\{N_{\epsilon,q}, n(G)\}$;

The memory cost of the certification is $O(|G_{\mathcal{S}}|)$, and hence depends on the granularity of the graph. Moreover, there is a trade-off between the operations number and the memory space: weak granularity implies a large number of tasks. Consequently, the memory cost increases but the certification time (asymptotically bounded by the constant value $N_{\epsilon,q}$) is negligible. The following section illustrates the previous notions in the definition of a distributed software architecture for certification.

6. Distributed architecture for certification

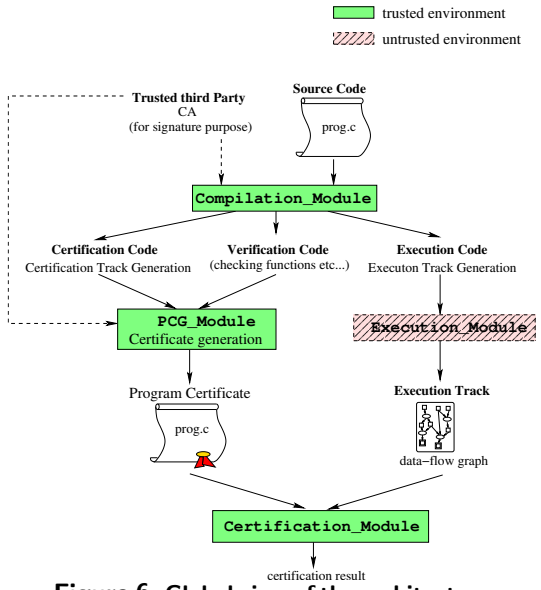


Figure 6. Global view of the architecture

In this section, the source code of a program to certify is considered. The previous notions are integrated in a software architecture to provide the certification of this program. Figure 6 gives an overview of the proposed architecture that is divided in four modules related to the four steps of the certification.

1. The `Compilation_Module` takes in input the initial source code of the program to certify and, thanks to the macro data-flow API, generates three codes that will be used by the other modules:
   - The *Certification Code* is the code which execution provides a partial data-flow graph that is the certification track of the initial program. This graph requires only a partial execution of the source code to be generated.
   - The *Execution Code* provides the execution track of the program, a data-flow graph which modelled the execution of the source code.
   - The *Verification Code* contains all the checking functions that will be used by the elementary oracles.

   Note that the `Compilation_Module` corresponds to an initialization step and has to be executed in a secure environment: the three different codes it provides have to be signed by a trusted third party for further authentication. -
2. The `PCG_Module` for "Program Certificate Generation Module" provides a certificate of the program. This certificate consists in representations of both the certification track and the verification code; it is also signed by a trusted third party for authentication purpose.
3. The `Execution_Module` submits the Execution Code to the clusters grid it is linked to. It provides the execution track (including effective output results of all tasks) that has to be certified.
4. The `Certification_Module` is performed on trusted oracles and decides whether the computation is accepted or not. Its behavior is detailed in figure 7. Firstly, it checks the partial post-condition defined in §4.2: the execution track has to match the pattern described by the certification track. Secondly, it performs the dynamic parallel algorithm with error correction described in §5 to deliver a probabilistic certification of the output results in the execution track. Note that both phases are independent and can be done in parallel: the partial post-condition checks the aspect of the tracks whereas the certification algorithm checks a set of terminal outputs to certify.

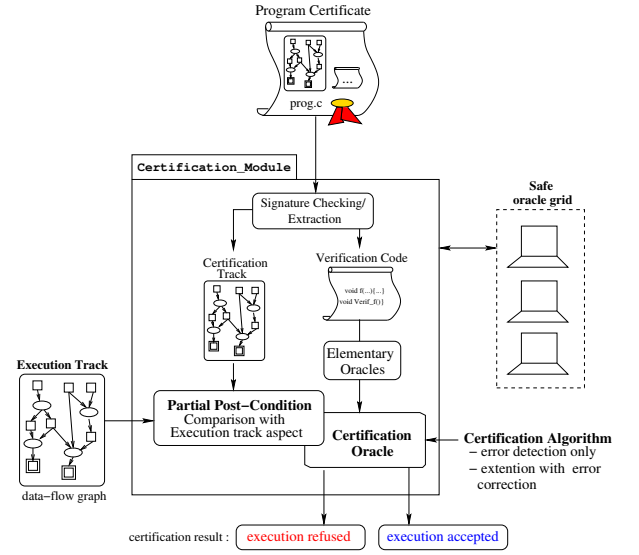


Figure 7. The `Certification_Module` certifies the execution track provided by the Execution Module.

Robustness to attacks and resilience. Historically, the first infrastructure which highlighted the certification issue was the SETI@Home project [1] in 1999. Whereas

the project succeeded beyond the wildest dreams of its creators, people who believed the SETI@Home client software too slow decided to provide a patch to makes the client faster [10]. The previous architecture would have managed to detect the patched clients thanks to the partial post-condition checking.
By using a trusted third party which signs the certificates and the codes generated by the Compilation Module, this architecture provides solutions for authentication and integrity checking. Confidentiality can also be set thanks to SSL protocols for example. By the way, usurpation threats and snooping attacks can be avoided. Yet, DoS attacks are still dangerous on this architecture (like many others), more particularly if the safe grid used for the oracles can be targeted. It introduces the issue of the resilience in the nodes availability. Classic solutions implements periodic challenges (or with an adaptive step). Typically, an authentication challenge with public key as the one used in SSL is considered. Such challenge allows to guarantee not only the presence of a resource but also its authentication. Nevertheless, an alternative could be to assimilate the challenge to a particular computation which is not different than a real computation for the worker point of view. Such tasks will not be duplicated during the certification. Under these assumptions, challenges checking allows to estimate the confidence to place in the distant resource.

7. Illustration and experimentation

Experimental results on a didactic example are exhibited in figures 8 and 9. Figure 8 confirms that a partial execution of the program (around 20% of the time needed to compute an execution track) is sufficient to generate a certification track. In the experiment described in figure 9, no error was introduced in the computation of the program. In this context, the certification time by complete duplication (all the tasks are replayed) introduced in §2 is compared to the certification time by partial duplication expounded in §3 (only $N_{\epsilon,q}$ uniformly chosen tasks are replayed, with $\epsilon = 0,1$ and $q = 0,01$). If the number of tasks is small, the second approach comes down to the first one as all the tasks are checked. But an increase of the number of tasks quickly favours the partial duplication approach in terms of certification time, even if the certification is then probabilistic.

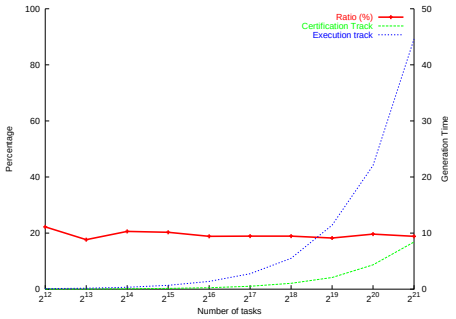


Figure 8. Comparison between times needed to generate both types of tracks relatively to the number of tasks in the tracks with $\epsilon = 5\%$

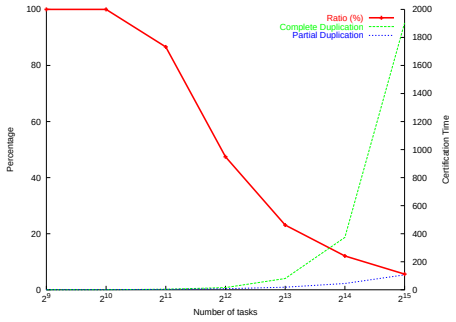


Figure 9. Comparison between certification by complete and partial duplication. In the later case, the parameters are $\epsilon = 0,1$ and $q = 0,01$

8. Conclusion

In this article, a certification scheme for programs with dependencies is proposed. The knowledge of the dependencies thanks to a data-flow graph allows two levels of certification: on the one hand, the structure of the graph given by the *certification track* supplies a partial post-condition that can be applied to any program. In addition, the complete graph of dependencies, also known as the *execution track* allows to define a result certification with a customizable error level. Asymptotically, and if the number of tasks is large enough, we proved that this error level can be reached by the reliable certification of a limited number of uniformly chosen tasks. This number does not depend on the total number of the executed tasks. This method, associated with the post-condition mentioned previously, allows the detection of forgeries with a customizable probability of errors. This result is obtained under the assumption that all the tasks of the graph have the same probability to be forged. A subject of future research consists in considering that the tasks (or even the com-

putation machines - the *workers*) have different fault probabilities.

On the other hand, the execution track allows to correct the detected errors by executing again only a subgraph composed of the tasks detected as forged and their successors. Furthermore, this correction can be done in parallel to the global certification of the program. Finally, a certification architecture based on the previous concepts has been proposed. Experimentations have been done on a didactic example to compare the additional cost required to generate both types of tracks, and to highlight the interest of certification approach by partial duplication with regards to the complete duplication. It is important to notice that our method addresses general parallel programs, not only those written in a data flow language.

This study is part of the framework of the French ACI Grid-DOCG: in this context, we study the development of certification architectures optimized and adapted to scalability on distributed grids for applications on optimization problems. In the context of the RAGTIME project (Région Rhône-Alpes) FlowCert is used to provide result certification for medical applications based on remote computations involving distributed databases. The perspectives concern the experiment on this class of applications on a grid where the nodes are not equivalent. A particular point is the estimation of the forgery rate of a task, which is in fact unknown and related to the execution architecture. This evaluation is important for the estimation of the number of tests required for the partial certification at a customizable error rate proposed in this paper (see [14]).

References

[1] The SETI@Home project, 1999. `http://setiathome.ssl.berkeley.edu/`

[2] M. Blum and H. Wasserman. Software Reliability via Run-Time Result-Checking. *Journal of the ACM*, 44(6):826–849, Novembre 1997.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* McGraw-Hill, 2001.

[4] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *Fifth ACM Conference on Computer and Communications Security Conference*, pages 83–92, San Francisco, California, 3–5 Novembre 1998.

[5] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. Security for Grid Services. In I. Press, editor, *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, Seattle, Washington, 22–24 Juin 2003.

[6] F. Galilée, J.-L. Roch, G. Cavalheiro, and M. Doreille. Athapascan-1: On-line Building Data Flow Graph in a Parallel Language. In IEEE, editor, *International Conference on Parallel Architectures and Compilation Techniques, PACT'98*, pages 88–95, Paris, France, Octobre 1998.

[7] C. Germain and N. Playez. Result checking in global computing systems. In ACM, editor, *Proceedings of the 17th Annual ACM International Conference on Supercomputing (ICS 03)*, San Francisco, California, 23–26 Juin 2003.

[8] R. Keryell. Cryptopage-1 : vers la fin du piratage informatique? In *6e symposium sur les architectures nouvelles de machine (SympA'6)*, Besançon, France, 19–22 Juin 2000.

[9] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Inc, 1997.

[10] D. Molnar. The SETI@Home Problem. November 2000.

[11] M. Rinard. The design, implementation and evaluation of Jade : a portable, implicitly parallel programming language. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, Mai 1998.

[12] L. F. G. Sarmenta. Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. In *ACM/IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*, Brisbane, Australia, Mai 2001.

[13] E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th International Conference on Supercomputing*, San Francisco, California, 23–26 Juin 2003.

[14] S. Varrette and J.-L. Roch. Certification logicielle de calcul global avec dépendances sur grille. In *15èmes rencontres francophones du parallélisme (RenPar'15)*, pages 169–176, La-Colle-Sur-Loup, France, 15–17 Octobre 2003.

[15] A. Wald. *Sequential Analysis.* Wiley Pub. in Math. Stat., 1966.